NeetCode 150 – C++ Pattern Handbook

Purpose: This handbook serves as a concise *answer book* for the NeetCode 150. For each problem, you record the **core pattern**, **when to use it**, **key invariants**, and a **reusable C++ template**. The intent is speed, recall, and correctness under interview conditions.

How to Use This Handbook

For **every question**, follow the same structure: 1. **Problem Name + Link** 2. **Pattern Category** (e.g., Sliding Window, Binary Search) 3. **When to Use / Trigger Conditions** 4. **Key Invariants** (what must always hold true) 5. **Time & Space Complexity** 6. **C++ Pattern Template** (generic, reusable) 7. **Problem-Specific Notes** (edge cases, tweaks)

---

1. Arrays & Hashing (Problem-wise C++ Templates)

> **Rule for this section**: One dominant pattern per problem. Templates are minimal and recall-oriented.

---

1. Contains Duplicate

Pattern: Hash Set — existence check

Trigger: Detect any repeated element

Invariant: `seen` contains only unique elements processed so far

```
unordered_set<int> seen;
for (int x : nums) {
    if (seen.count(x)) return true;
    seen.insert(x);
}
return false;
```

---

2. Valid Anagram

Pattern: Fixed-size frequency array

Trigger: Order irrelevant, frequency must match

Invariant: Net frequency of every character is zero

```
vector<int> freq(26, 0);
for (char c : s) freq[c - 'a']++;
for (char c : t) freq[c - 'a']--;
for (int f : freq) if (f != 0) return false;
return true;
```

---

3. Two Sum

Pattern: Hash Map (value → index)

Trigger: Find pair summing to target

Invariant: Map stores elements strictly before current index

```
unordered_map<int,int> mp;
for (int i = 0; i < nums.size(); i++) {
    int need = target - nums[i];
    if (mp.count(need)) return {mp[need], i};
    mp[nums[i]] = i;
}
```

---

4. Group Anagrams

Pattern: Canonical key hashing (sorted string)

Trigger: Group strings with identical character composition

Invariant: Same sorted key ⇒ same group

```
unordered_map<string, vector<string>> groups;
for (string s : strs) {
    string key = s;
    sort(key.begin(), key.end());
    groups[key].push_back(s);
}
```

---

5. Top K Frequent Elements

Pattern: Frequency map + bucket sort

Trigger: Need top-K by frequency (no ordering requirement)

Invariant: Bucket index represents frequency

```cpp
unordered_map<int,int> freq;
for (int x : nums) freq[x]++;

vector<vector<int>> buckets(nums.size() + 1);
for (auto &[num, f] : freq) buckets[f].push_back(num);

vector<int> ans;
for (int i = buckets.size() - 1; i >= 0 && ans.size() < k; i--) {
    for (int x : buckets[i]) ans.push_back(x);
}
```

---

6. Encode and Decode Strings

Pattern: Length-prefixed encoding

Trigger: Safe serialization without delimiter collision

Invariant: Length prefix defines exact substring boundary

```cpp
// Encode
string enc;
for (string s : strs) enc += to_string(s.size()) + '#' + s;

// Decode
vector<string> res;
for (int i = 0; i < enc.size();) {
    int j = i;
    while (enc[j] != '#') j++;
    int len = stoi(enc.substr(i, j - i));
    res.push_back(enc.substr(j + 1, len));
    i = j + 1 + len;
}
```

---

7. Product of Array Except Self

Pattern: Prefix × suffix products

Trigger: Product excluding index, no division

Invariant: `res[i] = product(left of i) * product(right of i)`

```
int n = nums.size();
vector<int> res(n, 1);
int pre = 1;
for (int i = 0; i < n; i++) {
    res[i] = pre;
    pre *= nums[i];
}
int suf = 1;
for (int i = n - 1; i >= 0; i--) {
    res[i] *= suf;
    suf *= nums[i];
}
```

---

8. Valid Sudoku

Pattern: Hash set for constraints

Trigger: Validate uniqueness across rows, cols, boxes

Invariant: No duplicate constraint key allowed

```
unordered_set<string> seen;
for (int r = 0; r < 9; r++) {
    for (int c = 0; c < 9; c++) {
        if (board[r][c] == '.') continue;
        string v(1, board[r][c]);
        if (!seen.insert(v + "r" + to_string(r)).second ||
            !seen.insert(v + "c" + to_string(c)).second ||
            !seen.insert(v + "b" + to_string(r/3) + to_string(c/3)).second)
            return false;
    }
}
```

---

9. Longest Consecutive Sequence

Pattern: Hash set + sequence start detection

Trigger: Longest run of consecutive integers

Invariant: Start counting only if `(x - 1)` is absent

```
unordered_set<int> s(nums.begin(), nums.end());
int best = 0;
for (int x : s) {
    if (!s.count(x - 1)) {
        int cur = x, len = 1;
        while (s.count(cur + 1)) { cur++; len++; }
        best = max(best, len);
    }
}
```

End of Arrays & Hashing

---

2. Two Pointers (Problem-wise C++ Templates)

> **Rule for this section**: Array or string traversal using two indices with a clear monotonic movement.

---

10. Valid Palindrome

Pattern: Two pointers with skipping

Trigger: Check palindrome ignoring non-alphanumerics

Invariant: `l` and `r` always point to valid characters

```
int l = 0, r = s.size() - 1;
while (l < r) {
    while (l < r && !isalnum(s[l])) l++;
    while (l < r && !isalnum(s[r])) r--;
    if (tolower(s[l]) != tolower(s[r])) return false;
    l++; r--;
}
return true;
```

11. Two Sum II – Input Array Is Sorted

Pattern: Opposite-end two pointers

Trigger: Sorted array, target sum

Invariant: Moving left increases sum, moving right decreases sum

```
int l = 0, r = numbers.size() - 1;
while (l < r) {
    int sum = numbers[l] + numbers[r];
    if (sum == target) return {l + 1, r + 1};
    if (sum < target) l++;
    else r--;
}
```

12. 3Sum

Pattern: Sort + fixed pointer + two pointers

Trigger: Triplets summing to zero

Invariant: Skip duplicates at every pointer

```
sort(nums.begin(), nums.end());
for (int i = 0; i < nums.size(); i++) {
    if (i > 0 && nums[i] == nums[i - 1]) continue;
    int l = i + 1, r = nums.size() - 1;
    while (l < r) {
        int sum = nums[i] + nums[l] + nums[r];
        if (sum == 0) {
            // record triplet
            while (l < r && nums[l] == nums[l + 1]) l++;
            while (l < r && nums[r] == nums[r - 1]) r--;
            l++; r--;
        } else if (sum < 0) l++;
        else r--;
    }
}
```

13. Container With Most Water

Pattern: Two pointers with greedy movement

Trigger: Max area between two vertical lines

Invariant: Move the pointer with smaller height

```
int l = 0, r = height.size() - 1;
int best = 0;
while (l < r) {
    best = max(best, min(height[l], height[r]) * (r - l));
    if (height[l] < height[r]) l++;
    else r--;
}
```

14. Trapping Rain Water

Pattern: Two pointers with left/right max

Trigger: Water trapped depends on min(maxLeft, maxRight)

Invariant: Water added only when current height < boundary

```
int l = 0, r = height.size() - 1;
int leftMax = 0, rightMax = 0, water = 0;
while (l < r) {
    if (height[l] < height[r]) {
        leftMax = max(leftMax, height[l]);
        water += leftMax - height[l];
        l++;
    } else {
        rightMax = max(rightMax, height[r]);
        water += rightMax - height[r];
        r--;
    }
}
```

End of Two Pointers

---

3. Sliding Window (Problem-wise C++ Templates)

> **Rule for this section**: Maintain a dynamic window — expand with `r`, shrink with `l` to satisfy constraints.

