Complete Data Structures & Algorithms Reference Guide

Table of Contents

Arrays

Theory

Arrays are contiguous memory locations storing elements of the same data type. They provide O(1) access time using indices but have fixed size in most languages.

Time Complexity:

- Access: O(1)
- Search: O(n)
- Insert/Delete: O(n)

Example

Finding the maximum element in an array: [3, 7, 1, 9, 2] → Maximum is 9

Code

python

```python
# Find max element
def find_max(arr):
    max_val = arr[0]
    for num in arr:
        if num > max_val:
            max_val = num
    return max_val

# Reverse array
def reverse(arr):
    left, right = 0, len(arr) - 1
    while left < right:
        arr[left], arr[right] = arr[right], arr[left]
        left += 1
        right -= 1
```

Strings

Theory

Strings are sequences of characters. In most languages, they're immutable (can't be changed in place). Common operations include substring search, palindrome checking, and pattern matching.

Key Operations:

- Concatenation: O(n)
- Substring: O(n)
- Comparison: O(n)

Example

Check if "racecar" is a palindrome → Yes (reads same forwards and backwards)

Code

python

```python
# Check palindrome
def is_palindrome(s):
    return s == s[::-1]

# Count character frequency
def char_frequency(s):
    freq = {}
    for char in s:
        freq[char] = freq.get(char, 0) + 1
    return freq
```

Linked Lists

Theory

A linked list is a linear data structure where elements (nodes) are connected via pointers. Each node contains data and a reference to the next node.

Types:

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Time Complexity:

- Access: O(n)
- Search: O(n)
- Insert/Delete at head: O(1)
- Insert/Delete at position: O(n)

Example

List: 1 → 3 → 5 → 7 Insert 4 after 3: 1 → 3 → 4 → 5 → 7

Code

python

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Reverse linked list
def reverse_list(head):
    prev = None
    current = head
    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node
    return prev

# Detect cycle
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
```

Stacks

Theory

A stack follows Last In First Out (LIFO) principle. Think of a stack of plates where you add and remove from the top.

Operations:

- Push: O(1)
- Pop: O(1)
- Peek: O(1)

Applications: Function calls, undo operations, expression evaluation, backtracking

Example

Stack operations: Push(1) → Push(2) → Push(3) → Pop() returns 3 → Peek() returns 2

Code

python

```python
# Using list as stack
stack = []

# Valid parentheses
def is_valid_parentheses(s):
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}
    for char in s:
        if char in mapping:
            if not stack or stack[-1] != mapping[char]:
                return False
            stack.pop()
        else:
            stack.append(char)
    return len(stack) == 0
```

Queues

Theory

A queue follows First In First Out (FIFO) principle. Like a line at a ticket counter where the first person in line is served first.

Operations:

- Enqueue: O(1)
- Dequeue: O(1)
- Front: O(1)

Types: Simple Queue, Circular Queue, Priority Queue, Deque

Example

Queue operations: Enqueue(1) → Enqueue(2) → Enqueue(3) → Dequeue() returns 1

Code

python

```python
from collections import deque

# Using deque
queue = deque()
queue.append(1)  # enqueue
queue.popleft()  # dequeue

# Implement queue using stacks
class QueueUsingStacks:
    def __init__(self):
        self.s1 = []
        self.s2 = []

    def enqueue(self, x):
        self.s1.append(x)

    def dequeue(self):
        if not self.s2:
            while self.s1:
                self.s2.append(self.s1.pop())
        return self.s2.pop() if self.s2 else None
```

Trees

Theory

A tree is a hierarchical data structure with a root node and children nodes. Each node can have multiple children but only one parent.

Terminology:

- Root: Top node
- Leaf: Node with no children
- Height: Longest path from root to leaf
- Depth: Distance from root to a node

Traversals:

- Inorder (Left, Root, Right): Used for BST to get sorted order
- Preorder (Root, Left, Right): Used for tree copying
- Postorder (Left, Right, Root): Used for tree deletion
- Level Order: Level by level (BFS)

Example

