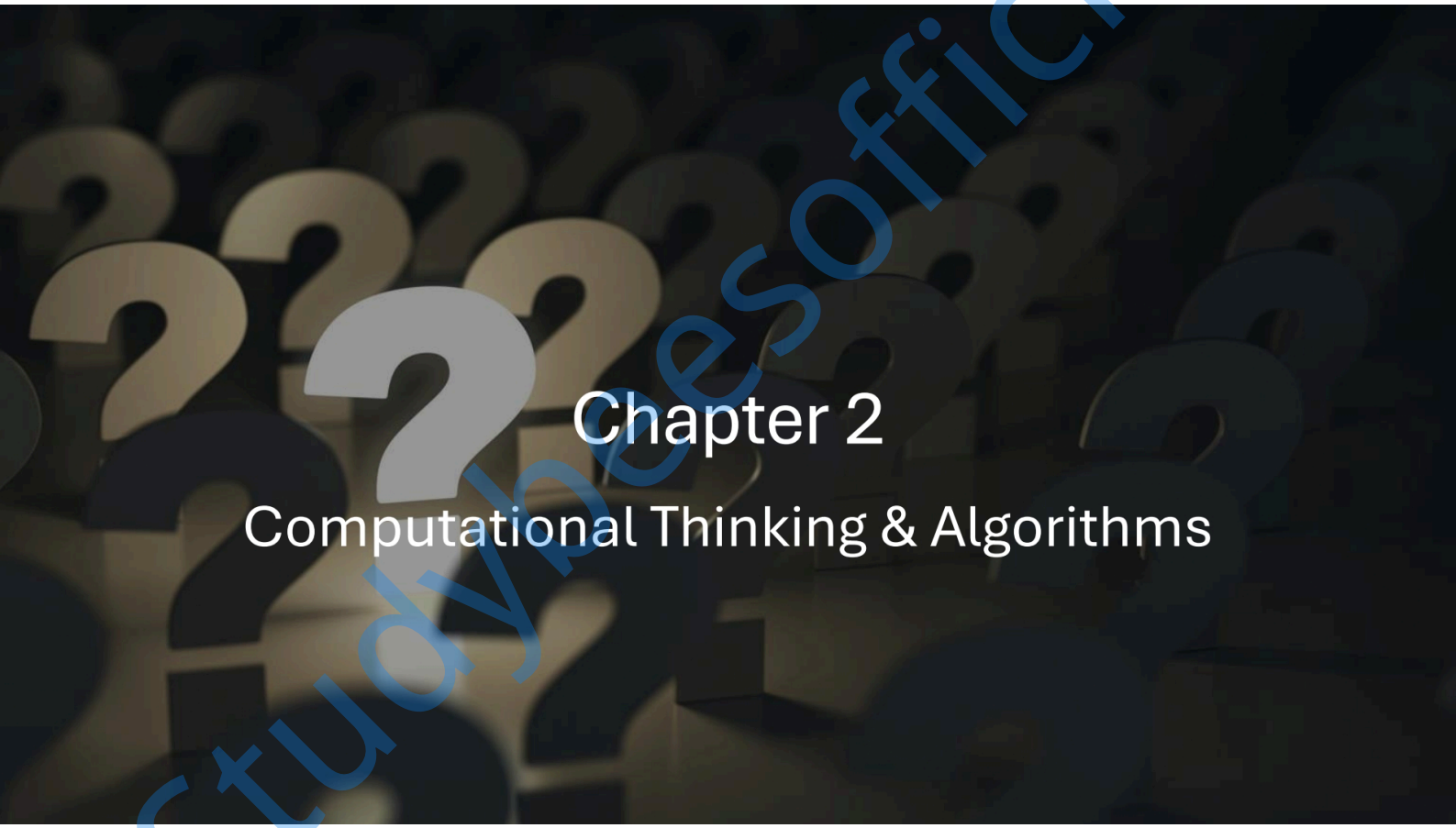
Studybeesofficial

Where Minds Pollinate

Class 12
Computer 🖥️

Chapter 2
Computational thinking
notes



Chapter 2

Computational Thinking & Algorithms

Computational Thinking (CT)

Computational Thinking is a **problem-solving process** that involves breaking down complex problems into manageable parts, identifying patterns, abstracting important information, and designing step-by-step solutions (algorithms).

It is not just for computer science—it helps in solving problems across many fields like science, maths, engineering, and even daily life.

Components of Computational Thinking

Concept	Purpose
Decomposition	Break problem into parts
Pattern Recognition	Spot similarities to reuse solutions
Abstraction	Focus on what really matters
Algorithm	Create a clear solution

🔢 Component	💡 Explanation in Website Context
1. Decomposition	**Break down** the website creation process into smaller tasks: – Choose a website topic/purpose– Design the layout– Create individual pages (Home, About, Contact, etc.)– Add content (text, images, videos)– Code using HTML/CSS/JS– Test functionality– Host it online
2. Pattern Recognition	**Identify similarities** or repeated tasks: – Most pages share a common header and footer– Contact forms usually include name, email, and message fields– Navigation bars behave the same across pages
3. Abstraction	**Focus on essential elements** only: – What should the user see? (User Interface)– What is the goal of the website? (Information, sales, services, etc.)– Ignore unnecessary design details at the beginning (like choosing between hundreds of fonts)
4. Algorithm	**Plan a step-by-step process** to build the site:1. Choose domain name 2. Plan content and structure 3. Design layout wireframes 4. Write code for each page 5. Test and debug 6. Launch the website

🧩 Component	💡 Explanation in Birthday Party Context
1. Decomposition	Break the big task (planning the party) into smaller, manageable parts: – Decide date & time – Make a guest list – Choose a venue – Send invitations – Plan food and drinks – Buy decorations – Arrange games/activities – Order or bake a cake
2. Pattern Recognition	Recognize repeated or common patterns from past parties: – Kids enjoy balloon decorations and games – Most guests prefer finger foods – Cake-cutting usually happens mid-party – Party bags are expected at the end
3. Abstraction	Focus on what's important, ignore unnecessary details: – You need the number of guests, not their full address history – You need to know dietary restrictions, not everyone's favorite dish – You focus on the party essentials, not the color of the walls
4. Algorithm	Create a step-by-step **plan or checklist** to organize the party:

What is a Data Structure?

- A data structure is a way of organizing and storing data efficiently.
- Helps in organizing, processing, and accessing data.
- Examples: Arrays, Linked Lists, Trees, Stacks, Queues, Graphs

Why do we need Data Structures?

