Sorting Algorithms

Charles Aunkan Gomes
Lecturer, Dept. of CSE
United International University
charles@cse.uiu.ac.bd

Sorting

A **sorting algorithm** is used to arrange the elements of an array or list in a specific order:

- **Ascending Order**: From **minimum to maximum** (increasing order).
- **Descending Order**: From **maximum to minimum** (decreasing order).

Input Array: {2, 1, 5, 3, 4}

- Ascending Order: {1, 2, 3, 4, 5}
- Descending Order: {5, 4, 3, 2, 1}

Bubble Sort

Bubble sort is a sorting algorithm that compares **two adjacent elements** and **swaps** them until they are in the intended order.

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

Bubble Sort

Algorithm:

1. Start with the first element of the array.
2. Compare the current element with the next element.
   - If the current element is greater than the next element, swap them.
3. Move to the next element and repeat Step 2 until the end of the array.
4. Repeat Steps 1-3 for all elements except the last one each time (as the largest element "bubbles" to the end after each pass).
5. Stop when no swaps are needed, as the array is sorted.

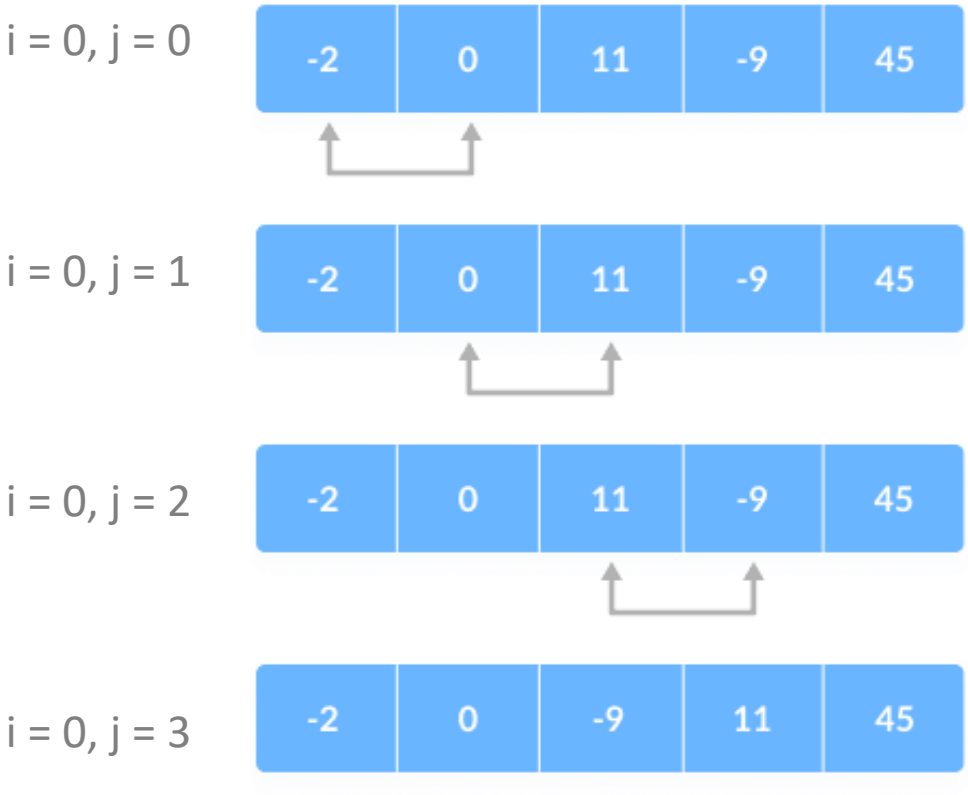
Bubble Sort

Pseudocode:

```
BubbleSort(array, n)
  For i = 0 to n-1
    For j = 0 to n-i-1
      If array[j] > array[j+1]
        Swap array[j] and array[j+1]
End
```

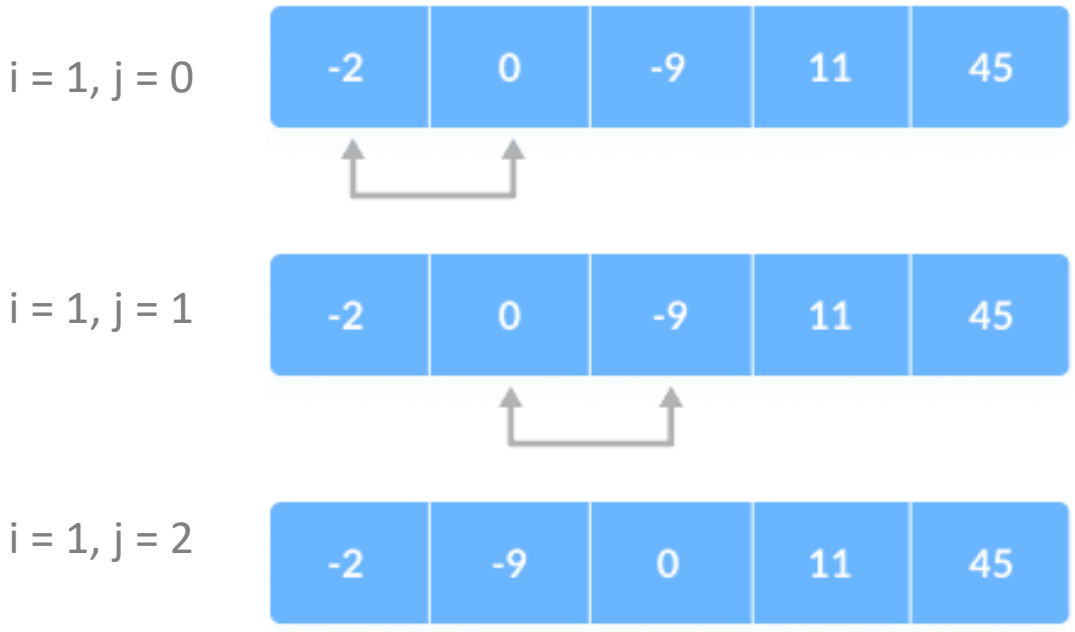
Bubble Sort

1st Iteration



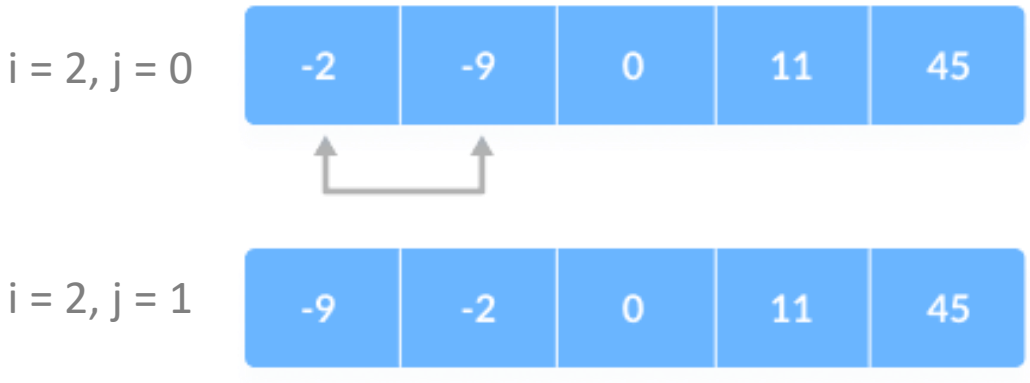
Bubble Sort

2nd Iteration



Bubble Sort

3rd Iteration



Bubble Sort

Best Case: The best case occurs when the array is already sorted. So, the number of comparisons required is N-1 and the number of swaps required = 0. Hence the best-case complexity is **O(N)**.

