Data Structures Lab 7

Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Muhammad Nouman Hanif

Note:

- Lab manual cover following below Advance sorting algorithms: **{Quick Sort, Merge Sort, Radix Sort}**
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Quick Sort

Quick Sort Algorithm is a **Divide & Conquer algorithm**. It divides input **arrays in two partitions**, calls itself for the two partitions (recursively) and performs in-place sorting while doing so. A separate partition () function is used for performing this in-place sorting at every iteration.

There are 2 Phases in the Quick Sort Algorithm.

Division Phase:

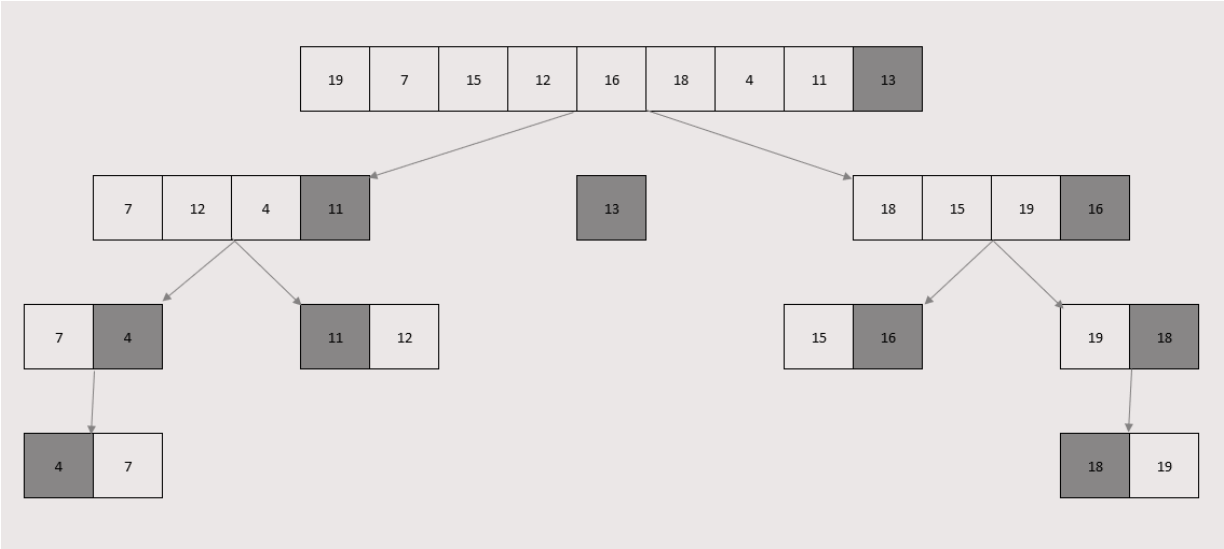
- Divide the array into 2 halves by finding the pivot point to perform the partition of the array.
- The in-place sorting happens in this partition process itself.

Recursion Phase:

- Call Quick Sort on the left partition
- Call Quick Sort on the right partition.

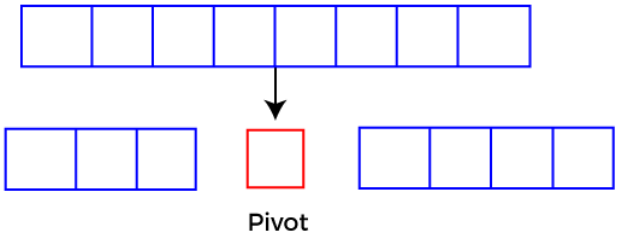
Quick Sort Algorithm:

- Make the right-most index value pivot
- Partition the array using pivot value
- Quicksort left partition recursively
- Quicksort right partition recursively



Pivot Selection:

- ➔ Choose the **any index value** as pivot
- ➔ Take two variables to point left and right of the list **excluding pivot**
- ➔ Left points to the **low index** & Right points to the **high index**
- ➔ While value at **left is less than pivot move right** & While value at **right is greater than pivot move left**
- ➔ If both left & right step does not match **swap** left and right
- ➔ If left = right, the point where they met is new pivot



Pseudocode:

