Nguyễn Đức Đông

DSA

Thêm mô tả ngắn tại đây

- CREATE A DESIGN SPECIFICATION FOR DATA STRUCTURES, EXPLAINING THE VALID OPERATIONS THAT CAN BE CARRIED OUT ON THE STRUCTURES.

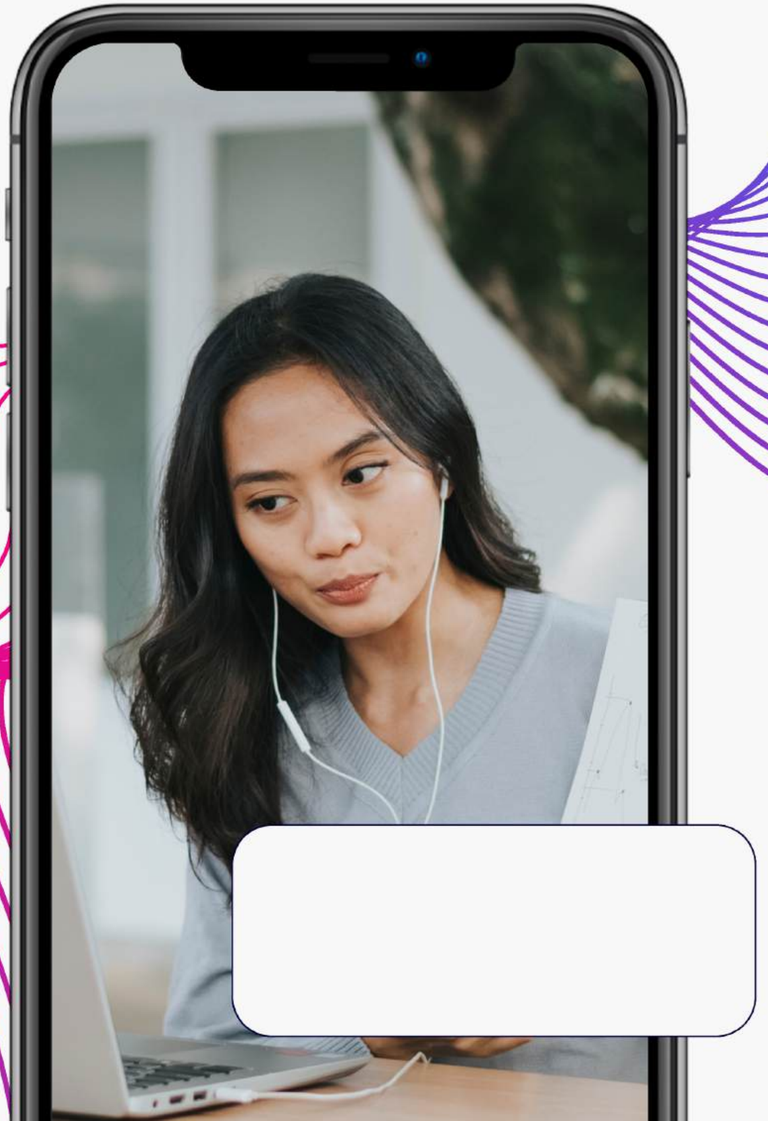
Dentify the Data Structures

- Arrays
- Linked Lists
- Stacks
- Queues
- Hash Tables
- Trees (Binary Trees, AVL Trees, etc.)
- Graphs

Define the Operations

Quay lại chương trình làm việc

- Push: Add an element to the top of the stack.
- Pop: Remove the top element from the stack.
- Peek: Return the top element without removing it.
- IsEmpty: Check if the stack is empty.



Specify Input Parameters

- **Push(value):**

Input: value (the element to be added to the stack)

- **Pop():**

Input: None (removes the top element)

- **Peek():**

Input: None (returns the top element)

Define Pre- and Post-conditions

Example: Stack Operations

- Push(value)
  - Pre-condition: The stack can accommodate more elements (no overflow).
  - Post-condition: The stack size increases by one, and value is at the top.
- Pop()
  - Pre-condition: The stack is not empty.
  - Post-condition: The stack size decreases by one, and the top element is removed.

Example: Queue Operations

- Enqueue(value)
  - Pre-condition: There is space in the queue.
  - Post-condition: The queue size increases by one, and value is at the end.
- Dequeue()
  - Pre-condition: The queue is not empty.
  - Post-condition: The queue size decreases by one, and the front element is removed.

