*“An algorithm is like a recipe.” ― Muhammad Waseem*

Dear reader, welcome to a new problem based on ***Sorting*** and the ***two pointer technique***. The problem name is *‘*[*Sort 01*2](https://www.pepcoding.com/resources/online-java-foundation/time-and-space-complexity/sort-012-official/ojquestion)’. It is an extension to the previous problem named ‘[Sort 01](https://www.pepcoding.com/resources/online-java-foundation/time-and-space-complexity/sort-01-official/ojquestion)’.

***Problem Statement***

You are given an array of size n. The array can contain only three types of values, which are 0, 1 and 2. You have to sort this array of 0s, 1s and 2s, i.e. place all the zeros before ones and all the ones before twos.

Note: You are ***not*** required to write a *stable* sort algorithm. It means that the 0’s need can be arranged in any order among themselves, and similarly 1’s can be in any order among themselves, but must be after 0’s only, and the same goes for all the 2’s.

***Example:*** If the array is {0, 1, 2, 0, 2, 1, 0, 2, 1}, then the final sorted array should be: {0, 0, 0, 1, 1, 1, 2, 2, 2}

***Deducing Algorithm***

A ***O(n log2n)*** solution based on merge sort/quick sort/ heap sort is very *trivial*. Just ignore the fact that the array can have only 3 elements 0,1 and 2, and treat it as a normal array and sort it.

I am sure Interviewer will not expect such a solution. Activate your brain cells and think, how can you use the condition that there can be only three types of values in the array to solve the problem in less than O(n logn) time.

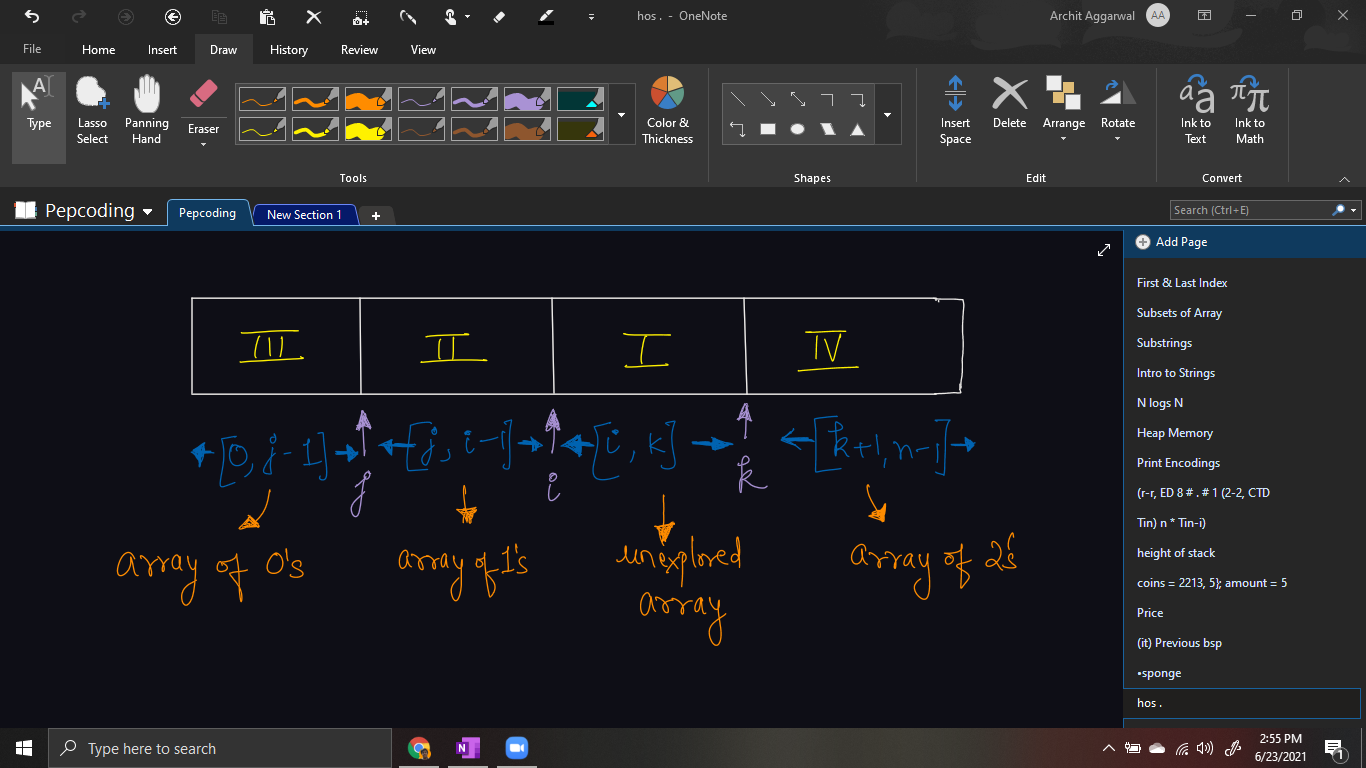
A ***two traversal algorithm*** which may come to your mind is to first count the number of 0s and 1s in the entire array. Let the count of 0 be *zeros* and count of 1 be *ones*. Then traverse the array again, and make the first ‘*zeros’* elements as ‘0’, then the next ‘*ones’* elements as 1, and the remaining (n - zeros - ones) elements as 2. This solution has a O(2 \* n) = ***O(n) time*** complexity and will take ***O(1) auxiliary space***.

