Bubble Sort

Given an array, arr[]. Sort the array using bubble sort algorithm.

Examples :

Input: arr[] = [4, 1, 3, 9, 7]

Output: [1, 3, 4, 7, 9]

Input: arr[] = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

public class BubbleSort {

public static void main(String[] args) {

// Example array to sort

int[] arr = {4, 1, 3, 9, 7};

// Print the original array

System.out.println("Original Array:");

printArray(arr);

// Perform Bubble Sort

bubbleSort(arr);

// Print the sorted array

System.out.println("Sorted Array:");

printArray(arr);

}

// Bubble Sort function

public static void bubbleSort(int[] arr) {

int n = arr.length;

// Outer loop for the number of passes

for (int i = 0; i < n - 1; i++) {

// Inner loop for comparing adjacent elements

for (int j = 0; j < n - i - 1; j++) {

// Swap if the current element is greater than the next element

if (arr[j] > arr[j + 1]) {

// Swap arr[j] and arr[j + 1]

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

// Utility function to print an array

public static void printArray(int[] arr) {

for (int value : arr) {

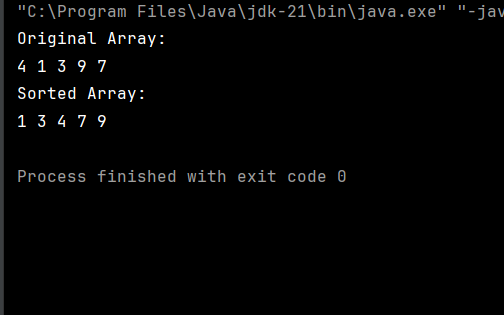
System.out.print(value + " ");

}

System.out.println();

}

}



Implement Quick Sort, a Divide and Conquer algorithm, to sort an array, arr[] in ascending order. Given an array, arr[], with starting index low and ending index high, complete the functions partition() and quickSort(). Use the last element as the pivot so that all elements less than or equal to the pivot come before it, and elements greater than the pivot follow it.

Note: The low and high are inclusive.

Examples:

Input: arr[] = [4, 1, 3, 9, 7]

Output: [1, 3, 4, 7, 9]

Explanation: After sorting, all elements are arranged in ascending order.

public class QuickSort {

// Function to sort an array using Quick Sort algorithm

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

if (low < high) {

// Partition the array and get the pivot index

int pivotIndex = partition(arr, low, high);

// Recursively sort elements before and after partition

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

// Partition function

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

// Choose the last element as the pivot

int pivot = arr[high];

// Index for the smaller element

int i = low - 1;

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

// If current element is smaller than or equal to pivot

if (arr[j] <= pivot) {

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

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

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Place the pivot in its correct position

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

// Return the partition index

return i + 1;

}

// Utility function to print an array

static void printArray(int arr[]) {

for (int value : arr) {

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

}

System.out.println();

}

// Main method to test the Quick Sort implementation

public static void main(String[] args) {

int arr[] = {10, 80, 30, 90, 40, 50, 70};

int n = arr.length;

System.out.println("Original Array:");

printArray(arr);

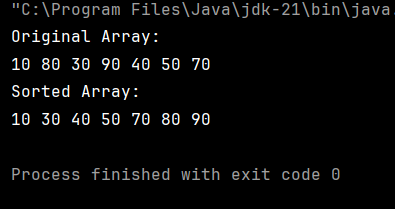
quickSort(arr, 0, n - 1);

System.out.println("Sorted Array:");

printArray(arr);

}

}



Non Reapeting Characters

Given a string s consisting of lowercase Latin Letters. Return the first non-repeating character in s. If there is no non-repeating character, return '$'.

Note: When you return '$' driver code will output -1.

Examples:

Input: s = "geeksforgeeks"

Output: 'f'

Explanation: In the given string, 'f' is the first character in the string which does not repeat.

import java.util.HashMap;

import java.util.Map;

public class NonRepeatingChars {

// Function to find the first non-repeating character in a string.

static char nonRepeatingChar(String s) {

// Step 1: Create a hashmap to store character frequencies

Map<Character, Integer> charCount = new HashMap<>();

// Step 2: Count the frequency of each character in the string

for (char c : s.toCharArray()) {

charCount.put(c, charCount.getOrDefault(c, 0) + 1);

}

// Step 3: Find the first character with frequency 1

for (char c : s.toCharArray()) {

if (charCount.get(c) == 1) {

return c; // Return the first non-repeating character

}

}

// If no non-repeating character is found, return a placeholder

return '-';

}

// Main method to test the function

public static void main(String[] args) {

String s = "swiss";

System.out.println(s);

char result = nonRepeatingChar(s);

if (result != '-') {

System.out.println("First non-repeating character: " + result);

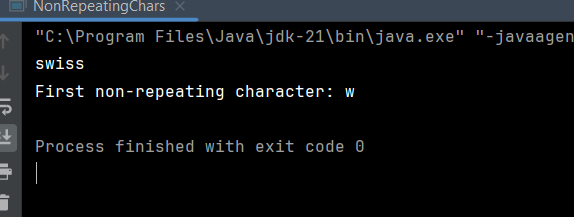
} else {

System.out.println("No non-repeating character found.");

}

}

}



Edit Distance

Given two strings s1 and s2. Return the minimum number of operations required to convert s1 to s2.

The possible operations are permitted:

Insert a character at any position of the string.

Remove any character from the string.

Replace any character from the string with any other character.

Examples:

Input: s1 = "geek", s2 = "gesek"

Output: 1

Explanation: One operation is required, inserting 's' between two 'e'.

