**Problem Statement**

Raju, an aspiring programmer, is working on a sorting program using the merge sort algorithm. He wants to sort the array and print the sorted array in descending order. Write a program to implement the merge sort algorithm.

**Input format :**

The first line consists of an integer **N,**representing the number of elements in the array.

The second line consists of**N** space-separated integers, representing the elements of the array.

**Output format :**

The output displays **N** space-separated integers sorted in descending order.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ N ≤ 20

1 ≤ elements of the array ≤ 100

**Sample test cases :**

**Input 1 :**

8

78 34 23 76 12 10 4 79

**Output 1 :**

79 78 76 34 23 12 10 4

**Input 2 :**

5

6 5 4 3 2

**Output 2 :**

6 5 4 3 2

**Solution:**

import java.util.Scanner;

class MergeSortDescending {

    // Function to merge two subarrays into a single sorted array in descending order

    public static void merge(int[] arr, int left, int mid, int right) {

        // Calculate sizes of the two subarrays to be merged

        int n1 = mid - left + 1; // Size of left subarray

        int n2 = right - mid;    // Size of right subarray

        // Create temporary arrays to hold the subarrays

        int[] L = new int[n1];

        int[] R = new int[n2];

        // Copy data to temporary arrays

        for (int i = 0; i < n1; ++i)

            L[i] = arr[left + i]; // Copy left subarray

        for (int j = 0; j < n2; ++j)

            R[j] = arr[mid + 1 + j]; // Copy right subarray

        // Initial indexes for left, right, and merged subarrays

        int i = 0, j = 0;

        int k = left;

        // Merge the temp arrays back into arr[left..right]

        while (i < n1 && j < n2) { // While there are elements in both subarrays

            if (L[i] >= R[j]) { // Compare elements from both subarrays

                arr[k] = L[i]; // Place the larger element in the merged array

                i++; // Move to the next element in the left subarray

            } else {

                arr[k] = R[j]; // Place the larger element from the right subarray

                j++; // Move to the next element in the right subarray

            }

            k++; // Move to the next position in the merged array

        }

        // Copy remaining elements of left subarray, if any

        while (i < n1) {

            arr[k] = L[i];

            i++;

            k++;

        }

        // Copy remaining elements of right subarray, if any

        while (j < n2) {

            arr[k] = R[j];

            j++;

            k++;

        }

    }

    // Recursive function to sort the array using merge sort

    public static void mergeSort(int[] arr, int left, int right) {

        if (left < right) {

            // Find the middle point to divide the array into two halves

            int mid = (left + right) / 2;

            // Recursively sort the first and second halves

            mergeSort(arr, left, mid);

            mergeSort(arr, mid + 1, right);

            // Merge the sorted halves

            merge(arr, left, mid, right);

        }

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Read the number of elements in the array

        int N = scanner.nextInt();

        int[] array = new int[N];

        // Read the elements of the array

        for (int i = 0; i < N; i++) {

            array[i] = scanner.nextInt();

        }

        scanner.close(); // Close the scanner to free up resources

        // Call the mergeSort function to sort the array in descending order

        mergeSort(array, 0, N - 1);

        // Print the sorted array

        for (int i = 0; i < N; i++) {

            System.out.print(array[i] + " ");

        }

    }

}

**Detailed Comments:**

1. **Import Statement**:
   * import java.util.Scanner; – Imports the Scanner class to read input from the user.
2. **Class Definition**:
   * class MergeSortDescending – Defines the class named MergeSortDescending.
3. **Merge Function**:
   * public static void merge(int[] arr, int left, int mid, int right) – Merges two sorted subarrays into a single sorted array in descending order.
     + int n1 = mid - left + 1; – Calculates the size of the left subarray.
     + int n2 = right - mid; – Calculates the size of the right subarray.
     + int[] L = new int[n1]; and int[] R = new int[n2]; – Creates temporary arrays for the left and right subarrays.
     + for (int i = 0; i < n1; ++i) – Copies data from the original array to the left subarray.
     + for (int j = 0; j < n2; ++j) – Copies data from the original array to the right subarray.
     + while (i < n1 && j < n2) – Merges the two subarrays while comparing elements.
     + while (i < n1) – Copies any remaining elements from the left subarray.
     + while (j < n2) – Copies any remaining elements from the right subarray.
4. **MergeSort Function**:
   * public static void mergeSort(int[] arr, int left, int right) – Recursively sorts the array using the merge sort algorithm.
     + int mid = (left + right) / 2; – Finds the middle point of the array.
     + mergeSort(arr, left, mid); – Recursively sorts the first half.
     + mergeSort(arr, mid + 1, right); – Recursively sorts the second half.
     + merge(arr, left, mid, right); – Merges the sorted halves.
5. **Main Method**:
   * public static void main(String[] args) – Entry point of the program.
     + Scanner scanner = new Scanner(System.in); – Creates a Scanner object to read input.
     + int N = scanner.nextInt(); – Reads the number of elements.
     + int[] array = new int[N]; – Initializes an array of size N.
     + for (int i = 0; i < N; i++) – Reads elements into the array.
     + scanner.close(); – Closes the scanner.
     + mergeSort(array, 0, N - 1); – Sorts the array in descending order using merge sort.
     + for (int i = 0; i < N; i++) – Prints the sorted array.

