Sorting Algorithms

[**Learn more about Sorting in DSA Self Paced Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=geeksforgeeks&utm_medium=articles+sorting_lp+header_link_click&utm_campaign=dsa+course+tracker)

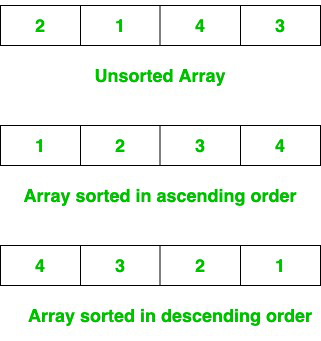
[**Practice Problems on Sorting**](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Sorting&sortBy=submissions&utm_source=geeksforgeeks&utm_medium=articles+sorting_lp+header_link_click&utm_campaign=practice+tracker)

[**Recent articles on Sorting Algorithms**](https://www.geeksforgeeks.org/category/dsa/algorithm/sorting/)

**What is Sorting?**

*A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.*

**For Example:** The below list of characters is sorted in increasing order of their ASCII values. That is, the character with a lesser ASCII value will be placed first than the character with a higher ASCII value.



*Example of Sorting*

**Sorting Algorithms:**

* [Selection Sort](https://www.geeksforgeeks.org/selection-sort/)
* [Bubble Sort](https://www.geeksforgeeks.org/bubble-sort/)
* [Insertion Sort](https://www.geeksforgeeks.org/insertion-sort/)
* [Merge Sort](https://www.geeksforgeeks.org/merge-sort/)
* [Quick Sort](https://www.geeksforgeeks.org/quick-sort/)
* [Heap Sort](https://www.geeksforgeeks.org/heap-sort/)
* [Counting Sort](https://www.geeksforgeeks.org/counting-sort/)
* [Radix Sort](https://www.geeksforgeeks.org/radix-sort/)
* [Bucket Sort](https://www.geeksforgeeks.org/bucket-sort-2/)
* [Bingo Sort Algorithm](https://www.geeksforgeeks.org/bingo-sort-algorithm/)
* [ShellSort](https://www.geeksforgeeks.org/shellsort/)
* [TimSort](https://www.geeksforgeeks.org/timsort/)
* [Comb Sort](https://www.geeksforgeeks.org/comb-sort/)
* [Pigeonhole Sort](https://www.geeksforgeeks.org/pigeonhole-sort/)
* [Cycle Sort](https://www.geeksforgeeks.org/cycle-sort/)
* [Cocktail Sort](https://www.geeksforgeeks.org/cocktail-sort/)
* [Strand Sort](https://www.geeksforgeeks.org/strand-sort/)
* [Bitonic Sort](https://www.geeksforgeeks.org/bitonic-sort/)
* [Pancake sorting](https://www.geeksforgeeks.org/pancake-sorting/)
* [BogoSort or Permutation Sort](https://www.geeksforgeeks.org/bogosort-permutation-sort/)
* [Gnome Sort](https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/)
* [Sleep Sort – The King of Laziness](https://www.geeksforgeeks.org/sleep-sort-king-laziness-sorting-sleeping/)
* [Structure Sorting in C++](https://www.geeksforgeeks.org/structure-sorting-in-c/)
* [Stooge Sort](https://www.geeksforgeeks.org/stooge-sort/)
* [Tag Sort (To get both sorted and original)](https://www.geeksforgeeks.org/tag-sort/)
* [Tree Sort](https://www.geeksforgeeks.org/tree-sort/)
* [Odd-Even Sort / Brick Sort](https://www.geeksforgeeks.org/odd-even-sort-brick-sort/)
* [3-way Merge Sort](https://www.geeksforgeeks.org/3-way-merge-sort/)

**Table of Complexity Comparison:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Name | Best Case | Average Case | Worst Case | Memory | Stable | Method Used |
| [Quick Sort](http://www.geeksforgeeks.org/quick-sort/) |  |  |  |  | No | Partitioning |
| [Merge Sort](http://www.geeksforgeeks.org/merge-sort/) |  |  |  | n | Yes | Merging |
| [Heap Sort](https://www.geeksforgeeks.org/heap-sort/) |  |  |  | 1 | No | Selection |
| [Insertion Sort](http://www.geeksforgeeks.org/insertion-sort/) | n |  |  | 1 | Yes | Insertion |
| [Tim Sort](https://www.geeksforgeeks.org/timsort/) | n |  |  | n | Yes | Insertion & Merging |
| [Selection Sort](http://www.geeksforgeeks.org/selection-sort/) |  |  |  | 1 | No | Selection |
| [Shell Sort](https://www.geeksforgeeks.org/shellsort/) |  |  |  | 1 | No | Insertion |
| [Bubble Sort](http://www.geeksforgeeks.org/bubble-sort/) | n |  |  | 1 | Yes | Exchanging |
| [Tree Sort](https://www.geeksforgeeks.org/tree-sort/) |  |  |  | n | Yes | Insertion |
| [Cycle Sort](https://www.geeksforgeeks.org/cycle-sort/) |  |  |  | 1 | No | Selection |
| [Strand Sort](https://www.geeksforgeeks.org/strand-sort/) | n |  |  | n | Yes | Selection |
| [Cocktail Shaker Sort](https://www.geeksforgeeks.org/cocktail-sort/) | n |  |  | 1 | Yes | Exchanging |
| [Comb Sort](https://www.geeksforgeeks.org/comb-sort/) |  |  |  | 1 | No | Exchanging |
| [Gnome Sort](https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/) | n |  |  | 1 | Yes | Exchanging |
| [Odd–even Sort](https://www.geeksforgeeks.org/odd-even-sort-brick-sort/) | n |  |  | 1 | Yes | Exchanging |

**Library Implementations:**

1. [Introsort – C++’s Sorting Weapon](https://www.geeksforgeeks.org/introsort-cs-sorting-weapon/)
2. [Comparator function of qsort() in C](https://www.geeksforgeeks.org/comparator-function-of-qsort-in-c/)
3. [sort() in C++ STL](https://www.geeksforgeeks.org/sort-c-stl/)
4. [C qsort() vs C++ sort()](https://www.geeksforgeeks.org/c-qsort-vs-c-sort/)
5. [Arrays.sort() in with examples](https://www.geeksforgeeks.org/arrays-sort-in-java-with-examples/)
6. [Collections.sort() in with Examples](https://www.geeksforgeeks.org/collections-sort-java-examples/)

**Some standard problems on Sorting:**

* **Easy:**
  1. [Sort elements by frequency](https://www.geeksforgeeks.org/sort-elements-by-frequency/)
  2. [Sort an array of 0s, 1s and 2s](https://www.geeksforgeeks.org/sort-an-array-of-0s-1s-and-2s/)
  3. [Sort numbers stored on different machines](https://www.geeksforgeeks.org/sort-numbers-stored-on-different-machines/)
  4. [Sort an array in wave form](https://www.geeksforgeeks.org/sort-array-wave-form-2/)
  5. [Check if any two intervals overlap among a given set of intervals](https://www.geeksforgeeks.org/check-if-any-two-intervals-overlap-among-a-given-set-of-intervals/)
  6. [How to sort an array of dates in C/C++?](https://www.geeksforgeeks.org/how-to-sort-an-array-of-dates-in-cc/)
  7. [Sorting Strings using Bubble Sort](https://www.geeksforgeeks.org/sorting-strings-using-bubble-sort-2/)
  8. [Find missing elements of a range](https://www.geeksforgeeks.org/find-missing-elements-of-a-range/)
  9. [Sort an array according to count of set bits](https://www.geeksforgeeks.org/sort-array-according-count-set-bits/)
  10. [Sort even-placed elements in increasing and odd-placed in decreasing order](https://www.geeksforgeeks.org/sort-even-placed-elements-increasing-odd-placed-decreasing-order/)
  11. [Sort an array when two halves are sorted](https://www.geeksforgeeks.org/sort-array-two-halves-sorted/)
  12. [Sorting Big Integers](https://www.geeksforgeeks.org/sorting-big-integers/)
  13. [Sort a linked list of 0s, 1s and 2s](https://www.geeksforgeeks.org/sort-a-linked-list-of-0s-1s-or-2s/)
* **Medium:**
  1. [Inversion count in Array using Merge Sort](https://www.geeksforgeeks.org/inversion-count-in-array-using-merge-sort/https:/www.geeksforgeeks.org/inversion-count-in-array-using-merge-sort/)
  2. [Find the Minimum length Unsorted Subarray, sorting which makes the complete array sorted](https://www.geeksforgeeks.org/minimum-length-unsorted-subarray-sorting-which-makes-the-complete-array-sorted/)
  3. [Sort a nearly sorted (or K sorted) array](https://www.geeksforgeeks.org/nearly-sorted-algorithm/)
  4. [Sort n numbers in range from 0 to n^2 – 1 in linear time](https://www.geeksforgeeks.org/sort-n-numbers-range-0-n2-1-linear-time/)
  5. [Sort an array according to the order defined by another array](https://www.geeksforgeeks.org/sort-array-according-order-defined-another-array/)
  6. [Find the point where maximum intervals overlap](https://www.geeksforgeeks.org/find-the-point-where-maximum-intervals-overlap/)
  7. [Find a permutation that causes worst case of Merge Sort](https://www.geeksforgeeks.org/find-a-permutation-that-causes-worst-case-of-merge-sort/)
  8. [Sort Vector of Pairs in ascending order in C++](https://www.geeksforgeeks.org/sort-vector-of-pairs-in-ascending-order-in-c/)
  9. [Minimum swaps to make two arrays identical](https://www.geeksforgeeks.org/minimum-swaps-to-make-two-array-identical/)
  10. [Chocolate Distribution Problem](https://www.geeksforgeeks.org/chocolate-distribution-problem/)
  11. [Permute two arrays such that sum of every pair is greater or equal to K](https://www.geeksforgeeks.org/permute-two-arrays-sum-every-pair-greater-equal-k/)
  12. [Bucket Sort To Sort an Array with Negative Numbers](https://www.geeksforgeeks.org/bucket-sort-to-sort-an-array-with-negative-numbers/)
  13. [Sort a Matrix in all way increasing order](https://www.geeksforgeeks.org/sort-matrix-way-increasing-order/)
  14. [Convert an Array to reduced form using Vector of pairs](https://www.geeksforgeeks.org/convert-an-array-to-reduced-form-using-vector-of-pairs/)
  15. [Smallest Difference Triplet from Three arrays](https://www.geeksforgeeks.org/smallest-difference-triplet-from-three-arrays/)
  16. [Check if it is possible to sort an array with conditional swapping of adjacent allowed](https://www.geeksforgeeks.org/check-possible-sort-array-conditional-swapping-adjacent-allowed/)
* **Hard:**
  1. [Find Surpasser Count of each element in array](https://www.geeksforgeeks.org/find-surpasser-count-of-each-element-in-array/)
  2. [Count distinct occurrences as a subsequence](https://www.geeksforgeeks.org/count-distinct-occurrences-as-a-subsequence/)
  3. [Count minimum number of subsets (or subsequences) with consecutive numbers](https://www.geeksforgeeks.org/count-minimum-number-subsets-subsequences-consecutive-numbers/)
  4. [Chose k array elements such that difference of maximum and minimum is minimized](https://www.geeksforgeeks.org/k-numbers-difference-maximum-minimum-k-number-minimized/)
  5. [Minimum swap required to convert binary tree to binary search tree](https://www.geeksforgeeks.org/minimum-swap-required-convert-binary-tree-binary-search-tree/)
  6. [K-th smallest element after removing some integers from natural numbers](https://www.geeksforgeeks.org/k-th-smallest-element-removing-integers-natural-numbers/)
  7. [Maximum difference between frequency of two elements such that element having greater frequency is also greater](https://www.geeksforgeeks.org/maximum-difference-between-frequency-of-two-elements-such-that-element-having-greater-frequency-is-also-greater/)
  8. [Minimum swaps to reach permuted array with at most 2 positions left swaps allowed](https://www.geeksforgeeks.org/minimum-swaps-reach-permuted-array-2-positions-left-swaps-allowed/)
  9. [Find whether it is possible to make array elements same using one external number](https://www.geeksforgeeks.org/find-whether-possible-make-array-elements-using-one-external-number/)
  10. [Sort an array after applying the given equation](https://www.geeksforgeeks.org/sort-array-applying-given-equation/)
  11. [Print array of strings in sorted order without copying one string into another](https://www.geeksforgeeks.org/print-array-strings-sorted-order-without-copying-one-string-another/)