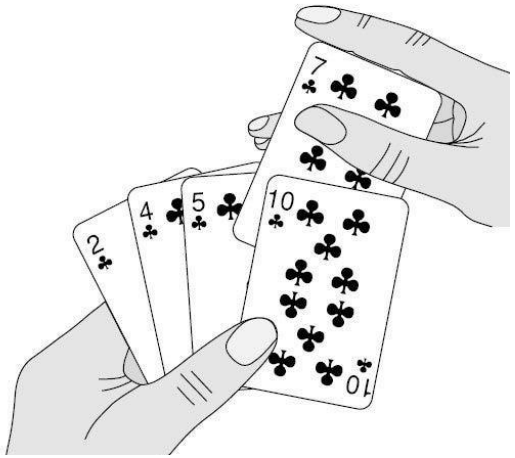
Worst Case: The worst-case condition for bubble sort occurs when elements of the array are arranged in reverse order. In the worst case, the total number of iterations or passes required to sort a given array is (N-1) and each iteration (N-i-1) comparisons are required. Hence the worst-case complexity is $O(N^2)$.

Average Case: The average case complexity is $O(N^2)$.

Insertion Sort

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we **sort cards in our hand in a card game**. We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place. A similar approach is used by insertion sort.



Insertion Sort

Algorithm:

1. Start from the second element (index 1) because a single element (index 0) is trivially sorted.
2. Compare the current element with the elements in the sorted part of the array (left side).
3. Shift all the larger elements in the sorted part one position to the right to create space.
4. Insert the current element into its correct position.
5. Repeat the process for all elements until the array is sorted.

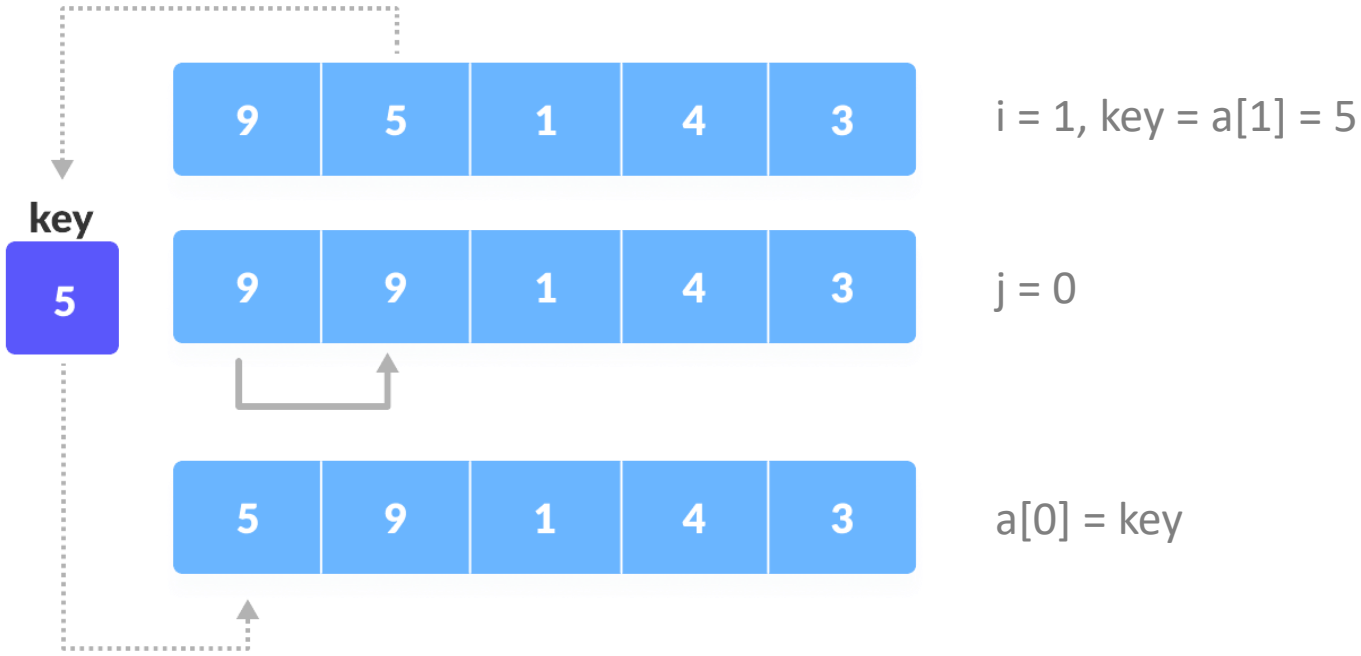
Insertion Sort

Pseudocode:

```
InsertionSort(array, n)
    For i = 1 to n-1:
        key = array[i]
        j = i – 1
        While j >= 0 and array[j] > key:
            array[j + 1] = array[j]
            j = j – 1
        array[j + 1] = key
End
```