**Merge Function**

The merge function is responsible for combining two sorted subarrays into a single sorted array. In this case, the function merges subarrays in descending order.

**Parameters:**

* arr: The main array that contains the subarrays to be merged.
* left: The starting index of the left subarray.
* mid: The ending index of the left subarray and the index just before the beginning of the right subarray.
* right: The ending index of the right subarray.

**Process:**

1. **Calculate the Sizes**:
   * n1 = mid - left + 1: Size of the left subarray.
   * n2 = right - mid: Size of the right subarray.
2. **Create Temporary Arrays**:
   * int[] L = new int[n1];: Temporary array for the left subarray.
   * int[] R = new int[n2];: Temporary array for the right subarray.
3. **Copy Data**:
   * Copy elements from the main array into L and R.
4. **Merge**:
   * Compare elements of L and R.
   * Place the larger element into the main array to ensure descending order.
5. **Copy Remaining Elements**:
   * If there are any remaining elements in L or R, copy them to the main array.

**Example:**

Consider an array [9, 3, 5, 2, 8, 1], and we want to merge subarrays [9, 3, 5] and [2, 8, 1].

* **Left Subarray (L)**: [9, 3, 5]
* **Right Subarray (R)**: [2, 8, 1]

Steps:

1. **Initialize Pointers**:
   * i = 0 (for L)
   * j = 0 (for R)
   * k = left (start index for merging)
2. **Comparison and Merging**:
   * Compare L[i] (9) and R[j] (2). Since 9 > 2, put 9 in the main array.
   * Compare L[i] (3) and R[j] (2). Since 3 > 2, put 3 in the main array.
   * Compare L[i] (5) and R[j] (2). Since 5 > 2, put 5 in the main array.
   * All elements in L are used up. Put remaining elements of R into the main array.

Result after merging: [9, 5, 3, 8, 2, 1].

**MergeSort Function**

The mergeSort function is a recursive function that divides the array into two halves, sorts each half, and then merges them.

**Parameters:**

* arr: The array to be sorted.
* left: The starting index of the subarray to be sorted.
* right: The ending index of the subarray to be sorted.

**Process:**

1. **Base Case**:
   * If left < right, proceed with the sorting. If not, the array or subarray has one or no elements and is already sorted.
2. **Divide**:
   * Find the middle index: mid = (left + right) / 2.
   * Recursively sort the left half: mergeSort(arr, left, mid).
   * Recursively sort the right half: mergeSort(arr, mid + 1, right).
3. **Merge**:
   * Merge the two sorted halves: merge(arr, left, mid, right).

**Example:**

Consider the array [9, 3, 5, 2, 8, 1].

1. **Initial Call**:
   * mergeSort(arr, 0, 5)
2. **Divide**:
   * mid = (0 + 5) / 2 = 2
   * Sort left half: mergeSort(arr, 0, 2)
   * Sort right half: mergeSort(arr, 3, 5)
3. **Sort Left Half**:
   * mergeSort(arr, 0, 2) divides into [9, 3, 5]
     + mergeSort(arr, 0, 1) divides into [9, 3]
       - Further divides into [9] and [3], and merges them to [9, 3]
     + mergeSort(arr, 2, 2) sorts [5]
     + Merge [9, 3] and [5] into [9, 5, 3]
4. **Sort Right Half**:
   * mergeSort(arr, 3, 5) divides into [2, 8, 1]
     + mergeSort(arr, 3, 4) divides into [2, 8]
       - Further divides into [2] and [8], and merges them to [8, 2]
     + mergeSort(arr, 5, 5) sorts [1]
     + Merge [8, 2] and [1] into [8, 2, 1]
5. **Final Merge**:
   * Merge [9, 5, 3] and [8, 2, 1] into [9, 8, 5, 3, 2, 1]

So, after applying merge sort, the sorted array in descending order is [9, 8, 5, 3, 2, 1].

This step-by-step breakdown should help clarify how the merge and mergeSort functions work together to sort an array in descending order.

**Why n1 = mid - left + 1?**

When you calculate the size of the left subarray, you are determining how many elements are included from the left index up to and including the mid index.

Consider the following example:

**Example Array:**

css

Copy code

arr = [a, b, c, d, e, f, g, h]

If you want to sort the subarray from index left = 0 to mid = 3, the subarray is:

csharp

Copy code

[a, b, c, d]

* **left**: 0 (starting index of the left subarray)
* **mid**: 3 (ending index of the left subarray)

The size of this subarray is calculated as follows:

* mid - left + 1 = 3 - 0 + 1 = 4

The +1 accounts for including both left and mid indices in the subarray. Without the +1, you would incorrectly calculate the size as mid - left, which would exclude the element at the mid index.

