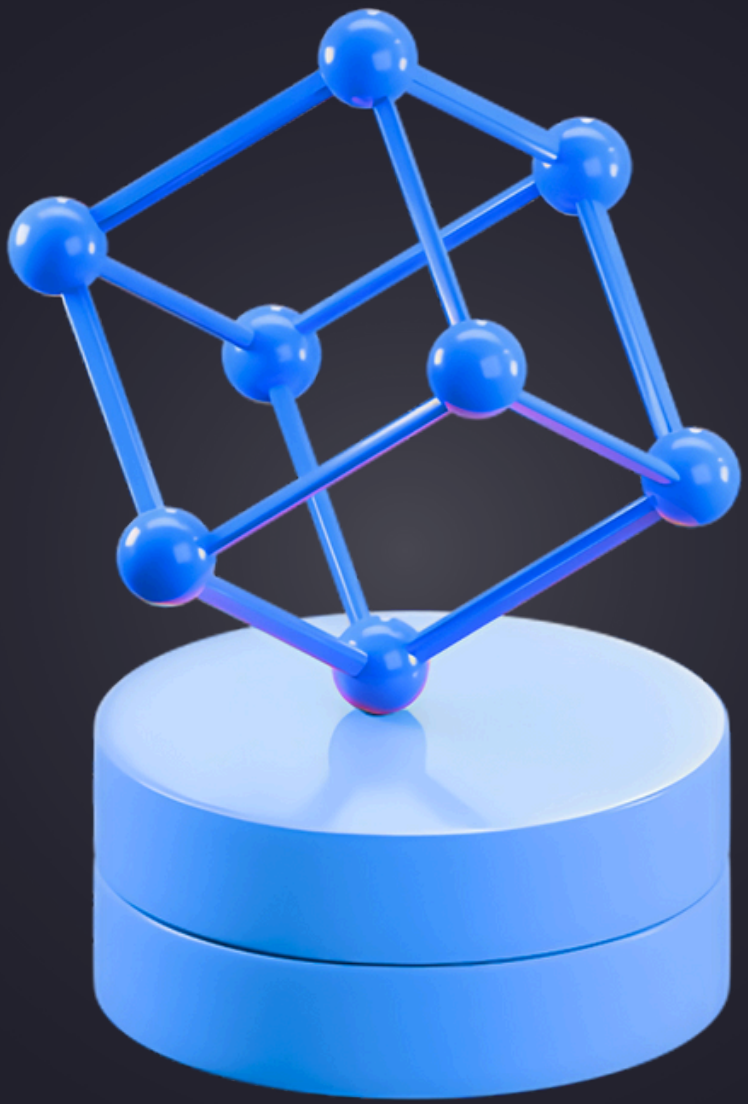
DSA CheatSheet



1. INTRODUCTION TO DSA

1.1 What is DSA?

- *DSA* = Data Structures + Algorithms.
- Data Structures are like containers to hold data. Imagine your data is like fruits, and data structures are baskets or boxes to keep those fruits organized.
- Examples of data structures: Arrays, Linked Lists, Stacks, Queues, Trees, etc.
- Algorithms are the step-by-step instructions or recipes you follow to solve a problem using data structures.
- Example: To find a fruit in your basket, you might look through it one by one or sort the fruits first to find it quickly. That's an algorithm!
- *Why use DSA?* Because it helps computers solve problems faster and use less memory.

1.2 Why is DSA important in programming and interviews?

- When companies hire programmers, they want to see if you can solve problems efficiently.
- Knowing DSA helps you solve tricky problems quickly.
- Many tech interviews ask questions based on DSA, so it's a must-know skill.
- Good knowledge of DSA also helps you build better apps and games that run faster.

1.3 Time and Space Complexity Basics

- When you write a program, you want it to run fast (time) and use less memory (space).
- Time complexity tells us how the running time of a program changes when the input size grows.
- Space complexity tells us how much extra memory the program uses.
- For example, if your program takes 1 second for 10 inputs, it might take 10 seconds for 100 inputs (linear growth, called O(n)).

1.4 Big O Notation Explained

- Big O notation is a way to describe how fast or slow an algorithm is.
- It tells us the worst-case scenario — the longest time or most space your program might need.

Examples:

- O(1): Instant! The program takes the same time no matter how big input is.
- O(n): Time grows linearly as input grows.
- $O(n^2)$: Time grows much faster as input grows (like nested loops).
- O(log n): Very efficient, grows slowly (like binary search).

2. MATHEMATICS FOR DSA

2.1 Prime Numbers

- Prime numbers are numbers greater than 1 that only divide evenly by 1 and themselves.
- Examples: 2, 3, 5, 7, 11
- Primes are important in encryption and algorithms.
- To check if a number is prime, try dividing it by every number from 2 up to its square root. If any divides evenly, it's not prime.

2.2 GCD and LCM (Euclidean Algorithm)

- GCD (Greatest Common Divisor): The biggest number that divides two numbers without remainder.
- LCM (Least Common Multiple): The smallest number divisible by two numbers.
- Euclidean Algorithm is a fast way to find GCD:
  - Keep dividing the bigger number by the smaller one, replace numbers, until remainder is 0. The last non-zero remainder is the GCD.
- LCM can be found using GCD by formula:

```Mathematica
LCM(a, b) = (a * b) / GCD(a, b)
```

2.3 Modular Arithmetic

- Imagine a clock — after 12 hours, it resets to 1.
- Modular arithmetic works the same way, "wrapping around" numbers.
- Useful to keep numbers small in big calculations (like 1000000007 in coding contests).
- Example:

```Lua
(8 + 7) mod 5 = 15 mod 5 = 0
```

2.4 Sieve of Eratosthenes

- A method to find all prime numbers up to a number n quickly.
- How it works:
  - Start with a list of all numbers marked as prime.
  - Start with 2, remove all its multiples.
  - Move to the next number still marked prime and remove its multiples.
  - Continue until you reach the square root of n.
- At the end, the numbers still marked prime are the primes.

2. MATHEMATICS FOR DSA

2.5 Fast Exponentiation

- If you want to calculate a^b (a to the power b), doing a * a * a... b times takes a long time.
- Fast exponentiation reduces the steps by using:
  - If b is even, a^b = (a^(b/2)) * (a^(b/2))
  - If b is odd, a^b = a * (a^(b-1))
- This method uses divide and conquer to calculate power in O(log b) steps.

2.6 Bit Manipulation Basics

- Computers work in bits (0s and 1s).
- Bit manipulation means changing or checking individual bits for quick math or logic.
- Important bit operations:
  - AND (&): Both bits 1 → 1, else 0
  - OR (|): At least one bit 1 → 1
  - XOR (^): Bits different → 1, else
  - 0 NOT (~): Flip bits (0 → 1, 1 →
  - 0) Left shift (<<): Multiply by 2
  - Right shift (>>): Divide by 2
- Example: Check if a number is even or odd by num & 1 (1 means odd).

3. ARRAYS

3.1 Introduction to Arrays

- An array is a collection of items stored next to each other in memory.
- You can access each item using its index (starting from 0).
- Example: [10, 20, 30, 40], arr[2] is 30.

3.2 Traversal, Insertion, Deletion

- Traversal: Going through each element one by one.
- Insertion: Adding an element at a position.
- Deletion: Removing an element from a position.
- In arrays, insertion and deletion in the middle need shifting elements, which takes time (O(n)).

3.3 Prefix Sum & Sliding Window

- Prefix Sum: Pre-calculated sums from the start up to each position.
  - Helps find sum of any subarray quickly.
  - Example: If prefix sums are [2, 5, 9, 14], sum from index 1 to 3 is 14 - 2 = 12.
- Sliding Window: Technique to find results over a continuous window of size k.
  - Instead of summing all elements each time, update by removing left element and adding right element.