---

15. Best Time to Buy and Sell Stock

Pattern: Sliding window (min so far)

Trigger: Maximize difference with sell after buy

Invariant: `minPrice` is the minimum price before current day

```cpp
int minPrice = prices[0], profit = 0;
for (int i = 1; i < prices.size(); i++) {
    profit = max(profit, prices[i] - minPrice);
    minPrice = min(minPrice, prices[i]);
}
```

---

16. Longest Substring Without Repeating Characters

Pattern: Variable window + hash set

Trigger: Unique characters only

Invariant: Window contains no duplicates

```cpp
unordered_set<char> st;
int l = 0, best = 0;
for (int r = 0; r < s.size(); r++) {
    while (st.count(s[r])) {
        st.erase(s[l++]);
    }
    st.insert(s[r]);
    best = max(best, r - l + 1);
}
```

---

17. Longest Repeating Character Replacement

Pattern: Sliding window + max frequency

Trigger: Replace at most `k` chars to make window uniform

Invariant: `(window size - maxFreq) <= k`

```
vector<int> freq(26, 0);
int l = 0, maxFreq = 0, best = 0;
for (int r = 0; r < s.size(); r++) {
    maxFreq = max(maxFreq, ++freq[s[r] - 'A']);
    while ((r - l + 1) - maxFreq > k) {
        freq[s[l++] - 'A']--;
    }
    best = max(best, r - l + 1);
}
```

---

18. Permutation in String

Pattern: Fixed window + frequency matching

Trigger: Check if any permutation exists

Invariant: Window frequency equals target frequency

```
vector<int> need(26, 0), win(26, 0);
for (char c : s1) need[c - 'a']++;
int l = 0;
for (int r = 0; r < s2.size(); r++) {
    win[s2[r] - 'a']++;
    if (r - l + 1 > s1.size()) win[s2[l++] - 'a']--;
    if (win == need) return true;
}
return false;
```

---

19. Minimum Window Substring

Pattern: Variable window + requirement counter

Trigger: Smallest window containing all chars

Invariant: `have == need` means valid window

```
unordered_map<char,int> need, win;
for (char c : t) need[c]++;
int have = 0, req = need.size();
int l = 0;
for (int r = 0; r < s.size(); r++) {
    win[s[r]]++;
    if (need.count(s[r]) && win[s[r]] == need[s[r]]) have++;
    while (have == req) {
        // update answer
        if (need.count(s[l]) && win[s[l]] == need[s[l]]) have--;
        win[s[l++]]--;
    }
}
```

---

20. Sliding Window Maximum

Pattern: Monotonic deque

Trigger: Max in every fixed-size window

Invariant: Deque stores indices in decreasing order

```
deque<int> dq;
for (int i = 0; i < nums.size(); i++) {
    while (!dq.empty() && dq.front() <= i - k) dq.pop_front();
    while (!dq.empty() && nums[dq.back()] <= nums[i]) dq.pop_back();
    dq.push_back(i);
    if (i >= k - 1) ans.push_back(nums[dq.front()]);
}
```

End of Sliding Window

---

4. Stack (Problem-wise C++ Templates)

> **Rule for this section**: Use stack to track previous unresolved elements or maintain monotonic order.

---

21. Valid Parentheses

Pattern: Stack matching

Trigger: Validate balanced brackets

Invariant: Stack top must match current closing bracket

```
stack<char> st;
unordered_map<char,char> mp = {{')','('}, {']','['}, {'}','{'}};
for (char c : s) {
    if (mp.count(c)) {
        if (st.empty() || st.top() != mp[c]) return false;
        st.pop();
    } else st.push(c);
}
return st.empty();
```

---

22. Min Stack

Pattern: Auxiliary stack for minimums

Trigger: Retrieve min in O(1)

Invariant: Min stack mirrors minimum so far

```
stack<int> st, minSt;
void push(int x) {
    st.push(x);
    if (minSt.empty() || x <= minSt.top()) minSt.push(x);
}
void pop() {
    if (st.top() == minSt.top()) minSt.pop();
    st.pop();
}
int getMin() { return minSt.top(); }
```

---

23. Evaluate Reverse Polish Notation

Pattern: Operand stack

Trigger: Postfix expression evaluation

Invariant: Stack holds valid operands only

```
stack<int> st;
for (string t : tokens) {
    if (t == "+" || t == "-" || t == "*" || t == "/") {
        int b = st.top(); st.pop();
        int a = st.top(); st.pop();
        if (t == "+") st.push(a + b);
        if (t == "-") st.push(a - b);
        if (t == "*") st.push(a * b);
        if (t == "/") st.push(a / b);
    } else st.push(stoi(t));
}
return st.top();
```

---

24. Daily Temperatures

Pattern: Monotonic decreasing stack

Trigger: Next greater element to the right

Invariant: Stack indices correspond to decreasing temperatures

```
stack<int> st;
vector<int> res(n, 0);
for (int i = 0; i < n; i++) {
    while (!st.empty() && temps[i] > temps[st.top()]) {
        int idx = st.top(); st.pop();
        res[idx] = i - idx;
    }
    st.push(i);
}
```

---

25. Car Fleet

Pattern: Stack on processed times

Trigger: Merge cars based on arrival time

Invariant: Stack stores decreasing arrival times

```
vector<pair<int,int>> cars;
for (int i = 0; i < n; i++) cars.push_back({pos[i], speed[i]});
sort(cars.begin(), cars.end(), greater<>());
stack<double> st;
for (auto &[p, s] : cars) {
    double time = (double)(target - p) / s;
    if (st.empty() || time > st.top()) st.push(time);
}
return st.size();
```

---

26. Largest Rectangle in Histogram

Pattern: Monotonic increasing stack

Trigger: Max area under histogram

Invariant: Stack stores indices of increasing heights

```
stack<int> st;
for (int i = 0; i <= n; i++) {
    int h = (i == n ? 0 : heights[i]);
    while (!st.empty() && h < heights[st.top()]) {
        int height = heights[st.top()]; st.pop();
        int width = st.empty() ? i : i - st.top() - 1;
        maxArea = max(maxArea, height * width);
    }
    st.push(i);
}
```

End of Stack

---

5. Binary Search (Problem-wise C++ Templates)

> **Rule for this section**: Search on a monotonic space. Maintain invariant that the answer always lies within `[l, r]`.