**Why n2 = right - mid?**

For the right subarray, the size calculation is slightly different:

When you calculate the size of the right subarray, you are considering elements from mid + 1 to right.

Using the same example array and indices:

css

Copy code

arr = [a, b, c, d, e, f, g, h]

If the left index is 0, mid is 3, and right is 7, the right subarray is:

csharp

Copy code

[e, f, g, h]

* **mid + 1**: 4 (starting index of the right subarray)
* **right**: 7 (ending index of the right subarray)

The size of this subarray is:

* right - (mid + 1) + 1 = 7 - 4 + 1 = 4

In this case, the +1 ensures you correctly count the element at right index.

**Summary**

The +1 in n1 = mid - left + 1 is necessary to ensure that both the starting (left) and ending (mid) indices are included in the subarray size calculation. Similarly, n2 = right - mid is used for the right subarray size calculation where the +1 is accounted for when merging to ensure all elements are included correctly.

**Problem Statement**

Given an array of integers, the program should sort the array in non-decreasing order using the 3-way merge sort technique. This involves dividing the array into three parts, recursively sorting each part, and then merging them back together in sorted order. The program should read an array of integers, perform the 3-way merge sort, and output the sorted array.

**Input format :**

The first line of input consists of an integer**N,** representing the number of elements in the array.

The second line of input consists of **N** space-separated integers, representing the array elements.

**Output format :**

The output prints a single line containing the sorted array, with each integer separated by a space.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

In this scenario, the test cases fall under the following constraints:

1 ≤ N ≤ 20

-100 ≤ array element ≤ 100

**Sample test cases :**

**Input 1 :**

10

42 -2 -45 78 30 -42 10 19 73 93

**Output 1 :**

-45 -42 -2 10 19 30 42 73 78 93

**Input 2 :**

2

23 -19

**Output 2 :**

-19 23

**Solution:**

import java.util.Scanner;

class ThreeWayMergeSort {

    // Function to merge three sorted subarrays into a single sorted array

    private static void merge(int[] arr, int left, int mid1, int mid2, int right) {

        // Calculate sizes of the three subarrays

        int n1 = mid1 - left + 1; // Size of the first subarray

        int n2 = mid2 - mid1;     // Size of the second subarray

        int n3 = right - mid2;    // Size of the third subarray

        // Create temporary arrays for the three subarrays

        int[] L1 = new int[n1];

        int[] L2 = new int[n2];

        int[] L3 = new int[n3];

        // Copy data into the temporary arrays

        for (int i = 0; i < n1; i++)

            L1[i] = arr[left + i]; // Copy first subarray

        for (int i = 0; i < n2; i++)

            L2[i] = arr[mid1 + 1 + i]; // Copy second subarray

        for (int i = 0; i < n3; i++)

            L3[i] = arr[mid2 + 1 + i]; // Copy third subarray

        // Initialize pointers for the temporary arrays and the main array

        int i = 0, j = 0, k = 0;

        int l = left;

        // Merge the three subarrays

        while (i < n1 && j < n2 && k < n3) {

            // Compare elements from all three subarrays

            if (L1[i] <= L2[j] && L1[i] <= L3[k]) {

                arr[l++] = L1[i++]; // Place smallest element in main array

            } else if (L2[j] <= L1[i] && L2[j] <= L3[k]) {

                arr[l++] = L2[j++]; // Place smallest element in main array

            } else {

                arr[l++] = L3[k++]; // Place smallest element in main array

            }

        }

        // Merge remaining elements from L1 and L2

        while (i < n1 && j < n2) {

            if (L1[i] <= L2[j]) {

                arr[l++] = L1[i++]; // Place smallest element in main array

            } else {

                arr[l++] = L2[j++]; // Place smallest element in main array

            }

        }

        // Merge remaining elements from L2 and L3

        while (j < n2 && k < n3) {

            if (L2[j] <= L3[k]) {

                arr[l++] = L2[j++]; // Place smallest element in main array

            } else {

                arr[l++] = L3[k++]; // Place smallest element in main array

            }

        }

        // Merge remaining elements from L1 and L3

        while (i < n1 && k < n3) {

            if (L1[i] <= L3[k]) {

                arr[l++] = L1[i++]; // Place smallest element in main array

            } else {

                arr[l++] = L3[k++]; // Place smallest element in main array

            }

        }

        // Copy remaining elements from L1

        while (i < n1) {

            arr[l++] = L1[i++];

        }

        // Copy remaining elements from L2

        while (j < n2) {

            arr[l++] = L2[j++];

        }

        // Copy remaining elements from L3

        while (k < n3) {

            arr[l++] = L3[k++];

        }

    }

    // Function to sort the array using 3-way merge sort

    private static void mergeSort(int[] arr, int left, int right) {

        if (left < right) {

            // Calculate indices for dividing the array into three parts

            int third = (right - left + 1) / 3;

            int mid1 = left + third - 1;

            int mid2 = left + 2 \* third;