```
    1
   / \
  2   3
 / \
4   5
```

Inorder: 4, 2, 5, 1, 3 Preorder: 1, 2, 4, 5, 3 Postorder: 4, 5, 2, 3, 1

Code

python

```python
class TreeNode:
    def __init__(self, val=0):
        self.val = val
        self.left = None
        self.right = None

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

# Level order traversal
def level_order(root):
    if not root:
        return []
    result, queue = [], [root]
    while queue:
        level_size = len(queue)
        level = []
        for _ in range(level_size):
            node = queue.pop(0)
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result

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

Binary Search Trees

Theory

A BST is a binary tree where for each node, all values in the left subtree are smaller and all values in the right subtree are larger.

Properties:

- Inorder traversal gives sorted sequence
- Average case operations: O(log n)
- Worst case (skewed tree): O(n)

Operations: Insert, Delete, Search

Example

```
    5
   / \
  3   7
 / \   \
2   4   8
```

Search 4: Go left (4 < 5), go right (4 > 3), found!

Code

python

```python
# Insert in BST
def insert_bst(root, val):
    if not root:
        return TreeNode(val)
    if val < root.val:
        root.left = insert_bst(root.left, val)
    else:
        root.right = insert_bst(root.right, val)
    return root

# Search in BST
def search_bst(root, val):
    if not root or root.val == val:
        return root
    if val < root.val:
        return search_bst(root.left, val)
    return search_bst(root.right, val)

# Validate BST
def is_valid_bst(root, min_val=float('-inf'), max_val=float('inf')):
    if not root:
        return True
    if root.val <= min_val or root.val >= max_val:
        return False
    return (is_valid_bst(root.left, min_val, root.val) and
            is_valid_bst(root.right, root.val, max_val))
```

Heaps

Theory

A heap is a complete binary tree that satisfies the heap property. In a max heap, parent nodes are larger than children. In a min heap, parent nodes are smaller than children.

Properties:

- Complete binary tree
- Max Heap: parent ≥ children
- Min Heap: parent ≤ children

Time Complexity:

- Insert: O(log n)
- Delete Max/Min: O(log n)

- Get Max/Min: O(1)
- Heapify: O(n)

Applications: Priority Queue, Heap Sort, finding Kth largest/smallest

Example

Max Heap: [50, 30, 20, 15, 10, 8, 16]

```
        50
       /  \
     30    20
    / \    / \
   15 10  8  16
```

Code

python

```python
import heapq

# Min heap operations
heap = []
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 5)
min_val = heapq.heappop(heap)  # Returns 1

# Find Kth largest
def kth_largest(nums, k):
    heap = nums[:k]
    heapq.heapify(heap)
    for num in nums[k:]:
        if num > heap[0]:
            heapq.heapreplace(heap, num)
    return heap[0]

# For max heap, negate values
max_heap = []
heapq.heappush(max_heap, -5)
heapq.heappush(max_heap, -3)
max_val = -heapq.heappop(max_heap)  # Returns 5
```

Hashing

Theory

Hashing maps data to fixed-size values using a hash function. Hash tables provide average O(1) time for insert, delete, and search operations.

Collision Resolution:

- Chaining: Store colliding elements in a linked list
- Open Addressing: Find another empty slot

Load Factor: n/m (where n = elements, m = table size) When load factor > threshold, rehashing occurs

Example

Hash function: h(x) = x % 10 Insert 25 → Index 5, Insert 35 → Index 5 (collision, resolve using chaining)

Code

python

```python
# Two sum using hash map
def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
    return []

# First non-repeating character
def first_unique_char(s):
    freq = {}
    for char in s:
        freq[char] = freq.get(char, 0) + 1
    for i, char in enumerate(s):
        if freq[char] == 1:
            return i
    return -1
```

Graphs

Theory

A graph consists of vertices (nodes) and edges connecting them. Graphs can be directed or undirected, weighted or unweighted.

