*"Any fool can bomb a train, but just try sorting out the mess."*

*~ Aleksandr Solzhenitsyn*

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

***Problem Statement***

You are given a binary array of size n. Binary array refers to an array with all elements being either 0 or 1. You have to sort the array, i.e. place all the zeros before all the ones.

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.

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

***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 has only 2 elements 0 or 1, and treat it as a normal array and sort it.

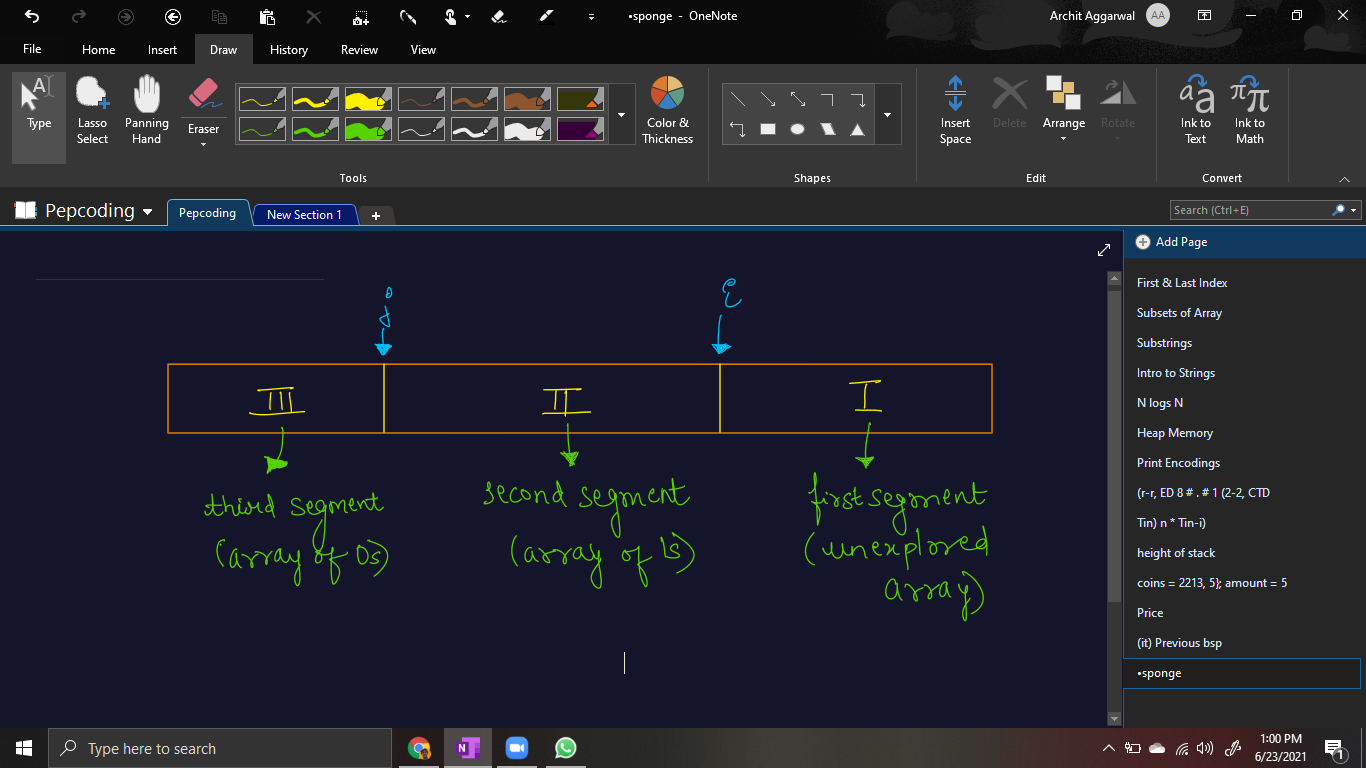
I am sure Interviewer will not expect such a solution. Activate your brain cells and think, how can you use the condition that there can be only two types of values in the array to solve the problem in less than O(n logn) time.

A ***two traversal algorithm*** which may come to your mind is to first count the number of 0s in the entire array. Let the count of zeroes be *cnt*. Then traverse the array again, and make the first *cnt* elements as ‘0’, and the rest of n - cnt elements as ‘1’. 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 the problem. Let us look at the algorithm.

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

1. **First segment** will be i to n - 1 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***.



