2460. Apply Operations to an Array

You are given a **0-indexed** array `nums` of size `n` consisting of **non-negative** integers.

You need to apply `n - 1` operations to this array where, in the `ith` operation (**0-indexed**), you will apply the following on the `ith` element of `nums`:

- If `nums[i] == nums[i + 1]`, then multiply `nums[i]` by `2` and set `nums[i + 1]` to `0`. Otherwise, you skip this operation.

After performing **all** the operations, **shift** all the `0`'s to the **end** of the array.

- For example, the array `[1,0,2,0,0,1]` after shifting all its `0`'s to the end, is `[1,2,1,0,0,0]`.

Return *the resulting array*.

Note that the operations are applied **sequentially**, not all at once.

Example 1:

```
Input: nums = [1,2,2,1,1,0]
Output: [1,4,2,0,0,0]
Explanation: We do the following operations:
- i = 0: nums[0] and nums[1] are not equal, so we skip this operation.
- i = 1: nums[1] and nums[2] are equal, we multiply nums[1] by 2 and change nums[2]
to 0. The array becomes [1,4,0,1,1,0].
- i = 2: nums[2] and nums[3] are not equal, so we skip this operation.
- i = 3: nums[3] and nums[4] are equal, we multiply nums[3] by 2 and change nums[4]
to 0. The array becomes [1,4,0,2,0,0].
- i = 4: nums[4] and nums[5] are equal, we multiply nums[4] by 2 and change nums[5]
to 0. The array becomes [1,4,0,2,0,0].
After that, we shift the 0's to the end, which gives the array [1,4,2,0,0,0].
```

Example 2:

```
Input: nums = [0,1]
Output: [1,0]
Explanation: No operation can be applied, we just shift the 0 to the end.
```

Constraints:

- `2 <= nums.length <= 2000`
- `0 <= nums[i] <= 1000`

2460. Apply Operations to an Array

Overview

We are given an integer array `nums` consisting of `n` non-negative integers. We must iterate through the array, one step at a time, checking each pair of adjacent numbers starting from the first element:

- If two neighboring numbers are the same, we double the first number and turn the second one into 0.
- If they are different, we leave them as they are.

We repeat this process from left to right, one pair at a time. Finally, we must move all `0`s to the end of the array while preserving the order of non-zero elements and return the resulting array.

Because of the smaller constraints on the size of `nums` (`n ≤ 2000`), we can start from a brute force approach that simulates the rules mentioned in the problem and then think of further optimizing the approach.

2460. Apply Operations to an Array

```
/*
    Two passes: Simulation+fill answer
    Time complexity: O(n)
    Space complexity: O(n)
*/
class Solution {
public:
    std::vector<int> applyOperations(std::vector<int>& nums) {
        int n=nums.size();
         // Pass#1: Simulate the process
        for(int i=0;i<n-1;++i){
            if(nums[i]==nums[i+1]){
                nums[i]=2*nums[i];
                nums[i+1]=0;
            }
        }
       // Pass#2: Fill the answer array
       std::vector<int> ans(n);
        int l=0,r=n-1;
        for(int i=0;i<n;++i){
            if(nums[i]==0) ans[r--]=0;
            else ans[l++]=nums[i];
        }
        return ans;
    }
};
```

2460. Apply Operations to an Array

```
/*
    One pass: read/write pointers (in place)
    Time complexity: O(n)
    Space complexity: O(1)
*/
class Solution {
    public:
        std::vector<int> applyOperations(std::vector<int>& nums) {
            int n=nums.size();
            int w=0;
            for(int r=0;r<n;++r){
                if(r<n-1 && nums[r]==nums[r+1]){
                    nums[r]=2*nums[r];
                    nums[r+1]=0;
                }

                if(nums[r]!=0){
                    if(r!=w) std::swap(nums[r],nums[w]);
                    w++;
                }
            }

            return nums;
        }
};
```