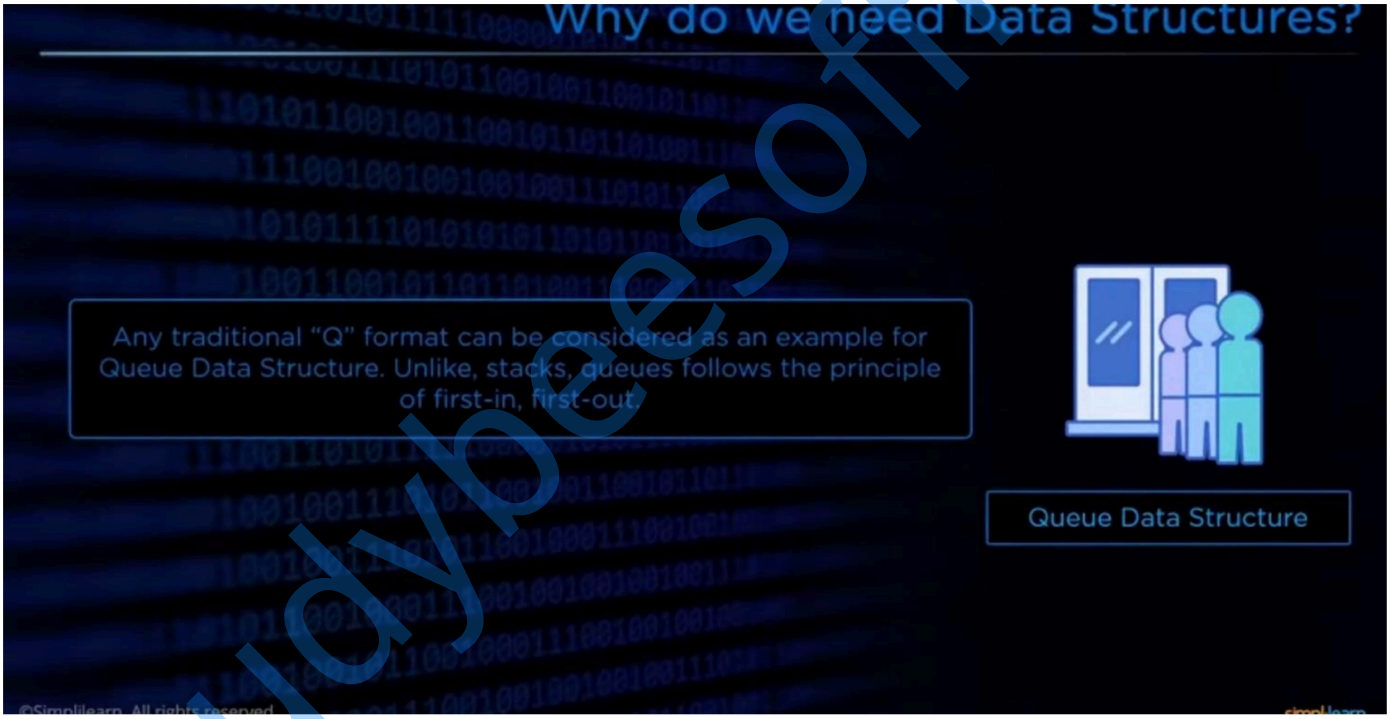
Assume we are looking for the word "Simplilearn" in dictionary, and it begins with the letter "s," so we can search for this word beginning with the letter "s," and this is an example array data structure.

Array Data Structure

Why do we need Data Structures?

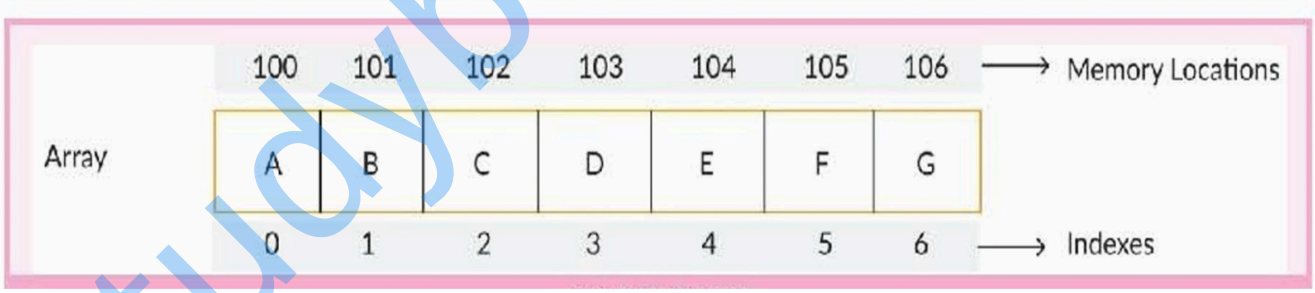
In a stack of books, you must first remove the topmost book before you access your desired book. And if you want to add a new book, you can add it to the top of the book stack.

Stack Data Structure



Array

- Collection of elements in contiguous memory locations.
- Fixed size and same data type.
- Example: int numbers[5] = {10, 20, 30, 40, 50};



Array Types:

1-Dimensional Array (1-D Array)

- Single row of data.
- Access using array[index].
- Example: marks = [90, 85, 88]

2-Dimensional Array (2-D Array)

- Like a table: rows and columns.
- Access using array[row][column].
- Example: matrix = [[1,2], [3,4]]

Linked List

- Linear collection of nodes, each pointing to the next.
- Types: Singly, Doubly, Circular
- Advantage: Dynamic size, efficient insertion/deletion.

Singly Linked List

- Each node contains data and pointer to next node.
- Structure: Node1 -> Node2 -> NULL
- Use Case: Music playlists.

Linked List

Doubly Linked List

- Each node has pointers to next and previous nodes.
- Structure: NULL <- Node1 <-> Node2 -> NULL
- Use Case: Browser history, Undo/Redo functionality, Navigation systems

Circular Linked List

- Last node points back to the first node.
- Can be singly or doubly circular.
- Use Case: In a **clock**, after 12 comes 1 again, Multiplayer games, Circular Race Track, Round Table Discussion, Multimedia Playlist with Previous/Next

Stack (LIFO)

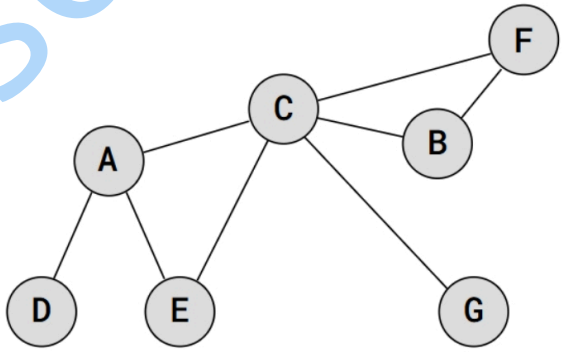
- Follows Last In First Out principle.
- Basic operations we can do on a stack are:
  - **Push:** Adds a new element on the stack.
  - **Pop:** Removes and returns the top element from the stack.
  - **Peek:** Returns the top (last) element on the stack.
  - **isEmpty:** Checks if the stack is empty.
  - **Size:** Finds the number of elements in the stack.
- **Use Case**: Undo feature

Queue (FIFO)

- Follows First In First Out principle.
- Basic operations we can do on a queue are:
  - **Enqueue:** Adds a new element to the queue.
  - **Dequeue:** Removes and returns the first (front) element from the queue.
  - **Peek:** Returns the first element in the queue.
  - **isEmpty:** Checks if the queue is empty.
  - **Size:** Finds the number of elements in the queue.
- **Use Case:** Print queue, task scheduling.