Representations:

- Adjacency Matrix: 2D array, $O(V^2)$ space
- Adjacency List: Array of lists, $O(V+E)$ space

Types:

- Directed vs Undirected
- Weighted vs Unweighted
- Cyclic vs Acyclic (DAG)
- Connected vs Disconnected

Common Algorithms:

- BFS (Breadth First Search): Level-by-level exploration
- DFS (Depth First Search): Explore as deep as possible
- Dijkstra: Shortest path in weighted graph
- Topological Sort: Linear ordering of vertices (DAG)

Example

Graph: A-B, A-C, B-D, C-D BFS from A: A, B, C, D DFS from A: A, B, D, C (or A, C, D, B)

Code

python

```python
from collections import defaultdict, deque

# Graph representation
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

# BFS
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)
    result = []

    while queue:
        node = queue.popleft()
        result.append(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return result

# DFS
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    result = [node]
    for neighbor in graph[node]:
        if neighbor not in visited:
            result.extend(dfs(graph, neighbor, visited))
    return result

# Detect cycle in directed graph
def has_cycle(graph, node, visited, rec_stack):
    visited.add(node)
    rec_stack.add(node)
```

```
    for neighbor in graph[node]:
        if neighbor not in visited:
            if has_cycle(graph, neighbor, visited, rec_stack):
                return True
        elif neighbor in rec_stack:
            return True

    rec_stack.remove(node)
    return False
```

Sorting Algorithms

Theory

1. Bubble Sort

- Compare adjacent elements and swap if needed
- Time: $O(n^2)$, Space: $O(1)$

2. Selection Sort

- Find minimum and place it at the beginning
- Time: $O(n^2)$, Space: $O(1)$

3. Insertion Sort

- Build sorted array one element at a time
- Time: $O(n^2)$, Space: $O(1)$

4. Merge Sort

- Divide and conquer, merge sorted halves
- Time: $O(n \log n)$, Space: $O(n)$

5. Quick Sort

- Choose pivot, partition around it
- Time: $O(n \log n)$ average, $O(n^2)$ worst, Space: $O(\log n)$

6. Heap Sort

- Build max heap, extract max repeatedly
- Time: $O(n \log n)$, Space: $O(1)$

Example

Array: [5, 2, 8, 1, 9] After sorting: [1, 2, 5, 8, 9]

Code

python

```python
# Merge Sort
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

# Quick Sort
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Searching Algorithms

Theory

1. Linear Search

- Check each element sequentially
- Time: O(n), Space: O(1)

2. Binary Search

- Only works on sorted arrays
- Divide search space in half each time
- Time: O(log n), Space: O(1)

3. Binary Search Variations

- First occurrence
- Last occurrence
- Count occurrences

Example

Binary Search in [1, 3, 5, 7, 9, 11] for target 7: Step 1: mid = 5, 7 > 5, search right Step 2: mid = 9, 7 < 9, search left Step 3: mid = 7, found!

Code

python

```python
# Binary Search
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# Find first occurrence
def first_occurrence(arr, target):
    left, right = 0, len(arr) - 1
    result = -1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            result = mid
            right = mid - 1  # Continue searching left
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return result

# Search in rotated sorted array
def search_rotated(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        if arr[left] <= arr[mid]:  # Left half sorted
            if arr[left] <= target < arr[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:  # Right half sorted
```

```python
        if arr[mid] < target <= arr[right]:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Recursion & Backtracking

Theory

Recursion: A function calling itself with a base case to stop infinite recursion.

Components:

- Base case: Condition to stop recursion
- Recursive case: Problem reduced to smaller subproblem

Backtracking: Try all possibilities, backtrack when you hit a dead end. Used for constraint satisfaction problems.