3.4 Two Pointer Technique

- Use two pointers (indexes) moving through the array.
- Useful for sorted arrays to find pairs or subarrays.
- Move pointers based on condition to reach solution faster.

3.5 Kadane's Algorithm (Max Subarray Sum)

- Find the contiguous subarray with the maximum sum.
- Keep track of current subarray sum and max sum so far.
- If current sum becomes negative, reset it to zero (start a new subarray).

3.6 Sorting Techniques

- Sorting means arranging numbers in order (ascending or descending).
- Some simple sorting methods:
  - Bubble Sort: Compare neighbors, swap if out of order. Repeat.
  - Selection Sort: Find smallest item and place it at the start.
  - Insertion Sort: Build sorted array by inserting one element at a time.
- Faster, advanced sorting methods:
  - Merge Sort: Divide array into halves, sort each half, merge them.
  - Quick Sort: Pick a pivot, partition array around pivot, sort partitions.

4. STRINGS

4.1 String Basics and Operations

- A string is a sequence of characters, like "hello" or "123abc".
- Strings are stored as arrays of characters.
- Basic operations:
  - Access characters by index (str[0] = 'h')
  - Concatenate strings ("hello" + " world" = "hello world")
  - Length gives number of characters
  - Substring extracts part of a string

4.2 Palindrome Check

- A palindrome reads the same forwards and backwards.
- Example: "madam", "racecar"
- How to check?
  - Compare characters from start and end moving towards center.
  - If all pairs match, it's a palindrome.

4.3 Anagram Check

- Two strings are anagrams if they have the same letters in any order.
- Example: "listen" and "silent"
- How to check?
  - Sort both strings and compare.
  - Or count character frequency and compare counts.

4.4 String Matching Algorithms

- These algorithms find if a pattern exists inside a text.
- Naive algorithm:
  - Check every position in text if pattern matches.
  - Simple but slow for big data (O(n*m)).
- KMP (Knuth-Morris-Pratt):
  - Uses a prefix table to avoid repeating comparisons.
  - Faster (O(n + m)).
- Rabin-Karp:
  - Uses hashing to find matches quickly.
  - Good for multiple pattern searching.

4.5 Pattern Matching

- Finding occurrences of a pattern in text.
- Uses above algorithms to efficiently locate patterns.
- Useful in search engines, plagiarism detection, and DNA sequencing.

4. STRINGS

4.6 Z-Algorithm

- Calculates an array (Z-array) which tells how many characters from a position match the prefix of the string.
- Helps in pattern searching and string problems.
- Runs in O(n) time.

4.7 Manacher's Algorithm (Advanced)

- Finds the longest palindromic substring in a string efficiently.
- Runs in O(n) time (much faster than checking all substrings).
- Complex but useful in advanced string problems.

5. LINKED LISTS

5.1 Singly Linked List

- A linked list is a chain of nodes.
- Each node has data and a pointer to the next node.
- Singly linked list means each node points only to the next node.

5.2 Doubly Linked List

- Each node has a pointer to the next and previous nodes.
- Can move forward and backward easily.

5.3 Circular Linked List

- The last node points back to the head node.
- Can be singly or doubly linked.
- Useful for applications like round-robin scheduling.

5.4 Operations: Insert, Delete, Reverse

- Insert: Add node at beginning, end, or middle.
- Delete: Remove a node by value or position.
- Reverse: Change pointers so list order reverses.

5.5 Detect Cycle (Floyd's Algorithm)

- Sometimes linked list has a cycle (loop).
- Floyd's Cycle Detection (Tortoise and Hare):
  - Use two pointers moving at different speeds.
  - If they meet, cycle exists.

5.6 Intersection Point, Merge Two Lists

- Intersection point: Where two linked lists merge.
- Find by comparing nodes or using difference in lengths.
- Merge two sorted lists: Combine into one sorted list by comparing nodes one by one.

6. STACKS

6.1 Stack Basics

- Stack is a Last In, First Out (LIFO) data structure.
- Imagine a stack of books: the last book you put on top is the first one you take off.

6.2 Push/Pop Operations

- Push: Add element on top of stack.
- Pop: Remove element from top of stack.
- Both operations run in O(1) time.

6.3 Infix to Postfix/Prefix Conversion

- Infix expression: Operators between operands (e.g. A + B).
- Postfix expression: Operators after operands (e.g. A B +).
- Prefix expression: Operators before operands (e.g. + A B).

Why convert?

- Easier to evaluate postfix or prefix expressions programmatically.

How stacks help?

- Use stack to hold operators and decide order based on precedence.

6.4 Valid Parentheses

- Check if every opening bracket has a matching closing bracket in correct order.
- How to check?
  - Traverse string.
  - Push opening brackets onto stack.
  - When a closing bracket found, pop stack and check if it matches the type.

6.5 Next Greater Element

- For each element in array, find the next element to the right that is greater.
- How to do efficiently?
  - Use a stack to keep track of elements for which we need to find the next greater element.
  - Traverse from right to left, popping smaller elements until a greater one is found.

6.6 Min Stack

- A stack that supports returning the minimum element in O(1) time.
- How?
  - Use an auxiliary stack to store minimum values.
  - When pushing, also update minimum stack.
  - When popping, pop from minimum stack if top matches.

7. QUEUES

7.1 Queue Basics

Definition:

A Queue is a linear data structure that follows the First In First Out (FIFO) principle.

Analogy:

Like people standing in a line — the person who enters first gets served first.

Operations:

- Enqueue: Add element at rear.
- Dequeue: Remove element from front.
- Peek/Front: See the front element.
- isEmpty: Check if the queue is empty.

- Use Cases:
- Print queues, task scheduling, real-time systems.

7.2 Circular Queue

- Problem in normal queue:
- After many dequeues, front moves forward and space at the start is wasted.
- Solution:
- Circular Queue wraps around and uses space efficiently.
- Circular behavior:
- When rear reaches the end, it goes to index 0 if space is available.
- Implemented using: Arrays with modular arithmetic:

7.3 Deque (Double Ended Queue)

- Definition:
- A Deque allows insertion and deletion from both front and rear ends.
- Operations:
  - pushFront, pushRear
  - popFront, popRear
- Use Cases:
- Palindrome checker, sliding window problems, browser history.

7. QUEUES

7.4 Priority Queue / Heap

- Priority Queue:
- Elements are served based on priority instead of order.
- Implementation:
- Usually done using a Heap.
- Types:
  - Min-Heap: Smallest element at top
  - Max-Heap: Largest element at top
- Use Cases:
  - CPU scheduling
  - Dijkstra's algorithm (shortest path)
  - Top K frequent elements

7.5 Stack using Queue and vice versa

- Stack using Queue:
- Use 2 queues. For push, enqueue in one. For pop, dequeue all except last and swap.
- Queue using Stack:
- Use 2 stacks. For enqueue, push normally. For dequeue, reverse the stack using the second one.

8. RECURSION AND BACKTRACKING

8.1 Recursion Basics

- What is recursion?
- A function calling itself to solve smaller subproblems.

Every recursive function has:

- Base case: Condition to stop recursion
- Recursive case: Calls itself with smaller input

8.2 Factorial, Fibonacci

- Factorial(n): n! = n × (n-1)!

Example:

Fibonacci(n): f(n) = f(n-1) + f(n-2)

- Use memoization or DP to avoid recomputation.

8.3 Tower of Hanoi

- Problem:
- Move n disks from source to destination using helper rod.
- Rules:
  - Move only one disk at a time
  - Larger disk cannot be placed on a smaller one
- Recursive logic:
  a. Move n-1 disks to helper
  b. Move nth disk to destination
  c. Move n-1 disks from helper to destination

8.4 N-Queens Problem

Goal:

- Place N queens on N×N board so that no two queens attack each other.