**Quick Links :**

1. [‘Practice Problems’ on Sorting](https://practice.geeksforgeeks.org/topics/Sorting/)
2. [‘Quizzes’ on Sorting](https://www.geeksforgeeks.org/algorithms-gq/searching-and-sorting-gq/)

**Recomended:**

* [**Learn Data Structure and Algorithms | DSA Tutorial**](https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial?utm_source=Website&utm_medium=Landing+Page+Click&utm_campaign=DSA+Page+Tracker&utm_id=DSA-Page-Tracker&utm_term=DSA+Page+Promo&utm_content=Course+Page)

----------------------------------------------------------------------------------------------------------------------------------------------

**Easy Questions:**

**1.Sort elements by frequency**

Print the elements of an array in the decreasing frequency if 2 numbers have the same frequency then print the one which came first

**Examples:**

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

***Output:****arr[] = {8, 8, 8, 2, 2, 5, 5, 6}*

***Input:****arr[] = {2, 5, 2, 6, -1, 9999999, 5, 8, 8, 8}*

***Output:****arr[] = {8, 8, 8, 2, 2, 5, 5, 6, -1, 9999999}*

[Recommended: Please solve it on “***PRACTICE*** ” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/sorting-elements-of-an-array-by-frequency/0)

**2.Sort elements by frequency using**[sorting](https://www.geeksforgeeks.org/sorting-algorithms/)**:**

Follow the given steps to solve the problem:

* Use a sorting algorithm to sort the elements
* Iterate the sorted array and construct a 2D array of elements and count
* Sort the 2D array according to the count

Below is the illustration of the above approach:

***Input:****arr[] = {2 5 2 8 5 6 8 8}*

***Step1:****Sort the array,*

*After sorting we get: 2 2 5 5 6 8 8 8*

***Step 2:****Now construct the 2D array to maintain the count of every element as*

*{{2, 2}, {5, 2}, { 6, 1}, {8, 3}}*

***Step 3:****Sort the array by count*

*{{8, 3}, {2, 2}, {5, 2}, {6, 1}}*

**3.How to maintain the order of elements if the frequency is the same?**

The above approach doesn’t make sure the order of elements remains the same if the frequency is the same. To handle this, we should use indexes in step 3, if two counts are the same then we should first process(or print) the element with a lower index. In step 1, we should store the indexes instead of elements.

***Input:****arr[] = {2 5 2 8 5 6 8 8}*

***Step1:****Sort the array,*

*After sorting we get: 2 2 5 5 6 8 8 8*

*indexes:                    0 2 1 4 5 3 6 7*

***Step 2:****Now construct the 2D array to maintain the count of every element as*

*Index, Count*

*0,      2*

*1,      2*

*5,      1*

*3,      3*

***Step 3:****Sort the array by count (consider indexes in case of tie)*

*{{3, 3}, {0, 2}, {1, 2}, {5, 1}}*

*Print the elements using indexes in the above 2D array*

Below is the implementation of the above approach:

# Python3 program that performs the following

# operations: Sort elements by frequency. If two elements

# have same count, then put the elements that appears first

# Used for sorting

**class** ele:

**def** \_\_init\_\_(self):

        self.count **=** 0

        self.index **=** 0

        self.val **=** 0

**def** mycomp(a):

**return** a.val

# Used for sorting by frequency. And if frequency is same,

# then by appearance

**def** mycomp2(a):

    # using negative value for a.index

    # since the sorting should be in

    # descending order

**return** (a.count, **-**a.index)

**def** sortByFrequency(arr, n):

    element **=** [None **for** \_ **in** range(n)]

**for** i **in** range(n):

        element[i] **=** ele()

        # Fill Indexes

        element[i].index **=** i

        # Initialize counts as 0

        element[i].count **=** 0

        # Fill values in structure

        # elements

        element[i].val **=** arr[i]

    # Sort the structure elements according to value,

    # we used stable sort so relative order is maintained.

    #

    element.sort(key**=**mycomp)

    # initialize count of first element as 1

    element[0].count **=** 1

    # Count occurrences of remaining elements

**for** i **in** range(1, n):

**if** (element[i].val **==** element[i **-** 1].val):

            element[i].count **+=** element[i **-** 1].count **+** 1

            # Set count of previous element as -1, we are

            #  doing this because we'll again sort on the

            #  basis of counts (if counts are equal than on

            # the basis of index)\*/

            element[i **-** 1].count **= -**1

            # Retain the first index (Remember first index

            #  is always present in the first duplicate we

            #  used stable sort. \*/

            element[i].index **=** element[i **-** 1].index

        # Else If previous element is not equal to current

        #  so set the count to 1

**else**:

            element[i].count **=** 1

    # Now we have counts and first index for each element

    # so now sort on the basis of count and in case of tie

    # use index to sort.\*/

    element.sort(key**=**mycomp2)

    index **=** 0

**for** i **in** range(n **-** 1, **-**1, **-**1):

**if** (element[i].count !**= -**1):

**for** j **in** range(element[i].count):

                arr[index] **=** element[i].val

                index **+=** 1

# Driver code

arr **=** [2, 5, 2, 6, **-**1, 9999999, 5, 8, 8, 8]

n **=** len(arr)

# Function call

sortByFrequency(arr, n)

**print**(**\***arr)

# This code is contributed by phasing17

**Output**

8 8 8 2 2 5 5 6 -1 9999999

**Time Complexity:**O(N log N), where the N is the size of the array

**Auxiliary Space:**O(N)

**4.Sort elements by frequency using**[hashing](https://www.geeksforgeeks.org/hashing-set-1-introduction/)**and sorting:**

To solve the problem follow the below idea:

*Using a hashing mechanism, we can store the elements (also the first index) and their counts in a hash. Finally, sort the hash elements according to their counts*

Below is the implementation of the above approach:

# Python3 program for above approach

**from** collections **import** defaultdict

# Sort by Frequency

**def** sortByFreq(arr, n):

    # arr -> Array to be sorted

    # n   -> Length of Array

    # d is a hashmap(referred as dictionary in python)

    d **=** defaultdict(**lambda**: 0)

**for** i **in** range(n):

        d[arr[i]] **+=** 1

    # Sorting the array 'arr' where key

    # is the function based on which

    # the array is sorted

    # While sorting we want to give

    # first priority to Frequency

    # Then to value of item

    arr.sort(key**=lambda** x: (**-**d[x], x), reverse **=** True)

    #require Updation:- reverse = True, to sort an array in descending order (Jayesh Verma)

**return** (arr)

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    arr **=** [2, 5, 2, 6, **-**1, 9999999, 5, 8, 8, 8]

    n **=** len(arr)

    # Function call

    solution **=** sortByFreq(arr, n)

    print(**\***solution)

**Output**

8 8 8 2 2 5 5 6 -1 9999999

**Time Complexity:**O(N log N), where the N is the size of the array

**Auxiliary Space:**O(N)

**Note:**This can also be solved by Using two maps, one for array element as an index and after this second map whose keys are frequency and value are array elements.

**5.Sort elements by frequency using**[BST](https://www.geeksforgeeks.org/binary-search-tree-data-structure/)**:**

Follow the given steps to solve the problem:

* Insert elements in BST one by one and if an element is already present then increment the count of the node. The node of the Binary Search Tree (used in this approach) will be as follows.

**struct** tree {

**int** element;

**int** first\_index /\*To handle ties in counts\*/

**int** count;

} BST;

* Store the first indexes and corresponding counts of BST in a 2D array.
* Sort the 2D array according to counts (and use indexes in case of a tie).

Implementation of the this approach: [Set 2](https://www.geeksforgeeks.org/sort-elements-by-frequency-set-2/)

**Time Complexity:** O(N log N) if a [Self Balancing Binary Search Tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) is used.

**Auxiliary Space:**O(N)

**6.Sort elements by frequency using**[Heap](https://www.geeksforgeeks.org/heap-data-structure/)**:**

Follow the given steps to solve the problem:

* Take the arr and use unordered\_map to have VALUE : FREQUENCY Table
* Then make a HEAP such that high frequency remains at TOP and when frequency is same, just keep in ascending order (Smaller at TOP)
* Then After full insertion into Heap
* Pop one by one and copy it into the Array

Below is the implementation of the above approach:

**from** collections **import** defaultdict

**from** queue **import** PriorityQueue

**class** Compare:

**def** \_\_init\_\_(self, freq, val):

        self.freq **=** freq

        self.val **=** val

**def** \_\_lt\_\_(self, other):

**if** self.freq **==** other.freq:

**return** self.val < other.val

**return** self.freq > other.freq

**def** solve(arr):

    n **=** len(arr)

    mpp **=** defaultdict(int)

**for** a **in** arr:

        mpp[a] **+=** 1

    max\_heap **=** PriorityQueue()

**for** key, value **in** mpp.items():

        max\_heap.put(Compare(value, key))

    i **=** 0

**while not** max\_heap.empty():

        item **=** max\_heap.get()

        freq **=** item.freq

        val **=** item.val

**for** \_ **in** range(freq):

            arr[i] **=** val

            i **+=** 1

**return** arr

vec **=** [2, 5, 2, 8, 5, 6, 8, 8]

print(solve(vec))

**Output**

8 8 8 2 2 5 5 6

**Time Complexity:**O(d \* log(d))(Dominating factor O(n + 2 \* d \* log (d))). O(n) (unordered map insertion- as 1 insertion takes O(1)) + O(d\*log(d)) (Heap insertion – as one insertion is log N complexity) + O(d\*log(d)) (Heap Deletion – as one pop takes Log N complexity)

Here d = No. of Distinct Elements, n = Total no. of elements (size of input array). (Always d<=n  depends on the array)

**Auxiliary Space:**O(d),As heap and map is used

**7.Sort an array of 0s, 1s and 2s | Dutch National Flag problem**

Given an array **A[]** consisting of only **0s**, **1s,** and **2s**. The task is to write a function that sorts the given array. The functions should put all 0s first, then all 1s and all 2s in last.