---

27. Binary Search

Pattern: Classic binary search on index

Trigger: Sorted array, exact target

Invariant: Target (if exists) lies within `[l, r]`

```
int l = 0, r = nums.size() - 1;
while (l <= r) {
    int m = l + (r - l) / 2;
    if (nums[m] == target) return m;
    if (nums[m] < target) l = m + 1;
    else r = m - 1;
}
return -1;
```

---

28. Search a 2D Matrix

Pattern: Binary search on virtual 1D array

Trigger: Matrix sorted row-wise and row-to-row

Invariant: Treat matrix as flattened sorted array

```
int m = matrix.size(), n = matrix[0].size();
int l = 0, r = m * n - 1;
while (l <= r) {
    int mid = l + (r - l) / 2;
    int val = matrix[mid / n][mid % n];
    if (val == target) return true;
    if (val < target) l = mid + 1;
    else r = mid - 1;
}
return false;
```

---

29. Koko Eating Bananas

Pattern: Binary search on answer (rate)

Trigger: Minimize maximum rate under time constraint

Invariant: If rate works, any higher rate also works

```
int l = 1, r = *max_element(piles.begin(), piles.end());
while (l < r) {
    int m = l + (r - l) / 2;
    long hours = 0;
    for (int p : piles) hours += (p + m - 1) / m;
    if (hours <= h) r = m;
    else l = m + 1;
}
return l;
```

---

30. Find Minimum in Rotated Sorted Array

Pattern: Binary search with rotation

Trigger: Sorted array rotated once

Invariant: Minimum lies in unsorted half

```
int l = 0, r = nums.size() - 1;
while (l < r) {
    int m = l + (r - l) / 2;
    if (nums[m] > nums[r]) l = m + 1;
    else r = m;
}
return nums[l];
```

---

31. Search in Rotated Sorted Array

Pattern: Binary search with sorted-half detection

Trigger: Rotated sorted array, unique elements

Invariant: One half is always sorted

```
int l = 0, r = nums.size() - 1;
while (l <= r) {
    int m = l + (r - l) / 2;
    if (nums[m] == target) return m;
    if (nums[l] <= nums[m]) {
        if (nums[l] <= target && target < nums[m]) r = m - 1;
        else l = m + 1;
    } else {
        if (nums[m] < target && target <= nums[r]) l = m + 1;
        else r = m - 1;
    }
}
return -1;
```

---

32. Time Based Key-Value Store

Pattern: Binary search on timestamps per key

Trigger: Retrieve latest value ≤ given timestamp

Invariant: Values stored in increasing timestamp order

```
unordered_map<string, vector<pair<int,string>>> mp;

string get(string key, int t) {
    auto &v = mp[key];
    int l = 0, r = v.size() - 1;
    string res = "";
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (v[m].first <= t) {
            res = v[m].second;
            l = m + 1;
        } else r = m - 1;
    }
    return res;
}
```

---

33. Median of Two Sorted Arrays

Pattern: Binary search on partition

Trigger: Median from two sorted arrays

Invariant: Left partitions contain half elements and maxLeft ≤ minRight

```
if (A.size() > B.size()) swap(A, B);
int m = A.size(), n = B.size();
int l = 0, r = m;
while (l <= r) {
    int i = (l + r) / 2;
    int j = (m + n + 1) / 2 - i;
    int Aleft = (i == 0 ? INT_MIN : A[i - 1]);
    int Aright = (i == m ? INT_MAX : A[i]);
    int Bleft = (j == 0 ? INT_MIN : B[j - 1]);
    int Bright = (j == n ? INT_MAX : B[j]);
    if (Aleft <= Bright && Bleft <= Aright) {
        if ((m + n) % 2 == 0)
            return (max(Aleft, Bleft) + min(Aright, Bright)) / 2.0;
        return max(Aleft, Bleft);
    } else if (Aleft > Bright) r = i - 1;
    else l = i + 1;
}
```

End of Binary Search

6. Linked List (Problem-wise C++ Templates)

> **Rule for this section**: Pointer manipulation with careful handling of `nullptr` and list boundaries.

34. Reverse Linked List

Pattern: Iterative pointer reversal

Trigger: Reverse entire list

Invariant: `prev` points to reversed prefix

```
ListNode* prev = nullptr;
while (head) {
    ListNode* nxt = head->next;
    head->next = prev;
    prev = head;
    head = nxt;
}
return prev;
```

---

35. Merge Two Sorted Lists

Pattern: Dummy head + linear merge

Trigger: Merge two sorted linked lists

Invariant: Tail always points to smallest next node

```
ListNode dummy;
ListNode* tail = &dummy;
while (l1 && l2) {
    if (l1->val < l2->val) { tail->next = l1; l1 = l1->next; }
    else { tail->next = l2; l2 = l2->next; }
    tail = tail->next;
}
tail->next = l1 ? l1 : l2;
return dummy.next;
```

---

36. Linked List Cycle

Pattern: Floyd's Tortoise & Hare

Trigger: Detect cycle in linked list

Invariant: Fast moves 2× slow

```
ListNode* slow = head;
ListNode* fast = head;
while (fast && fast->next) {
    slow = slow->next;
    fast = fast->next->next;
    if (slow == fast) return true;
}
return false;
```

---

37. Reorder List

Pattern: Split + reverse + merge

Trigger: Reorder L0→Ln→L1→Ln-1...

Invariant: Second half reversed before merge

```
// find middle
ListNode* slow = head;
ListNode* fast = head->next;
while (fast && fast->next) {
    slow = slow->next;
    fast = fast->next->next;
}
// reverse second half
ListNode* prev = nullptr;
ListNode* cur = slow->next;
slow->next = nullptr;
while (cur) {
    ListNode* nxt = cur->next;
    cur->next = prev;
    prev = cur;
    cur = nxt;
}
// merge
ListNode* first = head;
ListNode* second = prev;
while (second) {
    ListNode* t1 = first->next;
    ListNode* t2 = second->next;
    first->next = second;
    second->next = t1;
    first = t1;
    second = t2;
}
```

38. Remove Nth Node From End of List

Pattern: Two pointers with gap

Trigger: Remove node N from end

Invariant: Distance between pointers = n

```
ListNode dummy(0, head);
ListNode* fast = &dummy;
ListNode* slow = &dummy;
for (int i = 0; i < n; i++) fast = fast->next;
while (fast->next) {
    fast = fast->next;
    slow = slow->next;
}
slow->next = slow->next->next;
return dummy.next;
```

---

39. Copy List With Random Pointer

Pattern: Hash map old → new

Trigger: Deep copy with random pointers

Invariant: Each original node maps to exactly one clone

```
unordered_map<Node*, Node*> mp;
for (Node* cur = head; cur; cur = cur->next)
    mp[cur] = new Node(cur->val);
for (Node* cur = head; cur; cur = cur->next) {
    mp[cur]->next = mp[cur->next];
    mp[cur]->random = mp[cur->random];
}
return mp[head];
```

