



# **Project presentation**

**Presented by Minh Chinh**



# **Design a specification for data structures, detailing the valid operations they support.**

- 01. Identify the Data Structures
- 02. Define the Operations
- 03. Specify Input Parameters
- 04. Define Pre- and Post-conditions
- 05. Discuss Time and Space Complexity
- 06. Provide Examples and Code Snippets

# 01. Identify the Data Structures

## Linked List

- A sequence of elements, where each element points to the next element.
- Elements are not stored in contiguous memory locations.

## Stack

- A Last In, First Out (LIFO) data structure.
- Operations: push (add element), pop (remove element).

## Array

- A collection of elements, each identified by an index or key.
- Elements are stored in contiguous memory locations.

## Queue

- A First In, First Out (FIFO) data structure.
- Operations: enqueue (add element), dequeue (remove element).

## Hash Table

- Stores key-value pairs for efficient lookup.
- Uses a hash function to compute an index into an array of buckets or slots.

# 01. Identify the Data Structures

## Set

- A collection of unique elements, with no duplicates allowed.
- Operations: Union, Intersection, Difference.

## Heap

- A special tree-based data structure that satisfies the heap property.
- Types: Max Heap (parent nodes are greater than child nodes), Min Heap (parent nodes are less than child nodes).

## Graph

- A collection of nodes (vertices) connected by edges.
- Types: Directed, Undirected, Weighted, Unweighted.

## Trie (Prefix Tree)

- A tree-like data structure used to store strings.
- Efficient for searching prefixes of words.

## Binary Tree

- A hierarchical data structure where each node has at most two children (left and right).
- Types: Binary Search Tree (BST), AVL Tree, Red-Black Tree.



# 02. Define the Operations

## Array Operations

- Access: Retrieve an element using its index in constant time  $O(1)$ .
- Insert: Add an element at a specific position. If inserting at the end, it's  $O(1)$ , but inserting in the middle may require shifting, making it  $O(n)$ .
- Delete: Remove an element from a specific position. Similar to insert, deleting from the middle requires shifting,  $O(n)$ .
- Search: Search for an element by value,  $O(n)$  for linear search, or  $O(\log n)$  for a sorted array using binary search.

## Linked List Operations

- Traversal: Visit each node in sequence to access data,  $O(n)$ .
- Insert: Add a node at the beginning ( $O(1)$ ), end ( $O(n)$ ), or at a specified position ( $O(n)$ ).
- Delete: Remove a node, which requires finding the node ( $O(n)$ ) and unlinking it.
- Search: Find a node with a specific value,  $O(n)$ .

## Stack Operations

- Push: Add an element to the top of the stack,  $O(1)$ .
- Pop: Remove and return the element at the top of the stack,  $O(1)$ .
- Peek (Top): Return the element at the top without removing it,  $O(1)$ .
- IsEmpty: Check if the stack is empty,  $O(1)$ .

## Queue Operations

- Enqueue: Add an element to the rear (end) of the queue,  $O(1)$ .
- Dequeue: Remove and return the element from the front of the queue,  $O(1)$ .
- Peek (Front): Return the element at the front without removing it,  $O(1)$ .
- IsEmpty: Check if the queue is empty,  $O(1)$ .

## 02. Define the Operations

### Hash Table Operations

- Insert: Insert a key-value pair,  $O(1)$  on average.
- Delete: Remove a key-value pair,  $O(1)$  on average.
- Search: Look up the value associated with a given key,  $O(1)$  on average.
- Rehashing: Resize the hash table when the load factor increases,  $O(n)$ .

### Binary Tree Operations

- Insert: Add a node while maintaining the tree structure,  $O(\log n)$  in a balanced tree,  $O(n)$  in a degenerate (skewed) tree.
- Delete: Remove a node while maintaining tree properties,  $O(\log n)$  in a balanced tree.
- Search: Find a node by value,  $O(\log n)$  in a balanced tree.
- Traversal: Visit all nodes in a specific order (Inorder, Preorder, Postorder, Level-order),  $O(n)$ .

### Heap Operations

- Insert: Add an element while maintaining the heap property,  $O(\log n)$ .
- ExtractMax / ExtractMin: Remove and return the maximum (or minimum) element,  $O(\log n)$ .
- Peek (Max / Min): Return the maximum (or minimum) element without removing it,  $O(1)$ .
- Heapify: Restore the heap property after an insertion or deletion,  $O(\log n)$ .

### Trie Operations

- Insert: Add a word to the trie,  $O(m)$  where  $m$  is the length of the word.
- Search: Check if a word or prefix exists in the trie,  $O(m)$ .
- Delete: Remove a word from the trie,  $O(m)$ .

## 02. Define the Operations

### Graph Operations

- Add Vertex: Add a vertex to the graph,  $O(1)$ .
- Add Edge: Add an edge between two vertices,  $O(1)$ .
- Remove Vertex: Remove a vertex and all associated edges,  $O(V + E)$  where  $V$  is the number of vertices and  $E$  is the number of edges.
- Remove Edge: Remove an edge between two vertices,  $O(1)$ .
- Traversal: Visit all vertices using Breadth-First Search (BFS) or Depth-First Search (DFS),  $O(V + E)$ .

### Set Operations

- Insert: Add an element to the set,  $O(1)$ .
- Delete: Remove an element from the set,  $O(1)$ .
- Contains: Check if the set contains a specific element,  $O(1)$ .
- Union: Combine two sets into one,  $O(n)$ .
- Intersection: Find common elements between two sets,  $O(n)$ .
- Difference: Find elements present in one set but not the other,  $O(n)$ .



**Input Format:** Define how the input should be structured. For instance, a graph might be represented using an adjacency matrix or an adjacency list.

**Constraints:** Specify the range or constraints on the input values (e.g., array size limits, value ranges, etc.) to ensure the algorithm performs efficiently and avoids invalid inputs.