This solution seems perfect, right? I must tell you, friend, that there exists a ***single-pass algorithm*** for this problem as well. Let us look at the algorithm.

We will take three pointers: i, j and k, to solve the problem in one traversal. We can maintain four regions or segments of the array using the three pointers.

1. **First segment** will be i to k which represents the ***unexplored array***.
2. **Second segment** will be j to i which represents the ***array of 1s***.
3. **Third segment** will be 0 to j-1 which represents the ***array of 0s***.
4. ***Fourth segment*** will be k + 1 to n-1 which represents the ***array of 2s***.

For each element in the unexplored array, we need to add it to either the array of 1s or the array of 0s or the array of 2s depending on its value. Please note that we will ***initialize i = 0, j = 0 and k = n-1***.



Try to figure out an algorithm! Cases for current elements as 0 or 1 will be similar to the previous problem. But we need to maintain a segment of 2s at the end of the array as well.

***Current Element = 1***

* If the first element in the unexplored array, i.e. arr[i] is equal to 1, then it should be added to the second segment (array of 1s).
* Hence, we can simply move the pointer ‘*i*’ to the next position.
* Moving pointer i by 1 position means we are expanding the array of 1s by one length and reducing the unexplored array.

***Current Element = 0***

* If the first element in the unexplored array, i.e. arr[i] is equal to 0, then it should be added to the third segment (array of 0s).
* Similar to the previous solution ([***sort 01***](https://www.pepcoding.com/resources/online-java-foundation/time-and-space-complexity/sort-01-official/ojquestion)), what we can do is swap the elements at arr[i] and arr[j].
* By doing it, we are able to add the current element (= 0) to the array of 0s and also shift the array of 1s to one place right (occupying arr[j]).
* After swapping the elements, now since the third segment (array of 0s) has increased, we will increment the j pointer by 1.
* Also, since the unexplored element is now explored, we will increment the i pointer by 1.

***Current Element = 2***

* If the current element (in the unexplored array), i.e. arr[i] = 2, then we need to place it in the fourth segment, i.e. in the array of 2s.
* Hence we can swap the values at index i and k. We will also move the pointer k in the left direction by one position, i.e. decrement k by 1.
* By doing so, we will add the element 2 in the array of 2s.
* **Q)** Why are we not incrementing the pointer i to the next position in the unexplored array in this case?