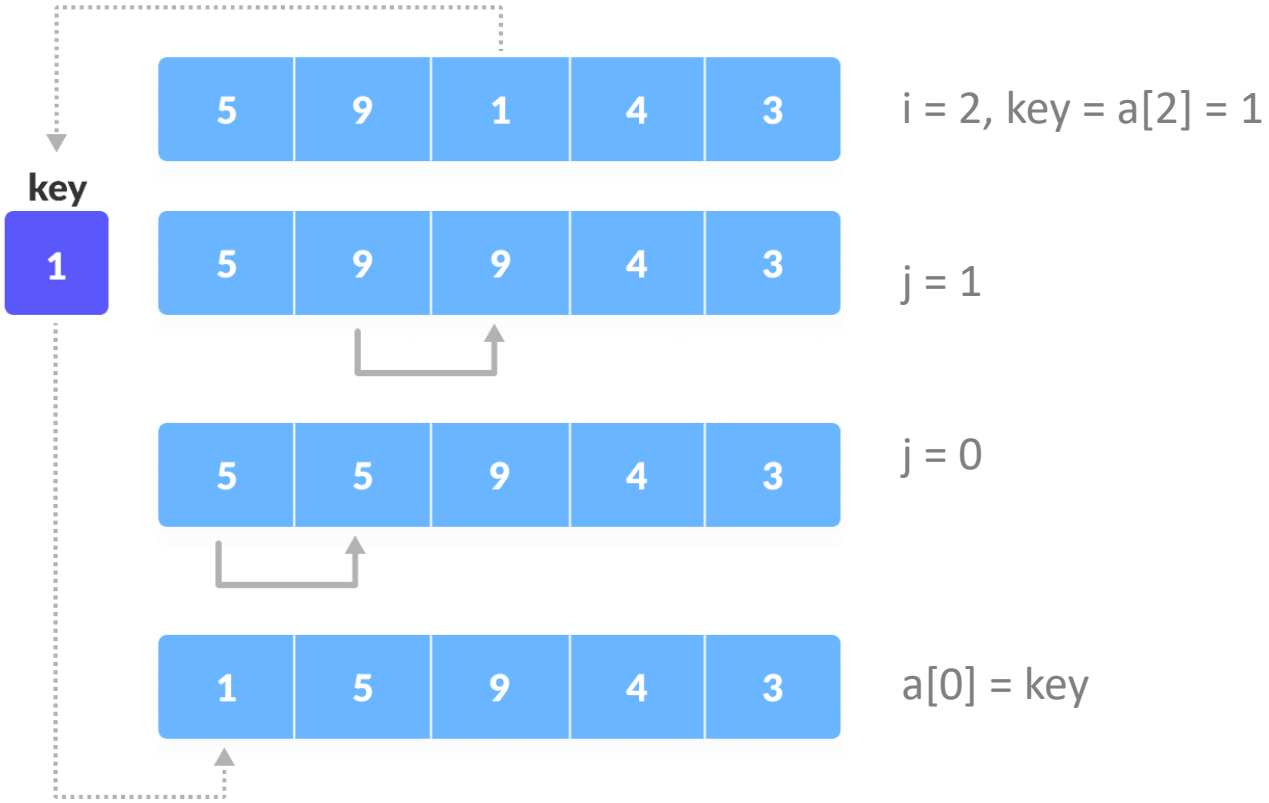
Insertion Sort

1st Iteration



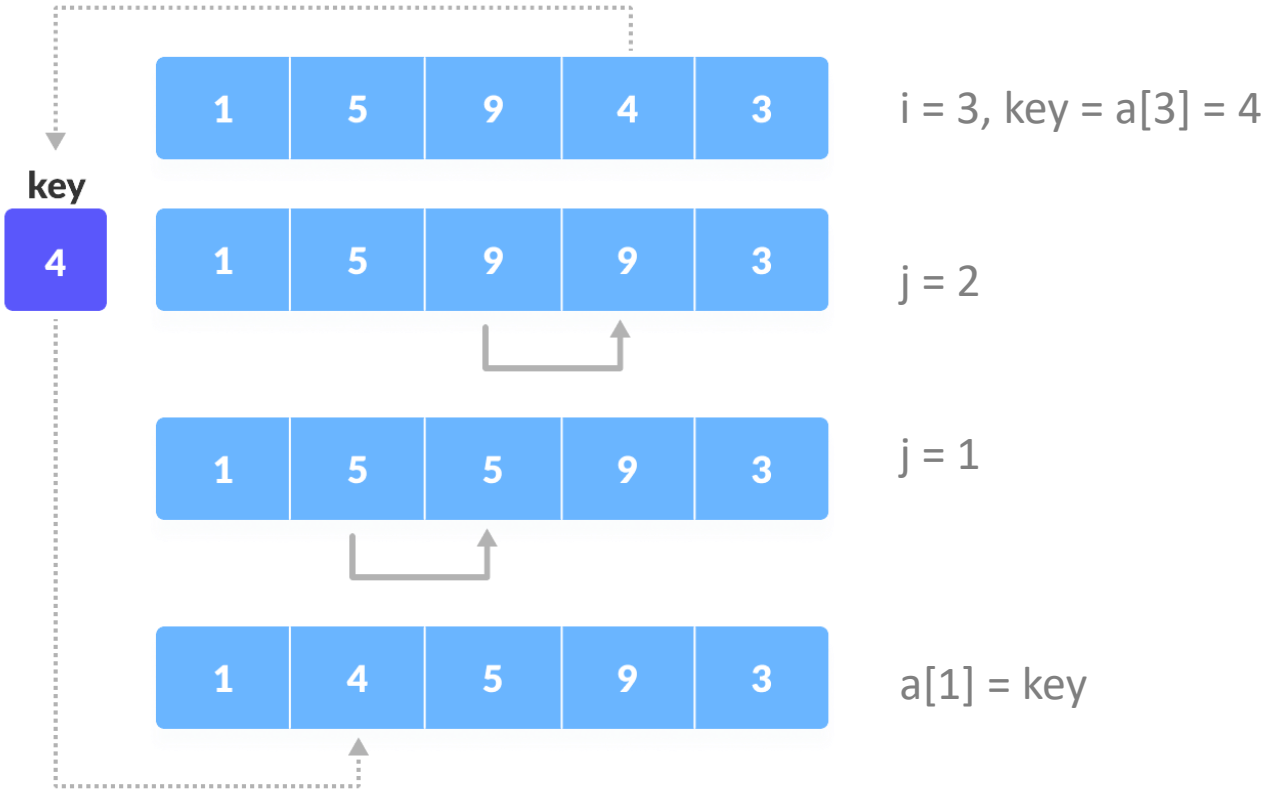
Insertion Sort

2nd Iteration

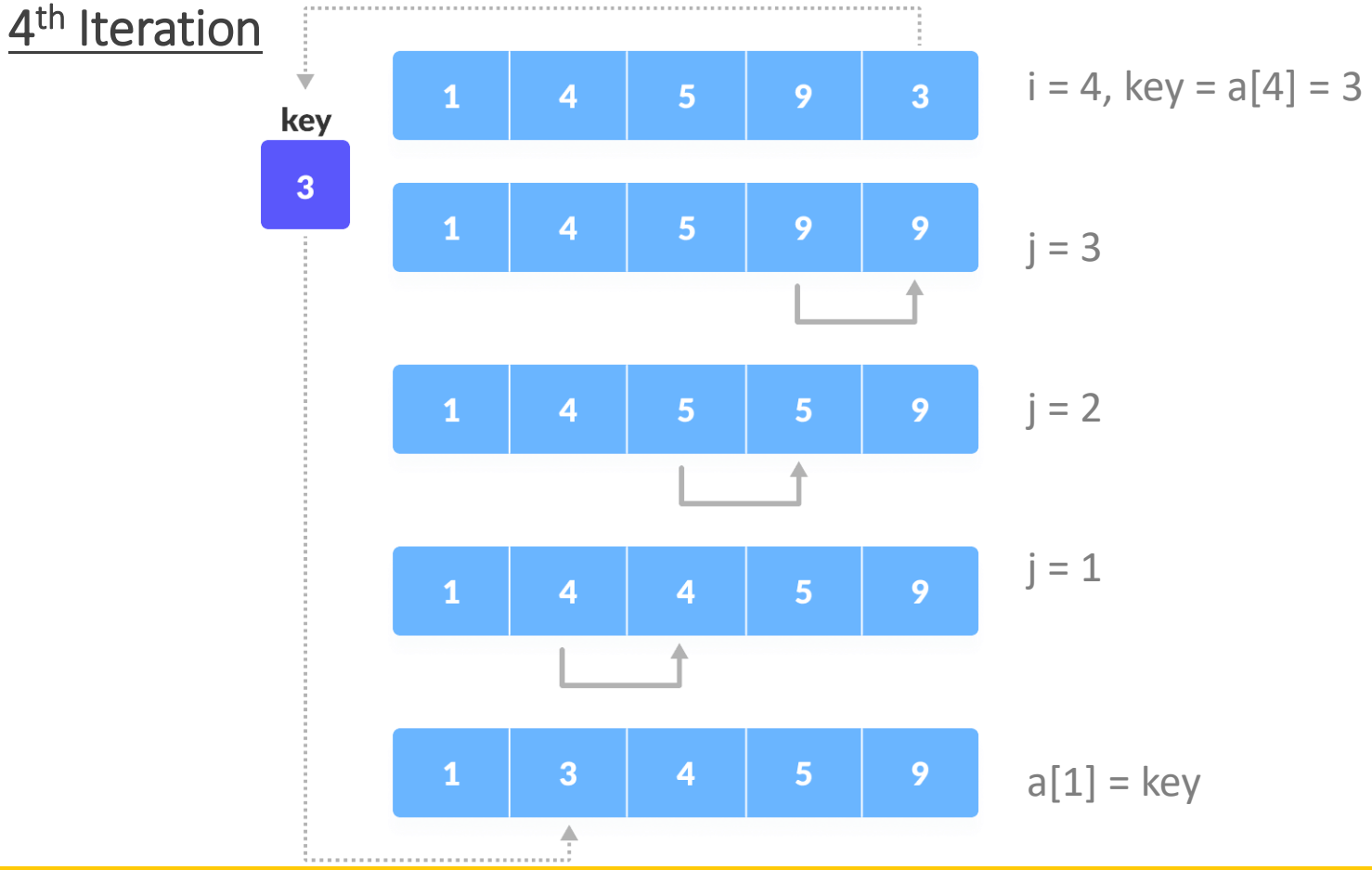


Insertion Sort

3rd Iteration



Insertion Sort



Insertion Sort

Best case: If the list is already sorted, where n is the number of elements in the list. Hence best-case time complexity is **O(n)**.

Worst case: If the list is in reverse order. Then worst-case time complexity is **$O(n^2)$.**

Average case: The average case complexity is **$O(n^2)$**.

Selection Sort

Selection sort is a sorting algorithm that selects the **smallest element from an unsorted list** in each iteration and places that element at the **beginning of the unsorted list**.

In a team selection scenario, you might want to line up players in ascending order of their heights. To achieve this, you scan the group to find the shortest player and move them to the front of the line. This process is repeated for the remaining players until everyone is sorted by height. A similar approach is used by selection sort.