Input : s1 = "gfg", s2 = "gfg"

Output: 0

Explanation: Both strings are same.

Input : s1 = "abc", s2 = "def"

Output: 3

Explanation: All characters need to be replaced to convert str1 to str2, requiring 3 replacement operations.

public class EditDistance {

public static int editDistance(String s1, String s2) {

int m = s1.length(); // Length of string s1

int n = s2.length(); // Length of string s2

// Create a DP table to store results of subproblems

int[][] dp = new int[m + 1][n + 1];

// Fill the DP table

for (int i = 0; i <= m; i++) {

for (int j = 0; j <= n; j++) {

if (i == 0) {

// If s1 is empty, we need to insert all characters of s2

dp[i][j] = j;

} else if (j == 0) {

// If s2 is empty, we need to remove all characters of s1

dp[i][j] = i;

} else if (s1.charAt(i - 1) == s2.charAt(j - 1)) {

// If the characters are the same, no operation is required

dp[i][j] = dp[i - 1][j - 1];

} else {

// If the characters are different, consider all possibilities

dp[i][j] = 1 + Math.min(

dp[i - 1][j], // Deletion

Math.min(

dp[i][j - 1], // Insertion

dp[i - 1][j - 1] // Substitution

)

);

}

}

}

// The value in dp[m][n] is the edit distance

return dp[m][n];

}

// Main method to test the function

public static void main(String[] args) {

String s1 = "geek";

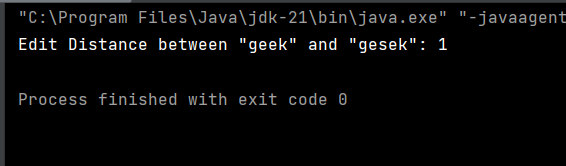
String s2 = "gesek";

int distance = editDistance(s1, s2);

System.out.println("Edit Distance between \"" + s1 + "\" and \"" + s2 + "\": " + distance);

}

}



k largest elements

Difficulty: MediumAccuracy: 53.56%Submissions: 163K+Points: 4

Given an array arr[] of positive integers and an integer k, Your task is to return k largest elements in decreasing order.

Examples

Input: arr[] = [12, 5, 787, 1, 23], k = 2

Output: [787, 23]

Explanation: 1st largest element in the array is 787 and second largest is 23.

Input: arr[] = [1, 23, 12, 9, 30, 2, 50], k = 3

Output: [50, 30, 23]

Explanation: Three Largest elements in the array are 50, 30 and 23.

Input: arr[] = [12, 23], k = 1

Output: [23]

Explanation: 1st Largest element in the array is 23.

import java.util.\*;

public class KthLargestArray {

static List<Integer> kLargest(int arr[], int k) {

// Create a min-heap (PriorityQueue)

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

// Add elements to the heap

for (int num : arr) {

minHeap.add(num);

// If heap size exceeds k, remove the smallest element

if (minHeap.size() > k) {

minHeap.poll();

}

}

// Extract elements from the heap into a list

List<Integer> result = new ArrayList<>(minHeap);

// Sort the result in descending order

result.sort(Collections.reverseOrder());

return result;

}

public static void main(String[] args) {

int arr1[] = {12, 5, 787, 1, 23};

int k1 = 2;

int arr2[] = {1, 23, 12, 9, 30, 2, 50};

int k2 = 3;

int arr3[] = {12, 23};

int k3 = 1;

System.out.println("Input: " + Arrays.toString(arr1) + ", k = " + k1);

System.out.println("Output: " + kLargest(arr1, k1));

System.out.println("\nInput: " + Arrays.toString(arr2) + ", k = " + k2);

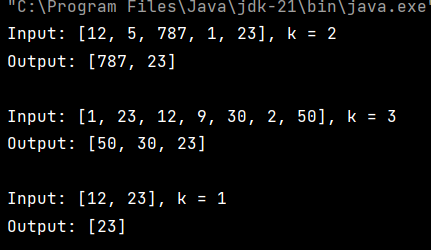
System.out.println("Output: " + kLargest(arr2, k2));

System.out.println("\nInput: " + Arrays.toString(arr3) + ", k = " + k3);

System.out.println("Output: " + kLargest(arr3, k3));

}

}



Form the Largest Number

Difficulty: MediumAccuracy: 37.82%Submissions: 162K+Points: 4

Given an array of strings arr[] representing non-negative integers, arrange them so that after concatenating them in order, it results in the largest possible number. Since the result may be very large, return it as a string.

Note: There are no leading zeros in each array element.

Examples:

Input: arr[] = ["3", "30", "34", "5", "9"]

Output: "9534330"

Explanation: Given numbers are {"3", "30", "34", "5", "9"}, the arrangement "9534330" gives the largest value.

import java.util.Arrays;

public class LargestNumber {

public static String printLargest(String[] arr) {

// Sort the array using a custom comparator

Arrays.sort(arr, (a, b) -> (b + a).compareTo(a + b));

// If the largest number is "0", return "0" (handles edge cases like ["0", "0"])

if (arr[0].equals("0")) {

return "0";

}

// Build the largest number by concatenating the sorted strings

StringBuilder result = new StringBuilder();

for (String s : arr) {

result.append(s);

}

return result.toString();

}

public static void main(String[] args) {

String[] arr1 = {"3", "30", "34", "5", "9"};

String[] arr2 = {"10", "2"};

String[] arr3 = {"0", "0"};

for(String s : arr1){

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

}

System.out.println();

System.out.println(printLargest(arr1)); // Output: 9534330

System.out.println(printLargest(arr2)); // Output: 210

System.out.println(printLargest(arr3)); // Output: 0

}

}

