Algorithm & Data Structure Cheatsheets

Table of Contents

Introduction

Welcome to the Algorithm & Data Structure cheatsheets! This repository contains comprehensive resources for mastering algorithms and data structures commonly used in software engineering interviews and competitive programming.

How to Use This Repository

- **Beginners**: Start with the Learning Plan below
- **Interview Prep**: Focus on the algorithm references organized by category
- **Practice**: Use the Learning Activities to test your knowledge

If you're new to algorithms or need a refresher:

1. Begin with fundamental data structures: arrays, linked lists, stacks, and queues
2. Follow the structured learning plan
3. Practice with examples of increasing difficulty (Easy → Medium → Hard)
4. Use the interactive activities to reinforce your understanding

Array Algorithms

Kadane's Algorithm

> **Difficulty**: ⭐⭐ Medium

Kadane's algorithm efficiently finds the maximum sum contiguous subarray within a one-dimensional array of numbers. It uses dynamic programming to track the maximum sum ending at each position.

When to Use:

- Finding maximum/minimum sum subarray
- Problems requiring contiguous elements with optimal value
- When you need O(n) solution for subarray sum problems

Key Insight: At each position, you have two choices:

1. Start a new subarray from current position
2. Extend the previous subarray by including current element

Implementation:

```python
def kadane(nums):
    max_so_far = float('-inf')
    max_ending_here = 0

    for num in nums:
        max_ending_here = max(num, max_ending_here + num)
        max_so_far = max(max_so_far, max_ending_here)

    return max_so_far
# Time: O(n), Space: O(1)
```

Sliding Window (Fixed Size)

> **Difficulty**: ⭐ Easy

The fixed-size sliding window algorithm maintains a subarray/substring of constant length `k` that "slides" through the data from left to right, updating results at each step.

When to Use:

- Computing running averages, sums, or statistics over a fixed-size range
- Finding subarrays/substrings of fixed length with certain properties

Implementation:

```python
def sliding_window(arr, k):
    # Initialize window and result
    window_sum = sum(arr[:k])
    max_sum = window_sum

    # Slide window from left to right
    for i in range(k, len(arr)):
        # Update window: add new element, remove oldest
        window_sum = window_sum + arr[i] - arr[i-k]
        # Update result
        max_sum = max(max_sum, window_sum)

    return max_sum
# Time: O(n), Space: O(1)
```

Sliding Window (Variable Size)

> **Difficulty**: ⭐⭐ Medium

Variable-size sliding window uses two pointers to define a window that expands and contracts based on certain conditions.

Implementation:

```python
def sliding_window_variable(arr, condition):
    left = 0
    result = initial_value
    current_state = {}  # or other tracking mechanism

    for right in range(len(arr)):
        # Expand window by including arr[right]
        # Update current_state
```

```
        # Contract window if needed
        while condition_to_shrink:
            # Remove arr[left] from window consideration
            # Update current_state
            left += 1

        # Update result based on current window

    return result
```

Two Pointers Technique

> **Difficulty**: ⭐ Easy

Uses two pointers to solve problems in linear time O(n) by eliminating nested loops.

Key Patterns:

- From both ends: left → | ← right
- Same direction: slow → fast →
- Fast/slow: slow → fast →→ (different speeds)

Implementation (from both ends):

```
def two_pointers_from_ends(arr):
    left, right = 0, len(arr) - 1

    while left < right:
        # Process elements at left and right

        if condition:
            left += 1
        else:
            right -= 1

    return result
```

Prefix Sums

> **Difficulty**: ⭐ Easy

Precompute cumulative sums to efficiently answer range queries. Transforms O(n) range sum operations into O(1).

Implementation:

```
def prefix_sum_setup(arr):
    n = len(arr)
    prefix = [0] * (n + 1)  # +1 for convenience

    for i in range(n):
        prefix[i+1] = prefix[i] + arr[i]

    return prefix

# Get sum of range [i,j] inclusive (0-indexed)
def range_sum(prefix, i, j):
    return prefix[j+1] - prefix[i]
```

Linked Lists

Fast & Slow Pointers

> **Difficulty**: ⭐⭐ Medium

Uses two pointers moving at different speeds to solve problems like finding cycles or the middle node.

Key Applications:

- Cycle detection
- Finding the middle element
- Finding the kth element from the end

Implementation (cycle detection):

```
def has_cycle(head):
    if not head or not head.next:
        return False

    slow = head
    fast = head.next

    while slow != fast:
        if not fast or not fast.next:
            return False
        slow = slow.next
        fast = fast.next.next
```

```
    return True
```

Iterative Linked List Reversal

> **Difficulty**: ⭐⭐ Medium

Reverses a linked list in-place using iteration.