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1
   while True do
      while A[++leftPointer] < pivot do
      //do-nothing
      end while
      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while
      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
      end if
   end while
   swap leftPointer,right
   return left pointer
end function
```

Time Complexity:

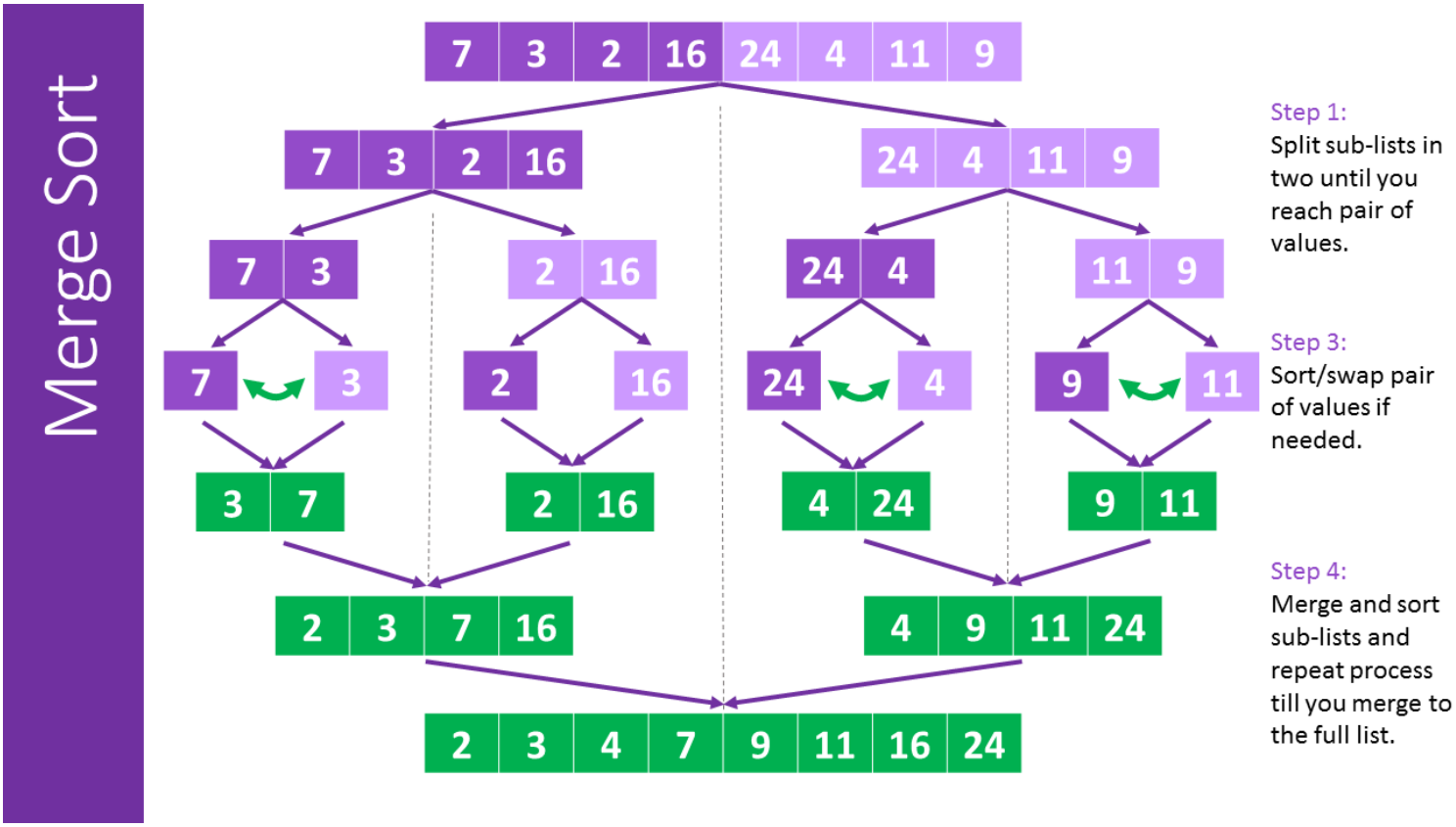
Worst Case - $O(N^2)$ | Average Case - O(N LOG N) | Best Case - O(N LOG N)

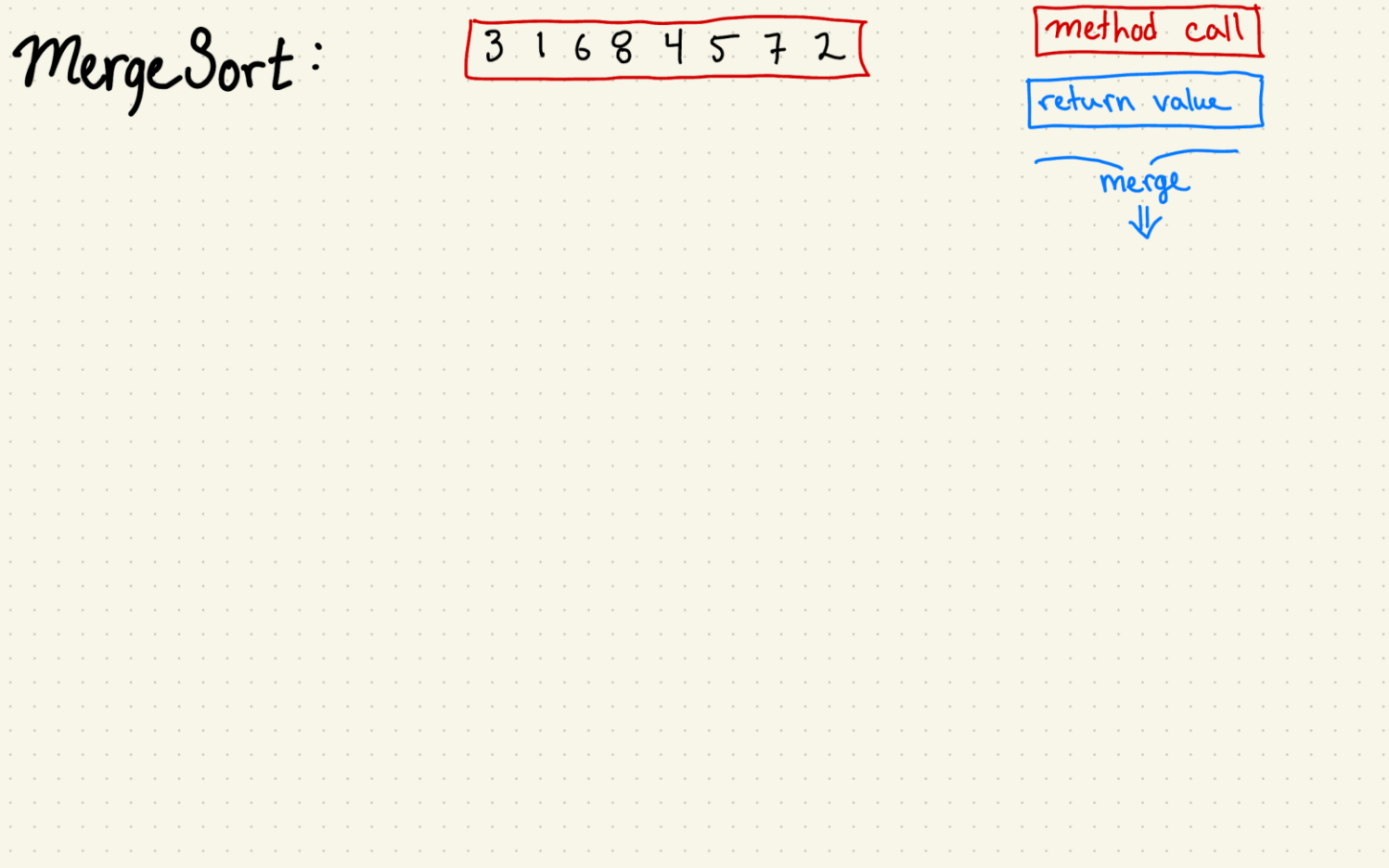
Merge Sort

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 word, it **continuously splits the array in half** until it cannot be further divided i.e., the array has **only one element left** (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.
Here's a step-by-step explanation of how merge sort works:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.





Merge Sort Algorithm:

- ➔ Divide the list recursively into two halves until it can no more be divided.
- ➔ Merge the smaller lists into new list in sorted order.

Pseudocode:

```
function mergeSort(arr, left, right)
    if left < right do
        mid = left + (right - left) / 2
        mergeSort(arr, left, mid)
        mergeSort(arr, mid + 1, right)
        merge(arr, left, mid, right)
    end if
end function

function merge(arr, left, mid, right)
    n1 = mid - left + 1
    n2 = right - mid
    leftArr[n1], rightArr[n2]

    for i = 0 to n1 - 1 do
        leftArr[i] = arr[left + i]
    end for
    for j = 0 to n2 - 1 do
        rightArr[j] = arr[mid + 1 + j]
    end for

    i = 0, j = 0, k = left

    while i < n1 and j < n2 do
        if leftArr[i] <= rightArr[j] then
            arr[k] = leftArr[i]
            i = i + 1
        else
            arr[k] = rightArr[j]
            j = j + 1
        end if
        k = k + 1
    end while

    while i < n1 do
        arr[k] = leftArr[i]
        i = i + 1
        k = k + 1
    end while

    while j < n2 do
        arr[k] = rightArr[j]
        j = j + 1
        k = k + 1
    end while
end function
```

Time Complexity:

Worst Case - O(N LOG N) | Average Case - O(N LOG N) | Best Case - O(N LOG N)

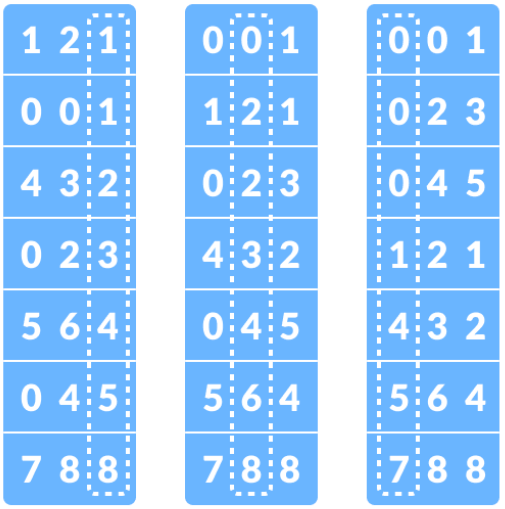
Radix Sort (Bucket Sort)

Radix Sort is a linear sorting algorithm that sorts elements by processing them **digit by digit**. It is an efficient sorting algorithm for **integers or strings with fixed-size keys**. It is a **non-comparative sorting algorithm** and avoids comparison by creating and distributing elements into **buckets according to their radix**. For elements with **more than one significant digit**, this bucketing process is **repeated for each digit**, while preserving the ordering of the prior step, until all digits have been considered. For this reason, radix sort has also been called **bucket sort and digital sort**.