---

40. Add Two Numbers

Pattern: Digit-wise addition with carry

Trigger: Numbers stored in reverse order

Invariant: Carry propagated each step

```
ListNode dummy;
ListNode* cur = &dummy;
int carry = 0;
while (l1 || l2 || carry) {
    int sum = carry;
    if (l1) sum += l1->val, l1 = l1->next;
    if (l2) sum += l2->val, l2 = l2->next;
    cur->next = new ListNode(sum % 10);
    carry = sum / 10;
    cur = cur->next;
}
return dummy.next;
```

---

41. Find the Duplicate Number

Pattern: Floyd’s cycle detection (array as list)

Trigger: Numbers in range [1,n]

Invariant: Duplicate forms a cycle

```
int slow = nums[0], fast = nums[0];
do {
    slow = nums[slow];
    fast = nums[nums[fast]];
} while (slow != fast);
slow = nums[0];
while (slow != fast) {
    slow = nums[slow];
    fast = nums[fast];
}
return slow;
```

---

42. LRU Cache

Pattern: Doubly linked list + hash map

Trigger: O(1) get/put with eviction

Invariant: Most recently used at front

```
unordered_map<int, list<pair<int,int>>::iterator> mp;
list<pair<int,int>> dll;

void touch(int key) {
    auto it = mp[key];
    dll.splice(dll.begin(), dll, it);
}
```

---

43. Merge K Sorted Lists

Pattern: Min-heap of list heads

Trigger: Merge multiple sorted lists

Invariant: Heap top is smallest current node

```
priority_queue<ListNode*, vector<ListNode*>, cmp> pq;
for (auto l : lists) if (l) pq.push(l);
ListNode dummy, *tail = &dummy;
while (!pq.empty()) {
    auto node = pq.top(); pq.pop();
    tail->next = node; tail = node;
    if (node->next) pq.push(node->next);
}
return dummy.next;
```

---

44. Reverse Nodes in K Group

Pattern: Group reversal

Trigger: Reverse nodes in fixed-size groups

Invariant: Only complete groups are reversed

```
ListNode dummy(0, head);
ListNode* prev = &dummy;
while (true) {
    ListNode* tail = prev;
    for (int i = 0; i < k && tail; i++) tail = tail->next;
    if (!tail) break;
    ListNode* next = tail->next;
    // reverse [prev->next, tail]
    ListNode* cur = prev->next;
    ListNode* p = next;
    while (cur != next) {
        ListNode* t = cur->next;
        cur->next = p;
        p = cur;
        cur = t;
    }
    ListNode* start = prev->next;
    prev->next = tail;
    prev = start;
}
return dummy.next;
```

End of Linked List

Next section to be appended: **Trees**

Trees

Invert Binary Tree

Pattern: DFS / BFS **Invariant:** Swap left and right subtrees at every node

```
TreeNode* invertTree(TreeNode* root){
    if(!root) return nullptr;
    swap(root->left, root->right);
    invertTree(root->left);
    invertTree(root->right);
    return root;
}
```

Maximum Depth of Binary Tree

Pattern: DFS height **Invariant:** depth = 1 + max(left, right)

```
int maxDepth(TreeNode* root){
    if(!root) return 0;
    return 1 + max(maxDepth(root->left), maxDepth(root->right));
}
```

Diameter of Binary Tree

Pattern: DFS with global answer **Invariant:** diameter passes through node = leftDepth + rightDepth

```
int ans=0;
int dfs(TreeNode* root){
    if(!root) return 0;
    int l=dfs(root->left), r=dfs(root->right);
    ans=max(ans, l+r);
    return 1+max(l,r);
}
```

Balanced Binary Tree

Pattern: DFS height check **Invariant:** subtree height diff ≤1

```
int dfs(TreeNode* root){
    if(!root) return 0;
    int l=dfs(root->left); if(l==-1) return -1;
    int r=dfs(root->right); if(r==-1) return -1;
    if(abs(l-r)>1) return -1;
    return 1+max(l,r);
}
```

Same Tree

Pattern: DFS compare **Invariant:** values and structure identical

```
bool isSameTree(TreeNode* p, TreeNode* q){
    if(!p||!q) return p==q;
    return p->val==q->val && isSameTree(p->left,q->left) && isSameTree(p-
>right,q->right);
}
```

Subtree of Another Tree

Pattern: DFS + SameTree **Invariant:** check match at each node

```
bool isSubtree(TreeNode* s, TreeNode* t){
    if(!s) return false;
    if(isSameTree(s,t)) return true;
    return isSubtree(s->left,t)||isSubtree(s->right,t);
}
```

Lowest Common Ancestor of BST

Pattern: BST property **Invariant:** split where p and q diverge

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q){
    if(p->val<root->val && q->val<root->val) return lowestCommonAncestor(root->left,p,q);
    if(p->val>root->val && q->val>root->val) return lowestCommonAncestor(root->right,p,q);
    return root;
}
```

Binary Tree Level Order Traversal

Pattern: BFS **Invariant:** process level by level

```
vector<vector<int>> levelOrder(TreeNode* root){
    vector<vector<int>> res;
    if(!root) return res;
    queue<TreeNode*> q; q.push(root);
    while(!q.empty()){
        int n=q.size(); vector<int> lvl;
        while(n--){ auto cur=q.front(); q.pop(); lvl.push_back(cur->val);
            if(cur->left) q.push(cur->left);
            if(cur->right) q.push(cur->right);
        }
        res.push_back(lvl);
    }
    return res;
}
```

Binary Tree Right Side View

Pattern: BFS last node **Invariant:** take last element of each level

```
vector<int> rightSideView(TreeNode* root){
    vector<int> res;
    if(!root) return res;
    queue<TreeNode*> q; q.push(root);
    while(!q.empty()){
        int n=q.size();
        for(int i=0;i<n;i++){
            auto cur=q.front(); q.pop();
            if(i==n-1) res.push_back(cur->val);
            if(cur->left) q.push(cur->left);
            if(cur->right) q.push(cur->right);
        }
    }
    return res;
}
```

Validate Binary Search Tree

Pattern: DFS bounds **Invariant:** min < node < max

```
bool dfs(TreeNode* root,long mn,long mx){
    if(!root) return true;
    if(root->val<=mn||root->val>=mx) return false;
    return dfs(root->left,mn,root->val)&&dfs(root->right,root->val,mx);
}
```

Kth Smallest Element in BST

Pattern: Inorder traversal **Invariant:** inorder sorted order

```
int k;
int dfs(TreeNode* root){
    if(!root) return -1;
    int l=dfs(root->left);
    if(l!=-1) return l;
    if(--k==0) return root->val;
    return dfs(root->right);
}
```

Construct Binary Tree from Preorder and Inorder

Pattern: Divide & conquer **Invariant:** preorder root splits inorder

```
TreeNode* build(vector<int>& pre,int ps,int pe, vector<int>& in,int is,int ie, unordered_map<int,int>& mp){
    if(ps>pe) return nullptr;
    TreeNode* root=new TreeNode(pre[ps]);
    int idx=mp[root->val]; int l=idx-is;
    root->left=build(pre,ps+1,ps+l,in,is,idx-1,mp);
    root->right=build(pre,ps+l+1,pe,in,idx+1,ie,mp);
    return root;
}
```

Binary Tree Maximum Path Sum

Pattern: DFS with global max **Invariant:** path may start/end anywhere

