**Geeks for Geeks DS Questions:**

**Arrays:-**

**Array Rearrangement**

* **Rearrange an array such that arr[i] = i**

Given an array of elements of length N, ranging from 0 to N – 1. All elements may not be present in the array. If the element is not present then there will be -1 present in the array. Rearrange the array such that A[i] = i and if i is not present, display -1 at that place.

Solution:

***Approach 1 :- (Using HashSet) :***

1. Store all the numbers present in the array into a HashSet
2. Iterate through the length of the array, if the corresponding position element is present in the HashSet, then set A[i] = i, else A[i] = -1

Time Complexity: O(n), Space Complexity: O(n)

***Another Approach (Swap elements in Array) :***

1) Iterate through elements in an array   
2) If arr[i] >= 0 && arr[i] != i, put arr[i] at i ( swap arr[i] with arr[arr[i]])

Time Complexity :- O(n) , Space Complexity: O(1)

* **Write a program to reverse an array or string**

1. *Initialize start and end indexes as start = 0, end = n-1*
2. *In a loop, swap arr[start] with arr[end] and change start and end as follows :   
   start = start +1, end = end – 1*

* **Rearrange array such that arr[i] >= arr[j] if i is even and arr[i]<=arr[j] if i is odd and j < i**

Given an array of n elements. Our task is to write a program to rearrange the array such that elements at even positions are greater than all elements after it and elements at odd positions are greater than all elements before it.

Input : arr[] = {1, 2, 3, 4, 5, 6, 7}

Output : 4 5 3 6 2 7 1

Input : arr[] = {1, 2, 1, 4, 5, 6, 8, 8}

Output : 4 5 2 6 1 8 1 8

Solution :-

The basic idea is to create a copy of the input array, sort it and copy the numbers alternatively from sorted array to original array.

Total number of positions will be n/2 even and n – n/2 odd positions.

1. Copy the array in a new array -> Sort it
2. Start from middle element -1 and go towards left in sorted array and keep copying the element (+2) alternatively 1 chhod ke… and same process karo next half ke jo greater elements hai unke liye right move karke sorted array mai 1 chhod ke even positions mai daal do..

O(nlogn), O(n)

*Code given below :-*

public class GfG{

    // function to rearrange the array

    public static void rearrangeArr(int arr[],

                                        int n)

    {

        // total even positions

        int evenPos = n / 2;

        // total odd positions

        int oddPos = n - evenPos;

        int[] tempArr = new int [n];

        // copy original array in an

        // auxiliary array

        for (int i = 0; i < n; i++)

            tempArr[i] = arr[i];

        // sort the auxiliary array

        Arrays.sort(tempArr);

        int j = oddPos - 1;

        // fill up odd position in

        // original array

        for (int i = 0; i < n; i += 2) {

            arr[i] = tempArr[j];

            j--;

        }

        j = oddPos;

        // fill up even positions in

        // original array

        for (int i = 1; i < n; i += 2) {

            arr[i] = tempArr[j];

            j++;

        }

        // display array

        for (int i = 0; i < n; i++)

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

    }

    // Driver function

    public static void main(String argc[]){

        int[] arr = new int []{ 1, 2, 3, 4, 5,

                                        6, 7 };

        int size = 7;

        rearrangeArr(arr, size); }}

**Rearrange positive and negative numbers in O(n) time and O(1) extra space (Not maintain order of element)**

An array contains both positive and negative numbers in random order. Rearrange the array elements so that positive and negative numbers are placed alternatively. Number of positive and negative numbers need not be equal. If there are more positive numbers they appear at the end of the array. If there are more negative numbers, they too appear in the end of the array.

For example, if the input array is [-1, 2, -3, 4, 5, 6, -7, 8, 9], then the output should be [9, -7, 8, -3, 5, -1, 2, 4, 6]

**Note:** The partition process changes relative order of elements. I.e., the order of the appearance of elements is not maintained with this approach. [See this](https://www.geeksforgeeks.org/rearrange-array-alternating-positive-negative-items-o1-extra-space/) for maintaining order of appearance of elements in this problem.

