**Bare Minimum algorithms**

**1. Sorting Algorithms**

* **Bubble Sort**
* **Selection Sort**
* **Insertion Sort**
* **Merge Sort**
* **Quick Sort**
* **Heap Sort**
* **Counting Sort**
* **Radix Sort**

**2. Searching Algorithms**

* **Linear Search**
* **Binary Search** (Iterative & Recursive)
* **Ternary Search**
* **Jump Search**
* **Interpolation Search**
* **Exponential Search**

**3. Recursion & Backtracking**

* **Tower of Hanoi**
* **N-Queens Problem**
* **Sudoku Solver**
* **Rat in a Maze**
* **Word Search (Matrix-based Backtracking)**
* **Generate All Subsets / Permutations / Combinations**

**4. Divide and Conquer**

* **Merge Sort**
* **Quick Sort**
* **Binary Search**
* **Strassen’s Matrix Multiplication**
* **Closest Pair of Points Problem**

**5. Dynamic Programming (DP)**

* **Fibonacci Series (Memoization & Tabulation)**
* **0/1 Knapsack Problem**
* **Unbounded Knapsack Problem**
* **Coin Change Problem**
* **Longest Common Subsequence (LCS)**
* **Longest Increasing Subsequence (LIS)**
* **Edit Distance Problem**
* **Matrix Chain Multiplication**
* **Subset Sum Problem**
* **Rod Cutting Problem**
* **Egg Dropping Puzzle**

**6. Graph Algorithms**

* **Breadth-First Search (BFS)**
* **Depth-First Search (DFS)**
* **Dijkstra’s Algorithm (Shortest Path)**
* **Bellman-Ford Algorithm**
* **Floyd-Warshall Algorithm**
* **Prim’s Algorithm (Minimum Spanning Tree - MST)**
* **Kruskal’s Algorithm (MST)**
* **Topological Sorting (Kahn’s Algorithm & DFS-based)**
* **Strongly Connected Components (Kosaraju’s Algorithm, Tarjan’s Algorithm)**
* **Cycle Detection (Directed & Undirected Graphs)**
* **Bipartite Graph Check (BFS/DFS Method)**
* **Graph Coloring Problem**

**7. Tree Algorithms**

* **Binary Tree Traversals (Inorder, Preorder, Postorder)**
* **Lowest Common Ancestor (LCA)**
* **Binary Search Tree (BST) Operations (Insert, Delete, Search)**
* **Diameter of a Binary Tree**
* **Balanced Binary Trees (AVL Trees, Red-Black Trees, Splay Trees)**
* **Trie Data Structure (Insert, Search, Delete)**
* **Morris Traversal (Tree Traversal Without Recursion/Stack)**
* **Segment Tree (Range Queries, Lazy Propagation)**
* **Fenwick Tree / Binary Indexed Tree (BIT)**

**8. String Algorithms**

* **KMP (Knuth-Morris-Pratt) Algorithm**
* **Rabin-Karp Algorithm**
* **Z-Algorithm (Pattern Matching)**
* **Manacher’s Algorithm (Longest Palindromic Substring)**
* **Aho-Corasick Algorithm (Multi-pattern Matching)**
* **Suffix Array and LCP Array**
* **Suffix Tree**
* **Trie-Based String Matching**

**9. Bit Manipulation**

* **Check if a Number is Power of Two**
* **Count Set Bits in an Integer (Kernighan’s Algorithm)**
* **Find the Only Non-Repeating Element in an Array of Duplicates (XOR Trick)**
* **Find Two Non-Repeating Elements in an Array (Using XOR)**
* **Swap Two Numbers Without Using Temporary Variable**
* **Find the Missing Number in an Array of Size N with Elements 1 to N+1**

**10. Greedy Algorithms**

* **Activity Selection Problem**
* **Huffman Encoding**
* **Job Scheduling Problem**
* **Fractional Knapsack Problem**
* **Optimal Merge Pattern**
* **Dijkstra’s Algorithm (Greedy Shortest Path)**
* **Prim’s Algorithm (MST - Greedy Approach)**