**Edge Cases:** Identify special or edge cases that should be considered for input (e.g., empty input, negative values, null pointers, etc.).

## 03. Specify Input Parameters



```
graph TD; A[Input Format: Define how the input should be structured. For instance, a graph might be represented using an adjacency matrix or an adjacency list.] --> C[03. Specify Input Parameters]; B[Constraints: Specify the range or constraints on the input values (e.g., array size limits, value ranges, etc.) to ensure the algorithm performs efficiently and avoids invalid inputs.] --> C; D[Edge Cases: Identify special or edge cases that should be considered for input (e.g., empty input, negative values, null pointers, etc.).] --> C; E[Size of Input: In many algorithms, the size of the input is a critical parameter. This is especially relevant for arrays, lists, trees, and graphs, where knowing the number of elements is essential for controlling the algorithm's flow and determining its complexity.] --> C; F[Data Type: Specify the type of data (e.g., integer, float, string, array, etc.) the algorithm will accept. For example, an algorithm to find the maximum number in a list needs to know that the input is an array or list of integers.] --> C;
```

**Size of Input:** In many algorithms, the size of the input is a critical parameter. This is especially relevant for arrays, lists, trees, and graphs, where knowing the number of elements is essential for controlling the algorithm's flow and determining its complexity.

**Data Type:** Specify the type of data (e.g., integer, float, string, array, etc.) the algorithm will accept. For example, an algorithm to find the maximum number in a list needs to know that the input is an array or list of integers.



# 04. Define Pre- and Post-conditions



-Pre-conditions refer to the requirements or conditions that must be true before a function or operation is executed. These conditions define the valid input state for a function to work properly. If the pre-conditions are not met, the function may fail or produce incorrect results.

-Purpose: To ensure that a function operates in a predictable and correct manner.

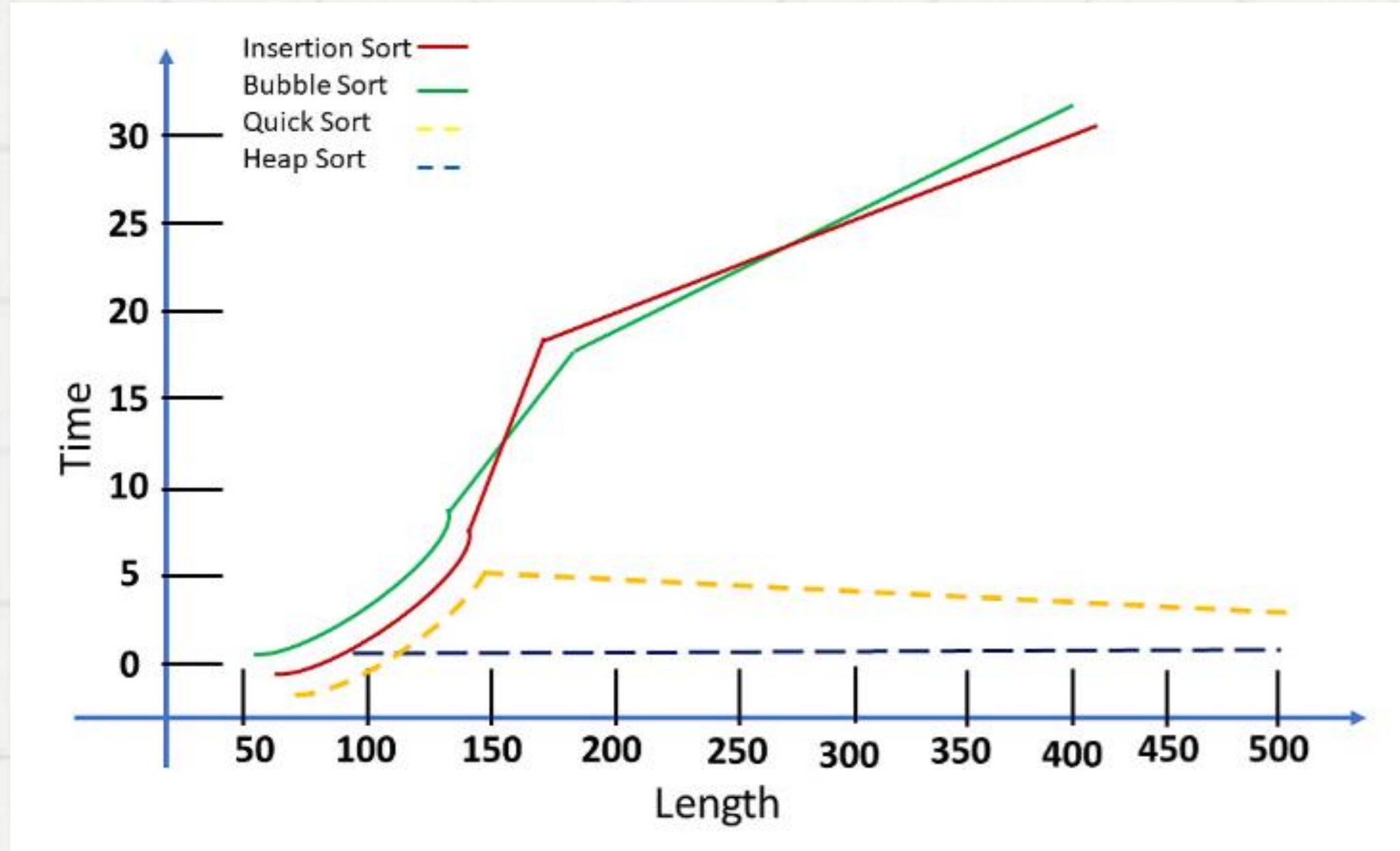
-Post-conditions define the expected state or outcome after a function or operation is executed. These conditions describe the intended results of the function once it has been completed successfully. It ensures that the function has performed as expected and that the output or changes meet certain criteria.