This problem is also the same as the famous **“Dutch National Flag problem”**. The problem was proposed by Edsger Dijkstra. The problem is as follows:

*Given N balls of colour red, white or blue arranged in a line in random order. You have to arrange all the balls such that the balls with the same colours are adjacent with the order of the balls, with the order of the colours being red, white and blue (i.e., all red coloured balls come first then the white coloured balls and then the blue coloured balls).*

**Examples:**

***Input****: {0, 1, 2, 0, 1, 2}*

***Output****: {0, 0, 1, 1, 2, 2}*

***Input****: {0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1}*

***Output****: {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2}*

Recommended Problem

Sort an array of 0s, 1s and 2s

[Solve Problem](https://practice.geeksforgeeks.org/problems/sort-an-array-of-0s-1s-and-2s4231/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

**8.Sort an array of 0s, 1s, and 2s using the Pointer Approach:**

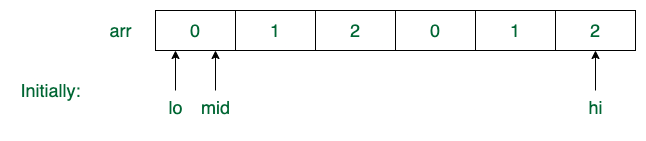
This approach is based on the following idea:

* *The problem is similar to*[*“Segregate 0s and 1s in an array”*](https://www.geeksforgeeks.org/segregate-0s-and-1s-in-an-array-by-traversing-array-once/)*.*
* *The problem was posed with three colors, here `0′, `1′ and `2′. The array is divided into four sections:*
* *arr[1] to arr[low – 1]*
* *arr[low] to arr[mid – 1]*
* *arr[mid] to arr[high – 1]*
* *arr[high] to arr[n]*
* *If the ith element is 0 then swap the element to the low range.*
* *Similarly, if the element is 1 then keep it as it is.*
* *If the element is 2 then swap it with an element in high range.*

**Illustration:**

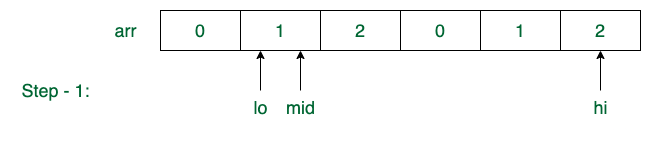
***arr[] = {0, 1, 2, 0, 1, 2}***

***lo****= 0,****mid****= 0,****hi****= 5*



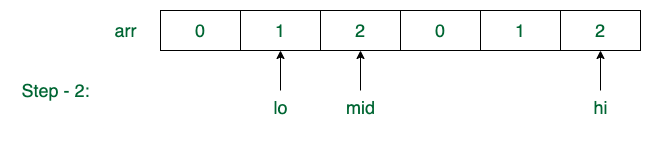
***Step – 1:****arr[mid] == 0*

* *swap(arr[lo], arr[mid])*
* *lo = lo + 1 = 1*
* *mid = mid + 1 = 1*
* *arr[] = {0, 1, 2, 0, 1, 2}*



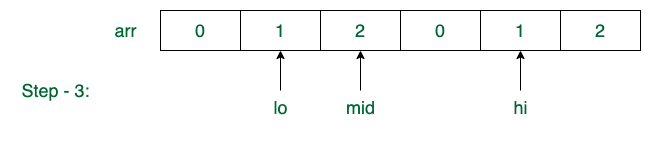
***Step – 2:****arr[mid] == 1*

* *mid = mid + 1 = 2*
* *arr[] = {0, 1, 2, 0, 1, 2}*



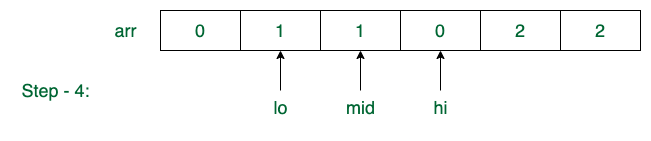
***Step – 3:****arr[mid] == 2*

* *swap(arr[mid], arr[hi])*
* *hi = hi – 1 = 4*
* *arr[] = {0, 1, 2, 0, 1, 2}*



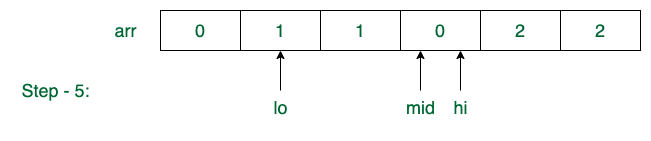
***Step – 4:****arr[mid] == 2*

* *swap(arr[mid], arr[hi])*
* *hi = hi – 1 = 3*
* *arr[] = {0, 1, 1, 0, 2, 2}*



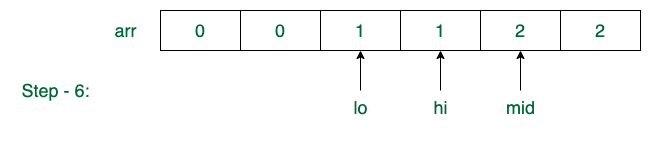
***Step – 5:****arr[mid] == 1*

* *mid = mid + 1 = 3*
* *arr[] = {0, 1, 1, 0, 2, 2}*



***Step – 6:****arr[mid] == 0*

* *swap(arr[lo], arr[mid])*
* *lo = lo + 1 = 2*
* *mid = mid + 1 = 4*
* *arr[] = {0, 0, 1, 1, 2, 2}*



*Hence,****arr[] = {0, 0, 1, 1, 2, 2}***

Follow the steps below to solve the given problem:

* Keep three indices low = 1, mid = 1, and high = N and there are four ranges, 1 to low (the range containing 0), low to mid (the range containing 1), mid to high (the range containing unknown elements) and high to N (the range containing 2).
* Traverse the array from start to end and mid is less than high. (Loop counter is i)
* If the element is 0 then swap the element with the element at index low and update low = low + 1 and mid = mid + 1
* If the element is 1 then update mid = mid + 1
* If the element is 2 then swap the element with the element at index high and update high = high – 1 and update i = i – 1. As the swapped element is not processed
* Print the array.

# Python program to sort an array with

# 0, 1 and 2 in a single pass

# Function to sort array

**def** sort012(a, arr\_size):

    lo **=** 0

    hi **=** arr\_size **-** 1

    mid **=** 0

    # Iterate till all the elements

    # are sorted

**while** mid <**=** hi:

        # If the element is 0

**if** a[mid] **==** 0:

            a[lo], a[mid] **=** a[mid], a[lo]

            lo **=** lo **+** 1

            mid **=** mid **+** 1

        # If the element is 1

**elif** a[mid] **==** 1:

            mid **=** mid **+** 1

        # If the element is 2

**else**:

            a[mid], a[hi] **=** a[hi], a[mid]

            hi **=** hi **-** 1

**return** a

# Function to print array

**def** printArray(a):

**for** k **in** a:

        print(k, end**=**' ')

# Driver Program

arr **=** [0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1]

arr\_size **=** len(arr)

arr **=** sort012(arr, arr\_size)

printArray(arr)

# Contributed by Harshit Agrawal

**Output**

0 0 0 0 0 1 1 1 1 1 2 2

**Time Complexity:** O(n), Only one traversal of the array is needed.

**Space Complexity:** O(1), No extra space is required.

**9.Sort an array of 0s, 1s, and 2s using Counting Approach:**

This approach is based on the following idea:

*Count the number of 0s, 1s, and 2s in the given array. Then store all the 0s at the beginning followed by all the 1s and then all the 2s.*

**Illustration:**

***arr[] = {0, 1, 2, 0, 1, 2}***

*cnt0 = 0, cnt1 = 0, cnt2 = 0*

*At i = 0: arr[0] == 0*

* *cnt0 = cnt0 + 1 = 1*

*At i = 1: arr[1] == 1*

* *cnt1 = cnt1 + 1 = 1*

*At i = 2: arr[2] == 2*

* *cnt2 = cnt2 + 1 = 1*

*At i = 3: arr[3] == 0*

* *cnt0 = cnt0 + 1 = 2*

*At i = 4: arr[4] == 1*

* *cnt1 = cnt1 + 1 = 2*

*At i = 5: arr[5] == 2*

* *cnt2 = cnt2 + 1 = 2*

*Replace****cnt0****number of elements with****0****in****arr***

* ***arr[] = {0, 0, 2, 0, 1, 2}***

*Replace****cnt1****number of elements with****1****in****arr***

* ***arr[] = {0, 0, 1, 1, 1, 2}***

*Replace****cnt2****number of elements with****2****in****arr***

* ***arr[] = {0, 0, 1, 1, 2, 2}***

*Hence, arr[] =****{0, 0, 1, 1, 2, 2}***

Follow the steps below to solve the given problem:

* Keep three counters c0 to count 0s, c1 to count 1s, and c2 to count 2s
* Traverse through the array and increase the count of c0 if the element is 0, increase the count of c1 if the element is 1 and increase the count of c2 if the element is 2
* Now again traverse the array and replace the first c0 elements with 0, the next c1 elements with 1, and the next c2 elements with 2.

Below is the implementation of the above idea :

# Python implementation of the approach

# Utility function to print contents of an array

**def** printArr(arr, n):

**for** i **in** range(n):

        print(arr[i], end**=**" ")

# Function to sort the array of 0s, 1s and 2s

**def** sortArr(arr, n):

    cnt0 **=** 0

    cnt1 **=** 0

    cnt2 **=** 0

    # Count the number of 0s, 1s and 2s in the array

**for** i **in** range(n):

**if** arr[i] **==** 0:

            cnt0 **+=** 1

**elif** arr[i] **==** 1:

            cnt1 **+=** 1

**elif** arr[i] **==** 2:

            cnt2 **+=** 1

    # Update the array

    i **=** 0

    # Store all the 0s in the beginning

**while** (cnt0 > 0):

        arr[i] **=** 0

        i **+=** 1

        cnt0 **-=** 1

    # Then all the 1s

**while** (cnt1 > 0):

        arr[i] **=** 1

        i **+=** 1

        cnt1 **-=** 1

    # Finally all the 2s

**while** (cnt2 > 0):

        arr[i] **=** 2

        i **+=** 1

        cnt2 **-=** 1

    # Print the sorted array

    printArr(arr, n)

# Driver code

arr **=** [0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1]

n **=** len(arr)

sortArr(arr, n)

# This code is contributed by shubhamsingh10

**Output**

0 0 0 0 0 1 1 1 1 1 2 2

**Time Complexity:** O(n), Only nonnested traversals of the array are needed.

**Space Complexity:** O(1)

**10.Sort numbers stored on different machines**

Given N machines. Each machine contains some numbers in sorted form. But the amount of numbers, each machine has is not fixed. Output the numbers from all the machine in sorted non-decreasing form.