Discuss Time and Space Complexity

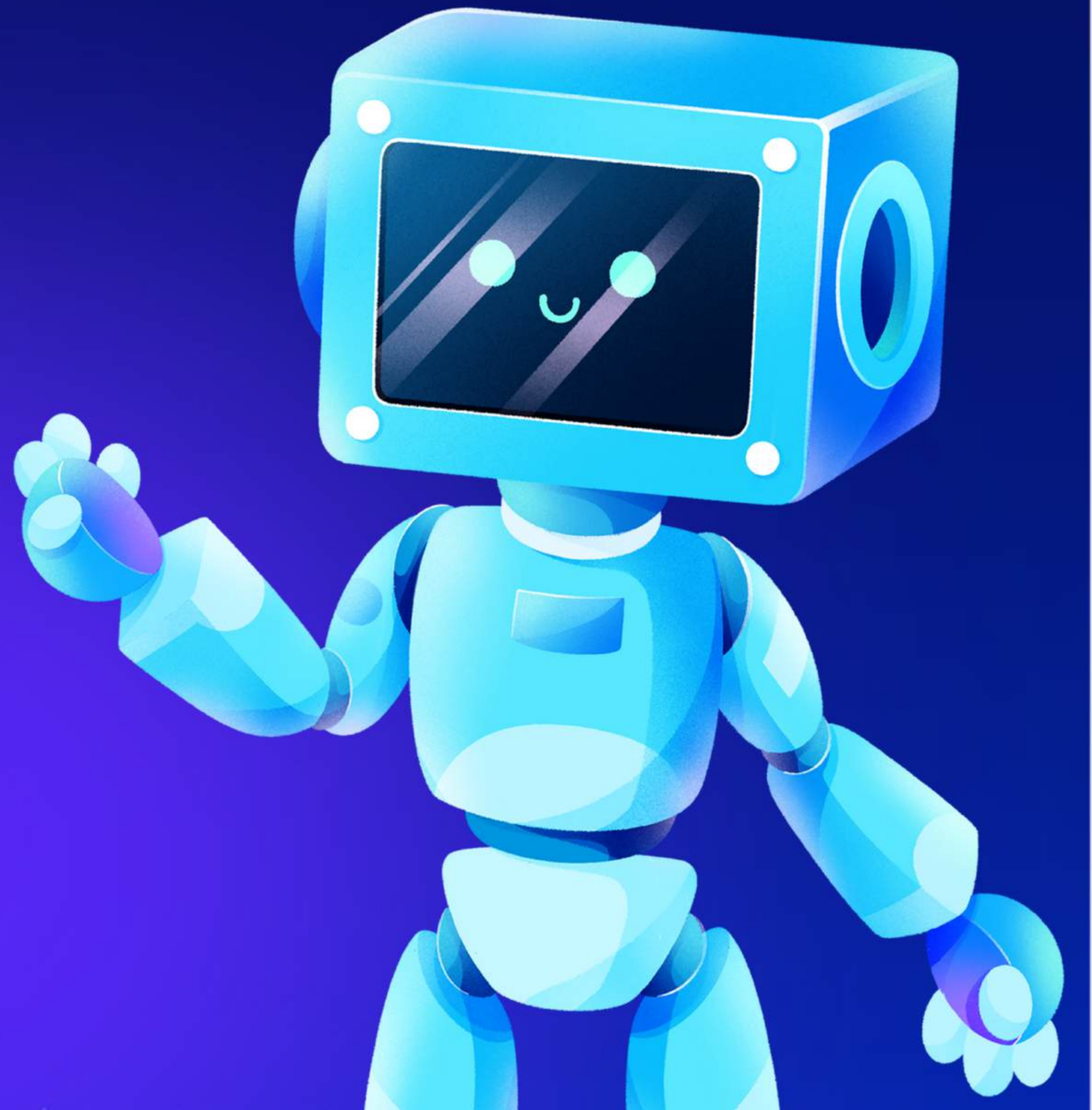
Example: Stack

- Push:
  - Time Complexity: O(1)
  - Space Complexity: O(n) (n is the number of elements in the stack)
- Pop:
  - Time Complexity: O(1)
  - Space Complexity: O(n)

Example: Queue

- Enqueue:
  - Time Complexity: O(1)
  - Space Complexity: O(n)
- Dequeue:
  - Time Complexity: O(1)
  - Space Complexity: O(n)

DETERMINE THE OPERATIONS OF A MEMORY STACK AND HOW IT IS USED TO IMPLEMENT FUNCTION CALLS IN A COMPUTER.



Define a Memory Stack

A Memory Stack, commonly referred to as the call stack, is a special type of data structure used by most modern programming languages for managing function calls, local variables, and control flow during program execution. The stack operates on the Last In, First Out (LIFO) principle, meaning the last function called is the first one to be finished and popped from the stack.
Each function call creates a stack frame (or activation record), which stores the function's:

- Return address: The point in the program where control should return after the function finishes.
- Function parameters: Arguments passed to the function.
- Local variables: Variables that are declared within the function.
- Saved registers: CPU registers, including the program counter, to preserve the program state.

The stack grows as new function calls are made and shrinks as functions return.