            // Recursively sort the three subarrays

            mergeSort(arr, left, mid1);

            mergeSort(arr, mid1 + 1, mid2);

            mergeSort(arr, mid2 + 1, right);

            // Merge the sorted subarrays

            merge(arr, left, mid1, mid2, right);

        }

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Read the number of elements in the array

        int N = scanner.nextInt();

        int[] array = new int[N];

        // Read the elements of the array

        for (int i = 0; i < N; i++) {

            array[i] = scanner.nextInt();

        }

        scanner.close(); // Close the scanner to free up resources

        // Call the mergeSort function to sort the array

        mergeSort(array, 0, N - 1);

        // Print the sorted array

        for (int i = 0; i < N; i++) {

            System.out.print(array[i] + " ");

        }

    }

}

**Detailed Explanation:**

**merge Function**

1. **Calculate Sizes**:
   * n1 = mid1 - left + 1: Size of the first subarray.
   * n2 = mid2 - mid1: Size of the second subarray.
   * n3 = right - mid2: Size of the third subarray.
2. **Create Temporary Arrays**:
   * int[] L1 = new int[n1]; – Temporary array for the first subarray.
   * int[] L2 = new int[n2]; – Temporary array for the second subarray.
   * int[] L3 = new int[n3]; – Temporary array for the third subarray.
3. **Copy Data**:
   * Copy elements from the main array into L1, L2, and L3.
4. **Merge the Subarrays**:
   * Compare elements from L1, L2, and L3 and merge them into the main array arr while maintaining sorted order.
5. **Copy Remaining Elements**:
   * Copy any remaining elements from L1, L2, and L3 to the main array if there are any left after the main merging loop.

**mergeSort Function**

1. **Base Case**:
   * If left < right, the array segment has more than one element and needs to be sorted.
2. **Divide the Array**:
   * Calculate third to divide the array into approximately equal parts.
   * Determine mid1 and mid2 to split the array into three segments.
3. **Recursive Sort**:
   * Recursively sort the three segments of the array.
4. **Merge**:
   * Merge the sorted segments using the merge function.

**main Method**

1. **Read Input**:
   * Use Scanner to read the number of elements and the array elements from the user.
2. **Sort the Array**:
   * Call mergeSort to sort the array.
3. **Print the Sorted Array**:
   * Print the sorted array elements.

This code provides a detailed implementation of 3-way merge sort in Java, handling sorting and merging of array segments efficiently.

Certainly! Let's walk through an example to demonstrate how each function (merge and mergeSort) works in the ThreeWayMergeSort class.

**Example Array**

Consider the following example array to be sorted using the 3-way merge sort algorithm:

css

Copy code

arr = [6, 3, 8, 7, 2, 5, 4, 1]

**Step-by-Step Execution**

**1. Initial Call to mergeSort**

* **Input**: arr = [6, 3, 8, 7, 2, 5, 4, 1], left = 0, right = 7
* **Calculation**:
  + third = (7 - 0 + 1) / 3 = 8 / 3 = 2 (integer division)
  + mid1 = 0 + 2 - 1 = 1
  + mid2 = 0 + 2 \* 2 = 4
* **Array Split**:
  + First subarray: [6, 3, 8, 7] (left = 0 to mid1 = 1)
  + Second subarray: [2, 5] (mid1 + 1 = 2 to mid2 = 4)
  + Third subarray: [4, 1] (mid2 + 1 = 5 to right = 7)

**2. Recursive Calls**

* **First Subarray**: [6, 3, 8, 7]
  + **Input**: arr = [6, 3, 8, 7], left = 0, right = 3
  + third = (3 - 0 + 1) / 3 = 4 / 3 = 1
  + mid1 = 0 + 1 - 1 = 0
  + mid2 = 0 + 2 \* 1 = 2
  + **Split**:
    - [6] (left = 0 to mid1 = 0)
    - [3, 8] (mid1 + 1 = 1 to mid2 = 2)
    - [7] (mid2 + 1 = 3 to right = 3)
* **Second Subarray**: [2, 5]
  + **Input**: arr = [2, 5], left = 2, right = 4
  + third = (4 - 2 + 1) / 3 = 3 / 3 = 1
  + mid1 = 2 + 1 - 1 = 2
  + mid2 = 2 + 2 \* 1 = 4
  + **Split**:
    - [2] (left = 2 to mid1 = 2)
    - [5] (mid1 + 1 = 3 to right = 4)
* **Third Subarray**: [4, 1]
  + **Input**: arr = [4, 1], left = 5, right = 7
  + third = (7 - 5 + 1) / 3 = 3 / 3 = 1
  + mid1 = 5 + 1 - 1 = 5
  + mid2 = 5 + 2 \* 1 = 7
  + **Split**:
    - [4] (left = 5 to mid1 = 5)
    - [1] (mid1 + 1 = 6 to right = 7)

**3. Base Case Reached**

For each single-element subarray (e.g., [6], [3], [8], etc.), the array is already sorted. These subarrays are merged back together.