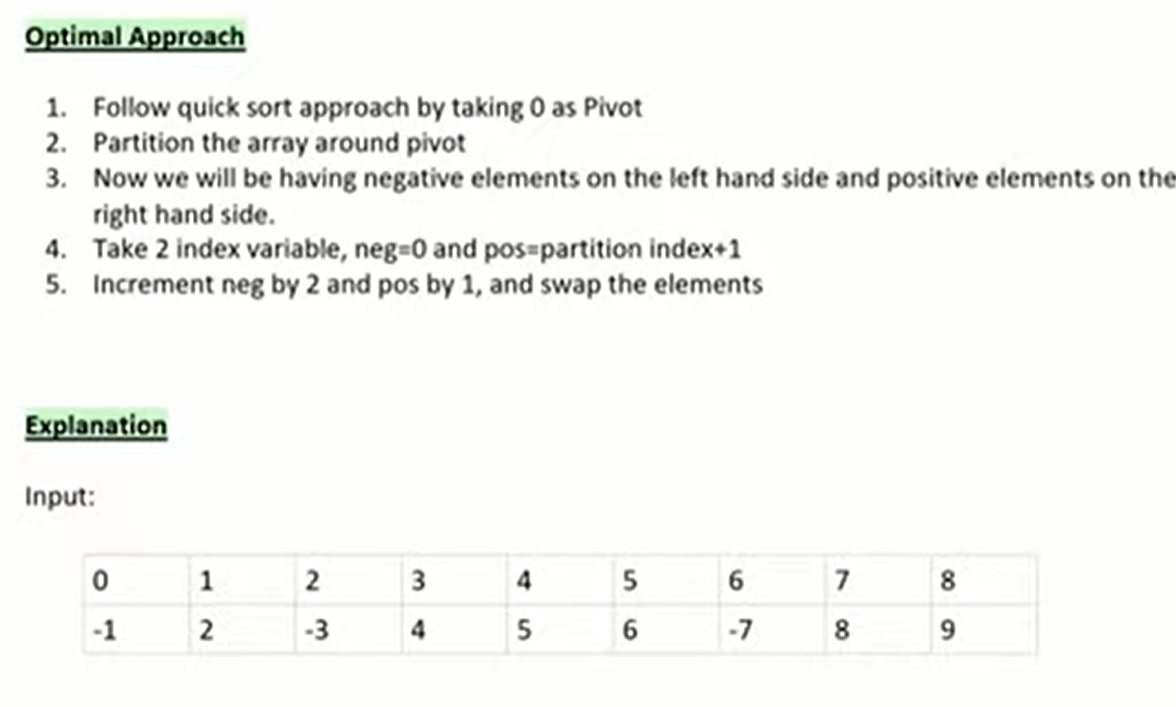
Solution :-

Approach 1 : To solve in O(nlogn) & Space O(n):-

* Sort the Array
* After sorting, use one pointer at beginning of array and one at the end of the array and keep storing in the new output array alternatively until start != high.

Approach 2 :- To solve in O(n) & O(1) :-

The solution is to first separate positive and negative numbers using partition process of QuickSort. In the partition process, consider 0 as value of pivot element so that all negative numbers are placed before positive numbers. Once negative and positive numbers are separated, we start from the first negative number and first positive number, and swap every alternate negative number with next positive number.



Partitioned Array would be :