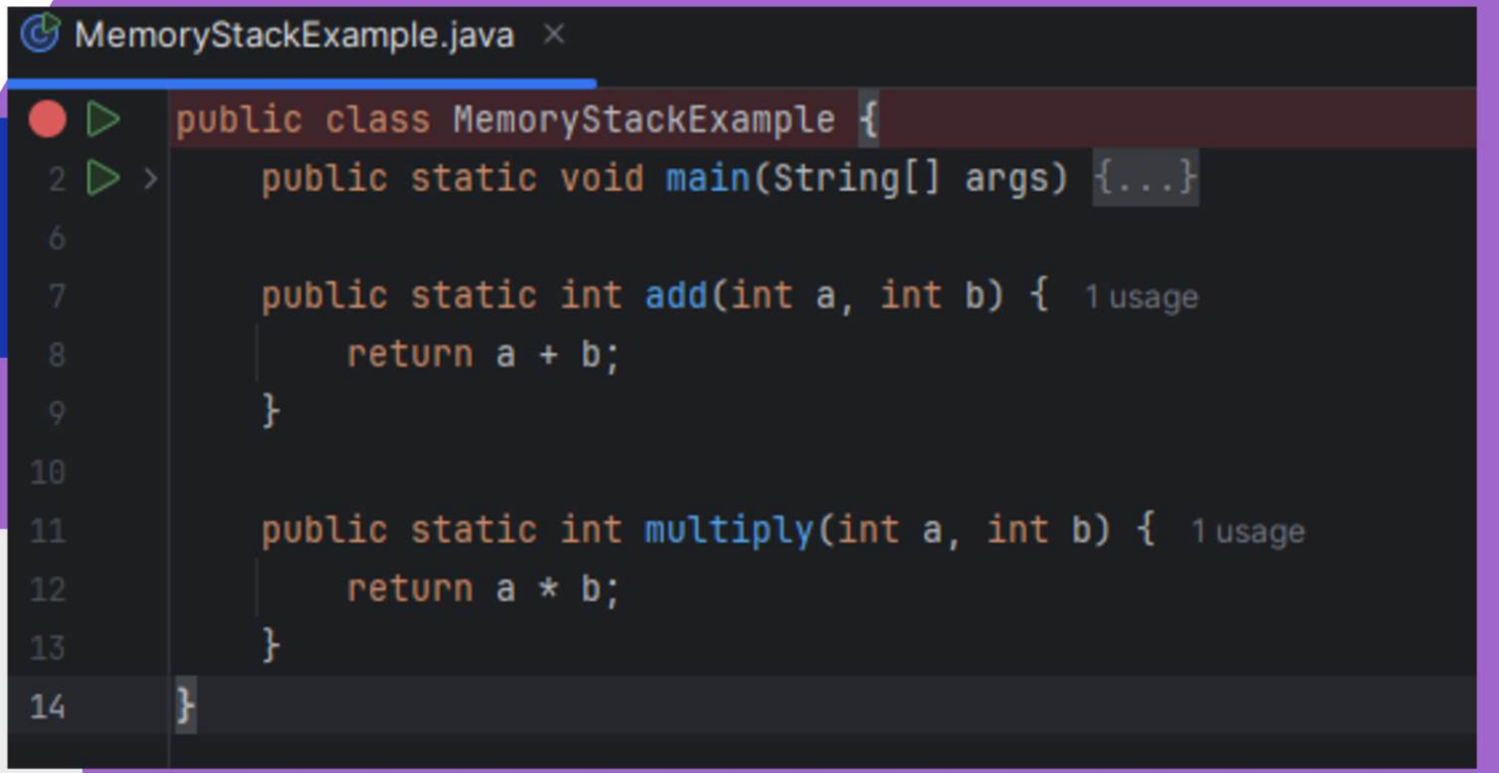
Identify Operations on the Memory Stack

Key operations on the memory stack include:

- Push:
  - Adds a new stack frame onto the stack when a function is called.
  - Contains return address, function parameters, local variables, and saved registers.
- Pop:
  - Removes the topmost stack frame when the function returns, restoring the previous function's state.
- Peek:
  - Retrieves information from the topmost stack frame without removing it, often used for debugging or checking the current function context.

Công nghệ A&T

Function Call Implementation



Discuss the Importance of Memory Stack

The Memory Stack is crucial for managing function calls, particularly recursive ones, by tracking the order in which functions are called and returned. It stores local variables in each function's stack frame, ensuring proper scope and isolation. The stack is memory-efficient due to its bounded size and automatic LIFO allocation and deallocation. It also manages control flow by keeping track of return addresses.

However, its fixed size can lead to stack overflow errors during deep recursion, and automatic deallocation makes it challenging to preserve data across function calls without using external memory like the heap.

xample of Stack Overflow

StackOverflowExample.java

```java
public class StackOverflowExample {
    public static void main(String[] args) {
        recurse();
    }

    public static void recurse() {  2 usages
        // This function keeps calling itself, causing a stack overflow
        recurse();
    }
}
```

In this case, the stack keeps growing as more and more calls to recurse() are made, eventually resulting in a StackOverflowError.

Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue.

Introduction FIFO

A First In, First Out (FIFO) queue is a type of data structure where the first element added to the queue is the first one to be removed. This principle makes it ideal for scenarios where order matters, such as scheduling tasks or processing requests in the order they arrive.
Key operations include:

- Enqueue: Adding an element to the back of the queue.
- Dequeue: Removing the element from the front of the queue.
- Peek: Viewing the element at the front without removing it.

Define the Structure

A FIFO queue can be implemented using different underlying data structures like arrays or linked lists. The choice of implementation affects the efficiency of enqueue and dequeue operations.

Main.java × ArrayQueue.java

```java
public class Main {
    public static void main(String[] args) {
        ArrayQueue queue = new ArrayQueue(5);
        queue.enqueue( i: 10);
        queue.enqueue( i: 20);
        queue.enqueue( i: 30);

        System.out.println("Front element: " + queue.peekFront()); // Output: 10

        System.out.println("Dequeued: " + queue.dequeue()); // Output: 10
        System.out.println("Dequeued: " + queue.dequeue()); // Output: 20
    }
}
```

Compare the performance of two sorting algorithms.

Introduction to Two Sorting Algorithms

Let's compare Merge Sort and Quick Sort, two well-known sorting algorithms that use the divide and conquer strategy but have different performance characteristics.

Merge Sort:

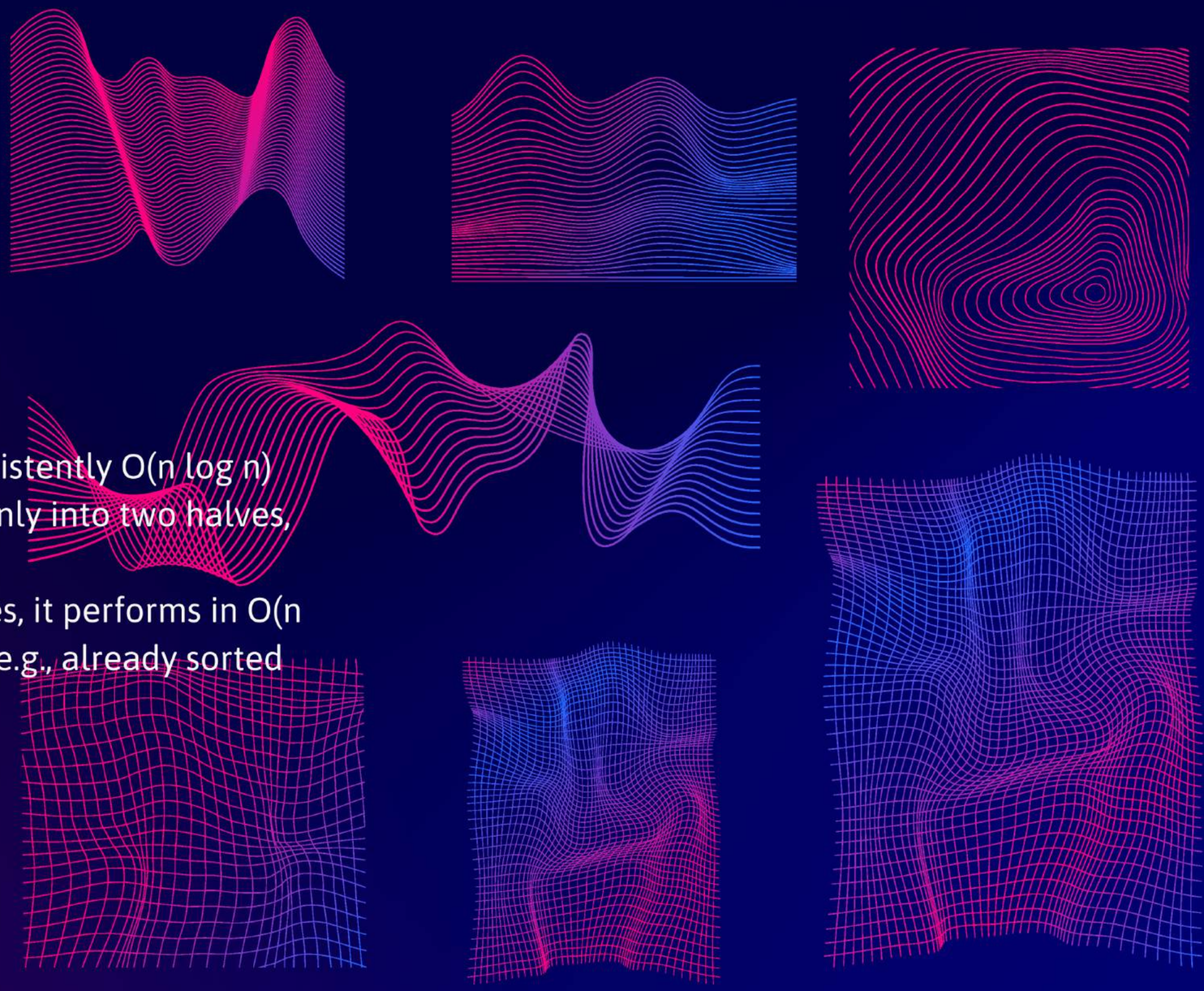
- Type: Comparison-based, divide and conquer.
- Approach: Recursively divides the array into two halves, sorts each half, and then merges them.
- Best/Avg/Worst Time Complexity: O(n log n) in all cases.