**4. Merging Process**

* **Merging [6], [3, 8], and [7]**:
  + Merge [6] and [3, 8]:
    - Compare elements: [3, 6, 8]
  + Merge [3, 6, 8] and [7]:
    - Compare elements: [3, 6, 7, 8]
* **Merging [2] and [5]**:
  + Result: [2, 5]
* **Merging [4] and [1]**:
  + Result: [1, 4]
* **Final Merge**:
  + Merge [3, 6, 7, 8], [2, 5], and [1, 4]:
    - Compare elements:
      * [1, 2, 3, 4, 5, 6, 7, 8]

**Detailed Example for merge Function**

Consider merging these arrays:

* L1 = [6]
* L2 = [3, 8]
* L3 = [7]

**Merging Process**:

1. Compare the smallest elements:
   * 6 (from L1), 3 (from L2), and 7 (from L3)
   * 3 is the smallest, so place it in the main array: [3]
   * Continue comparing with updated smallest elements
2. Resulting merged array after full comparison:
   * [3, 6, 7, 8]

Certainly! Let's walk through an example of how each function works in the ThreeWayMergeSort class using a specific array.

**Example Array**

Consider the array: [3, 1, 4, 1, 5, 9, 2, 6, 5]

**mergeSort Function**

**Input:**

* Array: [3, 1, 4, 1, 5, 9, 2, 6, 5]
* Left: 0
* Right: 8

**Process:**

1. **Calculate Thirds:**
   * Length of array segment: right - left + 1 = 8 - 0 + 1 = 9
   * third = (9) / 3 = 3
   * mid1 = left + third - 1 = 0 + 3 - 1 = 2
   * mid2 = left + 2 \* third = 0 + 2 \* 3 = 6
2. **Divide Array into Three Segments:**
   * First segment: [3, 1, 4] (from index 0 to 2)
   * Second segment: [1, 5, 9] (from index 3 to 6)
   * Third segment: [2, 6, 5] (from index 7 to 8)
3. **Recursively Sort Each Segment:**
   * Sort [3, 1, 4]:
     + Divide into [3] and [1, 4]
     + Sort [1, 4] into [1, 4]
     + Merge [3] and [1, 4] into [1, 3, 4]
   * Sort [1, 5, 9]:
     + Divide into [1] and [5, 9]
     + Sort [5, 9] into [5, 9]
     + Merge [1] and [5, 9] into [1, 5, 9]
   * Sort [2, 6, 5]:
     + Divide into [2] and [6, 5]
     + Sort [6, 5] into [5, 6]
     + Merge [2] and [5, 6] into [2, 5, 6]
4. **Merge Sorted Segments:**
   * Merge [1, 3, 4], [1, 5, 9], and [2, 5, 6] into [1, 1, 2, 3, 4, 5, 5, 6, 9]

**merge Function**

Let's take the merging part for the array segments [1, 3, 4], [1, 5, 9], and [2, 5, 6].

**Input:**

* arr: [1, 3, 4, 1, 5, 9, 2, 5, 6]
* left: 0
* mid1: 2
* mid2: 5
* right: 8

**Process:**

1. **Create Temporary Arrays:**
   * L1: [1, 3, 4]
   * L2: [1, 5, 9]
   * L3: [2, 5, 6]
2. **Merge Temporary Arrays:**
   * Initialize indices: i = 0, j = 0, k = 0, and l = 0.
   * Compare and place elements in arr:
     + Compare L1[0] (1), L2[0] (1), L3[0] (2)
     + Place 1 from L1, move i to 1.
     + Compare L1[1] (3), L2[0] (1), L3[0] (2)
     + Place 1 from L2, move j to 1.
     + Continue this process until all elements are merged into arr.

Resulting arr: [1, 1, 2, 3, 4, 5, 5, 6, 9]

**Full Example Walkthrough:**

**Initial Array:**

csharp

Copy code

[3, 1, 4, 1, 5, 9, 2, 6, 5]

**After First Level of Sorting:**

* First segment: [1, 3, 4]
* Second segment: [1, 5, 9]
* Third segment: [2, 5, 6]

**After Merging Segments:**

csharp

Copy code

[1, 1, 2, 3, 4, 5, 5, 6, 9]

**Summary**

1. **mergeSort Function:**
   * Recursively divides the array into three parts.
   * Calls itself to sort each part.
   * Uses the merge function to combine the sorted parts.
2. **merge Function:**
   * Merges three sorted subarrays into one sorted array.
   * Uses temporary arrays to hold subarrays.
   * Merges these arrays by comparing elements from each subarray.

By following these steps, the ThreeWayMergeSort class efficiently sorts an array of integers using a three-way merge approach.

**Problem Statement**

Seema, a dedicated enthusiast in computer science, is eager to explore different sorting algorithms as part of her learning journey. She has recently become intrigued by the efficiency and elegance of the quick sort algorithm and wishes to implement it herself additionally, she wants to display the pivot element at each partition.