**Example:**

Machine M1 contains 3 numbers: {30, 40, 50}  
Machine M2 contains 2 numbers: {35, 45}   
Machine M3 contains 5 numbers: {10, 60, 70, 80, 100}

Output: {10, 30, 35, 40, 45, 50, 60, 70, 80, 100}

Representation of stream of numbers on each machine is considered as a linked list. A Min Heap can be used to print all numbers in sorted order. Following is the detailed process

1. Store the head pointers of the linked lists in a minHeap of size N where N is a number of machines.
2. Extract the minimum item from the minHeap. Update the minHeap by replacing the head of the minHeap with the next number from the linked list or by replacing the head of the minHeap with the last number in the minHeap followed by decreasing the size of heap by 1.
3. Repeat the above step 2 until heap is not empty. Below is C++ implementation of the above approach.

**Implementation:**

# A program to take numbers from different machines and print them in sorted order

# A Linked List node

**class** ListNode:

**def** \_\_init\_\_(self, val**=**0, next**=**None):

        self.val **=** val

        self.next **=** next

# A Min Heao (Collection of Min Heap nodes)

**class** MinHeap:

**def** \_\_init\_\_(self, capacity):

        self.count **=** 0

        self.capacity **=** capacity

        self.array **=** []

# A utility function to insert a new node at the beginning of linked list

**def** push(head\_ref, new\_data):

    new\_node **=** ListNode(new\_data)

    new\_node.next **=** head\_ref

    head\_ref **=** new\_node

**return** head\_ref

# A utility function to swap two min heap nodes. This function

# is needed in minHeapify

**def** swap(a, b):

    temp **=** a

    a **=** b

    b **=** temp

**return** a, b

# The standard minHeapify function.

**def** min\_heapify(min\_heap, idx):

    left **=** 2 **\*** idx **+** 1

    right **=** 2 **\*** idx **+** 2

    smallest **=** idx

**if** (left < min\_heap.count **and**

            min\_heap.array[left][0].val < min\_heap.array[smallest][0].val):

        smallest **=** left

**if** (right < min\_heap.count **and**

            min\_heap.array[right][0].val < min\_heap.array[smallest][0].val):

        smallest **=** right

**if** smallest !**=** idx:

        min\_heap.array[smallest], min\_heap.array[idx] **=** swap(

            min\_heap.array[smallest], min\_heap.array[idx])

        min\_heapify(min\_heap, smallest)

# A utility function to check whether a Min Heap is empty or not

**def** is\_empty(min\_heap):

**return** min\_heap.count **==** 0

# A standard function to build a heap

**def** build\_min\_heap(min\_heap):

    n **=** min\_heap.count **-** 1

**for** i **in** range((n **-** 1) **//** 2, **-**1, **-**1):

        min\_heapify(min\_heap, i)

# This function inserts array elements to heap and then calls

# buildHeap for heap property among nodes

**def** populate\_min\_heap(min\_heap, array, n):

**for** i **in** range(n):

        min\_heap.array.append((array[i], i))

        min\_heap.count **+=** 1

    build\_min\_heap(min\_heap)

# Return minimum element from all linked lists

**def** extract\_min(min\_heap):

**if** is\_empty(min\_heap):

**return** None

    temp **=** min\_heap.array[0][0]

**if** temp.next **is not** None:

        min\_heap.array[0] **=** (temp.next, min\_heap.array[0][1])

**else**:

        min\_heap.array[0] **=** min\_heap.array[min\_heap.count **-** 1]

        min\_heap.count **-=** 1

    min\_heapify(min\_heap, 0)

**return** temp

# The main function that takes an array of lists from N machines

# and generates the sorted output

**def** external\_sort(array, n):

     # Create a min heap of size equal to number of machines

    min\_heap **=** MinHeap(n)

    populate\_min\_heap(min\_heap, array, n)

**while not** is\_empty(min\_heap):

        temp **=** extract\_min(min\_heap)

        print(temp.val, end**=**" ")

# Driver program to test above functions

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    N **=** 3  # Number of machines

    array **=** [None] **\*** N

    # an array of pointers storing the head nodes of the linked lists

    array[0] **=** None

    array[0] **=** push(array[0], 50)

    array[0] **=** push(array[0], 40)

    array[0] **=** push(array[0], 30)

    # Create a Linked List 35->45 for second machine

    array[1] **=** None

    array[1] **=** push(array[1], 45)

    array[1] **=** push(array[1], 35)

    # Create Linked List 10->60->70->80 for third machine

    array[2] **=** None

    array[2] **=** push(array[2], 100)

    array[2] **=** push(array[2], 80)

    array[2] **=** push(array[2], 70)

    array[2] **=** push(array[2], 60)

    array[2] **=** push(array[2], 10)

    external\_sort(array, N)

**Output**

10 30 35 40 45 50 60 70 80 100

**Time complexity**: O(N) for min heap

**Auxiliary Space:**O(N)

**11.Sort an array in wave form**

Given an unsorted array of integers, sort the array into a wave array. An array **arr[0..n-1]** is sorted in wave form if:

**arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= …..**

**Examples:**

***Input:****arr[] = {10, 5, 6, 3, 2, 20, 100, 80}*

***Output:****arr[] = {10, 5, 6, 2, 20, 3, 100, 80}*

***Explanation:***

*here you can see {10, 5, 6, 2, 20, 3, 100, 80} first element is larger than the second and the same thing is repeated again and again. large element – small element-large element -small element and so on .it can be small element-larger element – small element-large element -small element too. all you need to maintain is the up-down fashion which represents a wave. there can be multiple answers.*

***Input:****arr[] = {20, 10, 8, 6, 4, 2}*

***Output:****arr[] = {20, 8, 10, 4, 6, 2}*

[Wave Array](https://practice.geeksforgeeks.org/problems/wave-array-1587115621/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/wave-array-1587115621/1/)

**What is a wave array?**

well, you have seen waves right? how do they look? if you will form a graph of them it would be some in some up-down fashion. that is what you have to do here, you are supposed to arrange numbers in such a way that if we will form a graph it will be in an up-down fashion rather than a straight line.

**Wave Array using sorting**

*A idea is to use sorting. First sort the input array, then swap all adjacent elements.*

Follow the steps mentioned below to implement the idea:

* Sort the array.
* Traverse the array from index**0** to **N-1,** and increase the value of the index by **2**.
* While traversing the array swap **arr[i]** with **arr[i+1].**
* Print the final array.

Below is the implementation of the above approach:

# Python function to sort the array arr[0..n-1] in wave form,

# i.e., arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]

**def** sortInWave(arr, n):

    #sort the array

    arr.sort()

    # Swap adjacent elements

**for** i **in** range(0,n**-**1,2):

        arr[i], arr[i**+**1] **=** arr[i**+**1], arr[i]

# Driver program

arr **=** [10, 90, 49, 2, 1, 5, 23]

sortInWave(arr, len(arr))

**for** i **in** range(0,len(arr)):

    print (arr[i],end**=**" ")

# This code is contributed by \_\_Devesh Agrawal\_\_

**Output**

2 1 10 5 49 23 90

**Time Complexity:**O(N\*log(N))

**Auxiliary Space:**O(1)

**Wave Array Optimized Approach**

*The idea is based on the fact that if we make sure that all even positioned (at index 0, 2, 4, ..) elements are greater than their adjacent odd elements, we don’t need to worry about oddly positioned elements.*

Follow the steps mentioned below to implement the idea:

* Traverse all even positioned elements of the input array, and do the following.
* If the current element is smaller than the previous odd element, swap the previous and current.
* If the current element is smaller than the next odd element, swap next and current.

Below is the implementation of the above approach:

# Python function to sort the array arr[0..n-1] in wave form,

# i.e., arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]

**def** sortInWave(arr, n):

    # Traverse all even elements

**for** i **in** range(0, n **-** 1, 2):

        # If current even element is smaller than previous

**if** (i > 0 **and** arr[i] < arr[i**-**1]):

            arr[i], arr[i**-**1] **=** arr[i**-**1], arr[i]

        # If current even element is smaller than next

**if** (i < n**-**1 **and** arr[i] < arr[i**+**1]):

            arr[i], arr[i**+**1] **=** arr[i**+**1], arr[i]

# Driver program

arr **=** [10, 90, 49, 2, 1, 5, 23]

sortInWave(arr, len(arr))

**for** i **in** range(0, len(arr)):

**print**(arr[i], end**=**" ")

# This code is contributed by \_\_Devesh Agrawal\_\_

**Output**

90 10 49 1 5 2 23

**Time Complexity:**O(N)

**Auxiliary Space:**O(1)

**12.Check if any two intervals intersects among a given set of intervals**

An interval is represented as a combination of start time and end time. Given a set of intervals, check if any two intervals intersect.

**Examples:**

**Input:** arr[] = {{1, 3}, {5, 7}, {2, 4}, {6, 8}}  
**Output:** true  
The intervals {1, 3} and {2, 4} overlap

**Input:** arr[] = {{1, 3}, {7, 9}, {4, 6}, {10, 13}}  
**Output:** false  
No pair of intervals overlap.

Expected time complexity is O(nLogn) where n is number of intervals.

**We strongly recommend to minimize your browser and try this yourself first.**

A **Simple Solution** is to consider every pair of intervals and check if the pair intersects or not. The time complexity of this solution is O(n2)

**Method 1**

A better solution is to **Use Sorting**. Following is complete algorithm.

1) Sort all intervals in increasing order of start time. This step takes O(nLogn) time.

2) In the sorted array, if start time of an interval is less than end of previous interval, then there is an overlap. This step takes O(n) time.

So overall time complexity of the algorithm is O(nLogn) + O(n) which is O(nLogn).

Below is the implementation of above idea.

