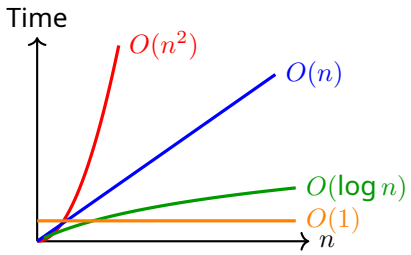
Data Structures & Algorithms Cheat Sheet

Complexity, Structures, Sorting & Searching

1 Complexity Analysis (Big O)

Concept: Measures how runtime or space requirements grow as input size (n) increases.



Common Complexities (Best to Worst)

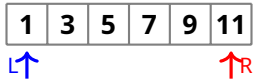
- $O(1)$: Constant (Hash Map lookup, Array index).
- $O(\log n)$: Logarithmic (Binary Search).
- $O(n)$: Linear (Iteration, Linear Search).
- $O(n \log n)$: Linearithmic (Merge Sort, Quick Sort).
- $O(n^2)$: Quadratic (Nested loops, Bubble Sort).
- $O(2^n)$: Exponential (Recursive Fibonacci).

2 Arrays & Strings

Properties

- Contiguous memory.
- Access: $O(1)$. Insert/Delete: $O(n)$ (shifting).

Two Pointer Technique *Efficiently iterate sorted arrays (e.g., Two Sum).*

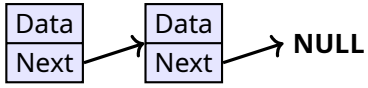



```
def two_sum(arr, target):
    left, right = 0, len(arr) - 1
    while left < right:
        curr = arr[left] + arr[right]
        if curr == target: return True
        elif curr < target: left += 1
        else: right -= 1
    return False
```

3 Linked Lists

Structure: Nodes with value and pointer(s). Non-contiguous memory.

- **Singly**: `Head -> Next -> NULL`
- **Doubly**: `NULL <- Prev <-> Next -> NULL`



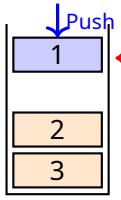
Complexity

- Access: $O(n)$ (Traversal).
- Insert/Delete (Head): $O(1)$.
- Insert/Delete (Middle): $O(1)$ (if pointer known), else $O(n)$.

Reverse Linked List

```
def reverse(head):
    prev = None
    curr = head
    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node
    return prev
```

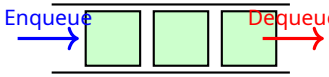
4 Stacks & Queues



Stack (LIFO - Last In First Out)

- Operations: `push()`, `pop()`, `peek()`.
- Use cases: Recursion, Undo, DFS.

Queue (FIFO - First In First Out)



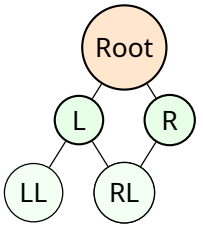
- Operations: `enqueue()`, `dequeue()`.
- Use cases: Task scheduling, BFS.

```
from collections import deque
q = deque()
q.append(1)   # Enqueue
val = q.popleft() # Dequeue -> 1
```

5 Trees

Binary Tree: Each node has ≤ 2 children. **BST (Binary Search Tree)**: Left < Node < Right.

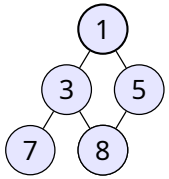
- Search/Insert/Delete: $O(\log n)$ (Balanced), $O(n)$ (Skewed).



Traversals (DFS)

- **In-Order**: Left → Root → Right (Sorted in BST).
- **Pre-Order**: Root → Left → Right.
- **Post-Order**: Left → Right → Root.

Heaps (Min-Heap) *Root is minimum. Complete Binary Tree.*

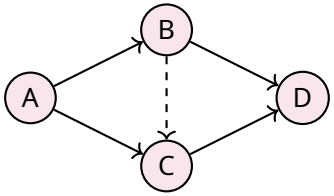


Min Heap

6 Graphs

Representation

- **Adjacency Matrix**: $V \times V$ grid. Space $O(V^2)$. Fast check.
- **Adjacency List**: Dictionary of lists. Space $O(V+E)$. Sparse efficient.



Algorithms

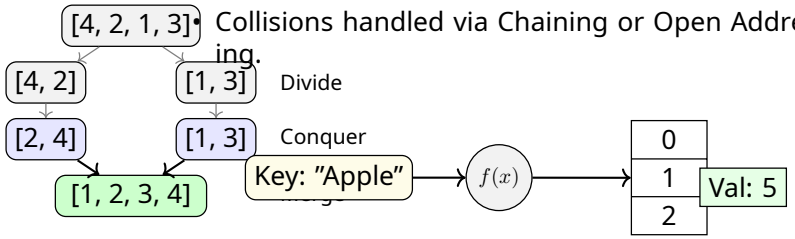
- **BFS (Shortest Path, Unweighted)**: Use Queue.
- **DFS (Path Finding, Cycles)**: Use Stack/Recursion.
- **Dijkstra**: Shortest path in weighted graph (Priority Queue).

```
def dfs(graph, node, visited):
    if node not in visited:
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)
```

7 Sorting Algorithms

	Algo	Time (Avg)	Space
Comparison	Bubble	$O(n^2)$	$O(1)$
	Merge	$O(n \log n)$	$O(n)$
	Quick	$O(n \log n)$	$O(\log n)$
	Heap	$O(n \log n)$	$O(1)$

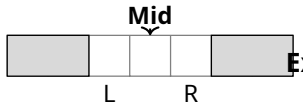
Merge Sort (Divide & Conquer)




```
def quicksort(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 quicksort(left) + middle + quicksort(right)
```

8 Searching & Hashing

Binary Search (Sorted Array) *Repeatedly divide search interval in half.*




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

Hash Table (Dictionary)

- Key-Value pairs using a hash function.
- Access/Insert/Delete: $O(1)$ average.
- Collisions handled via Chaining or Open Addressing.

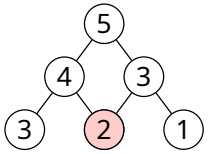
9 Dynamic Programming

Solving complex problems by breaking them into simpler subproblems and storing results (Memoization).

Key Properties

1. **Overlapping Subproblems**: Recomputing same states.
2. **Optimal Substructure**: Optimal solution from sub-solutions.

Example: Fibonacci




```
# Memoization (Top-Down)
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n <= 1: return n
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```