Approach:

- Backtracking — try placing a queen in each row and check for safety.

Key function:

- isSafe(row, col) checks if it's safe to place queen at that position.

8. RECURSION AND BACKTRACKING

8.5 Sudoku Solver

Problem:

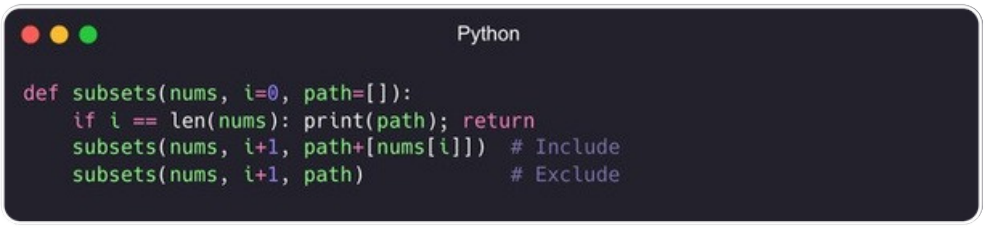
- Fill a 9x9 board so every row, column, and 3x3 box has numbers 1 to 9 without repetition.

Approach:

- Backtracking
  - Try numbers from 1–9 at each empty cell
  - If valid, move to next cell
  - If not valid, backtrack

8.6 Subset / Permutation / Combination Generation

- Subset (Power Set):
- Include or exclude each element.



Permutations:

- Swap each element and recurse.

Combinations:

- Choose k elements from n without caring about order.

9. SEARCHING ALGORITHMS

9.1 Linear Search

Definition:

- Linear Search is the most basic searching technique. It checks each element of the array one by one until the target element is found or the end is reached.

Steps:

- Start from the first element.
- Compare it with the target.
- If it matches, return the index.
- If not, move to the next element.
- If you reach the end without finding the target, return -1.

Python Example:

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

Time Complexity:

- Best Case: O(1) (if target is at the start)
- Worst Case: O(n) (if target is at the end or not found)

Use Case:

- When the array is unsorted or small in size.

9.2 Binary Search

Definition:

- Binary Search is an efficient algorithm that works on sorted arrays by repeatedly dividing the search interval in half.

Steps:

1.Find the middle index.
2.If the middle element equals the target, return the index.
3.If the target is smaller, search in the left half. 4.If the target is larger, search in the right half. 5.Repeat the process until the search space is empty.

Python Example:

```python
def binary_search(arr, target):
    low = 0
    high = 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
```

9. SEARCHING ALGORITHMS

9.2 Binary Search

Time Complexity:

- Always: O(log n)

Use Case:

- Only when the array is sorted.

9.3 Binary Search on Answer

Definition:

- Used when you're not searching for a specific value in an array, but trying to optimize or find a boundary/limit (e.g., smallest value that satisfies a condition).

Typical Use Cases:

- Minimum number of pages a student can read per day (Book Allocation Problem).
- Minimum time to complete tasks under certain constraints.

Steps:

1. Define the search space (e.g., min and max possible answers).
2. Apply binary search in that range.
3. Use a helper function to check if a mid value is a valid solution.
4. Narrow the search space based on the result.

Example:

- Finding the minimum number such that a condition becomes true.

9.4 Search in Rotated Sorted Array

Definition:

- This is a modified binary search problem where the array is sorted but rotated at some pivot.

Example:

- [6, 7, 8, 1, 2, 3, 4, 5] — a sorted array rotated at index 3.

Steps:

1. Use binary search.
2. Check if the left half or right half is sorted.
3. Decide where to continue searching based on which part is sorted and where the target may lie.

9. SEARCHING ALGORITHMS

9.4 Search in Rotated Sorted Array

Python Example:

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

- Time Complexity: O(log n)

9.5 Lower Bound / Upper Bound

Lower Bound:

- The first index where the element is greater than or equal to the target.
- Useful in sorted arrays.
- Can be found using binary search.

Upper Bound:

- The first index where the element is strictly greater than the target.

Python Example (using bisect module):

```python
import bisect

arr = [1, 2, 4, 4, 5, 6]

# Lower bound of 4 (first position where 4 or more starts)
print(bisect.bisect_left(arr, 4))  # Output: 2

# Upper bound of 4 (first position where value is > 4)
print(bisect.bisect_right(arr, 4))  # Output: 4
```

- Time Complexity: O(log n)

Use Case:

- Binary search-related problems.
- Frequency count, range queries, etc.

10. SORTING ALGORITHMS

Sorting is the process of arranging elements (numbers, strings, etc.) in a particular order — ascending or descending.

10.1 Bubble Sort

- Idea: Repeatedly swap adjacent elements if they're in the wrong order.

Steps:

1. Compare 1st and 2nd → swap if needed.
2. Move to next pair.
3. After 1st pass → largest element goes to the end.
4. Repeat until array is sorted.

Python Code:

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Time:

- Best: O(n) (if already sorted)
- Worst: $O(n^2)$

10.2 Selection Sort

- Idea: Select the minimum element and place it at the beginning.

Steps:

1. Find smallest from unsorted part.
2. Swap it with the first unsorted element.
3. Repeat.

Python Code:

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

Time: Always $O(n^2)$

10. SORTING ALGORITHMS

10.3 Insertion Sort

- Idea: Pick one element and place it in the correct position of the sorted par t .

Steps:

1. Start from 2nd element.
2. Compare it with previous elements.
3. Shift elements and insert at correct spot.

Python Code:

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Time:

- Best: O(n)
- Worst: $O(n^2)$

10.4 Merge Sort

- Idea: Divide array into halves, sort each half, then merge them.

Steps:

1. Divide until single-element arrays.
2. Merge two sorted arrays.

Python Code:

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1; k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1; k += 1
```

Time: Always O(n log n)

Space: O(n)

10. SORTING ALGORITHMS

10.5 Quick Sort

- Idea: Pick a pivot, place elements smaller on left, greater on right → recursively sort.

Python Code:

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = [x for x in arr[1:] if x <= pivot]
    right = [x for x in arr[1:] if x > pivot]
    return quick_sort(left) + [pivot] + quick_sort(right)
```

Time:

- Best: O(n log n)
- Worst: $O(n^2)$ (if badly chosen pivot)

10.6 Heap Sort

- Idea: Use a heap to extract max/min repeatedly and sort.

Python Code:

```python
import heapq

def heap_sort(arr):
    heapq.heapify(arr)
    return [heapq.heappop(arr) for _ in range(len(arr))]
```

Time: O(n log n)
Space: O(1) if using inplace heap

10.7 Counting Sort

- Idea: Count occurrences and use that to place elements in sorted order. Only for non-negative integers.

Python Code:

```python
def counting_sort(arr):
    max_val = max(arr)
    count = [0] * (max_val + 1)
    for num in arr:
        count[num] += 1
    i = 0
    for num in range(len(count)):
        for _ in range(count[num]):
            arr[i] = num
            i += 1
```

Time: O(n + k)
(k = max element)

10. SORTING ALGORITHMS

10.8 Radix Sort

- Idea: Sort numbers digit-by-digit using Counting Sort as a subroutine. Only for integers.