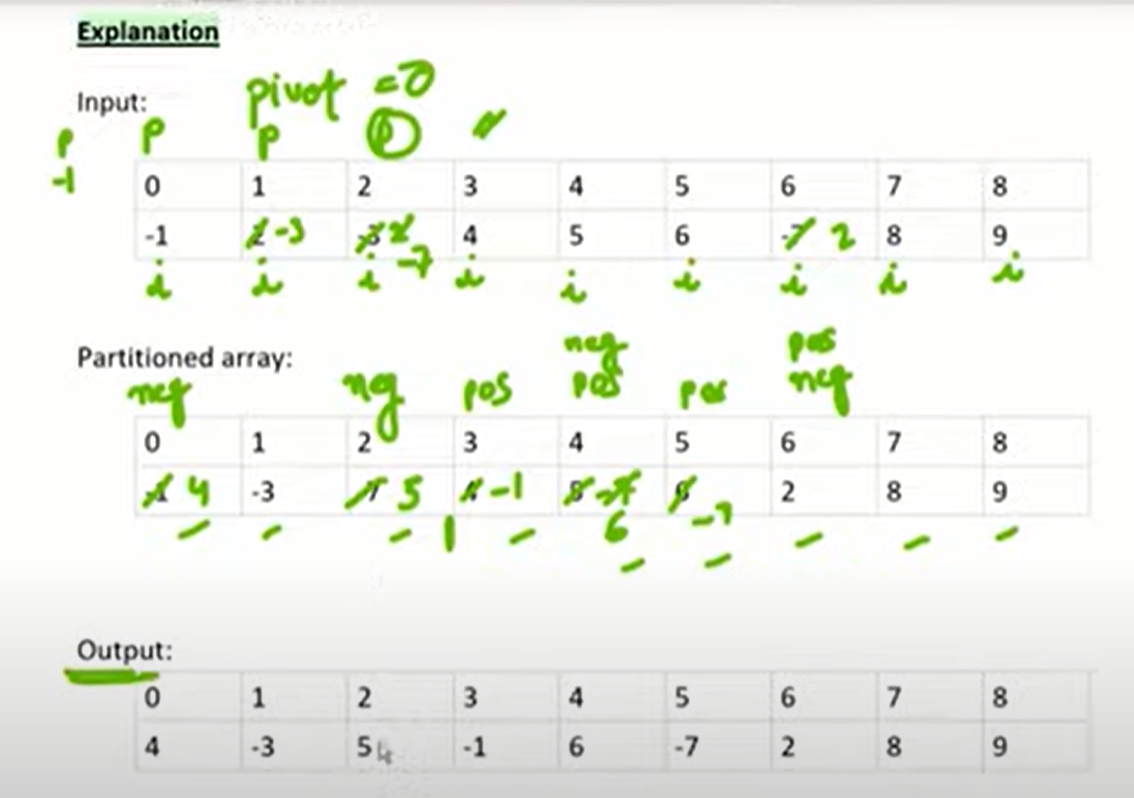
-1 , -3, -7, 4, 5, 6, 2, 8, 9

Logic :

* Initialize Pivot element at index -1.
* Iterate through the existing array from I = 0 to i<n , and for each element check if its <0 i.e, negative.
* If element is negative, Increment the Pivot element index and then swap it with ith element.

After this all process, you will have a partitioned array, where elements on the left side will be negative, and elements on the right side will be all positive.

* Once you have the partitioned array, you can start from index 0 and ‘index of Pivot currently + 1’, and increment 0th index i.e., negative index value by 2 and positive index value by 1, and keep swapping each other until end of the array.



class Alternate {

    // The main function that rearranges elements of given

    // array.  It puts positive elements at even indexes (0,

    // 2, ..) and negative numbers at odd indexes (1, 3, ..).

    static void rearrange(int arr[], int n)

    {

        int i = -1, temp = 0;

        for (int j = 0; j < n; j++)

        {

            if (arr[j] < 0)

            {

                i++;

                temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

        // Now all positive numbers are at end and negative numbers at

        // the beginning of array. Initialize indexes for starting point

        // of positive and negative numbers to be swapped

        int pos = i+1, neg = 0;

        // Increment the negative index by 2 and positive index by 1, i.e.,

        // swap every alternate negative number with next positive number

 while (pos < n && neg < pos && arr[neg] < 0)

        {

            temp = arr[neg];

            arr[neg] = arr[pos];

            arr[pos] = temp;

            pos++;

            neg += 2;

        }

    }

    // A utility function to print an array

    static void printArray(int arr[], int n)

    {

        for (int i = 0; i < n; i++)

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

    }

    /\*Driver function to check for above functions\*/

    public static void main (String[] args)