Quick Sort:

- Type: Comparison-based, divide and conquer.
- Approach: Selects a pivot, partitions the array into two sub-arrays (elements less than the pivot and elements greater than the pivot), and recursively sorts the sub-arrays.
- Best/Avg Time Complexity: O(n log n), Worst Time Complexity: $O(n^2)$ (when the pivot is poorly chosen).

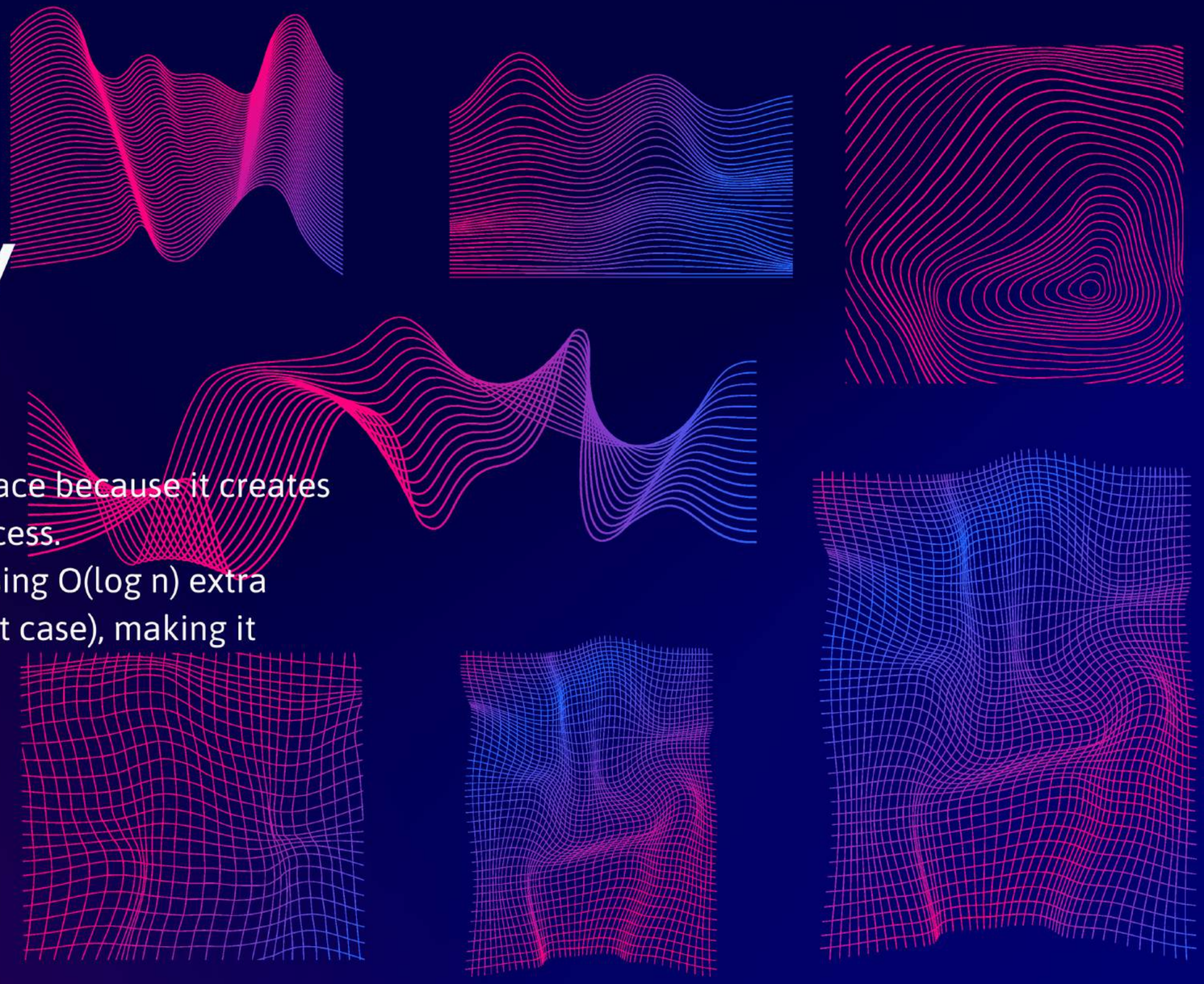
Time Complexity Analysis

- Merge Sort: The time complexity is consistently $O(n \log n)$ because the array is always divided evenly into two halves, regardless of the data.
- Quick Sort: In the best and average cases, it performs in $O(n \log n)$, but if the pivot is poorly chosen (e.g., already sorted array), it degrades to $O(n^2)$.