Selection Sort

Algorithm:

1. Start with the first element of the array.
2. Assume the first element is the smallest. Set it as the minimum.
3. Compare this element with the rest of the array:
   - If any element is smaller than the current minimum, update the minimum to that element.
4. After finding the minimum, swap it with the first element of the unsorted part.
5. Move to the next element and repeat steps 2–4 for the remaining unsorted part of the array.
6. Continue until the entire array is sorted.

Selection Sort

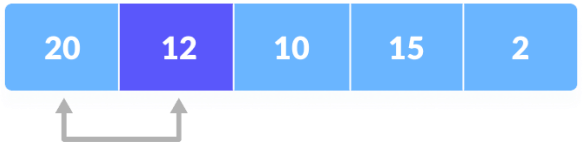
Pseudocode:

```
SelectionSort(array, n)
    For i = 0 to n-2:
        minIndex = i
        For j = i+1 to n-1:
            If array[j] < array[minIndex]:
            minIndex = j
        Swap array[i] and array[minIndex]
End
```

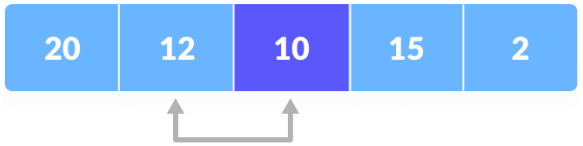
Selection Sort

1st Iteration

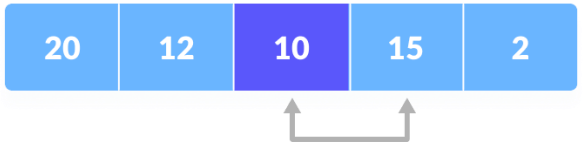
i = 0, j = 1, minIndex = 1



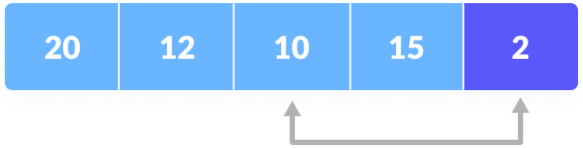
i = 0, j = 2, minIndex = 2



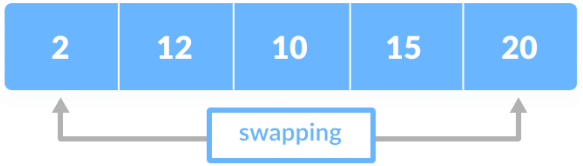
i = 0, j = 3, minIndex = 2



i = 0, j = 4, minIndex = 4



swap(a[0], a[4])

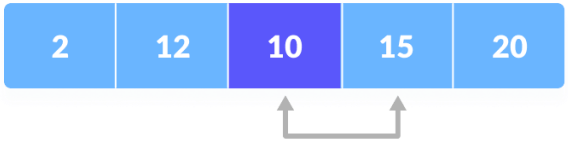


Selection Sort

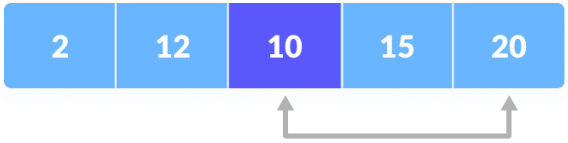
2nd Iteration

i = 1, j = 2, minIndex = 2

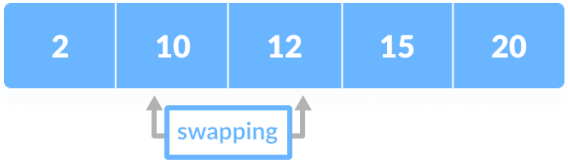
i = 1, j = 3, minIndex = 2



i = 1, j = 4, minIndex = 2



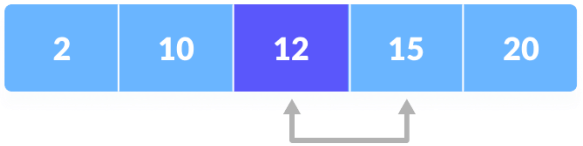
swap(a[1], a[2])



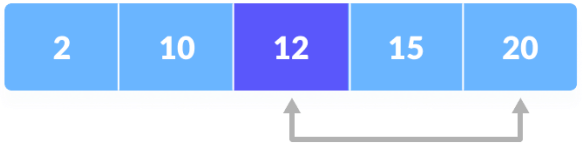
Selection Sort

3rd Iteration

i = 2, j = 3, minIndex = 2



i = 2, j = 4, minIndex = 2

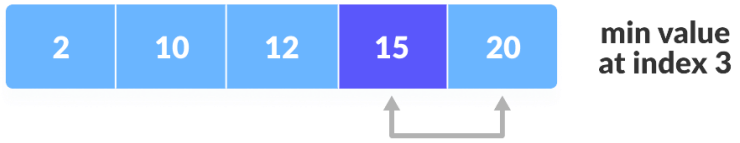


swap(a[2], a[2])

Selection Sort

4th Iteration

i = 3, j = 4, minIndex = 3



swap(a[3], a[3])

Selection Sort

Best Case: Even if the array is already sorted, Selection Sort still scans the entire unsorted portion of the array to find the smallest element in each iteration. There are no early exits since every comparison is performed. Hence best-case time complexity is $O(n^2)$.

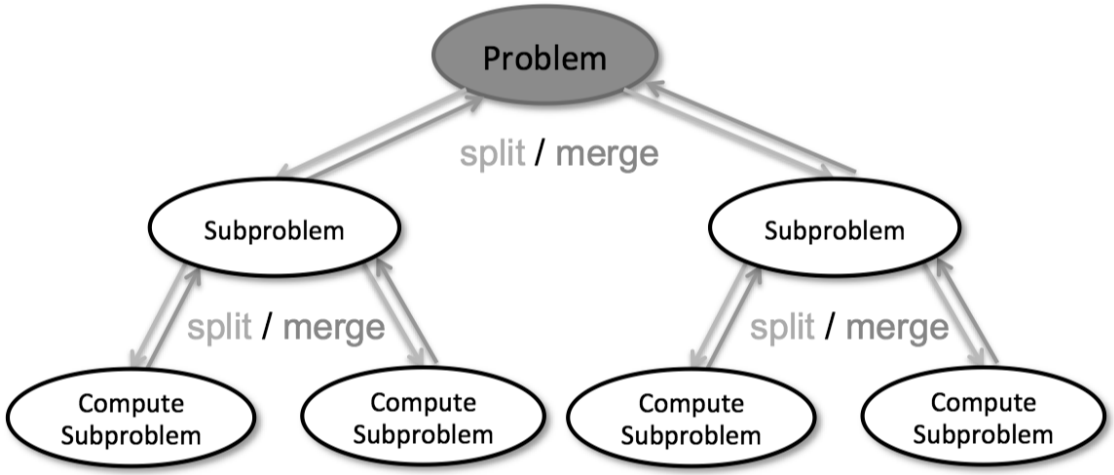
Worst Case: In the worst-case scenario (e.g., reverse-sorted array), Selection Sort makes the same number of comparisons and swaps as in the best case. Hence worst-case time complexity is $O(n^2)$.

Average Case: The average-case time complexity is $O(n^2)$.

Merge Sort

Merge Sort is one of the most popular sorting algorithms that is based on the principle of **Divide and Conquer** Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.



Merge Sort

Algorithm:

1. **Base Case**: If the array has one or no elements, it is already sorted.
2. **Divide**:
   - Find the middle index **m =⌊ (low + high) / 2 ⌋**.
   - Split the array into two subarrays:
     - Left half: **A[low…m]**
     - Right half: **A[m+1…high]**
3. **Conquer**: Apply Merge Sort to the left and right subarrays.
4. **Merge:** Combine the two sorted subarrays into a single sorted array.