    {

        int arr[] = {-1, 2, -3, 4, 5, 6, -7, 8, 9};

        int n = arr.length;

        rearrange(arr,n);

        System.out.println("Array after rearranging: ");

        printArray(arr,n);

    }

}

**Move all zeroes to end of array :**

Input : arr[] = {1, 2, 0, 4, 3, 0, 5, 0};

Output : arr[] = {1, 2, 4, 3, 5, 0, 0};

Approach 1:- O(n) and O(1)

Traverse the given array ‘arr’ from left to right. While traversing, maintain count of non-zero elements in array. Let the count be ‘count’. For every non-zero element arr[i], put the element at ‘arr[count]’ and increment ‘count’. After complete traversal, all non- zero elements have already been shifted to front end and ‘count’ is set as index of first 0. Now all we need to do is that run a loop which makes all elements zero from ‘count’ till end of the array.

static void pushZerosToEnd(int arr[], int n)

{

int count = 0; // Count of non-zero elements

// Traverse the array. If element encountered is

// non-zero, then replace the element at index 'count'

// with this element

for (int i = 0; i < n; i++)

if (arr[i] != 0)

arr[count++] = arr[i]; // here count is

// incremented

// Now all non-zero elements have been shifted to

// front and 'count' is set as index of first 0.

// Make all elements 0 from count to end.

while (count < n)

arr[count++] = 0;

}

Approach 2:- O(n) and O(1)

Start from i = 0 and j = n-1;

Iterate from i = 0 till it meets j

Check if i != 0 then increment i

Check if j == 0 then decrement j

Check if i == 0 then swap arr[i] and arr[j] and decrement j.

i = 0, j = n-1;

while (i!=j){

if(a[i] != 0) i++;

if(a[j]==0) j--;

if(a[i]==0 && a[j] !=0)

swap(a[i],a[j])

i++; j--;

}

Approach 3 :- (Only single traversal would be required) O(n)

moveZerosToEnd(arr, n)

Initialize count = 0

for i = 0 to n-1

if (arr[i] != 0) then

swap(arr[count++], arr[i])

Rearrange array such that even positioned are greater than odd

Given an array A of n elements, sort the array according to the following relations :

Input : A[] = {1, 2, 2, 1}

Output : 1 2 1 2 Ex., Position of digit 1 is 0th, and positition of 2 is 1st

Explanation :

For 1st element, 1 1, i = 2 is even.

3rd element, 1 1, i = 4 is even.

**Method 1 –**  
Observe that array consists of [n/2] even positioned elements. If we assign the largest [n/2] elements to the even positions and the rest of the elements to the odd positions, our problem is solved. Because element at the odd position will always be less than the element at the even position as it is the maximum element and vice versa. Sort the array and assign the first [n/2] elements at even positions.  
Below is the implementation of the above approach:

|  |
| --- |
| class GFG {      static void assign(int a[], int n)      {          // Sort the array          Arrays.sort(a);          int ans[] = new int[n];          int p = 0, q = n - 1;          for (int i = 0; i < n; i++) {              // Assign even indexes with maximum elements              if ((i + 1) % 2 == 0)                  ans[i] = a[q--];              // Assign odd indexes with remaining elements              else                  ans[i] = a[p++];          }          // Print result          for (int i = 0; i < n; i++)              System.out.print(ans[i] + " ");      }      // Driver code     public static void main(String args[])      {          int A[] = { 1, 3, 2, 2, 5 };          int n = A.length;          assign(A, n);      }  } |

Output: 

1 5 2 3 2

Time Complexity: O(n \* log n)

Auxiliary Space: O(n)

**Method 2 –  O(n) O(1)**  
One other approach is to traverse the array from the second element and swap the element with the previous one if the condition is not satisfied. This is implemented as follows:

public static void rearrange(int[] arr, int n)

  {

    for(int i = 1; i < n; i++)

    {

      // if index is even

      if(i % 2 == 0)

      {

        if(arr[i] > arr[i - 1])

        {

          // swap two elements

          int temp = arr[i];

          arr[i] = arr[i - 1];

          arr[i - 1] = temp;