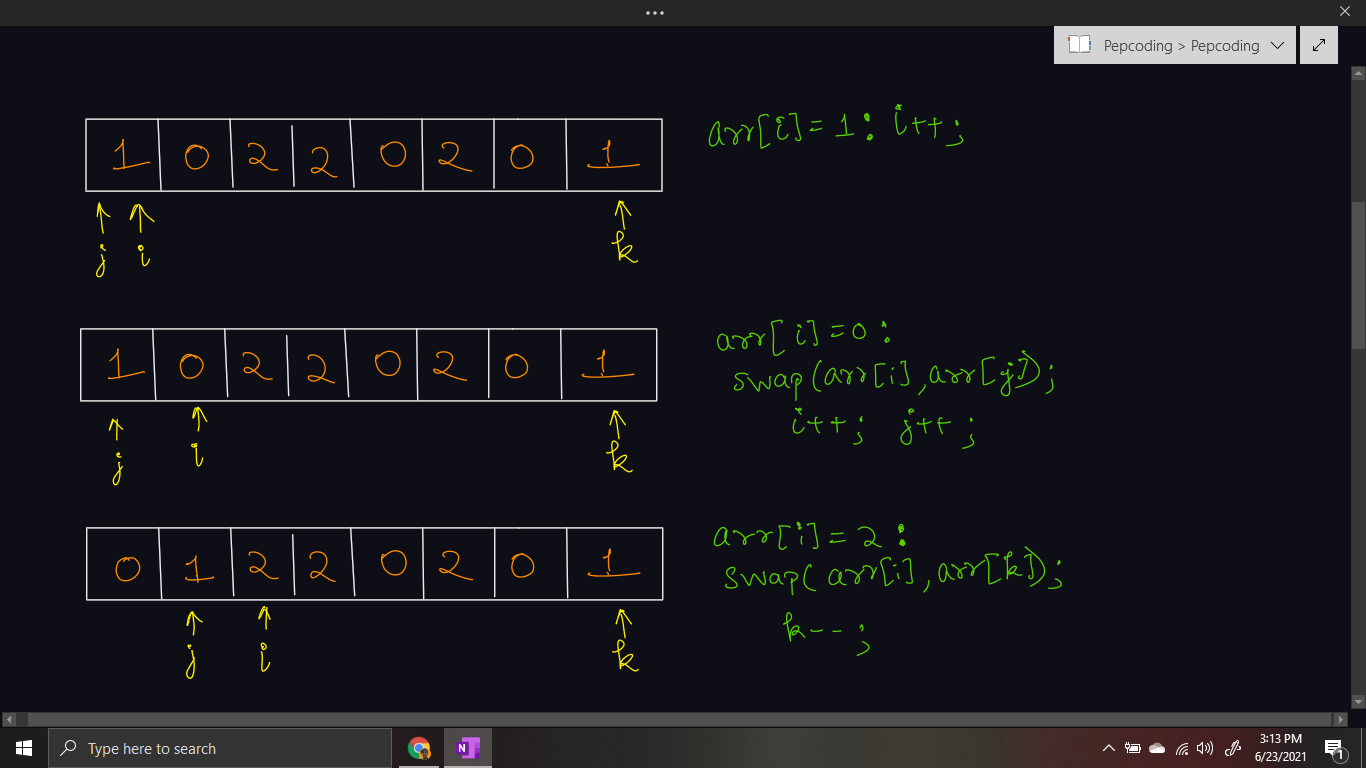
* Please note that we have reduced the unexplored segment by 1 element from the right side in this case (by decrementing k).
* We do not know what the value at index k (arr[k]) is, because before swapping values at i and k, arr[k] is also the part of the unexplored array (i to k).
* Hence, if we increment i and decrement k both at the same time, we are reducing the unexplored segment by 2 elements.
* Also, since we do not know arr[k]’s value and if we increment i, then we are adding it to the segment of 1s (j to i). But maybe arr[k] was not equal to 1 but equal to 0 only. Hence, we do not increment i, as after swapping i and k, arr[i] remains unexplored.