Python Code:

```python
def counting_sort_exp(arr, exp):
    n = len(arr)
    output = [0]*n
    count = [0]*10
    for i in arr:
        index = (i // exp) % 10
        count[index] += 1
    for i in range(1, 10):
        count[i] += count[i-1]
    i = n - 1
    while i >= 0:
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
        count[index] -= 1
        i -= 1
    for i in range(len(arr)):
        arr[i] = output[i]

def radix_sort(arr):
    max_val = max(arr)
    exp = 1
    while max_val // exp > 0:
        counting_sort_exp(arr, exp)
        exp *= 10
```

Time: O(nk) (k = number of digits)

10.9 Bucket Sort

- Idea: Distribute elements into buckets, sort each bucket, then combine. Best for uniformly distributed floats (e.g., [0.2, 0.5, 0.8]).

Python Code:

```python
def bucket_sort(arr):
    buckets = [[] for _ in range(10)]
    for num in arr:
        index = int(num * 10)
        buckets[index].append(num)
    for i in range(10):
        buckets[i].sort()
    return [num for bucket in buckets for num in bucket]
```

Time: O(n + k)

11. HASHING

Hashing is like a smart indexingsystem—ithelps us store and access data quickly, usually in O(1) time.

11.1 Hash Table Basics

- A hash table stores key-value pairs.

Think of it like labeled boxes:

- You put data inside a box using a key.
- A hash function turns your key into an index.

```python
# Python dict is a hash table
hash_table = {}
hash_table["apple"] = 2
print(hash_table["apple"])  # Output: 2
```

✅ Fast Access

❌Needs good hash function to avoid collisions

11.2 Frequency Counting

- Used when you want to count how many times something appears (like words, numbers, etc.).

Example: Count letters

```python
text = "banana"
freq = {}
for ch in text:
    freq[ch] = freq.get(ch, 0) + 1
print(freq)  # {'b':1, 'a':3, 'n':2}
```

Used in:

- Finding duplicates
- Anagram checking
- Word counters

11.3 HashSet vs HashMap

Feature	HashSet	HashMap
Stores	Only keys	Key + Value
Duplicates?	✕ Not allowed	✕ Keys unique
Access	O(1)	O(1)
Example (Python)	set()	dict()

11. HASHING

11.3 HashSet vs HashMap

Use case:

- HashSet → check existence (e.g., visited nodes, unique emails)
- HashMap → store actual data linked with keys

11.4 Collision Resolution

- Sometimes different keys generate the same index. That's a collision.

How to handle it?

1. Chaining (Linked Lists in Buckets)

- Each index holds a list of entries.

```python
hash_table = [[] for _ in range(10)]
def insert(key, value):
    index = hash(key) % 10
    hash_table[index].append((key, value))
```

✅ Easy to implement

❌Slower if many collisions

2. Open Addressing

- Instead of a list, if a spot is full, find next empty spot.

Types:

- Linear probing → check next cell
- Quadratic probing → check 1^2, 2^2, 3^2 steps away
- Double hashing → use second hash

```python
# Example idea
# hash(key) → index
# if occupied → index + 1, +2, +3, ...
```

✅ Saves memory

❌Slower as table fills

12. TREES

12.1 Tree Terminology

- Node: Basic unit of a tree containing data.
- Root: The topmost node of the tree.
- Child: A node directly connected to another node when moving away from the root.
- Parent: The node which has children.
- Leaf: A node with no children.
- Subtree: Any node and its descendants.
- Depth: Number of edges from the root to the node.
- Height: Number of edges on the longest path from the node to a leaf.
- Binary Tree: A tree where each node has at most two children (left and right).

12.2 Binary Trees

A binary tree is a tree where each node has at most two children.

Example structure:

- Each node has a left and right pointer.

Python definition:

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

12.3 Binary Search Trees (BST)

- A special kind of binary tree where:
- Left subtree has values less than the node.
- Right subtree has values greater than the node.
- This allows fast search, insert, and delete operations in O(log n) time (if the tree is balanced).
- BST Insert Example:

```python
def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.data:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root
```

12. TREES

12.4 Tree Traversals

Ways to visit each node in a tree:

- Inorder (Left, Root, Right): Used in BSTs to get sorted order.
- Preorder (Root, Left, Right): Used to copy the tree.
- Postorder (Left, Right, Root): Used to delete the tree.
- Level-order: Visit level by level (uses queue).

Example (Inorder):

```python
def inorder(root):
    if root:
        inorder(root.left)
        print(root.data)
        inorder(root.right)
```

12.5 Height and Diameter of a Tree

- Height: Length of the longest path from root to a leaf.
- Diameter: Longest path between any two nodes (may or may not pass through root).

Height:

```python
def height(root):
    if root is None:
        return 0
    return 1 + max(height(root.left), height(root.right))
```

Diameter:

```python
def diameter(root):
    if root is None:
        return 0
    l = height(root.left)
    r = height(root.right)
    d1 = diameter(root.left)
    d2 = diameter(root.right)
    return max(l + r + 1, max(d1, d2))
```

12.6 LCA (Lowest Common Ancestor)

- LCA of two nodes is the deepest node that is an ancestor of both.

In a Binary Tree:

```python
def lca(root, n1, n2):
    if root is None:
        return None
    if root.data == n1 or root.data == n2:
        return root
    left = lca(root.left, n1, n2)
    right = lca(root.right, n1, n2)
    if left and right:
        return root
    return left if left else right
```

12. TREES

12.7 Balanced Binary Tree

A binary tree is balanced if for every node:

- Height of left and right subtrees differ by at most 1.

Check if balanced:

```python
def is_balanced(root):
    def check(root):
        if root is None:
            return 0
        left = check(root.left)
        right = check(root.right)
        if abs(left - right) > 1:
            raise Exception("Unbalanced")
        return 1 + max(left, right)
    try:
        check(root)
        return True
    except:
        return False
```

12.8 AVL Tree (Advanced)

- AVL Tree is a self-balancing binary search tree. After every insertion or deletion, the tree checks and restores the balance using rotations.

Properties:

- Balance factor = height(left) - height(right)
- Balance factor must be -1, 0, or 1

Used where frequent insertions/deletions occur.

12.9 Segment Tree (Advanced)

- Segment Tree is a tree used for storing information about intervals or segments.

Useful for:

- Range minimum query
- Range sum query
- Range updates

Construction takes O(n), queries and updates in O(log n).
Used in competitive programming and data range queries.

13. TRIES (PREFIX TREE)

A Trie is a special tree used to store strings efficiently, especially for problems involving prefixes and dictionary words.

13.1 Trie Basics

- Trie is a tree where each node represents a character of a word.
- The root node is empty, and each path from root to a node spells out a word/prefix.
- Mainly used for autocomplete, prefix searching, and spell checking.

Example:
Words: cat, cap, can
Structure:

Each path forms a word starting from the root.

13.2 Insert & Search Words

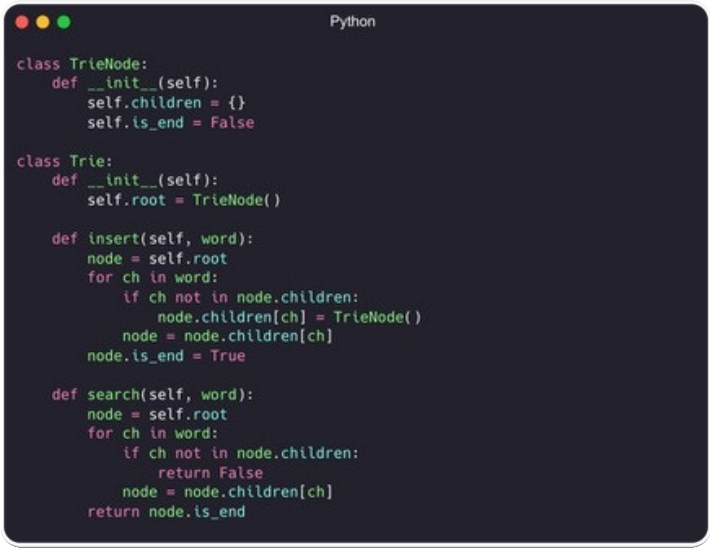
Insert:

- Start from root.
- For each character in the word:
  - If node for the character doesn't exist, create it.
- After last character, mark node as end of word.