Your task is to assist Seema in developing a program that sorts an array of integers in descending order using the quick sort algorithm.

**Input format :**

The first line contains an integer**n,** representing the number of elements in the array.

The second line contains **n** space-separated integers, representing the elements of the array.

**Output format :**

The output prints **n** space-separated integers, representing the sorted elements in descending order.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

In this scenario, the given test cases will fall under the following constraints:

1 ≤ n ≤ 100

-100 ≤ Each element of the array ≤ 103

**Sample test cases :**

**Input 1 :**

5

5 2 9 3 -5

**Output 1 :**

Pivot element: -5

Pivot element: 3

Pivot element: 9

9 5 3 2 -5

**Input 2 :**

8

-5 0 -9 -7 4 3 -1 5

**Output 2 :**

Pivot element: 5

Pivot element: -5

Pivot element: -1

Pivot element: 3

Pivot element: -7

5 4 3 0 -1 -5 -7 -9

**Input 3 :**

6

10 7 8 9 1 5

**Output 3 :**

Pivot element: 5

Pivot element: 9

Pivot element: 7

10 9 8 7 5 1

**Solution:**

import java.util.Scanner;

class QuickSortDescending {

    /\*\*

     \* Partitions the array into two segments: elements greater than the pivot and elements less than or equal to the pivot.

     \*

     \* @param arr The array to partition.

     \* @param low The starting index of the segment to partition.

     \* @param high The ending index of the segment to partition.

     \* @return The index of the pivot element after partitioning.

     \*/

    private static int partition(int[] arr, int low, int high) {

        int pivot = arr[high]; // Choose the last element as the pivot

        int i = low - 1; // Index of the smaller element

        // Iterate over the array from 'low' to 'high - 1'

        for (int j = low; j < high; j++) {

            if (arr[j] > pivot) { // Compare current element with the pivot

                i++; // Move index of the smaller element

                // Swap arr[i] and arr[j]

                int temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

        // Swap the pivot element with the element at (i + 1)

        int temp = arr[i + 1];

        arr[i + 1] = arr[high];

        arr[high] = temp;

        // Print the pivot element

        System.out.println("Pivot element: " + pivot);

        // Return the index of the pivot element

        return i + 1;

    }

    /\*\*

     \* Sorts the array using the QuickSort algorithm.

     \*

     \* @param arr The array to sort.

     \* @param low The starting index of the segment to sort.

     \* @param high The ending index of the segment to sort.

     \*/

    private static void quickSort(int[] arr, int low, int high) {

        if (low < high) { // Base case: if the segment has more than one element

            int pi = partition(arr, low, high); // Partitioning index

            // Recursively sort the elements before and after the partition

            quickSort(arr, low, pi - 1);

            quickSort(arr, pi + 1, high);

        }

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        int n = scanner.nextInt(); // Read the number of elements

        int[] array = new int[n];

        for (int i = 0; i < n; i++) {

            array[i] = scanner.nextInt(); // Read each element into the array

        }

        scanner.close(); // Close the scanner

        quickSort(array, 0, n - 1); // Call quickSort to sort the array

        // Print the sorted array

        for (int i = 0; i < n; i++) {

            System.out.print(array[i] + " ");

        }

    }

}

**Detailed Explanation with Example**

**partition Function**

The partition function is crucial for the QuickSort algorithm. It rearranges the elements of the array such that all elements greater than the pivot come before all elements less than or equal to the pivot.

**Example:**

Array: [3, 1, 4, 1, 5]  
Pivot: 5 (last element)

**Steps:**

1. **Initial Setup:**
   * pivot = 5
   * i = -1 (index before the start of the segment)
2. **Iterate through the Array:**
   * Compare 3 with 5: 3 > 5 is false. No swap.
   * Compare 1 with 5: 1 > 5 is false. No swap.
   * Compare 4 with 5: 4 > 5 is false. No swap.
   * Compare 1 with 5: 1 > 5 is false. No swap.
3. **Swap Pivot:**
   * Swap pivot 5 with element at i + 1 (index 0).
   * Resulting array: [5, 1, 4, 1, 3]
4. **Output:**
   * Print pivot: 5
   * Return pivot index: 0

**quickSort Function**

The quickSort function sorts the array by dividing it into segments and recursively sorting those segments using the partition function.

**Example:**

Array: [5, 1, 4, 1, 3]  
Segment to sort: low = 0, high = 4

**Steps:**

1. **Partitioning:**
   * Call partition(arr, 0, 4)
   * Pivot: 5
   * Resulting array: [5, 1, 4, 1, 3]
   * Pivot index: 0
2. **Recursive Calls:**
   * quickSort(arr, 0, -1) (empty segment, base case).
   * quickSort(arr, 1, 4) (sort remaining segment [1, 4, 1, 3]).