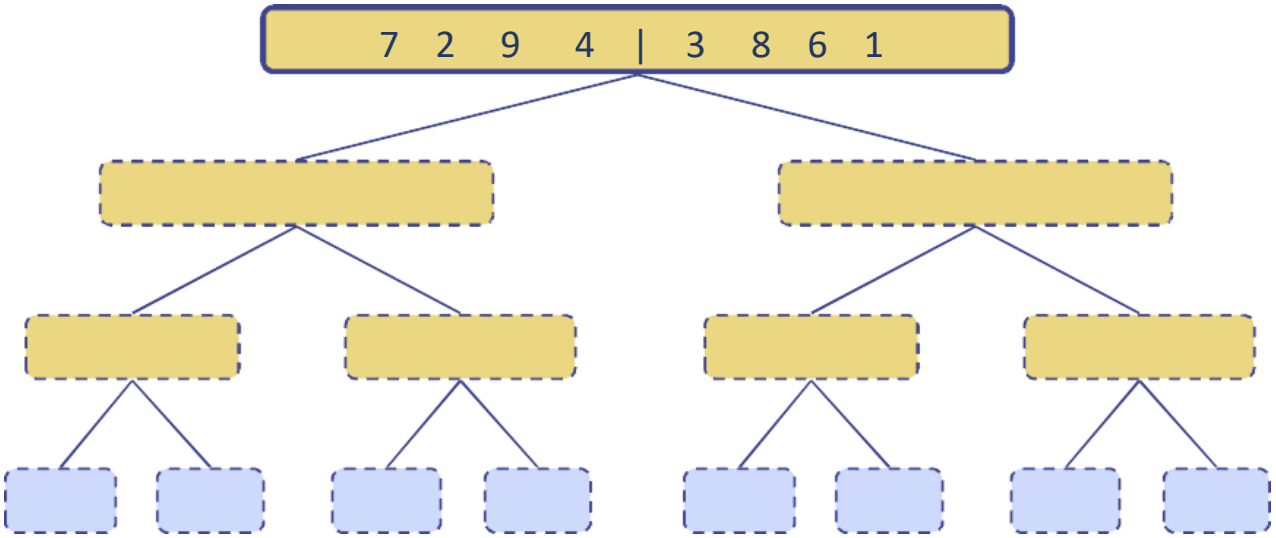
Merge Sort

Pseudocode:

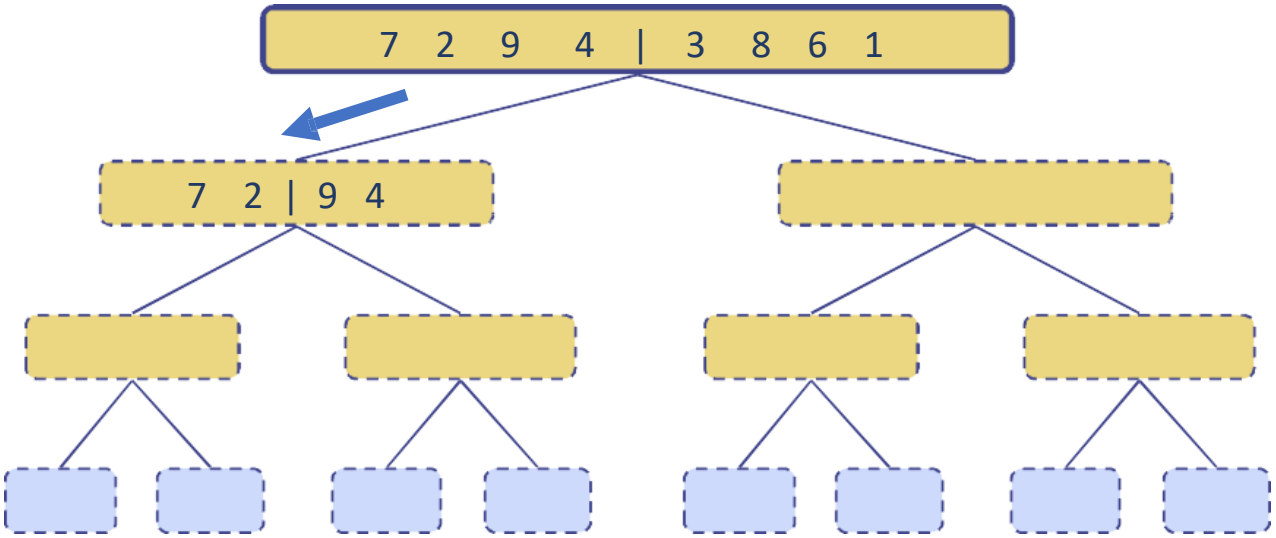
```
MergeSort(array, low, high)
    If low < high:
        mid = (low + high) / 2
        MergeSort(array, low, mid)
        MergeSort(array, mid + 1, high)
        Merge(array, low, mid, high)
```

```
Merge(array, low, mid, high)
    Create left = array[low...mid],
    right =array[mid+1...high]
    Initialize i = 0, j = 0, k = low
    While i < len(left) and j < len(right):
        If left[i] <= right[j]:
            array[k] = left[i], i = i + 1
        Else:
            array[k] = right[j], j = j + 1
        k = k + 1
    Copy remaining elements of left and right
    into array
```