Graph

- A **graph** is a data structure made up of:
- **Vertices (nodes)** — data points
- **Edges (connections)** — links between vertices
- Graphs are used to model relationships between pairs of objects — like friends in a social network, routes between cities, or webpage links.



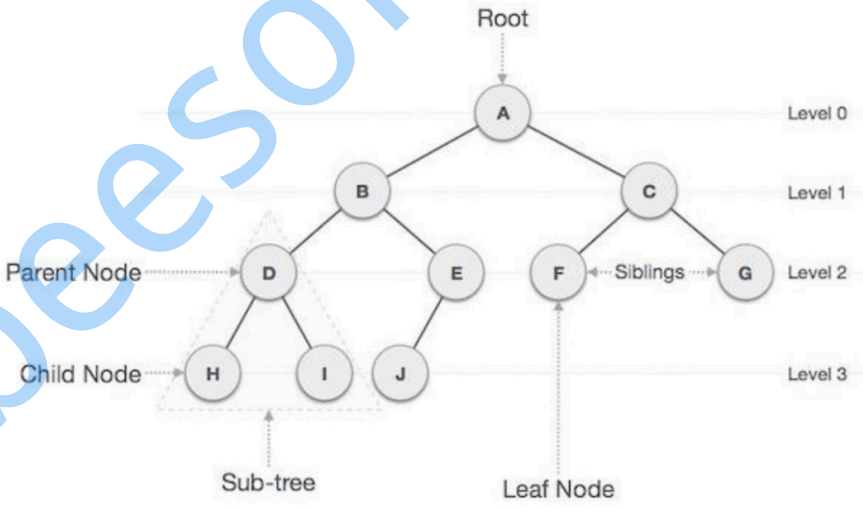
Operation	Description
Add Vertex	Add a new node to the graph
Add Edge	Create a connection between two vertices
Remove Vertex	Delete a node and all connected edges
Remove Edge	Delete a specific connection
Search (Traversal)	Visit nodes using DFS (Depth First Search) or BFS (Breadth First Search)
Check Path	Determine if there is a path between two nodes

Types:

Type	Description	Example
Connected Graph	There is a path between every pair of nodes	A delivery network
Directed Graph (Digraph)	Edges have direction (A → B is not the same as B → A)	Twitter following

Tree

- Non-linear structure with root and child nodes.
- **Binary Tree**: Each node has up to two children.
- **Use Case**: File systems, decision trees.



Data Structure	Access Pattern	Real-Life Example
Stack	LIFO (Last In First Out)	Undo action, plates, books
Queue	FIFO (First In First Out)	Bus line, printer queue, orders
Graph	Many-to-many links	Maps, Social networks, flight routes
Tree	One-to-many (hierarchy)	Family tree, folder system

Data Structure	Use Case
Array	Student marks
Linked List	Playlists, memory mgmt
Stack	Undo operations
Queue	Print jobs
Tree	File directories
Graph	Networks, maps

Feature	Tree	Graph
Definition	A hierarchical structure with parent-child relationships	A set of nodes connected by edges
Cycles	Cannot contain cycles	Can contain cycles
Connectivity	All nodes are connected and there is exactly one path between any two nodes	May or may not be connected
Root Node	Has a root node (starting point)	No concept of a root node
Direction	Usually directed (parent → child)	Can be **directed** or **undirected**
Parent-Child Relation	Yes (one parent, many children)	No strict parent-child relation
Example	File system, family tree, organization chart	Road network, social network, web pages linking each other
Traversal	Tree traversal (DFS, BFS, Preorder, Inorder, Postorder)	Graph traversal (DFS, BFS)

Tree Traversal

- There are three different types of DFS traversals:
- pre-order
- in-order
- post-order

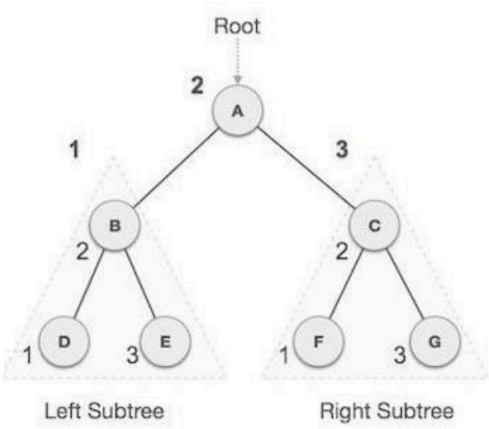
Binary Trees

- A Binary Tree is a type of tree data structure where each node can have a maximum of two child nodes, a left child node and a right child node.

Condition for Binary Tree	Description
Max 2 children	Each node can have **0, 1, or 2** child nodes only.
Unique paths	Each child has only **one parent** (no cycles).
One root node	Only one node (topmost) has **no parent** — it's the root.

In-order Traversal

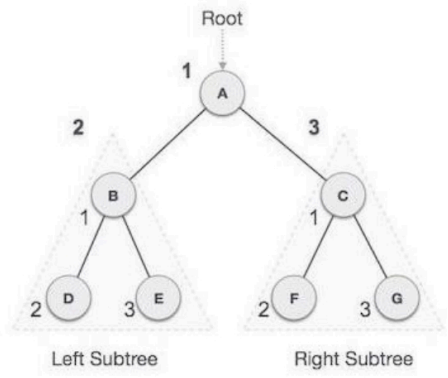
- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.
- We should always remember that every node may represent a subtree itself.



- The output of in-order traversal of this tree will be –
- **D B E A F C G**

Pre-order Traversal

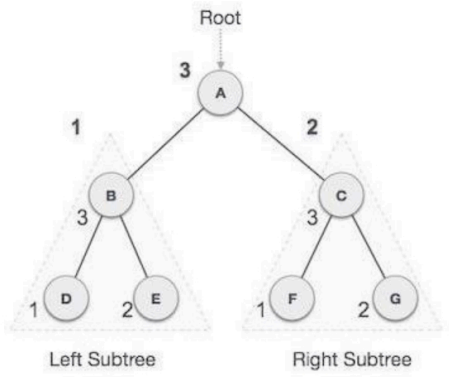
- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



- The output of pre-order traversal of this tree will be –
- **A → B → D → E → C → F → G**

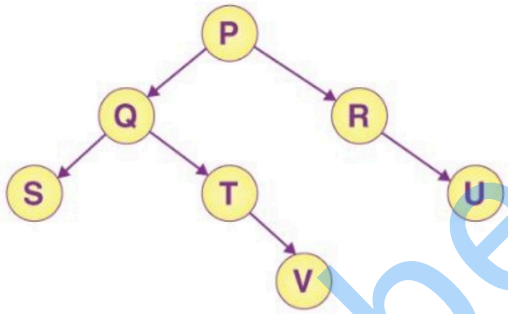
Post-order Traversal

- In this traversal method, the root node is visited last, hence the name. First, we traverse the left subtree, then the right subtree and finally the root node.

