1. Two Sum

Use a hash map to store number → index. For each element, check if target - num is already seen. O(n).

```
def two_sum(nums, target):
    seen = {}  # map from value to index
    for i, num in enumerate(nums):
        diff = target - num
        if diff in seen:
            return [seen[diff], i]
        seen[num] = i
    return []
```

2. Reverse Linked List

Iteratively rewrite next pointers. Maintain prev and curr pointers.

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

3. Detect Cycle in Linked List

Classic Floyd cycle detection with fast and slow pointers.

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

4. Merge Two Sorted Lists

Iteratively merge two sorted lists by always linking the smaller node.

```
def merge_two_lists(l1, l2):
    dummy = tail = ListNode(0)
    while l1 and l2:
        if l1.val < l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    tail.next = l1 or l2
    return dummy.next
```

5. Valid Parentheses

Use a stack. Push opening brackets. Pop when matching closing appears.

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

6. Maximum Subarray (Kadane)

Track running best and current sum. O(n).

```
def max_subarray(nums):
    curr = best = nums[0]
    for n in nums[1:]:
        curr = max(n, curr + n)
        best = max(best, curr)
    return best
```

7. Binary Search

Classic divide-and-conquer search over sorted array.

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

8. Missing Number

Use XOR property: a^a=0, a^0=a. XOR all indices and numbers.

```
def missing_number(nums):
    xor = 0
    for i, n in enumerate(nums):
        xor ^= i ^ n
    return xor ^ len(nums)
```

9. Climbing Stairs

DP with Fibonacci recurrence: ways[n] = ways[n-1] + ways[n-2].

```
def climb_stairs(n):
    a, b = 1, 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

10. Min Stack

Keep an auxiliary stack storing the minimum at each state.

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

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

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

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

    def getMin(self):
        return self.min_stack[-1]
```

11. Number of Islands

DFS flood fill to count distinct connected components of land.

```
def num_islands(grid):
    if not grid:
        return 0
    h, w = len(grid), len(grid[0])
    visited = set()

    def dfs(r, c):
        if r<0 or c<0 or r>=h or c>=w:
            return
        if grid[r][c]=='0' or (r,c) in visited:
            return
        visited.add((r,c))
        dfs(r+1,c); dfs(r-1,c)
        dfs(r,c+1); dfs(r,c-1)

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

12. Kth Largest Element (Quickselect)

Partition array around a random pivot until pivot lands at correct index.

```
import random

def kth_largest(nums, k):
    k = len(nums) - k

    def quickselect(l, r):
        pivot = nums[random.randint(l, r)]
        i, j = l, r
        while i <= j:
            while nums[i] < pivot:
                i += 1
            while nums[j] > pivot:
                j -= 1
            if i <= j:
                nums[i], nums[j] = nums[j], nums[i]
                i += 1; j -= 1
        if k <= j:
            return quickselect(l, j)
        if k >= i:
            return quickselect(i, r)
        return nums[k]

    return quickselect(0, len(nums)-1)
```

13. LRU Cache

Use OrderedDict or hashmap + linked list.

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key):
        if key not in self.cache:
            return -1
        self.cache.move_to_end(key)
        return self.cache[key]

    def put(self, key, value):
        if key in self.cache:
            self.cache.move_to_end(key)
        self.cache[key] = value
        if len(self.cache)>self.capacity:
            self.cache.popitem(last=False)
```

14. Rotate Array

Reverse approach: reverse whole, reverse first k, reverse remaining.

```
def rotate(nums, k):
    k %= len(nums)
    def rev(l, r):
        while l<r:
            nums[l], nums[r] = nums[r], nums[l]
            l+=1; r-=1
    rev(0, len(nums)-1)
    rev(0, k-1)
    rev(k, len(nums)-1)
```

15. Product of Array Except Self

No division. Use prefix array then multiply by suffix.

```
def product_except_self(nums):
    n = len(nums)
    res = [1]*n
    prefix = 1
    for i in range(n):
        res[i] = prefix
        prefix *= nums[i]
    suffix = 1
    for i in reversed(range(n)):
        res[i] *= suffix
        suffix *= nums[i]
    return res
```