For each element in the unexplored array, we need to add it to either the array of 1s or 0s. Please note that we will ***initialize i = 0 and j = 0*** 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.
* 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***

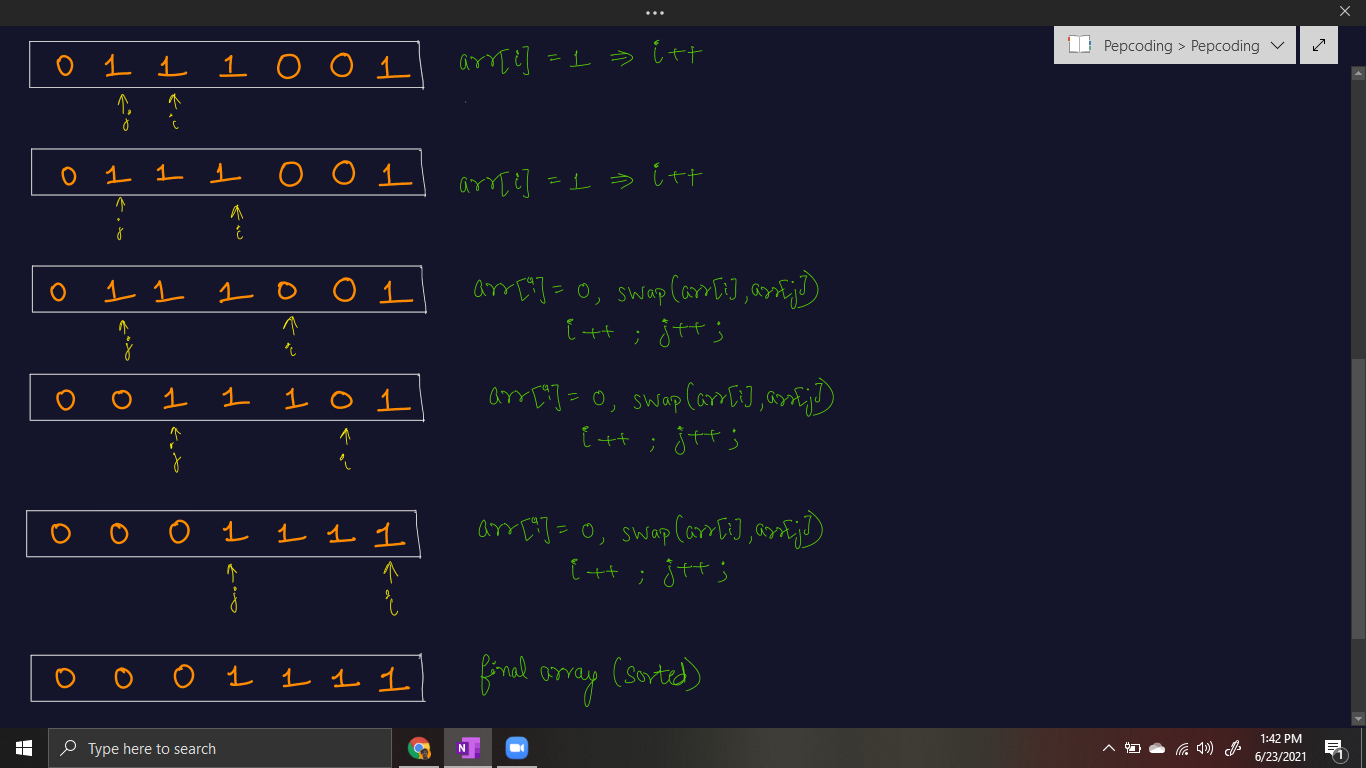
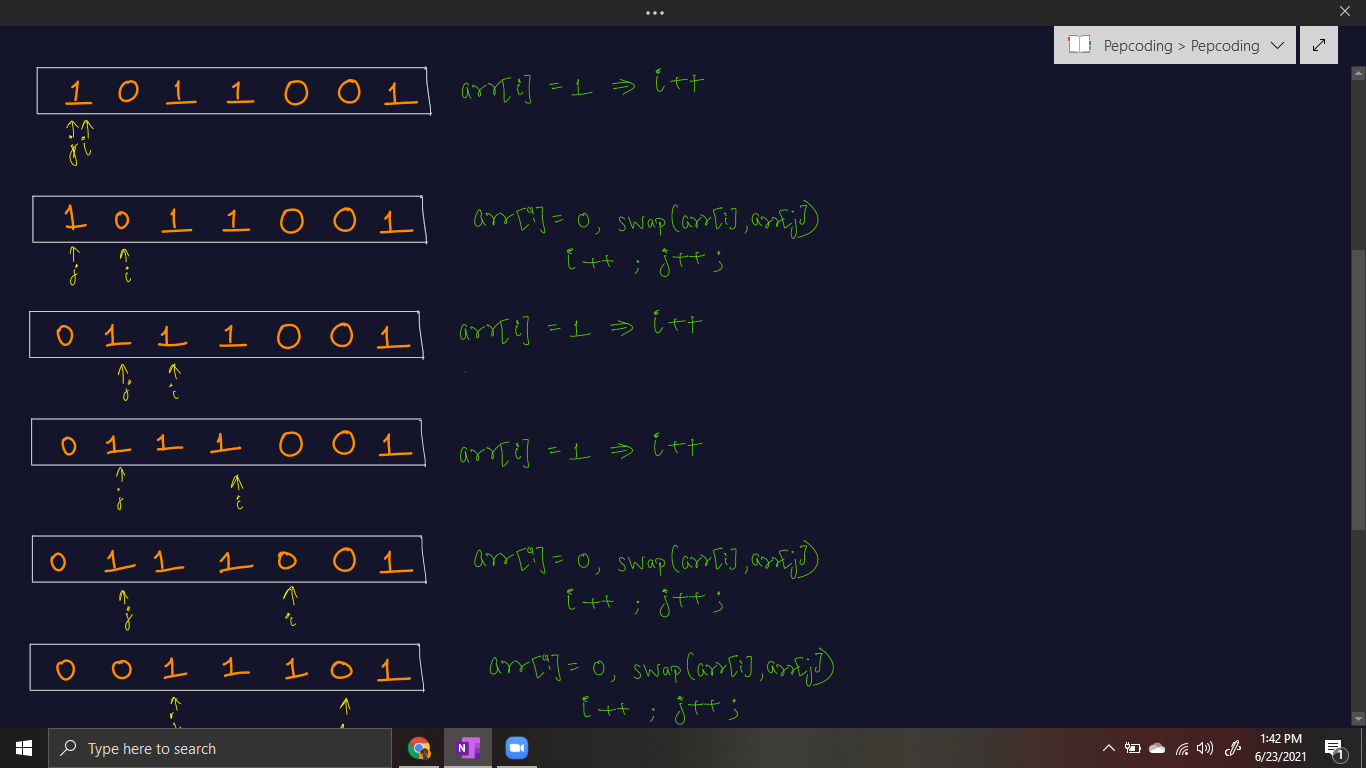
* If the first element in the unexplored array, i.e. arr[i] is equal to 0, then it should be added to the third segment of array of 0s.
* But it is not simple to add it to the first index, as we cannot add the element at any index of the array. Instead, we can shift the second segment (array of 1s) by one place to the right.
* But, shifting array elements in the second segment is an O(j - i) = O(n) task. Hence, 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.

