Algorithms Handbook

Vladimir Bolshakov

Contents

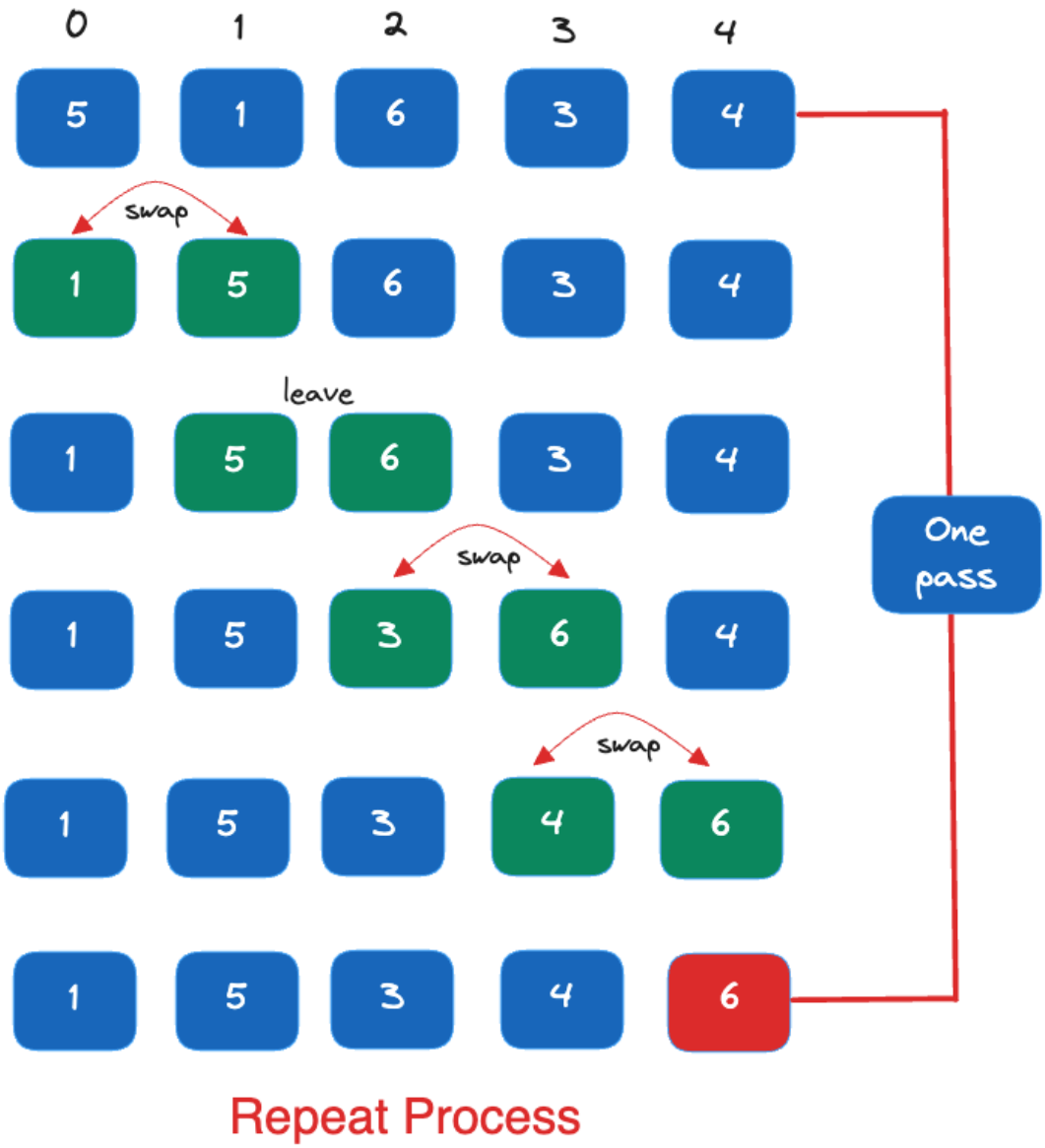
Bubble sort

Bubble sorting is one of the simplest sorting algorithms that is not used in practice, but is actively used for training purposes. It works by repeatedly going through the sorted list, comparing each pair of neighboring elements and replacing them if they are in the wrong order. The pass through the list is repeated until no permutations are needed, indicating that the list is sorted.

How it works:

1. **Comparing neighboring elements:**
    - The algorithm starts by comparing the first two elements of the array. If the first element is greater than the second, they are swapped. Otherwise, they remain in their positions.
2. **Iterative process:**
    - This process is then repeated for each pair of neighboring elements throughout the array. After the first iteration, the largest element will "pop up" to the last position.
3. **Next passes:**
    - The algorithm then repeats the process for the remaining elements (excluding those already sorted at the end of the array). At each pass, the next largest element is placed in the correct position.
4. **Termination:**
    - The algorithm terminates when a pass through the entire array is made without any swaps, indicating that the array is now sorted.

Time complexity: - Bubble sort has a time complexity of O(nˆ2) in the worst and average cases, where n is the number of elements in the array. This makes it inefficient for large datasets but useful for educational purposes because of its simplicity.




```
function bubbleSort(array: number[] | string[]) {
  for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length - 1 - i; j++) {
      if (array[j] > array[j + 1]) {
        [array[j], array[j + 1]] = [array[j + 1], array[j]];
      }
    }
  }
  return array;
}

console.log(bubbleSort([2, 5, 2, 6, 7, 2, 22, 5, 7, 9, 0, 2, 3]));
```

```
public static void bubbleSort(int[] array) {
    for(int i = 0; i < array.length - 1; i++) {
        for(int j = 0; j < array.length - i - 1; j++) {
            if(array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}
```

- Go back

Selection sort

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

How it works:

1. **Step 1:**
    - Set Min to location 0 in Step 1.
2. **Step 2**
    - Look for the smallest element on the list.
3. **Step 3:**
    - Replace the value at location Min with a different value.
4. **Step 4:**
    - Increase Min to point to the next element
5. **Step 5:**
    - Continue until the list is sorted.

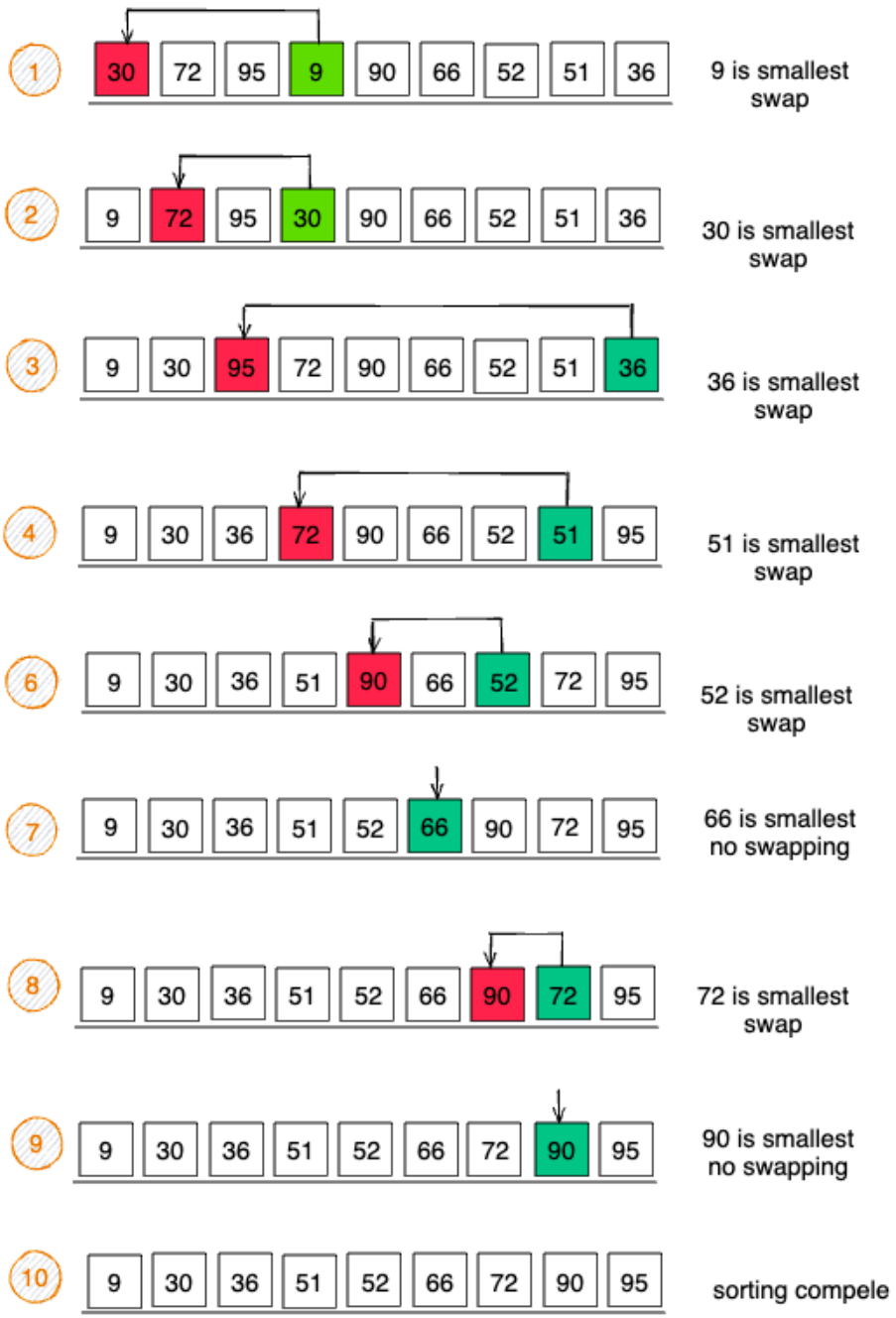
Selection sort

Time complexity

Best case: O(N^2)
Worst case: O(N^2)
Average case O(N^2)