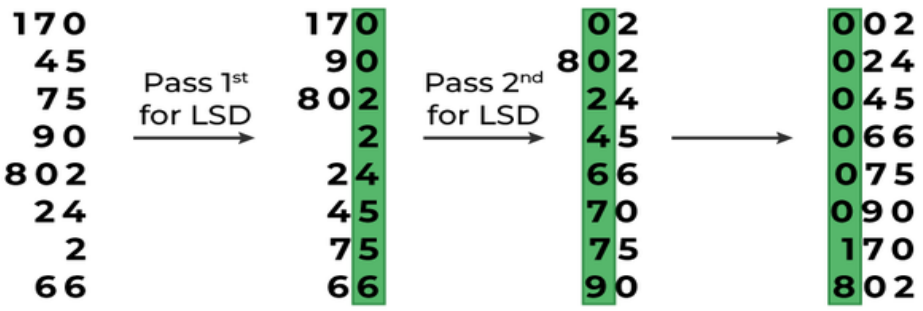
Here's a step-by-step explanation of how radix sort works:

1. Check whether all the input elements have the same number of digits. If not, check for numbers that have **maximum number of digits** in the list and add **leading zeroes to the ones** that do not.
2. Take the **least significant digit** of each element.
3. Sort these digits using **counting sort logic** and changing the order of elements based on the output achieved. For example, the possible values each digit can take would be 0-9, so index the digits based on these values.
4. Repeat Step 2 for the next least significant digits until all the digits in the elements are sorted.
5. The final list of elements achieved after kth loop is the sorted output.
6. https://www.youtube.com/watch?v=XiuSW_mEn7g

36	987	654	2	20	99	456	957	555	420	66	3
0	1	2	3	4	5	6	7	8	9	10	11
20	420	2	3	654	555	36	456	66	987	957	99
0	1	2	3	4	5	6	7	8	9	10	11
2	3	20	420	36	654	555	456	957	66	987	99
0	1	2	3	4	5	6	7	8	9	10	11
2	3	20	36	66	99	420	456	555	654	957	987
0	1	2	3	4	5	6	7	8	9	10	11



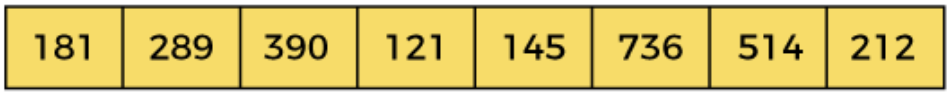
sorting the integers according to units, tens and hundreds place digits



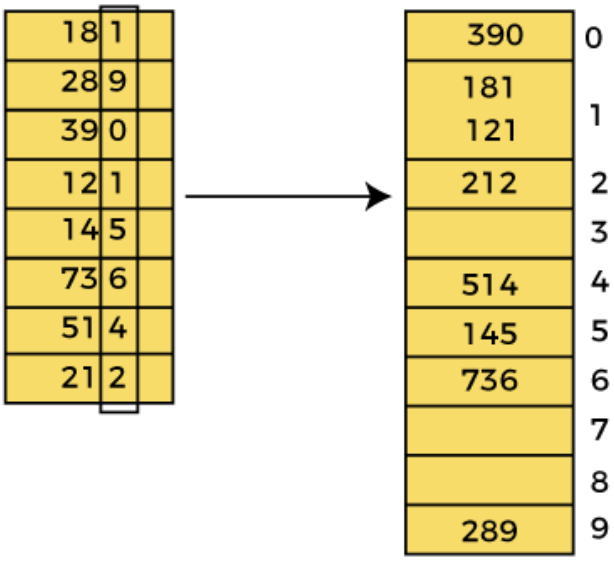
Ans.

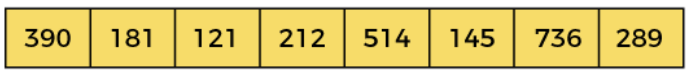
2	24	45	66	75	90	170	802

Original Array:

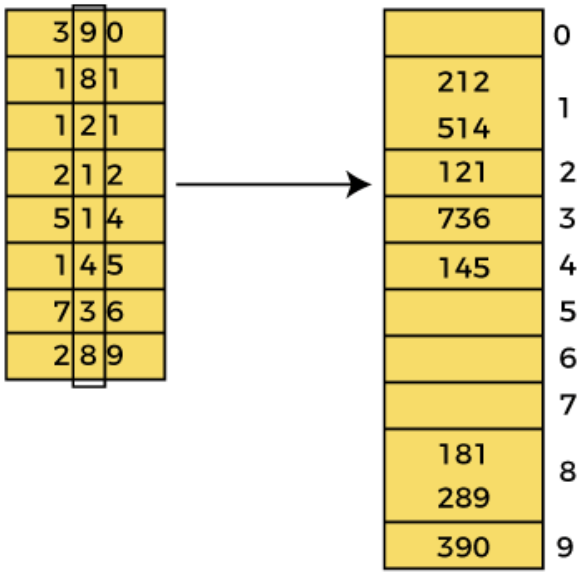


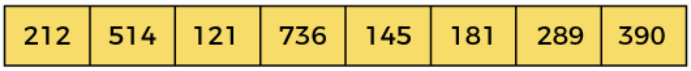
Pass 1:



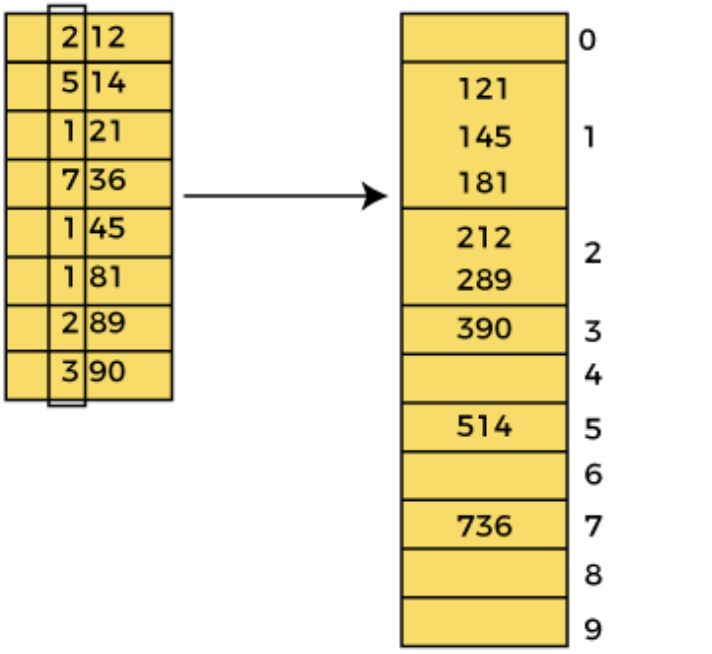


Pass 2:





Pass 3:



Pseudocode:

```
function getMax(arr, n)
    mx = arr[0]
    for i = 1 to n - 1 do
        if arr[i] > mx then
            mx = arr[i]
        end if
    end for
    return mx
end function

function countSort(arr, n, exp)
    output[n]
    count[10] = {0}

    for i = 0 to n - 1 do
        count[(arr[i] / exp) % 10] += 1
    end for

    for i = 1 to 9 do
        count[i] += count[i - 1]
    end for

    for i = n - 1 downto 0 do
        output[count[(arr[i] / exp) % 10] - 1] = arr[i]
        count[(arr[i] / exp) % 10] -= 1
    end for

    for i = 0 to n - 1 do
        arr[i] = output[i]
    end for
end function

function radixsort(arr, n)
    m = getMax(arr, n)
    for exp = 1 to m do
        countSort(arr, n, exp)
        exp *= 10
    end for
end function
```

Time Complexity:

Worst Case - O(N * K) | Average Case - O(N * K) | Best Case - O(N + K)

Lab Tasks

Q1: Given below the array, implement quick sort that simply chooses the middle element as the pivot and sort accordingly.

54 26 93 17 77 31 44 55 20

Q2: Given below the array, sort the array in ascending as well as descending order and return it using radix sort

36 987 654 2 20 99 456 957 555 420 66 3

Q3. Given below the array, sort the array in ascending as well as descending order and return it using Merge sort.

88 56 26 45 21 35 78 62 25 12 35

Q4. Create a program that visualizes the merge sort algorithm, showing how the array is divided and merged step by step. Discuss how visual representation aids in understanding the algorithm.

Example Output:

Initial Array: [38] [27] [43] [3] [9] [82] [10] [56]

Merging: [38] [27] [43] [3] [9] [82] [10] [56] [27] [38] [43] [3] [9] [82] [10] [56]

Merging: [27] [38] [43] [3] [9] [82] [10] [56] [3] [9] [27] [38] [43] [82] [10] [56]

Merging: [3] [9] [27] [38] [43] [82] [10] [56] [3] [9] [27] [38] [43] [82] [10] [56]

Merging: [3] [9] [27] [38] [43] [82] [10] [56] [3] [9] [10] [27] [38] [43] [82] [56]

Merging: [3] [9] [10] [27] [38] [43] [82] [56] [3] [9] [10] [27] [38] [43] [56] [82]

Sorted Array: [3] [9] [10] [27] [38] [43] [56] [82]

Q5. Given a list of IPv4 addresses, implement a function to sort them in ascending order using Radix Sort. Discuss how to treat each octet as a separate digit during the sorting process.

Example Output:

Initial IP Addresses:

192.168.1.1

10.0.0.2

172.16.0.1

192.168.0.1

10.0.0.1

Sorted IP Addresses:

10.0.0.1

10.0.0.2

172.16.0.1

192.168.0.1

192.168.1.1