**For Segment [1, 4, 1, 3]:**

1. **Partitioning:**
   * Pivot: 3
   * Resulting array: [1, 1, 3, 4, 5]
   * Pivot index: 2
2. **Recursive Calls:**
   * quickSort(arr, 1, 1) (single element, base case).
   * quickSort(arr, 3, 4) (sort segment [4, 5]).

**For Segment [4, 5]:**

1. **Partitioning:**
   * Pivot: 4
   * Resulting array: [1, 1, 3, 4, 5]
   * Pivot index: 3
2. **Recursive Calls:**
   * quickSort(arr, 3, 2) (empty segment, base case).
   * quickSort(arr, 4, 4) (single element, base case).

**Final Sorted Array:**

csharp

Copy code

[5, 4, 3, 1, 1]

**main Method**

The main method reads input, invokes quickSort, and prints the sorted array.

**Steps:**

1. **Read Input:**
   * Number of elements n
   * Array elements from the user
2. **Sort the Array:**
   * Call quickSort(array, 0, n - 1)
3. **Print Sorted Array:**
   * Iterate through the array and print each element.

**Summary**

* **partition Function:** Rearranges elements around the pivot and returns the pivot index.
* **quickSort Function:** Recursively sorts the array by partitioning it into segments.
* **main Method:** Handles user input, invokes the sorting function, and prints the result.

This detailed breakdown should help you understand how each function operates and how they work together to sort the array in descending order using QuickSort.

**Problem Statement**

Implement a program that performs QuickSort on an array of floating-point numbers and removes duplicate elements from the sorted array.

**Input format :**

The first line of input consists of the size of the array N.

The second line of input consists of the N elements of the array, separated by space.

**Output format :**

For each pass of the QuickSort algorithm, print a line containing the pivot element used for that pass.

Print a line containing "Final array: " followed by the space-separated elements of the final sorted array without any duplicates.

**Code constraints :**

1 ≤ N ≤ 15

1.0 ≤ array elements ≤ 10.0

**Sample test cases :**

**Input 1 :**

6

3 1 5 2 4 1

**Output 1 :**

Pivot for pass: 1.0

Pivot for pass: 3.0

Pivot for pass: 5.0

Final array: 1.0 2.0 3.0 4.0 5.0

**Input 2 :**

8

9.1 3.2 7.5 1.4 5.7 2.1 6.4 4.1

**Output 2 :**

Pivot for pass: 4.1

Pivot for pass: 2.1

Pivot for pass: 9.1

Pivot for pass: 6.4

Final array: 1.4 2.1 3.2 4.1 5.7 6.4 7.5 9.1

**Solution:**

import java.util.Arrays;

import java.util.Scanner;

import java.util.HashSet;

import java.util.LinkedHashSet;

import java.util.Set;

class QuickSortWithDuplicates {

    /\*\*

     \* Partitions the array into elements less than or equal to the pivot and those greater than the pivot.

     \*

     \* @param arr The array to partition.

     \* @param low The starting index of the segment to partition.

     \* @param high The ending index of the segment to partition.

     \* @return The index of the pivot element after partitioning.

     \*/

    private static int partition(float[] arr, int low, int high) {

        float pivot = arr[high]; // Choose the last element as the pivot

        int i = low - 1; // Index of the smaller element

        // Iterate over the array from 'low' to 'high - 1'

        for (int j = low; j < high; j++) {

            if (arr[j] <= pivot) { // Compare current element with the pivot

                i++; // Move index of the smaller element

                // Swap arr[i] and arr[j]

                float temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

        // Swap the pivot element with the element at (i + 1)

        float temp = arr[i + 1];

        arr[i + 1] = arr[high];

        arr[high] = temp;

        // Print the pivot element for debugging

        System.out.println("Pivot for pass: " + pivot);

        // Return the index of the pivot element

        return i + 1;

    }

    /\*\*

     \* Sorts the array using the QuickSort algorithm.

     \*

     \* @param arr The array to sort.

     \* @param low The starting index of the segment to sort.

     \* @param high The ending index of the segment to sort.

     \*/

    private static void quickSort(float[] arr, int low, int high) {

        if (low < high) { // Base case: if the segment has more than one element

            int pi = partition(arr, low, high); // Partitioning index

            // Recursively sort the elements before and after the partition

            quickSort(arr, low, pi - 1);

            quickSort(arr, pi + 1, high);

        }

    }

    /\*\*

     \* Removes duplicate elements from the array.

     \*

     \* @param arr The array from which duplicates are to be removed.

     \* @return A new array containing only unique elements.

