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

Problem Statement: Given an array consisting of only 0s, 1s, and 2s. Write a program to in-place sort the array without using inbuilt sort functions. (Expected: Single pass-O(N) and constant space)

▼ **Examples**

```
Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]

Input: nums = [2,0,1]
Output: [0,1,2]

Input: nums = [0]
Output: [0]
```

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Practice: Solve Problem

▼ Brute Force Approach

▼ Algorithm / Intuition ›

Sorting (even if it is not the expected solution here but it can be considered as one of the approaches). Since the array contains only 3 integers, 0, 1, and 2. Simply sorting the array would arrange the elements in increasing order.

▼ Complexity Analysis ›

Time Complexity: O(N*logN)

Space Complexity: O(1)

▼ Better Approach

▼ Algorithm / Intuition ›

Keeping count of values

Intuition: Since in this case there are only 3 distinct values in the array so it's easy to maintain the count of all, Like the count of 0, 1, and 2. This can be followed by overwriting the array based on the frequency(count) of the values.

Approach:

1. Take 3 variables to maintain the count of 0, 1 and 2.
2. Travel the array once and increment the corresponding counting variables

(let's consider **count_0 = a, count_1 = b, count_2 = c**)

3. In 2nd traversal of array, we will now over write the first 'a' indices / positions in array with '0', the next 'b' with '1' and the remaining 'c' with '2'.

▼ Code ›

```
def sortArray(arr):
    cnt0 = 0
    cnt1 = 0
    cnt2 = 0

    for num in arr:
        if num == 0:
            cnt0 += 1
        elif num == 1:
            cnt1 += 1
        else:
            cnt2 += 1

    for i in range(cnt0):
        arr[i] = 0

    for i in range(cnt0, cnt0 + cnt1):
        arr[i] = 1

    for i in range(cnt0 + cnt1, len(arr)):
        arr[i] = 2

n = 6
arr = [0, 2, 1, 2, 0, 1]
sortArray(arr)
print("After sorting:")
for num in arr:
    print(num, end=" ")
print()
```

Output: After sorting:
0 0 1 1 2 2

▼ Complexity Analysis ›

Time Complexity: O(N) + O(N), where N = size of the array. First O(N) for counting the number of 0's, 1's, 2's, and second O(N) for placing them correctly in the original array.

Space Complexity: O(1) as we are not using any extra space.

▼ Optimal Approach

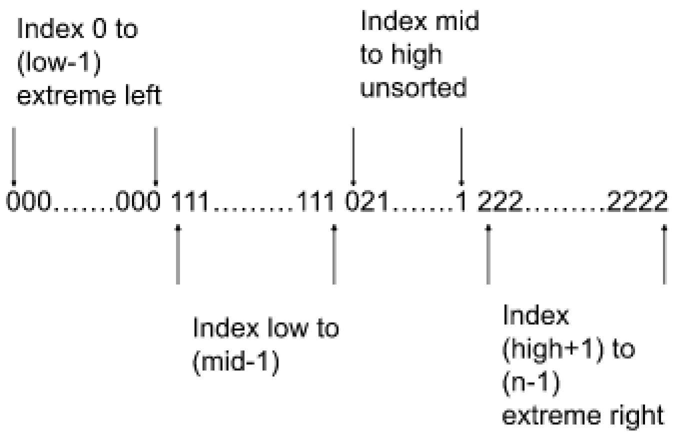
▼ Algorithm / Intuition ›

*This problem is a variation of the popular **Dutch National flag algorithm**.*

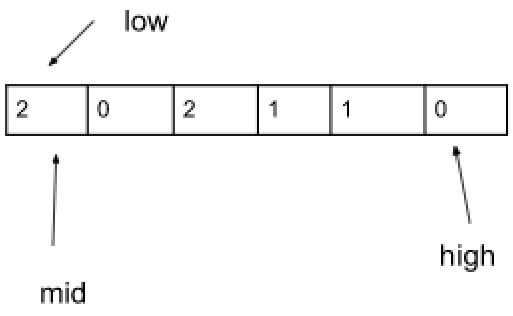
This algorithm contains 3 pointers i.e. low, mid, and high, and 3 main rules. The rules are the following:

- arr[0….low-1] contains 0. [Extreme left part]
- arr[low….mid-1] contains 1.
- arr[high+1….n-1] contains 2. [Extreme right part], n = size of the array

The middle part i.e. arr[mid….high] is the unsorted segment. So, hypothetically the array with different markers will look like the following:


Index 0 to (low-1) extreme left
Index mid to high unsorted
000…….000 111………111 021…….1 222………2222
Index low to (mid-1)
Index (high+1) to (n-1) extreme right

In our case, we can assume that the entire given array is unsorted and so we will place the pointers accordingly. For example, if the given array is: [2,0,2,1,1,0], the array with the 3 pointers will look like the following:



Here, as the entire array is unsorted, we have placed the mid pointer in the first index and the high pointer in the last index. The low is also pointing to the first index as we have no other index before 0. Here, we are mostly interested in placing the 'mid' pointer and the 'high' pointer as they represent the unsorted part in the hypothetical array.

Now, let's understand how the pointers will work to make the array sorted.

Approach:

Note: *Here in this tutorial we will work based on the value of the mid pointer.*

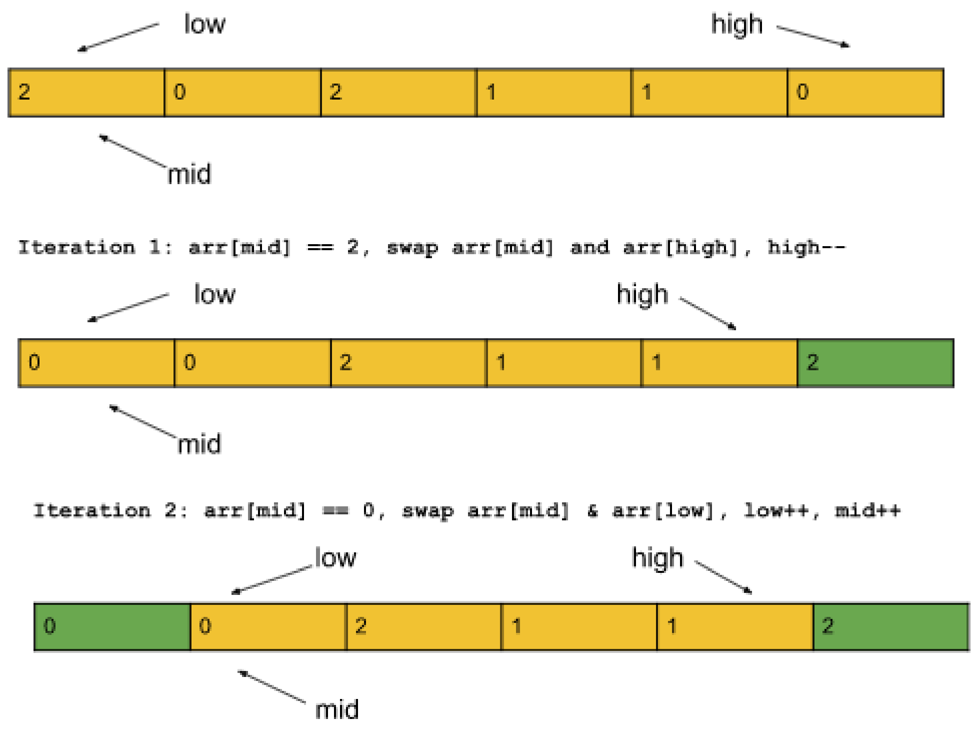
The steps will be the following:

1. First, we will run a loop that will continue until **mid <= high**.

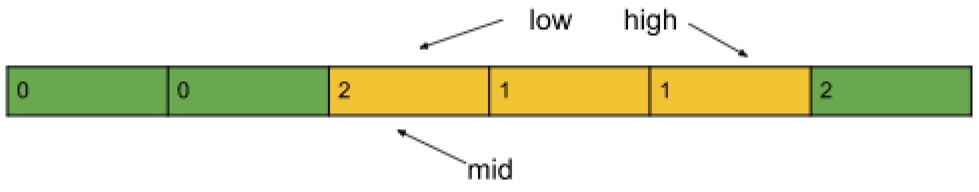
2. There can be three different values of mid pointer i.e. arr[mid]
    1. **If arr[mid] == 0,** we will swap arr[low] and arr[mid] and will increment both low and mid. Now the subarray from index 0 to (low-1) only contains 0.
    2. **If arr[mid] == 1,** we will just increment the mid pointer and then the index (mid-1) will point to 1 as it should according to the rules.
    3. **If arr[mid] == 2,** we will swap arr[mid] and arr[high] and will decrement high. Now the subarray from index high+1 to (n-1) only contains 2.
       In this step, we will do nothing to the mid-pointer as even after swapping, the subarray from mid to high(*after decrementing high*) might be unsorted. So, we will check the value of mid again in the next iteration.
3. Finally, our array should be sorted.

