Name: Achyut Shukla PRN: 20070122005 Batch: CS – AA1

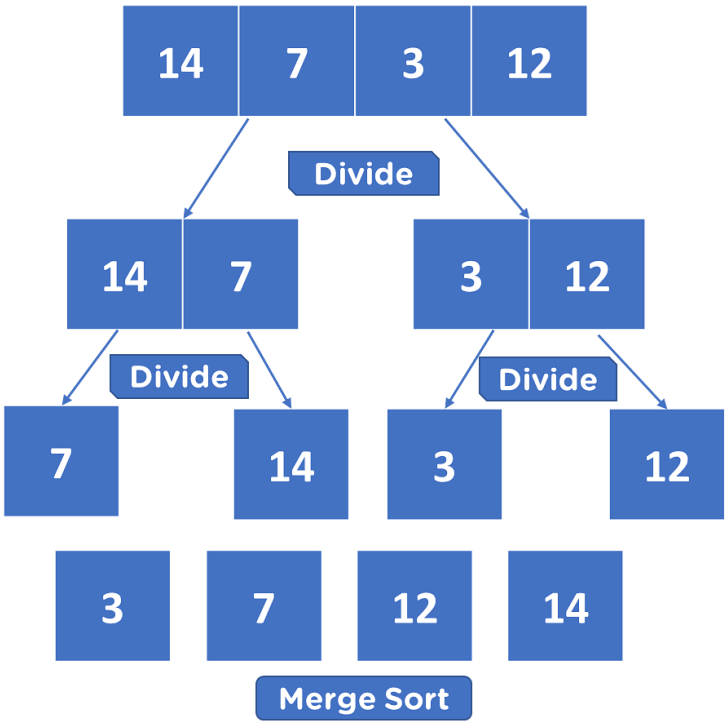
Title: Merge Sort

Introduction: Merge sort is a popular and efficient sorting algorithm that uses the divide-and-conquer approach to sort an array or a list of elements. The basic idea behind this algorithm is to divide the array into two halves, sort each half individually, and then merge the two sorted halves back together into a single sorted array. This process is repeated until the array is completely sorted.

Mechanism: The mechanism of the merge sort algorithm is as follows:

I. Divide the array into two halves: The array is repeatedly divided into two halves until each sub-array contains only one element.
II. Sort each half recursively: Each sub-array is then sorted using the same merge sort algorithm.
III. Merge the two sorted halves: The two sorted sub-arrays are merged back together by comparing the first element of each sub-array and arranging them in the correct order. This process continues until all the elements from both sub-arrays have been merged into a single sorted array.
IV. Repeat the process: The process is repeated until the entire array is sorted.

For example: Taking an array [14, 7, 3, 12].



Pseudo Code:

```
function mergeSort(array)
  if array length is less than or equal to 1
    return array
  end if

  middle = array length / 2
  leftHalf = array[0 to middle - 1]
  rightHalf = array[middle to array length - 1]

  leftHalf = mergeSort(leftHalf)
  rightHalf = mergeSort(rightHalf)

  return merge(leftHalf, rightHalf)
end function

function merge(leftHalf, rightHalf)
  result = []
  leftIndex = 0
  rightIndex = 0
  while leftIndex < leftHalf length and rightIndex < rightHalf length
    if leftHalf[leftIndex] < rightHalf[rightIndex]
      result.append(leftHalf[leftIndex])
      leftIndex++
    else
      result.append(rightHalf[rightIndex])
      rightIndex++
    end if
  end while
```

```
 while leftIndex < leftHalf length
  result.append(leftHalf[leftIndex])
  leftIndex++
 end while

 while rightIndex < rightHalf length
  result.append(rightHalf[rightIndex])
  rightIndex++
 end while

 return result
end function
```

Code:

1. Iterative:

```
import java.util.Arrays;
class Codechef {
   public static void sort(int[] arr) {
      int n = arr.length;
      int currSize;
      int leftStart;
      for (currSize = 1; currSize <= n-1; currSize = 2*currSize) {
         for (leftStart = 0; leftStart < n-1; leftStart += 2*currSize) {
            int mid = Math.min(leftStart + currSize - 1, n-1);
            int rightEnd = Math.min(leftStart + 2*currSize - 1, n-1);
            merge(arr, leftStart, mid, rightEnd);
         }
```

```
    }
}

private static void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int[] L = new int[n1];
    int[] R = new int[n2];
    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[mid + 1 + j];
    int i = 0, j = 0;
    int k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
```

```
            while (j < n2) {
                arr[k] = R[j];
                j++;
                k++;
            }
        }

        public static void main(String[] args) {
            int[] arr = {14, 7, 3, 12};
            sort(arr);
            System.out.println("Sorted array is " + Arrays.toString(arr));
        }
    }
```

2. Recursive:

```
import java.util.Arrays;
class Codechef {
    public static void sort(int[] arr, int l, int r) {
        if (l < r) {
            int m = (l + r) / 2;
            sort(arr, l, m);
            sort(arr, m + 1, r);
            merge(arr, l, m, r);
        }
    }
```

```
private static void merge(int[] arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int[] L = new int[n1];
    int[] R = new int[n2];
    for (int i = 0; i < n1; i++) {
        L[i] = arr[l + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[m + 1 + j];
    }
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
```

```
            j++;
            k++;
        }
    }

    public static void main(String[] args) {
        int[] arr = {14, 7, 3, 12};
        sort(arr, 0, arr.length - 1);
        System.out.println("Sorted array is " + Arrays.toString(arr));
    }
}
```

Time complexity Proof:

To prove the time complexity of the merge sort algorithm, we can use mathematical induction.