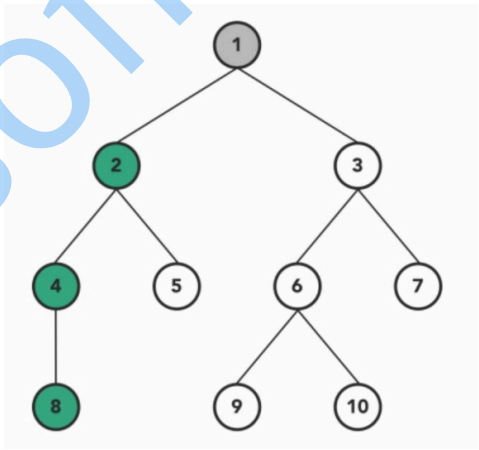


- The output of post-order traversal of this tree will be
- **D → E → B → F → G → C → A**

Examples:



Pre: P->Q->S->T->V->R->U
In: S->Q->T->V->P->R->U
Post: S->V->T->Q->U->R->P



Pre: 1->2->4->8->5->3->6->9->10->7
In: 8->4->2->5->1->9->6->10->3->7
Post: 8->4->5->2->9->10->6->7->3->1

BFS (Breadth-First Search)

Idea: Explore the graph **level by level** (like waves spreading out).
Steps:
1.Start from a source node.
2.Visit all its immediate neighbors first.
3.Then visit neighbors of neighbors, and so on.

Applications of BFS:

- Shortest path in unweighted graphs (like Google Maps).
- Finding connected components.
- Web crawlers (search engines).

DFS (Depth-First Search)

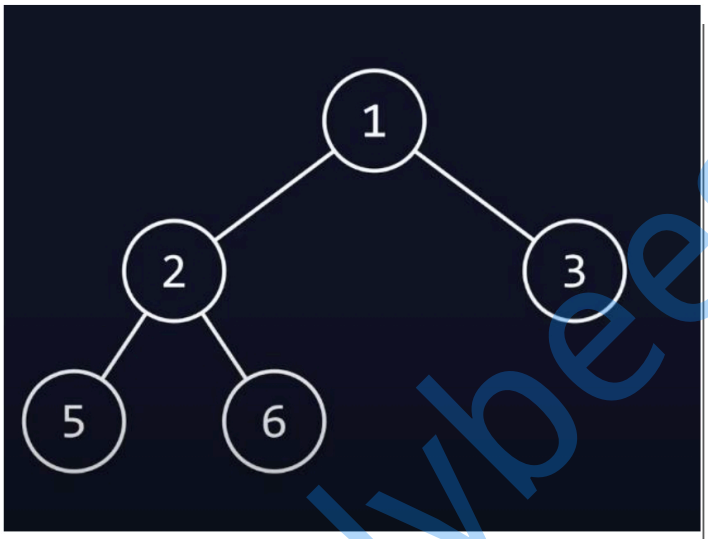
Idea: Explore the graph as **deep as possible** along one branch before backtracking.

Steps:

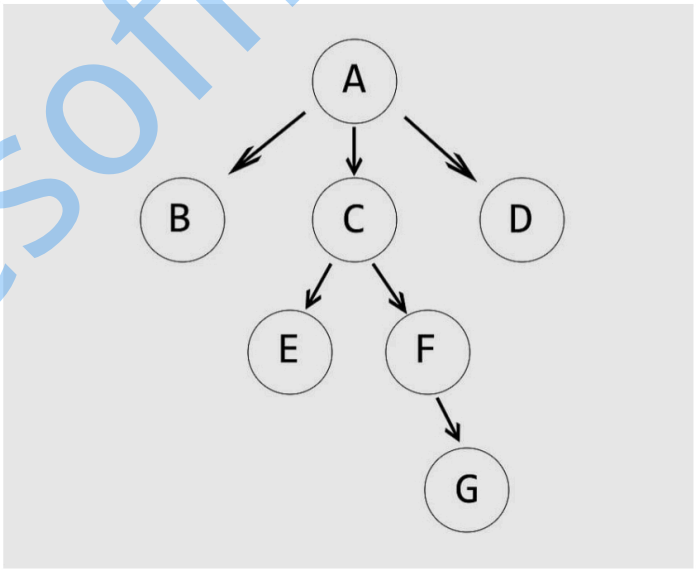
1.Start from a source node.
2.Go to one neighbor, then that neighbor's neighbor, and so on.
3.If stuck, backtrack and explore unvisited nodes.

Applications of DFS:

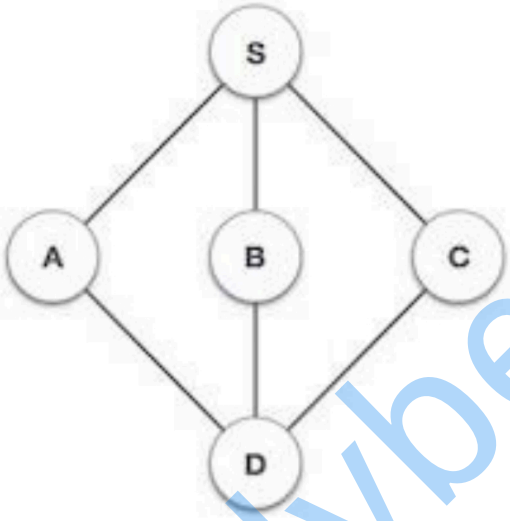
- Detecting cycles in a graph.
- Solving puzzles/mazes.