Search:

- Traverse character by character from root.
- If path breaks, word not found.
- If final character reached and node is marked end, word exists.

Python Example:



13. TRIES (PREFIX TREE)

13.3 Prefix Matching

- Used to check if any word in the Trie starts with a given prefix.
- Same as search, but you don't check for is_end.
- Just ensure all prefix characters exist in the Trie.

Example:

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

If you have stored ["apple", "app", "ape"], then:

- starts_with("ap") → True
- starts_with("bat") → False

13.4 Word Suggestions (Autocomplete)

- Given a prefix, return all words in the Trie that begin with it.
- First, find the node for the prefix.
- Then do a DFS from that node to collect all words.

Example:

```python
def suggestions(self, prefix):
    def dfs(node, path, results):
        if node.is_end:
            results.append(path)
        for ch in node.children:
            dfs(node.children[ch], path + ch, results)

    node = self.root
    for ch in prefix:
        if ch not in node.children:
            return []
        node = node.children[ch]

    result = []
    dfs(node, prefix, result)
    return result
```

If Trie contains ["car", "cat", "cart", "care"]:

- suggestions("ca") → ["car", "cart", "care", "cat"]

14. GRAPHS

A graph is a collection of nodes (vertices) connected by edges. Graphs are used to model real-world relationships — like roads between cities or friendships on social media.

14.1 Graph Representation

1. Adjacency List

- Each node stores a list of its neighbors.
- Efficient for sparse graphs.

Example:

```python
graph = {
    0: [1, 2],
    1: [0, 3],
    2: [0],
    3: [1]
}
```

2. Adjacency Matrix

- A 2D array where matrix[i][j] = 1 if edge exists.
- Good for dense graphs, but uses more space.

Example:

```markdown
  0 1 2 3
 ---------
0| 0 1 1 0
1| 1 0 0 1
2| 1 0 0 0
3| 0 1 0 0
```

14.2 BFS & DFS

BFS (Breadth-First Search)

- Level-by-level traversal.
- Uses queue.
- Good for shortest path in unweighted graphs.

```python
from collections import deque
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            queue.extend(graph[node])
```

14. GRAPHS

14.2 BFS & DFS

DFS (Depth-First Search)

- Goes deep before backtracking.
- Uses stack (or recursion).

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

14.3 Detect Cycle

Undirected Graph:

- Use DFS and check if a visited node is not the parent.

Directed Graph:

- Use DFS with a recursion stack to track nodes in the current path.

14.4 Topological Sort

- Works on Directed Acyclic Graphs (DAG).
- Linear ordering of tasks based on dependencies.
- Use DFS or Kahn's Algorithm (BFS + in-degree).

14.5 Connected Components

- A connected component is a group of nodes where each node is reachable from any other.
- Use DFS/BFS to count all components.

14.6 Bipartite Graph

- A graph is bipartite if nodes can be colored with 2 colors without adjacent nodes having the same color.
- Check using BFS/DFS with coloring.

14.7 Bridges and Articulation Points

- Bridges: Removing the edge increases components.
- Articulation Points: Removing the node increases components.
- Found using DFS + low time & discovery time.

14. GRAPHS

14.8 Dijkstra's Algorithm

- Finds shortest path in a weighted graph (no negative weights).
- Uses a min-heap (priority queue).

```python
import heapq
def dijkstra(graph, start):
    dist = {node: float('inf') for node in graph}
    dist[start] = 0
    pq = [(0, start)]
    while pq:
        d, node = heapq.heappop(pq)
        for neighbor, weight in graph[node]:
            if d + weight < dist[neighbor]:
                dist[neighbor] = d + weight
                heapq.heappush(pq, (dist[neighbor], neighbor))
```

14.9 Bellman-Ford Algorithm

- Works with negative weights.
- Slower than Dijkstra.
- Runs V - 1 times over all edges (V = vertices).

14.10 Floyd-Warshall Algorithm

- All pairs shortest path.
- Time: $O(V^3)$
- Dynamic Programming-based.

```python
for k in range(V):
    for i in range(V):
        for j in range(V):
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

14.11 MST (Minimum Spanning Tree)

Prim's Algorithm

- Start from one node and expand to nearest unvisited nodes.
- Uses priority queue.

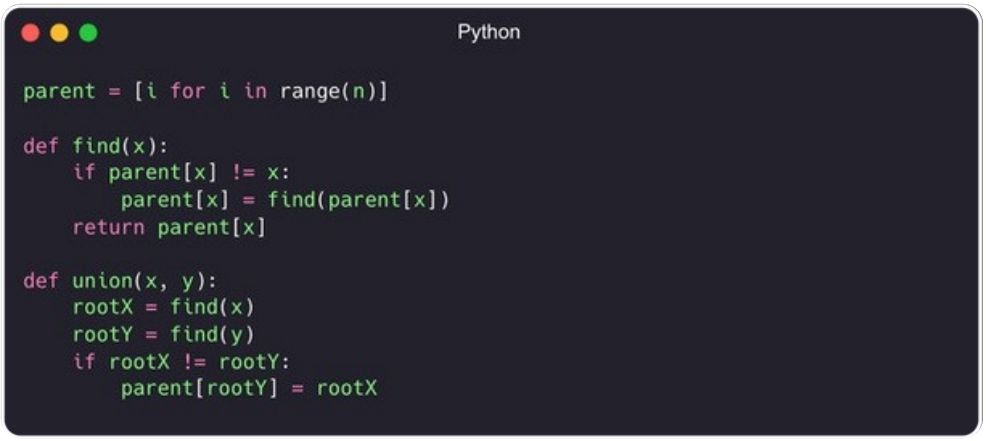
Kruskal's Algorithm

- Sort all edges, add smallest one if it doesn't form a cycle.
- Uses Union-Find (DSU) to detect cycles.

14. GRAPHS

14.12 Union-Find (Disjoint Set Union)

- A structure to manage connected components efficiently.
- Supports:
  - Find: Get root of a node.
  - Union: Connect two components.
- Uses path compression and union by rank for speed.



15. GREEDY ALGORITHMS

Greedy Algorithms make the best possible choice at each step — hoping it leads to the overall best solution. They don't always guarantee the optimal answer, but often work well for specific problems.

15.1 Activity Selection Problem

Goal: Select the maximum number of non-overlapping activities.

Approach:

- Sort activities by end time.
- Always pick the next activity that ends the earliest and doesn't overlap.

```python
activities = [(1, 2), (3, 4), (0, 6), (5, 7), (8, 9)]
activities.sort(key=lambda x: x[1])  # sort by end time

count = 1
end_time = activities[0][1]

for i in range(1, len(activities)):
    if activities[i][0] >= end_time:
        count += 1
        end_time = activities[i][1]

print("Max activities:", count)
```

15.2 Fractional Knapsack

Goal: Maximize profit by taking items with given weight and value, but you can take fractions of items.

Approach:

- Calculate value/weight ratio for all items.
- Sort by highest ratio.
- Take as much as you can from highest ratio item until the bag is full.