        }

      }

      // if index is odd

      else

      {

        if (arr[i] < arr[i - 1])

        {

          // swap two elements

          int temp = arr[i];

          arr[i] = arr[i - 1];

          arr[i - 1] = temp;

        }

      }

    }

    for (int i = 0; i < n; i++)

    {

      System.out.print(arr[i] + " ");

    }

  }

**Rearrange an array in order – smallest, largest, 2nd smallest, 2nd largest**

An **efficient solution** is to use [sorting](https://www.geeksforgeeks.org/sorting-algorithms/).  
1. Sort the elements of array.  
2. Take two variables say i and j and point them to the first and last index of the array respectively.  
3. Now run a loop and store the elements in the array one by one by incrementing i and decrementing j.

**Time Complexity :** O(n Log n)  
**Auxiliary Space :** O(n)

**Double the first element and move zero to end**

Method 1:-

For a given array of n integers and assume that ‘0’ as an invalid number and all other as a valid number. Convert the array in such a way that if both current and next element is valid then double current value and replace the next number with 0. After the modification, rearrange the array such that all 0’s shifted to the end.   
**Examples:**

Input : arr[] = {2, 2, 0, 4, 0, 8}

Output : 4 4 8 0 0 0

Input : arr[] = {0, 2, 2, 2, 0, 6, 6, 0, 0, 8}

Output : 4 2 12 8 0 0 0 0 0 0

static void modifyAndRearrangeArr(int arr[], int n)

    {

        // if 'arr[]' contains a single element

        // only

        if (n == 1)

            return;

        // traverse the array

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

            // if true, perform the required modification

            if ((arr[i] != 0) && (arr[i] == arr[i + 1]))

            {

                // double current index value

                arr[i] = 2 \* arr[i];

                // put 0 in the next index

                arr[i + 1] = 0;

                // increment by 1 so as to move two

                // indexes ahead during loop iteration

                i++;

            }

        }

        // push all the zeros at

        // the end of 'arr[]'

        pushZerosToEnd(arr, n); // Refer Previous code in prev pages…

    }

**Arrange given numbers to form the biggest number**

In the used sorting algorithm, instead of using the default comparison, write a comparison function **myCompare()** and use it to sort numbers.

Given two numbers **X** and **Y**, how should **myCompare()** decide which number to put first – we compare two numbers XY (Y appended at the end of X) and YX (X appended at the end of Y). If **XY** is larger, then X should come before Y in output, else Y should come before. For example, let X and Y be 542 and 60. To compare X and Y, we compare 54260 and 60542. Since 60542 is greater than 54260, we put Y first.

static void printLargest(Vector<String> arr)

    {

        Collections.sort(arr, new Comparator<String>()

        {

            // A comparison function which is used by

            // sort() in printLargest()

            @Override public int compare(String X, String Y)

            {

                // first append Y at the end of X

                String XY = X + Y;

                // then append X at the end of Y

                String YX = Y + X;

                // Now see which of the two

                // formed numbers

                // is greater

                return XY.compareTo(YX) > 0 ? -1 : 1;

            }

        });

        Iterator it = arr.iterator();

        while (it.hasNext())

            System.out.print(it.next());

    }

O(nlogn), O(1)

**Rearrange an array in maximum minimum form | Set 2 (O(1) extra space)**

In this post a solution that requires O(n) time and O(1) extra space is discussed. The idea is to use multiplication and modular trick to store two elements at an index.