# A Python program to check if any two intervals overlap

# An interval has start time and end time

**class** Interval:

**def** \_\_init\_\_(self, start, end):

        self.start **=** start

        self.end **=** end

# Function to check if any two intervals overlap

**def** isIntersect(arr, n):

    # Sort intervals in increasing order of start time

    arr.sort(key**=lambda** x: x.start)

    # In the sorted array, if start time of an interval

    # is less than end of previous interval, then there

    # is an overlap

**for** i **in** range(1, n):

**if** (arr[i **-** 1].end > arr[i].start):

**return** True

    # If we reach here, then no overlap

**return** False

# Driver code

arr1 **=** [Interval(1, 3), Interval(7, 9), Interval(4, 6), Interval(10, 13)]

n1 **=** len(arr1)

**if** (isIntersect(arr1, n1)):

    print("Yes")

**else**:

    print("No")

arr2 **=** [Interval(6, 8), Interval(1, 3), Interval(2, 4), Interval(4, 7)]

n2 **=** len(arr2)

**if** (isIntersect(arr2, n2)):

    print("Yes")

**else**:

**print**("No")

# This code is contributed by Saurabh Jaiswal

**Output:**

No  
Yes

**Time Complexity:** O(nlogn)

**Auxiliary Space**: O(1)

**Method 2**: This approach is suggested by **Anjali Agarwal**. Following are the steps:

*1. Find the overall maximum element. Let it be****max\_ele***

*2. Initialize an array of size max\_ele with 0.*

*3. For every interval [start, end], increment the value at index start, i.e. arr[start]++ and decrement the value at index (end + 1), i.e. arr[end + 1]- -.*

*4. Compute the prefix sum of this array (arr[]).*

*5. Every index, i of this prefix sum array will tell how many times i has occurred in all the intervals taken together. If this value is greater than 1, then it occurs in 2 or more intervals.*

*6. So, simply initialize the result variable as false and while traversing the prefix sum array, change the result variable to true whenever the value at that index is greater than 1.*

Below is the implementation of this (Method 2) approach.

# A Python program to check if any two intervals overlap

# An interval has start time and end time

**class** Interval:

**def** \_\_init\_\_(self, start, end):

        self.start **=** start

        self.end **=** end

# Function to check if any two intervals overlap

**def** is\_intersect(arr, n):

    max\_ele **=** 0

    # Find the overall maximum element

**for** i **in** range(n):

**if** max\_ele < arr[i].end:

            max\_ele **=** arr[i].end

    # Initialize an array of size max\_ele

    aux **=** [0] **\*** (max\_ele **+** 1)

**for** i **in** range(max\_ele **+** 1):

        aux[i] **=** 0

**for** i **in** range(n):

        # starting point of the interval

        x **=** arr[i].start

        # end point of the interval

        y **=** arr[i].end

        aux[x] **+=** 1

        aux[y] **-=** 1

**for** i **in** range(1, max\_ele **+** 1):

        # Calculating the prefix Sum

        aux[i] **+=** aux[i **-** 1]

        # Overlap

**if** aux[i] > 1:

**return** True

    # If we reach here, then no Overlap

**return** False

# Driver program

arr1 **=** [Interval(1, 3), Interval(7, 9), Interval(4, 6), Interval(10, 13)]

n1 **=** len(arr1)

**if** is\_intersect(arr1, n1):

**print**("Yes")

**else**:

    print("No")

arr2 **=** [Interval(6, 8), Interval(1, 3), Interval(2, 4), Interval(4, 7)]

n2 **=** len(arr2)

**if** is\_intersect(arr2, n2):

**print**("Yes")

**else**:

**print**("No")

# this code is contributed by phasing17

**Output:**

No  
Yes

**Time Complexity :** O(max\_ele + n)

**Auxiliary Space**: O(max\_ele)

**Note:** This method is more efficient than Method 1 if there are more number of intervals and at the same time maximum value among all intervals should be low, since time complexity is directly proportional to O(max\_ele).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**13.Sorting Strings using Bubble Sort**

Given an array of strings arr[]. Sort given strings using Bubble Sort and display the sorted array.

In [Bubble Sort](https://www.geeksforgeeks.org/bubble-sort/), the two successive strings arr[i] and arr[i+1] are exchanged whenever arr[i]> arr[i+1]. The larger values sink to the bottom and hence called sinking sort. At the end of each pass, smaller values gradually “bubble” their way upward to the top and hence called bubble sort.

 After all the passes, we get all the strings in sorted order. The complexity of the above algorithm will be O(N2).

Let us look at the code snippet

# Python Implementation

**def** compare(a, b):

**return** ((a < b) **-** (a > b))

**def** sort\_string(arr, n):

    temp **=** ""

    # Sort string using the bubble sort

**for** i **in** range(n**-**1):

**for** j **in** range(i**+**1, n):

**if** compare(arr[j], arr[i]) > 0:

                temp **=** arr[j]

                arr[j] **=** arr[i]

                arr[i] **=** temp

**print**("String in sorted order are: ")

**for** i **in** range(n):

        print(f'Strings {i + 1} is {arr[i]}')

# Driver code

arr **=** ["GeeksforGeeks", "Quiz", "Practice", "Gblogs", "Coding"]

n **=** len(arr)

sort\_string(arr, n)

# This code is contributed by Prince Kumar

**Output**

Strings in sorted order are :   
 String 1 is Coding  
 String 2 is Gblogs  
 String 3 is GeeksforGeeks  
 String 4 is Practice  
 String 5 is Quiz

**14.Find missing elements of a range**

Given an array, arr[0..n-1] of distinct elements and a range [low, high], find all numbers that are in a range, but not the array. The missing elements should be printed in sorted order.

**Examples:**

**Input:** arr[] = {10, 12, 11, 15},   
 low = 10, high = 15  
**Output:** 13, 14

**Input:** arr[] = {1, 14, 11, 51, 15},   
 low = 50, high = 55  
**Output:** 50, 52, 53, 54 55

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

There can be two approaches to solve the problem.

**Use Sorting:** Sort the array, then do a binary search for ‘low’. Once the location of low is found, start traversing the array from that location and keep printing all missing numbers.

**Implementation:**

# Python library for binary search

**from** bisect **import** bisect\_left

# A sorting based C++ program to find missing

# elements from an array

# Print all elements of range [low, high] that

# are not present in arr[0..n-1]

**def** printMissing(arr, n, low, high):

    # Sort the array

    arr.sort()

    # Do binary search for 'low' in sorted

    # array and find index of first element

    # which either equal to or greater than

    # low.

    ptr **=** bisect\_left(arr, low)

    index **=** ptr

    # Start from the found index and linearly

    # search every range element x after this

    # index in arr[]

    i **=** index

    x **=** low

**while** (i < n **and** x <**=** high):

    # If x doesn't match with current element

    # print it

**if**(arr[i] !**=** x):

            print(x, end **=**" ")

    # If x matches, move to next element in arr[]

**else**:

            i **=** i **+** 1

    # Move to next element in range [low, high]

        x **=** x **+** 1

    # Print range elements that are greater than the

    # last element of sorted array.

**while** (x <**=** high):

**print**(x, end **=**" ")

        x **=** x **+** 1

# Driver code

arr **=** [1, 3, 5, 4]

n **=** len(arr)

low **=** 1

high **=** 10

printMissing(arr, n, low, high);

# This code is contributed by YatinGupta

**Output**

2 6 7 8 9 10

***Time Complexity:*** O(n log n + k) where k is the number of missing elements

***Auxiliary Space:***O(n) or O(1) depending on the type of the array.

**Using Arrays:** Create a boolean array, where each index will represent whether the (i+low)th element is present in the array or not. Mark all those elements which are in the given range and are present in the array. Once all array items present in the given range have been marked true in the array, we traverse through the Boolean array and print all elements whose value is false.

**Implementation:**

# An array-based Python3 program to

# find missing elements from an array

# Print all elements of range

# [low, high] that are not

# present in arr[0..n-1]

**def** printMissing(arr, n, low, high):

    # Create boolean list of size

    # high-low+1, each index i

    # representing whether (i+low)th

    # element found or not.

    points\_of\_range **=** [False] **\*** (high**-**low**+**1)

**for** i **in** range(n) :

        # if ith element of arr is in range

        # low to high then mark corresponding

        # index as true in array

**if** ( low <**=** arr[i] **and** arr[i] <**=** high ) :

            points\_of\_range[arr[i]**-**low] **=** True

    # Traverse through the range

    # and print all elements  whose value

    # is false

**for** x **in** range(high**-**low**+**1) :

**if** (points\_of\_range[x]**==**False) :

            print(low**+**x, end **=** " ")

# Driver Code

arr **=** [1, 3, 5, 4]

n **=** len(arr)

low, high **=** 1, 10

printMissing(arr, n, low, high)

# This code is contributed

# by Shubh Bansal

**Output**

2 6 7 8 9 10

***Time Complexity:* O(n + (high-low+1))**

***Auxiliary Space:*O(n)**

**Use Hashing:** Create a hash table and insert all array items into the hash table. Once all items are in hash table, traverse through the range and print all missing elements.

# A hashing based Python3 program to

# find missing elements from an array

# Print all elements of range

# [low, high] that are not

# present in arr[0..n-1]

**def** printMissing(arr, n, low, high):

    # Insert all elements of

    # arr[] in set

    s **=** set(arr)

    # Traverse through the range

    # and print all missing elements

**for** x **in** range(low, high **+** 1):

**if** x **not in** s:

            print(x, end **=** ' ')

# Driver Code

arr **=** [1, 3, 5, 4]

n **=** len(arr)

low, high **=** 1, 10

printMissing(arr, n, low, high)

# This code is contributed

# by SamyuktaSHegde

**Output**

2 6 7 8 9 10

***me Complexity:*O(n + (high-low+1))**

***Auxiliary Space:* O(n)**

**Which approach is better?**

The time complexity of the first approach is O(nLogn + k) where k is the number of missing elements (Note that k may be more than n Log n if the array is small and the range is big)

The time complexity of the second and third solutions is O(n + (high-low+1)).

If the given array has almost all elements of the range, i.e., n is close to the value of (high-low+1), then the second and third approaches are definitely better as there is no Log n factor. But if n is much smaller than the range, then the first approach is better as it doesn’t require extra space for hashing. We can also modify the first approach to print adjacent missing elements as range to save time. For example, if 50, 51, 52, 53, 54, 59 are missing, we can print them as 50-54, 59 in the first method. And if printing this way is allowed, the first approach takes only O(n Log n) time. Out of the Second and Third Solutions, the second solution is better because the worst-case time complexity of the second solution is better than the third.

**15.Sort an array according to count of set bits**

Given an array of positive integers, sort the array in decreasing order of count of set bits in binary representations of array elements. For integers having the same number of set bits in their binary representation, sort according to their position in the original array i.e., a stable sort. For example, if the input array is {3, 5}, then the output array should also be {3, 5}. Note that both 3 and 5 have the same number set bits.