Implementation:

```
def reverse_list(head):
    prev = None
    current = head

    while current:
        next_temp = current.next
        current.next = prev
        prev = current
        current = next_temp

    return prev
```

Trees & Advanced Data Structures

Trie Data Structure

> **Difficulty**: ⭐⭐ Medium

A tree-like data structure for efficient string operations and prefix matching.

Implementation:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = 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_of_word = 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_of_word

    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
```

Union-Find (Disjoint Set)

> **Difficulty**: ⭐⭐ Medium

Data structure for efficiently tracking connected components and performing union operations.

Implementation:

```
class UnionFind:
    def __init__(self, size):
        self.root = list(range(size))
        self.rank = [1] * size

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

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
```

```python
                self.root[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.root[rootX] = rootY
            else:
                self.root[rootY] = rootX
                self.rank[rootX] += 1

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

Segment Tree

> **Difficulty**: ⭐⭐⭐ Hard

Tree-based data structure for efficient range queries and updates.

Iterative DFS

> **Difficulty**: ⭐⭐ Medium

Non-recursive implementation of depth-first search using an explicit stack.

Implementation:

```python
def iterative_dfs(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()

        if node not in visited:
            visited.add(node)
            print(node)  # Process node

            # Add neighbors in reverse order for same traversal as recursive
            for neighbor in reversed(graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)

    return visited
```

Heaps

Two Heaps Algorithm

> **Difficulty**: ⭐⭐ Medium

Uses two heaps (min and max) to efficiently track medians and partition elements.

Implementation (median finding):

```
import heapq

class MedianFinder:
    def __init__(self):
        self.small = []  # max heap (negative values)
        self.large = []  # min heap

    def addNum(self, num):
        # By default, add to max heap
        heapq.heappush(self.small, -num)

        # Ensure max of small <= min of large
        if self.small and self.large and -self.small[0] > self.large[0]:
            heapq.heappush(self.large, -heapq.heappop(self.small))

        # Balance heaps
        if len(self.small) > len(self.large) + 1:
            heapq.heappush(self.large, -heapq.heappop(self.small))
        if len(self.large) > len(self.small) + 1:
            heapq.heappush(self.small, -heapq.heappop(self.large))

    def findMedian(self):
        if len(self.small) > len(self.large):
            return -self.small[0]
        if len(self.large) > len(self.small):
            return self.large[0]
        return (-self.small[0] + self.large[0]) / 2
```

Backtracking

Subsets Algorithm

> **Difficulty**: ⭐⭐ Medium

Generates all possible subsets of a given set.

Implementation:

```python
def subsets(nums):
    result = []

    def backtrack(start, current):
        result.append(current[:])

        for i in range(start, len(nums)):
            current.append(nums[i])
            backtrack(i + 1, current)
            current.pop()

    backtrack(0, [])
    return result
```

Combinations Algorithm

> **Difficulty**: ⭐⭐ Medium

Generates all possible k-sized combinations from n elements.

Implementation:

```python
def combine(n, k):
    result = []

    def backtrack(start, current):
        if len(current) == k:
            result.append(current[:])
            return

        for i in range(start, n + 1):
            current.append(i)
            backtrack(i + 1, current)
            current.pop()

    backtrack(1, [])
    return result
```

Permutations Algorithm

> **Difficulty**: ⭐⭐ Medium

Generates all possible arrangements of a set of elements.

Implementation:

```
def permute(nums):
    result = []

    def backtrack(current):
        if len(current) == len(nums):
            result.append(current[:])
            return

        for num in nums:
            if num not in current:
                current.append(num)
                backtrack(current)
                current.pop()

    backtrack([])
    return result
```

Graphs

Dijkstra's Algorithm

> **Difficulty**: ⭐⭐⭐ Hard

Finds shortest paths from a source vertex to all other vertices in a weighted graph.

Implementation:

```
import heapq

def dijkstra(graph, start):
    # Initialize distances with infinity
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # Skip if we've found a better path
        if current_distance > distances[current_node]:
            continue

        # Check neighbors
        for neighbor, weight in graph[current_node].items():
```

```
            distance = current_distance + weight

            # Update if we found a better path
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances
```

Prim's Algorithm

> **Difficulty**: ⭐⭐⭐ Hard

Finds a minimum spanning tree using a greedy approach.

Kruskal's Algorithm

> **Difficulty**: ⭐⭐⭐ Hard

Finds a minimum spanning tree using a disjoint set (Union-Find).

Topological Sort

> **Difficulty**: ⭐⭐ Medium

Orders vertices in a directed acyclic graph such that for every edge (u, v), vertex u comes before v.

