Mastering Sorting Algorithms In Javascript

Sayyed Siddique
Frontend Developer

Swipe

1 Introduction

Sorting algorithms are fundamental in optimizing performance. In this post, we'll explore:

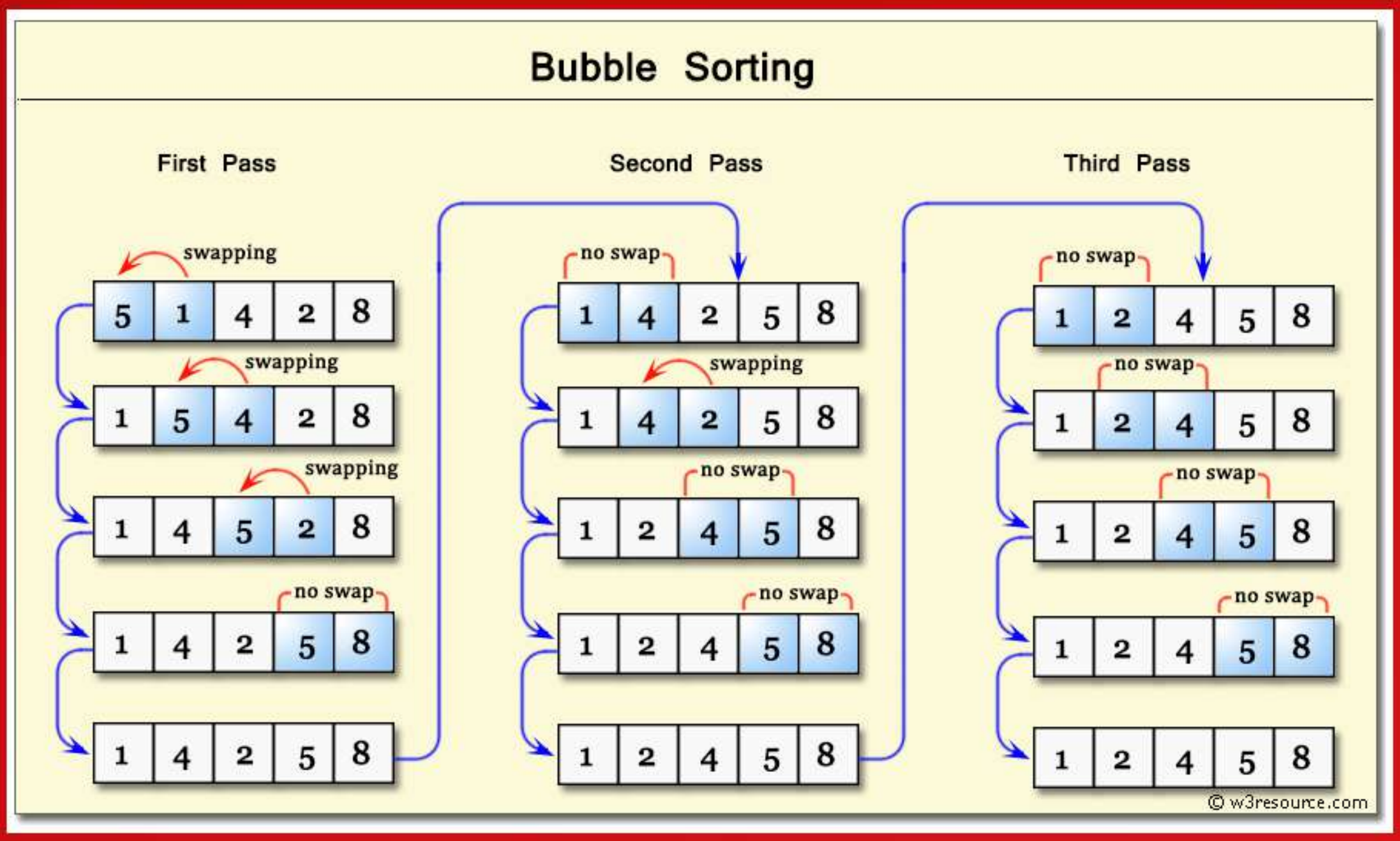
1. **Bubble Sort**
2. **Insertion Sort**
3. **Selection Sort**
4. **Merge Sort**
5. **Quick Sort**

Sayyed Siddique
Frontend Developer

Swipe

2 Bubble Sort

Bubble Sort is one of the simplest sorting algorithms. It works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The process repeats until the list is sorted. The name "bubble sort" comes from the way larger elements "bubble" to the top of the list, or the end of the array, after each pass through the list.



