## 一、数组去重

#### 1.理解问题

设计一个算法将一个可能包含重复元素的序列去重，使得每个元素最多出现一次，分析你的算法。

本质上是一个数组去重的问题。

输入：依次输入数组长度n，数组中的每个元素

例如：10 3 4 0 0 9 1 6 9 3 3

输出：0 1 3 4 6 9 （排好序的去重序列）

#### **2.**算法设计

首先想到利用先排序再去重，想到可以通过改变mergesort合并成新数组的这个小步骤来干预新数组的生成，通过加上一些处理条件来进行去重。时间性能会比暴力套两层循环好。

对merge过程的修改：增加一个mergeLast函数对最后一轮merge过程进行修改，从而达到去重的效果：

1. 增加一个检验左/右数组右连续相等值的去重操作：

当arr[p1]==arr[p1-1]时，p1++;不去管重复的数，用while循环，因为可能右多个连着的重复的数；

1. 增加对左右数组拷贝时，左右值相等的去重操作：

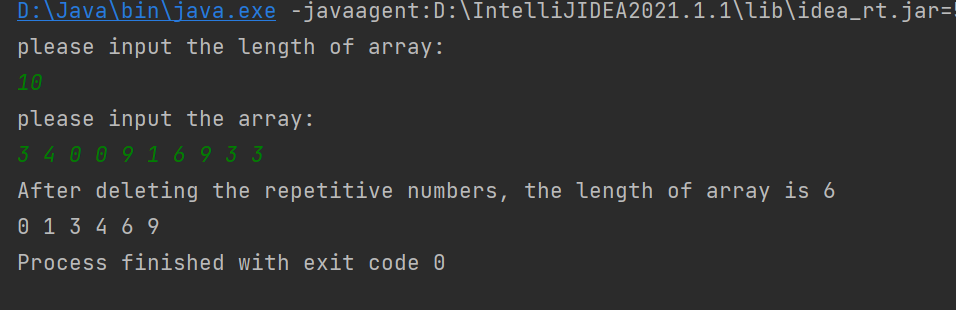
当arr[p1]==arr[p2]时，拷贝p1，p1++,p2++。

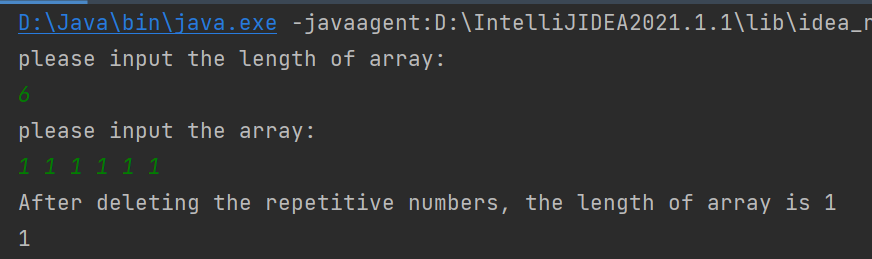
因为经过1的处理，p1和p2都是它们的数组中最后一个指向这个数的指针，所以需要拷贝一个，然后两边的指针都++。

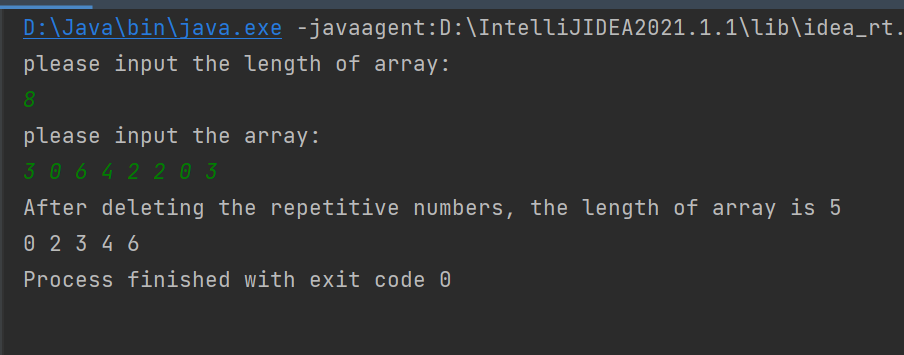
#### 算法代码与分析

package Test;  
  
import java.util.Scanner;  
  
public class test {  
 /\*  
 符合master公式  
 a=2 b=2 d=1  
 T(N) = 2\*T(N/2) + O(N)  
 Z最终时间复杂度 O(N\*logN)  
 额外空间复杂度 O（N）  
 \*/  
   
 public static int *trueL*; //记录真正的L  
 public static int *trueR*; //记录真正的R，方便比较mergeLast的条件  
 public static int *size*; //去重后数组的长度  
  
 public static void mergeSort(int[] arr) {  
 if (arr == null || arr.length < 2) {  
 return;  
 }  
 *process*(arr, 0, arr.length - 1);  
 }  
  