**11. Mathematical Algorithms**

* **Greatest Common Divisor (GCD) – Euclidean Algorithm**
* **Least Common Multiple (LCM) using GCD**
* **Sieve of Eratosthenes (Finding Prime Numbers Efficiently)**
* **Checking if a Number is Prime (Optimized Primality Test)**
* **Modular Exponentiation (Power Function with Modulo)**
* **Fast Exponentiation (Binary Exponentiation)**
* **Chinese Remainder Theorem**
* **Fibonacci Using Matrix Exponentiation**

**12. Miscellaneous Algorithms**

* **Reservoir Sampling Algorithm**
* **Floyd’s Tortoise and Hare (Cycle Detection in Linked List)**
* **Kadanes Algorithm (Maximum Subarray Sum)**
* **Moore’s Voting Algorithm (Finding Majority Element in an Array)**
* **Tortoise-Hare Algorithm (Finding a Cycle in a Linked List)**

**13. Advanced Topics (Optional but Useful)**

* **Disjoint Set Union (Union-Find Algorithm, Path Compression & Rank Optimization)**
* **Heavy-Light Decomposition (HLD) for Trees**
* **Treap (Tree + Heap)**
* **Persistent Segment Tree**
* **Suffix Automaton**
* **Network Flow Algorithms (Ford-Fulkerson, Edmonds-Karp Algorithm)**

**Where to Practice?**

* **Leetcode** (Best for interview prep)
* **Codeforces** (For competitive programming)
* **GeeksforGeeks** (Good for explanations)
* **HackerRank**
* **AtCoder**
* **SPOJ**

**How to Learn These Efficiently?**

1. **Start with implementation-based problems** (Sorting, Searching, Recursion).
2. **Move to DP and Graphs** (They appear frequently in interviews).
3. **Practice problems from different difficulty levels** (Easy → Medium → Hard).
4. **Revise regularly** to retain concepts.
5. **Solve variations of the same problem** to build deeper intuition.

### ****Algorithm for BFS****

#### **Input**: A graph (as an adjacency list or adjacency matrix) and a starting node.

#### **Output**: The order in which nodes are visited.

1. **Initialize a queue** and enqueue the starting node.
2. **Mark the starting node as visited** to avoid revisiting.
3. While the queue is **not empty**:
   * Dequeue a node from the queue.
   * Process the current node (print/store it).
   * Enqueue all its **unvisited adjacent nodes**, marking them as visited.
4. Repeat until all reachable nodes are visited.

**Time & Space Complexity**

| **Operation** | **Complexity** |
| --- | --- |
| **Time Complexity** | **O(V + E)** (V = vertices, E = edges) |
| **Space Complexity** | **O(V)** (for storing visited nodes & queue) |

**Applications of BFS**

1. **Shortest Path in an Unweighted Graph**
   * BFS finds the shortest path in terms of the number of edges in an unweighted graph.
2. **Network Packet Routing**
   * Used in network broadcasting and routing algorithms.
3. **Web Crawlers**
   * Search engines use BFS to traverse the web.
4. **Finding Connected Components**
   * BFS helps in finding connected components in an undirected graph.
5. **Solving Maze Problems**
   * BFS helps find the shortest path in a maze/grid.
6. from collections import deque
7. def bfs(graph,start):
8. visited = set()
9. queue = deque([start])
10. while queue:
11. node = queue.popleft()
12. if node not in visited:
13. print(node , end = ' ')
14. visited.add(node)
15. for neighbor in graph[node]:
16. if neighbor not in visited:
17. queue.append(neighbor)
18. graph = {
19. 'A': ['B', 'C'],
20. 'B': ['A', 'D', 'E'],
21. 'C': ['A', 'F'],
22. 'D': ['B'],
23. 'E': ['B', 'F'],
24. 'F': ['C', 'E']
25. }
26. bfs(graph,'A')

**++++++++++++++++++++++++++++++++++++++++++**

### ****Depth-First Search (DFS) Algorithm****

#### **What is DFS?**

DFS is a graph traversal algorithm that explores as far as possible along a branch before backtracking. It uses a **stack (LIFO structure)** or recursion to traverse the graph.