Implementation:

```
def topological_sort(graph):
    # Graph: adjacency list where graph[node] = list of neighbors

    # Initialize in-degree for all nodes
    in_degree = {node: 0 for node in graph}
    for node in graph:
        for neighbor in graph[node]:
            in_degree[neighbor] += 1

    # Start with nodes that have no dependencies
    queue = [node for node in graph if in_degree[node] == 0]
    result = []

    while queue:
```

```
        node = queue.pop(0)
        result.append(node)

        # Remove edges from this node
        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # Check for cycles
    if len(result) != len(graph):
        return []  # Graph has a cycle

    return result
```

Dynamic Programming

0/1 Knapsack

> **Difficulty**: ⭐⭐ Medium

Solves the problem of selecting items with weight constraints where each item can be used at most once.

Implementation:

```
def knapsack_01(weights, values, capacity):
    n = len(weights)
    dp = [[0 for _ in range(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]
```

Unbounded Knapsack

> **Difficulty**: ⭐⭐ Medium

Similar to 0/1 knapsack but allows using items unlimited times.

Longest Common Subsequence

> **Difficulty**: ⭐⭐ Medium

Finds the longest subsequence present in two sequences.

Implementation:

```
def longest_common_subsequence(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i-1] == text2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]
```

Palindrome Problems

> **Difficulty**: ⭐⭐ Medium

Algorithms for identifying and working with palindromic substrings and subsequences.

Additional Patterns

Binary Search Variations

> **Difficulty**: ⭐⭐ Medium

Modified binary search approaches for complex scenarios.

Implementation:

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

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

    return -1  # Not found
```

Monotonic Stack/Queue

> **Difficulty**: ⭐⭐ Medium

Data structures for efficiently solving next greater/smaller element problems.

BFS for Shortest Path

> **Difficulty**: ⭐⭐ Medium

Uses breadth-first search to find shortest paths in unweighted graphs.

Implementation:

```python
from collections import deque

def bfs_shortest_path(graph, start, end):
    if start == end:
        return [start]

    visited = {start}
    queue = deque([(start, [start])])

    while queue:
        node, path = queue.popleft()

        for neighbor in graph[node]:
            if neighbor not in visited:
                if neighbor == end:
                    return path + [neighbor]
```

```
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))

    return []  # No path exists
```

Bit Manipulation

> **Difficulty**: ⭐⭐ Medium

Techniques using bitwise operations for optimizations.

Dutch National Flag

> **Difficulty**: ⭐⭐ Medium

Three-way partitioning algorithm for sorting arrays with three distinct values.

Boyer-Moore Voting Algorithm

> **Difficulty**: ⭐ Easy

Efficiently finds the majority element in an array.

Implementation:

```python
def majority_element(nums):
    count = 0
    candidate = None

    for num in nums:
        if count == 0:
            candidate = num
        count += (1 if num == candidate else -1)

    return candidate
```

Floyd's Cycle Finding

> **Difficulty**: ⭐⭐ Medium

Algorithm for detecting cycles in sequences.

Learning Plan

A structured 3-week algorithm learning acceleration plan:

Week 1: Fundamentals

- Day 1-2: Arrays & Strings (Two Pointers, Sliding Window)
- Day 3-4: Linked Lists (Fast & Slow Pointers, Reversal)
- Day 5-7: Stacks, Queues, and Basic Trees (BFS, DFS)

Week 2: Intermediate Techniques

- Day 8-10: Binary Search & Divide and Conquer
- Day 11-12: Backtracking & Recursion
- Day 13-14: Dynamic Programming Fundamentals

Week 3: Advanced Patterns

- Day 15-16: Graphs & Network Flow
- Day 17-18: Advanced Dynamic Programming
- Day 19-21: System Design & Complex Problem Patterns

Glossary

A brief glossary of common algorithm and data structure terms:

- **Asymptotic Notation**: Mathematical notation to describe algorithm efficiency
- **BFS**: Breadth-First Search, a graph traversal algorithm
- **Binary Search**: Divide and conquer search algorithm for sorted arrays
- **DFS**: Depth-First Search, a graph traversal algorithm
- **Dynamic Programming**: Breaking complex problems into simpler overlapping subproblems

- **Greedy Algorithm**: Making locally optimal choices at each stage
- **Hash Table**: Data structure that maps keys to values using a hash function
- **Memoization**: Optimization technique storing results of expensive function calls
- **Recursion**: Function that calls itself to solve smaller instances of the same problem
- **Time Complexity**: Measurement of algorithm efficiency relative to input size

Learning Activities

Interactive learning activities to reinforce algorithm knowledge:

- Algorithm Flash Cards
- Template Skeleton Exercises
- Algorithm Decision Tree
- Time Attack Implementation Challenges
- Complexity Analysis Quizzes
- Pattern Matching Games
- Memory Optimization Challenges

This document combines key algorithms and data structures from the repository. For detailed implementations, variations, and practice problems, refer to individual algorithm files.