Sayyed Siddique
Frontend Developer

Swipe

3 Bubble Sort Example

Sorting Algorithms

```javascript
function bubbleSort(arr) {
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]]; // Swap
      }
    }
  }
  return arr;
}

// Example Call
console.log(bubbleSort([5, 2, 9, 1, 5, 6])); // Output: [1, 2, 5, 5, 6, 9]
```

Explanation:

- The outer loop runs through the array to make sure every element is checked.
- The inner loop compares adjacent elements and swaps them if the first one is larger than the second one.
- The process repeats until the array is sorted.
- The swapped flag helps optimize the algorithm by stopping early if no swaps were made in an inner loop, meaning the array is already sorted.

Sayyed Siddique
Frontend Developer

Swipe

4 Insertion Sort

Insertion Sort inserts each element into its correct position in the already sorted part of the array. It's efficient for small or nearly sorted arrays.

How it works:

1. Start from the second element of the array (assuming the first is already sorted).
2. Compare this element to the elements before it and insert it into its correct position by shifting the larger elements one position to the right.
3. Repeat this process for each element until the entire array is sorted.

Sayyed Siddique
Frontend Developer

Swipe

5 Insertion Sort Example Explanation

Sorting Algorithms

```javascript
function insertionSort(arr) {
    for (let i = 1; i < arr.length; i++) {
        let key = arr[i];  // Current element to be inserted
        let j = i - 1;

        // Shift elements of arr[0...i-1] that are greater than key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];  // Shift element to the right
            j--;
        }
        arr[j + 1] = key;  // Insert the key in its correct position
    }
    return arr;
}

// Example usage:
let arr = [12, 11, 13, 5, 6];
console.log("Sorted Array:", insertionSort(arr));

// Output: Sorted Array: [5, 6, 11, 12, 13]
```

Explanation:

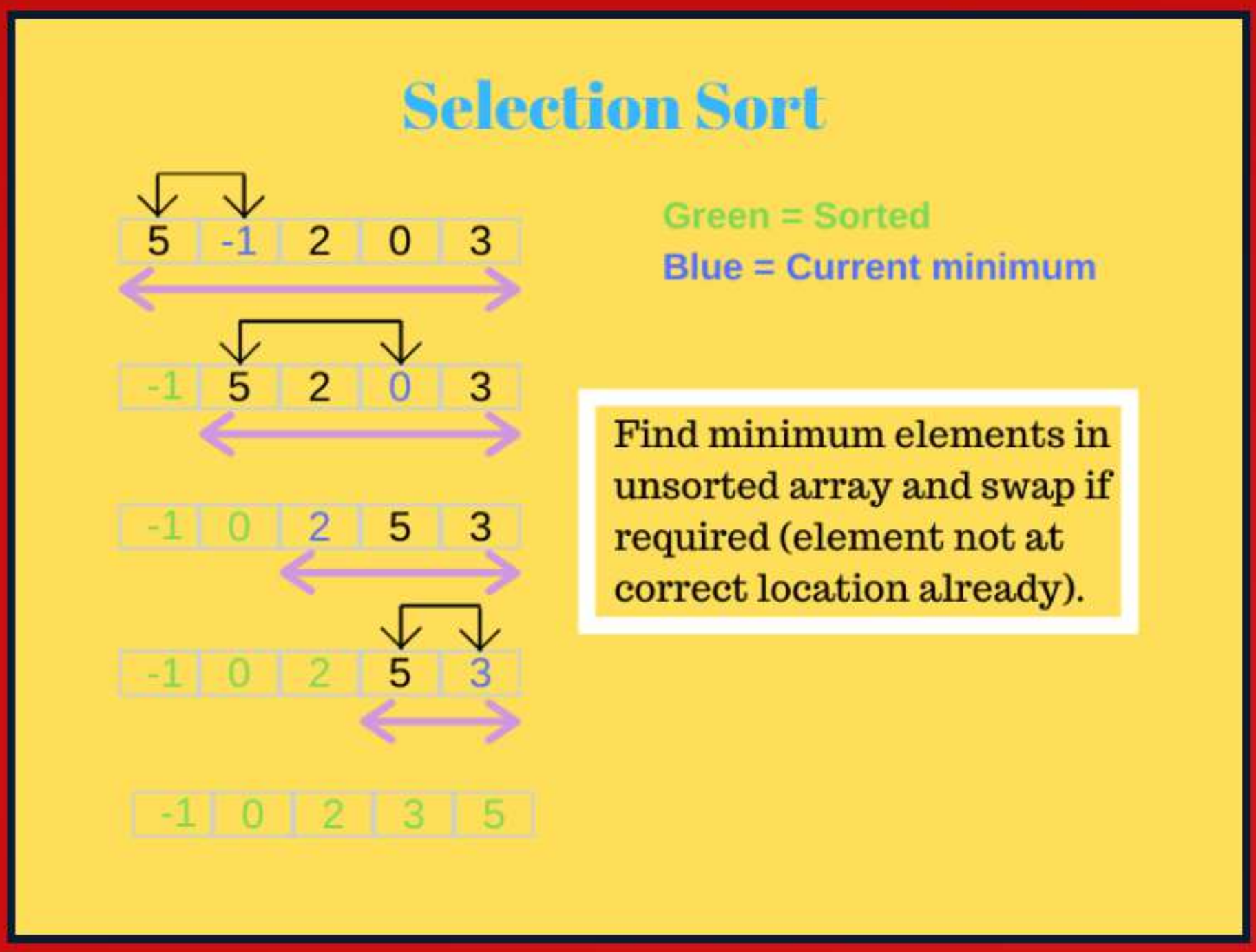
1. Key: At each step, the key is the current element to be inserted into the sorted portion of the array.
2. Shifting: All elements that are greater than the key are shifted one position to the right.
3. Insertion: The key is then placed in its correct position after the shifting is done.

Sayyed Siddique
Frontend Developer

Swipe

6 Selection Sort

Description: Selection Sort is a simple comparison-based sorting algorithm. It works by dividing the input array into two parts: the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty, and the unsorted part is the entire array. The algorithm repeatedly selects the smallest (or largest, depending on sorting order) element from the unsorted part, swaps it with the leftmost unsorted element, and moves the boundary between the sorted and unsorted parts one element to the right.



Sayyed Siddique
Frontend Developer

Swipe

7 Selection Sort Example

Sorting Algorithms

```javascript
function selectionSort(arr) {
  for (let i = 0; i < arr.length; i++) {
    let minIndex = i;
    for (let j = i + 1; j < arr.length; j++) {
      if (arr[j] < arr[minIndex]) {
        minIndex = j;
      }
    }
    if (minIndex !== i) {
      [arr[i], arr[minIndex]] = [arr[minIndex], arr[i]]; // Swap
    }
  }
  return arr;
}

// Example Call
console.log(selectionSort([64, 25, 12, 22, 11])); // Output: [11, 12, 22, 25, 64]
```

Sayyed Siddique
Frontend Developer

Swipe

7 Selection Sort Example Explaination

How it works:

1. Start with the first element and assume it's the smallest.
2. Compare this element with the rest of the unsorted array.
3. If you find a smaller element, note its index.
4. Once the entire unsorted array has been checked, swap the first element with the smallest element found.
5. Move the boundary between the sorted and unsorted arrays.
6. Repeat the process for the next element until the array is fully sorted.

Time Complexity:

- Worst-case: $O(n^2)$
- Best-case: $O(n^2)$ (even if the array is sorted, the algorithm still goes through all comparisons)
- Space Complexity: O(1) (in-place sorting)

Sayyed Siddique
Frontend Developer

Swipe

8 Merge Sort

Merge Sort is a classic, efficient, and stable sorting algorithm based on the divide-and-conquer approach. It recursively splits the array into two halves, sorts each half, and then merges the two sorted halves back together. The time complexity is O(n log n) for all cases (best, worst, and average), which makes it an efficient choice compared to other algorithms like bubble sort or insertion sort.