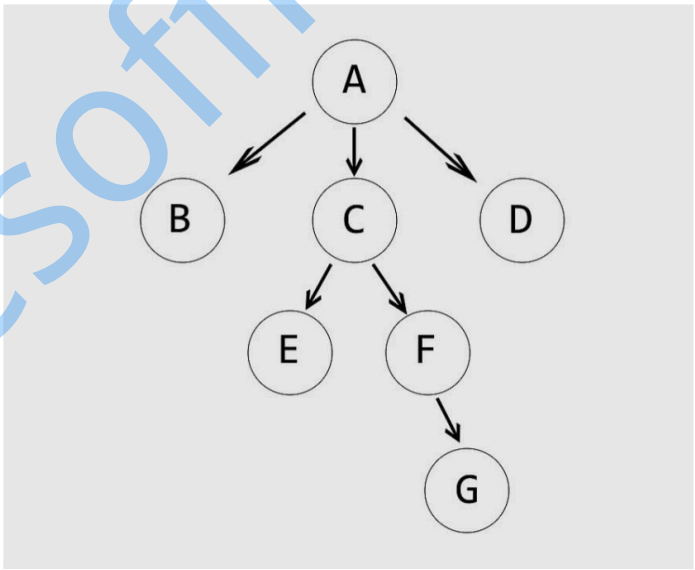
BFS: 1 2 3 5 6
DFS: 1 2 5 6 3



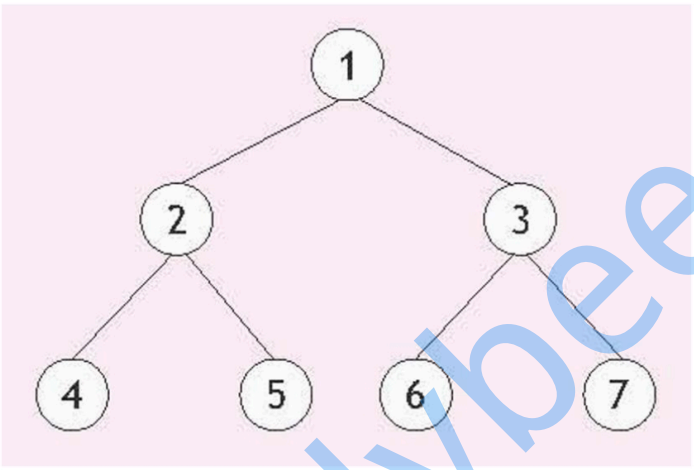
BFS: A B C D E F G
DFS: A B C E F G D



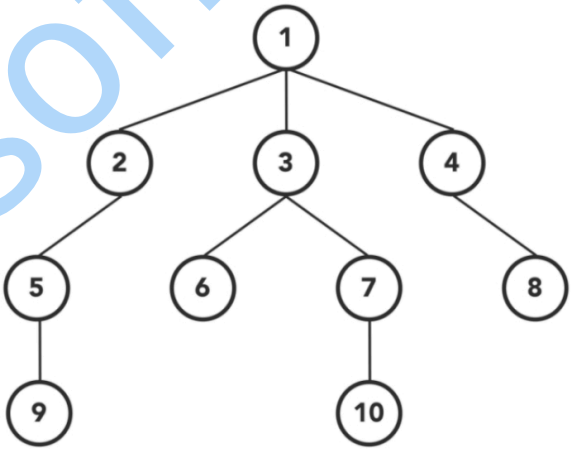
BFS: S A B C D
DFS: S A D B C



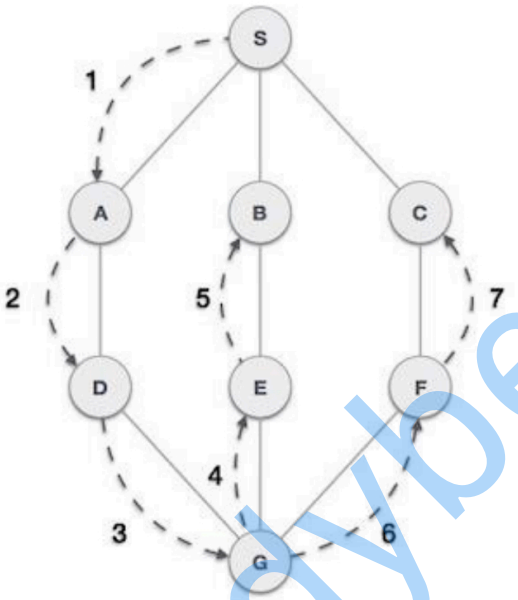
BFS: A B C D E F G
DFS: A B C E F G D



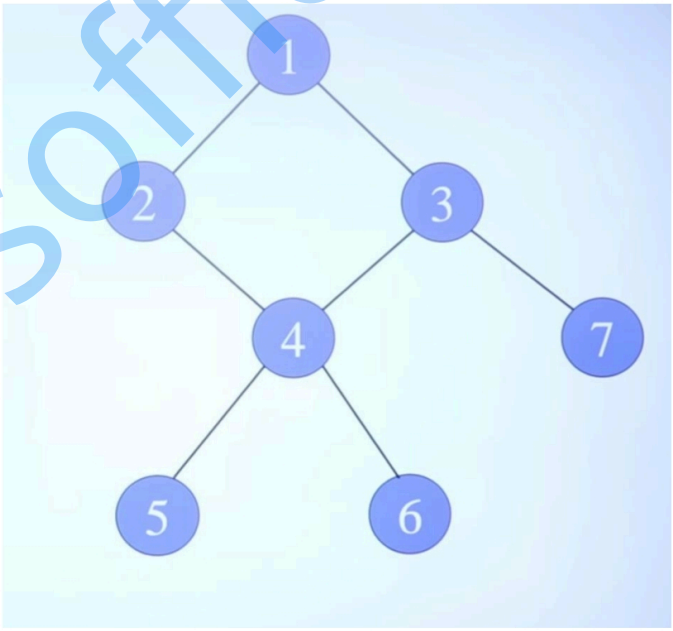
BFS: 1 2 3 4 5 6 7
DFS: 1 2 4 5 3 6 7



BFS: 1 2 3 4 5 6 7 8 9 10
DFS: 1 2 5 9 3 6 7 10 4 8



BFS: S A B C D E F G
DFS: S A D G E B F C



BFS: 1 2 3 4 7 5 6
DFS: 1 2 4 5 6 3 7

Evaluating Computational Solutions

1. Correctness

- A solution is **correct** if it produces the right **output for all valid inputs**.
- It must **meet all problem requirements**.
- **Example:**
  A program that sorts a list must always return a correctly sorted version of the list.

Evaluating Computational Solutions

2. Clarity

- Clarity means how **easy it is to read, understand, and maintain** the code.
- Good code has:
  ✓ Clear variable names
  ✓ Comments where needed
  ✓ Simple, modular structure
- **Why it matters:**
  Easier for debugging, teamwork, future improvements.

Clarity:

- In Python, the **ternary operator** (also called **conditional expression**) lets you write an **if-else** statement in a **single line**.
- Syntax:

```
value_if_true if condition else value_if_false
```

```
age = 18
status = "Adult" if age >= 18 else "Minor"
print(status)
```

```python
print("List comprehension result:")
print([x for x in range(1,101) if x % 10 == 0])

# The list comprehension above accomplishes the same result as
# the long form version of the code:
print("Long form code result:")
my_list = []
for x in range(1,101):
    if x % 10 == 0:
        my_list.append(x)
print(my_list)

```

Evaluating Computational Solutions

3. Efficiency

- Efficiency measures **how well a solution uses resources**:
- **Time Complexity:** How fast it runs
- **Space Complexity:** How much memory it uses.