Space complexity

Best case: O(1)
Worst case: O(N)
Average case O(N)




```
function selectionSort(array: any[]) {
  for (let i = 0; i < array.length - 1; i++) {
    let min = i;
    for (let j = i + 1; j < array.length; j++) {
      if (array[min] > array[j]) min = j;
```

```
    }
    [array[i], array[min]] = [array[min], array[i]];
  }
  return array;
}
```

```
    public static void selectionSort(int[] array) {
        for(int i = 0; i < array.length - 1; i++) {
            int min = i;
            for(int j = i + 1; j < array.length; j++) {
                if(array[min] > array[j]) {
                    min = j;
                }
            }
            int temp = array[i];
            array[i] = array[min];
            array[min] = temp;
        }
    }
```

```
print('This is selection sort')

def find_smallest(arr):
    smallest = arr[0]
    smallest_index = 0
    for i in range(1, len(arr)):
        if arr[i] < smallest:
            smallest = arr[i]
            smallest_index = i
    return smallest_index

def selection_sort(arr):
    newArr = []
    for i in range(len(arr)):
        smallest = find_smallest(arr)
        newArr.append(arr.pop(smallest))
    return newArr
```

- Go back

Insertion sort

Insertion Sort is a straightforward sorting algorithm that builds the sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, it has some advantages: it is simple to implement, efficient for small datasets, and performs well for partially sorted arrays.

How Insertion Sort Works:

1. **Dividing the Array:**
    - The algorithm starts with the first element of the array considered as the sorted part.

2. **Inserting Elements:**

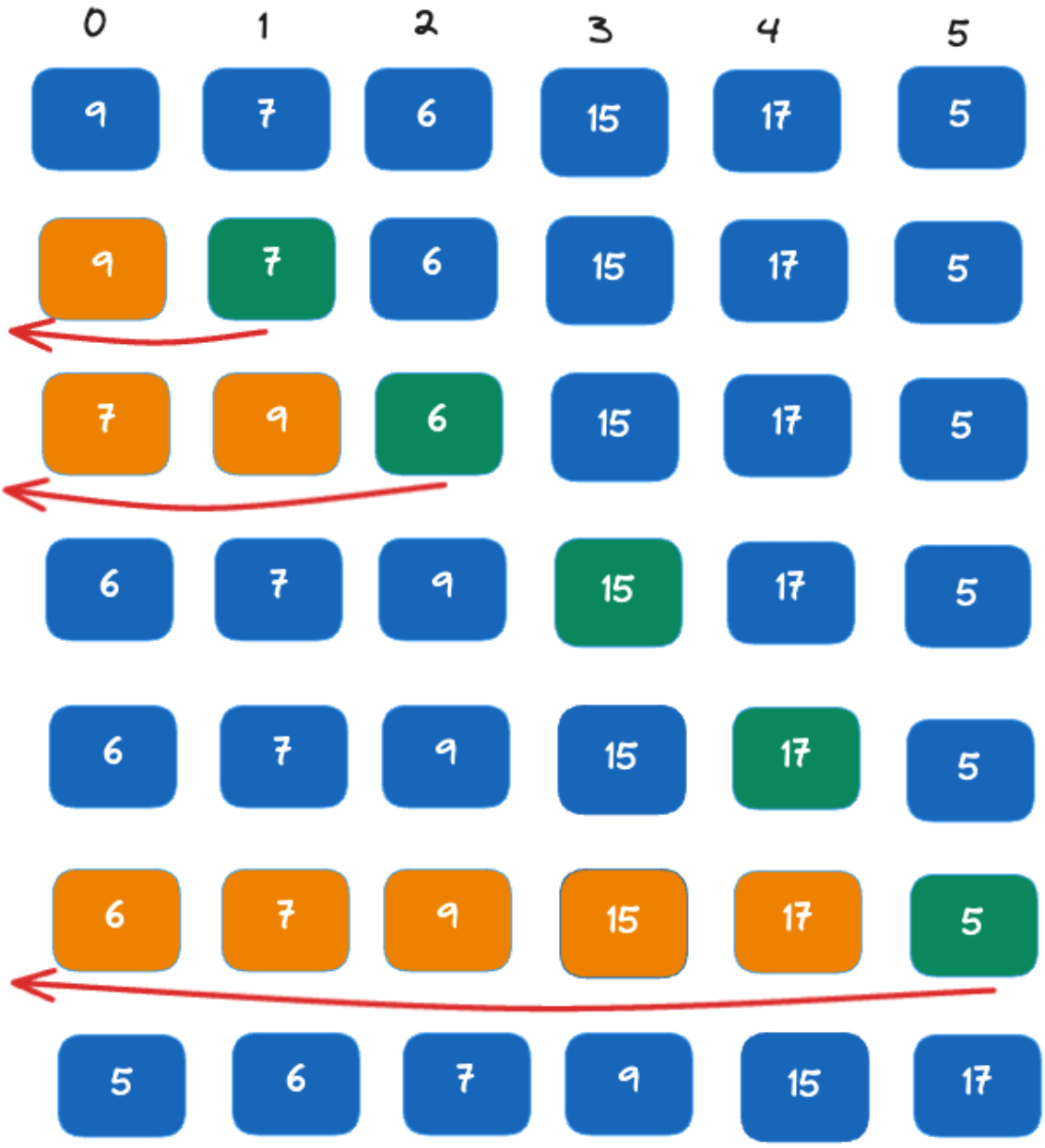
    - For each element in the unsorted part of the array, Insertion Sort compares it with the elements in the sorted part.
    - It then inserts the element into its correct position in the sorted part, shifting the other elements if necessary.

3. **Iterative Process:**

    - This process is repeated until all elements are sorted.

Time Complexity:

- Insertion Sort has a time complexity of O(n^2) in the worst case, where 'n' is the number of elements in the array. Despite its quadratic time complexity, Insertion Sort is often more efficient on small datasets or partially sorted arrays compared to other quadratic sorting algorithms. It's also an in-place sorting algorithm, meaning it doesn't require additional memory.




```
function insertionSort(array: number[] | string[]) {
    for (let i = 1; i < array.length; i++) {
        let curr = array[i];
        let j = i - 1;
        for (j; j >= 0 && array[j] > curr; j--) {
            array[j + 1] = array[j];
        }
        array[j + 1] = curr;
    }
    return array;
}
```

```
console.log(insertionSort([1, 4, 2, 8, 345, 123, 43, 32]));
```

```
    class Solution {
        void insertionSort (int[] arr) {
            int n = arr.length;
            for(int i = 1; i < n; i++) {
                int current = arr[i];
                int position = i - 1;
                while(position >= 0 && arr[position] > current) {
                    arr[position + 1] = arr[position];
                    position--;
                }
                arr[position + 1] = current;
            }
        }
    }
```

Quicksort

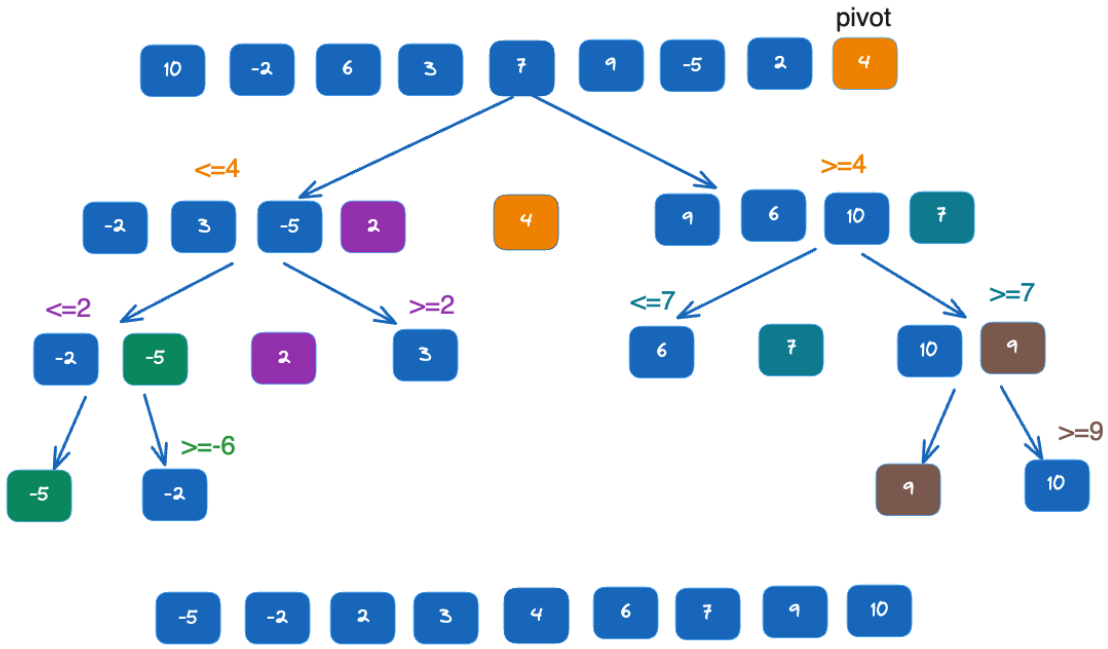
Quick Sort is an efficient, comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

How Quick Sort Works:

1. **Choosing a Pivot:**
    - The algorithm selects a pivot element from the array. The choice of pivot can affect the efficiency of the algorithm.
2. **Partitioning:**
    - Elements smaller than the pivot are moved to its left, and elements greater than the pivot are moved to its right. The pivot is now in its final sorted position.
3. **Recursive Sorting:**
    - The algorithm is applied recursively to the sub-arrays on the left and right of the pivot until the entire array is sorted.