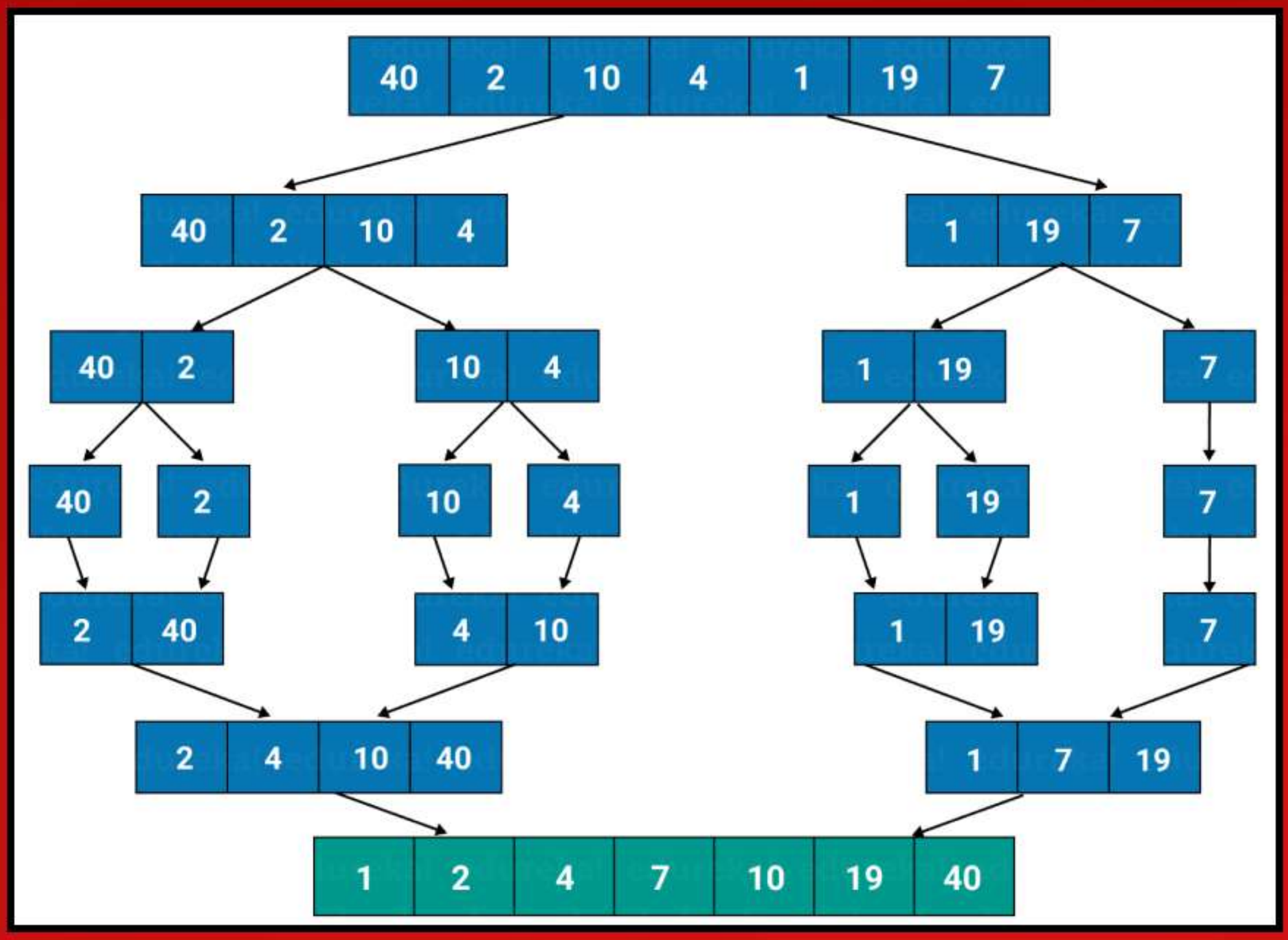
Steps of Merge Sort:

1. Divide: Split the array into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge the two sorted halves into one sorted array.