Applications: Permutations, combinations, N-Queens, Sudoku, maze solving

Example

Factorial: 5! = 5 × 4! = 5 × 4 × 3! = 5 × 4 × 3 × 2 × 1 = 120

Code

python

```python
# Factorial
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

# Generate all permutations
def permutations(arr):
    if len(arr) <= 1:
        return [arr]
    result = []
    for i in range(len(arr)):
        rest = arr[:i] + arr[i+1:]
        for p in permutations(rest):
            result.append([arr[i]] + p)
    return result

# Generate all subsets
def subsets(nums):
    result = []
    def backtrack(start, path):
        result.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()
    backtrack(0, [])
    return result

# N-Queens
def solve_n_queens(n):
    result = []
    board = [['.'] * n for _ in range(n)]

    def is_safe(row, col):
        for i in range(row):
            if board[i][col] == 'Q':
                return False
        i, j = row - 1, col - 1
        while i >= 0 and j >= 0:
            if board[i][j] == 'Q':
```

```python
            return False
        i -= 1
        j -= 1
    i, j = row - 1, col + 1
    while i >= 0 and j < n:
        if board[i][j] == 'Q':
            return False
        i -= 1
        j += 1
    return True

def backtrack(row):
    if row == n:
        result.append([''.join(r) for r in board])
        return
    for col in range(n):
        if is_safe(row, col):
            board[row][col] = 'Q'
            backtrack(row + 1)
            board[row][col] = '.'

backtrack(0)
return result
```

Dynamic Programming

Theory

Dynamic Programming solves complex problems by breaking them into overlapping subproblems and storing results to avoid recomputation.

Approaches:

1. **Top-Down (Memoization):** Recursion + caching
2. **Bottom-Up (Tabulation):** Iterative, fill table

When to use DP:

- Optimal substructure (optimal solution contains optimal solutions to subproblems)
- Overlapping subproblems (same subproblems solved multiple times)

Steps:

1. Define the state
2. Find recurrence relation

3. Handle base cases
4. Decide order of computation

Example

Fibonacci: F(5) = F(4) + F(3) = (F(3) + F(2)) + (F(2) + F(1)) Without DP: F(3) and F(2) computed multiple times With DP: Compute once, store result

Code

python

```python
# Fibonacci (Memoization)
def fib_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
    return memo[n]

# Fibonacci (Tabulation)
def fib_tab(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# 0/1 Knapsack
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]],
                               dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][capacity]

# Longest Common Subsequence
def lcs(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i-1] == s2[j-1]:
```

```
            dp[i][j] = 1 + dp[i-1][j-1]
        else:
            dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[m][n]

# Coin Change
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0

    for i in range(1, amount + 1):
        for coin in coins:
            if coin <= i:
                dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1
```

Greedy Algorithms

Theory

Greedy algorithms make locally optimal choices at each step, hoping to find a global optimum.

Properties:

- Greedy choice property: Local optimum leads to global optimum
- Optimal substructure: Optimal solution contains optimal solutions to subproblems

When to use:

- Problem has greedy choice property
- Simpler than DP when applicable

Common Problems: Activity selection, Huffman coding, fractional knapsack, job scheduling

Example

Coin change (greedy): Make change for 36 cents using [25, 10, 5, 1] Take 25 (largest), remaining 11 Take 10, remaining 1 Take 1, done. Total: 3 coins

Code

python

```python
# Activity Selection
def activity_selection(start, finish):
    activities = sorted(zip(start, finish), key=lambda x: x[1])
    selected = [activities[0]]
    last_finish = activities[0][1]

    for activity in activities[1:]:
        if activity[0] >= last_finish:
            selected.append(activity)
            last_finish = activity[1]
    return len(selected)

# Fractional Knapsack
def fractional_knapsack(weights, values, capacity):
    items = sorted(zip(values, weights),
                  key=lambda x: x[0]/x[1], reverse=True)
    total_value = 0

    for value, weight in items:
        if capacity >= weight:
            total_value += value
            capacity -= weight
        else:
            total_value += value * (capacity / weight)
            break
    return total_value

# Jump Game
def can_jump(nums):
    max_reach = 0
    for i in range(len(nums)):
        if i > max_reach:
            return False
        max_reach = max(max_reach, i + nums[i])
    return True
```

Bit Manipulation

Theory

Bit manipulation operates directly on binary representations of numbers.