Time Complexity:

- Quick Sort has an average and best-case time complexity of O(n log n), where 'n' is the number of elements in the array. In the worst case, it is O(n^2), but this is rare when a good pivot selection strategy is used. Quick Sort is often faster in practice than other O(n log n) algorithms, and it is widely used in various applications due to its efficiency.




```
class Solution {

    int makePartition(int [] arr, int low, int high) {
        int pivot = arr[high];
        int currentIndex = low - 1;
        for(int i = low; i < high; i++) {
            if(arr[i] < pivot) {
                currentIndex++;
                int temp = arr[i];
                arr[i] = arr[currentIndex];
                arr[currentIndex] = temp;
            }

        }
        int temp = arr[high];
        arr[high] = arr[currentIndex + 1];
        arr[currentIndex + 1] = temp;
        return currentIndex + 1;
    }

    void quicksort(int[] arr, int low, int high) {
        if(low < high) {
            int pivot = makePartition(arr, low, high);
            quicksort(arr, low, pivot - 1);
            quicksort(arr, pivot + 1, high);
        }
    }

    void quickSort (int[] arr) {
        int n = arr.length;
```

```
        quicksort(arr, 0, n - 1);
    }
}
```

```
def quicksort(arr):
    if len(arr) < 2:
        return arr
    else:
        pivot = arr[len(arr)/2]
        less = [i for i in arr[1:] if i <= pivot]
        greater = [i for i in arr[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

print(quicksort([10,2,3,1,5,4]))
```

```
class Solution {
    static void swap(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    private static void quickSort(int[] array, int start, int end) {
        if(end <= start) return; // base case

        int pivot = partition(array, start, end);

        quickSort(array, start, pivot -1);
        quickSort(array, pivot + 1, end);
    }

    private static int partition(int[] array, int start, int end) {
        int pivot = array[end];

        int i = start - 1;

        for(int j = start; j <= end -1; j++) {
            if(array[j] < pivot) {
                i++;
                swap(array, i, j);
            }
        }
        i++;
        swap(array, i, end);

        return i;
    }
}
```

```
function quicksort(arr: number[]): number[] {
  if (arr.length < 2) {
    return arr;
  } else {
```

```
    const pivot = arr[Math.floor(arr.length / 2)];
    const less = arr.slice(1).filter((i) => i <= pivot);
    const greater = arr.slice(1).filter((i) => i > pivot);
    return [...quicksort(less), pivot, ...quicksort(greater)];
  }
}
```

- Go back

Merge sort

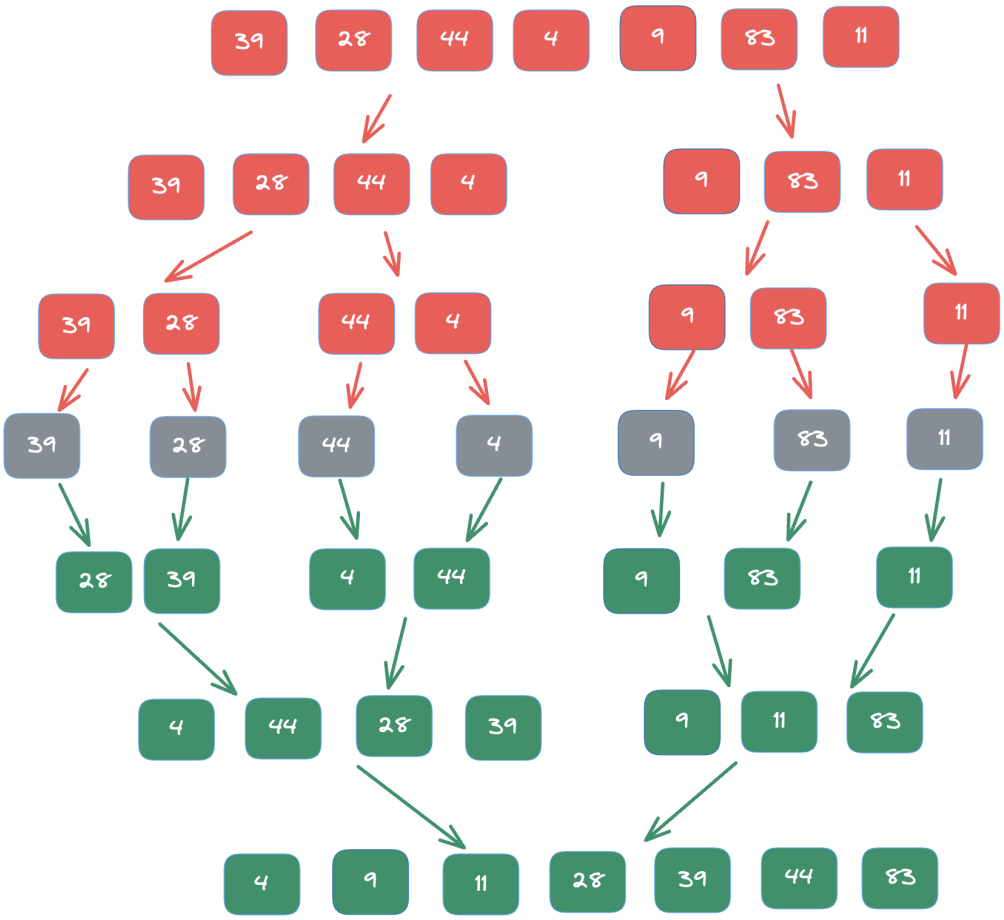
Merge Sort is a comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It works by dividing the unsorted array into 'n' sub-arrays, each containing one element. It then repeatedly merges these sub-arrays to produce new sorted sub-arrays until there is only one sub-array remaining – the fully sorted array.

How Merge Sort Works:

1. **Divide:**
    - The unsorted array is recursively divided into two halves until each sub-array contains only one element. This is the base case of the recursion.
2. **Conquer:**
    - The adjacent sub-arrays are then recursively merged to produce new sorted sub-arrays. This process continues until there is only one sub-array remaining – the fully sorted array.
3. **Merge:**
    - The key operation in Merge Sort is the merging of two sorted sub-arrays to produce a single, sorted sub-array. This involves comparing elements from the two sub-arrays and placing them in the correct order.

Time Complexity:

- Merge Sort has a consistent time complexity of O(n log n) in all cases, where 'n' is the number of elements in the array. It is a stable sorting algorithm, meaning that equal elements maintain their relative order in the sorted output. While Merge Sort has a slightly higher space complexity due to the need for additional memory, its stability and predictable performance make it a widely used and reliable sorting algorithm.




```
class Solution {

    void merge(int[] arr, int low, int mid, int high) {
        int subArr1Size = mid - low + 1;
        int subArr2Size = high - mid;

        int [] subArr1 = new int[subArr1Size];
        int [] subArr2 = new int[subArr2Size];

        for (int i = 0; i < subArr1Size; i++) {
           subArr1[i] = arr[low + i];
           }
           for (int i = 0; i < subArr2Size; i++) {
           subArr2[i] = arr[mid + 1 + i];
        }
        int i = 0, j = 0, k = low;

        while(i < subArr1Size && j < subArr2Size) {
            if(subArr1[i] <= subArr2[j]) {
                arr[k] = subArr1[i];
                i++;
```

```
            } else {
                arr[k] = subArr2[j];
                j++;
            }
            k++;
        }
        while(i < subArr1Size) {
            arr[k++] = subArr1[i++];
        }
        while (j < subArr2Size) {
           arr[k++] = subArr2[j++];
       }
    }

    void mergesort(int[] arr, int low, int high){
        if(high > low) {
            int mid = (high + low) / 2;
            mergesort(arr, low, mid);
            mergesort(arr, mid + 1, high);
            merge(arr, low, mid, high);
        }
    }

    void mergeSort (int[] arr) {
        int n = arr.length;
        mergesort(arr, 0, n - 1);
    }
}
```

```
function mergeSort(arr: number[]): number[] {
  if (arr.length <= 1) {
    return arr;
  }

  const middle = Math.floor(arr.length / 2);
  const left = arr.slice(0, middle);
  const right = arr.slice(middle);

  return merge(mergeSort(left), mergeSort(right));
}

function merge(left: number[], right: number[]): number[] {
  let result: number[] = [];
  let leftIndex = 0;
  let rightIndex = 0;

  while (leftIndex < left.length && rightIndex < right.length) {
    if (left[leftIndex] < right[rightIndex]) {
      result.push(left[leftIndex]);
      leftIndex++;
    } else {
      result.push(right[rightIndex]);
      rightIndex++;
```

```
    }
  }

  return result.concat(left.slice(leftIndex)).concat(right.slice(rightIndex));
}
```

Linear search

Linear Search, also known as sequential search, is a simple searching algorithm that finds the position of a target value within a list or array. It works by iterating through the elements one by one until the target value is found or the entire list has been searched.