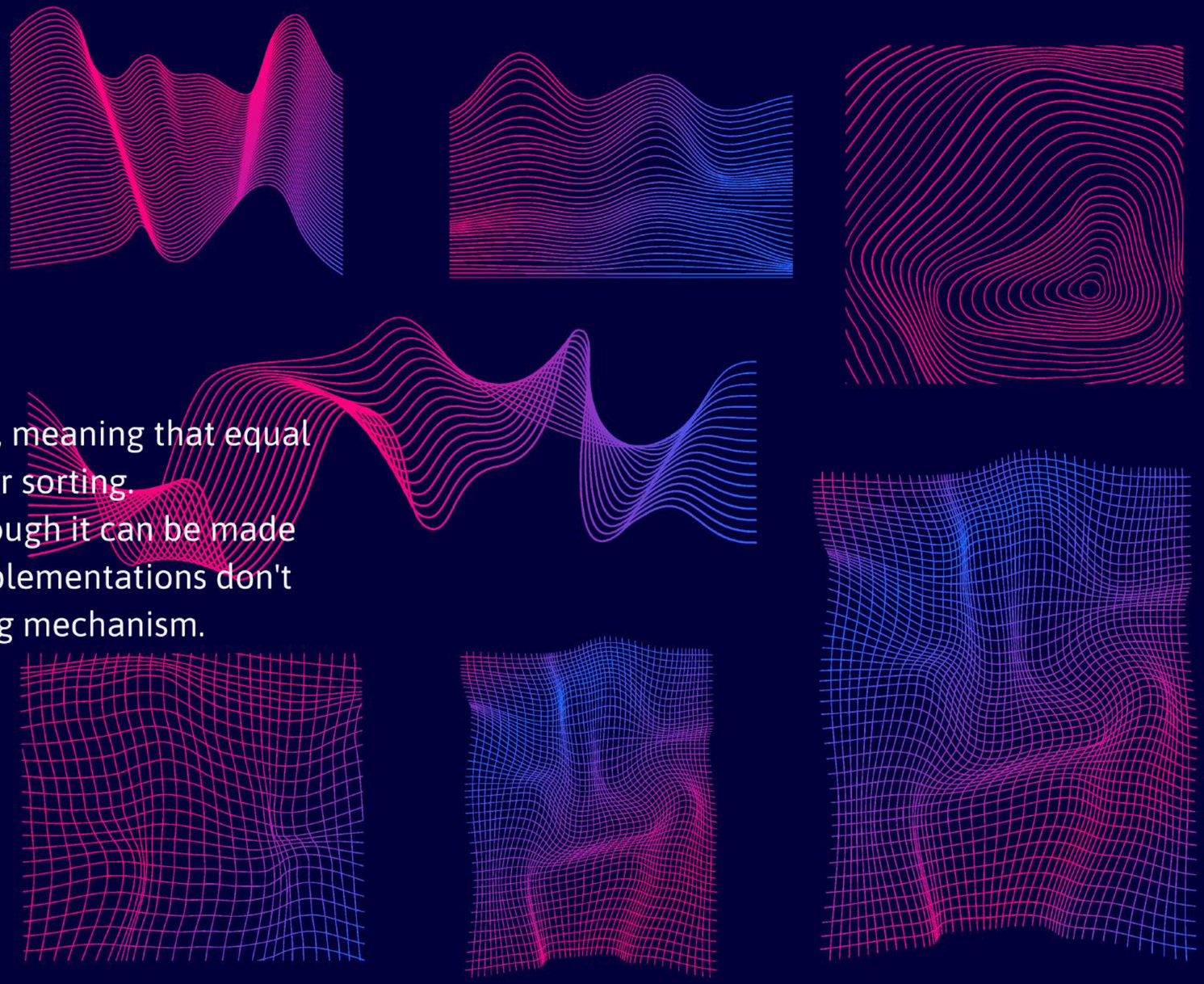
Space Complexity Analysis

- Merge Sort: Requires O(n) additional space because it creates temporary arrays during the merge process.
- Quick Sort: Performs in-place sorting, using O(log n) extra space for the recursion stack (in the best case), making it more space-efficient.



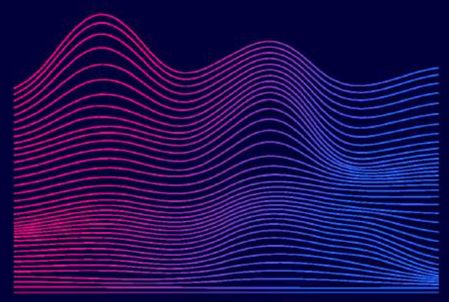
Stability

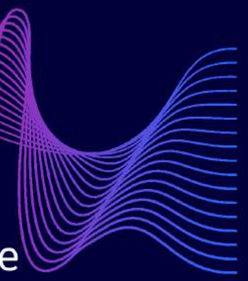
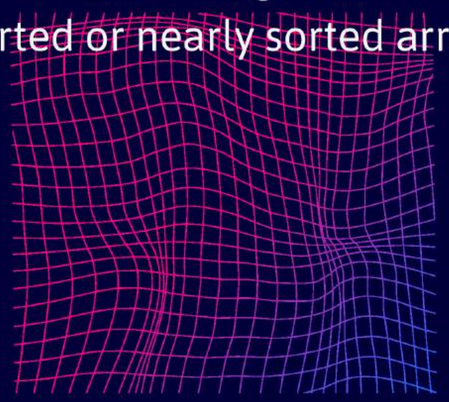
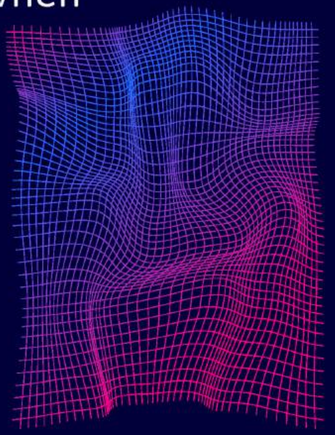
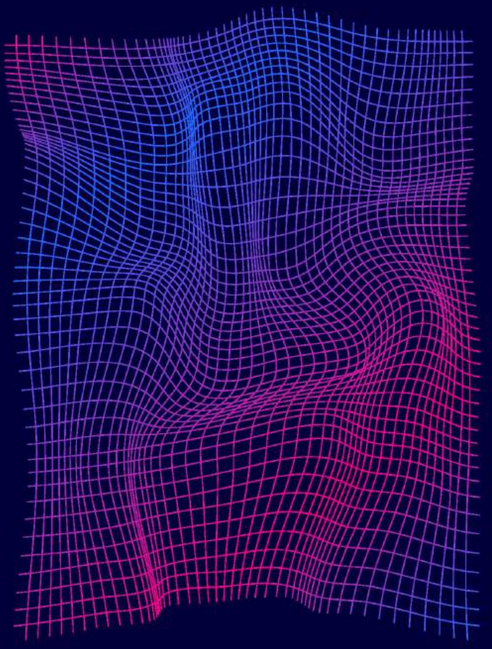
- Merge Sort is a stable sorting algorithm, meaning that equal elements retain their relative order after sorting.
- Quick Sort is not stable by default, although it can be made stable with modifications. Typically, implementations don't ensure stability because of the swapping mechanism.