 public static void process(int[] arr, int L, int R) {  
 if (L == R) return;  
 int mid = L + ((R-L) >> 1);  
 *process*(arr, L, mid);  
 *process*(arr, mid + 1, R);  
 if (*trueL* == L && *trueR* == R) {  
 *mergeLast*(arr, L, mid, R);  
 } else {  
 *merge*(arr, L, mid, R);  
 }  
 }  
  
 public static void mergeLast(int[] arr, int L, int M, int R) {  
 int[] help = new int[R - L + 1];  
 int i = 0;  
 int p1 = L;  
 int p2 = M + 1;  
 //用一个while将所有的套起来，保证p1 p2更新的时候重新都指向相同元素的最后一个  
 //判断边界只能用<= 为了将所有元素都拷贝进去  
 while (p1 <= M && p2 <= R) {  
 //p1 p2指向相同元素的最后一个   
 //和arr[p1+1]判断可以解决0处的特殊情况，但是最后一个边界需要p1<M来判断  
 // 首先考虑边界情况 再考虑是否相等  
 while (p1<M && arr[p1] == arr[p1 + 1] ) p1++;   
 while (p2<R && arr[p2] == arr[p2 + 1] ) p2++;  
 //merge的拷贝过程 加一个相等的处理  
 if (arr[p1] == arr[p2]) {  
 help[i++] = arr[p1++];  
 p2++;  
 } else {  
 help[i++] = arr[p1] < arr[p2] ? arr[p1++] : arr[p2++];  
 }  
 }  
  
 //可能左右两边会有剩下的未处理的  
 while (p1<=M){  
 // 防止数组越界访问  
 if ( p1<M && arr[p1]==arr[p1+1]){  
 p1++;  
 continue;  
 }  
 help[i++]=arr[p1++];  
 }  
 while (p2<=R){  
 if ( p2<R && arr[p2]==arr[p2+1]){  
 p2++;  
 continue;  
 }  
 help[i++]=arr[p2++];  
 }  
   
 //拷贝回去  
 for (int j = 0; j < i; j++) {  
 arr[L+j]=help[j]; //一定注意这里用的是L+j  
 }  
 *size* = i;//i就是最终数组的长度  
 }  
  
 // O(N)  
 public static void merge(int[] arr, int L, int M, int R) {  
 //辅助空间  
 int[] help = new int[R - L + 1];  
 int i = 0;  
 int p1 = L;  
 int p2 = M + 1;  
 while (p1 <= M && p2 <= R) {  
 help[i++] = arr[p1] < arr[p2] ? arr[p1++] : arr[p2++];  
 }  
 while (p1 <= M) {  
 help[i++] = arr[p1++]; //一开始又嵌套了一个for循环 错了 直接用p1  
 }  
 while (p2 <= R) {  
 help[i++] = arr[p2++];  
 }  
 for (int i1 = 0; i1 < help.length; i1++) {  
 arr[L + i1] = help[i1];  
 }  
 }  
  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("please input the length of array:");  
 int n = sc.nextInt();  
 System.*out*.println("please input the array:");  
 int[] arr = new int[n];  
 for (int i = 0; i < arr.length; i++) {  
 arr[i] = sc.nextInt();  
 }  
 *trueL* = 0;  
 *trueR* = n - 1;  
 *mergeSort*(arr);  
 System.*out*.println("After deleting the repetitive numbers, the length of array is " + *size*);  
 for (int i = 0; i < *size*; i++) {  
 System.*out*.print(arr[i] + " ");  
 }  
 }  
}

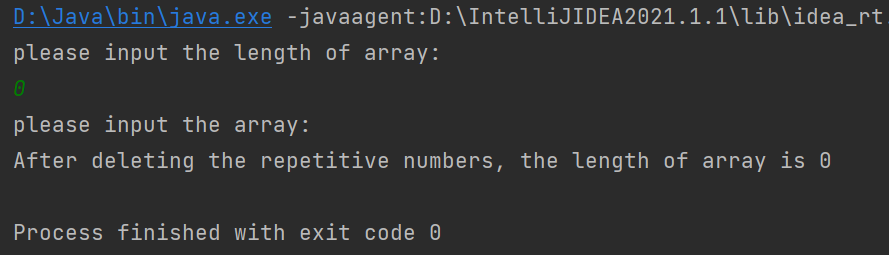
测试数据：







特殊情况：数组为空



#### 性能分析

由于只是在mergeSort的最后一步merge过程加上一些简单的处理，所以整体时间复杂度和空间复杂度和归并排序保持一致，时间复杂度为O（nlogn），空间复杂度为O（N）。