A simpler approach will be to observe indexing positioning of maximum elements and minimum elements. The even index stores maximum elements and the odd index stores the minimum elements. With every increasing index, the maximum element decreases by one and the minimum element increases by one. A simple traversal can be done and arr[] can be filled in again.  
**Note:**This approach is only valid when elements of given sorted array are consecutive i.e., vary by one unit.  
Below is the implementation of the above approach:

public static void rearrange(int arr[], int n)

    {

        // initialize index of first minimum and first

        // maximum element

        int max\_ele = arr[n - 1];

        int min\_ele = arr[0];

        // traverse array elements

        for (int i = 0; i < n; i++) {

            // at even index : we have to put maximum element

            if (i % 2 == 0) {

                arr[i] = max\_ele;

                max\_ele -= 1;

            }

            // at odd index : we have to put minimum element

            else {

                arr[i] = min\_ele;

                min\_ele += 1;

            }

        }

    }

**Move all negative numbers to beginning and positive to end with constant extra space**

**Two Pointer Approach:**The idea is to solve this problem with constant space and linear time is by using a [two-pointer](https://www.geeksforgeeks.org/two-pointers-technique/)or two-variable approach where we simply take two variables like left and right which hold the 0 and N-1 indexes. Just need to check that :

1. Check If the left and right pointer elements are negative then simply increment the left pointer.
2. Otherwise, if the left element is positive and the right element is negative then simply swap the elements, and Simultaneously increment or decrement the left and right pointers.
3. Else if the left element is positive and the right element is also positive then simply decrement the right pointer.
4. Repeat the above 3 steps until the left pointer ≤ right pointer.

O(n) O(1)

**Rearrange array such that even index elements are smaller and odd index elements are greater**

An **efficient solution** is to iterate over the array and swap the elements as per the given condition.   
If we have an array of length n, then we iterate from index 0 to n-2 and check the given condition.   
At any point of time if i is even and arr[i] > arr[i+1], then we swap arr[i] and arr[i+1]. Similarly, if i is odd and   
arr[i] < arr[i+1], then we swap arr[i] and arr[i+1].

O(n) and O(1)

**Replace every array element by multiplication of previous and next**

Given an array of integers, update every element with multiplication of previous and next elements with following exceptions.  
a) First element is replaced by multiplication of first and second.  
b) Last element is replaced by multiplication of last and second last.

**Example:**

Input: arr[] = {2, 3, 4, 5, 6}

Output: arr[] = {6, 8, 15, 24, 30}

Approach 1: To create an auxiliary array, copy contents of given array to auxiliary array. Finally traverse the auxiliary array and update given array using copied values. Time complexity of this solution is O(n), but it requires O(n) extra space.

Approach 2: To Keep track of previous element - An efficient solution can solve the problem in O(n) time and O(1) space. The idea is to keep track of previous element in loop.

Use two variables to keep track of previous element and to track the current element.

 static void modify(int arr[], int n)

    {

        // Nothing to do when array size is 1

        if (n <= 1)

            return;

        // store current value of arr[0] and update it

        int prev = arr[0];

        arr[0] = arr[0] \* arr[1];

        // Update rest of the array elements

        for (int i=1; i<n-1; i++)

        {

            // Store current value of next interation

            int curr = arr[i];

            // Update current value using previos value

            arr[i] = prev \* arr[i+1];

            // Update previous value

            prev = curr;

        }

        // Update last array element

        arr[n-1] = prev \* arr[n-1];

    }

**Segregate even and odd numbers | Set 3**

Given an array of integers, segregate even and odd numbers in the array. All the even numbers should be present first, and then the odd numbers.

Solution : We can solve it using 2 pointer approach where one pointer will be pointing to the first element and other pointer will be pointing to the last of the element.

1. Maintain a pointer i = 0 and j = n-1;
2. Traverse the array until i != j, if even number is encountered on left i.e., a[i] then increment i.
3. Check for a[j] as well, if a[j] is odd then decrement j.
4. If a[i] is odd then don’t increment, wait for a[j] to be even after decrement nd then swap with it..so main idea should be that the even numbers are in the beginning and odd numbers at the end.
5. Continue the traversal.

***Time Complexity :****O(n)*  
***Auxiliary Space :****O(1)*

**Positive elements at even and negative at odd positions (Relative order not maintained)**

Input : arr[] = {1, -3, 5, 6, -3, 6, 7, -4, 9, 10}

Output : arr[] = {1, -3, 5, -3, 6, 6, 7, -4, 9, 10}

We take two pointers positive and negative. We set the positive pointer at start of the array and the negative pointer at 1st position of the array.  
We move positive pointer two steps forward till it finds a negative element. Similarly we move negative pointer forward by two places till it finds a positive value at its position.  
If the positive and negative pointers are in the array then we will swap the values at these indexes otherwise we will stop executing the process.