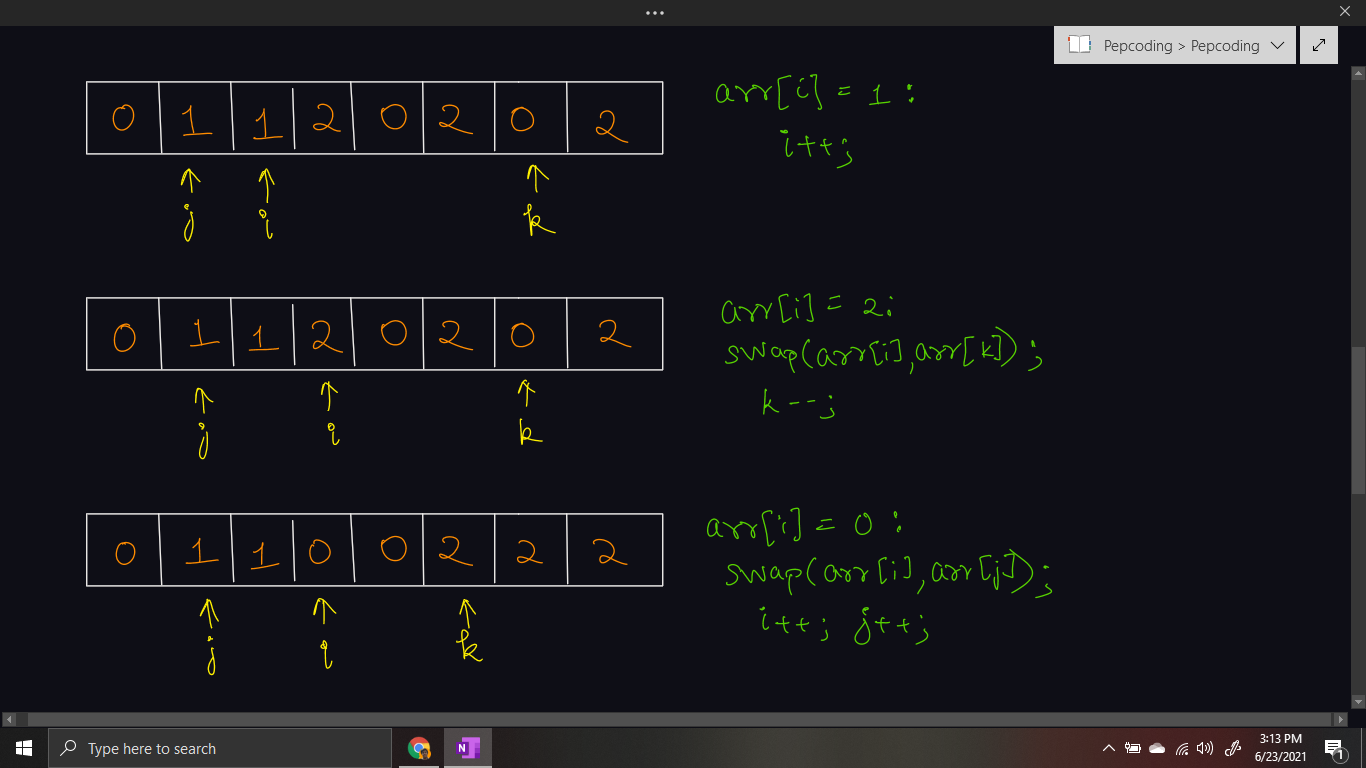
Finally, when all the elements are explored from the unexplored array, i.e. the ***unexplored segment contains 0 elements***, then the ***array is sorted***, since there remains only 3 segments: Array of 0s, Array of 1s and Array of 2s respectively. The unexplored segment will contain 0 elements when ‘*i*’ becomes equal to k + 1.