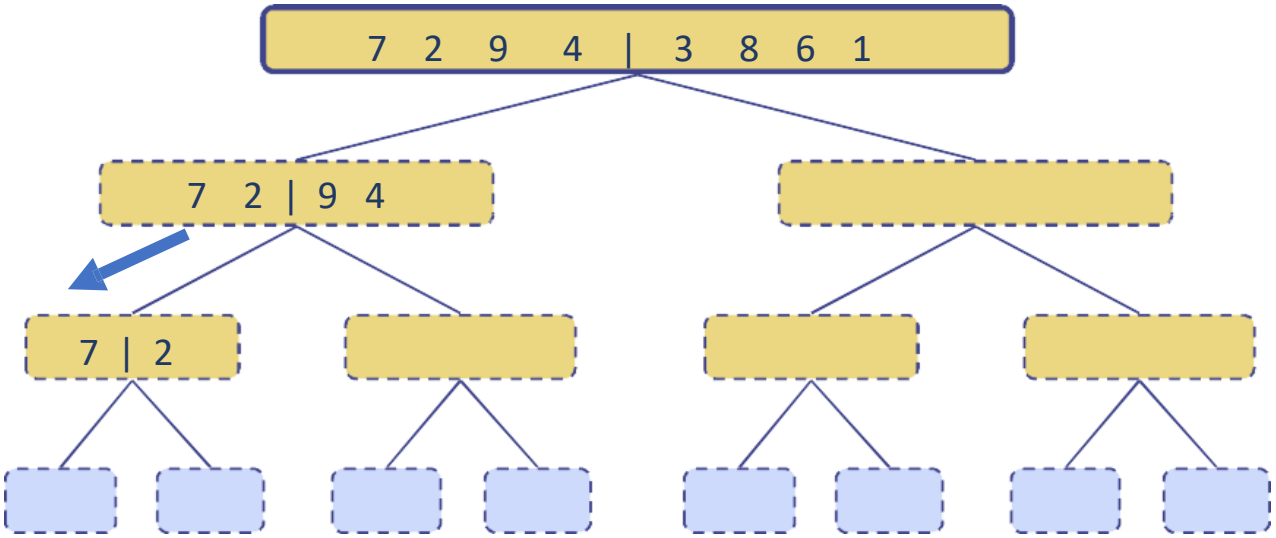
Merge Sort



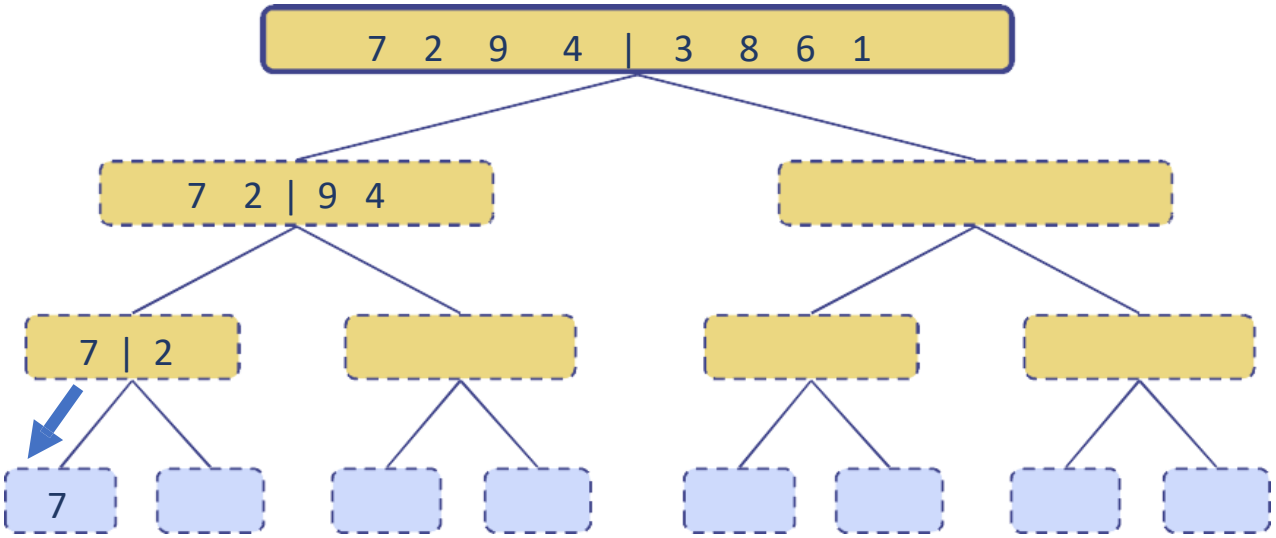
Merge Sort



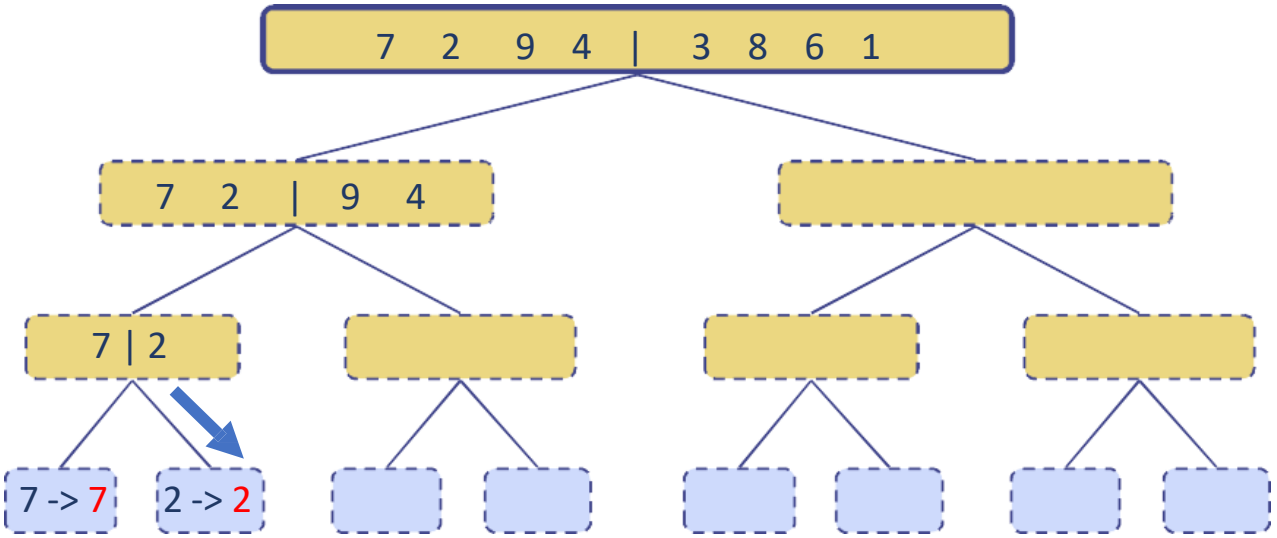
Merge Sort



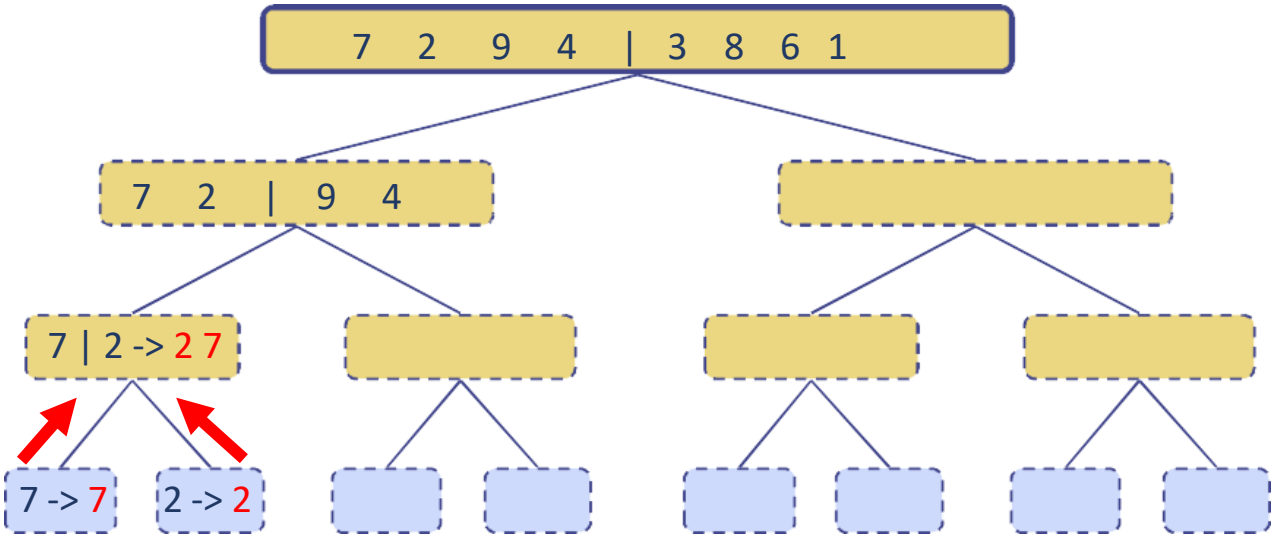
Merge Sort



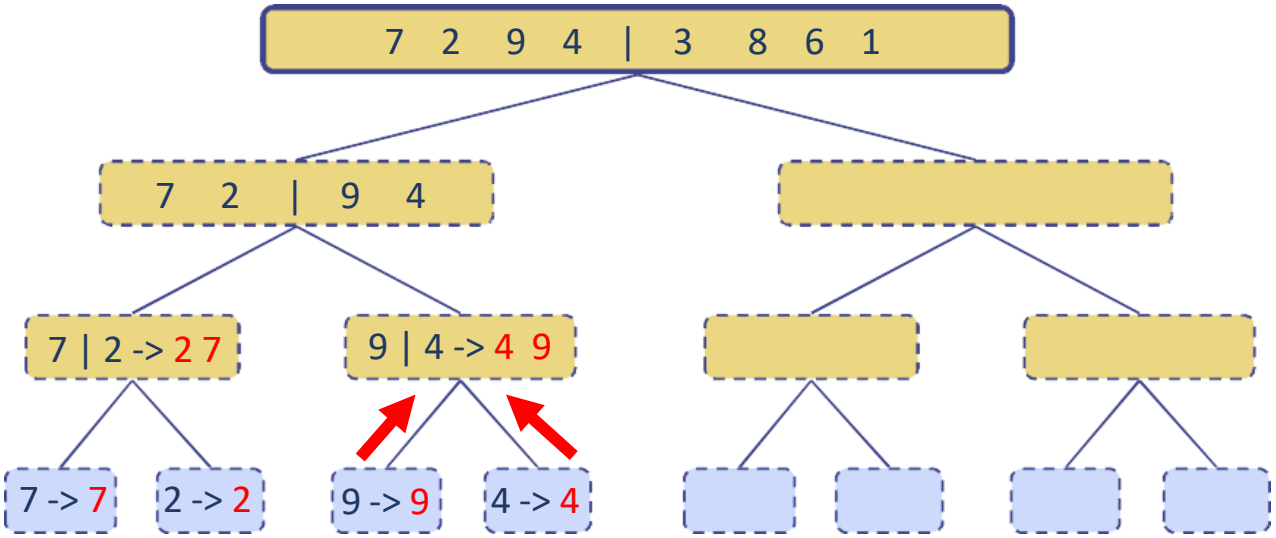
Merge Sort



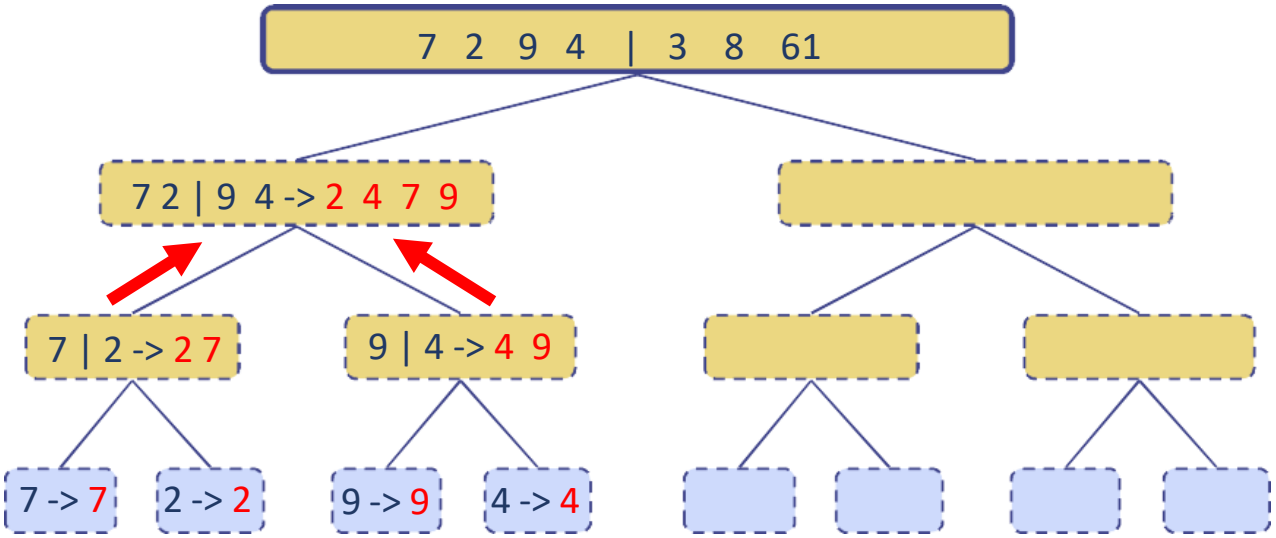
Merge Sort



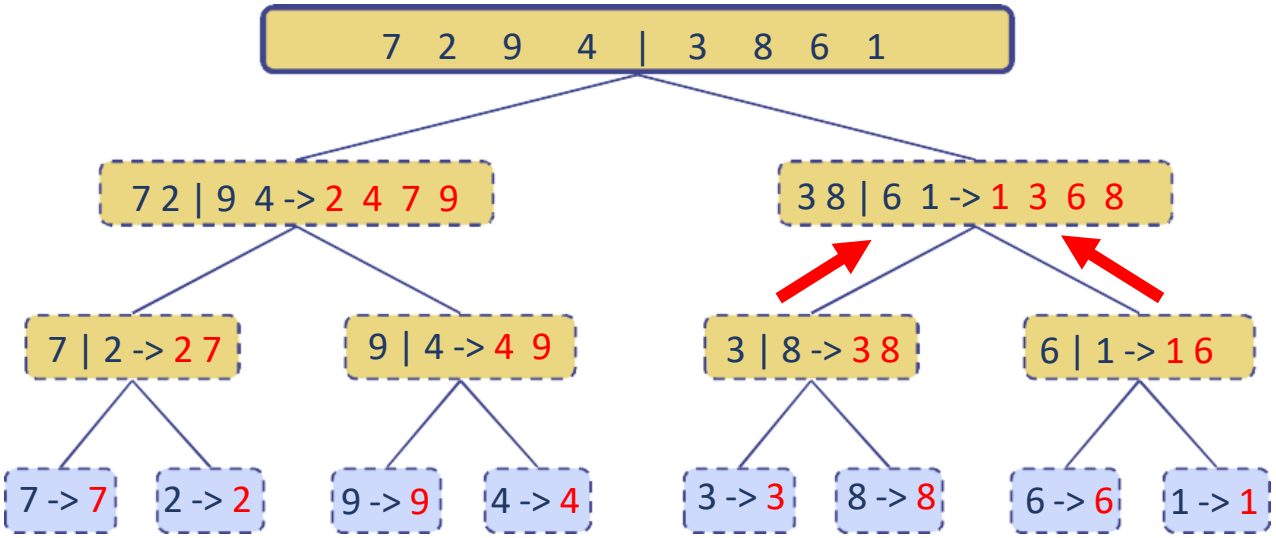
Merge Sort



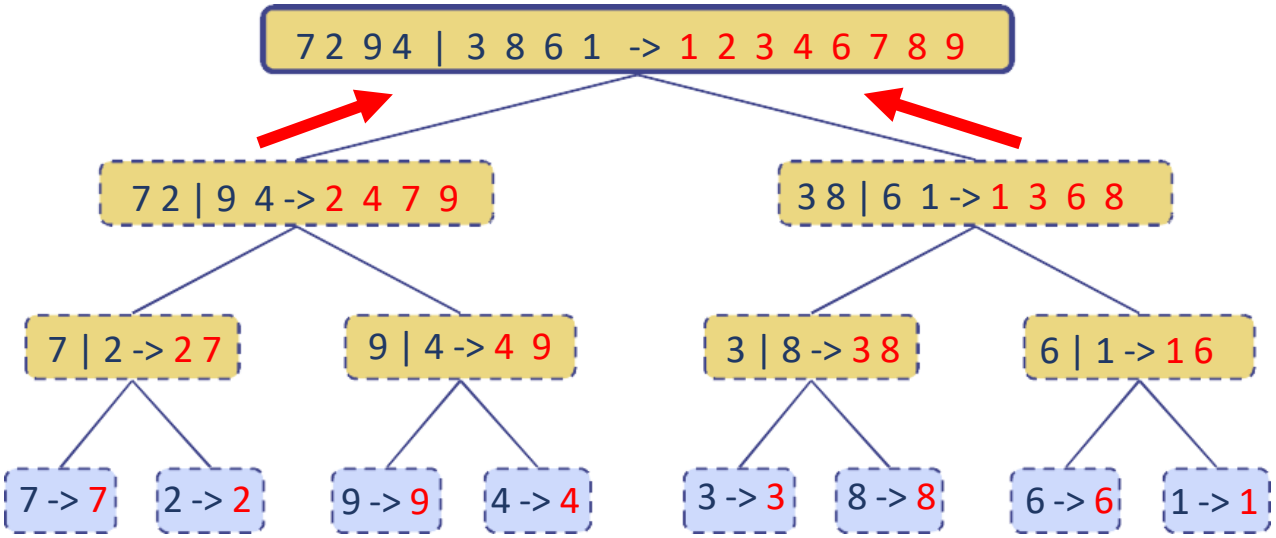
Merge Sort



Merge Sort


7 2 9 4 | 3 8 6 1
7 2 | 9 4 -> 2 4 7 9
3 8 | 6 1 -> 1 3 6 8
7 | 2 -> 2 7
9 | 4 -> 4 9
3 | 8 -> 3 8
6 | 1 -> 1 6
7 -> 7
2 -> 2
9 -> 9
4 -> 4
3 -> 3
8 -> 8
6 -> 6
1 -> 1

Merge Sort


7 2 9 4 | 3 8 6 1 -> 1 2 3 4 6 7 8 9
7 2 | 9 4 -> 2 4 7 9
3 8 | 6 1 -> 1 3 6 8
7 | 2 -> 2 7
9 | 4 -> 4 9
3 | 8 -> 3 8
6 | 1 -> 1 6
7 -> 7
2 -> 2
9 -> 9
4 -> 4
3 -> 3
8 -> 8
6 -> 6
1 -> 1

Merge Sort

Best Case: O(n log n)

Worst Case: O(n log n)

Average Case: O(n log n)

Explanation: The array is divided logn times, and merging takes O(n) time.

