26. Remove Duplicates from Sorted Array

```
给定一个排序数组，你需要在 原地 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。
不要使用额外的数组空间，你必须在 原地 修改输入数组 并在使用 O(1) 额外空间的条件下完成。
示例 1:
给定数组 nums = [1,1,2],
函数应该返回新的长度 2, 并且原数组 nums 的前两个元素被修改为 1, 2。
你不需要考虑数组中超出新长度后面的元素。
示例 2:
给定 nums = [0,0,1,1,1,2,2,3,3,4],
函数应该返回新的长度 5, 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。
你不需要考虑数组中超出新长度后面的元素。

说明:
为什么返回数值是整数，但输出的答案是数组呢?
请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。
你可以想象内部操作如下:
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);
// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度, 它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

Analysis

1.双指针(基本) O(n)

- 左指针指向第一个， 右指针从下一个开始遍历（遍历到尾即结束），在此过程中：
  - 如果值相等，就右指针一直往后移一个位置；
  - 如果值不等且指针相邻，则左，右指针都往后移一个位置；
  - 如果值不等且指针不相邻，则将左指针往后移一个位置，并且将这个位置的值改为右指针位置上的值，然后右指针往后移一个位置；

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.empty()) { return 0; }
        int i = 0;
        for(int j = 1; j < (int)nums.size();) {
```

```
            if(nums[i] == nums[j]) { j++; }
            else {
                if (j == i + 1) { i++; j++; }
                else {
                    nums[++i] = nums[j++];
                }
            }
        }
        nums.resize(i + 1);
        return nums.size();
    }
};
```

2.双指针(优化) O(n) – 推荐

再次分析执行过程，无论发生何种情况，在遍历的过程中右指针都会往后移一个位置，所以可以简化双指针判断逻辑：

- 左指针指向第一个， 右指针从下一个开始遍历（遍历到尾即结束），在此过程中：
  - 只要值不等，则左指针往后移一个位置；
    - 这时如果左指针和右指针所指向的位置不同，则将这个位置的值改为右指针位置上的值；
  - 每执行完一次判断，右指针都往后移动一个位置

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.empty()) { return 0; }
        int i = 0;
        for(int j = 1; j < (int)nums.size(); j++) {
            if (nums[i] != nums[j]) {
                i = i + 1;
                if(i != j) { nums[i] = nums[j]; }
            }
        }
        nums.resize(i + 1);
        return nums.size();
    }
};
```

3.双指针(快慢指针) O(n)：只关注不同的数 – 推荐

因为是排序数组，其实在遍历的过程中，只需要关注上一个位置的数和下一个位置的数不一样即可，所以：

- 慢指针指向第一个，**快指针也指向第一个，快指针用来遍历（遍历到倒数第二个即结束）**，在此过程中：
  - 如果快指针当前位置的数和下一个位置的数不一样，则将慢指针向后移动一个位置，并且将慢指针现在的位置的值改为**快指针下一个位置的值；**
  - 每判断一次，都往后移动一个位置；

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.empty()) { return 0; }
        int i = 0;
        for(int j = 0; j < (int)nums.size() - 1; j++) {
            if (nums[j] != nums[j+1]) {
                nums[++i] = nums[j+1];
            }
        }
        nums.resize(i + 1);
        return nums.size();
    }
};
```

4.慢指针+"range-based" 循环O(n):（非常巧妙，但晦涩，难理解）

- 首先让慢指针指向第一个位置，然后使用for each进行元素值循环：
  - 首次时，慢指针指向第一个位置，则元素值也在第一个位置，接着慢指针指向了第二个位置。
  - 至此之后，判断当前的数是否大于慢指针前面一个位置的数
    - 如果大于，则将慢指针位置上的值改为当前遍历到的值，且慢指针向后移动一个位置
    - 如果不大于（其实就是等于），则慢指针向后移动一个位置