### ****Algorithm for DFS****

#### **Input**: A graph (as an adjacency list or adjacency matrix) and a starting node.

#### **Output**: The order in which nodes are visited.

1. **Initialize a stack** (or use recursion) and push the starting node.
2. **Mark the starting node as visited** to avoid revisiting.
3. While the stack is **not empty**:
   * Pop a node from the stack.
   * Process the current node (print/store it).
   * Push all its **unvisited adjacent nodes** onto the stack, marking them as visited.
4. Repeat until all reachable nodes are visited.

### ****DFS Pseudocode (Iterative)****

DFS(graph, start\_node):

create a stack S

create a set visited

push start\_node into S

add start\_node to visited

while S is not empty:

current = pop S

process(current) # Print or store the node

for each neighbor in graph[current] (in reverse order for correct order):

if neighbor is not in visited:

push neighbor into S

add neighbor to visited

### ****DFS Pseudocode (Recursive)****

DFS(graph, node, visited):

if node is not in visited:

process(node) # Print or store the node

add node to visited

for neighbor in graph[node]:

DFS(graph, neighbor, visited)

### ****Example of DFS****

#### **Graph Representation**

Consider this graph:

markdown

CopyEdit

1

/ \

2 3

/ \ \

4 5 6

**Adjacency List Representation:**

python

CopyEdit

graph = {

1: [2, 3],

2: [4, 5],

3: [6],

4: [],

5: [],

6: []

}

#### **DFS Traversal (Starting from Node 1)**

1. Start with **stack = [1]**, visited = {1}
2. Pop 1, process it → **Output: 1**
3. Push 3, 2 → stack = [3, 2], visited = {1, 2, 3}
4. Pop 2, process it → **Output: 1, 2**
5. Push 5, 4 → stack = [3, 5, 4], visited = {1, 2, 3, 4, 5}
6. Pop 4, process it → **Output: 1, 2, 4**
7. Pop 5, process it → **Output: 1, 2, 4, 5**
8. Pop 3, process it → **Output: 1, 2, 4, 5, 3**
9. Push 6 → stack = [6], visited = {1, 2, 3, 4, 5, 6}
10. Pop 6, process it → **Output: 1, 2, 4, 5, 3, 6**
11. Stack is empty, so **DFS is complete**.

#### **Final DFS Output:**

CopyEdit

1 → 2 → 4 → 5 → 3 → 6

### ****Python Code for DFS****

#### **1. Iterative DFS using Stack**

def dfs\_iterative(graph, start):

visited = set()

stack = [start]

while stack:

node = stack.pop() # Pop last element (LIFO)

if node not in visited:

print(node, end=" ") # Process the node

visited.add(node)

stack.extend(reversed(graph[node])) # Push neighbors in reverse order

# Example graph

graph = {

1: [2, 3],

2: [4, 5],

3: [6],

4: [],

5: [],

6: []

}

dfs\_iterative(graph, 1)

**Output:**

1 2 4 5 3 6

#### **2. Recursive DFS**

python

CopyEdit

def dfs\_recursive(graph, node, visited=None):

if visited is None:

visited = set()

if node not in visited:

print(node, end=" ") # Process the node

visited.add(node)

for neighbor in graph[node]:

dfs\_recursive(graph, neighbor, visited)

# Example graph

graph = {

1: [2, 3],

2: [4, 5],

3: [6],

4: [],

5: [],

6: []

}

dfs\_recursive(graph, 1)

**Output:**

CopyEdit

1 2 4 5 3 6

### ****Time & Space Complexity****

| **Operation** | **Complexity** |
| --- | --- |
| **Time Complexity** | **O(V + E)** (V = vertices, E = edges) |
| **Space Complexity (Recursive DFS)** | **O(V)** (due to recursion stack) |
| **Space Complexity (Iterative DFS)** | **O(V)** (for storing visited nodes & stack) |

### ****Applications of DFS****

1. **Cycle Detection in Graphs**
   * DFS helps detect cycles in directed and undirected graphs.
2. **Topological Sorting**
   * Used in dependency resolution (e.g., course prerequisites).
3. **Solving Maze Problems**
   * DFS explores all possible paths before backtracking.
4. **Connected Components in Graphs**
   * DFS is used to find connected components in an undirected graph.
5. **Solving Puzzles (Sudoku, Word Search, N-Queens, etc.)**
   * DFS helps explore all possible solutions before finding the correct one.