```
int ans=INT_MIN;
int dfs(TreeNode* root){
    if(!root) return 0;
    int l=max(0,dfs(root->left));
    int r=max(0,dfs(root->right));
    ans=max(ans, root->val+l+r);
    return root->val+max(l,r);
}
```

Serialize and Deserialize Binary Tree

Pattern: DFS preorder **Invariant:** null markers preserve structure

~~~
void ser(TreeNode* root,string& s){
    if(!root){ s+="#,"; return; }
    s+=to_string(root->val)+",";
    ser(root->left,s); ser(root->right,s);
}
TreeNode* des(queue<string>& q){
    if(q.front()=="#"){ q.pop(); return nullptr; }
    TreeNode* root=new TreeNode(stoi(q.front())); q.pop();
    root->left=des(q); root->right=des(q);
    return root;
}


# Heap / Priority Queue

## Kth Largest Element in a Stream
**Pattern:** Min-heap of size k
**Invariant:** Heap contains k largest elements
```cpp
priority_queue<int, vector<int>, greater<int>> pq;
int add(int val){
    pq.push(val);
    if(pq.size()>k) pq.pop();
~~~

```
    return pq.top();
}
```

Last Stone Weight

Pattern: Max-heap **Invariant:** Always smash two largest stones

```
priority_queue<int> pq(stones.begin(), stones.end());
while(pq.size()>1){
    int a=pq.top(); pq.pop();
    int b=pq.top(); pq.pop();
    if(a!=b) pq.push(a-b);
}
return pq.empty()?0:pq.top();
```

K Closest Points to Origin

Pattern: Max-heap size k **Invariant:** Keep k closest so far

```
priority_queue<pair<int,int>> pq;
for(int i=0;i<n;i++){
    int d=points[i][0]*points[i][0]+points[i][1]*points[i][1];
    pq.push({d,i});
    if(pq.size()>k) pq.pop();
}
```

Kth Largest Element in an Array

Pattern: Min-heap size k **Invariant:** Heap top is kth largest

```
priority_queue<int,vector<int>,greater<int>> pq;
for(int x:nums){ pq.push(x); if(pq.size()>k) pq.pop(); }
return pq.top();
```

Task Scheduler

Pattern: Max-heap + greedy **Invariant:** Always schedule most frequent task

```
unordered_map<char,int> cnt;
for(char c:tasks) cnt[c]++;
```

```
priority_queue<int> pq;
for(auto&p:cnt) pq.push(p.second);
int time=0;
while(!pq.empty()){
    int cycle=n+1;
    vector<int> tmp;
    while(cycle-- && !pq.empty()){
        tmp.push_back(pq.top()-1); pq.pop(); time++;
    }
    for(int x:tmp) if(x>0) pq.push(x);
    if(pq.empty()) break;
    time+=cycle+1;
}
```

Design Twitter

Pattern: Heap merge k lists **Invariant:** Tweets sorted by timestamp

```
priority_queue<Tweet> pq; // custom comparator by time
```

Find Median from Data Stream

Pattern: Two heaps **Invariant:** size diff ≤1, maxLeft ≤ minRight

```
priority_queue<int> left;
priority_queue<int,vector<int>,greater<int>> right;
void addNum(int x){
    left.push(x);
    right.push(left.top()); left.pop();
    if(right.size()>left.size()){
        left.push(right.top()); right.pop();
    }
}
double findMedian(){
    if(left.size()>right.size()) return left.top();
    return (left.top()+right.top())/2.0;
}
```

Backtracking

Subsets

Pattern: Decision tree (pick / skip) **Invariant:** Each index has two choices

```
void dfs(int i, vector<int>& nums, vector<int>& cur){
    if(i==nums.size()){ res.push_back(cur); return; }
    dfs(i+1, nums, cur);
    cur.push_back(nums[i]); dfs(i+1, nums, cur); cur.pop_back();
}
```

Combination Sum

Pattern: Unlimited reuse **Invariant:** Stay at same index for reuse

```
void dfs(int i,int sum){
    if(sum==0){ res.push_back(cur); return; }
    if(i==n||sum<0) return;
    cur.push_back(c[i]); dfs(i,sum-c[i]); cur.pop_back();
    dfs(i+1,sum);
}
```

Combination Sum II

Pattern: Skip duplicates **Invariant:** Sort + skip same-level duplicates

```
for(int j=i;j<n;j++){
    if(j>i && c[j]==c[j-1]) continue;
    cur.push_back(c[j]); dfs(j+1,sum-c[j]); cur.pop_back();
}
```

Permutations

Pattern: Used-array **Invariant:** Each element used once

```
void dfs(){
    if(cur.size()==n){ res.push_back(cur); return; }
    for(int i=0;i<n;i++){
        if(used[i]) continue;
        used[i]=1; cur.push_back(nums[i]); dfs(); cur.pop_back(); used[i]=0;
    }
}
```

Subsets II

Pattern: Handle duplicates **Invariant:** Skip duplicate branches

```
for(int j=i;j<n;j++){
    if(j>i && nums[j]==nums[j-1]) continue;
    cur.push_back(nums[j]); dfs(j+1); cur.pop_back();
}
```

Generate Parentheses

Pattern: Count open/close **Invariant:** close ≤ open ≤ n

```
void dfs(int o,int c,string s){
    if(s.size()==2*n){ res.push_back(s); return; }
    if(o<n) dfs(o+1,c,s+'(');
    if(c<o) dfs(o,c+1,s+')');
}
```

Word Search

Pattern: DFS grid **Invariant:** visited once per path

```
bool dfs(int i,int j,int k){
    if(k==w.size()) return true;
    if(i<0||j<0||i>=m||j>=n||b[i][j]!=w[k]) return false;
    char t=b[i][j]; b[i][j]='#';
    bool f=dfs(i+1,j,k+1)||dfs(i-1,j,k+1)||dfs(i,j+1,k+1)||dfs(i,j-1,k+1);
    b[i][j]=t; return f;
}
```

Palindrome Partitioning

Pattern: Partition + check **Invariant:** Each piece palindrome

```
void dfs(int i){
    if(i==n){ res.push_back(cur); return; }
    for(int j=i;j<n;j++){
        if(isPal(i,j)){
            cur.push_back(s.substr(i,j-i+1)); dfs(j+1); cur.pop_back();
        }
    }
}
```

Letter Combinations of Phone Number

Pattern: Cartesian product **Invariant:** One char per digit

```
void dfs(int i,string& s){
    if(i==d.size()){ res.push_back(s); return; }
    for(char c:mp[d[i]]){ s.push_back(c); dfs(i+1,s); s.pop_back(); }
}
```