Quick Sort

Quick Sort is a **divide-and-conquer** algorithm that works as follows:

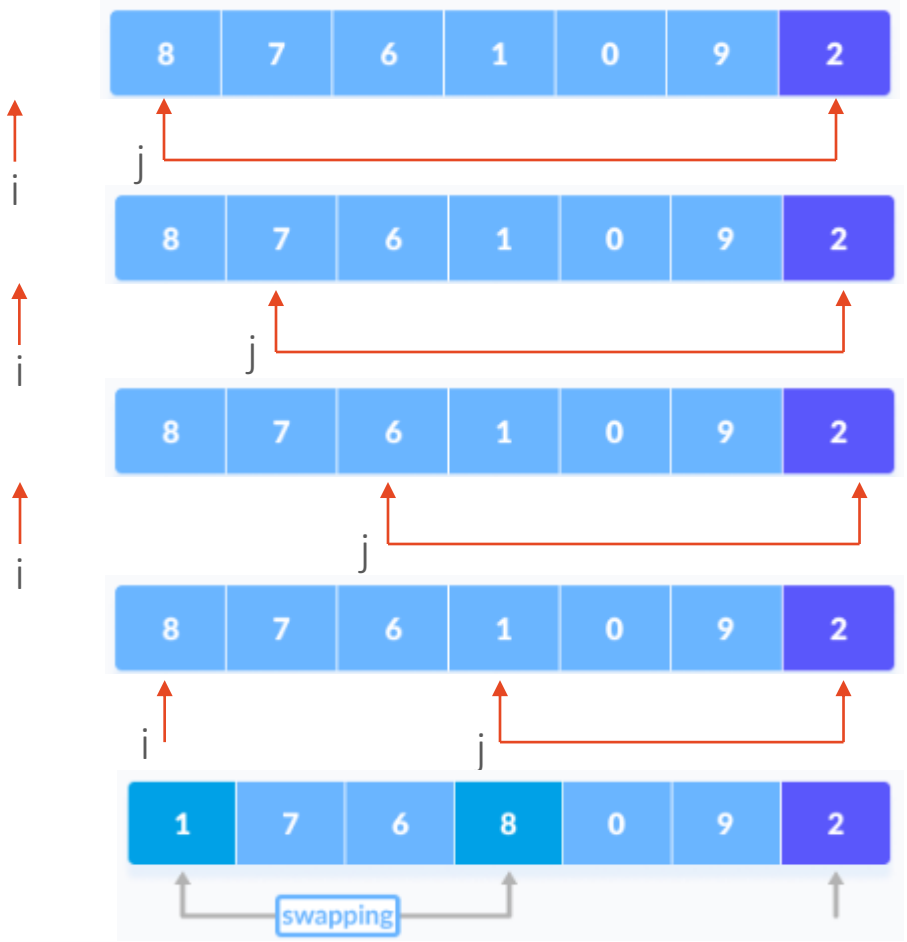
1. **Choose a pivot element** (any element from the array; often the last, first, or middle element).
2. **Partition the array**: Rearrange elements such that all elements smaller than the pivot come before it, and all elements larger come after.
3. Recursively apply Quick Sort to the subarrays on either side of the pivot.

Quick Sort

```
QUICKSORT(arr, low, high)
    IF low < high
        pivotIndex ← PARTITION(arr, low, high)
        QUICKSORT(arr, low, pivotIndex - 1)
        QUICKSORT(arr, pivotIndex + 1, high)
    END IF
END QUICKSORT
```

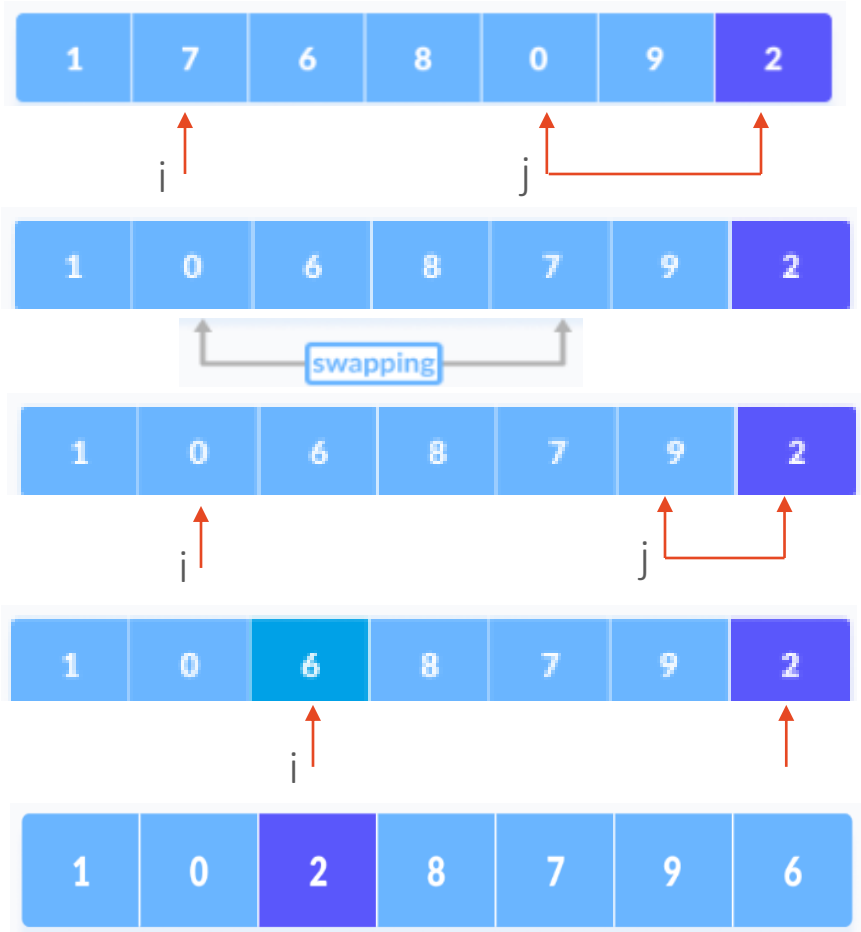
```
PARTITION(arr, low, high)
        pivot ← arr[high]
        i ← low - 1
        FOR j ← low TO high - 1
            IF arr[j] ≤ pivot
                i ← i + 1
                SWAP(arr[i], arr[j])
            END IF
        END FOR
        SWAP(arr[i + 1], arr[high])
        RETURN i + 1
END PARTITION
```

Quick Sort



pivot = 2

Quick Sort



pivot = 2

Quick Sort

Sorting the elements on the left of pivot using recursion

Quick Sort

Sorting the elements on the right of pivot using recursion

Quick Sort

Best Case: In the best-case scenario, the pivot divides the array into two roughly equal halves at each step. This leads to a balanced recursion tree. Hence best-case time complexity is **O(nlogn)**.

Worst Case: The worst-case occurs when the pivot chosen is the smallest or largest element, causing the array to be split into sub-arrays with a size difference of 1. This happens if the array is already sorted or nearly sorted. Hence worst-case time complexity is $\mathbf{O(n^2)}$.

Average Case: The average case occurs when the pivot is reasonably balanced on average, leading to a balanced recursion tree. Hence average-case time complexity is **O(nlogn)**.

THANK YOU