Basic Operations:

- AND (&): Both bits 1
- OR (|): At least one bit 1
- XOR (^): Bits different
- NOT (~): Flip bits
- Left Shift (<<): Multiply by 2
- Right Shift (>>): Divide by 2

Common Tricks:

- Check if even: `n & 1 == 0`
- Get ith bit: `(n >> i) & 1`
- Set ith bit: `n | (1 << i)`
- Clear ith bit: `n & ~(1 << i)`
- Toggle ith bit: `n ^ (1 << i)`
- Check power of 2: `n & (n-1) == 0`
- Count set bits: Brian Kernighan's algorithm

Example

XOR properties: a ^ a = 0, a ^ 0 = a Find single number in [2,2,3,4,4]: 2^2^3^4^4 = 0^3^0 = 3

Code

python

```python
# Count set bits
def count_set_bits(n):
    count = 0
    while n:
        n &= n - 1  # Remove rightmost set bit
        count += 1
    return count

# Single number (XOR trick)
def single_number(nums):
    result = 0
    for num in nums:
        result ^= num
    return result

# Power of 2
def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0

# Swap two numbers
def swap(a, b):
    a = a ^ b
    b = a ^ b
    a = a ^ b
    return a, b

# Get ith bit
def get_bit(n, i):
    return (n >> i) & 1

# Set ith bit
def set_bit(n, i):
    return n | (1 << i)

# Clear ith bit
def clear_bit(n, i):
    return n & ~(1 << i)
```

Sliding Window

Theory

Sliding window technique maintains a window of elements and slides it across the array/string to solve problems efficiently.

Types:

1. **Fixed Size Window:** Window size is constant
2. **Variable Size Window:** Window size changes based on condition

Pattern:

- Expand window by moving right pointer
- Contract window by moving left pointer
- Update answer at each step

When to use: Problems involving contiguous subarrays/substrings

Example

Max sum of subarray of size 3 in [2,1,5,1,3,2]: Window [2,1,5] sum=8, [1,5,1] sum=7, [5,1,3] sum=9, [1,3,2] sum=6
Maximum: 9

Code

python

```python
# Max sum subarray of size k (Fixed window)
def max_sum_subarray(arr, k):
    window_sum = sum(arr[:k])
    max_sum = window_sum

    for i in range(k, len(arr)):
        window_sum += arr[i] - arr[i - k]
        max_sum = max(max_sum, window_sum)
    return max_sum

# Longest substring without repeating chars (Variable window)
def longest_unique_substring(s):
    char_set = set()
    left = max_len = 0

    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right])
        max_len = max(max_len, right - left + 1)
    return max_len

# Minimum window substring
def min_window(s, t):
    if not t or not s:
        return ""

    dict_t = {}
    for char in t:
        dict_t[char] = dict_t.get(char, 0) + 1

    required = len(dict_t)
    left = right = 0
    formed = 0
    window_counts = {}
    ans = float('inf'), None, None

    while right < len(s):
        char = s[right]
        window_counts[char] = window_counts.get(char, 0) + 1
```

```python
        if char in dict_t and window_counts[char] == dict_t[char]:
            formed += 1

        while left <= right and formed == required:
            if right - left + 1 < ans[0]:
                ans = (right - left + 1, left, right)

            char = s[left]
            window_counts[char] -= 1
            if char in dict_t and window_counts[char] < dict_t[char]:
                formed -= 1
            left += 1

        right += 1

    return "" if ans[0] == float('inf') else s[ans[1]:ans[2] + 1]

# Max consecutive ones with k flips
def max_consecutive_ones(nums, k):
    left = max_len = zeros = 0

    for right in range(len(nums)):
        if nums[right] == 0:
            zeros += 1

        while zeros > k:
            if nums[left] == 0:
                zeros -= 1
            left += 1

        max_len = max(max_len, right - left + 1)
    return max_len
```

Two Pointers

Theory

Two pointers technique uses two pointers to iterate through data structures, often from different ends or at different speeds.

Patterns:

1. **Opposite Ends:** Start from both ends, move towards center
2. **Same Direction:** Both move forward, at different speeds
3. **Fast & Slow:** Detect cycles, find middle

When to use: Sorted arrays, linked lists, pairs/triplets problems

Example

Two sum in sorted array [1,2,3,4,6], target=6: left=0, right=4: 1+6=7 > 6, move right left=0, right=3: 1+4=5 < 6, move left left=1, right=3: 2+4=6, found!