Finally, when all the elements are explored from the unexplored array, i.e. the ***first segment contains 0 elements***, then the ***array is sorted***, as first the segment of 0s occurs and then the segment of 1s.

***Pseudo Code/ Algorithm***

* Maintain two indexes. Initialize first index *i* as 0 and second index *j* as 0 as well.
* Run the loop until i hits arr.length
  + If arr[i] = 1, then increment i.
  + Else (arr[i] = 0),
    - Swap values at arr[i] and arr[j]
    - Increment both i and j.

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



***Implementation (Java)***

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

import java.io.\*;

import java.util.\*;

public class Main {

public static void sort01(int[] arr){

// 0 to j-1 -> All Zeroes

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

// i to arr.length-1 -> All unknowns

int i = 0, j = 0;

while (i < arr.length) {

if (arr[i] == 0) {

swap(arr, i, j);

i++;

j++;

}else {

i++;

}

}

}

// 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();

}

sort01(arr);

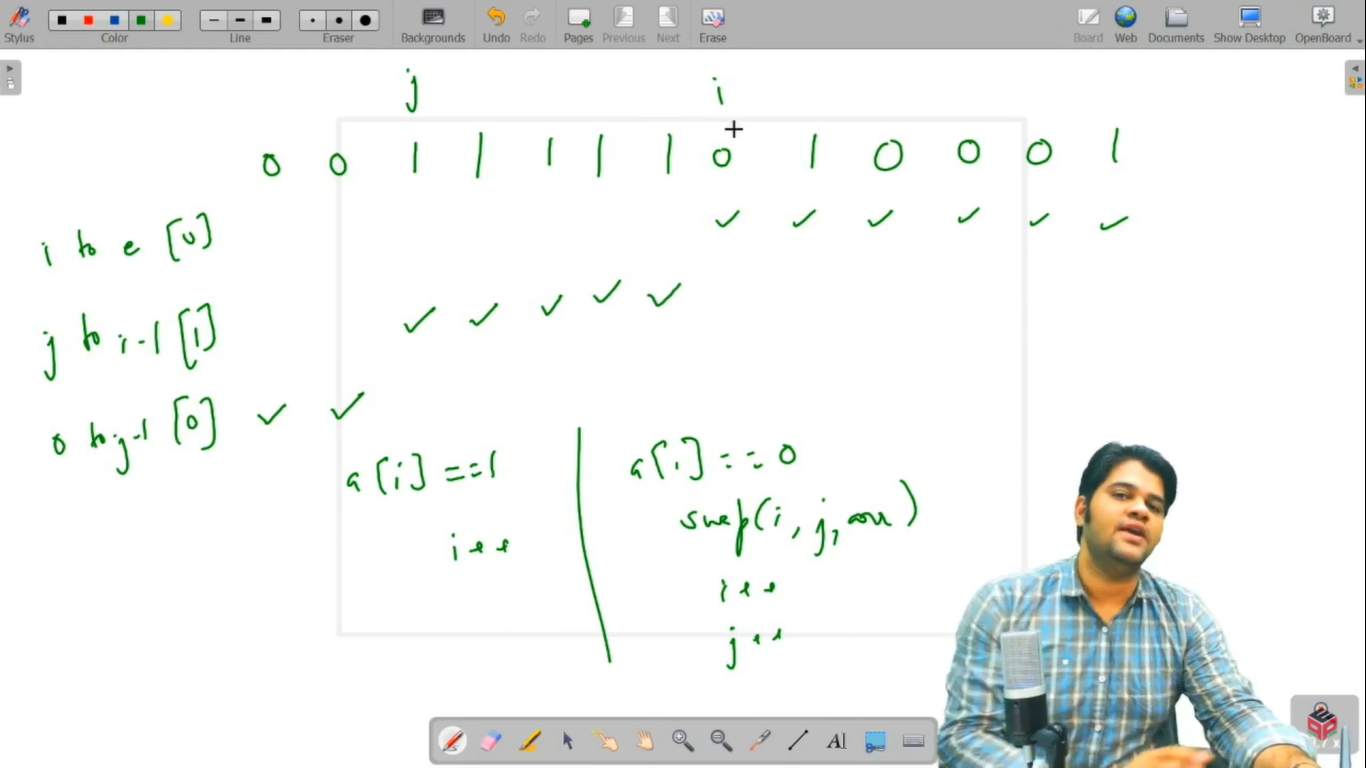
print(arr);

}

}

This code is written and explained by our team in [this video](https://www.youtube.com/watch?v=jFrUwjx4eoA) from *[6:52, 7:40]*. Please refer to it if you are stuck somewhere.

Dry run of the code is explained in the [same video](https://www.youtube.com/watch?v=jFrUwjx4eoA) from [0:25, 6:50], using an example test case.



***Time & Space Complexity Analysis***

Come on friend, Give it a try!

**Time Complexity** - It may seem that the time complexity is more than O(n^2) if you consider two pointers as equivalent to nested loops. But, please analyze carefully. We are reducing the unexplored segment by one element at each iteration. We either add it to the second or third segment. Hence time complexity is ***O(n)*** only.

You can also say that both the pointers i and j, move *independently* for a maximum of n steps, hence time is O(n) only.

**Space Complexity** - Since we are sorting the binary 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 and 1 like:

1. Segregate positive integers from negative integers.
2. Segregate odd integers from even integers
3. Segregate red color from blue color.
4. Sort array of Xs and Ys.

And so on, but the gist of the algorithm is the same, i.e. partition the array in 3 segments and solve using the concept of 2 pointers.

**Asked in Companies**: *Accolite, InfoEdge, Intuit, Paytm*

I hope you enjoyed solving the problem with me. I will see you in the next problem: ‘[***Sort 012’***](https://www.pepcoding.com/resources/online-java-foundation/time-and-space-complexity/sort-012-official/ojquestion), which is an extension to the current problem. Until then, *sayō nara*!

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