-Purpose: To validate that the function has achieved its desired result.

## 05. Discuss Time and Space Complexity

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input. Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered

Significant in Terms of Space Complexity: The input size has a strong relationship with time complexity. As the size of the input increases, so does the runtime, or the amount of time it takes the algorithm to run.



## 05. Discuss Time and Space Complexity

-When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and temporal values while running, the space complexity is auxiliary and input space.

-Space complexity refers to the total amount of memory space used by an algorithm/program, including the space of input values for execution. Calculate the space occupied by variables in an algorithm/program to determine space complexity.

-However, people frequently confuse Space-complexity with auxiliary space. Auxiliary space is simply extra or temporary space, and it is not the same as space complexity. To put it another way,

Auxiliary space + space use by input values = Space Complexity

-The best algorithm/program should have a low level of space complexity. The less space required, the faster it executes.






## 06. Provide Examples and Code Snippets

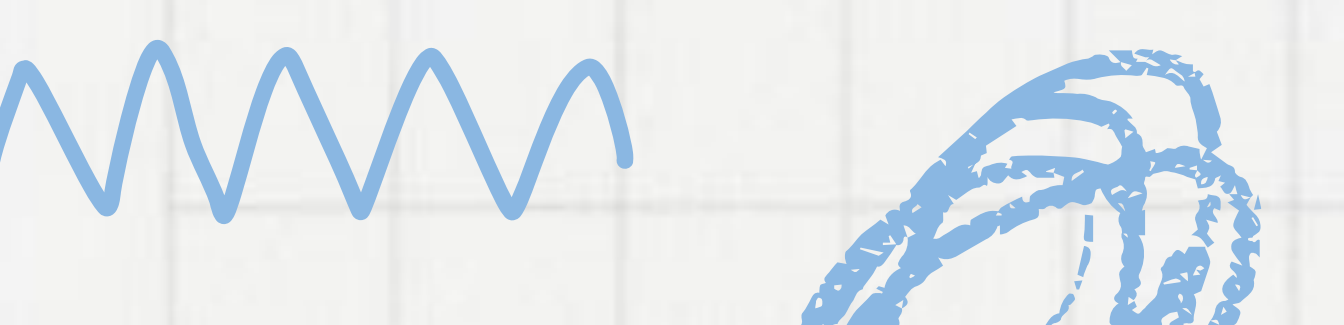


```
public class QuickSort {  
    public void quickSort(int[] array, int low, int high) { 3 usages  
        if (low < high) {  
            int pivotIndex = partition(array, low, high);  
            quickSort(array, low, high: pivotIndex - 1); // Recursively sort the left part  
            quickSort(array, low: pivotIndex + 1, high); // Recursively sort the right part  
        }  
    }  
  
    private int partition(int[] array, int low, int high) { 1 usage  
        int pivot = array[high]; // Choosing the last element as pivot  
        int i = (low - 1); // Pointer for the smaller element  
        for (int j = low; j < high; j++) {  
            if (array[j] <= pivot) {  
                i++;  
                // Swap array[i] and array[j]  
                int temp = array[i];  
                array[i] = array[j];  
                array[j] = temp;  
            }  
        }  
        // Swap array[i + 1] and array[high] (or pivot)  
        int temp = array[i + 1];  
        array[i + 1] = array[high];  
        array[high] = temp;  
        return i + 1;  
    }  
  
    public static void main(String[] args) {  
        QuickSort qs = new QuickSort();  
        int[] array = { 34, 7, 23, 32, 5, 62 };  
        qs.quickSort(array, low: 0, high: array.length - 1);  
        System.out.println("Sorted array: " + Arrays.toString(array));  
    }  
}
```





# **Determine the operations of a memory stack and how it is used to implement function calls in a computer.**

- 01. Define a Memory Stack
  - 02. Identify Operations
  - 03. Function Call Implementation
  - 04. Demonstrate Stack Frames
  - 05. Discuss the Importance
- 

# 01. Define Memory Stack

## Define

Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage that behaves as a first-in, last-out buffer.

## Activity

When a function is called, the stack stores:

- The return address for the function once it finishes.
- The input parameters of the function.
- The local variables within the function.



## 02. Identify Operations

A typical memory stack supports the following operations:

- Push: Push an element (value) onto the stack (e.g., function parameters or return addresses).
- Pop: Remove an element from the stack.
- Peek/Top: View the value of the element on top of the stack without removing it.
- IsEmpty: Check if the stack is empty.
- IsFull: Check if the stack is full (in case the stack has a limited size).



## 03. Function Call Implementation

During function execution, the memory stack operates as follows:

- Push Call Stack: When a function is called, the system pushes the input parameters, the return address (where the program will continue after the function finishes), and the local variables of the function onto the memory stack.
- Function Execution: After pushing the necessary data onto the stack, the program begins executing the function's code.
- Pop Call Stack: Once the function finishes, the return address is popped from the stack, and the program continues execution from that point. The local variables and function parameters are also removed from the stack.





## 04. Demonstrate Stack Frames

A Stack Frame is a block of data on the stack that contains the information necessary for the execution of a function, including:

- The return address.
- The function's parameters.
- The local variables.

```
public class Example {  
    public static void main(String[] args) {  
        int a = 10;  
        int result = sum(a, y: 5);  
        System.out.println(result);  
    }  
  
    public static int sum(int x, int y) { 1 usage  
        return x + y;  
    }  
}
```




## 05. Discuss the Importance

### Importance of Memory Stack in Function Execution

- Efficient memory management: The memory stack automatically manages local variables and function parameters. Once a function finishes, the memory occupied by these variables is freed.
- Execution flow control: The stack tracks the flow of the program by storing the return addresses when executing nested function calls. This is essential when dealing with recursive functions.
- Error detection: When the stack exceeds its capacity (stack overflow), the program halts, helping detect logic errors such as infinite recursion.





