# 排序

经典的十大排序算法！



## 冒泡排序

### 执行流程

1.从头开始比较每一对相邻元素，如果第1个比第2个大，就交换它们的位置执行完一轮后，最未尾那个元素就是最大的元素。

2.忽略1中曾经找到的最大元素，重复执行步骤1，直到全部元素有序。

### 代码

交换数组两个元素

|  |
| --- |
| private void swap(int[] arr,int i ,int j){  int tmp = arr[i];  arr[i] = arr[j];  arr[j] = tmp; } |

* 普通版冒泡

|  |
| --- |
| @Test public void bubble1(){  for (int i = arr.length-1; i > 0 ; i--) {  for(int j = 0;j< i;j++){  if(arr[j]>arr[j+1]){  swap(arr,j,j+1);  }  }  } }  性能：  稳定性：true 耗时：3.512s(3512ms) 比较：2.00亿 交换：9985.77万 |

* 冒泡排序优化1

在某一轮完整遍历结束后，发现没有产生交换那么意味着全部有序，后面就轮就不用再执行了。

|  |
| --- |
| *//优化* @Test public void bubble2(){   for (int i = arr.length-1; i > 0 ; i--) {  int exchange = 0;  for(int j = 0;j< i;j++){  if(arr[j]>arr[j+1]){  swap(arr,j,j+1);  exchange ++;  }  }  if(exchange == 0) break;  } }  性能：  稳定性：true 耗时：3.287s(3287ms) 比较：2.00亿 交换：9935.99万 |

* 冒泡排序优化2

如果序列尾部已经局部有序，可以记录最后1次交换的位置，减少比较次数。

|  |
| --- |
| public void bubble3(){  *//假设最后一次交换位置为1，也就是假设全部有序  //如果产生交换就更新交换位置，记录最大的交换位置  //然后下一次只交换到此位置-1的位置* for (int i = arr.length-1; i > 0 ; i--) {  int end = 1;  for(int j = 0;j< i;j++){  if(arr[j]>arr[j+1]){  swap(arr,j,j+1);  end=j+1;  }  }  i = end;  } }  性能：  稳定性：true 耗时：3.679s(3679ms) 比较：2.00亿 交换：1.00亿 |

### 额外补充

冒泡排序又叫做起泡排序；大的不断地从左边往右边跑，因此向冒泡一样

## 选择排序

每一轮中，从无序序列中选出最大的元素，放在有序序列的第一个位置。

|  |
| --- |
| public void select1(){  for (int i = arr.length -1; i > 0; i--) {  int maxIndex = 0;  for(int j = 1;j<=i;j++){  if(arr[j] > arr[maxIndex]){  maxIndex = j;  }  }  swap(arr,maxIndex,i);  } } |