How Linear Search Works:

1. **Start at the Beginning:**
    - Linear Search begins by looking at the first element in the list.
2. **Compare with Target:**
    - It compares the current element with the target value that we are searching for.
3. **Search Iteratively:**
    - If the current element is equal to the target value, the search is successful, and the index or position of the element is returned.
    - If the current element is not equal to the target value, the search continues by moving to the next element in the list.
    - This process is repeated until either the target value is found or the end of the list is reached.

Time Complexity:

The time complexity of Linear Search is O(n), where 'n' is the number of elements in the array. In the worst case, the algorithm may need to iterate through the entire list to find the target value. While Linear Search is simple, it may not be the most efficient for large datasets, especially when compared to more advanced search algorithms like binary search on sorted lists. However, it is easy to understand and implement.


```
function linearSearch(arr: number[], target: number): number {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i;
    }
  }

  return -1;
}
```

Jump search

Jump Search is a searching algorithm designed for sorted arrays. It is a block-based search algorithm that works by jumping ahead by fixed steps and then linearly searching within the block for the target element. Jump Search combines the efficiency of binary search with the simplicity of linear search.

How Jump Search Works:

1. **Determine Jump Size:**
    - Determine the jump size by taking the square root of the array length. This ensures a balanced trade-off between the number of jumps and the linear search within a block.

2. **Jump Ahead:**
    - Start at the beginning of the array and jump ahead by the calculated jump size until finding a value that is greater than or equal to the target.
3. **Linear Search within Block:**
    - Perform a linear search within the block from the previous jump until finding the target element or determining that it is not present in the block.
4. **Repeat or Conclude:**
    - Repeat the process until the entire array is searched or the target element is found.

Key Characteristics:

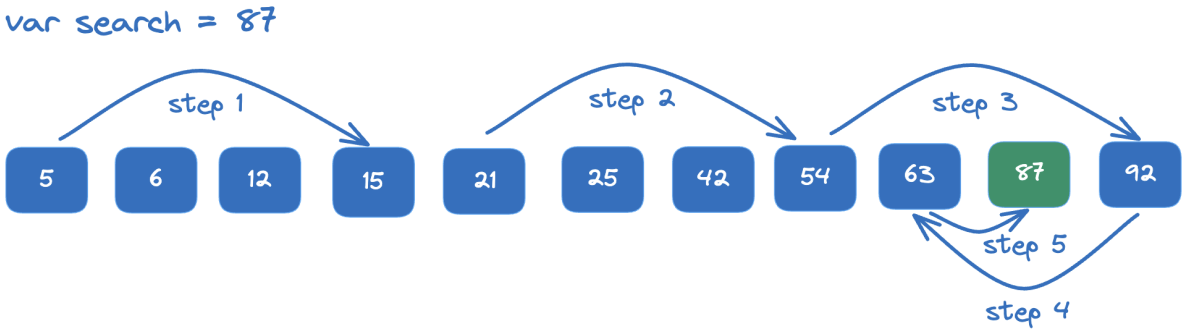
- **Efficiency:**
    - Jump Search has a time complexity of -add formula-, making it more efficient than linear search -add formula-, and comparable to binary search -add formula- for large datasets.
- **Sorted Arrays:**
    - Jump Search requires the array to be sorted.
- **Jump Size:**
    - The jump size is a critical factor in the efficiency of Jump Search. The optimal jump size is often calculated as the square root of the array length.

Applications:

- **Database Searching:**
    - Jump Search is used in database systems for searching in large datasets.
- **Sorted Arrays:**
    - Useful when the data is sorted, and random access to elements is feasible.

Time Complexity:

- The time complexity of Jump Search is -add formula-„ where-add formula-, is the size of the array. This makes it efficient for large datasets when compared to linear search but may be outperformed by binary search for certain scenarios.



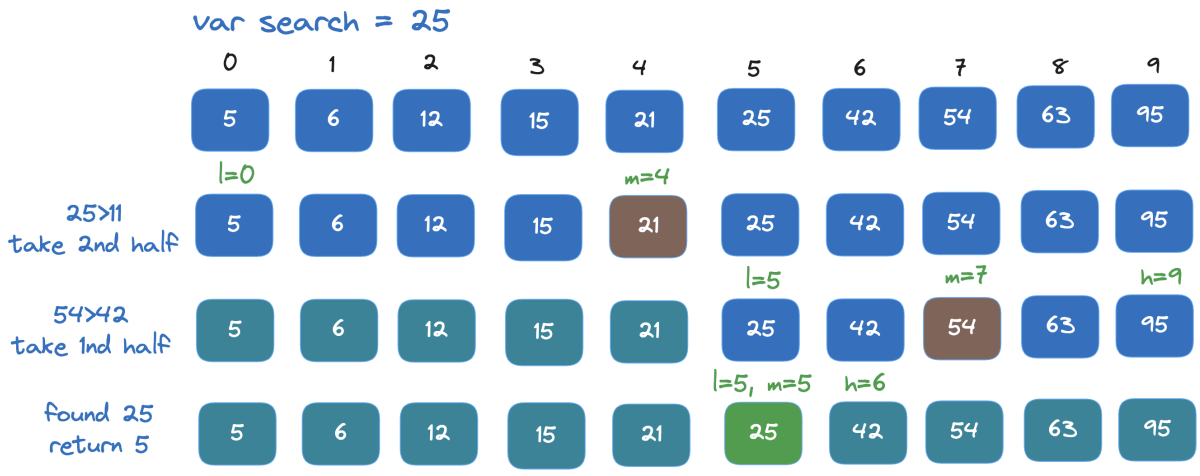
```
function jumpSearch(arr: number[], target: number): number {
  const n = arr.length;
  const step = Math.floor(Math.sqrt(n));
  let prev = 0;

  while (arr[Math.min(step, n) - 1] < target) {
    prev = step;
    step += Math.floor(Math.sqrt(n));
    if (prev >= n) {
      return -1;
    }
  }

  for (let i = prev; i < Math.min(step, n); i++) {
    if (arr[i] === target) {
      return i;
    }
  }

  return -1;
}
```

Binary search



Steps:

- Step 1 - Read the search element from the user.
- Step 2 - Find the middle element in the sorted list.
- Step 3 - Compare the search element with the middle element in the sorted list.
- Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.
- Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
- Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

- Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.
- Step 9 - If that element also doesn't match with the search element, then returns -1;

Time Complexity:

- Worst case: O(log n)
- Average case: O(log n)
- Best case: O(1)

```
function binarySearch(nums: number[], target: number): number {
  let left: number = 0;
  let right: number = nums.length - 1;

  while (left <= right) {
    const mid: number = Math.floor((left + right) / 2);

    if (nums[mid] === target) return mid;
    if (target < nums[mid]) right = mid - 1;
    else left = mid + 1;
  }

  return -1;
}
```

```
class Solution {
    private static int binarySearch(int[] array, int target) {

        int low = 0;
        int high = array.length - 1;

        while(low <= high) {
            int middle = low + (high - low) / 2;
            int value = array[middle];

            if(value < target) {
                low = middle + 1;
            } else if(value > target) {
                high = middle - 1;

            } else {
                return middle;
            }
        }
        return -1;
    }
}
```

```
def binary_search(list, item):
    low = 0
    high = len(list) - 1
    while low <= high:
        mid = (low+high)/2
        guess = list[mid]
        if guess == item:
            return mid
        if guess > item:
            high = mid - 1
        else:
            low = mid +1
    return None

my_list = [1, 3, 5, 7, 9]

res = binary_search(my_list, 3)

print(my_list[res])
```