**Examples:**

**Input:** arr[] = {5, 2, 3, 9, 4, 6, 7, 15, 32};  
**Output:** 15 7 5 3 9 6 2 4 32  
**Explanation:**  
The integers in their binary representation are:  
 15 -1111  
 7 -0111  
 5 -0101  
 3 -0011  
 9 -1001  
 6 -0110  
 2 -0010  
 4- -0100  
 32 -10000  
hence the non-increasing sorted order is:  
{15}, {7}, {5, 3, 9, 6}, {2, 4, 32}

**Input:** arr[] = {1, 2, 3, 4, 5, 6};  
**Output:** 3 5 6 1 2 4  
**Explanation:**  
 3 - 0011  
 5 - 0101  
 6 - 0110  
 1 - 0001  
 2 - 0010  
 4 - 0100  
hence the non-increasing sorted order is  
{3, 5, 6}, {1, 2, 4}

**Method 1: Simple**

1. Create an auxiliary array and store the set-bit counts of all integers in the aux array
2. Simultaneously sort both arrays according to the non-increasing order of auxiliary array. (Note that we need to use a stable sort algorithm)

**Before sort:**  
int arr[] = {1, 2, 3, 4, 5, 6};  
int aux[] = {1, 1, 2, 1, 2, 2}  
**After sort:**  
arr = {3, 5, 6, 1, 2, 4}  
aux = {2, 2, 2, 1, 1, 1}

**Implementation:**

# Python 3 program to implement simple approach to sort

# an array according to count of set bits.

# a utility function that returns total set bits

# count in an integer

**def** countBits(a):

    count **=** 0

**while** (a):

**if** (a & 1):

            count**+=** 1

        a **=** a>>1

**return** count

# Function to simultaneously sort both arrays

# using insertion sort

# ( <https://www.geeksforgeeks.org/insertion-sort/> )

**def** insertionSort(arr,aux, n):

**for** i **in** range(1,n,1):

        # use 2 keys because we need to sort both

        # arrays simultaneously

        key1 **=** aux[i]

        key2 **=** arr[i]

        j **=** i**-**1

        # Move elements of arr[0..i-1] and aux[0..i-1],

        #  such that elements of aux[0..i-1] are

        # greater than key1, to one position ahead

        #  of their current position \*/

**while** (j >**=** 0 **and** aux[j] < key1):

            aux[j**+**1] **=** aux[j]

            arr[j**+**1] **=** arr[j]

            j **=** j**-**1

        aux[j**+**1] **=** key1

        arr[j**+**1] **=** key2

# Function to sort according to bit count using

# an auxiliary array

**def** sortBySetBitCount(arr, n):

    # Create an array and store count of

    # set bits in it.

    aux **=** [0 **for** i **in** range(n)]

**for** i **in** range(0,n,1):

        aux[i] **=** countBits(arr[i])

    # Sort arr[] according to values in aux[]

    insertionSort(arr, aux, n)

# Utility function to print an array

**def** printArr(arr, n):

**for** i **in** range(0,n,1):

**print**(arr[i],end **=** " ")

# Driver Code

**if** \_\_name\_\_ **==**'\_\_main\_\_':

    arr **=** [1, 2, 3, 4, 5, 6]

    n **=** len(arr)

    sortBySetBitCount(arr, n)

    printArr(arr, n)

# This code is contributed by

# Surendra\_Gangwar

**Output**

3 5 6 1 2 4

**Auxiliary Space: O(n)**

**Time complexity: O(n2)**

**Note:** Time complexity can be improved to O(nLogn) by using a stable O(nlogn) sorting algorithm.

**Method 2: Using**[**std::sort()**](https://www.geeksforgeeks.org/sort-c-stl/)

Using custom comparator of std::sort to sort the array according to set-bit count

# Using custom comparator lambda function

arr **=** [1, 2, 3, 4, 5, 6]

# form a tuple with val, index

n **=** len(arr)

arr **=** [(arr[i], i) **for** i **in** range(n)]

**def** countSetBits(val):

    cnt **=** 0

**while** val:

        cnt **+=** val **%** 2

        val **=** val**//**2

**return** cnt

# first criteria to sort is number of set bits,

# then the index

sorted\_arr **=** sorted(arr, key**=lambda** val: (

    countSetBits(val[0]), n**-**val[1]), reverse**=**True)

sorted\_arr **=** [val[0] **for** val **in** sorted\_arr]

print(sorted\_arr)

**Output**

3 5 6 1 2 4

**Auxiliary Space: O(1)**

**Time complexity: O(n log n)**

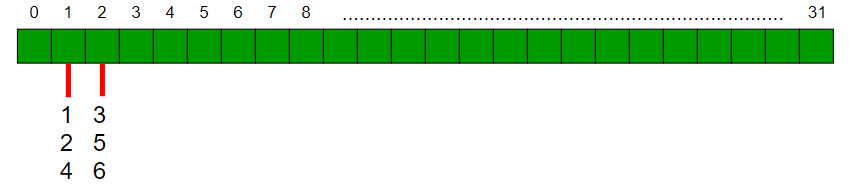
**Method 3:**[**Counting Sort**](https://www.geeksforgeeks.org/counting-sort/)**based**

This problem can be solved in O(n) time. The idea is similar to counting sort.

**Note:** There can be a minimum 1 set-bit and only a maximum of 31set-bits in an integer.

**Steps (assuming that an integer takes 32 bits):**

1. Create a vector “count” of size 32. Each cell of count i.e., count[i] is another vector that stores all the elements whose set-bit-count is i
2. Traverse the array and do the following for each element:
3. Count the number set-bits of this element. Let it be ‘setbitcount’
4. count[setbitcount].push\_back(element)
5. Traverse ‘count’ in reverse fashion(as we need to sort in non-increasing order) and modify the array.



# Python3 program to sort an array according to

# count of set bits using std::sort()

# a utility function that returns total set bits

# count in an integer

**def** countBits(a):

    count **=** 0

**while** (a):

**if** (a & 1 ):

            count **+=** 1

        a **=** a>>1

**return** count

# Function to sort according to bit count

# This function assumes that there are 32

# bits in an integer.

**def** sortBySetBitCount(arr,n):

    count **=** [[] **for** i **in** range(32)]

    setbitcount **=** 0

**for** i **in** range(n):

        setbitcount **=** countBits(arr[i])

        count[setbitcount].append(arr[i])

    j **=** 0 # Used as an index in final sorted array

    # Traverse through all bit counts (Note that we

    # sort array in decreasing order)

**for** i **in** range(31, **-**1, **-**1):

        v1 **=** count[i]

**for** i **in** range(len(v1)):

            arr[j] **=** v1[i]

            j **+=** 1

# Utility function to print an array

**def** printArr(arr, n):

**print**(**\***arr)

# Driver Code

arr **=** [1, 2, 3, 4, 5, 6]

n **=** len(arr)

sortBySetBitCount(arr, n)

printArr(arr, n)

# This code is contributed by mohit kumar 29

**Output**

3 5 6 1 2 4

**Time complexity:** O(n)

**Auxiliary Space:** O(n)

**Method 4: Using MultiMap**

**Steps:**

* Create a MultiMap whose key values will be the negative of the number of set-bits of the element.
* Traverse the array and do following for each element:
* Count the number set-bits of this element. Let it be ‘setBitCount’
* count.insert({(-1) \* setBitCount, element})
* Traverse ‘count’ and print the second elements.

Below is the implementation of the above approach:

# Python3 program to implement

# simple approach to sort

# an array according to

# count of set bits.

# Function to count setbits

**def** setBitCount(num):

    count **=** 0

**while** (num):

**if** (num & 1):

            count **+=** 1

        num **=** num >> 1

**return** count

# Function to sort By SetBitCount

**def** sortBySetBitCount(arr, n):

    count **=** []

    # Iterate over all values and

    # insert into multimap

**for** i **in** range(n):

        count.append([(**-**1) **\***

        setBitCount(arr[i]), arr[i]])

    count.sort(key **= lambda** x:x[0])

**for** i **in** range(len(count)):

        print(count[i][1], end **=** " ")

# Driver Code

arr **=** [ 1, 2, 3, 4, 5, 6 ]

n **=** len(arr)

sortBySetBitCount(arr, n)

# This code is contributed by rag2127

**Output**

3 5 6 1 2 4

**Time complexity: O(n log n)**

**Auxiliary Space: O(n)**

**16.Sort even-placed elements in increasing and odd-placed in decreasing order**

We are given an array of n distinct numbers. The task is to sort all even-placed numbers in increasing and odd-placed numbers in decreasing order. The modified array should contain all sorted even-placed numbers followed by reverse sorted odd-placed numbers.

Note that the first element is considered as even placed because of its index 0.

**Examples:**

**Input:** arr[] = {0, 1, 2, 3, 4, 5, 6, 7}  
**Output:** arr[] = {0, 2, 4, 6, 7, 5, 3, 1}  
Even-place elements : 0, 2, 4, 6  
Odd-place elements : 1, 3, 5, 7  
Even-place elements in increasing order :   
0, 2, 4, 6  
Odd-Place elements in decreasing order :   
7, 5, 3, 1

**Input:** arr[] = {3, 1, 2, 4, 5, 9, 13, 14, 12}  
**Output:** {2, 3, 5, 12, 13, 14, 9, 4, 1}  
Even-place elements : 3, 2, 5, 13, 12  
Odd-place elements : 1, 4, 9, 14  
Even-place elements in increasing order :   
2, 3, 5, 12, 13  
Odd-Place elements in decreasing order :   
14, 9, 4, 1

Recommended Problem

Bitonic Generator Sort

The idea is simple. We create two auxiliary arrays evenArr[] and oddArr[] respectively. We traverse input array and put all even-placed elements in evenArr[] and odd placed elements in oddArr[]. Then we sort evenArr[] in ascending and oddArr[] in descending order. Finally, copy evenArr[] and oddArr[] to get the required result.

**Implementation:**

# Python3 program to separately sort

# even-placed and odd placed numbers

# and place them together in sorted array.

**def** bitonicGenerator(arr, n):

    # create evenArr[] and oddArr[]

    evenArr **=** []

    oddArr **=** []

    # Put elements in oddArr[] and evenArr[]

    # as per their position

**for** i **in** range(n):

**if** ((i **%** 2) **==** 0):

            evenArr.append(arr[i])

**else**:

            oddArr.append(arr[i])

    # sort evenArr[] in ascending order

    # sort oddArr[] in descending order

    evenArr **=** sorted(evenArr)

    oddArr **=** sorted(oddArr)

    oddArr **=** oddArr[::**-**1]

    i **=** 0

**for** j **in** range(len(evenArr)):

        arr[i] **=** evenArr[j]

        i **+=** 1

**for** j **in** range(len(oddArr)):

        arr[i] **=** oddArr[j]

        i **+=** 1

# Driver Code

arr **=** [1, 5, 8, 9, 6, 7, 3, 4, 2, 0]

n **=** len(arr)

bitonicGenerator(arr, n)

**for** i **in** arr:

**print**(i, end **=** " ")

# This code is contributed by Mohit Kumar

**Output**

1 2 3 6 8 9 7 5 4 0

**Time Complexity:** O(n Log n)

**Auxiliary Space:**O(n)

The above problem can also be solved without the use of Auxiliary space. The idea is to swap the first half odd index positions with the second half even index positions and then sort the first half array in increasing order and the second half array in decreasing order. Thanks to **SWARUPANANDA DHUA** for suggesting this.