```python
items = [(60, 10), (100, 20), (120, 30)]  # (value, weight)
capacity = 50

items.sort(key=lambda x: x[0]/x[1], reverse=True)

total = 0
for value, weight in items:
    if capacity >= weight:
        total += value
        capacity -= weight
    else:
        total += value * (capacity / weight)
        break

print("Max value:", total)
```

15. GREEDY ALGORITHMS

15.3 Huffman Encoding

Goal: Compress data by assigning shorter binary codes to more frequent characters.

Approach:

- Build a min heap of characters based on frequency.
- Merge two smallest nodes until one remains (build tree).
- Assign binary codes by traversing the tree.

Example:

- For frequencies: a:5, b:9, c:12, d:13, e:16, f:45
- Build a binary tree where:
  - Frequent characters are lower in the tree.
  - Infrequent ones are higher and get longer codes.

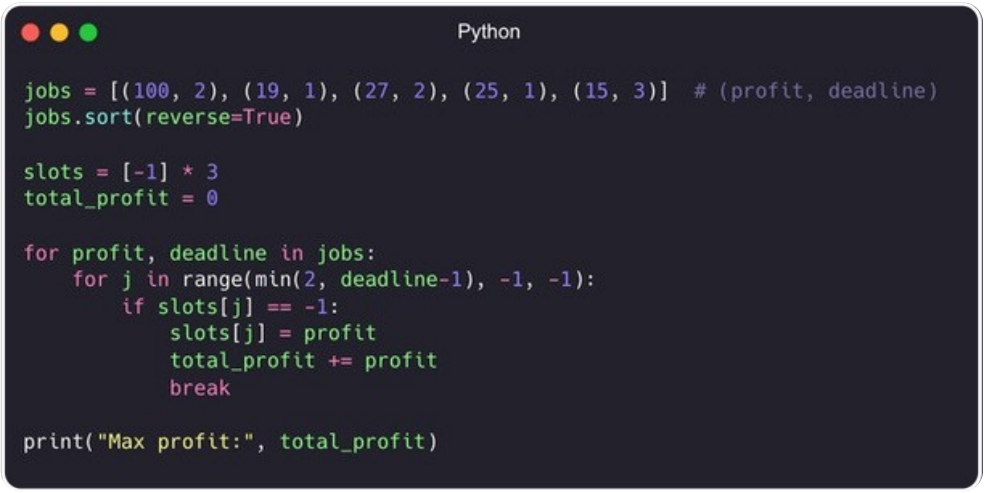
Used in file compression like .zip, .rar.

15.4 Job Scheduling Problem

Goal: Maximize profit by doing jobs with deadlines and profits (only one job at a time).

Approach:

- Sort jobs by profit (descending).
- For each job, try to schedule it as late as possible before its deadline.



15. GREEDY ALGORITHMS

15.5 Coin Change (Greedy)

Goal: Give change using the minimum number of coins.

Approach (Greedy):

- Sort coins in descending order.
- Pick largest coin possible each time until the amount becomes zero.

Works when coins are canonical (like Indian coins).

Note: Greedy may not work for all coin sets (like [9, 6, 1]), where dynamic programming is better.

16. DYNAMIC PROGRAMMING

Dynamic Programming is used to solve problems by breaking them into smaller subproblems and storing results to avoid recalculating.

16.1 Memoization vs Tabulation

- Memoization (Top-Down): Use recursion and store answers to subproblems in a map or array.
- Tabulation (Bottom-Up): Use loops to build up the solution from base cases.

```python
# Memoization example: Fibonacci
def fib(n, memo={}):
    if n <= 1:
        return n
    if n not in memo:
        memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

```python
# Tabulation example: Fibonacci
def fib(n):
    dp = [0, 1]
    for i in range(2, n+1):
        dp.append(dp[i-1] + dp[i-2])
    return dp[n]
```

16.2 0/1 Knapsack

- Given weights and values of items, find the maximum value you can get in a bag with limited capacity. Each item can be picked at most once.

```python
def knapsack(W, wt, val, n):
    dp = [[0]*(W+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for w in range(W+1):
            if wt[i-1] <= w:
                dp[i][w] = max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][W]
```

16.3 Longest Common Subsequence (LCS)

- Find the longest subsequence (not substring) present in both strings in the same order.

```python
def lcs(X, Y):
    m, n = len(X), len(Y)
    dp = [[0]*(n+1) for _ in range(m+1)]
    for i in range(m):
        for j in range(n):
            if X[i] == Y[j]:
                dp[i+1][j+1] = 1 + dp[i][j]
            else:
                dp[i+1][j+1] = max(dp[i][j+1], dp[i+1][j])
    return dp[m][n]
```

16. DYNAMIC PROGRAMMING

16.4 Longest Increasing Subsequence (LIS)

- Find the length of the longest increasing subsequence in an array.

```python
def lis(arr):
    n = len(arr)
    dp = [1]*n
    for i in range(1, n):
        for j in range(i):
            if arr[i] > arr[j]:
                dp[i] = max(dp[i], dp[j]+1)
    return max(dp)
```

16.5 Matrix Chain Multiplication

- Given matrices, find the best order to multiply them with the least cost (number of multiplications).

```python
# Cost array = matrix dimensions
def matrix_chain_order(p):
    n = len(p) - 1
    dp = [[0]*n for _ in range(n)]
    for l in range(2, n+1):
        for i in range(n-l+1):
            j = i + l - 1
            dp[i][j] = float('inf')
            for k in range(i, j):
                cost = dp[i][k] + dp[k+1][j] + p[i]*p[k+1]*p[j+1]
                dp[i][j] = min(dp[i][j], cost)
    return dp[0][n-1]
```

16.6 DP on Subsets, Strings, Grids, Trees

- Subsets: Use bitmasks or recursion + DP for problems like subset sum.
- Strings: Problems like LCS, edit distance use DP. Grids: Use 2D DP for
- path finding, max sum, etc. Trees: Use post-order DP for problems like
- diameter or counting.

16.7 Optimal BST, Egg Dropping, Rod Cutting

- Optimal BST: Minimize search cost using frequency-based DP.
- Egg Dropping: Minimize attempts to find threshold floor using DP.
- Rod Cutting: Maximize profit by cutting rods into pieces.

```python
# Rod Cutting
def rod_cutting(prices, n):
    dp = [0]*(n+1)
    for i in range(1, n+1):
        for j in range(i):
            dp[i] = max(dp[i], prices[j] + dp[i-j-1])
    return dp[n]
```

17. SLIDING WINDOW & TWO POINTER

17.1 Maximum Sum Subarray of Size K

- Find max sum of any subarray of size K.

```python
def max_sum_k(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
```

17.2 Longest Substring Without Repeating Characters

- Find longest substring without duplicate characters.

```python
def longest_unique_substring(s):
    start = 0
    seen = {}
    max_len = 0
    for end in range(len(s)):
        if s[end] in seen and seen[s[end]] >= start:
            start = seen[s[end]] + 1
        seen[s[end]] = end
        max_len = max(max_len, end - start + 1)
    return max_len
```

17.3 Count Anagrams

- Count how many substrings are anagrams of a given pattern.

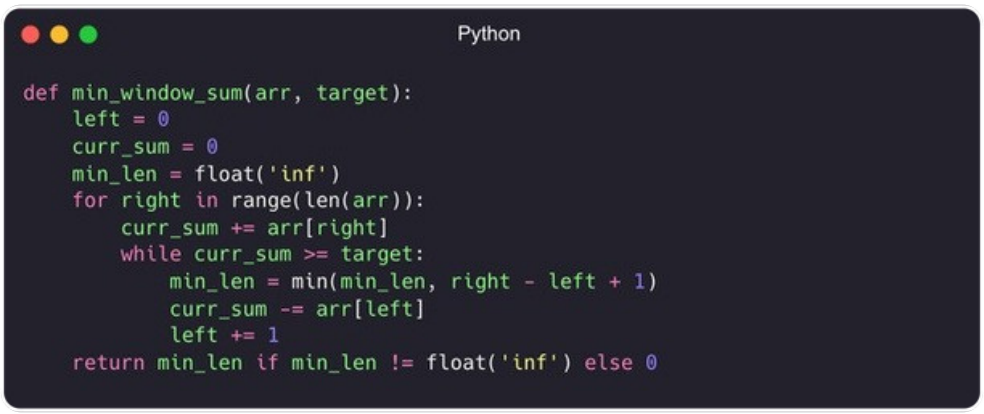
```python
from collections import Counter

def count_anagrams(s, p):
    k = len(p)
    target = Counter(p)
    window = Counter(s[:k])
    count = 0
    if window == target:
        count += 1
    for i in range(k, len(s)):
        window[s[i-k]] -= 1
        if window[s[i-k]] == 0:
            del window[s[i-k]]
        window[s[i]] += 1
        if window == target:
            count += 1
    return count
```

17. SLIDING WINDOW & TWO POINTER

17.4 Variable Window Size Problems

- Use when window size is not fixed. Example: Smallest window with a given sum.