Dry Run:

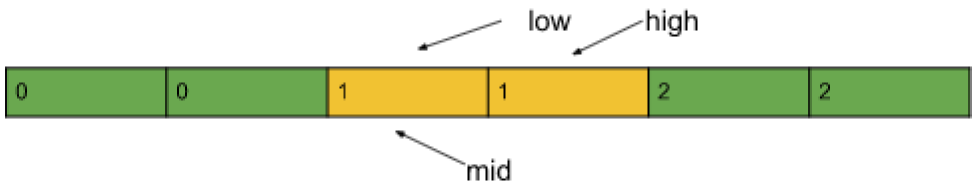
Assume the given array is [2,0,2,1,1,0]. The algorithm will be the following:


low
high
2
0
2
1
1
0
mid
Iteration 1: arr[mid] == 2, swap arr[mid] and arr[high], high--
low
high
0
0
2
1
1
2
mid
Iteration 2: arr[mid] == 0, swap arr[mid] & arr[low], low++, mid++
low
high
0
0
2
1
1
2
mid

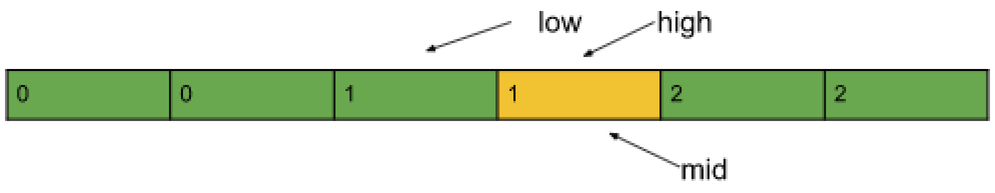
`Iteration 3: arr[mid] == 0, swap arr[low] & arr[mid], low++, mid++`



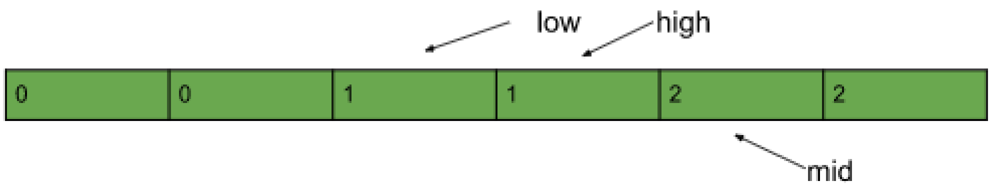
`Iteration 4: arr[mid] == 2, swap arr[high] & arr[mid], high--`



`Iteration 5: arr[mid] == 1, mid++`



`Iteration 6: arr[mid] == 1, mid++`



Now mid > high, loop will end.

In each iteration, if we check, the rules are always valid. This is how the algorithm works.

Note: *For a better understanding of intuition, please watch the video at the bottom of the page.*

▼ Code ›

```
def sortArray(arr):
    low = 0
    mid = 0
    high = len(arr) - 1

    while mid <= high:
        if arr[mid] == 0:
            arr[low], arr[mid] = arr[mid], arr[low]
            low += 1
            mid += 1
        elif arr[mid] == 1:
            mid += 1
        else:
            arr[mid], arr[high] = arr[high], arr[mid]
            high -= 1

n = 6
arr = [0, 2, 1, 2, 0, 1]
sortArray(arr)
print("After sorting:")
for num in arr:
    print(num, end=" ")
print()
```

Output: After sorting:
```
0 0 1 1 2 2
```

▼ Complexity Analysis ›

Time Complexity: O(N), where N = size of the given array.
Reason: We are using a single loop that can run at most N times.

Space Complexity: O(1) as we are not using any extra space.