Ternary search

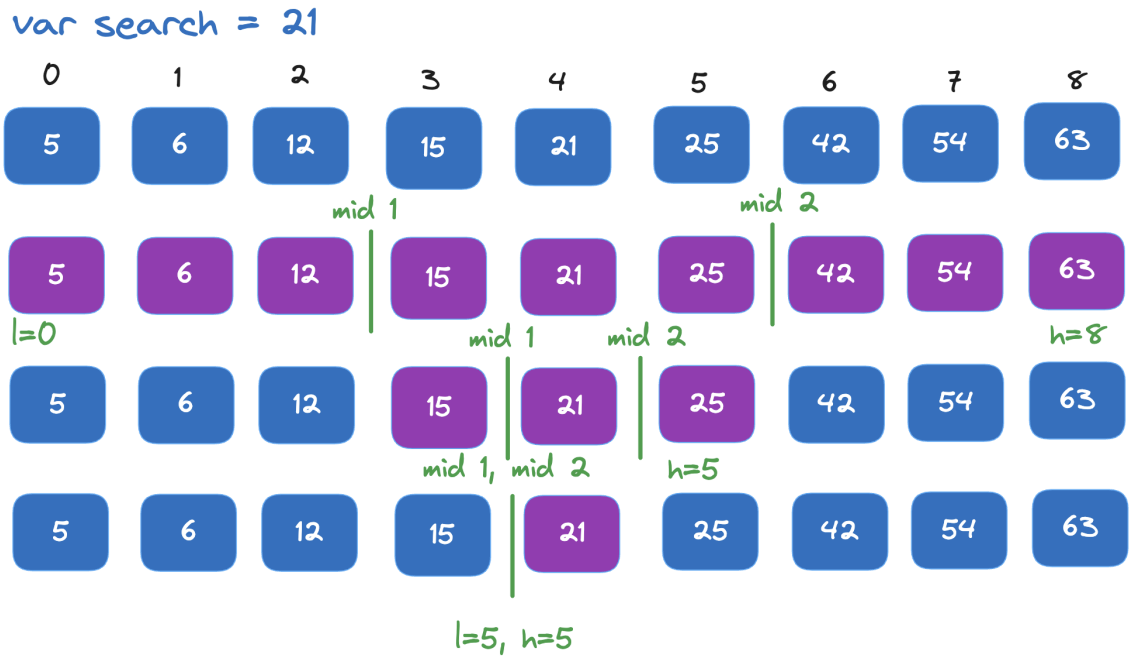
Ternary Search is a divide-and-conquer algorithm designed for efficiently finding the position of a target value in a sorted array. It operates by dividing the array into three parts and recursively narrowing down the search space until the target is found or determined to be absent.

How Ternary Search Works:

1. **Divide the Array:**
    - Ternary Search starts by dividing the sorted array into three parts.
2. **Compare with the Target:**
    - It then compares the target value with the elements at two points within the array, dividing it into three segments.
    - If the target is found at one of these points, the search is successful.
3. **Determine Search Space:**
    - Based on the comparisons, Ternary Search identifies whether the target lies in the first, second, or third segment of the array.
4. **Recursive Search:**
    - The algorithm then recursively applies the same process to the identified segment.
    - This recursion continues until the target is found or the search space is reduced to an empty array, indicating that the target is not present.

Time Complexity:

- Ternary Search has a time complexity of O(log3 n), where 'n' is the size of the array. This is an improvement over binary search when the search space can be significantly reduced at each step. However, it's worth noting that constant factors play a role, and in practice, binary search might be faster for smaller datasets due to simpler arithmetic operations. Ternary Search is particularly beneficial when the dataset is large and the search space can be significantly reduced with each iteration.




```
function ternarySearch(func: (x: number) => number, left: number, right: number, epsilon: number): numb
  while (right - left > epsilon) {
    const mid1 = left + (right - left) / 3;
    const mid2 = right - (right - left) / 3;

    const value1 = func(mid1);
    const value2 = func(mid2);

    if (value1 < value2) {
      left = mid1;
    } else {
      right = mid2;
    }
  }

  return (left + right) / 2;
}
```

Interpolation search

Interpolation Search is a an algorithm designed for finding a specific target value in a sorted array. Unlike linear or binary search, this algorithm utilizes the characteristics of the data distribution to make more

informed decisions about where to look for the target. It is particularly effective when the data has a uniform distribution.

How Interpolation Search Works:

1. **Linear Interpolation:**
   - Interpolation Search utilizes linear interpolation to estimate the likely position of the target value in the array.
2. **Estimate Position:**
   - Instead of evenly dividing the search space, as in binary search, Interpolation Search estimates the probable position of the target based on its value relative to the minimum and maximum values in the array.
3. **Calculation of Position:**
   - It calculates an estimate of the target's position by considering the relative location of the target with respect to the minimum and maximum values in the current search space.
4. **Refine Search:**
   - Based on the calculated estimate, the algorithm narrows down the search space and repeats the process until the target is found or the search space is exhausted.

Time Complexity:

The time complexity of Interpolation Search is O(log log n) on average, where "n" is the number of elements in the array. In the best case, it can be O(1), and in the worst case, it can be O(n). However, the average case is often more relevant, and it is O(log log n) under certain assumptions about the distribution of the data.


```
class Solution {

    private static int interpolationSearch(int[] array, int value) {
        int low = 0;
        int high = array.length - 1;

        while(value >=array[low] && value <= array[high] && low <= high) {
            int probe = low + (high - low) * (value - array[low]) / (array[high] - array[low]);
            if(array[probe] == value) {
                return probe;
            } else if(array[probe] > value) {
```

```
                low = probe + 1;
            } else {
                high = probe -1;
            }

        }

        return -1;
    }

}
```

```
function interpolationSearch(array: number[], value: number): number {
  let low = 0;
  let high = array.length - 1;

  while (value >= array[low] && value <= array[high] && low <= high) {
    const probe =
      low + ((high - low) * (value - array[low])) / (array[high] - array[low]);
    const roundedProbe = Math.floor(probe);

    if (array[roundedProbe] === value) {
      return roundedProbe;
    } else if (array[roundedProbe] < value) {
      low = roundedProbe + 1;
    } else {
      high = roundedProbe - 1;
    }
  }

  return -1;
}
```

Breadth-first search

Breadth-First Search (BFS) is a graph traversal algorithm that systematically explores all the vertices of a graph in breadthward motion, level by level. It starts at a chosen vertex and visits all its neighbors before moving on to their neighbors. BFS is commonly used to find the shortest path in an unweighted graph and to explore the structure of a graph.

How Breadth-First Search Works:

1. **Queue Initialization:**
    - Begin by selecting a starting vertex and enqueue it into a queue.
2. **Explore Neighbors:**
    - Dequeue a vertex from the front of the queue and explore all its neighbors.
    - Enqueue any unvisited neighbors, marking them as visited to avoid duplication.
3. **Level-wise Exploration:**

- Continue the process level by level, exploring all vertices at the current level before moving on to the next level.

4. **Termination:**
    - Repeat until the queue is empty, ensuring that all reachable vertices are visited.

Key Characteristics:

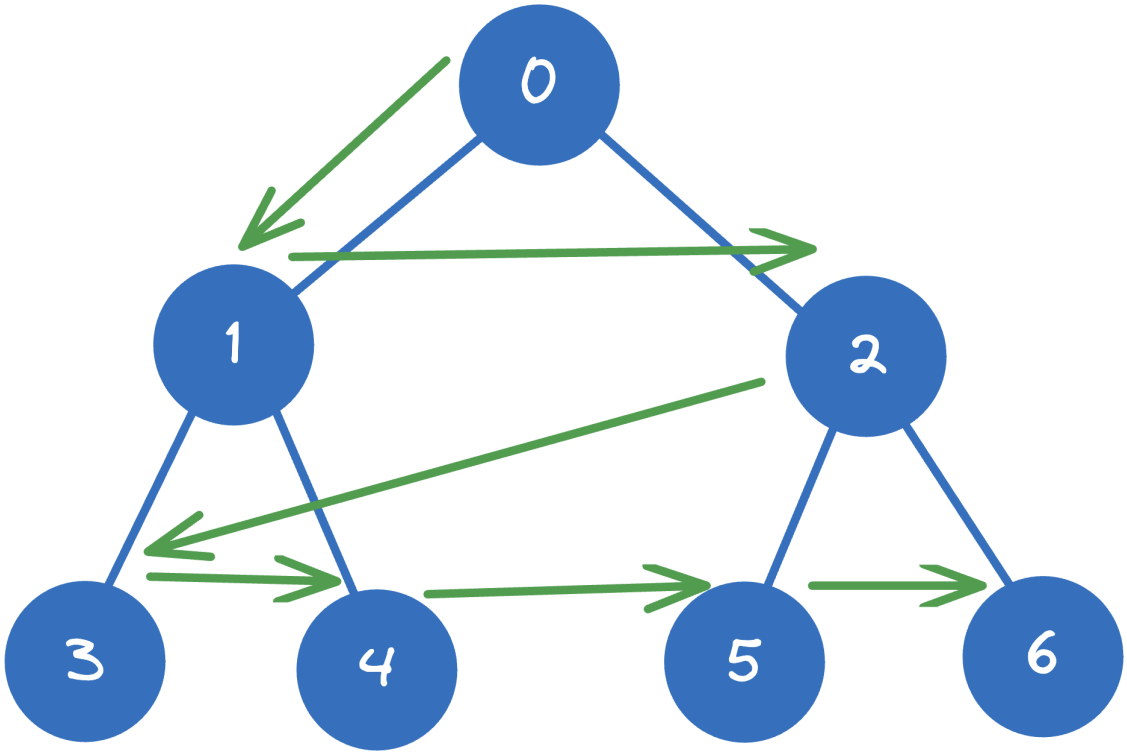
- **FIFO Structure:**
  - BFS uses a First-In-First-Out (FIFO) queue to maintain the order in which vertices are discovered and processed.
- **Visited Marking:**
  - To avoid revisiting vertices, mark each vertex as visited once it is dequeued from the queue.
- **Shortest Path:**
  - BFS guarantees that the shortest path to any reachable vertex is discovered first, making it valuable for pathfinding in unweighted graphs.