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.empty()) { return 0; }
        int i = 0;
        for (int &n : nums) {
            if (!i || n > nums[i - 1]) {
                if (nums[i] != n) { nums[i] = n;}
                i++;
            }
        }
        nums.resize(i);
        return nums.size();
    }
};
```

5.使用重复数的个数 O(n) – 推荐

去重之后的数组元素个数 = 原个数 – 重复元素的个数，**而恰巧这个重复元素的个数又可以用来计算需要修改值的位置。**

- 重复元素个数repeated开始时置为0，使用单指针i从第二个元素位置开始遍历数组。
- 如果当前位置和前一个位置的值不相同并且重复数不为0，则将nums[i-repeated]的值改为nums[i]的值。
- 如果当前位置和前一个位置的值相同，将重复元素个数repeated加1。

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.empty()) { return 0; }
        int repeated = 0;
        for (int i = 1; i < (int)nums.size(); i++) {
            if (nums[i] != nums[i - 1]) {
                if (repeated) { nums[i - repeated] = nums[i]; }
            }
            else { repeated++; }
        }
        nums.resize((int)nums.size() - repeated);
        return nums.size();
    }
};
```

6.使用STL函数

- unique函数：
  - 该函数去除相邻的重复元素，因为需要相邻，所以必须是有序数组，或者先使用了sort函数排好序。
  - 该函数并没有删除任何元素，而是将无重复的元素复制到序列的前段，从而覆盖相邻的重复元素。
  - 函数迭代器指向超出无重复的元素范围末端的下一个位置。
- erase函数：删除指定范围内的元素

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.empty()) { return 0; }
        nums.erase(std::unique(nums.begin(), nums.end()), nums.end());
        return nums.size();
    }
};
```

Code

```
#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
using namespace std;
//two pointers(basic) O(n)
// int removeDuplicates(vector<int>& nums) {
//     if(nums.empty()) { return 0; }
//     int i = 0;
//     for(int j = 1; j < (int)nums.size();) {
//         if(nums[i] == nums[j]) { j++; }
//         else {
//             if (j == i + 1) { i++; j++; }
```

```
//             else {
//                 nums[++i] = nums[j++];
//             }
//         }
//     }
//     nums.resize(i + 1);
//     return nums.size();
// }
//two pointers(optimazing) O(n)
// int removeDuplicates(vector<int>& nums) {
//     if(nums.empty()) { return 0; }
//     int i = 0;
//     for(int j = 1; j < (int)nums.size(); j++) {
//         if (nums[i] != nums[j]) {
//             i = i + 1;
//             if(i != j) { nums[i] = nums[j]; }
//         }
//     }
//     nums.resize(i + 1);
//     return nums.size();
// }
//fast and slow pointer: Only focus on diff O(n)
// int removeDuplicates(vector<int>& nums) {
//     if(nums.empty()) { return 0; }
//     int i = 0;
//     for(int j = 0; j < (int)nums.size() - 1; j++) {
//         if (nums[j] != nums[j+1]) {
//             nums[++i] = nums[j+1];
//         }
//     }
//     nums.resize(i + 1);
//     return nums.size();
// }
//slow pointer and "range-based" loops O(n)
// int removeDuplicates(vector<int>& nums) {
//     if(nums.empty()) { return 0; }
//     int i = 0;
//     for (int &n : nums) {
//         if (!i || n > nums[i - 1]) {
//             if (nums[i] != n) { nums[i] = n;}
//             i++;
//         }
//     }
//     nums.resize(i);
//     return nums.size();
// }
//number of calculation intervals O(n)
// int removeDuplicates(vector<int>& nums) {
//     if(nums.empty()) { return 0; }
```

```
//     int repeated = 0;
//     for (int i = 1; i < (int)nums.size(); i++) {
//         if (nums[i] != nums[i - 1]) {
//             if (repeated) { nums[i - repeated] = nums[i]; }
//         }
//         else { repeated++; }
//     }
//     nums.resize((int)nums.size() - repeated);
//     return nums.size();
// }
//stl
int removeDuplicates(vector<int>& nums) {
    if(nums.empty()) { return 0; }
    nums.erase(std::unique(nums.begin(), nums.end()), nums.end());
    return nums.size();
}
int main() {
    // int nums[] = {1,2};
    // std::vector<int> array(nums, nums + 2);

    // int nums[] = {1,1,2};
    // std::vector<int> array(nums, nums + 3);
    int nums[] = {0,0,1,1,1,2,2,3,3,4};
    std::vector<int> array(nums, nums + 10);
    std::cout << removeDuplicates(array) << std::endl;
    //print
    std::cout << "[";
    for (int i = 0; i < (int)array.size(); i++) {
        std::cout << array[i];
        if (i != (int)array.size() - 1) {
            std::cout << ",";
        }
    }
    std::cout << "]" <<std::endl;
    return 0;
}
```