Code

python

```python
# Two sum in sorted array
def two_sum_sorted(arr, target):
    left, right = 0, len(arr) - 1

    while left < right:
        curr_sum = arr[left] + arr[right]
        if curr_sum == target:
            return [left, right]
        elif curr_sum < target:
            left += 1
        else:
            right -= 1
    return []

# Three sum
def three_sum(nums):
    nums.sort()
    result = []

    for i in range(len(nums) - 2):
        if i > 0 and nums[i] == nums[i-1]:
            continue

        left, right = i + 1, len(nums) - 1
        while left < right:
            curr_sum = nums[i] + nums[left] + nums[right]
            if curr_sum == 0:
                result.append([nums[i], nums[left], nums[right]])
                while left < right and nums[left] == nums[left+1]:
                    left += 1
                while left < right and nums[right] == nums[right-1]:
                    right -= 1
                left += 1
                right -= 1
            elif curr_sum < 0:
                left += 1
            else:
                right -= 1
    return result

# Remove duplicates from sorted array
```

```python
def remove_duplicates(nums):
    if not nums:
        return 0

    write = 1
    for read in range(1, len(nums)):
        if nums[read] != nums[read - 1]:
            nums[write] = nums[read]
            write += 1
    return write

# Container with most water
def max_area(heights):
    left, right = 0, len(heights) - 1
    max_water = 0

    while left < right:
        width = right - left
        height = min(heights[left], heights[right])
        max_water = max(max_water, width * height)

        if heights[left] < heights[right]:
            left += 1
        else:
            right -= 1
    return max_water
```

Tries

Theory

A trie (prefix tree) is a tree data structure used to store strings efficiently. Each node represents a character, and paths from root to node form strings.

Properties:

- Root is empty
- Each path from root represents a prefix
- Common prefixes share paths

Operations:

- Insert: O(m) where m is word length
- Search: O(m)

- Prefix search: O(m)

Applications: Autocomplete, spell checker, IP routing, dictionary

Example

Insert "cat", "car", "dog":

```
       root
       / \
      c   d
      |   |
      a   o
     / \  |
    t   r g
```

Code

python

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

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

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end

    def starts_with(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

# Find all words with given prefix
def find_words_with_prefix(trie, prefix):
    node = trie.root
    for char in prefix:
        if char not in node.children:
            return []
        node = node.children[char]
```

```python
    words = []
    def dfs(node, path):
        if node.is_end:
            words.append(prefix + path)
        for char, child in node.children.items():
            dfs(child, path + char)

    dfs(node, "")
    return words
```

Union Find (Disjoint Set)

Theory

Union Find is a data structure that keeps track of elements partitioned into disjoint sets. It supports two operations efficiently:

Operations:

- Find: Determine which set an element belongs to
- Union: Merge two sets

Optimizations:

- Path Compression: Make tree flat during find
- Union by Rank: Attach smaller tree under larger tree

Time Complexity:

- With optimizations: Nearly O(1) amortized

Applications: Connected components, cycle detection, Kruskal's MST, network connectivity

Example

Elements: {1,2,3,4,5} Union(1,2), Union(3,4), Union(2,4) Result: {1,2,3,4} and {5} are separate components

Code

python

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x == root_y:
            return False

        # Union by rank
        if self.rank[root_x] < self.rank[root_y]:
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
        else:
            self.parent[root_y] = root_x
            self.rank[root_x] += 1
        return True

    def connected(self, x, y):
        return self.find(x) == self.find(y)

# Number of connected components
def count_components(n, edges):
    uf = UnionFind(n)
    for u, v in edges:
        uf.union(u, v)

    return len(set(uf.find(i) for i in range(n)))

# Detect cycle in undirected graph
def has_cycle_undirected(n, edges):
    uf = UnionFind(n)
```

```
    for u, v in edges:
        if not uf.union(u, v):
            return True
    return False
```

Advanced Topics Summary

Segment Trees

Used for range queries and updates in O(log n) time.