Base case:

For n = 1, the time complexity of the merge sort algorithm is O(1), which is a constant time. Hence, the base case holds true.

Inductive step:

Assume that the time complexity of the merge sort algorithm for n = k is $O(k * \log_2 k)$.

For n = k + 1:

The merge sort algorithm splits the input array of size k + 1 into two halves, and then performs the merge operation. The size of the two halves is k/2 and (k + 1) / 2, respectively. The time complexity of the merge operation is O(k) because it has to iterate through all the elements of

the two halves. The time complexity of the divide operation is $O(\log_2 (k + 1))$. Hence, the total time complexity for $n = k + 1$ is $O((k / 2 + (k + 1) / 2) * \log_2 (k + 1)) = O(k * \log_2 (k + 1))$.

Since the time complexity for $n = k + 1$ is $O(k * \log_2 (k + 1))$, and the time complexity for $n = k$ is $O(k * \log_2 (k + 1))$, we can conclude that the time complexity of the merge sort algorithm is $O(n * \log_2 (n))$.

Space complexity analysis:

The space complexity of the merge sort algorithm is O(n) because it requires an additional array of size n to store the temporary result during the merge operation. This additional array is used to store the elements of the two sub-arrays being merged.

In the case of the iterative implementation of the merge sort algorithm, the space complexity is still O(n) because it requires a temporary array of size n to store the result of the merge operation.

In the case of the recursive implementation of the merge sort algorithm, the space complexity is O(log(n)) because it requires a limited amount of additional memory for the function call stack. Each recursive call uses a constant amount of memory for the function call, and the maximum number of function calls is $\log_2 (n)$. Hence, the space complexity of the recursive implementation of the merge sort algorithm is O(log(n)).

Advantages:

- **Stability:** Merge sort is a stable sorting algorithm, meaning that it maintains the relative order of equal elements. This property is important for applications where the relative order of equal elements needs to be preserved.
- **Performance:** Merge sort has a guaranteed time complexity of O(n * log(n)), making it one of the fastest sorting algorithms for large datasets. This makes it a good choice for applications where large datasets need to be sorted efficiently.
- **Simplicity:** The merge sort algorithm is relatively simple to implement and understand, especially when compared to other sorting algorithms like quicksort or heapsort. This makes it a good choice for educational purposes or for less experienced programmers.

Disadvantage:

- **Space complexity:** The merge sort algorithm requires additional memory to store the temporary result during the merge operation. This makes it a poor choice for applications where memory is a constraint, as the space complexity of the algorithm is O(n).
- **Time overhead:** The merge sort algorithm requires a significant amount of time overhead due to the need to divide the input array into smaller sub-arrays and then merge the sub-arrays back into a single sorted array. This overhead can make the algorithm slow for small datasets, where other sorting algorithms like insertion sort or selection sort may be faster.
- **Unsuitable for real-time systems:** The merge sort algorithm is not suitable for real-time systems, as it requires a significant amount of time to sort the data. Real-time systems require fast sorting algorithms that can sort the data in a small amount of time.

Real life applications:

- **Database management:** Sorting large databases efficiently to retrieve specific information or perform data analysis.
- **File management:** Sorting large numbers of files by size, name, or date modified.
- Music/audio processing: Sorting audio files by length, tempo, or genre.
- **Scientific simulations:** Sorting data generated by scientific simulations in a way that facilitates analysis and interpretation.
- **Medical records management:** Sorting medical records by patient name, date of birth, or medical condition to make it easier to find specific records.
- **Logistics and supply chain management:** Sorting shipments and deliveries by delivery date, destination, or priority to ensure efficient and timely delivery.
- **Financial record management:** Sorting financial records by date, amount, or type to make it easier to find specific transactions or perform analysis.
- **Sorting data for machine learning algorithms:** Sorting data in a way that is suitable for use as input for machine learning algorithms, such as decision trees or neural networks.

Optimizations and advancements:

- **Hybrid sort:** A hybrid sort combines the strengths of multiple sorting algorithms to achieve better performance than any single algorithm. For example, a hybrid sort algorithm might use quicksort to sort small arrays, and merge sort to sort large arrays. This optimization can improve the performance of the sorting algorithm for a wide range of inputs.
- **In-place merge sort:** An in-place merge sort is a variation of the merge sort algorithm that sorts the data in place, without using additional memory. This optimization can reduce the space complexity of the algorithm, making it a good choice for applications where memory is a constraint.

- **Parallel merge sort:** A parallel merge sort is a variation of the merge sort algorithm that sorts the data in parallel, using multiple processors or cores. This optimization can significantly improve the performance of the algorithm, especially for large datasets.
- **Cache-aware merge sort:** A cache-aware merge sort is a variation of the merge sort algorithm that is designed to take advantage of the memory hierarchy in modern computer systems. This optimization can significantly improve the performance of the algorithm, especially for large datasets, by reducing the number of cache misses.
- **External merge sort:** An external merge sort is a variation of the merge sort algorithm that sorts data that is too large to fit in memory. This optimization can be used to sort very large datasets by dividing the data into smaller chunks and sorting each chunk separately, then merging the sorted chunks.