N-Queens

Pattern: Row-wise placement **Invariant:** One queen per row, no conflicts

```
void dfs(int r){
    if(r==n){ res.push_back(board); return; }
    for(int c=0;c<n;c++){
        if(col[c]||d1[r-c+n]||d2[r+c]) continue;
        col[c]=d1[r-c+n]=d2[r+c]=1; board[r][c]='Q';
        dfs(r+1);
        board[r][c]='.'; col[c]=d1[r-c+n]=d2[r+c]=0;
    }
}
```

Trie

Implement Trie (Prefix Tree)

Pattern: Character-indexed tree **Invariant:** Each node represents a prefix

```
struct TrieNode{
    TrieNode* ch[26] = {};
    bool end=false;
};
TrieNode* root=new TrieNode();
void insert(string w){
    TrieNode* cur=root;
    for(char c:w){
        if(!cur->ch[c-'a']) cur->ch[c-'a']=new TrieNode();
        cur=cur->ch[c-'a'];
    }
    cur->end=true;
}
bool search(string w){
    TrieNode* cur=root;
    for(char c:w){ if(!cur->ch[c-'a']) return false; cur=cur->ch[c-'a']; }
    return cur->end;
}
bool startsWith(string p){
    TrieNode* cur=root;
    for(char c:p){ if(!cur->ch[c-'a']) return false; cur=cur->ch[c-'a']; }
    return true;
}
```

Design Add and Search Words Data Structure

Pattern: Trie + DFS wildcard **Invariant:** '.' can match any child

```
bool dfs(string& w,int i,TrieNode* cur){
    if(i==w.size()) return cur->end;
    if(w[i]=='.'){
        for(auto n:cur->ch) if(n && dfs(w,i+1,n)) return true;
        return false;
    }
    if(!cur->ch[w[i]-'a']) return false;
    return dfs(w,i+1,cur->ch[w[i]-'a']);
}
```

Word Search II

Pattern: Trie + DFS grid **Invariant:** Prune paths not in trie

```
void dfs(int i,int j,TrieNode* cur){
    char c=b[i][j];
    if(c=='#' || !cur->ch[c-'a']) return;
    cur=cur->ch[c-'a'];
    if(cur->end){ res.push_back(word); cur->end=false; }
    b[i][j]='#';
    for(auto [dx,dy]:dirs) dfs(i+dx,j+dy,cur);
    b[i][j]=c;
}
```

Graphs

Number of Islands

Pattern: DFS/BFS on grid **Invariant:** Visit land once

```
void dfs(int i,int j){
    if(i<0||j<0||i>=m||j>=n||g[i][j]=='0') return;
    g[i][j]='0';
    dfs(i+1,j); dfs(i-1,j); dfs(i,j+1); dfs(i,j-1);
}
```

Max Area of Island

Pattern: DFS count **Invariant:** Accumulate connected land

```
int dfs(int i,int j){
    if(i<0||j<0||i>=m||j>=n||g[i][j]==0) return 0;
    g[i][j]=0;
    return 1+dfs(i+1,j)+dfs(i-1,j)+dfs(i,j+1)+dfs(i,j-1);
}
```

Clone Graph

Pattern: DFS + hashmap **Invariant:** One clone per node

```
unordered_map<Node*,Node*> mp;
Node* dfs(Node* n){
    if(!n) return nullptr;
    if(mp[n]) return mp[n];
    Node* c=new Node(n->val);
    mp[n]=c;
    for(auto nei:n->neighbors) c->neighbors.push_back(dfs(nei));
    return c;
}
```

Walls and Gates

Pattern: Multi-source BFS **Invariant:** Shortest distance from gates

```
queue<pair<int,int>> q;
for(all gates) q.push(gate);
while(!q.empty()){
    auto [i,j]=q.front(); q.pop();
    for(dirs){ if(valid && g[x][y]==INF){ g[x][y]=g[i][j]+1; q.push({x,y}); }}
}
```

Rotting Oranges

Pattern: BFS level order **Invariant:** Each minute spreads rot

```
while(!q.empty()){
    int sz=q.size(); time++;
    while(sz--){ auto [i,j]=q.front(); q.pop(); for(dirs) rot; }
}
```

Pacific Atlantic Water Flow

Pattern: Reverse DFS **Invariant:** Flow from oceans inward

```
void dfs(int i,int j,vector<vector<bool>>& vis){
    vis[i][j]=true;
    for(dirs) if(valid && !vis[x][y] && h[x][y]>=h[i][j]) dfs(x,y,vis);
}
```

Course Schedule

Pattern: DFS cycle detection **Invariant:** No back-edge

```
bool dfs(int u){
    if(vis[u]==1) return false;
    if(vis[u]==2) return true;
    vis[u]=1;
    for(int v:g[u]) if(!dfs(v)) return false;
    vis[u]=2; return true;
}
```

Course Schedule II

Pattern: Topological sort **Invariant:** DAG ordering

```
void dfs(int u){ vis[u]=1; for(int v:g[u]) if(!vis[v]) dfs(v);
order.push_back(u); }
```

Graph Valid Tree

Pattern: DFS + edge count **Invariant:** Connected & edges = n-1

```
if(edges.size()!=n-1) return false;
```

Number of Connected Components

Pattern: DFS count **Invariant:** Each DFS marks one component

```
for(i) if(!vis[i]){ dfs(i); cnt++; }
```

Redundant Connection

Pattern: Union Find **Invariant:** Cycle edge detected

```
if(find(u)==find(v)) return {u,v}; unite(u,v);
```

Word Ladder

Pattern: BFS shortest path **Invariant:** One-letter transformations

```
queue<string> q; unordered_set<string> dict;
```

Advanced Graphs

Network Delay Time

Pattern: Dijkstra **Invariant:** Shortest known distance per node

```
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<>> pq;
vector<int> dist(n,INT_MAX);
pq.push({0,k-1}); dist[k-1]=0;
while(!pq.empty()){
    auto [d,u]=pq.top(); pq.pop();
    if(d>dist[u]) continue;
    for(auto [v,w]:adj[u]) if(dist[u]+w<dist[v]){ dist[v]=dist[u]+w;
pq.push({dist[v],v}); }
}
```

Reconstruct Itinerary

Pattern: Eulerian path + DFS **Invariant:** Lexicographical order with stack

```
void dfs(string u){
    while(!adj[u].empty()){
        string v=adj[u].top(); adj[u].pop(); dfs(v);
    }
    res.push_back(u);
}
```

Min Cost to Connect All Points

Pattern: MST (Prim or Kruskal) **Invariant:** Connect all points with minimal total cost

```
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<>> pq;
vector<int> vis(n,0);
```

Swim in Rising Water

Pattern: BFS / Heap **Invariant:** Expand lowest water first

```
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<>> pq;
while(!pq.empty()){
    auto [h,u]=pq.top(); pq.pop();
    for(dirs){ pq.push({max(h,grid[x][y]),x*n+y}); }
}
```

Alien Dictionary

Pattern: Topological sort **Invariant:** Maintain partial order from words

```
for(i) for(j) if(words[i][j]!=words[i+1][j]) add_edge(words[i][j], words[i+1][j]);
topo_sort();
```