Applications:

- Shortest Pathfinding.
- Connected Components.
- Web Crawling.
- Network Broadcasting.

Time Complexity:

- The time complexity of BFS is (O(V + E)), where (V) is the number of vertices and (E) is the number of edges. The algorithm visits each vertex and edge once.




```
class Graph {
  private adjacencyList: Map<string, string[]>;

  constructor() {
    this.adjacencyList = new Map();
  }

  addVertex(vertex: string) {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, []);
    }
  }

  addEdge(vertex1: string, vertex2: string) {
    this.adjacencyList.get(vertex1)?.push(vertex2);
    this.adjacencyList.get(vertex2)?.push(vertex1);
  }

  bfs(startingVertex: string) {
    const visited: Set<string> = new Set();
    const queue: string[] = [];

    visited.add(startingVertex);
    queue.push(startingVertex);

    while (queue.length > 0) {
      const currentVertex = queue.shift()!;
      console.log(currentVertex);
```

```
      const neighbors = this.adjacencyList.get(currentVertex) || [];

      for (const neighbor of neighbors) {
        if (!visited.has(neighbor)) {
          visited.add(neighbor);
          queue.push(neighbor);
        }
      }
    }
  }
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");

graph.bfs("A");
```

Depth-first search

Depth-First Search (DFS) is a graph traversal algorithm that systematically explores the vertices of a graph by going as deep as possible along each branch before backtracking. It starts at a chosen vertex, explores as far as possible along one branch, and then backtracks to explore other branches. DFS is commonly used to detect cycles in a graph, topologically sort vertices, and solve problems related to connected components.

How Depth-First Search Works:

1. **Start at a Vertex:**
   - Begin by selecting a starting vertex and mark it as visited.
2. **Explore Neighbors:**
   - Move to an unvisited neighbor of the current vertex and repeat the process.
   - If all neighbors are visited, backtrack to the previous vertex.
3. **Recursion or Stack:**
   - DFS can be implemented using recursion or an explicit stack to keep track of the vertices to be visited.
4. **Marking and Backtracking:**
   - Mark each visited vertex to avoid revisiting and use backtracking to explore other branches.
5. **Complete Exploration:**
   - Continue the process until all reachable vertices are visited.

Key Characteristics:

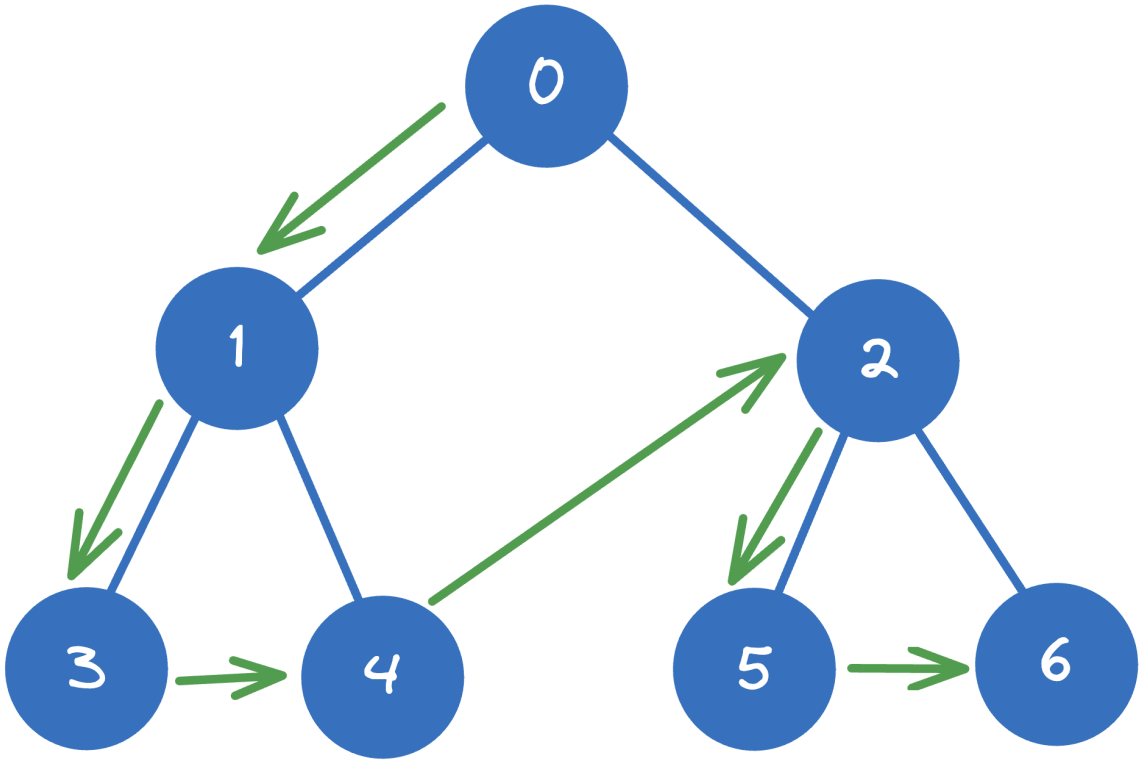
- **LIFO Structure:**
  - DFS often uses a Last-In-First-Out (LIFO) stack or recursion to maintain the order in which vertices are visited.
- **Visited Marking:**
  - Mark each vertex as visited once it is reached, preventing revisiting.
- **Backtracking:**
  - Backtrack to the previous vertex when all neighbors are explored.

Applications:

- Topological Sorting.
- Cycle Detection.
- Connected Components.
- Maze Solving.

Time Complexity:

- The time complexity of DFS is (O(V + E)), where (V) is the number of vertices and (E) is the number of edges. The algorithm visits each vertex and edge once. Recursive DFS has a space complexity of (O(V)) due to the call stack, while an explicit stack implementation can have a space complexity of (O(E + V)).



```
class Graph {
  private adjacencyList: Map<string, string[]>;

  constructor() {
    this.adjacencyList = new Map();
  }

  addVertex(vertex: string) {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, []);
    }
  }

  addEdge(vertex1: string, vertex2: string) {
    this.adjacencyList.get(vertex1)?.push(vertex2);
    this.adjacencyList.get(vertex2)?.push(vertex1);
  }

  dfs(startingVertex: string) {
    const visited: Set<string> = new Set();

    const dfsHelper = (vertex: string) => {
      console.log(vertex);
      visited.add(vertex);

      const neighbors = this.adjacencyList.get(vertex) || [];

      for (const neighbor of neighbors) {
        if (!visited.has(neighbor)) {
          dfsHelper(neighbor);
        }
      }
    };

    dfsHelper(startingVertex);
  }
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");

graph.dfs("A");
```

Dijkstra's algorithm

How Dijkstra's Algorithm Works:

1. **Initialization:**
   - Set the initial distance to the starting vertex as 0 and all other distances to infinity.
   - Create a priority queue or a min-heap to store vertices based on their current tentative distances.
2. **Explore Neighbors:**
   - While there are vertices in the priority queue, select the vertex with the smallest tentative distance.
   - Explore its neighbors and update their tentative distances if a shorter path is found.
3. **Relaxation:**
   - For each neighbor, calculate the sum of the tentative distance to the current vertex and the weight of the edge between them.
   - If this sum is smaller than the current tentative distance to the neighbor, update the tentative distance.
4. **Mark as Visited:**
   - Mark the current vertex as visited to avoid redundant calculations.
5. **Repeat:**
   - Repeat steps 2-4 until all vertices are visited or the destination vertex is reached.
6. **Result:**
   - The final result is an array of shortest distances from the starting vertex to all other vertices.

Key Characteristics:

- **Greedy Strategy:**
  - Dijkstra's Algorithm employs a greedy strategy, always choosing the vertex with the smallest tentative distance.
- **Priority Queue or Min-Heap:**
  - Efficient implementations use a priority queue or min-heap to efficiently retrieve the vertex with the smallest tentative distance.
-