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

01. Introduction FIFO

02. Define the Structure

03. Array-Based Implementation

04. Linked List-Based Implementation

05. Provide a concrete example to illustrate how the FIFO queue works

# 01. Introduction FIFO

-FIFO stands for First In, First Out. It is a fundamental concept in data structures that describes the order in which elements are processed. In a FIFO system, the first element that is added is the first one to be removed or processed. This concept is analogous to a queue in real life, such as people standing in line; the first person to enter the line is the first to be served.

In computer science, FIFO is typically implemented using a queue data structure. A queue allows elements to be inserted at the back (or rear) and removed from the front. This structure is particularly useful for managing tasks that need to be processed in the order they arrive, such as handling print jobs, process scheduling in operating systems, or breadth-first search in graph algorithms.

-Characteristics of FIFO:

- Insertion: New elements are added at the rear of the queue.
- Deletion: Elements are removed from the front of the queue.
- Order: The element that was inserted first is the one that gets processed first.





## 02. Define the Structure

### Define

The structure provides a blueprint or model that defines how the data is organized and accessed. It outlines the relationships between data items and the operations that can be performed on them.

### Storage Mechanism

- Array or Linked List: A queue can be implemented using an array or a linked list. The choice of implementation affects performance and flexibility (e.g., array-based queues might have fixed capacity, while linked list-based queues grow dynamically).
- Front and Rear Pointers: In the structure, two pointers or indices are maintained—front for tracking the element at the front of the queue (to be dequeued next) and rear for tracking where new elements will be added.




## 03. Array-Based Implementation



### Structure of Array-Based Queue:

- Array: Holds the queue elements.
- Front: Tracks the position of the front element (the next element to be dequeued).
- Rear: Tracks the position where the next element will be enqueued.
- Capacity: The maximum number of elements the queue can hold.
- Size: Tracks the number of elements currently in the queue.

### Key Operations:

- Enqueue: Adds an element to the rear.
  - Dequeue: Removes an element from the front.
  - Peek: Returns the element at the front without removing it.
  - isEmpty: Checks if the queue is empty.
  - isFull: Checks if the queue is full (since the array has a fixed size).
- 

# 04. Linked List-Based Implementation

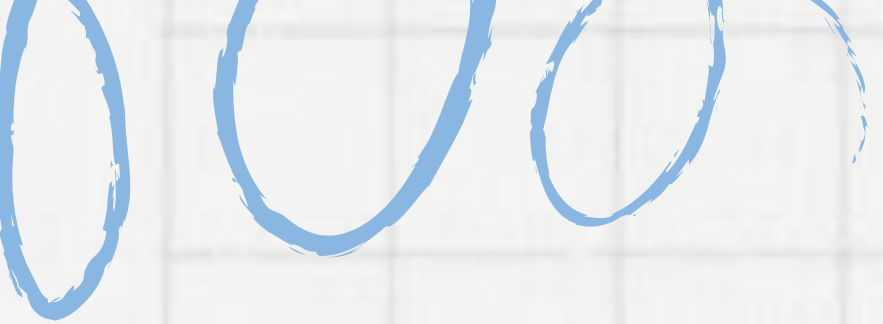


## Structure of Linked List-Based Queue:


- Node: Each node in the linked list contains:
  - Data: The value stored in the node.
  - Next: A reference to the next node in the list.
- Front: A pointer to the node at the front of the queue (where elements are dequeued from).
- Rear: A pointer to the node at the rear of the queue (where elements are enqueued).
- Size: The number of elements currently in the queue.

## Key Operations:

- Enqueue: Adds an element at the rear of the queue.
- Dequeue: Removes an element from the front of the queue.
- Peek: Returns the front element without removing it.
- isEmpty: Checks if the queue is empty.




## 05. Provide a concrete example to illustrate how the FIFO queue works



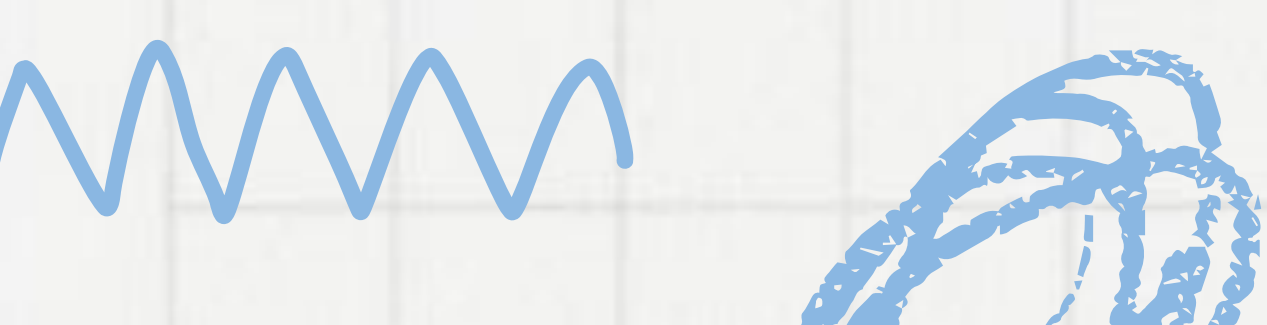
```
class MyQueue {
  constructor() {
    this.items = [];
  }
  enqueue(element) {
    this.items.push(element);
    console.log(`Enqueued: ${element}`);
  }
  dequeue() {
    if (this.isEmpty()) {
      return "Queue is empty";
    }
    const dequeuedElement = this.items.shift();
    console.log(`Dequeued: ${dequeuedElement}`);
    return dequeuedElement;
  }
  isEmpty() {
    return this.items.length === 0;
  }
  peek() {
    if (this.isEmpty()) {
      return "Queue is empty";
    }
    return this.items[0];
  }
  size() {
    return this.items.length;
  }
}

const queue = new MyQueue();
queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);
console.log("Peek:", queue.peek());
queue.dequeue();
queue.dequeue();
console.log("Queue size:", queue.size());
queue.dequeue();
console.log("Is the queue empty?", queue.isEmpty());
```