Cheapest Flights Within K Stops

Pattern: BFS / Dijkstra with stop limit **Invariant:** Track stops along with cost

```
queue<pair<int,int>> q; // node, cost
while(!q.empty()){
    auto [u,c]=q.front(); q.pop();
    if(stops>k) continue;
    for(auto[v,w]:adj[u]) q.push({v,c+w});
}
```

Dynamic Programming

Climbing Stairs

Pattern: 1D DP / Fibonacci **Invariant:** dp[i]=dp[i-1]+dp[i-2]

```
vector<int> dp(n+1,0); dp[0]=1; dp[1]=1;
for(int i=2;i<=n;i++) dp[i]=dp[i-1]+dp[i-2];
return dp[n];
```

Min Cost Climbing Stairs

Pattern: 1D DP **Invariant:** Cost to reach step i

```
for(int i=2;i<=n;i++) dp[i]=min(dp[i-1]+cost[i-1], dp[i-2]+cost[i-2]);
```

House Robber

Pattern: DP with previous states **Invariant:** max loot till i = max(i-1,i-2+nums[i])

```
int prev=0, curr=0;
for(int x:nums){ int tmp=max(curr, prev+x); prev=curr; curr=tmp; }
return curr;
```

House Robber II

Pattern: Circular array **Invariant:** Max of two ranges

```
return max(rob(nums,0,n-2), rob(nums,1,n-1));
```

Longest Palindromic Substring

Pattern: Expand around center / DP **Invariant:** dp[i][j]=true if s[i..j] palindrome

```
for(int len=2;len<=n;len++) for(int i=0;i+len<=n;i++){ int j=i+len-1; dp[i]
[j]=(s[i]==s[j]) && (len==2||dp[i+1][j-1]); }
```

Palindromic Substrings

Pattern: 1D/2D DP **Invariant:** Count all palindromes

```
for each center expand and count;
```

Decode Ways

Pattern: DP string **Invariant:** dp[i]=sum of ways for 1 or 2 digit

```
dp[i]=0; if(s[i-1]!='0') dp[i]+=dp[i-1]; if(valid(s[i-2..i-1])) dp[i]+=dp[i-2];
```

Coin Change

Pattern: 1D DP **Invariant:** dp[i]=min(dp[i],dp[i-coin]+1)

```
vector<int> dp(amount+1, INT_MAX); dp[0]=0;
for(int c:coins) for(int i=c;i<=amount;i++) if(dp[i-c]!=INT_MAX)
dp[i]=min(dp[i], dp[i-c]+1);
```

Maximum Product Subarray

Pattern: Track max/min **Invariant:** Product can flip with negative

```
int maxP=nums[0], minP=nums[0], res=nums[0];
for(int i=1;i<n;i++){
    if(nums[i]<0) swap(maxP,minP);
    maxP=max(nums[i], maxP*nums[i]); minP=min(nums[i], minP*nums[i]);
    res=max(res,maxP);
}
```

Word Break

Pattern: DP substring **Invariant:** dp[i]=true if s[0..i] can be segmented

```
for j=0..i dp[i]|=dp[j] && wordDict contains s[j..i]
```

Longest Increasing Subsequence

Pattern: DP / Patience sorting **Invariant:** dp[i]=max length ending at i

```
for i: dp[i]=1; for j<i if(nums[j]<nums[i]) dp[i]=max(dp[i], dp[j]+1);
```

Partition Equal Subset Sum

Pattern: Subset sum DP **Invariant:** dp[i]=true if sum i possible

```
for coin in nums for i=sum..coin dp[i]|=dp[i-coin];
```

Advanced Dynamic Programming

Unique Paths

Pattern: Grid DP **Invariant:** dp[i][j]=dp[i-1][j]+dp[i][j-1]

```
vector<vector<int>> dp(m, vector<int>(n,1));
for(int i=1;i<m;i++) for(int j=1;j<n;j++) dp[i][j]=dp[i-1][j]+dp[i][j-1];
```

Longest Common Subsequence

Pattern: 2D DP **Invariant:** dp[i][j]=dp[i-1][j-1]+1 if match else max(dp[i-1][j], dp[i][j-1])

```
for i,j dp[i][j]=s[i]==t[j]?dp[i-1][j-1]+1:max(dp[i-1][j],dp[i][j-1]);
```

Best Time to Buy and Sell Stock with Cooldown

Pattern: DP states **Invariant:** hold, sold, rest

```
hold=max(prevHold, prevRest-price);
sold=prevHold+price;
rest=max(prevRest, prevSold);
```

Coin Change II

Pattern: DP combinations **Invariant:** dp[i]+=dp[i-coin]

```
vector<int> dp(amount+1,0); dp[0]=1;
for(c:coins) for(i=c;i<=amount;i++) dp[i]+=dp[i-c];
```

Target Sum

Pattern: DP subset sum variation **Invariant:** Transform to sum/2 problem

```
dp[i]+=dp[i-num];
```

Interleaving String

Pattern: 2D DP **Invariant:** dp[i][j]=true if s1[0..i] and s2[0..j] form s3[0..i+j]

```
dp[i][j]=(dp[i-1][j] && s1[i-1]==s3[i+j-1])||(dp[i][j-1] && s2[j-1]==s3[i+j-1]);
```

Longest Increasing Path in a Matrix

Pattern: DFS + memo **Invariant:** max path from cell

```
int dfs(i,j){
    if(dp[i][j]) return dp[i][j];
    for dirs dp[i][j]=max(dp[i][j],1+dfs(x,y));
}
```

Distinct Subsequences

Pattern: 2D DP **Invariant:** dp[i][j]=ways s1[0..i] forms s2[0..j]

```
dp[i][j]=dp[i-1][j]+(s1[i-1]==s2[j-1]?dp[i-1][j-1]:0);
```

Edit Distance

Pattern: 2D DP **Invariant:** dp[i][j]=min(insert, delete, replace)

```
dp[i][j]=min({dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+(s1[i-1]!=s2[j-1])});
```

Burst Balloons

Pattern: Interval DP **Invariant:** dp[i][j]=max coins in interval i..j

```
for len in 1..n for i=0..n-len dp[i][i+len+1]=max(dp[i][k]+dp[k]
[i+len+1]+val[i]*val[k]*val[i+len+1]);
```

Regular Expression Matching

Pattern: 2D DP **Invariant:** dp[i][j]=true if s[0..i] matches p[0..j]

```
dp[i][j]=(p[j-1]==s[i-1]||p[j-1]=='.')?dp[i-1][j-1]:p[j-1]=='*'?dp[i][j-2]||
((p[j-2]==s[i-1]||p[j-2]=='.') && dp[i-1][j]):false;
```

Greedy & Interval

Maximum Subarray

Pattern: Kadane's Algorithm **Invariant:** max ending here = max(num, sum+num)

```
int maxSum=nums[0], cur=nums[0];
for(int i=1;i<n;i++){ cur=max(nums[i], cur+nums[i]); maxSum=max(maxSum,cur); }
```