16. Top K Frequent Elements

Use Counter + heap.

```
from collections import Counter
import heapq

def top_k_frequent(nums, k):
    freq = Counter(nums)
    return [x for x,_ in heapq.nlargest(k, freq.items(), key=lambda p: p[1])]
```

17. Palindrome Linked List

Find midpoint, reverse second half, compare.

```
def is_palindrome(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    prev = None
    while slow:
        nxt = slow.next
        slow.next = prev
        prev = slow
        slow = nxt
    left, right = head, prev
    while right:
        if left.val != right.val:
            return False
        left = left.next
        right = right.next
    return True
```

18. Flood Fill

Standard DFS repaint.

```
def flood_fill(image, sr, sc, newColor):
        h, w = len(image), len(image[0])
        orig = image[sr][sc]
        if orig == newColor:
            return image
        def dfs(r,c):
            if r<0 or c<0 or r>=h or c>=w: return
            if image[r][c] != orig: return
            image[r][c] = newColor
            dfs(r+1,c); dfs(r-1,c); dfs(r,c+1); dfs(r,c-1)
        dfs(sr,sc)
        return image
```

19. Meeting Rooms

Sort intervals and check for overlap.

```
    def can_attend_meetings(intervals):
        intervals.sort()
        for i in range(1, len(intervals)):
            if intervals[i][0] < intervals[i-1][1]:
                return False
        return True
```

20. Merge Intervals

Merge overlapping intervals after sorting.

```
    def merge_intervals(intervals):
        intervals.sort()
        merged = []
        for s,e in intervals:
            if not merged or merged[-1][1] < s:
                merged.append([s,e])
            else:
                merged[-1][1] = max(merged[-1][1], e)
        return merged
```

21. Longest Substring Without Repeating Characters

Sliding window with set to track seen chars.

```
    def length_of_longest_substring(s):
        seen = set()
        l = best = 0
        for r,c in enumerate(s):
            while c in seen:
                seen.remove(s[l])
                l += 1
            seen.add(c)
            best = max(best, r-l+1)
        return best
```

22. Search in Rotated Sorted Array

Modified binary search that checks which half is sorted.

```
    def search_rotated(nums, target):
```

```
    l, r = 0, len(nums)-1
    while l<=r:
        mid = (l+r)//2
        if nums[mid]==target:
            return mid
        if nums[l] <= nums[mid]:
            if nums[l] <= target < nums[mid]:
                r = mid-1
            else:
                l = mid+1
        else:
            if nums[mid] < target <= nums[r]:
                l = mid+1
            else:
                r = mid-1
    return -1
```

23. Min Cost Climbing Stairs

DP storing minimal cost to reach step.

```
def min_cost(cost):
    a, b = cost[0], cost[1]
    for i in range(2, len(cost)):
        a, b = b, cost[i] + min(a,b)
    return min(a,b)
```

24. Valid Anagram

Two strings are anagrams if char counts match.

```
from collections import Counter

def is_anagram(a,b):
    return Counter(a)==Counter(b)
```

25. Binary Tree Inorder Traversal

Left → root → right recursion.

```
def inorder(root):
    if not root:
        return []
    return inorder(root.left) + [root.val] + inorder(root.right)
```

26. Symmetric Tree

Tree is symmetric if left and right subtrees are mirror.

```
def is_symmetric(root):
    def mirror(a,b):
        if not a and not b: return True
        if not a or not b: return False
        return a.val==b.val and mirror(a.left,b.right) and mirror(a.right,b.left)
    return mirror(root,root)
```

27. Path Sum

DFS subtracting node value from target.

```
def has_path_sum(root, target):
    if not root:
        return False
    if not root.left and not root.right:
        return target==root.val
    return has_path_sum(root.left, target-root.val) or has_path_sum(root.right, target-root.val)
```

28. Diameter of Binary Tree

DFS returning height, updating global best diameter.

```
def diameter_of_binary_tree(root):
    best = 0
    def dfs(node):
        nonlocal best
        if not node:
            return 0
        left = dfs(node.left)
        right = dfs(node.right)
        best = max(best, left+right)
        return max(left,right)+1
    dfs(root)
    return best
```

29. Level Order Traversal

BFS processing nodes by level.

```
from collections import deque

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

30. House Robber

DP with two states: rob or skip.

```
def rob(nums):
    a = b = 0
    for n in nums:
        a, b = b, max(b, a+n)
    return b
```