# Compare the performance of two sorting algorithms.

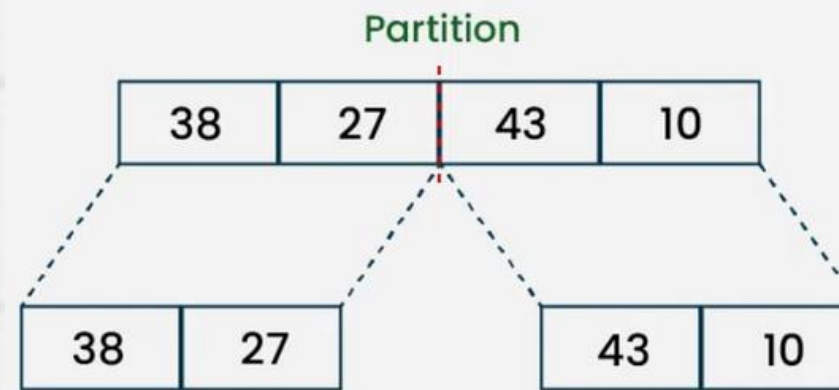
- 01. Introducing the two sorting algorithms you will be comparing
  - 02. Time Complexity Analysis
  - 03. Space Complexity Analysis
  - 04. Stability
  - 05. Comparison Table
  - 06. Performance Comparison
  - 07. Provide a concrete example to demonstrate the differences in performance between the two algorithms
- 

# 01. Introducing the two sorting algorithms you will be comparing

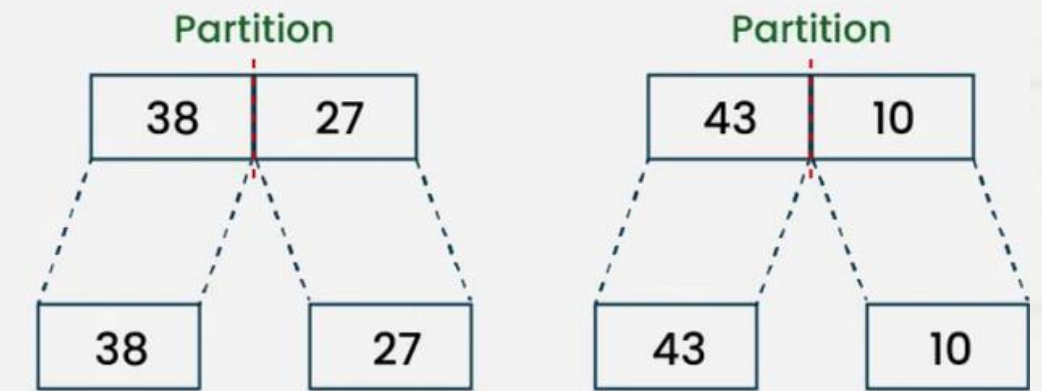
-Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

-In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

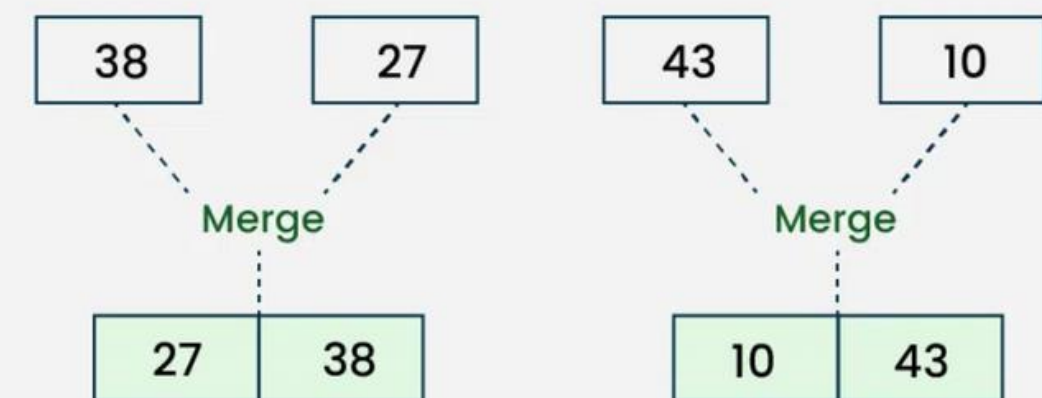
**Step 1** | Splitting the Array into two equal halves



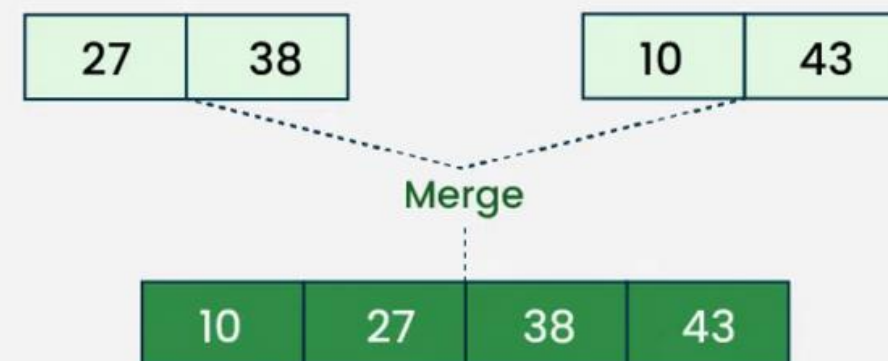
**Step 2** | Splitting the subarrays into two halves