### ****Merge Sort Algorithm****

#### **What is Merge Sort?**

Merge Sort is a **divide and conquer** sorting algorithm that splits an array into smaller subarrays, sorts them, and then merges them back together in a sorted manner.

### ****Algorithm Steps****

1. **Divide**:
   * If the array has **one or zero elements**, it is already sorted.
   * Otherwise, split the array into two halves (left and right).
2. **Conquer**:
   * Recursively sort both halves.
3. **Merge**:
   * Merge the two sorted halves into a single sorted array.

### ****Merge Sort Pseudocode****

MergeSort(arr):

if length of arr <= 1:

return arr

mid = length of arr // 2

left\_half = MergeSort(arr[0:mid]) # Recursively sort left half

right\_half = MergeSort(arr[mid:]) # Recursively sort right half

return Merge(left\_half, right\_half)

Merge(left, right):

result = []

i, j = 0, 0

while i < length of left and j < length of right:

if left[i] < right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

result.extend(left[i:]) # Add remaining elements from left half

result.extend(right[j:]) # Add remaining elements from right half

return result

### ****Example Execution****

#### **Given Array:**

[8, 3, 5, 4, 2, 7, 6, 1]

#### **Step-by-Step Breakdown**

1. **Divide:**
   * [8, 3, 5, 4] and [2, 7, 6, 1]
   * [8, 3] & [5, 4] → [8] [3] & [5] [4]
   * [2, 7] & [6, 1] → [2] [7] & [6] [1]
2. **Sort & Merge:**
   * [8] + [3] → [3, 8], [5] + [4] → [4, 5]
   * [3, 8] + [4, 5] → [3, 4, 5, 8]
   * [2] + [7] → [2, 7], [6] + [1] → [1, 6]
   * [2, 7] + [1, 6] → [1, 2, 6, 7]
   * [3, 4, 5, 8] + [1, 2, 6, 7] → \*\*[1, 2, 3, 4, 5, 6, 7, 8]\*\*

### ****Python Implementation****

def merge\_sort(arr):

if len(arr) <= 1:

return arr # Base case: already sorted

mid = len(arr) // 2

left\_half = merge\_sort(arr[:mid]) # Recursively sort left half

right\_half = merge\_sort(arr[mid:]) # Recursively sort right half

return merge(left\_half, right\_half)

def merge(left, right):

result = []

i = j = 0

# Merge two sorted halves

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

# Add any remaining elements

result.extend(left[i:])

result.extend(right[j:])

return result

# Example usage

arr = [8, 3, 5, 4, 2, 7, 6, 1]

sorted\_arr = merge\_sort(arr)

print("Sorted Array:", sorted\_arr)

**Output:**

javascript

CopyEdit

Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8]

### ****Time & Space Complexity****

| **Case** | **Time Complexity** |
| --- | --- |
| **Best Case** | **O(n log n)** |
| **Average Case** | **O(n log n)** |
| **Worst Case** | **O(n log n)** |

* **Space Complexity**: **O(n)** (due to auxiliary space for merging)

### ****Advantages of Merge Sort****

✔ **Stable Sort** (preserves the order of equal elements)  
✔ **Guaranteed O(n log n) Performance** (even in worst case)  
✔ **Good for Linked Lists (O(1) extra space needed for merging)**

### ****Disadvantages of Merge Sort****

❌ **Uses Extra Space** (O(n) auxiliary space for merging)  
❌ **Slower than QuickSort for small datasets**