**Implementation:**

# Python3 Program to sort even-placed elements in increasing and

# odd-placed in decreasing order with constant space complexity

**def** bitonicGenerator(arr, n):

    # first odd index

    i **=** 1

    # last index

    j **=** n **-** 1

    # if last index is odd

**if** (j **%** 2 !**=** 0):

        # decrement j to even index

        j **=** j **-** 1

    # swapping till half of array

**while** (i < j) :

        arr[j], arr[i] **=** arr[i], arr[j]

        i **=** i **+** 2

        j **=** j **-** 2

    arr\_f **=** []

    arr\_s **=** []

**for** i **in** range(int((n **+** 1) **/** 2)) :

        arr\_f.append(arr[i])

    i **=** int((n **+** 1) **/** 2)

**while**( i < n ) :

        arr\_s.append(arr[i])

        i **=** i **+** 1

    # Sort first half in increasing

    arr\_f.sort()

    # Sort second half in decreasing

    arr\_s.sort(reverse **=** True)

**for** i **in** arr\_s:

        arr\_f.append(i)

**return** arr\_f

# Driver Program

arr **=** [ 1, 5, 8, 9, 6, 7, 3, 4, 2, 0]

n **=** len(arr)

arr **=** bitonicGenerator(arr, n)

print(arr)

# This code is contributed by Arnab Kundu

**Output**

1 2 3 6 8 9 7 5 4 0

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

**Auxiliary Space:**O(1)

**Another approach:**

Another efficient approach to solve the problem in O(1) Auxiliary space is by **Using negative multiplication**.

The steps involved are as follows:

1. Multiply all the elements at even placed index by -1.
2. Sort the whole array. In this way, we can get all even placed index in the starting as they are negative numbers now.
3. Now revert the sign of these elements.
4. After this reverse the first half of the array which contains an even placed number to make it in increasing order.
5. And then reverse the rest half of the array to make odd placed numbers in decreasing order.

**Note:** This method is only applicable if all the elements in the array are non-negative.

**An illustrative example of the above approach:**

*Let given array:****arr[] = {0, 1, 2, 3, 4, 5, 6, 7}***

*Array after multiplying by -1 to even placed elements:****arr[] = {0, 1, -2, 3, -4, 5, -6, 7}***

*Array after sorting: arr[] =****{-6, -4, -2, 0, 1, 3, 5, 7}***

*Array after reverting negative values: arr[] =****{6, 4, 2, 0, 1, 3, 5, 7}***

*After reversing the first half of array: arr[] =****{0, 2, 4, 6, 1, 3, 5, 7}***

*After reversing the second half of array: arr[] =****{0, 2, 4, 6, 7, 5, 3, 1}***

Below is the code for the above approach:

**class** GFG :

    @staticmethod

**def** reverse( a,  l,  r) :

**while** (l <**=** r) :

            temp **=** a[l]

            a[l] **=** a[r]

            a[r] **=** temp

            l **+=** 1

            r **-=** 1

    @staticmethod

**def** bitonicGenerator( arr,  n) :

        # Making all even placed index

        # element negative

        i **=** 0

**while** (i < n) :

**if** (i **%** 2 **==** 0) :

                arr[i] **= -**1 **\*** arr[i]

            i **+=** 1

        # Sorting the whole array

        arr.sort()

        # Finding the middle value of

        # the array

        mid **=** int((n **-** 1) **/** 2)

        # Reverting the changed sign

        i **=** 0

**while** (i <**=** mid) :

            arr[i] **= -**1 **\*** arr[i]

            i **+=** 1

        # Reverse first half of array

        GFG.reverse(arr, 0, mid)

        # Reverse second half of array

        GFG.reverse(arr, mid **+** 1, n **-** 1)

    # Driver Code

    @staticmethod

**def** main( args) :

        arr **=** [1, 5, 8, 9, 6, 7, 3, 4, 2, 0]

        n **=** len(arr)

        GFG.bitonicGenerator(arr, n)

        i **=** 0

**while** (i < n) :

**print**(str(arr[i]) **+** " ", end **=**"")

            i **+=** 1

**if** \_\_name\_\_**==**"\_\_main\_\_":

    GFG.main([])

    # This code is contributed by aadityaburujwale.

**Output**

1 2 3 6 8 9 7 5 4 0

**Time Complexity:** O(n\*log(n))

**Auxiliary Space:**O(1)

**17.Sort an array when two halves are sorted**

Given an integer array of which both first half and second half are sorted. Task is to merge two sorted halves of array into single sorted array.

**Examples:**

Input : A[] = { 2, 3, 8, -1, 7, 10 }  
Output : -1, 2, 3, 7, 8, 10

Input : A[] = {-4, 6, 9, -1, 3 }  
Output : -4, -1, 3, 6, 9

Recommended Problem

Sort the Half Sorted

**Method 1:**A **Simple Solution**is to sort the array using built in functions (generally an implementation of quick sort).

Below is the implementation of above method:

# Python program to Merge two sorted

# halves of array Into Single Sorted Array

**def** mergeTwoHalf(A, n):

    # Sort the given array using sort STL

    A.sort()

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    A **=** [2, 3, 8, **-**1, 7, 10]

    n **=** len(A)

    mergeTwoHalf(A, n)

    # Print sorted Array

**for** i **in** range(n):

**print**(A[i], end**=**" ")

# This code is contributed by 29AjayKumar

**Output**

-1 2 3 7 8 10

**Time Complexity:**

\*\*\* QuickLaTeX cannot compile formula:

\*\*\* Error message:  
Error: Nothing to show, formula is empty

best & average case,

worst case (for quicksort)

**Space Complexity:**

\*\*\* QuickLaTeX cannot compile formula:

\*\*\* Error message:  
Error: Nothing to show, formula is empty

**to**

\*\*\* QuickLaTeX cannot compile formula:

\*\*\* Error message:  
Error: Nothing to show, formula is empty

depending on the case & implementation (for quicksort)

For more details, check out the GFG article on [Quicksort.](https://www.geeksforgeeks.org/quick-sort/)

**Method 2:**A **more efficient solution**is to use an auxiliary array which is very similar to the Merge Function of [Merge sort](https://www.geeksforgeeks.org/merge-sort/).

Below is the implementation of above approach :

# Python3 program to Merge Two Sorted Halves Of

# Array Into Single Sorted Array

# Merge two sorted halves of Array into single

# sorted array

**def** mergeTwoHalf(A, n):

    # Starting index of second half

    half\_i **=** 0

    # Temp Array store sorted resultant array

    temp **=** [0 **for** i **in** range(n)]

    # First Find the point where array is

    # divide into two half

**for** i **in** range(n **-** 1):

**if** (A[i] > A[i **+** 1]):

            half\_i **=** i **+** 1

**break**

    # If Given array is all-ready sorted

**if** (half\_i **==** 0):

**return**

    # Merge two sorted arrays in single

    # sorted array

    i **=** 0

    j **=** half\_i

    k **=** 0

**while** (i < half\_i **and** j < n):

**if** (A[i] < A[j]):

            temp[k] **=** A[i]

            k **+=** 1

            i **+=** 1

**else**:

            temp[k] **=** A[j]

            k **+=** 1

            j **+=** 1

    # Copy the remaining elements of A[i to half\_! ]

**while** i < half\_i:

        temp[k] **=** A[i]

        k **+=** 1

        i **+=** 1

    # Copy the remaining elements of A[ half\_! to n ]

**while** (j < n):

        temp[k] **=** A[j]

        k **+=** 1

        j **+=** 1

**for** i **in** range(n):

        A[i] **=** temp[i]

# Driver code

A **=** [ 2, 3, 8, **-**1, 7, 10 ]

n **=** len(A)

mergeTwoHalf(A, n)

# Print sorted Array

**print**(**\***A, sep **=** ' ')

# This code is contributed by avanitrachhadiya2155

**Output**

-1 2 3 7 8 10

**18.Sorting Big Integers**

Given a array of **n** positive integers where each integer can have digits upto 106, print the array elements in ascending order.

Input: arr[] = {54, 724523015759812365462, 870112101220845, 8723}   
Output: 54 8723 870112101220845 724523015759812365462  
Explanation:  
All elements of array are sorted in non-descending(i.e., ascending)  
order of their integer value

Input: arr[] = {3643641264874311, 451234654453211101231,  
 4510122010112121012121}  
Output: 3641264874311 451234654453211101231 4510122010112121012121

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

A **naive approach** is to use arbitrary precision data type such as **int** in python or **Biginteger** class in . But that approach will not be fruitful because internal conversion of string to int and then perform sorting will leads to slow down the calculations of addition and multiplications in binary number system.

**Efficient Solution :**As size of integer is very large even it can’t be fit in **long long** data type of C/C++, so we just need to input all numbers as strings and sort them using a comparison function. Following are the key points compare function:-

1. If lengths of two strings are different, then we need to compare lengths to decide sorting order.
2. If Lengths are same then we just need to compare both the strings in lexicographically order.

Assumption : There are no leading zeros.

# Below is Python code to sort the Big integers

# in ascending order

**def** SortingBigIntegers(arr, n):

  # Direct sorting using lambda operator

  # and length comparison

  arr.sort(key **= lambda** x: (len(x), x))

# Driver code of above implementation

arr **=** ["54", "724523015759812365462",

        "870112101220845", "8723"]

n **=** len(arr)

SortingBigIntegers(arr, n)

# Print the final sorted list using

# join method

print " ".join(arr)

**Output:** 54 8723 870112101220845 724523015759812365462

**Time complexity:**O(sum \* log(n)) where sum is the total sum of all string length and n is size of array

**Auxiliary space:**O(n)

**19.Sort an array when two halves are sorted**

Given an integer array of which both first half and second half are sorted. Task is to merge two sorted halves of array into single sorted array.

**Examples:**

Input : A[] = { 2, 3, 8, -1, 7, 10 }  
Output : -1, 2, 3, 7, 8, 10

Input : A[] = {-4, 6, 9, -1, 3 }  
Output : -4, -1, 3, 6, 9

Recommended Problem

Sort the Half Sorted

**Method 1:**A **Simple Solution**is to sort the array using built in functions (generally an implementation of quick sort).

Below is the implementation of above method:

# Python program to Merge two sorted

# halves of array Into Single Sorted Array

**def** mergeTwoHalf(A, n):

    # Sort the given array using sort STL

    A.sort()

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    A **=** [2, 3, 8, **-**1, 7, 10]

    n **=** len(A)

    mergeTwoHalf(A, n)

    # Print sorted Array

**for** i **in** range(n):

**print**(A[i], end**=**" ")

# This code is contributed by 29AjayKumar

**Output**

-1 2 3 7 8 10

**Time Complexity:**

\*\*\* QuickLaTeX cannot compile formula:

\*\*\* Error message:  
Error: Nothing to show, formula is empty

best & average case,

worst case (for quicksort)

**Space Complexity:**

\*\*\* QuickLaTeX cannot compile formula:

\*\*\* Error message:  
Error: Nothing to show, formula is empty

**to**

\*\*\* QuickLaTeX cannot compile formula:

\*\*\* Error message:  
Error: Nothing to show, formula is empty

depending on the case & implementation (for quicksort)

For more details, check out the GFG article on [Quicksort.](https://www.geeksforgeeks.org/quick-sort/)

**Method 2:**A **more efficient solution**is to use an auxiliary array which is very similar to the Merge Function of [Merge sort](https://www.geeksforgeeks.org/merge-sort/).

