1. Given an array containing only 0’s, 1’s, and 2’s, sort it in linear time and using constant space.

Solution:

public static void threeWayPartition(int[] A)

    {

        int start = 0, mid = 0;

        int pivot = 1;

        int end = A.length - 1;

        while (mid <= end)

        {

            if (A[mid] < pivot)         // current element is 0

            {

                swap(A, start, mid);

                ++start;

                ++mid;

            }

            else if (A[mid] > pivot)    // current element is 2

            {

                swap(A, mid, end);

                --end;

            }

            else {                      // current element is 1

                ++mid;

            }

        }

    }

    // Utility function to swap elements `A[i]` and `A[j]` in the array

    private static void swap(int[] A, int i, int j)

    {

        int temp = A[i];

        A[i] = A[j];

        A[j] = temp;

    }

TC: O(n)

SC: O(1)

1. Given two sorted arrays, X[] and Y[] of size m and n each, merge elements of X[] with elements of array Y[] by maintaining the sorted order, i.e., fill X[] with the first m smallest elements and fill Y[] with remaining elements.

Input:



X[] = { 1, 4, 7, 8, 10 }



Y[] = { 2, 3, 9 }

Output:



X[] = { 1, 2, 3, 4, 7 }

Y[] = { 8, 9, 10 }

Solution:

 // Function to in-place merge two sorted arrays X[] and Y[]

    // invariant: `X[]` and `Y[]` are sorted at any point

    public static void merge(int[] X, int[] Y)

    {

        int m = X.length;

        int n = Y.length;

        // Consider each element `X[i]` of array `X` and ignore the element if it is

        // already in the correct order; otherwise, swap it with the next smaller

        // element, which happens to be the first element of `Y`.

        for (int i = 0; i < m; i++)

        {

            // compare the current element of `X[]` with the first element of `Y[]`

            if (X[i] > Y[0])

            {

                // swap `X[i]` with `Y[0]`

                int temp = X[i];

                X[i] = Y[0];

                Y[0] = temp;

                int first = Y[0];

                // move `Y[0]` to its correct position to maintain the sorted

                // order of `Y[]`. Note: `Y[1…n-1]` is already sorted

                int k;

                for (k = 1; k < n && Y[k] < first; k++) {

                    Y[k - 1] = Y[k];

                }

                Y[k - 1] = first;

            }

        }

    }

TC: O(m\*n)

SC:O(1)



1. Given a schedule containing the arrival and departure time of trains in a station, find the minimum number of platforms needed to avoid delay in any train’s arrival.

Example: Trains arrival = { 2.00, 2.10, 3.00, 3.20, 3.50, 5.00 }

Trains departure = { 2.30, 3.40, 3.20, 4.30, 4.00, 5.20 }



The minimum platforms needed is 2

Solution:

// Function to find the minimum number of platforms needed

    // to avoid delay in any train arrival

    public static int findMinPlatforms(double[] arrival, double[] departure)

    {

        // sort arrival time of trains

        Arrays.sort(arrival);

        // sort departure time of trains

        Arrays.sort(departure);

        // maintains the count of trains

        int count = 0;

        // stores minimum platforms needed

        int platforms = 0;

        // take two indices for arrival and departure time

        int i = 0, j = 0;

        // run till all trains have arrived

        while (i < arrival.length)

        {

            // if a train is scheduled to arrive next

            if (arrival[i] < departure[j])

            {

                // increase the count of trains and update minimum

                // platforms if required

                platforms = Integer.max(platforms, ++count);

                // move the pointer to the next arrival

                i++;

            }

            // if the train is scheduled to depart next i.e.

            // `departure[j] < arrival[i]`, decrease trains' count

            // and move pointer `j` to the next departure.

            // If two trains are arriving and departing simultaneously,

            // i.e., `arrival[i] == departure[j]`, depart the train first

            else {

                count--;

                j++;

            }

        }

        return platforms;

    }

TC: O(nlogn)

SC: O(1)

1. An array contains both positive and negative numbers in random order. Rearrange the array elements so that all negative numbers appear before all positive numbers.

Examples :

Input: -12, 11, -13, -5, 6, -7, 5, -3, -6

Output: -12 -13 -5 -7 -3 -6 11 6 5



Solution:

// Function to shift all the

// negative elements on left side

**static** **void** shiftall(**int**[] arr, **int** left,

**int** right)

{

    // Loop to iterate over the

    // array from left to the right

**while** (left <= right)

    {

        // Condition to check if the left

        // and the right elements are

        // negative

**if** (arr[left] < 0 && arr[right] < 0)

            left++;

        // Condition to check if the left

        // pointer element is positive and

        // the right pointer element is negative

**else** **if** (arr[left] > 0 && arr[right] < 0)

        {

**int** temp = arr[left];

            arr[left] = arr[right];

            arr[right] = temp;

            left++;

            right--;

        }

        // Condition to check if both the

        // elements are positive

**else** **if** (arr[left] > 0 && arr[right] > 0)

            right--;

**else**

        {

            left++;

            right--;

        }

    }

}

TC: O(n)

SC: O(1)

1. Given an array that contains both positive and negative integers, find the product of the maximum product subarray.

Input: arr[] = {6, -3, -10, 0, 2}

Output: 180 // The subarray is {6, -3, -10}

**Solution:**

**static** **int** maxSubarrayProduct(**int** arr[], **int** n)

    {

        // max positive product

        // ending at the current position

**int** max\_ending\_here = arr[0];

        // min negative product ending

        // at the current position

**int** min\_ending\_here = arr[0];

        // Initialize overall max product

**int** max\_so\_far = arr[0];

        // /\* Traverse through the array.

        // the maximum product subarray ending at an index

        // will be the maximum of the element itself,

        // the product of element and max product ending

        // previously and the min product ending previously.

        // \*/

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

**int** temp = Math.max(

                Math.max(arr[i], arr[i] \* max\_ending\_here),

                arr[i] \* min\_ending\_here);

            min\_ending\_here = Math.min(

                Math.min(arr[i], arr[i] \* max\_ending\_here),

                arr[i] \* min\_ending\_here);

            max\_ending\_here = temp;

            max\_so\_far

                = Math.max(max\_so\_far, max\_ending\_here);

        }

**return** max\_so\_far;

    }

**TC: O(n)**

**SC: O(1)**

1. Given an array arr[] of size N. The task is to find the sum of the contiguous subarray within a arr[] with the largest sum.

**A[]=** { -2, -3, 4, -1, -2, 1, 5, -3 };

Maximum contiguous sum is 7

Starting index 2

Ending index 6



**Solution:**

**static** **void** maxSubArraySum(**int** a[], **int** size)

    {



**int** max\_so\_far = Integer.MIN\_VALUE,

            max\_ending\_here = 0, start = 0, end = 0, s = 0;

**for** (**int** i = 0; i < size; i++) {

            max\_ending\_here += a[i];

**if** (max\_so\_far < max\_ending\_here) {

                max\_so\_far = max\_ending\_here;

                start = s;

                end = i;

            }

**if** (max\_ending\_here < 0) {

                max\_ending\_here = 0;



                s = i + 1;

            }

        }

        System.out.println("Maximum contiguous sum is "

                           + max\_so\_far);

        System.out.println("Starting index " + start);

        System.out.println("Ending index " + end);

    }

TC: O(n)

SC: O(1)