## 二、最大跨距

#### 1.理解问题

假设一个整数序列，设计算法求序列的最大跨距d=j-i，使得aj-ai最大，其中j>i，分析你的算法。

输入：数组长度n，数组中的每一个元素

例如：10 3 4 0 0 9 1 6 9 3 3

输出：5

#### **2.**算法设计

首先想到最小的开销也要把所有数字都看一遍，线性的时间复杂度。

从前向后遍历，设置四个变量，min记录经过的元素中最小的数值，minIndex记录它的下标，maxDiff记录经过的元素和min的差值的最大值，maxIndex记录它的下标。遍历一遍即可得到。

注意：因为最小值想找最左边的，所以判断条件为

if (arr[i]<min)

等于的时候不更新；

因为最大值想找最右边的，所以判断条件为

if(arr[i]-min>=maxDiff)

等于的时候更新。

#### 3.算法代码与分析

#include <stdio.h>

int main(){

    int length;

    printf("Enter the length of the array: ");

    scanf("%d", &length);

    //检查length是否合法

    if(length<1){

        printf("Error: The length of the array should be greater than 0.\n");

        return 0;

    }else if (length==1)

    {

        printf("The max distance is 0\n");

        return 0;

    }

    int arr[100];

    printf("Enter the elements of the array: ");

    for (int i = 0; i < length; i++)

    {

        scanf("%d", &arr[i]);

    }

    int min=arr[0];

    int minIndex=0;

    int maxDiff=0;

    int maxIndex=0;

    for (int i = 1; i < length; i++)

    {

        //希望min找最靠左的，所以=时候不更新

        if (arr[i]<min)

        {

            min=arr[i];

            minIndex=i;

        }

        //希望max找最右的，所以=时候更新

        if(arr[i]-min>=maxDiff)

        {

            maxDiff=arr[i]-min;

            maxIndex=i;

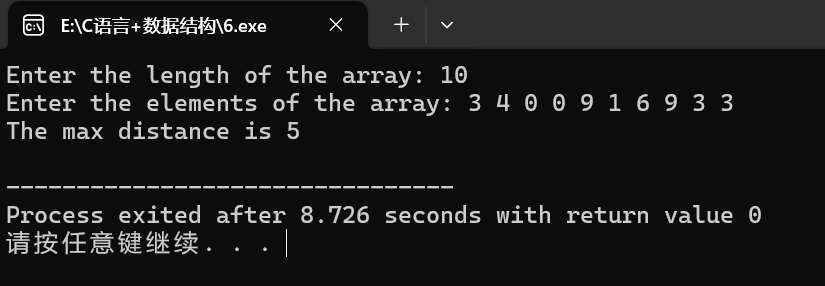
        }

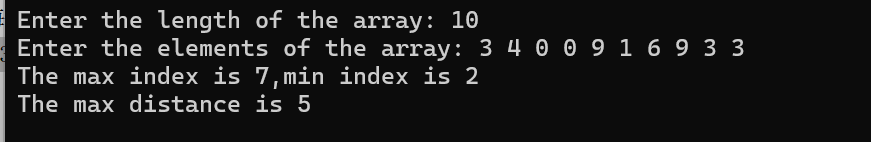
    }

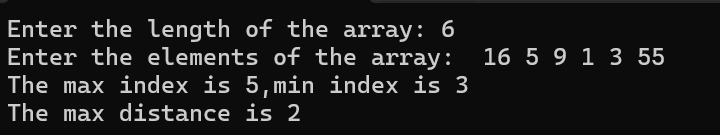
    printf("The max distance is %d\n", maxIndex-minIndex);

}

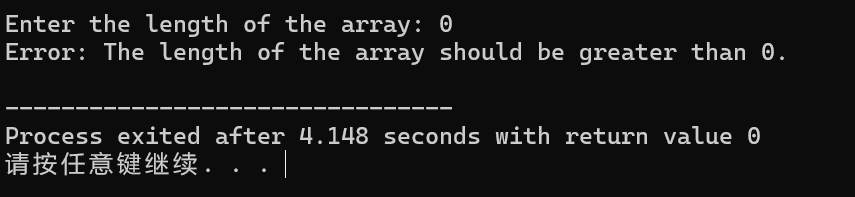
测试结果：

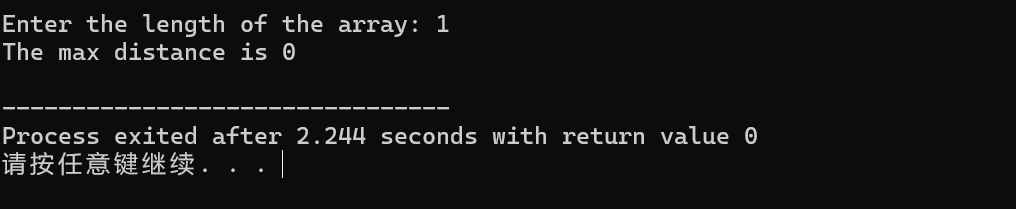






边界情况：





#### 4.性能分析

时间复杂度是线性的，从前到后扫描了一遍，为O（N）；

使用了4个额外的变量，空间复杂度为O（1）；