     \*/

    private static float[] removeDuplicates(float[] arr) {

        Set<Float> set = new LinkedHashSet<>(); // Use LinkedHashSet to maintain insertion order

        for (float value : arr) {

            set.add(value); // Add each element to the set (duplicates are automatically removed)

        }

        // Convert the set back to an array

        float[] uniqueArray = new float[set.size()];

        int index = 0;

        for (float value : set) {

            uniqueArray[index++] = value;

        }

        return uniqueArray;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        int N = scanner.nextInt(); // Read the number of elements

        float[] array = new float[N];

        for (int i = 0; i < N; i++) {

            array[i] = scanner.nextFloat(); // Read each element into the array

        }

        scanner.close(); // Close the scanner

        quickSort(array, 0, N - 1); // Call quickSort to sort the array

        float[] finalArray = removeDuplicates(array); // Remove duplicates from the sorted array

        // Print the final array with unique elements

        System.out.print("Final array: ");

        for (float num : finalArray) {

            System.out.print(num + " ");

        }

        System.out.println();

    }

}

**Detailed Explanation with Example**

**partition Function**

**Purpose:** The partition function rearranges elements in the array such that elements less than or equal to the pivot are moved to the left of the pivot, and elements greater than the pivot are moved to the right.

**Example:**

Array: [3.1, 1.5, 4.2, 2.3, 5.6]  
Pivot: 5.6 (last element)

**Steps:**

1. **Initial Setup:**
   * pivot = 5.6
   * i = -1
2. **Iterate through the Array:**
   * Compare 3.1 with 5.6: 3.1 <= 5.6 is true. Swap 3.1 with itself (i.e., no change). i = 0.
   * Compare 1.5 with 5.6: 1.5 <= 5.6 is true. Swap 1.5 with itself (i.e., no change). i = 1.
   * Compare 4.2 with 5.6: 4.2 <= 5.6 is true. Swap 4.2 with itself (i.e., no change). i = 2.
   * Compare 2.3 with 5.6: 2.3 <= 5.6 is true. Swap 2.3 with itself (i.e., no change). i = 3.
3. **Swap Pivot:**
   * Swap pivot 5.6 with element at i + 1 (index 4).
   * Resulting array: [3.1, 1.5, 4.2, 2.3, 5.6]
4. **Output:**
   * Print pivot: 5.6
   * Return pivot index: 4

**quickSort Function**

**Purpose:** The quickSort function recursively sorts the array by dividing it into segments around the pivot.

**Example:**

Array: [3.1, 1.5, 4.2, 2.3, 5.6]  
Segment to sort: low = 0, high = 4

**Steps:**

1. **Partitioning:**
   * Call partition(arr, 0, 4)
   * Pivot: 5.6
   * Resulting array: [3.1, 1.5, 4.2, 2.3, 5.6]
   * Pivot index: 4
2. **Recursive Calls:**
   * quickSort(arr, 0, 3) (sort segment [3.1, 1.5, 4.2, 2.3]).

**For Segment [3.1, 1.5, 4.2, 2.3]:**

1. **Partitioning:**
   * Pivot: 2.3
   * Resulting array: [1.5, 2.3, 4.2, 3.1, 5.6]
   * Pivot index: 1
2. **Recursive Calls:**
   * quickSort(arr, 0, 0) (base case, single element).
   * quickSort(arr, 2, 3) (sort segment [4.2, 3.1]).

**For Segment [4.2, 3.1]:**

1. **Partitioning:**
   * Pivot: 3.1
   * Resulting array: [1.5, 2.3, 3.1, 4.2, 5.6]
   * Pivot index: 2
2. **Recursive Calls:**
   * quickSort(arr, 2, 1) (base case, single element).
   * quickSort(arr, 3, 3) (base case, single element).

**Final Sorted Array:**

csharp

Copy code

[1.5, 2.3, 3.1, 4.2, 5.6]

**removeDuplicates Function**

**Purpose:** The removeDuplicates function eliminates duplicate elements from the sorted array.

**Example:**

Array: [1.5, 2.3, 2.3, 3.1, 4.2, 4.2, 5.6]

**Steps:**

1. **Use LinkedHashSet:**
   * A LinkedHashSet automatically removes duplicates and maintains the insertion order.
2. **Add Elements to Set:**
   * Add 1.5, 2.3, 3.1, 4.2, 5.6 to the set.
3. **Convert Set to Array:**
   * Create a new array from the set.
   * Resulting array: [1.5, 2.3, 3.1, 4.2, 5.6]

**Final Output:**

sql

Copy code

Final array: 1.5 2.3 3.1 4.2 5.6

**main Method**

**Purpose:** The main method reads input, sorts the array, removes duplicates, and prints the final result.

**Steps:**

1. **Read Input:**
   * Number of elements N
   * Array elements from the user
2. **Sort the Array:**
   * Call quickSort(array, 0, N - 1)
3. **Remove Duplicates:**
   * Call removeDuplicates(array)
4. **Print Final Array:**
   * Print the unique elements of the sorted array.

**Summary**

* **partition Function:** Rearranges elements around the pivot.
* **quickSort Function:** Recursively sorts the array by partitioning it.
* **removeDuplicates Function:** Removes duplicate elements from the array.
* **main Method:** Reads input, sorts the array, removes duplicates, and prints the result.

This detailed breakdown should help you understand how each function operates and how they work together to sort the array and remove duplicates.