Jump Game

Pattern: Greedy **Invariant:** maxReachable index

```
int reach=0;
for(int i=0;i<=reach;i++){ reach=max(reach,i+nums[i]); if(reach>=n-1) return
true; }
return false;
```

Jump Game II

Pattern: Greedy with range **Invariant:** jump when reach in current range

```
int jumps=0, curEnd=0, curFarthest=0;
for(int i=0;i<n-1;i++){ curFarthest=max(curFarthest,i+nums[i]); if(i==curEnd){
jumps++; curEnd=curFarthest; }}
```

Gas Station

Pattern: Greedy **Invariant:** sum of gas - cost >=0

```
if(totalGas<totalCost) return -1; int tank=0, start=0;
for(int i=0;i<n;i++){ tank+=gas[i]-cost[i]; if(tank<0){ tank=0; start=i+1; }}
return start;
```

Hand of Straights

Pattern: Greedy + map **Invariant:** Always start from smallest

```
map<int,int> mp; for(x:hand) mp[x]++;
for(auto &[k,v]:mp) while(v>0){ for(i=0;i<W;i++){ if(mp[k+i]<v) return false;
mp[k+i]-=v; }}
```

Merge Triplets to Form Target Triplet

Pattern: Greedy **Invariant:** Keep max for each dimension

```
for(t in triplets) if(t[i]<=target[i]) update maxs;
```

Partition Labels

Pattern: Greedy by last occurrence **Invariant:** Partition when i==end

```
for(i) end=max(end,last[s[i]]); if(i==end) res.push_back(end-start+1),
start=i+1;
```

Valid Parenthesis String

Pattern: Greedy count **Invariant:** balance>=0

```
int lo=0, hi=0;
for(c:s){ lo+=c=='('?-1:1; hi+=c=='('?1:-1; if(hi<0) break; lo=max(lo,0); }
return lo==0;
```

Insert Interval

Pattern: Merge intervals **Invariant:** Non-overlapping, insert at correct place

```
for(interval in intervals) if(interval.end<new.start) res.push_back(interval);
else if(interval.start>new.end) res.push_back(new); else
new.start=min(new.start,interval.start), new.end=max(new.end,interval.end);
```

Merge Intervals

Pattern: Sort + merge **Invariant:** merge overlapping intervals

```
sort(intervals.begin(),intervals.end());
for(interval in intervals) if(res.empty()||res.back()[1]<interval[0])
res.push_back(interval);
else res.back()[1]=max(res.back()[1],interval[1]);
```

Non Overlapping Intervals

Pattern: Greedy **Invariant:** Keep interval with earliest end

```
sort(intervals.begin(),intervals.end(),[](auto&a,auto&b){return a[1]<b[1];});
int end=INT_MIN, cnt=0;
for(interval in intervals) if(interval[0]>=end) end=interval[1]; else cnt++;
```

Meeting Rooms

Pattern: Sort + check overlaps **Invariant:** No overlapping start < prev end

```
sort(intervals.begin(),intervals.end());
for i=1..n if(intervals[i][0]<intervals[i-1][1]) return false;
```

Meeting Rooms II

Pattern: Heap **Invariant:** Min-heap for end times

```
sort(intervals.begin(),intervals.end());
priority_queue<int,vector<int>,greater<int>> pq;
for(interval in intervals){ if(!pq.empty() && pq.top()<=interval[0]) pq.pop();
pq.push(interval[1]); }
return pq.size();
```

Minimum Interval to Include Each Query

Pattern: Heap + sort **Invariant:** Keep smallest interval covering current query

```
sort(intervals,queries);
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<>> pq;
for each query: add intervals starting before q; remove intervals ending before q; res[q]=pq.top();
```

Math & Bit Manipulation

Rotate Image

Pattern: Matrix transpose + reverse **Invariant:** In-place rotation

```
for i: for j<i swap(matrix[i][j], matrix[j][i]);
for row: reverse(row.begin(), row.end());
```

Spiral Matrix

Pattern: Layer by layer **Invariant:** Traverse boundaries in order

```
while(top<=bottom && left<=right){ for i=left..right res.push_back(matrix[top][i]); top++;
for i=top..bottom res.push_back(matrix[i][right]); right--;
for i=right..left res.push_back(matrix[bottom][i]); bottom--;
for i=bottom..top res.push_back(matrix[i][left]); left++; }
```

Set Matrix Zeroes

Pattern: Marker in first row/col **Invariant:** Track zeros without extra space

```
for i,j if(matrix[i][j]==0) matrix[i][0]=matrix[0][j]=0;
for i=1..m for j=1..n if(matrix[i][0]==0||matrix[0][j]==0) matrix[i][j]=0;
```

Happy Number

Pattern: Cycle detection (Floyd) **Invariant:** Detect repeating sum

```
int f(int n){ int sum=0; while(n){ sum+=(n%10)*(n%10); n/=10; } return sum; }
```

Plus One

Pattern: Carry propagation **Invariant:** Handle digit overflow

```
for(int i=n-1;i>=0;i--){ digits[i]++; if(digits[i]<10) return digits;
digits[i]=0; }
digits.insert(digits.begin(),1);
```

Pow(x, n)

Pattern: Fast exponentiation **Invariant:** x^n = x^(n/2)*x^(n/2)*(x if odd)

```
double fastPow(double x,int n){ if(n==0) return 1; double half=fastPow(x,n/2);
return n%2?half*half*x:half*half; }
```

Multiply Strings

Pattern: Simulate multiplication **Invariant:** Carry propagation digit-wise

```
vector<int> res(m+n,0);
for i,j res[i+j+1]+= (num1[i]-'0')*(num2[j]-'0');
carry pass
```

Detect Squares

Pattern: Hash map for counts **Invariant:** Track counts per coordinate

```
map<int,map<int,int>> mp;
for x,y in mp do count combinations;
```

Single Number

Pattern: XOR **Invariant:** x^x=0, XOR all numbers

```
int res=0; for(int x:nums) res^=x;
```

Number of 1 Bits

Pattern: Bitmask iteration **Invariant:** count set bits

```
int count=0; while(n){ n&=n-1; count++; }
```

Counting Bits

Pattern: DP / Bit manipulation **Invariant:** dp[i]=dp[i>>1]+(i&1)

```
for i=1..n dp[i]=dp[i>>1]+(i&1);
```

Reverse Bits

Pattern: Bitwise shift **Invariant:** Build result from LSB to MSB

```
for i=0..31 res=(res<<1)|(n&1); n>>=1;
```

Missing Number

Pattern: XOR / sum **Invariant:** total xor - array xor

```
int res=n; for i=0..n-1 res^=i^nums[i];
```

Sum of Two Integers

Pattern: Bitwise addition **Invariant:** sum = a^b, carry=(a&b)<<1

```
while(b!=0){ int carry=(a&b)<<1; a^=b; b=carry; }
```

Reverse Integer

Pattern: Pop/push digits **Invariant:** Check overflow

```
int res=0;
while(x){ int pop=x%10; x/=10; if(res>INT_MAX/10||res<INT_MIN/10) return 0;
res=res*10+pop; }
```