(**Auxiliary space** = memory for temporary variables, extra data structures, recursion stack, etc.)

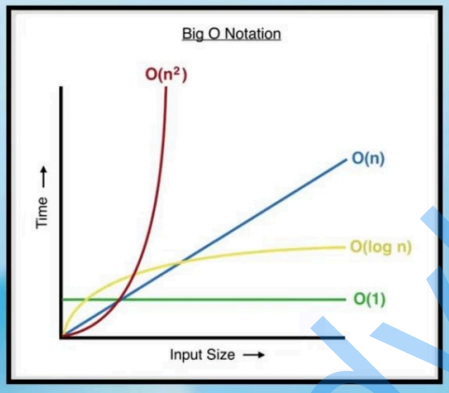
Time & Space Complexity

- **Example:**
  If you ask a friend to search for a word in a book:
- Time complexity → How long it takes to find it.
- Space complexity → How many extra pages or sticky notes they use while searching.
- If n=5, how many times does the loop run? → **5 times**
- If n=1000, how many times? → **1000 times**
- Then say: **This grows directly with n → O(n)**

Demonstrate different complexities with real-life examples

Complexity	Real-life analogy
O(1) (Constant time)	Checking first page of a book
O(n) (Linear)	Reading every page in a book
O(n^2) (Quadratic)	Comparing every student in class with every other student
O(log n) (Logarithmic)	Guessing a number between 1–100 by halving the range each time

Big O notation

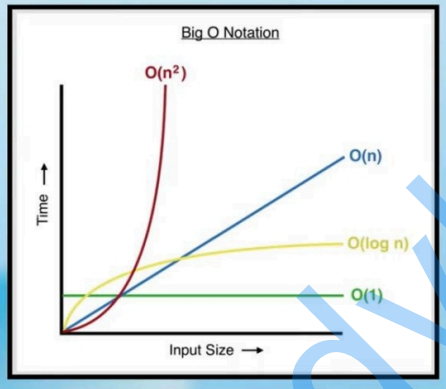


O(1) – Constant Time
The time does not change with input size.
Example: Getting 1 item from a list, no matter how long the list is.

O(log n) – Logarithmic Time
Time increases slowly as input gets bigger.
Example: Binary Search (cutting the list in half each time).

O(n) – Linear Time
Time increases directly with input size.
Example: Checking each item in a list using a loop.

Big O notation



O(n log n) – Linearithmic Time

A bit slower than linear, but faster than quadratic.

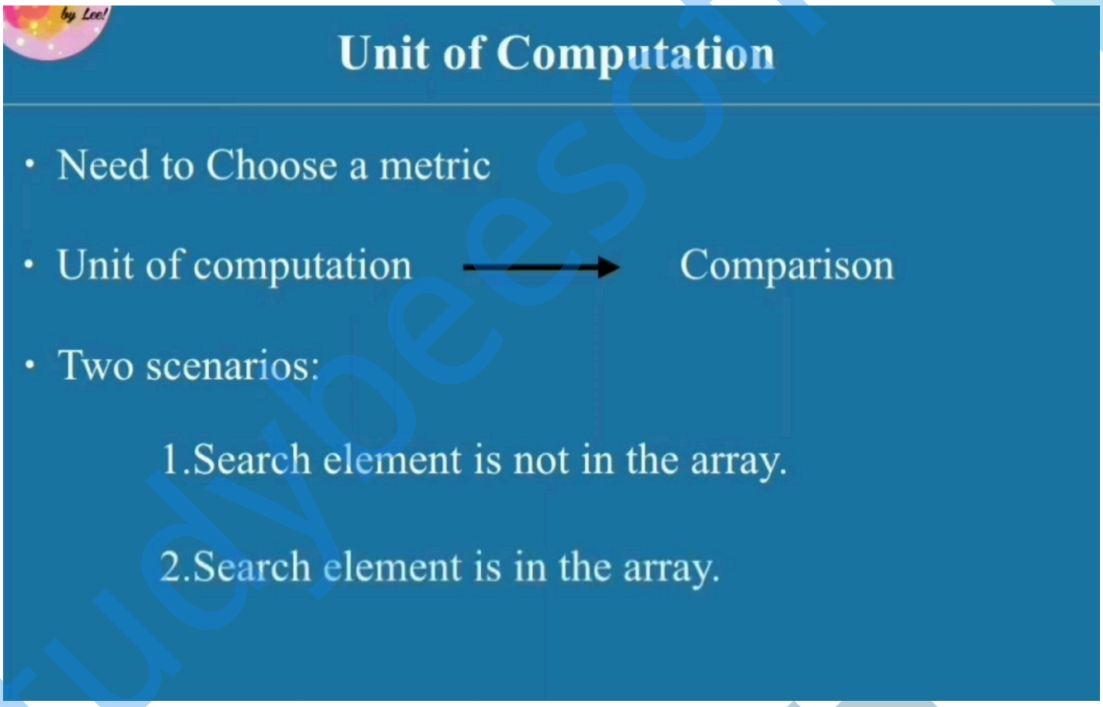
Example: Merge Sort or Quick Sort (used for fast sorting).

O(n²) – Quadratic Time

Time increases very fast with input size.

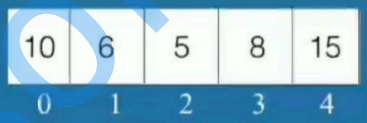
Example: Nested loops (like comparing every item with every other item).

Linear Search



Element is not Present

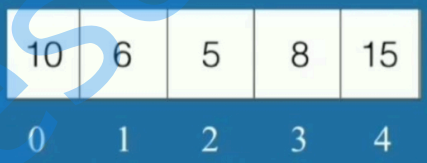
- Assume an unsorted array.



- Size n = 5 & search element = 4.
- Entire array has been searched.
- Number of comparisons = n. (n = Array size)
- Best case: n.
- Worst case: n.
- Average case: n.

Element is Present

- Assume an unsorted array.
- Size n = 5.



- Best case : One comparison.
- Worst case: n comparisons. (n = Array size)
- Average case: **(n/2) comparison**

Analysis of Binary Search Algorithm

Analysis of Binary Search Algorithm

- **Basic unit of computation:**

  Operation to complete a given task.