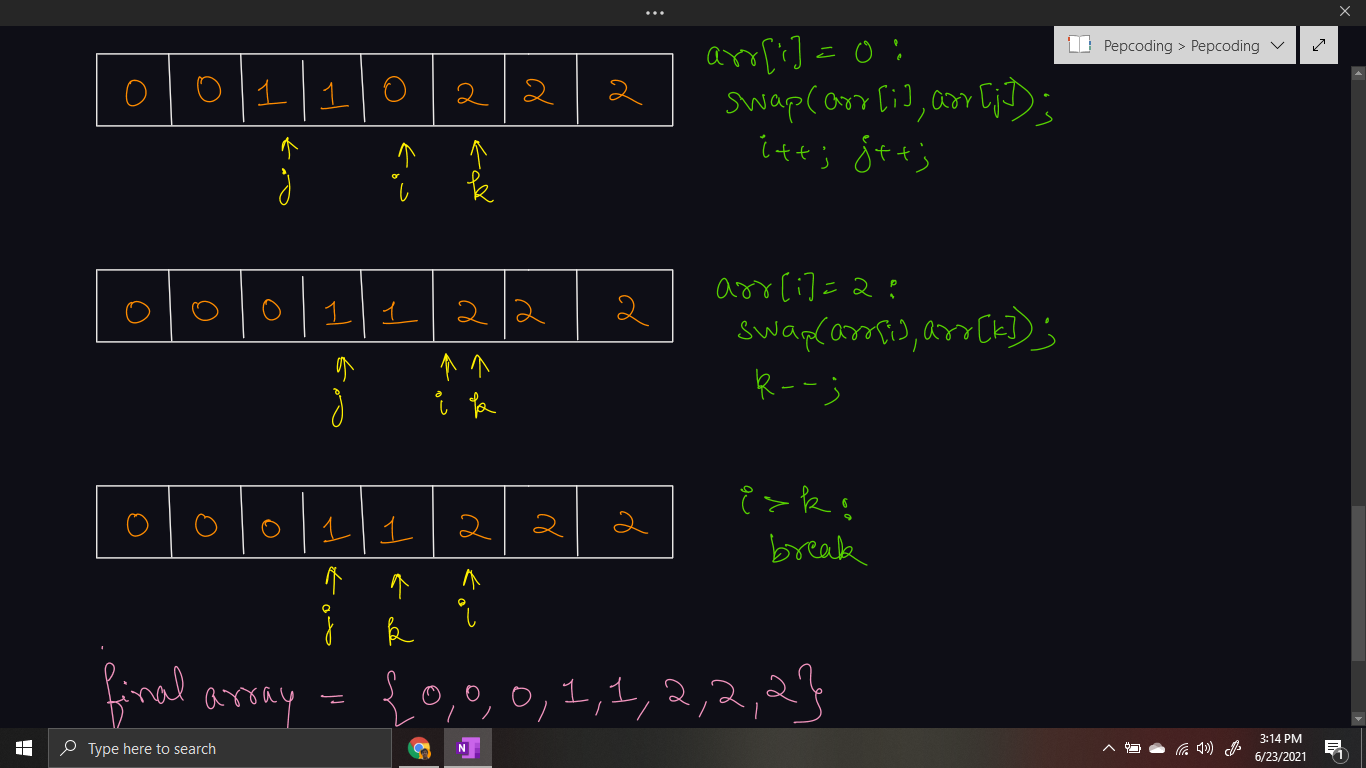
***Pseudo Code/ Algorithm***

* Maintain three indexes. Initialize first index *i* as 0 and second index *j* as 0, and third index k as n-1.
* Run the loop until i becomes equal to k
  + If arr[i] = 1, then increment i (i++).
  + Else if arr[i] = 0,
    - Swap values at arr[i] and arr[j]
    - Increment both i and j (i++, j++).
  + Else (arr[i] = 2),
    - Swap values at arr[i] and arr[k]
    - Decrement k only (k--).

Let us take a look at an example and run the algorithm discussed above.







Please try to code this without taking help of the video solution. It will help you develop an insight about the ***partitioning technique***.

***Implementation (Java)***

import java.io.\*;

import java.util.\*;

public class Main {

public static void sort012(int[] arr){

// 0 to j-1 -> All Zeroes

// j to i-1 -> All One's

// i to k - 1 -> All unknowns

// k to last -> All Two's

int i = 0 , j= 0 , k = arr.length - 1;

while(i <= k){

if(arr[i] == 0){

swap(arr,i,j);

i++;

j++;

}else if(arr[i] == 1){

i++;

}else{

swap(arr, i, k);

k--;

}

}

}

// used for swapping ith and jth elements of array

public static void swap(int[] arr, int i, int j) {

System.out.println("Swapping index " + i + " and index " + j);

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

public static void print(int[] arr){

for(int i = 0 ; i < arr.length; i++){

System.out.println(arr[i]);

}

}

public static void main(String[] args) throws Exception {

Scanner scn = new Scanner(System.in);

int n = scn.nextInt();

int[] arr = new int[n];

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

arr[i] = scn.nextInt();

}

sort012(arr);

print(arr);

}

}

This code is written and explained by our team in [this video](https://www.youtube.com/watch?v=MbV26HWqxrs) from *[17:05, 19:35]*. Please refer to it if you are stuck somewhere.

***Time & Space Complexity Analysis***

**Time Complexity** - If you analyze carefully then you can see that at each iteration, we are reducing the unexplored segment by 1 element. Hence, we are taking ***O(n) time*** only.

Don’t get persuaded by three pointers, all moving in different directions. It can be a little confusing that we are having three pointers, still the time complexity is O(n).

**Space Complexity** - Since we are sorting the array in-place, i.e. without taking any extra space, ***O(1)*** *auxiliary space* is required.

**Extra Gyaan (Knowledge)***:* This question can be asked in many ways, by changing 0,1 and 2 like:

1. Segregate -X, 0, and +X.
2. Segregate negative, zero and positive elements
3. Segregate red, blue and green colors
4. Sort array of Xs, Ys and Zs.

And so on, but the gist of the algorithm is the same, hence focus on the *partitioning technique*.

**Asked in Companies**: *Adobe, Amazon, Hike, MakeMyTrip, MAQ Software, Microsoft, Morgan Stanley, Ola Cabs, Paytm, Qualcomm, SAP Labs, Snapdeal, Walmart, Yatra.com*

I hope you enjoyed solving the problem with me. I will see you in the next problem: ‘[*Target Sum Pair - 1*](https://www.pepcoding.com/resources/online-java-foundation/time-and-space-complexity/target-sum-pair-1-official/ojquestion)’, until then *Hasta La Vista*!

Contributor : [Archit Aggarwal](https://www.linkedin.com/in/archit-aggarwal-6a7716189/)