18. BIT MANIPULATION

18.1 AND, OR, XOR, NOT

- Bitwise operations work on bits (0 and 1) of numbers.
- AND (&): 1 only if both bits are 1.
- OR (|): 1 if any bit is 1.
- XOR (^): 1 if bits are different.
- NOT (~): Flips bits (0 → 1, 1 → 0).

Example:

```python
a = 5  # 0101
b = 3  # 0011

print(a & b)  # 1  (0001)
print(a | b)  # 7  (0111)
print(a ^ b)  # 6  (0110)
print(~a)     # -6 (in two's complement)
```

18.2 Checking Even/Odd

- Check last bit using AND with 1.
- If (n & 1) == 0 → even, else odd.

```python
def is_even(n):
    return (n & 1) == 0
```

18.3 Set/Clear/Toggle Bits

- Set bit: Turn a bit ON → Use OR with (1 << pos).
- Clear bit: Turn a bit OFF → Use AND with ~(1 << pos).
- Toggle bit: Flip bit → Use XOR with (1 << pos).

```python
def set_bit(n, pos):
    return n | (1 << pos)

def clear_bit(n, pos):
    return n & ~(1 << pos)

def toggle_bit(n, pos):
    return n ^ (1 << pos)
```

18.4 Counting Set Bits

- Count how many 1s are in binary representation.
- Simple way: Check each bit or use n & (n-1) trick to remove last set bit.

```python
def count_set_bits(n):
    count = 0
    while n:
        n = n & (n-1)
        count += 1
    return count
```

18. BIT MANIPULATION

18.5 Power of Two Check

- A number is power of two if it has only 1 set bit.
- Check using n & (n-1) == 0 (and n > 0).

18.6 XOR Tricks (Single Number, Two Repeating Numbers)

- XOR of a number with itself is 0.
- XOR all numbers; result is the single number which appears once.
- For two numbers appearing once, use XOR and bit masking to separate.

19. HEAP / PRIORITY QUEUE

19.1 Min Heap / Max Heap Basics

- Heap is a complete binary tree with special order:
  - Min heap: Parent ≤ children.
  - Max heap: Parent ≥ children.
- Supports efficient insertion, deletion of min/max.

19.2 Heapify

- Process to build heap from unordered array.
- Heapify ensures heap property starting from bottom nodes.

19.3 K Largest/Smallest Elements

- Use heap of size K to keep track.
- For largest: use min heap of size K.
- For smallest: use max heap of size K (or invert values).

19.4 Merge K Sorted Lists

- Use a min heap to merge K sorted lists efficiently.
- Push first elements of each list into heap.
- Extract min and push next element from same list until done.

19.5 Median in Stream

- Maintain two heaps: max heap for lower half, min heap for upper half.
- Balancing heaps allows O(log n) median finding after each insertion.

20. SEGMENT TREES & BINARY INDEXED TREE (ADVANCED)

20.1 Range Sum/Min/Max Queries – Why Do We Need These?

- Let's say you have an array like this:

arr = [2, 4, 5, 7, 8, 9, 1]

Now imagine you're asked repeatedly:

- What's the sum from index 2 to 5?
- What's the minimum value between index 0 and 3?

You can solve this using a for loop in O(n) time for each query. But if you get thousands of queries, it becomes slow.

- That's where Segment Tree and BIT (Fenwick Tree) help.

They allow:

- Fast range queries (sum, min, max): in O(log n) time
- Fast updates to the array: in O(log n) time

20.2 Segment Tree

- A Segment Tree is a binary tree built on top of an array.
- Each node represents a range of values and stores a result (like sum, min, max) for that range.

Example:

- Array: [1, 3, 5, 7]

Segment tree will store:

- Root = sum(0 to 3) = 16
- Left child = sum(0 to 1) = 4
- Right child = sum(2 to 3) = 12
- And so on...

Operations:

- Build Tree: O(n)
- Range Query (sum/min/max): O(log n)
- Update an element: O(log n)

Use Case:

Imagine a leaderboard of scores – if a score changes or we want to know the sum of top 10 players, a Segment Tree gives fast answers.

20.3 Lazy Propagation

- Sometimes we want to update a whole range (e.g., add 5 to all values from index 2 to 6).

Naively updating each element takes O(n), but Lazy Propagation marks the update and applies it only when needed.

It helps us:

- Avoid unnecessary updates
- Keep operations in O(log n)

20. SEGMENT TREES & BINARY INDEXED TREE (ADVANCED)

20.4 Fenwick Tree (Binary Indexed Tree - BIT)

- A Fenwick Tree is another tree-like structure for prefix sums.

It's smaller and easier than a Segment Tree but only works with:

- Point updates (single element)
- Prefix queries (sum from 1 to i)

Example Use:

- Count how many numbers are ≤ 5 in a dynamic array.

Operations:

- Update(index, value) – O(log n)
- Query(index) – prefix sum up to index – O(log n)

Comparison:

Feature	Segment Tree	BIT (Fenwick)
Space	4 * n	n
Range queries	Yes	Only prefix
Range updates	With Lazy	No
Easy to implement	Medium	Easy

21. DISJOINT SET UNION (DSU)

- DSU is a data structure that helps you manage a collection of disjoint sets.

Used in problems where you need to:

- Check if two elements are in the same group
- Merge two groups

Example:

You have 5 people and friendships like:

- A knows B
- C knows D

Then:

- {A, B} is one set
- {C, D} is another
- {E} is alone

- DSU helps maintain these groups efficiently.

21.1 Union by Rank

- When merging two sets, we attach the smaller tree under the bigger one.
- This keeps the resulting tree short.
- And short trees mean faster future operations.

21.2 Path Compression

- When you find the "leader" (also called root or parent) of a set, you can flatten the path.
- Let all the children point directly to the root.
- This makes future searches faster.
- Combined with Union by Rank, this gives almost O(1) time per operation.

21.3 Applications in Graphs (Kruskal's Algorithm)

- In Kruskal's Algorithm (for Minimum Spanning Tree):
- You want to add edges without creating cycles.
- So, before adding an edge between u and v, use DSU:
- If u and v are in the same set ⇒ adding the edge creates a cycle ⇒ skip it.
- Otherwise, add the edge and merge the sets.

22. ADVANCED TOPICS (OPTIONAL)

These are high-level topics usually seen in competitive programming or advanced algorithmic applications. Each has a unique use case and shines in specific scenarios.

22.1 Rabin-Karp Algorithm – Pattern Matching with Hashing

Use Case:

- Find all occurrences of a pattern in a long string.

Example: Find how many times "abc" appears in "ababcabcabc".

How It Works:

- Compute a hash for the pattern (e.g., "abc").
- Slide a window of the same size over the main text.
- For each window, compute its hash and compare with the pattern's hash.
- If hash matches → double-check by comparing characters.