冒泡排序的交换次数远大于选择排序，因此选择排序性能要更好

## 堆排序

使用堆来选择最大值，可以大大提高选择速度。因此堆排序可以看作是对选择排序的一种优化。

### 执行流程：

① 对要排序的序列进行原地建堆（heapify）

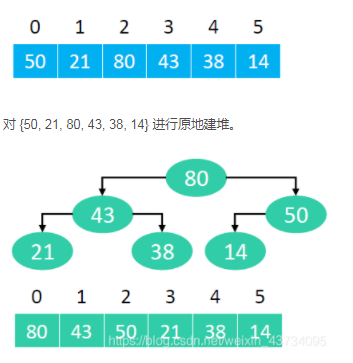
② 重复执行以下操作，直到堆的元素数量为 1

交换堆顶元素与堆尾元素（把最大值放到最后面）

堆的元素数量减 1（不管最后已经放到最后的最大值）

对 0 位置进行 1 次 siftDown 操作

### 堆排序图解

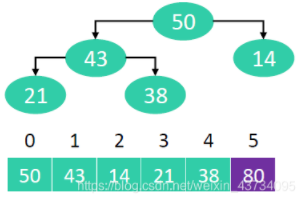


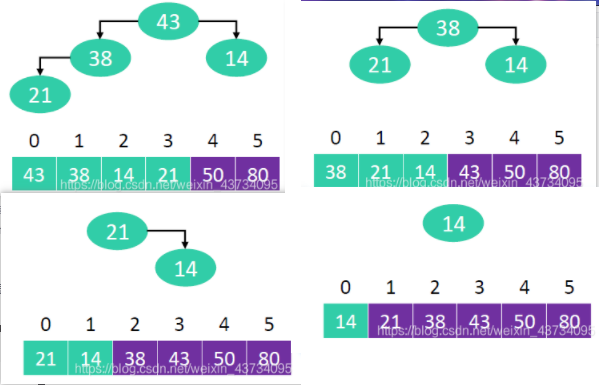
重复执行以下操作，直到堆的元素数量为1：

交换堆顶元素与尾元素

堆的元素数量减 1

对 0 位置进行 1 次 siftDown 操作





### 堆排序实现

原地建堆：****自下而上的下滤建堆****。

|  |
| --- |
| public class \_03\_HeapSort<T extends Comparable<T>> extends Sort<T> {  private int heapSize; *// 堆大小* @Override  protected void sort() {  *// 原地建堆（自下而上的下滤）* heapSize = array.length;  for (int i = (heapSize >> 1) - 1; i >= 0; i--) {  siftDown(i); //log(n)  } //nlog(n)  //n  while (heapSize > 1) {  *// 交换堆顶元素和尾部元素* swap(0, --heapSize);   *// 对0位置进行siftDown（恢复堆的性质）* siftDown(0);  }  **// n + n\*log(n)**  }    private void siftDown(int index) {  T element = array[index];   int half = heapSize >> 1;  while (index < half) { *// index必须是非叶子节点  // 默认是左边跟父节点比* int childIndex = (index << 1) + 1;  T child = array[childIndex];   int rightIndex = childIndex + 1;  *// 右子节点要存在, 并且比左子节点大* if (rightIndex < heapSize &&  cmp(array[rightIndex], child) > 0) {  child = array[childIndex = rightIndex];  }   *// 大于等于子节点* if (cmp(element, child) >= 0) break;   array[index] = child;  index = childIndex;  }  array[index] = element;  }  } |

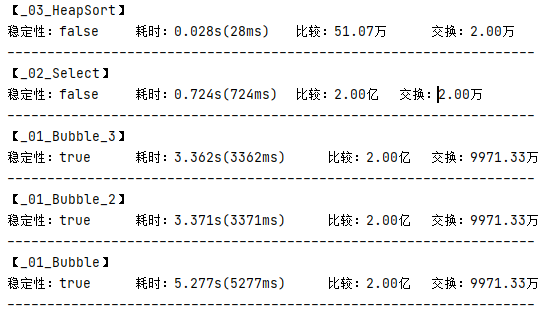
### 复杂度与稳定性

原地建堆：O(nlogn)

最好、最坏、平均时间复杂度：O(nlogn)

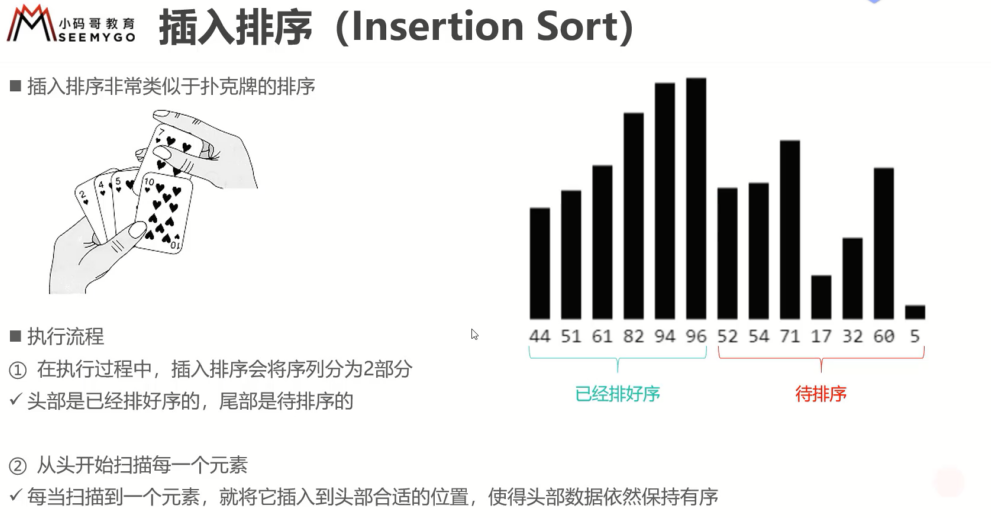
空间复杂度：O(1)