- Applications: Range sum, range minimum, range maximum

Fenwick Tree (Binary Indexed Tree)

Efficient for prefix sum queries and point updates.

- Time: O(log n) for both query and update

Suffix Arrays & Trees

Used for pattern matching and string problems.

- Applications: Longest common substring, pattern search

AVL Trees & Red-Black Trees

Self-balancing BSTs that maintain O(log n) height.

- Guarantee balanced operations

B-Trees & B+ Trees

Multi-way search trees used in databases and file systems.

- Minimize disk I/O operations

KMP Algorithm

Pattern matching in O(n+m) time.

- Builds failure function for efficient matching

Rabin-Karp Algorithm

String matching using hashing.

- Good for multiple pattern search

Floyd-Warshall Algorithm

All pairs shortest path in $O(V^3)$.

- Works with negative edges (no negative cycles)

Bellman-Ford Algorithm

Single source shortest path, handles negative edges.

- Time: $O(VE)$

Prim's & Kruskal's Algorithms

Minimum Spanning Tree algorithms.

- Prim's: $O(E \log V)$ with heap
- Kruskal's: $O(E \log E)$ with sorting

Dinic's Algorithm

Maximum flow in a network.

- Time: $O(V^2E)$

Problem-Solving Strategies

1. Understand the Problem

- Read carefully, identify inputs/outputs
- Consider edge cases
- Ask clarifying questions

2. Choose the Right Data Structure

- Arrays: Fast access
- Hash Maps: Fast lookup
- Heaps: Priority-based operations
- Trees: Hierarchical data
- Graphs: Relationships between entities

3. Recognize Patterns

- Two Pointers: Sorted arrays, pairs
- Sliding Window: Contiguous subarrays
- Binary Search: Sorted data, monotonic functions
- DFS/BFS: Tree/graph traversal
- DP: Optimal substructure, overlapping subproblems
- Greedy: Local optimum leads to global optimum

4. Time/Space Complexity Analysis

- Always analyze before coding
- Consider trade-offs
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

5. Test Your Solution

- Test with examples
- Consider edge cases: empty input, single element, duplicates
- Think about overflow, negative numbers

Common Complexity Classes

Complexity	Name	Example
$O(1)$	Constant	Array access, hash lookup
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Linear search, array traversal
$O(n \log n)$	Linearithmic	Merge sort, heap sort
$O(n^2)$	Quadratic	Bubble sort, nested loops
$O(2^n)$	Exponential	Recursive fibonacci
$O(n!)$	Factorial	Permutations

Tips for Coding Interviews

1. **Think out loud** - Explain your thought process
2. **Start with brute force** - Then optimize
3. **Use meaningful variable names** - Makes code readable
4. **Handle edge cases** - Empty inputs, single elements
5. **Test your code** - Walk through with examples
6. **Ask questions** - Clarify requirements
7. **Practice regularly** - Consistency is key
8. **Learn from mistakes** - Review failed attempts

Resources for Further Learning

Online Platforms

- LeetCode: Practice problems by pattern
- HackerRank: Structured learning paths
- CodeForces: Competitive programming
- GeeksforGeeks: Theory and practice

Books

- "Cracking the Coding Interview" by Gayle Laakmann McDowell
- "Introduction to Algorithms" by CLRS
- "Algorithm Design Manual" by Steven Skiena

Practice Strategy

- Start with easy problems
- Focus on one pattern at a time
- Gradually increase difficulty
- Review solutions of others
- Time yourself for real interview practice

Final Notes

This guide covers the fundamental DSA topics you need for interviews and competitive programming. The key to mastery is consistent practice. Start with basics, understand the underlying concepts, and gradually tackle more complex problems.

Remember:

- **Understand, don't memorize** - Focus on why algorithms work
- **Practice regularly** - 30 minutes daily beats weekend marathons
- **Learn patterns** - Most problems are variations of known patterns
- **Stay consistent** - Progress compounds over time

Good luck with your DSA journey!