Why It's Fast:

- Hashing allows skipping many unnecessary character-by-character checks.
- It works in O(n + m) time on average (n = text length, m = pattern length).

22.2 Tarjan's Algorithm – Strongly Connected Components (SCC)

Use Case:

- In a directed graph, find groups of nodes where every node can reach every other in the group.

Example:

- If A → B → C → A, then {A, B, C} is a Strongly Connected Component.

Why Use It:

- Useful in circuit design, web crawling, module dependency resolution.

How It Works:

- DFS-based approach.
- Uses discovery time and low-link values.
- Maintains a stack to track the current path.
- If it finds a cycle, it groups all nodes in that cycle into a component.

Time Complexity: O(V + E)

22.3 KMP Algorithm – Pattern Matching with Prefix Table

Use Case:

- Search for a pattern inside a text without re-checking characters.

Example: Search "aabaa" in a long paragraph efficiently.

Why KMP Is Smart:

- Builds a prefix table (also called LPS array – Longest Prefix which is also Suffix).
- This table helps you know where to continue matching after a mismatch.

Time Complexity: O(n + m)

22. ADVANCED TOPICS (OPTIONAL)

22.4 Convex Hull – Geometry Problem

Use Case:

- Given a set of points on a plane, find the smallest polygon (boundary) that encloses all points.

Example:

- If you have 10 scattered dots on a paper, the convex hull is the rubber band stretched around them.

Why It's Important:

- Used in computer graphics, GIS mapping, collision detection, robotics.

Popular Algorithms:

- Graham Scan (O(n log n))
- Jarvis March (O(nh), where h = number of hull points)

22.5 MO's Algorithm – Offline Range Queries

Use Case:

- Given multiple queries of form:

"What is the frequency of elements between index L to R?"

- Problem: Doing each query individually is slow for large data.

MO's Trick:

- Sort the queries cleverly using √n blocks (square root decomposition).
- Process them in that sorted order to minimize data movement.
- Great for range frequency problems, prefix sums, etc.

Time Complexity: O((n + q) * √n)

22.6 Square Root Decomposition (Sqrt Decomposition)

Use Case:

- To handle range queries and point updates faster than brute-force but simpler than Segment Trees.

How It Works:

- Divide the array into blocks of size √n.
- Precompute the result (like sum, min) for each block.
- When answering queries:
  - Use full blocks where possible.
  - Manually process the partial blocks on the edges.

Time Complexity:

- Build: O(n)
- Query: O(√n)
- Update: O(1) or O(√n)

Used In:

- Range Sum Queries
- Range GCD queries
- Updating single elements

22. ADVANCED TOPICS (OPTIONAL)

Final Thought:

These algorithms might seem advanced at first, but you don't need to memorize them—just understand:

- Where to use them
- Why they are efficient
- Basic idea of how they work

23. COMMON PATTERNS & TEMPLATES

These are reusable techniques used in many coding problems. Mastering them helps solve a wide range of problems faster.

23.1 Sliding Window Template

Use Case:

Used when dealing with contiguous subarrays or substrings, especially for problems like:

- Max/min sum of subarray
- Longest substring without repeating characters
- Count of substrings with some condition

Two Types:

- Fixed Window Size (e.g., size k)
- Variable Window Size (e.g., until a condition is met)

Template (Variable Window):

Python

```python
start = 0
for end in range(len(arr)):
    # Expand the window with arr[end]

    while some_condition:  # shrink if condition breaks
        # Shrink the window from start
        start += 1

    # Update result if needed
```

23.2 Recursion + Memoization Template

Use Case:

Problems that have overlapping subproblems, especially in Dynamic Programming, like:

- Fibonacci
- 0/1 Knapsack
- LCS (Longest Common Subsequence)

Template (Top-down DP):

Python

```python
def dp(i, j):
    if base_case:
        return some_value
    if (i, j) in memo:
        return memo[(i, j)]

    # Compute the result
    memo[(i, j)] = dp(i - 1, j) + dp(i, j - 1)
    return memo[(i, j)]

memo = {}
result = dp(start_i, start_j)
```

23. COMMON PATTERNS & TEMPLATES

23.3 BFS / DFS Template

- **DFS Template (Recursive):**

Used for:

- Tree traversals
- Connected components
- Cycle detection

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

- **BFS Template (Level Order):**

Used for:

- Shortest path in unweighted graphs
- Level order traversal in trees

```python
from collections import deque

def bfs(start):
    queue = deque([start])
    visited = set([start])

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

23.4 Binary Search on Answer Template

Use Case:

- When the answer lies within a range and the problem asks for min/max valid value.

Common Examples:

- Minimum largest subarray sum
- Capacity to ship packages in D days
- Koko eating bananas (Leetcode)

Template:

```python
low, high = 1, max_possible_value
result = -1

while low <= high:
    mid = (low + high) // 2
    if is_valid(mid):
        result = mid
        high = mid - 1  # try smaller
    else:
        low = mid + 1   # try larger
```

23. COMMON PATTERNS & TEMPLATES

Key Idea:

Use binary search when:

- You can define a range (e.g., capacity, time, size)
- You have a function to check if a guess is valid (is_valid)

24. COMMON BEGINNER MISTAKES IN DSA

Learning Data Structures and Algorithms (DSA) is like learning a new language. Mistakes are normal—but here are some common ones to avoid:

24.1 Forgetting to Check Edge Cases

Example mistakes:

- Empty array or string
- Array with only 1 element
- Very large/small input values
- Duplicates or negative numbers

Tip:

Always ask:
"What happens if the input is empty or at its limits?"

24.2 Not Understanding Time and Space Complexity

- **Mistake:** Writing a solution that is correct but too slow.

Why it matters:

- If your code runs in O(n^2) on large inputs (like n = 10^5), it may timeout or crash.

Tip:

- Analyze each loop and recursion to estimate complexity.

24.3 Using the Wrong Data Structure

Example:

- Using a list to search elements (O(n)) instead of a set (O(1))
- Using arrays instead of heaps for priority queues

Tip:

Learn which data structure fits which problem:

- Set for lookup
- Queue/Stack for order
- Heap for top-k
- HashMap for counting

24.4 Confusing Recursion Base Case

Mistake:

- Forgetting the base case or putting it in the wrong place.

Result:

- Stack overflow (infinite recursion) or wrong output.

Tip:

Always define:

- What is the base case?
- When should recursion stop?

24. COMMON BEGINNER MISTAKES IN DSA

24.5 Ignoring Input Constraints

Mistake:

- Using brute force when n is very large.

Tip:

Read constraints carefully:

- If n ≤ 10^5, use O(n log n)
- If n ≤ 20, brute force may be fine

24.6 Off-by-One Errors in Loops

Mistake:

Using <= instead of < in loops or starting at the wrong index.

Example:

Tip:

- Draw small input examples and trace the loop.

24.7 Mistakes in Indexing Arrays or Strings

Common Errors:

- Going out of bounds
- Using wrong start/end in slicing

Tip:

Use debugging prints or assert statements.

24.8 Not Handling Duplicates Properly

Example Mistake:

- Counting or skipping duplicates wrongly in problems like:
  - Two Sum
  - Subsets
  - Combinations

Tip:

Sort the input if needed and add logic to skip duplicates.

24. COMMON BEGINNER MISTAKES IN DSA

Final Advice:

Mistake	Fix
Forgetting edge cases	Write test cases manually
Wrong complexity	Estimate time for large inputs
Wrong data structure	Understand use-cases
Recursion bugs	Add base case first
Off-by-one	Use print/debug to check loop
Indexing error	Draw array and test logic
Duplicate issues	Sort input + check previous value