ALGORITHMS & DATA STRUCTURES

THE ULTIMATE REFERENCE CHEATSHEET

Sorting Algorithms Complexity

Algorithm	Best	Average	Worst	Space
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$
Mergesort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$
Bubble Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Radix Sort	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(n+k)$

Searching Algorithms

Linear Search: Check each element. $\mathcal{O}(n)$ time.

Binary Search: Half-interval search on sorted arrays. $\mathcal{O}(\log n)$ time, $\mathcal{O}(1)$ space.

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

Breadth-First Search (BFS)

Traverses level-by-level using a FIFO queue. Finds shortest path in unweighted graphs. Time: $\mathcal{O}(V+E)$, Space: $\mathcal{O}(V)$.

```
from collections import deque

def bfs(graph, start):
    visited = {start}
    queue = deque([start])
    while queue:
        node = queue.popleft()
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

Data Structure Complexity

Structure	Access	Search	Insert	Delete
Array / Vector	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Singly-Linked	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Doubly-Linked	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Stack / Queue	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Hash Table	–	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
BST (Avg)	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
BST (Worst)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
AVL / Red-Black	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

*Note: Hash Table operations are $\mathcal{O}(n)$ worst-case due to collisions. BST degrades to $\mathcal{O}(n)$ if unbalanced.

Quicksort (Lomuto Partition)

Partition-based divide-and-conquer algorithm. Average-case optimal, but degrades to $\mathcal{O}(n^2)$ on sorted arrays with poor pivot choices.

```
def quicksort(arr, low, high):
    if low < high:
        p = partition(arr, low, high)
        quicksort(arr, low, p - 1)
        quicksort(arr, p + 1, high)

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i + 1
```

Core Data Structures Notes

Singly-Linked List: Nodes store value and `next` pointer. Efficient insertion/deletion at head ($\mathcal{O}(1)$), but linear lookup.

Doubly-Linked List: Nodes store `next` and `prev` pointers. Allows bidirectional traversal; $\mathcal{O}(1)$ deletion when node reference is given.

Hash Table: Key-value store using a hash function. Collisions resolved via Chaining (linked lists) or Open Addressing (probing).

Binary Search Tree (BST): Left child $<$ parent $\leq$ right child. Inorder traversal yields sorted elements.

AVL Tree: Self-balancing BST where heights of two child subtrees differ by at most one. Uses rotations to rebalance.

Depth-First Search (DFS)

Traverses deep along paths using a recursion stack. Used for topological sorting and cycle detection. Time: $\mathcal{O}(V+E)$, Space: $\mathcal{O}(V)$.

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

Stack, Queue & Heaps

Stack: LIFO (Last-In-First-Out). Operations: Push/Pop are $\mathcal{O}(1)$. Used in function calls, backtracking.

Queue: FIFO (First-In-First-Out). Operations: Enqueue/Dequeue are $\mathcal{O}(1)$. Used in BFS, scheduling.

Min-Heap / Max-Heap: Complete binary tree where parent is always smaller/larger than children. Insertion/Extraction are $\mathcal{O}(\log n)$, Peek is $\mathcal{O}(1)$.

Advanced Graph Algorithms

Dijkstra's Algorithm: Finds shortest path from single source in non-negative weighted graph. Uses Min-Priority Queue. Time: $\mathcal{O}((V+E)\log V)$.

Bellman-Ford: Finds shortest path, allows negative edge weights and detects negative cycles. Time: $\mathcal{O}(VE)$.

Prim's & Kruskal's: Finds Minimum Spanning Tree (MST). Kruskal's uses Union-Find ($\mathcal{O}(E \log V)$); Prim's grows tree from vertex ($\mathcal{O}(E \log V)$).

Dynamic Programming Principles

Solves complex problems by breaking them into overlapping subproblems, storing results (memoization/tabulation) to avoid recomputation.

0/1 Knapsack recurrence relation:

$$DP[i][w] = \max(DP[i-1][w], DP[i-1][w-w_i] + v_i)$$

```
# 0/1 Knapsack - Space Optimized Tabulation
def knapsack(W, weights, values, n):
    dp = [0] * (W + 1)
    for i in range(n):
        for w in range(W, weights[i] - 1, -1):
            dp[w] = max(dp[w], dp[w - weights[i]] +
                values[i])
    return dp[W]
```

AVL Tree Rotations

AVL trees maintain a balance factor $|h_L - h_R| \leq 1$. Restores balance using local transformations (rotations) in $\mathcal{O}(1)$ time.

```
def right_rotate(y):
    x = y.left
    T2 = x.right
    # Perform rotation
    x.right = y
    y.left = T2
    # Update heights
    y.height = 1 + max(height(y.left), height(y.
        right))
    x.height = 1 + max(height(x.left), height(x.
        right))
    return x
```

Four Rotation Cases:

Left-Left (LL): Single Right Rotation.

Right-Right (RR): Single Left Rotation.

Left-Right (LR): Left Rotate child, then Right Rotate parent.

Right-Left (RL): Right Rotate child, then Left Rotate parent.

Bit Manipulation Tricks

Highly efficient operations performed directly on integer bits.

Operation	Bitwise Expression
Set i-th bit	`x | (1 << i)`
Clear i-th bit	`x & ~(1 << i)`
Toggle i-th bit	`x ^ (1 << i)`
Check i-th bit	`(x & (1 << i)) != 0`
Clear lowest set bit	`x & (x - 1)`
Get lowest set bit	`x & (-x)`