堆排序属于不稳定排序, 因为假设arr[0] = arr[1]，一下子就把arr[0]换到最后去了，直接破坏了稳定性。



## 插入排序

* **原理：**



* **代码如下:**

|  |
| --- |
| public static void sort(int[] arr){  for(int i = 1;i<arr.length;i++){  for(int j = i;j>0;j--){  if(arr[j]<arr[j-1]){  int tmp = arr[j];  arr[j] = arr[j-1];  arr[j-1] = tmp;  }else{  break;  }   }  } } |

* **性能**

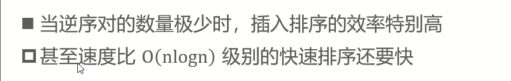
**稳定性：** cmp <= 0 不稳定 <0稳定

**时间复杂度：**

平均时间复杂度 O(n2)

最坏：O(n2)

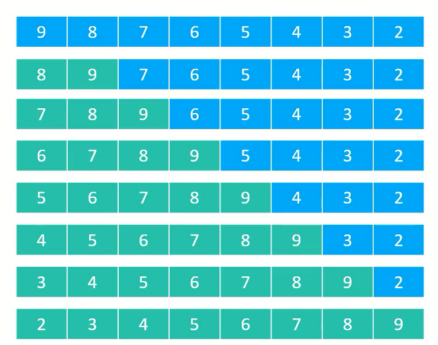
最好O(n) 仅做比较，不做交换



时间复杂度和逆序对个数有关；

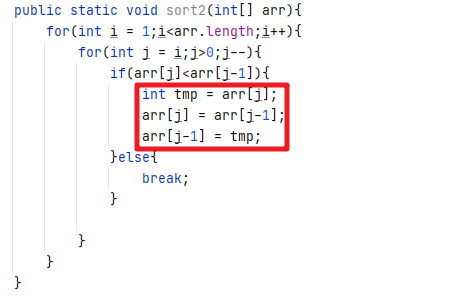


当逆序对最多的时候，时间复杂度为：1+2+3+...+n = O(n2)



空间复杂度：无额外空间 O(n)

### 优化1：将交换改为挪动



假设有序序列中大于arr[i]的元素有n个，那么交换为3行代码 3倍O(n)，如果是挪动：o(n)级别

|  |
| --- |
| *//优化：挪动* public static void sort2(int[] arr){  for(int i = 1;i<arr.length;i++){  int tmp = arr[i];  int cur=i-1; while(cur>=0 && arr[cur]>tmp){  arr[cur+1]=arr[cur];  cur--;  }  *//cur<0 或者 arr[cur]<= tmp* arr[cur+1] = tmp;  } } |