Performance Comparison

- Merge Sort is preferred when consistent performance is needed, as it guarantees O(n log n) in all cases. It is also ideal for external sorting (e.g., large files on disk) because it's stable and performs well with large datasets.
- Quick Sort is usually faster in practice due to lower constant factors in its time complexity and the use of in-place sorting, making it highly efficient for small to medium datasets. However, its performance can degrade to $O(n^2)$ when poorly handled, especially on already sorted or nearly sorted arrays.

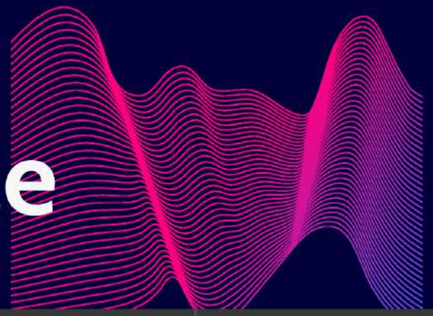
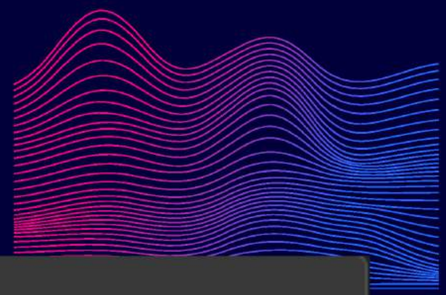



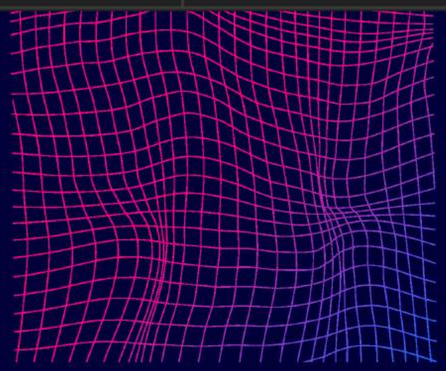
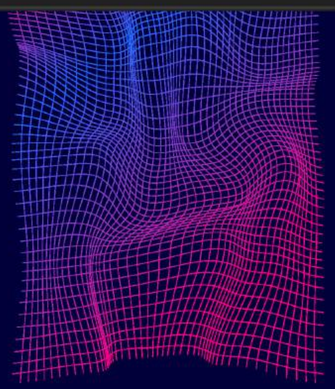
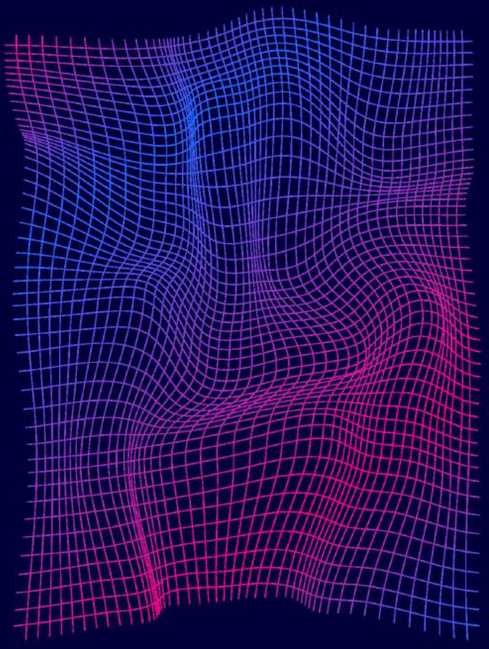





Comparison Table

-

Feature	Merge Sort	Quick Sort
Time Complexity	O(n log n) (all cases)	O(n log n) avg, $O(n^2)$ worst
Space Complexity	O(n)	O(log n) (in-place)
Stability	Stable	Not Stable (default)
In-Place	No	Yes
Best Use Case	Large datasets, guaranteed performance	Small to medium datasets, average performance is better