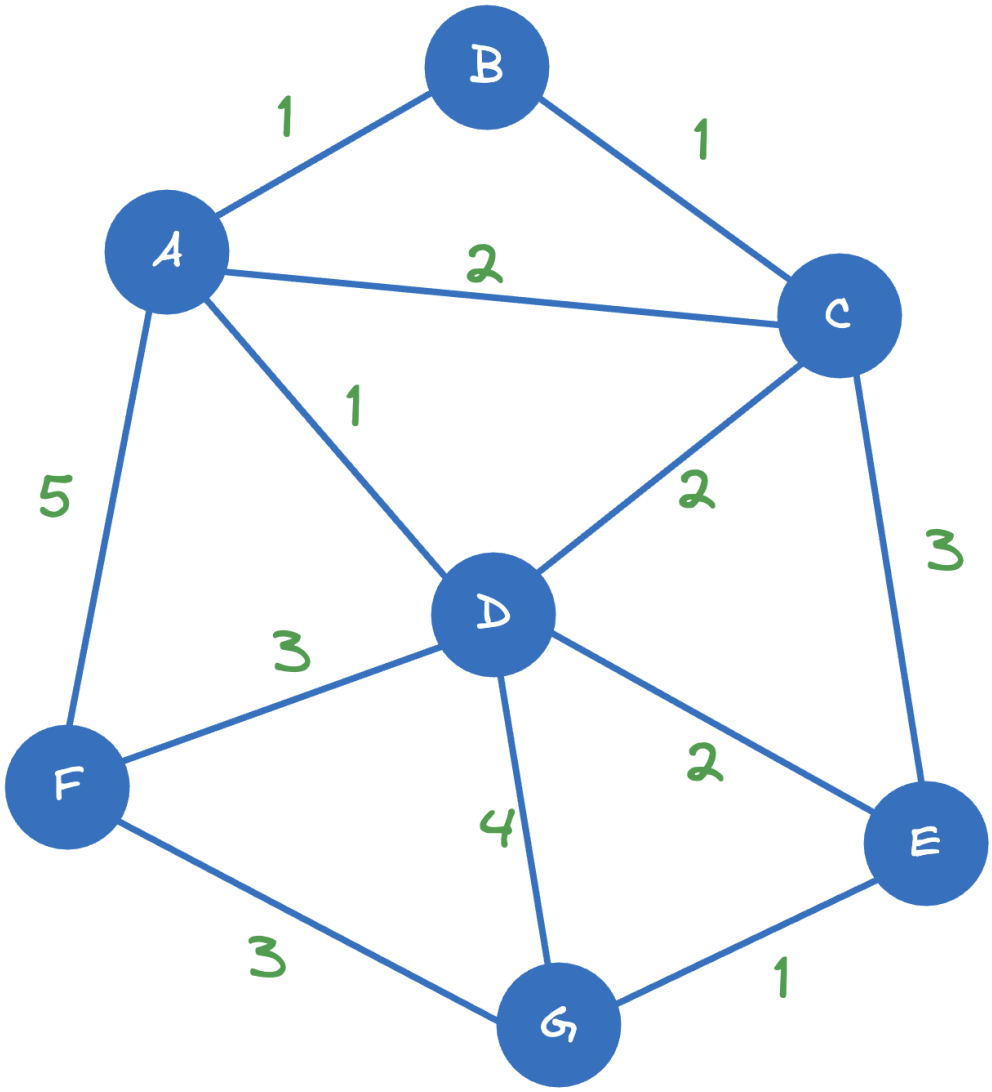
Non-Negative Edge Weights: - Dijkstra's Algorithm assumes non-negative edge weights. Negative weights can lead to incorrect results.

Applications:

- Network Routing.
- Shortest Path Problems.
- Transportation and Logistics.

Time Complexity:

- The time complexity of Dijkstra's Algorithm is O((V + E) log V) using a priority queue or min-heap, where V is the number of vertices and E is the number of edges.



```
class Graph {
  private adjacencyList: Map<string, Map<string, number>>;

  constructor() {
    this.adjacencyList = new Map();
  }

  addVertex(vertex: string) {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, new Map());
    }
  }

  addEdge(vertex1: string, vertex2: string, weight: number) {
    this.adjacencyList.get(vertex1)?.set(vertex2, weight);
    this.adjacencyList.get(vertex2)?.set(vertex1, weight);
  }

  dijkstra(startingVertex: string) {
    const distances: Map<string, number> = new Map();
    const previous: Map<string, string | null> = new Map();
    const priorityQueue = new PriorityQueue();

    for (const vertex of this.adjacencyList.keys()) {
      distances.set(vertex, vertex === startingVertex ? 0 : Infinity);
      previous.set(vertex, null);
      priorityQueue.enqueue(vertex, distances.get(vertex)!);
    }

    while (!priorityQueue.isEmpty()) {
      const currentVertex = priorityQueue.dequeue()!;
      const neighbors = this.adjacencyList.get(currentVertex);

      if (neighbors) {
        for (const neighbor of neighbors.keys()) {
          const distance = distances.get(currentVertex)! + neighbors.get(neighbor)!;

          if (distance < distances.get(neighbor)!) {
            distances.set(neighbor, distance);
            previous.set(neighbor, currentVertex);
            priorityQueue.enqueue(neighbor, distance);
          }
        }
      }
    }

    return { distances, previous };
  }

  shortestPath(startingVertex: string, targetVertex: string) {
    const { distances, previous } = this.dijkstra(startingVertex);

    const path: string[] = [];
```

```
    let currentVertex = targetVertex;

    while (currentVertex !== null) {
      path.unshift(currentVertex);
      currentVertex = previous.get(currentVertex)!;
    }

    return { path, distance: distances.get(targetVertex) };
  }
}

class PriorityQueue {
  private items: [string, number][] = [];

  enqueue(element: string, priority: number) {
    this.items.push([element, priority]);
    this.sort();
  }

  dequeue() {
    return this.items.shift();
  }

  isEmpty() {
    return this.items.length === 0;
  }

  private sort() {
    this.items.sort((a, b) => a[1] - b[1]);
  }
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B", 1);
graph.addEdge("A", "C", 4);
graph.addEdge("B", "C", 2);
graph.addEdge("B", "D", 5);
graph.addEdge("C", "D", 1);

const { path, distance } = graph.shortestPath("A", "D");
console.log("Shortest Path:", path); // Output: Shortest Path: [ 'A', 'B', 'C', 'D' ]
console.log("Distance:", distance); // Output: Distance: 4
```

Floyd-Warshall algorithm

The Floyd-Warshall Algorithm is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. Unlike Dijkstra's algorithm and Bellman-Ford algorithm, Floyd-

Warshall works with graphs that can have both positive and negative edge weights and can handle graphs with cycles. The algorithm iteratively updates the shortest path distances between all pairs until reaching the optimal solution.

How Floyd-Warshall Algorithm Works:

1. **Initialization:**
    - Create a matrix to represent the distances between all pairs of vertices. Initialize the matrix with the direct edge weights and set the distances to infinity where there is no direct edge.
    - Initialize the diagonal of the matrix to zeros since the distance from a vertex to itself is zero.
2. **Iterative Updates:**
    - For each vertex 'k', iterate through all pairs of vertices 'i' and 'j'.
    - Check if the path from 'i' to 'j' through 'k' is shorter than the current known path from 'i' to 'j'.
    - If yes, update the distance from 'i' to 'j' with the shorter path.
3. **Repeat:**
    - Repeat the process for all vertices as intermediate vertices ('k').
    - After each iteration, the matrix reflects the shortest distances between all pairs of vertices considering all possible intermediate vertices.
4. **Result:**
    - The final matrix contains the shortest distances between all pairs of vertices.

Key Characteristics:

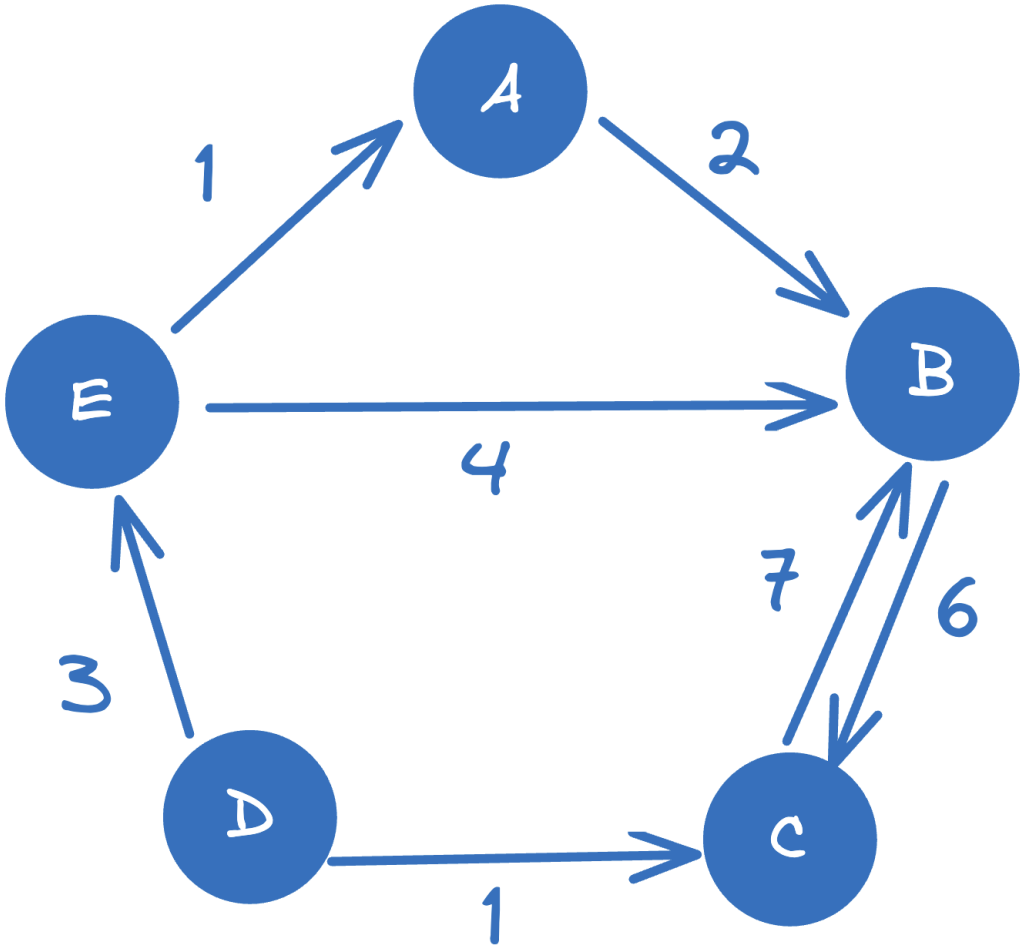
- **Dynamic Programming:**
    - Floyd-Warshall is a dynamic programming algorithm that builds solutions to subproblems to solve the overall problem.
- **Negative Cycles:**
    - The algorithm can detect negative cycles in the graph. If there exists a negative cycle, the algorithm won't converge to a solution.