逆序对越多，优化效果越明显。

### 优化2：二分搜索

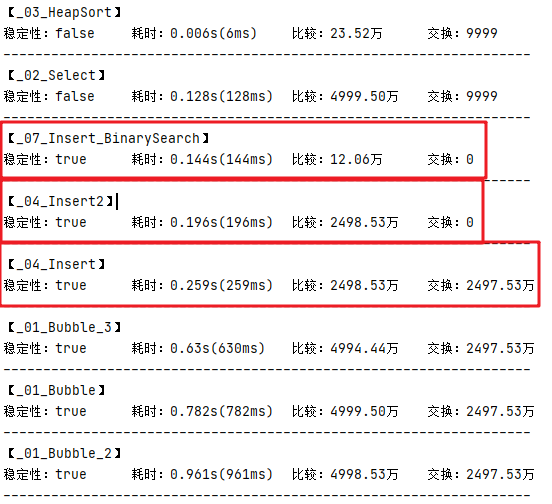
我们要寻找有序序列中，arr[i]应该插入的位置；

如果找到了，就插在i

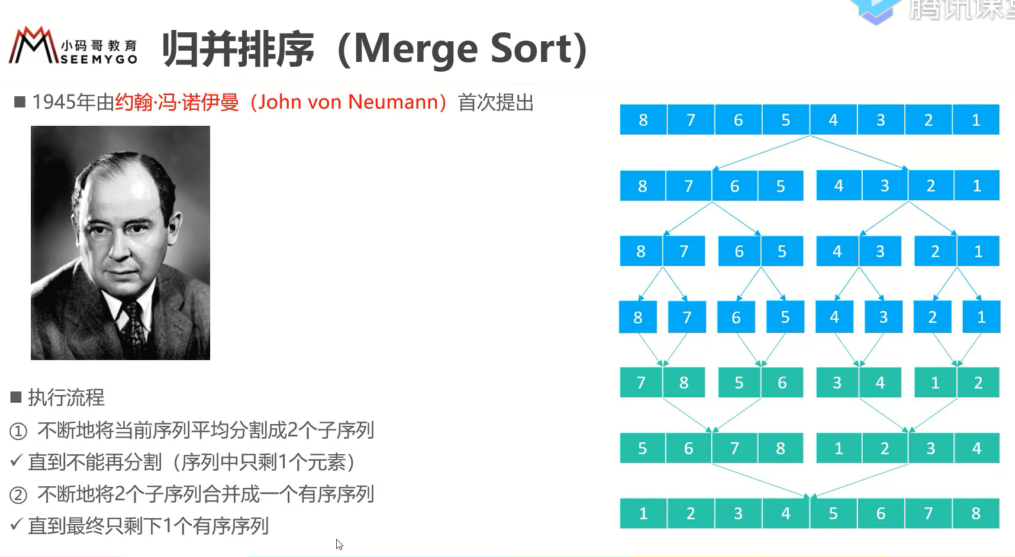
如果没找到，在最后的位置，

|  |
| --- |
| package \_01\_sort.cmp;  import \_01\_sort.Sort;  public class \_07\_Insert\_BinarySearch<T extends Comparable<T>> extends Sort<T> {   private int searchIndex(T[]array,T v){  return searchIndex(array,v,0,array.length-1);  }    private int searchIndex(T[] array,T v,int low,int high){  if(array == null || array.length == 0) return -1;   int mid;  while(high > low){  mid = (high + low )>>1;  if(array[mid] == v){  return mid;  }else if(cmp(array[mid] ,v)>0){  high = mid - 1;  }else{  low = mid + 1;  }  }  *//到这里说明 high = low* if(cmp(array[low],v)>0){ *//如果这里改为>= 就不是稳定排序了* return low;  }else{  return low+1;  }  }   @Override  protected void sort() {  for (int i = 1; i < array.length; i++) {  *//[0,i)是有序区域 arr[i] 是待插元素* T tmp = array[i];  int insertIndex = searchIndex(array, tmp,0,i-1); *//logN  //将[insertIndex,i-1]范围内的有序元素 往后移动1位* for(int j = i;j>insertIndex;j--){  array[j] = array[j-1];  }*//n  //将元素插进去* array[insertIndex] = tmp;  }  }  *// n\*(n+logN) => 因此时间复杂度还是O(n平方) 只不过优化了搜索 索引的位置* } |
| package \_01\_sort.cmp;  import \_01\_sort.Sort;  public class \_07\_Insert\_BinarySearch<T extends Comparable<T>> extends Sort<T> {   private int searchIndex(T[]array,T v){  return searchIndex(array,v,0,array.length-1);  }    private int searchIndex(T[] array,T v,int low,int high){  if(array == null || array.length == 0) return -1;   int mid;  while(high > low){  mid = (high + low )>>1;  if(array[mid] == v){  return mid;  }else if(cmp(array[mid] ,v)>0){  high = mid - 1;  }else{  low = mid + 1;  }  }  *//到这里说明 high = low* if(cmp(array[low],v)>0){ *//如果这里改为>= 就不是稳定排序了* return low;  }else{  return low+1;  }  }   @Override  protected void sort() {  for (int i = 1; i < array.length; i++) {  *//[0,i)是有序区域 arr[i] 是待插元素* T tmp = array[i];  int insertIndex = searchIndex(array, tmp,0,i-1); *//logN  //将[insertIndex,i-1]范围内的有序元素 往后移动1位* for(int j = i;j>insertIndex;j--){  array[j] = array[j-1];  }*//n  //将元素插进去* array[insertIndex] = tmp;  }  }  *// n\*(n+logN) => 因此时间复杂度还是O(n平方) 只不过优化了搜索 索引的位置* } |