**Step 3** | Merging unit length cells into sorted subarrays

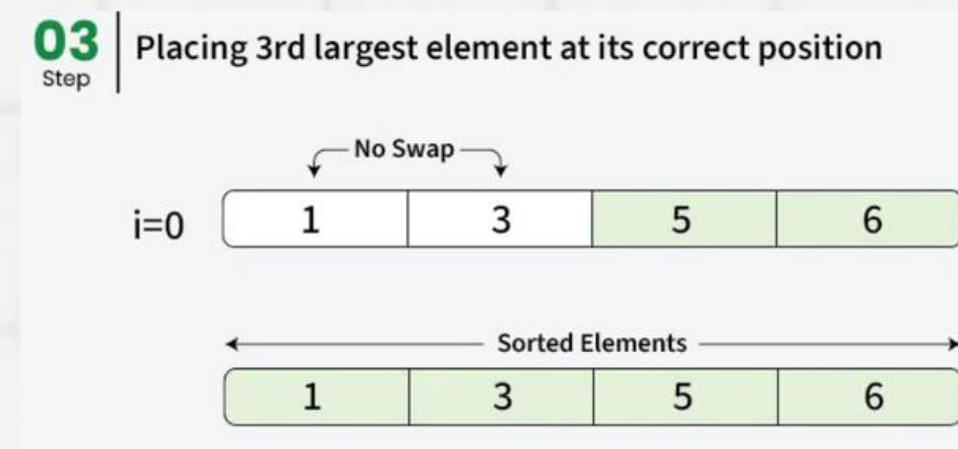
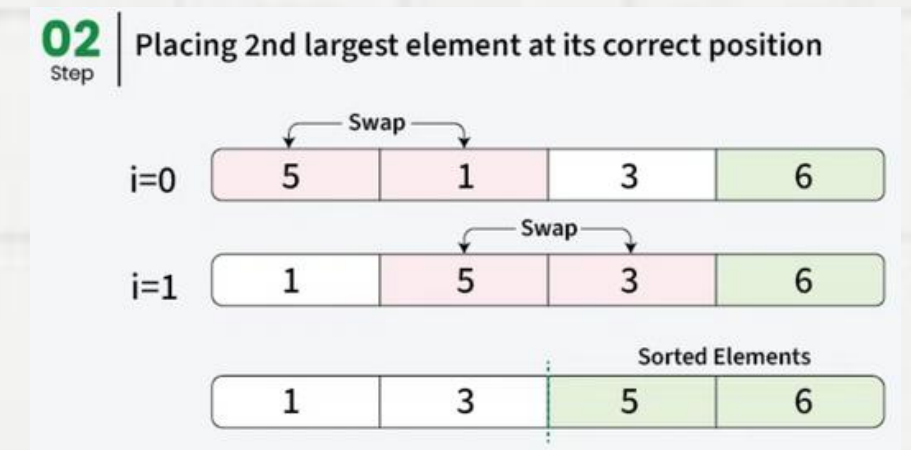
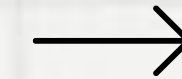
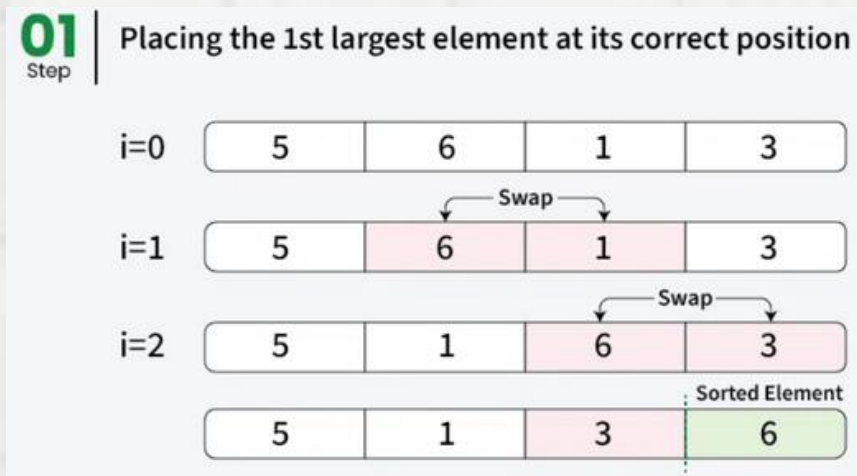


**Step 4** | Merging sorted subarrays into the sorted array



# 01. Introducing the two sorting algorithms you will be comparing

-Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.





# 02. Time Complexity Analysis



## Bubble Sort

- Best Case:  $O(n)$ 
  - If the array is already sorted, the algorithm only needs to pass through the array once, making  $n$  comparisons without any swaps.
- Worst Case:  $O(n^2)$ 
  - In the worst case (when the array is sorted in reverse order), Bubble Sort has to compare each element with every other element, leading to  $n \times (n-1) / 2$  comparisons, which simplifies to  $O(n^2)$ .
- Average Case:  $O(n^2)$ 
  - On average, Bubble Sort will perform similarly to the worst-case scenario because most unsorted arrays will require numerous comparisons and swaps.

## Merge Sort

- Best Case:  $O(n \log n)$ 
  - Even in the best case, Merge Sort must divide the array into subarrays and merge them back, which leads to a consistent logarithmic growth.
- Worst Case:  $O(n \log n)$ 
  - The number of divisions and merging operations remains the same regardless of the input's initial order.
- Average Case:  $O(n \log n)$ 
  - The average performance is also  $O(n \log n)$  since it consistently divides the array and merges it back in the same fashion.



# O3. Space Complexity Analysis



## Bubble Sort

- $O(1)$
- Bubble Sort is an in-place sorting algorithm, meaning it doesn't require any additional space for sorting the array except for temporary variables used for swapping.