**Maximum sum of pairs with specific difference**

**O(nlogn)**

Given an array of integers and a number k. We can pair two number of the array if the difference between them is strictly less than k. The task is to find the maximum possible sum of disjoint pairs. Sum of P pairs is the sum of all 2P numbers of pairs.

First we sort the given array in increasing order. Once array is sorted, we traverse the array. For every element, we try to pair it with its previous element first. Why do we prefer previous element? Let arr[i] can be paired with arr[i-1] and arr[i-2] (i.e. arr[i] – arr[i-1] < K and arr[i]-arr[i-2] < K). Since the array is sorted, value of arr[i-1] would be more than arr[i-2]. Also, we need to pair with difference less than k, it means if arr[i-2] can be paired, then arr[i-1] can also be paired in a sorted array.   
Now observing the above facts, we can formulate our dynamic programming solution as below,

class GFG {

    // Method to return maximum sum we can get by

    // finding less than K difference pairs

    static int maxSumPairWithDifferenceLessThanK(int arr[],

                                                 int N,

                                                 int k)

    {

        int maxSum = 0;

        // Sort elements to ensure every i and i-1 is

        // closest possible pair

        Arrays.sort(arr);

        // To get maximum possible sum,

        // iterate from largest

        // to smallest, giving larger

        // numbers priority over

        // smaller numbers.

        for (int i = N - 1; i > 0; --i)

        {

            // Case I: Diff of arr[i] and arr[i-1] is less

            // then K, add to maxSum

            // Case II: Diff between arr[i] and arr[i-1] is

            // not less then K, move to next i

            // since with sorting we know, arr[i]-arr[i-1] <

            // arr[i]-arr[i-2] and so on.

            if (arr[i] - arr[i - 1] < k)

            {

                // Assuming only positive numbers.

                maxSum += arr[i];

                maxSum += arr[i - 1];

                // When a match is found skip this pair

                --i;

            }

        }

        return maxSum;

    }

    // Driver code

    public static void main(String[] args)

    {

        int arr[] = { 3, 5, 10, 15, 17, 12, 9 };

        int N = arr.length;

        int K = 4;

        System.out.println(

            maxSumPairWithDifferenceLessThanK(arr, N, K));

    }

}

**Replace every element with the greatest element on right side**

A **tricky method** is to replace all elements using one traversal of the array. The idea is to start from the rightmost element, move to the left side one by one, and keep track of the maximum element. Replace every element with the maximum element.

O(n)

static void nextGreatest(int arr[])

    {

        int size = arr.length;

        int max\_from\_right =  arr[size-1];

        arr[size-1] = -1;

        for (int i = size-2; i >= 0; i--)

        {

            int temp = arr[i];

            arr[i] = max\_from\_right;

            if(max\_from\_right < temp)

            max\_from\_right = temp;

        }

    }

Good Questions below for array rearrangement :-

**Merge k sorted arrays**

***Input:****k = 3, n = 4   
arr[][] = { {1, 3, 5, 7},   
{2, 4, 6, 8},   
{0, 9, 10, 11}} ;****Output:****0 1 2 3 4 5 6 7 8 9 10 11*

*Ref :* [*https://medium.com/outco/how-to-merge-k-sorted-arrays-c35d87aa298e*](https://medium.com/outco/how-to-merge-k-sorted-arrays-c35d87aa298e)

Ideally, if we have 2 arrays, we just need to compare 2 elements and store in the output array, but if we have k arrays, the comparison will be complex.

Other way is to concat all the elements of k arrays, and then sort them which will take the complexity of nk\*log(n\*k).

Basically, on each iteration, if we can find out the minimum element out of k arrays and store in output we can solve this kind of problem easily, here comes the concept of minheap, where can get the minimum element at root and the complexity will be O(1).

**Merge 2 sorted arrays without using extra space**