时间复杂度：还是O(n2)



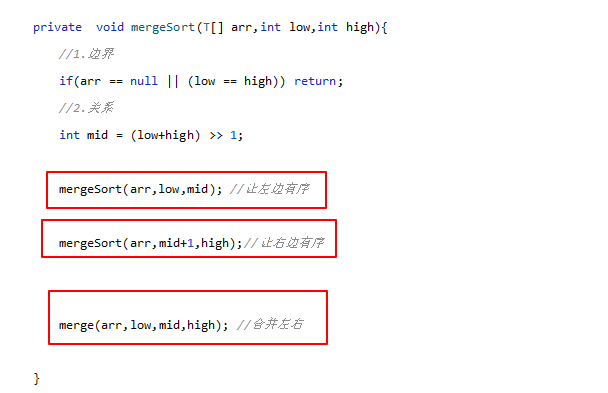
## 归并排序



* 代码

|  |
| --- |
| public class \_08\_MergeSort<T extends Comparable<T>> extends Sort<T> {    private void mergeSort(T[] arr,int low,int high){  *//1.边界* if(arr == null || (low == high)) return;  *//2.关系* int mid = (low+high) >> 1;   mergeSort(arr,low,mid); *//让左边有序* mergeSort(arr,mid+1,high);*//让右边有序* merge(arr,low,mid,high); *//合并左右* }   private Object[] SubList(T[] arr,int low,int high){  Object[] lowhigh = new Object[high-low+1];  int x = 0;  for(int i = low;i<=high;i++){  lowhigh[x] = arr[i];  x++;  }  return lowhigh;  }    private void merge(T[] arr,int low,int mid,int high){   Object[] leftArray = SubList(arr, low, mid);   int li=0;  int ri = mid+1;  int ai = low;  while(ai<=high){  if(li<leftArray.length && (ri > high || cmp((T)leftArray[li],arr[ri])<=0) ){  arr[ai] = (T)leftArray[li];  ai++;  li++;  }else{  arr[ai] = arr[ri];  ai++;  ri++;  }  }  }    @Override  protected void sort() {  mergeSort(array,0,array.length-1);  } } |

复杂度分析：



T(n) = T(n/2)\*2 + O(n)

T(n/2) = T(n/4)\*2 + O(n/2)

T(n) = 4\*T(n/4) + O(n)+2\*O(n/2)

T(n) = 4\*T(n/4) + 2O(n)

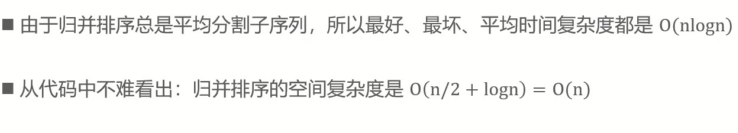
T(1) = 1

T(n) = 2k \* T(n/2K) + K\*O(n) n = 2k

T(2k) = 2k \* T(1) + K\*O(2k)