## Merge Sort

- $O(n)$
- Merge Sort requires additional memory to store the divided subarrays and merge them back. This additional memory is proportional to the size of the input array.

# 04. Stability



## Bubble Sort

- Stable: Yes
- Reason: In Bubble Sort, when two elements are compared and are found to be equal, the algorithm does not swap them. This ensures that the relative order of equal elements is preserved.
- Since Bubble Sort only swaps elements if the left one is greater than the right, equal elements are left in their original relative positions.

## Merge Sort

- Stable: Yes
- Reason: During the merge process, Merge Sort ensures stability by always taking elements from the left subarray before the right subarray if the elements are equal. This preserves the relative order of equal elements from the original input array.
- For example, if we have two elements 5a and 5b (where a and b denote different elements but with the same value), Merge Sort will maintain their original order in the sorted array.



# 05. Comparison Table

Criteria	Merge Sort	Bubble Sort
Time Complexity (Best)	$O(n \log n)$	$O(n)$ (if already sorted)
Time Complexity (Worst)	$O(n \log n)$	$O(n^2)$
Time Complexity (Average)	$O(n \log n)$	$O(n^2)$
Space Complexity	$O(n)$ (due to additional memory for merging)	$O(1)$ (in-place)
Stability	Stable	Stable
Algorithm Type	Divide and Conquer	Brute Force (Comparison-based)
Method	Recursively divides and merges arrays	Repeatedly compares and swaps adjacent elements
Best Use Case	Large datasets requiring fast sorting	Small datasets or nearly sorted arrays
Worst Use Case	Memory-constrained environments	Large, completely unsorted datasets
In-Place Sorting	No	Yes
Recursive	Yes	No



## 06. Performance Comparison

Aspect	Merge Sort	Bubble Sort
Time Complexity (Best Case)	$O(n \log n)$	$O(n)$
Time Complexity (Worst Case)	$O(n \log n)$	$O(n^2)$
Time Complexity (Average Case)	$O(n \log n)$	$O(n^2)$
Space Complexity	$O(n)$ (requires auxiliary arrays)	$O(1)$ (in-place sorting)
Number of Comparisons	$O(n \log n)$	$O(n^2)$
Number of Swaps	Minimal, only during merging	High, as every adjacent pair might be swapped
Stability	Stable	Stable
Efficiency for Large Data	Efficient (scalable due to $O(n \log n)$ )	Inefficient ( $O(n^2)$ for large data)
Best for	Large datasets requiring fast sorting	Small or nearly sorted datasets
Parallelizable	Yes (can be divided and processed in parallel)	No



# 07. Provide a concrete example to demonstrate the differences in performance between the two algorithms

Example: Array: [64, 34, 25, 12, 22, 11, 90]

## Bubble Sort Performance:

### 1. First Pass:

- Compare 64 and 34 → Swap → [34, 64, 25, 12, 22, 11, 90]
- Compare 64 and 25 → Swap → [34, 25, 64, 12, 22, 11, 90]
- Compare 64 and 12 → Swap → [34, 25, 12, 64, 22, 11, 90]
- Compare 64 and 22 → Swap → [34, 25, 12, 22, 64, 11, 90]
- Compare 64 and 11 → Swap → [34, 25, 12, 22, 11, 64, 90]
- 90 is already in the correct position.

### 2. Subsequent Passes:

- Bubble Sort continues comparing and swapping adjacent elements in each pass, resulting in  $O(n^2)$  operations.

## Time Complexity:

- In this random case, Bubble Sort takes 21 comparisons and swaps.

## Merge Sort Performance:

### 1. Divide Phase:


- Split the array into two halves: [64, 34, 25, 12] and [22, 11, 90].
- Split further: [64, 34] and [25, 12], [22, 11] and [90].
- Continue splitting until individual elements are left.

### 2. Merge Phase:

- Merge [64] and [34] → [34, 64]
- Merge [25] and [12] → [12, 25]
- Merge [34, 64] and [12, 25] → [12, 25, 34, 64]
- Merge [22] and [11] → [11, 22]
- Merge [11, 22] and [90] → [11, 22, 90]
- Merge [12, 25, 34, 64] and [11, 22, 90] → [11, 12, 22, 25, 34, 64, 90]

## Time Complexity:

- Merge Sort takes  $O(n \log n)$  time, making about 17 comparisons.



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

01. Introducing the concept of network shortest path algorithms

02. Algorithm 1: Dijkstra's Algorithm

03. Algorithm 2: Prim-Jarnik Algorithm

04. Performance Analysis

# 01. Introducing the concept of network shortest path algorithms

-In the context of graph theory and networking, shortest path algorithms are designed to find the most efficient path between two nodes (or vertices) in a graph. These nodes can represent cities, computers, routers, or any interconnected entities, and the edges between them represent the paths or links, often associated with a weight (e.g., distance, time, cost). The goal of the shortest path algorithm is to find the path between two points that minimizes the total cost (or weight) of traversing those edges.

-These algorithms are crucial in many applications, including:

- Internet routing protocols, to find the most efficient route for data packets.
- GPS navigation systems, to calculate the quickest or shortest driving route.
- Supply chain management, to optimize logistics and transportation routes.
- Telecommunications networks, to ensure optimal communication paths.





# 02. Algorithm 1: Dijkstra's Algorithm

-Dijkstra's Algorithm is a graph-based algorithm used to find the shortest path between a starting node and all other nodes in a weighted graph. It guarantees that, when it terminates, the shortest path from the source node to every other node is known. The algorithm works for graphs with non-negative weights and is widely used in applications like GPS navigation, network routing protocols, and many other optimization scenarios.

-Purpose

Dijkstra's Algorithm is primarily used for single-source shortest path problems:

- To find the shortest path from a source node to a destination node.
- To find the shortest path from a source node to all other nodes in the graph.