Below is the implementation of above approach :

# Python3 program to Merge Two Sorted Halves Of

# Array Into Single Sorted Array

# Merge two sorted halves of Array into single

# sorted array

**def** mergeTwoHalf(A, n):

    # Starting index of second half

    half\_i **=** 0

    # Temp Array store sorted resultant array

    temp **=** [0 **for** i **in** range(n)]

    # First Find the point where array is

    # divide into two half

**for** i **in** range(n **-** 1):

**if** (A[i] > A[i **+** 1]):

            half\_i **=** i **+** 1

**break**

    # If Given array is all-ready sorted

**if** (half\_i **==** 0):

**return**

    # Merge two sorted arrays in single

    # sorted array

    i **=** 0

    j **=** half\_i

    k **=** 0

**while** (i < half\_i **and** j < n):

**if** (A[i] < A[j]):

            temp[k] **=** A[i]

            k **+=** 1

            i **+=** 1

**else**:

            temp[k] **=** A[j]

            k **+=** 1

            j **+=** 1

    # Copy the remaining elements of A[i to half\_! ]

**while** i < half\_i:

        temp[k] **=** A[i]

        k **+=** 1

        i **+=** 1

    # Copy the remaining elements of A[ half\_! to n ]

**while** (j < n):

        temp[k] **=** A[j]

        k **+=** 1

        j **+=** 1

**for** i **in** range(n):

        A[i] **=** temp[i]

# Driver code

A **=** [ 2, 3, 8, **-**1, 7, 10 ]

n **=** len(A)

mergeTwoHalf(A, n)

# Print sorted Array

**print**(**\***A, sep **=** ' ')

# This code is contributed by avanitrachhadiya2155

**Output**

-1 2 3 7 8 10

**20.Sort a linked list of 0s, 1s and 2s**

Given a linked list of**0s, 1s and 2s**, The task is to sort and print it.

**Examples**:

***Input:****1 -> 1 -> 2 -> 0 -> 2 -> 0 -> 1 -> NULL*

***Output:****0 -> 0 -> 1 -> 1 -> 1 -> 2 -> 2 -> NULL*

***Input:****1 -> 1 -> 2 -> 1 -> 0 -> NULL*

***Output:****0 -> 1 -> 1 -> 1 -> 2 -> NULL*

Source: [Microsoft Interview | Set 1](https://www.geeksforgeeks.org/microsoft-interview-set-1/)

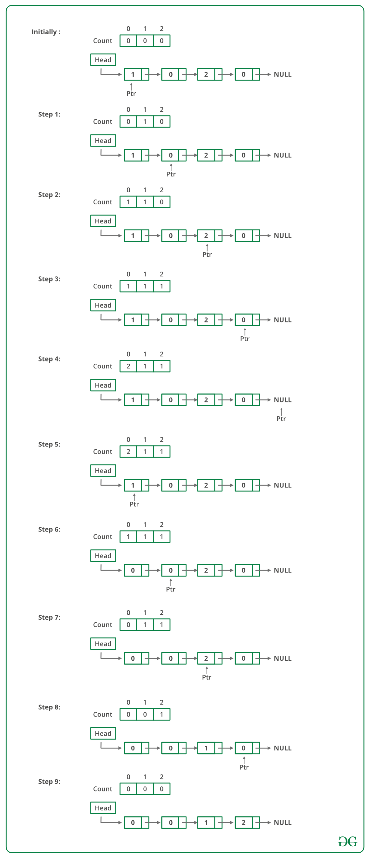
[Recommended: Please solve it on “***PRACTICE***” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/given-a-linked-list-of-0s-1s-and-2s-sort-it/1)

**Sort Linked List of 0s, 1s and 2s using frequency counting:**

Follow the below steps to implement the idea:

* Traverse the list and count the number of 0s, 1s, and 2s. Let the counts be n1, n2, and n3 respectively.
* Traverse the list again, fill the first n1 nodes with 0, then n2 nodes with 1, and finally n3 nodes with 2.

Below image is a dry run of the above approach:



Below is the implementation of the above approach.

# Python program to sort a linked list of 0, 1 and 2

**class** LinkedList(object):

**def** \_\_init\_\_(self):

         # head of list

         self.head **=** None

    # Linked list Node

**class** Node(object):

**def** \_\_init\_\_(self, d):

            self.data **=** d

            self.next **=** None

**def** sortList(self):

        # initialise count of 0 1 and 2 as 0

        count **=** [0, 0, 0]

        ptr **=** self.head

        # count total number of '0', '1' and '2'

        # \* count[0] will store total number of '0's

        # \* count[1] will store total number of '1's

        # \* count[2] will store total number of '2's

**while** ptr !**=** None:

            count[ptr.data]**+=**1

            ptr **=** ptr.next

        i **=** 0

        ptr **=** self.head

        # Let say count[0] = n1, count[1] = n2 and count[2] = n3

        # \* now start traversing list from head node,

        # \* 1) fill the list with 0, till n1 > 0

        # \* 2) fill the list with 1, till n2 > 0

        # \* 3) fill the list with 2, till n3 > 0

**while** ptr !**=** None:

**if** count[i] **==** 0:

                i**+=**1

**else**:

                ptr.data **=** i

                count[i]**-=**1

                ptr **=** ptr.next

    # Utility functions

    # Inserts a new Node at front of the list.

**def** push(self, new\_data):

        # 1 & 2: Allocate the Node &

        # Put in the data

        new\_node **=** self.Node(new\_data)

        # 3. Make next of new Node as head

        new\_node.next **=** self.head

        # 4. Move the head to point to new Node

        self.head **=** new\_node

    # Function to print linked list

**def** printList(self):

        temp **=** self.head

**while** temp !**=** None:

            print (str(temp.data),end**=**" ")

            temp **=** temp.next

**print**()

# Driver program to test above functions

llist **=** LinkedList()

llist.push(0)

llist.push(1)

llist.push(0)

llist.push(2)

llist.push(1)

llist.push(1)

llist.push(2)

llist.push(1)

llist.push(2)

print ("Linked List before sorting")

llist.printList()

llist.sortList()

print ("Linked List after sorting")

llist.printList()

# This code is contributed by BHAVYA JAIN

**Output**

Linked List before Sorting  
2 1 2 1 1 2 0 1 0   
Linked List after Sorting  
0 0 1 1 1 1 2 2 2

**Time Complexity:** O(n) where n is the number of nodes in the linked list.

**Auxiliary Space:** O(1)

[**Sort a linked list of 0s, 1s and 2s by changing links**](https://www.geeksforgeeks.org/sort-linked-list-0s-1s-2s-changing-links/)

**Using a stable sort algorithm:**

# Define a class for a node in a linked list

**class** Node:

**def** \_\_init\_\_(self, data):

        self.data **=** data

        self.next **=** None

# Function to insert a new node at the end of the linked list

**def** push(head, data):

    new\_node **=** Node(data)

**if** head **is** None:

        head **=** new\_node

**return** head

    last **=** head

**while** last.next **is not** None:

        last **=** last.next

    last.next **=** new\_node

**return** head

# Function to print the linked list

**def** printList(head):

    current **=** head

**while** current **is not** None:

        print(current.data, end**=**' ')

        current **=** current.next

    print()

# Function to sort the linked list containing 0's, 1's, and 2's

**def** sortList(head):

    count **=** [0, 0, 0]

    # Count the number of 0's, 1's, and 2's in the linked list

    current **=** head

**while** current **is not** None:

        count[current.data] **+=** 1

        current **=** current.next

    # Overwrite the linked list with the sorted elements

    current **=** head

    i **=** 0

**while** current **is not** None:

**if** count[i] **==** 0:

            i **+=** 1

**else**:

            current.data **=** i

            count[i] **-=** 1

            current **=** current.next

# Main function to test the implementation

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    head **=** None

    # Insert some elements into the linked list

    head **=** push(head, 0)

    head **=** push(head, 1)

    head **=** push(head, 0)

    head **=** push(head, 2)

    head **=** push(head, 1)

    head **=** push(head, 1)

    head **=** push(head, 2)

    head **=** push(head, 1)

    head **=** push(head, 2)

**print**("Linked List before Sorting:")

    printList(head)

    sortList(head)

**print**("Linked List after Sorting:")

    printList(head)

**Output**

Linked List before Sorting:   
0 1 0 2 1 1 2 1 2   
Linked List after Sorting:   
0 0 1 1 1 1 2 2 2

**Explanation of the above approach:**

* The Node structure: This structure is used to define a node in a linked list. It contains an integer data to store the data of the node and a pointer next to the next node in the linked list.
* The push function: This function is used to insert a new node at the end of the linked list. It takes two parameters: a pointer to the head of the linked list and an integer data to be stored in the new node. The function first creates a new node with the given data and sets its next pointer to nullptr. If the linked list is empty, the function sets the head of the linked list to the new node. Otherwise, it iterates through the linked list to find the last node and inserts the new node after it.
* The printList function: This function is used to print the linked list. It takes one parameter, a pointer to the head of the linked list, and iterates through the list while printing the data of each node.
* The sortList function: This function is used to sort the linked list. It takes one parameter, a pointer to the head of the linked list. The function first initializes an array count of size 3 to store the count of 0’s, 1’s, and 2’s in the linked list. It then iterates through the linked list and increments the count of each element as it appears. Finally, it overwrites the linked list with the sorted elements. The function iterates through the linked list again and whenever it encounters a node with a value of 0, it sets its data to 0 and decrements the count of 0’s. Similarly, whenever it encounters a node with a value of 1, it sets its data to 1 and decrements the count of 1’s. And whenever it encounters a node with a value of 2, it sets its data to 2 and decrements the count of 2’s.
* The main function: This function is the entry point of the program. It first creates an empty linked list and inserts some elements into it. It then calls the sortList function to sort the linked list. Finally, it prints the original and sorted linked lists to verify that the sorting is correct.

Note: This is a simple implementation of the “Using a stable sort algorithm” approach for sorting a linked list containing only 0’s, 1’s, and 2’s. It may not be the most efficient implementation for large linked lists, but it should give you a good understanding of the basic idea

**Time complexity:**O(n),

The time complexity of this implementation is O(n), where n is the number of elements in the linked list. This is because the program iterates through the linked list twice: once to count the number of 0’s, 1’s, and 2’s, and once to overwrite the linked list with the sorted elements. The count operation takes O(n) time and the overwrite operation takes O(n) time, so the total time complexity is O(n) + O(n) = O(n).

**Space complexity**: O(1),

The space complexity of this implementation is O(1), because it uses only a constant amount of extra memory to store the count array of size 3. The linked list itself is not used to store any additional information during the sorting, so its space complexity remains the same.

**Medium Questions:**

**Hard Questions:**