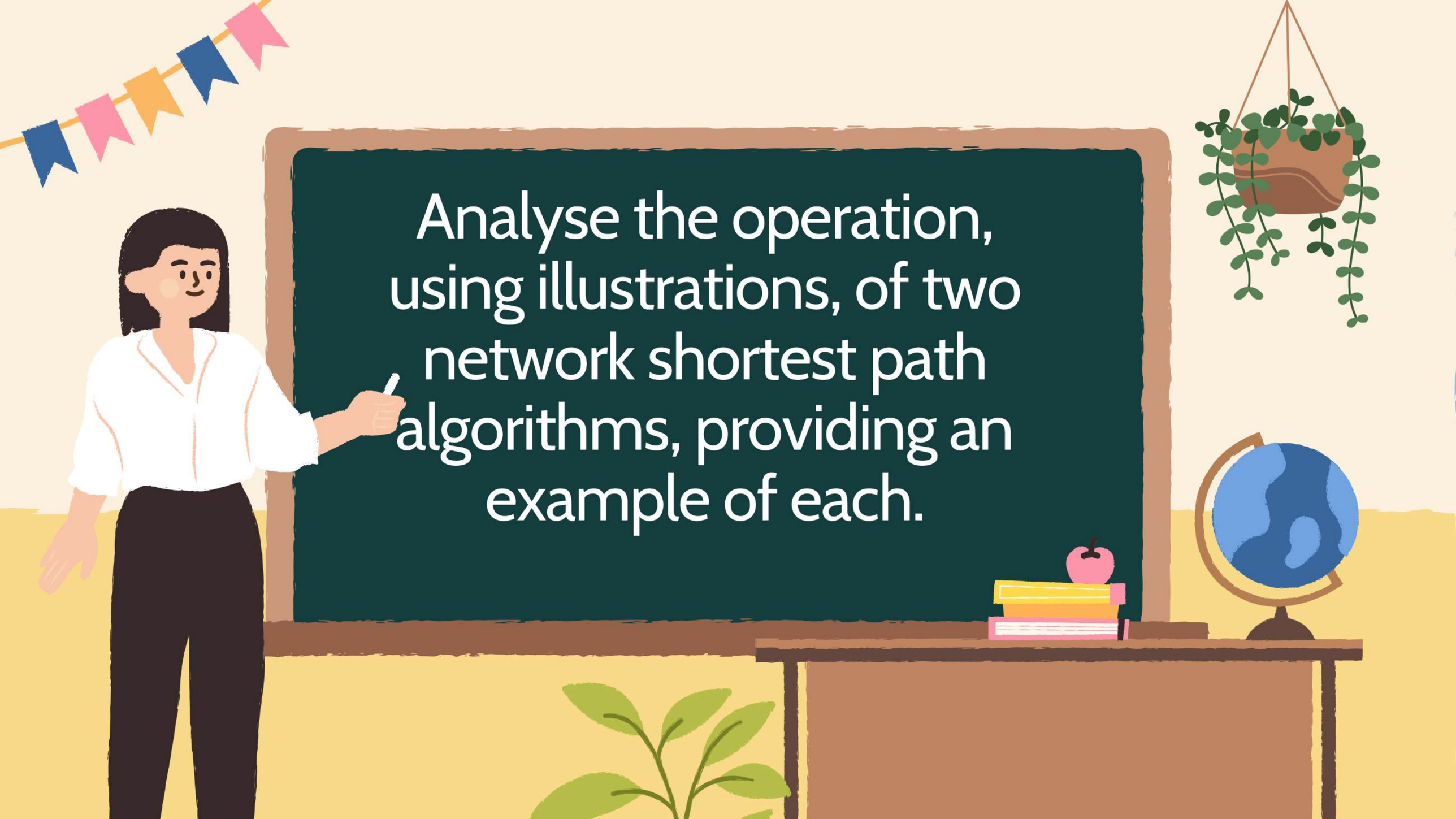
Performance Demonstration

Let’s analyze performance in terms of time:

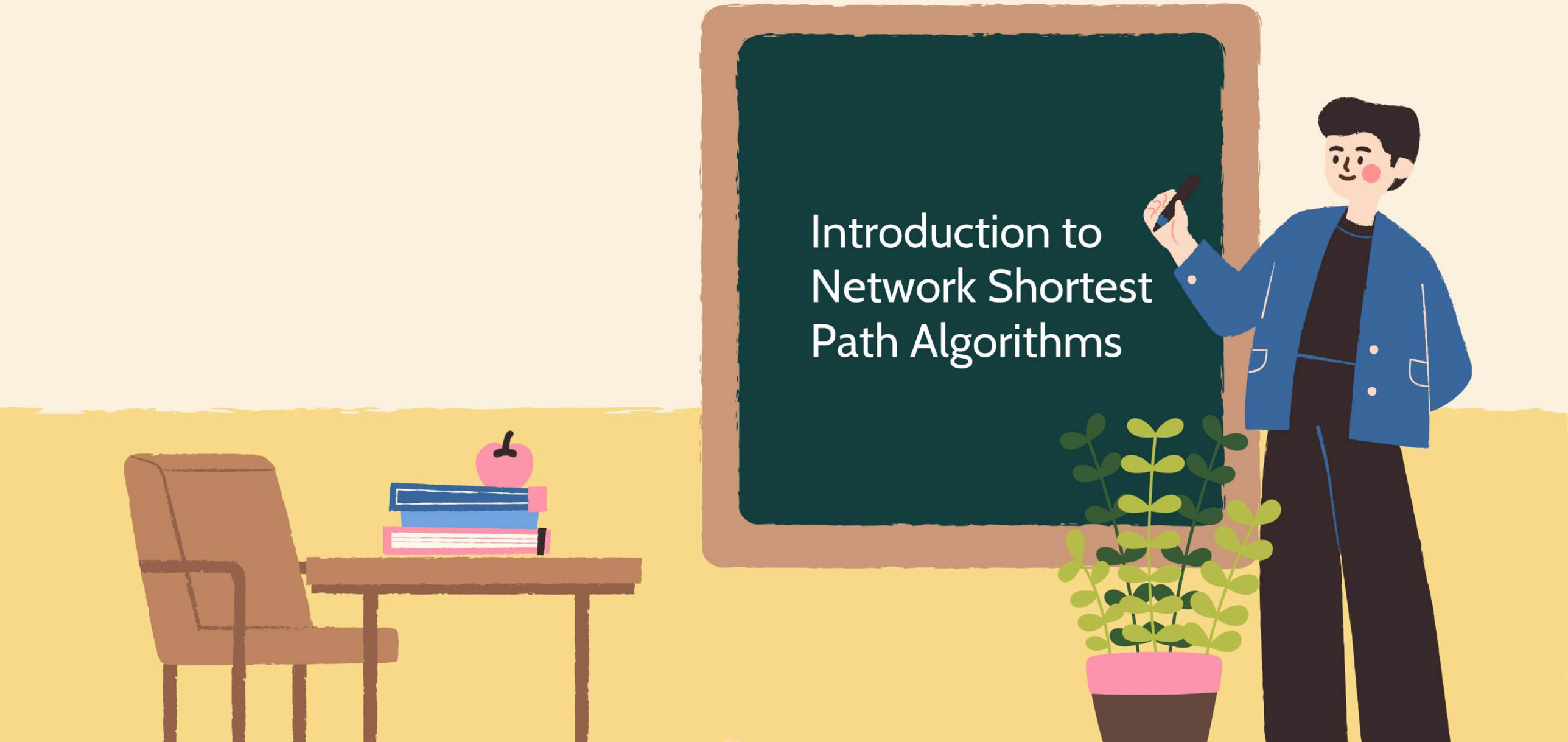
- If you run the above example with a large unsorted array, Merge Sort will always maintain O(n log n) efficiency, whereas Quick Sort may degrade if the pivot is poorly chosen.