-Steps of Dijkstra's Algorithm

1. Initialization:

- Set the distance to the starting node (source) as 0 and to all other nodes as infinity ( $\infty$ ).
- Mark all nodes as unvisited. Create a set of unvisited nodes.

2. Select Node:

- Pick the unvisited node with the smallest known distance (initially, this is the source node). Let's call this node current.

3. Update Neighbors:

- For each unvisited neighbor of the current node, calculate the tentative distance through the current node.
- If the tentative distance is smaller than the previously recorded distance for that neighbor, update it with the smaller distance.

4. Mark Node as Visited:

- Once the neighbors have been updated, mark the current node as visited. A visited node will not be checked again.

5. Repeat:

- Repeat steps 2-4 until all nodes have been visited or until the smallest tentative distance among the unvisited nodes is infinity. If the smallest distance is infinity, the remaining unvisited nodes are not connected to the source, and the algorithm can stop.



# 03. Algorithm 2: Prim-Jarnik Algorithm

The Prim-Jarnik algorithm, more commonly referred to as Prim's Algorithm, is a classic greedy algorithm in graph theory used to find a minimum spanning tree (MST) for a connected, undirected graph. A minimum spanning tree is a subgraph that connects all the vertices of the original graph without any cycles and with the minimum possible total edge weight.

## Purpose

The goal of Prim's algorithm is to construct a spanning tree that covers all vertices in a graph while minimizing the sum of the edge weights. It is used when we need to connect multiple points in the most cost-effective way, such as in network design, like wiring buildings or connecting nodes in a computer network.

## Steps of Prim's Algorithm

### 1. Initialization:

- Start with an arbitrary node (vertex) and treat it as part of the growing spanning tree.
- Mark this vertex as visited.

### 2. Edge Selection:

- From the set of all edges that connect the visited nodes to unvisited nodes, pick the edge with the smallest weight.

### 3. Expand the Tree:

- Add the edge and the connected unvisited vertex to the tree.
- Mark this newly connected vertex as visited.

### 4. Repeat:

- Repeat the process of selecting the smallest edge that connects the tree to a new unvisited node until all vertices are included in the tree.

# 04. Performance Analysis

## Prim-Jarnik Algorithm

Problem Addressed: Prim-Jarnik algorithm is used to find the Minimum Spanning Tree (MST) in a weighted, connected, undirected graph. The MST is a subgraph that connects all vertices with the minimum possible total edge weight without forming cycles.

Steps of Prim-Jarnik Algorithm:

1. Start with an arbitrary vertex and add it to the MST.
2. Choose the smallest edge that connects a vertex in the MST to a vertex not yet in the MST.
3. Repeat until all vertices are included in the MST.

Time Complexity:

- Using a Binary Heap:  $O(E \log V)$ , where  $E$  is the number of edges and  $V$  is the number of vertices.
  - Each vertex is processed once, and for each vertex, adjacent edges are added to a priority queue (min-heap).
  - Extracting the minimum edge from the heap takes  $O(\log V)$  time, and this happens for each of the  $E$  edges.
- Using an Adjacency Matrix:  $O(V^2)$ .
  - Prim's algorithm iterates through all the vertices and finds the minimum edge in each step, which results in a quadratic time complexity when using an adjacency matrix.

Space Complexity:

- $O(V)$  for storing the MST, the list of vertices, and the priority queue.

Practical Performance:

- Efficient for Dense Graphs: In dense graphs, where the number of edges  $E$  is close to  $V^2$ , Prim's algorithm performs efficiently due to its  $O(E \log V)$  complexity.
- Optimal for Sparse Graphs Using Heaps: Using binary heaps and adjacency lists makes Prim's algorithm practical for sparse graphs, with fewer edges.

# 04. Performance Analysis

## Dijkstra's Algorithm

Problem Addressed: Dijkstra's algorithm finds the shortest path from a given source vertex to all other vertices in a weighted graph with non-negative edge weights.

Steps of Dijkstra's Algorithm:

1. Set the distance to the source vertex to zero and to infinity for all other vertices.
2. Choose the vertex with the smallest tentative distance and explore its neighbors.
3. Update the distance for each neighbor if a shorter path is found.
4. Repeat until all vertices have been processed.

Time Complexity:

- Using a Binary Heap:  $O((V + E) \log V)$ 
  - Similar to Prim's algorithm, the time complexity depends on the number of edges  $E$  and vertices  $V$ .
  - Each vertex is processed once, and for each vertex, adjacent edges are added to the priority queue.
  - Extracting the minimum distance vertex takes  $O(\log V)$  time, and this happens  $V$  times, while each edge is relaxed once.
- Using an Adjacency Matrix:  $O(V^2)$ .
  - In the worst case, Dijkstra's algorithm iterates through all vertices and edges, resulting in quadratic time complexity when using an adjacency matrix.


Space Complexity:

- $O(V)$  for storing the distance table, priority queue, and the visited vertices.

Practical Performance:

- Efficient for Dense Graphs Using Heaps: Dijkstra's algorithm is efficient for dense graphs when used with binary heaps and adjacency lists.
- Sparse Graphs: In sparse graphs with fewer edges, the complexity  $O((V + E) \log V)$  is more favorable than  $O(V^2)$ , making it scalable for large networks, such as road networks or communication systems.



The background is a light blue grid. It is decorated with various hand-drawn blue doodles. In the top left, there are several overlapping circles. In the top center, there is a scribbled circle. In the top right, there are concentric circles and a star-like shape. On the right side, there are horizontal lines and a scribbled circle. In the bottom left, there are concentric circles and a scribbled circle. In the bottom center, there is a wavy line and a series of small 'v' marks. In the bottom right, there is a large, loose scribble.

**Thank you  
very much!**