- **Binary Search Algorithm:**

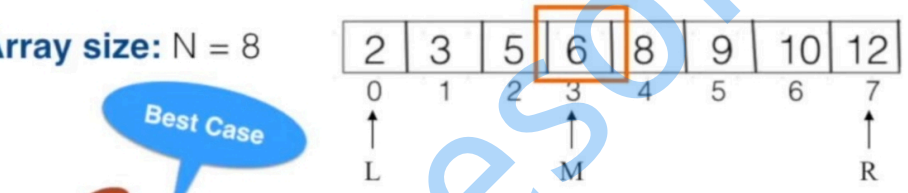
  Comparision.

Minimum No.of Comparision. ⟶ Best case

Maximum No.of Comparision. ⟶ Worst case

Analysis of Binary Search Algorithm

Array size: N = 8



Search element is equal
to the very first middle element.

Search element = 6

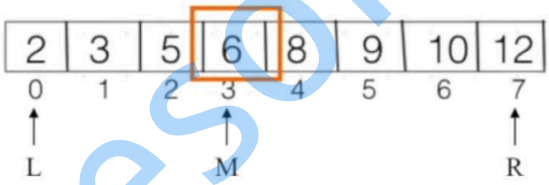
$$m = \frac{L + R}{2}$$

$$m = \frac{0 + 7}{2}$$

$$m = 3$$

Analysis of Binary Search Algorithm

Array size: N = 8



O(1)

Search element is equal
to the very first middle element.

Search element = 6

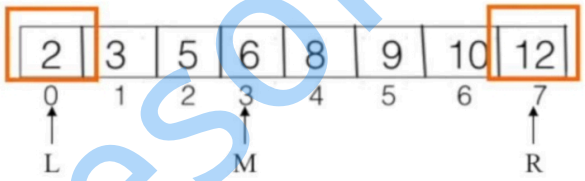
$$m = \frac{L + R}{2}$$

$$m = \frac{0 + 7}{2}$$

$$m = 3$$

Analysis of Binary Search Algorithm

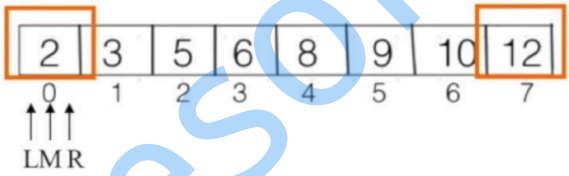
Array size: N = 8



Search element is present at either beginning of the array or end of the array.

Analysis of Binary Search Algorithm

Array size: N = 8



Search element = 2

Search element = Middle element

Search element is present at either beginning of the array or end of the array.

3rd comparision:

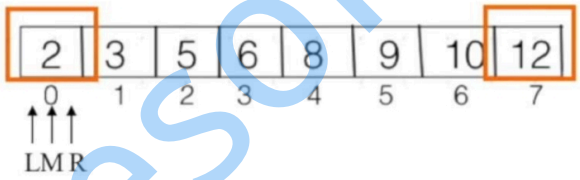
$$m = \frac{L + R}{2}$$

$$m = \frac{0 + 0}{2}$$

$$m = 0$$

Analysis of Binary Search Algorithm

Array size: N = 8



$8/2 \rightarrow 4 = 1$
$4/2 \rightarrow 2 = 2$
$2/2 \rightarrow 1 = 3$

$8/2^1 \rightarrow 4 = 1$
$8/2^2 \rightarrow 2 = 2$
$8/2^3 \rightarrow 1 = 3$

Search element is present at either beginning of the array or end of the array.

$N/2^k = 1$

Analysis of Binary Search Algorithm

Array size: N

$N/2^k = 1$

$N = 2^k$

$\log_2 N = k \log_2 2$

$k = \log_2 N$

Search element is not present in the array.

Bubble Sort:

A simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order.

Time Complexity:

- Best Case: O(n) (already sorted)
- Worst and Average Case: $O(n^2)$
- Space Complexity: O(1)

Bubble Sort

Algorithmic Design

1. Start from the first element of the array.
2. Compare the first element with the adjacent element.
   - If the first element is greater than the adjacent element, swap them.
   - If the first element is smaller or equal, do nothing and move to the next pair of elements.
3. Move to the next pair of elements and repeat the comparison and swapping process.
   - Continue this process until you reach the end of the array for the first Iteration.
4. After the first Iteration, the largest element is guaranteed to be at the end of the array.
   - You don't need to consider this element in the subsequent passes since it is already in its correct position.
5. Repeat the above steps for the remaining elements (excluding the ones that are already sorted).
   - In each Iteration, the next largest unsorted element will be placed in its correct position.
6. Continue this process until the entire array is sorted.
   - The number of Iterations required is one less than the total number of elements in the array.

Insertion Sort

Insertion Sort is a sorting algorithm that builds the sorted array one element at a time. It assumes that the first element is already sorted and then inserts each subsequent element into its correct position by comparing it with the previous elements. If necessary, larger elements are shifted to the right to make space for the new element. This process continues until the entire array is sorted.

Time Complexity:

- Best Case (Already Sorted Array): O(n)
- Worst Case :$O(n^2)$

Algorithmic Deign

1. Start with the second element:

   Begin with the second element of the array (assuming the first element is already "sorted").

2. Pick the current element:

   Pick the current element and remember it.

3. Compare with sorted elements:

   Compare the current element with the elements in the "sorted" part of the array.

4. Shift larger elements to the right:

   If the current element is smaller than the elements in the "sorted" part, shift those larger elements to the right to create space.

5. Insert the current element:

   Insert the current element into its correct position in the "sorted" part.

6. Repeat for each element:

   Repeat these steps for each element in the array, gradually expanding the "sorted" part.

Merge Sort

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Divide: Splitting the array into two halves

Conquer: Recursively sort each half (sub-array), until each half contains only one element.

Combine: Merge the sorted sub-arrays to produce a single sorted array.

Time Complexity:

- Best, Worst, and Average Case: **O(n log n)**

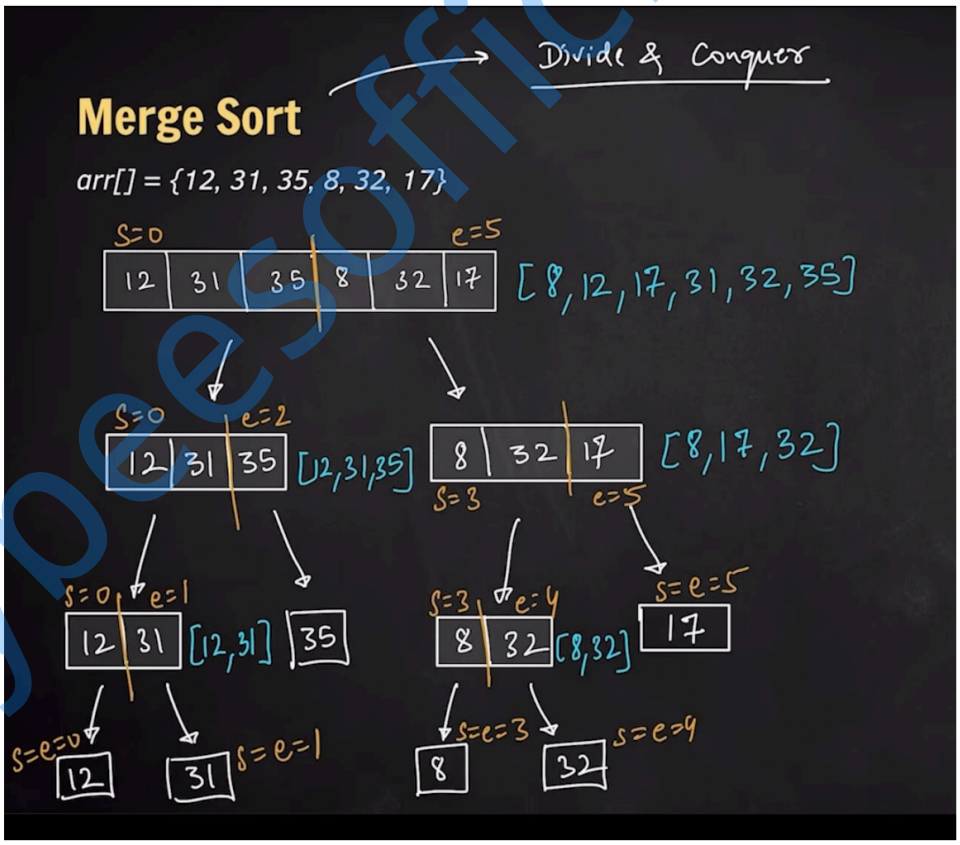
Merge Sort

Step 1: If it is only one element in the list, consider it already sorted, so return.

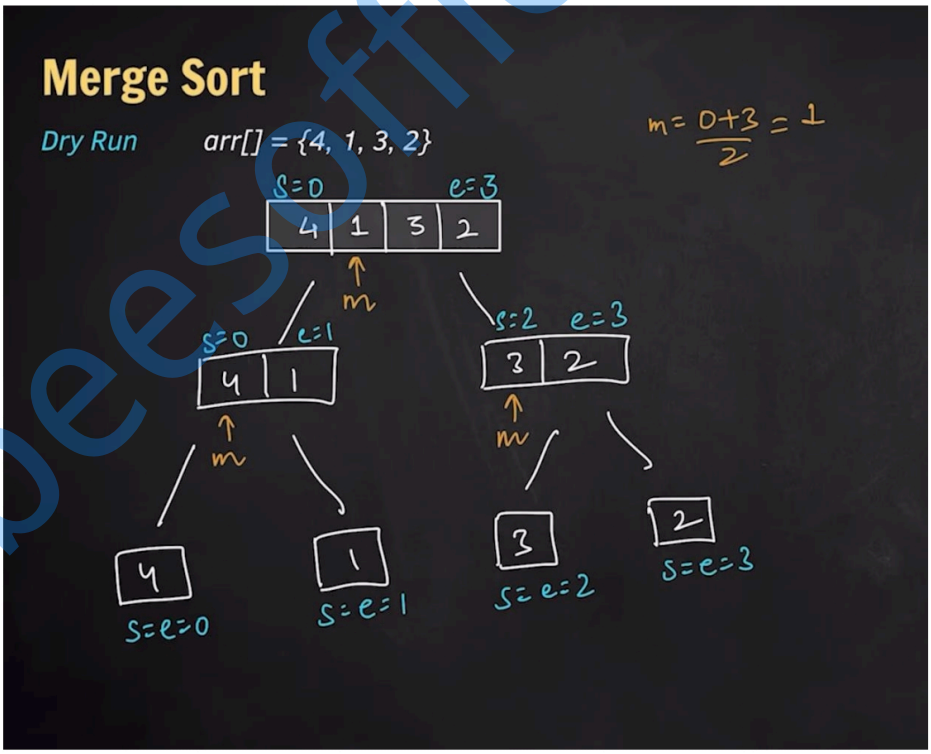
Step 2: Divide the list recursively into two halves until it can no more be divided.

Step 3: Merge the smaller lists into new list in sorted order.

Merge Sort

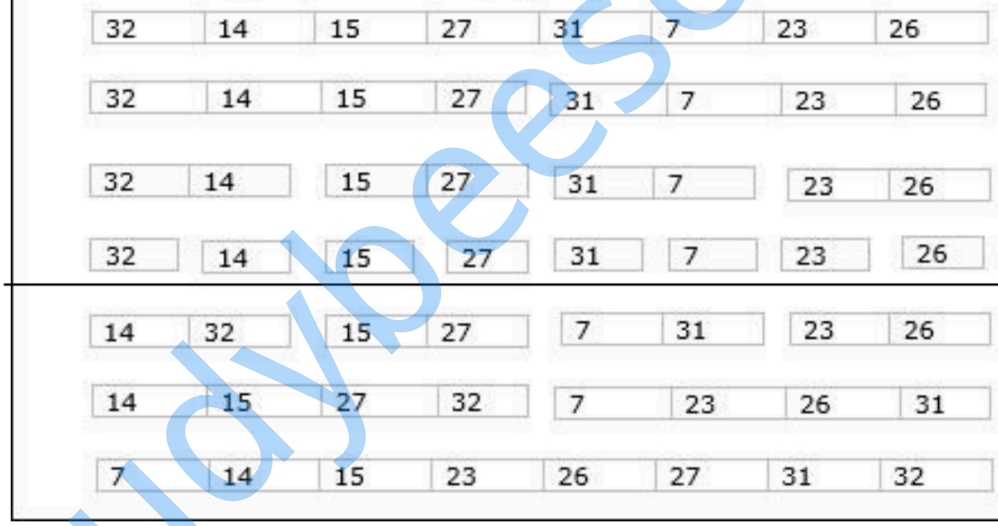


Merge Sort



Merge Sort

Divide and Merge operations step by step:



Sorting Algorithm	Best Case (Time)	Average Case (Time)	Worst Case (Time)	Space Complexity	Use Cases
Bubble Sort	**O(n)** (already sorted list)	**O(n^2)**	**O(n^2)**	**O(1)** (in-place)	- When teaching basic sorting concepts- For very small datasets- Rarely used in practice due to inefficiency
Insertion Sort	**O(n)** (already sorted list)	**O(n^2)**	**O(n^2)**	**O(1)** (in-place)	- Small datasets- Nearly sorted data (performs very well.
Merge Sort	**O(nlogn)**	**O(nlogn)**	**O(nlogn)**	**O(n)** (extra memory for merging)	- Large datasets- Linked lists (works efficiently)- External sorting (sorting data stored on disk)- Stable sorting is required