For small-to-medium datasets:

- Quick Sort often outperforms Merge Sort due to lower overhead and in-place sorting.
- Merge Sort shines in situations requiring stability or consistent performance across all cases.


Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

Introduction to Network Shortest Path Algorithms



Network shortest path algorithms are used to find the shortest path from a source node to one or more destination nodes in a graph. These algorithms are crucial in applications such as navigation systems, computer networks, and routing protocols. Two of the most commonly used algorithms for finding the shortest path are Dijkstra's Algorithm and Prim-Jarnik Algorithm. While both are based on greedy strategies, they serve slightly different purposes: Dijkstra’s is used for shortest path finding, and Prim-Jarnik is typically used for constructing a minimum spanning tree (MST).

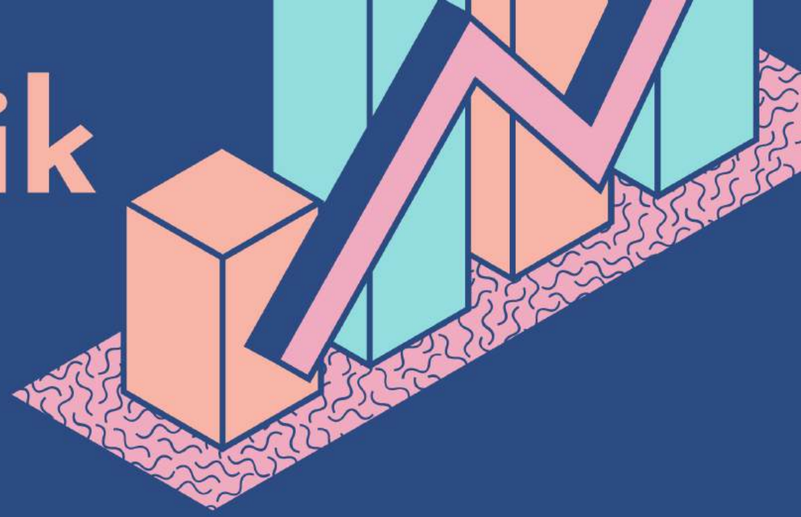
Algorithm 1: Dijkstra's Algorithm

- Overview:
- Purpose: Finds the shortest path from a source node to all other nodes in a weighted graph.
- Assumptions: Works with non-negative edge weights.
- Approach: Uses a greedy method to repeatedly select the node with the smallest tentative distance and update its neighbors.
-

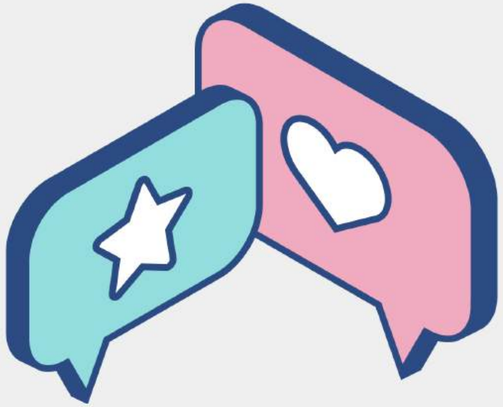
Steps:

1. Initialize distances from the source node to all other nodes as infinity ( ), except for the source itself, which is set to 0.
2. Mark all nodes as unvisited. Set the source node as the current node.
3. For each unvisited neighbor of the current node, calculate its tentative distance (current node's distance + edge weight). If this tentative distance is smaller than the recorded distance, update the distance.
4. Once all neighbors of the current node are checked, mark it as visited (it won’t be checked again).
5. Select the unvisited node with the smallest tentative distance as the new "current node" and repeat the process.
6. Stop when all nodes are visited or the smallest tentative distance among the unvisited nodes is infinity.

Algorithm 2: Prim-Jarnik Algorithm



- Purpose: Finds the Minimum Spanning Tree (MST), which is a subset of edges in a graph that connects all the nodes with the minimum possible total edge weight. Unlike Dijkstra's algorithm, which focuses on finding the shortest path from one node, Prim's algorithm connects all nodes.
- Assumptions: Works with graphs that have non-negative edge weights.
- Approach: Starts with a single node and expands the tree by adding the smallest edge that connects a new node.



Steps

- Start with any node (let's say node A) and mark it as part of the MST.
- Find the smallest edge connecting any node in the MST to a node not yet in the MST.
- Add that edge to the MST and mark the new node as part of the MST.
- Repeat until all nodes are included in the MST.
-

Performance Analysis

Feature	Dijkstra's Algorithm	Prim-Jarnik Algorithm
Purpose	Shortest path from a source node to all others	Minimum spanning tree (MST)
Graph Type	Weighted graphs with non-negative weights	Weighted graphs with non-negative weights
Time Complexity	O((V + E) log V) (with priority queue)	O(E log V) (with priority queue)
Space Complexity	O(V)	O(V)
Use Case	Finding shortest paths in navigation or routing systems	Connecting all nodes with minimal total edge weight
Updates in Weights	Handles dynamic updates in shortest paths	Handles dynamic changes to MST edges
Initial Node Choice	Must specify a source node	Can start from any node
Applications	Shortest path algorithms, network routing	Network design, constructing MSTs

Scenario: Consider a network where you need to:

1. Find the shortest path from a city A to all other cities.
2. Find the minimum cost to connect all cities (MST).