Applications:

- Network Routing.
- Shortest Path Problems.
- Traffic Engineering.

Time Complexity:

- The time complexity of Floyd-Warshall Algorithm is (O(V^3)), where (V) is the number of vertices in the graph.




```
class Graph {
  private adjacencyMatrix: number[][];

  constructor(numVertices: number) {
    this.adjacencyMatrix = Array.from({ length: numVertices }, () =>
      Array(numVertices).fill(Infinity)
    );

    // Set diagonal elements to 0
    for (let i = 0; i < numVertices; i++) {
      this.adjacencyMatrix[i][i] = 0;
    }
  }

  addEdge(source: number, destination: number, weight: number) {
    this.adjacencyMatrix[source][destination] = weight;
  }
```

```
  floydWarshall() {
    const numVertices = this.adjacencyMatrix.length;

    for (let k = 0; k < numVertices; k++) {
      for (let i = 0; i < numVertices; i++) {
        for (let j = 0; j < numVertices; j++) {
          if (
            this.adjacencyMatrix[i][k] + this.adjacencyMatrix[k][j] <
            this.adjacencyMatrix[i][j]
          ) {
            this.adjacencyMatrix[i][j] =
              this.adjacencyMatrix[i][k] + this.adjacencyMatrix[k][j];
          }
        }
      }
    }

    return this.adjacencyMatrix;
  }
}

// Example usage:
const graph = new Graph(4);

graph.addEdge(0, 1, 3);
graph.addEdge(0, 2, 6);
graph.addEdge(1, 2, 1);
graph.addEdge(1, 3, 4);
graph.addEdge(2, 3, 2);

const result = graph.floydWarshall();

console.log("Shortest Path Matrix:");
for (const row of result) {
  console.log(row);
}
```

Ford Fulkerson algorithm

The Ford-Fulkerson Algorithm is an iterative method to compute the maximum flow in a flow network. It was initially designed to solve the max-flow min-cut problem, where the objective is to find the maximum amount of flow that can be sent from a designated source to a designated sink in a directed graph. The algorithm iteratively augments paths from the source to the sink, increasing the flow until it reaches its maximum value.

How Ford-Fulkerson Algorithm Works:

1. **Initialization:**
   - Begin with an initial flow of zero.
   - Determine the residual graph, which represents the remaining capacity for each edge.
2. **Augmenting Paths:**

- Find an augmenting path from the source to the sink in the residual graph. An augmenting path is a path with available capacity on all its edges.

3. **Flow Augmentation:**
   - Determine the maximum flow that can be added along the augmenting path. This is the minimum capacity value of the edges on the path.
4. **Update Residual Graph:**
   - Update the residual graph by subtracting the flow added along the augmenting path and adding the reverse flow.
5. **Repeat:**
   - Repeat steps 2-4 until there are no more augmenting paths.
6. **Result:**
   - The final flow is the maximum flow in the network.

Key Characteristics:

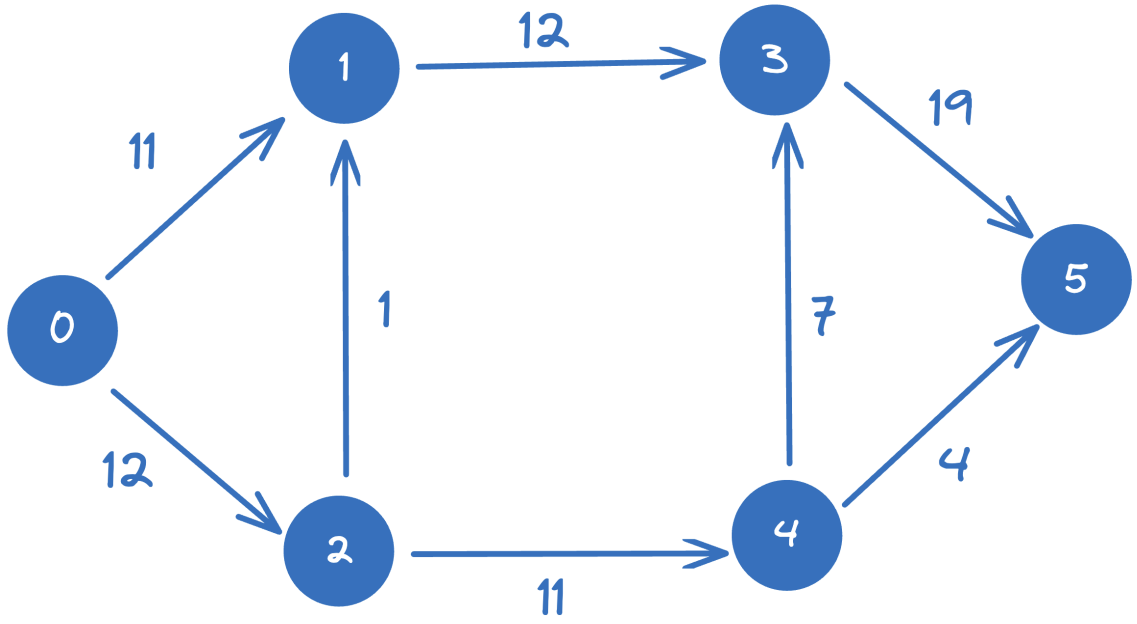
- **Residual Graph:**
  - The residual graph is crucial for the Ford-Fulkerson Algorithm. It represents the remaining capacity for each edge after the initial flow has been determined.
- **Augmenting Paths:**
  - The algorithm focuses on finding augmenting paths, paths in the residual graph with available capacity.
- **Termination:**
  - The algorithm terminates when no more augmenting paths can be found in the residual graph.

Applications:

- Network Flows.
- Transportation Networks.
- Telecommunication Networks.

Time Complexity:

- The time complexity of the Ford-Fulkerson Algorithm is not strictly defined, as it depends on the choice of augmenting paths. In the worst case, the algorithm may not terminate if the paths are not chosen carefully. When implemented with the Edmonds-Karp variant, where the shortest augmenting paths are chosen using Breadth-First Search, the time complexity is (O(VEˆ2)), where (V) is the number of vertices and (E) is the number of edges.




```
class FordFulkerson {
  private graph: number[][];
  private numVertices: number;

  constructor(graph: number[][]) {
    this.graph = graph;
    this.numVertices = graph.length;
  }

  fordFulkerson(source: number, sink: number): number {
    let maxFlow = 0;

    // Create a residual graph and initialize it with the original capacities.
    const residualGraph = this.graph.map((row) => [...row]);

    while (true) {
      const path = this.bfs(source, sink, residualGraph);
      if (!path) {
        break; // No augmenting path found, terminate the algorithm
      }

      // Find the minimum capacity along the augmenting path
      let minCapacity = Number.POSITIVE_INFINITY;
      for (let i = 0; i < path.length - 1; i++) {
        const u = path[i];
        const v = path[i + 1];
        minCapacity = Math.min(minCapacity, residualGraph[u][v]);
      }

      // Update residual capacities and reverse edges along the path
```

```
      for (let i = 0; i < path.length - 1; i++) {
        const u = path[i];
        const v = path[i + 1];
        residualGraph[u][v] -= minCapacity;
        residualGraph[v][u] += minCapacity;
      }

      // Add the flow of the augmenting path to the total flow
      maxFlow += minCapacity;
    }

    return maxFlow;
  }

  bfs(source: number, sink: number, graph: number[][]): number[] | null {
    const visited: boolean[] = new Array(this.numVertices).fill(false);
    const queue: number[] = [source];
    const parent: number[] = new Array(this.numVertices).fill(-1);

    while (queue.length > 0) {
      const u = queue.shift()!;

      for (let v = 0; v < this.numVertices; v++) {
        if (!visited[v] && graph[u][v] > 0) {
          queue.push(v);
          parent[v] = u;
          visited[v] = true;
        }
      }
    }

    if (!visited[sink]) {
      return null; // No augmenting path found
    }

    const path: number[] = [];
    for (let v = sink; v !== source; v = parent[v]) {
      path.unshift(v);
    }
    path.unshift(source);

    return path;
  }
}

// Example usage:
const graph = [
  [0, 16, 13, 0, 0, 0],
  [0, 0, 10, 12, 0, 0],
  [0, 4, 0, 0, 14, 0],
  [0, 0, 9, 0, 0, 20],
  [0, 0, 0, 7, 0, 4],
  [0, 0, 0, 0, 0, 0],
```

```
];

const fordFulkerson = new FordFulkerson(graph);
const maxFlow = fordFulkerson.fordFulkerson(0, 5);
console.log("Maximum Flow:", maxFlow);
```