Sayyed Siddique
Frontend Developer

Swipe

8 Merge Sort Demo



Sayyed Siddique
Frontend Developer

Swipe

9 Merge Sort Example

Sorting Algorithms

```javascript
function mergeSort(arr) {
  if (arr.length <= 1) {
    return arr;  // Base case: array with 1 or 0 elements is already sorted
  }

  // Split array into two halves
  const middle = Math.floor(arr.length / 2);
  const left = arr.slice(0, middle);
  const right = arr.slice(middle);

  // Recursively sort both halves and merge them
  return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right) {
  let sortedArray = [];
  let i = 0;
  let j = 0;

  // Compare each element from both arrays and merge in sorted order
  while (i < left.length && j < right.length) {
    if (left[i] < right[j]) {
      sortedArray.push(left[i]);
      i++;
    } else {
      sortedArray.push(right[j]);
      j++;
    }
  }

  // Concatenate any remaining elements from both halves
  return sortedArray.concat(left.slice(i)).concat(right.slice(j));
}

// Example usage:
const arr = [38, 27, 43, 3, 9, 82, 10];
const sortedArr = mergeSort(arr);
console.log(sortedArr);  // Output: [3, 9, 10, 27, 38, 43, 82]
```

Sayyed Siddique
Frontend Developer

Swipe

10 Quick Sort

Quick Sort is an efficient sorting algorithm based on the divide-and-conquer strategy. It operates by selecting a "pivot" element from the array and partitioning the other elements into two groups: those less than the pivot and those greater than or equal to the pivot. The process is recursively applied to the two subarrays, resulting in a sorted array.

Key Steps of Quick Sort:

Choose Pivot: Pick a pivot element (commonly the first or last element).
Partition: Divide the array into two subarrays: elements less than the pivot on the left, and elements greater than or equal to the pivot on the right.

Recursion: Recursively apply Quick Sort to the left and right subarrays.
Combine: Merge the sorted subarrays with the pivot in between.

Sayyed Siddique
Frontend Developer

Swipe

11 Quick Sort Example

Quick Sort

```
function quickSort(arr) {
  // Base case: if the array has 1 or no elements, it's already sorted
  if (arr.length <= 1) {
    return arr;
  }

  // Step 1: Choose the pivot (last element in this example)
  const pivot = arr[arr.length - 1];
  const left = [];  // Array for elements less than the pivot
  const right = []; // Array for elements greater than or equal to the pivot

  // Step 2: Partitioning elements
  for (let i = 0; i < arr.length - 1; i++) {
    if (arr[i] < pivot) {
      left.push(arr[i]); // Elements smaller than the pivot go into 'left'
    } else {
      right.push(arr[i]); // Elements greater or equal go into 'right'
    }
  }

  // Step 3: Recursively sort left and right arrays and combine with the pivot
  return [...quickSort(left), pivot, ...quickSort(right)];
}

// Example usage
const arr = [8, 3, 1, 7, 0, 10, 2];
console.log(quickSort(arr)); // Output: [0, 1, 2, 3, 7, 8, 10]

```

Sayyed Siddique
Frontend Developer

Swipe

12 Quick Sort Explaination

Explanation of the Code:

1. **Base Case:** If the array has 1 or no elements (arr.length <= 1), it is already sorted, and we return the array.
2. **Pivot:** The pivot is chosen (in this case, the last element of the array, arr[arr.length - 1]).
3. **Partitioning:** The array is split into two parts:
   - Left subarray: Contains elements smaller than the pivot.
   - Right subarray: Contains elements greater than or equal to the pivot.
4. **Recursive Step:** The function is recursively called on the left and right subarrays. The result is concatenated with the pivot in between to form a fully sorted array.

Time Complexity:

- **Best/Average Case:** O(n log n), where the array is divided evenly during partitioning.
- **Worst Case:** $O(n^2)$, when the pivot consistently partitions the array in a highly unbalanced manner (e.g., when the array is already sorted, and the pivot is the smallest or largest element).

Sayyed Siddique
Frontend Developer

Swipe

Sayyed Siddique
Frontend Developer

Thanks for reading

Engage with Us!

Share your thoughts on Sorting Algorithms