T(n) = n + logN\*O(n) => **O(nlogN)**

**空间复杂度：**



是稳定排序

## 6.快速排序

也是nlogn,但是比归并和堆排序要快

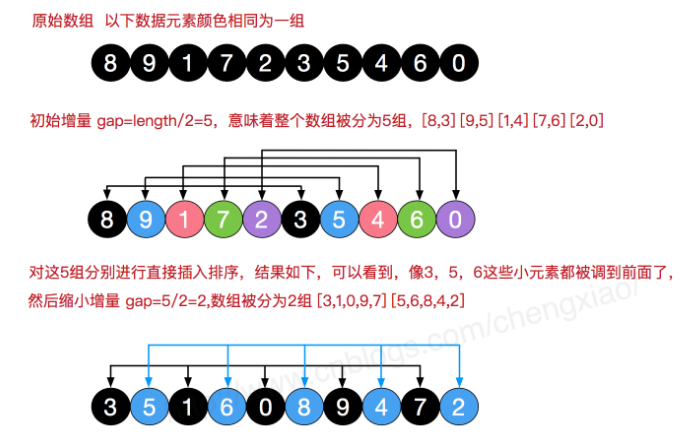
### 执行过程

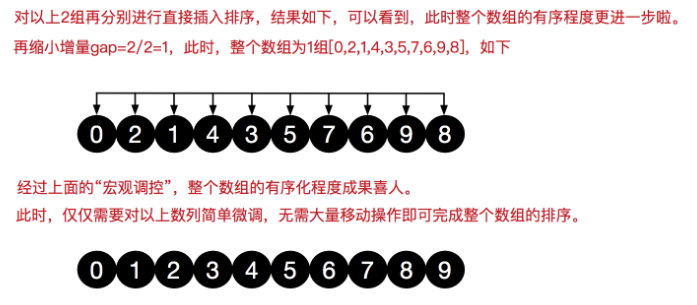


当每一个元素变为轴点元素的时候，array就有序了

# 7.希尔排序

　希尔排序是希尔（Donald Shell）于1959年提出的一种排序算法。希尔排序也是一种插入排序，它是简单插入排序经过改进之后的一个更高效的版本，也称为缩小增量排序，同时该算法是冲破O(n2）的第一批算法之一。本文会以图解的方式详细介绍希尔排序的基本思想及其代码实现。





# 计数排序

* 之前学习的冒泡、选择、堆、插入、归并、快速、希尔排序都是基于比较的排序。

平均时间复杂度目前最低为O(nlogn)

* 计数排序、桶排序、基数排序，都不是基于比较的排序。

它们是典型的用空间换时间，在某些时候，平均时间复杂度可以比 O(nlogn) 更低

* 计数排序于1954年由Harold H. Seward提出，适合 **对一定范围内的整数** 进行排序。
* 计数排序的核心思想：

统计每个整数在序列中出现的次数，进而推导出每个整数在有序序列中的索引

## 最简单的实现



新建一个数组，数组大小为arr中最大元素的值

将arr的元素 映射到 数组的index中

数组的index的元素对应 index 在arr中出现的次数，从而可以知道排序后的样子

这个版本的实现存在以下问题

* 无法对负整数进行排序
* 极其浪费内存空间
* 是个不稳定的排序

## 改进



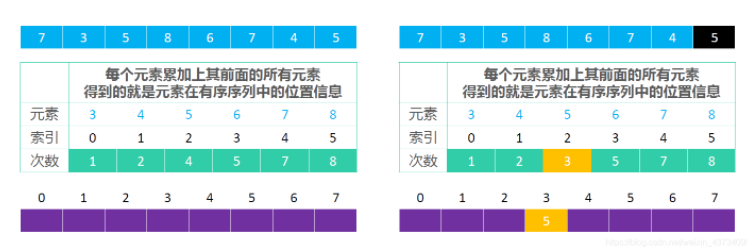
改进1：减小count数组的大小 获取max和min，min对应索引0

改进2：index里放置的不再是index元素出现的次数，而是前面所有元素包括当前元素出现次数之和



图解过程：

蓝色数组为原始乱序数组，从右往左找到元素在排好序的数组中的对应的位置



第一步：创建一个数组counts，其长度= max-min+1 也就是8-3+1=6

第二步：counts进行初始化，其index为元素值，index对应的数据为 元素出现的次数+ 有序的情况下前面的元素的个数

第三步：**无序数组从后往前开始排序（从右往前排序能够保证稳定性）**

5 => counts[5-3] = counts[2] => 总次数=4 => arr[4-1] = 5

当当前元素已经放到排好序的数组中的位置后，将counts[2]中的出现次数-1(-1表示下一次再次遇到这个元素的时候，该元素的位置往前调一格，保证稳定性)

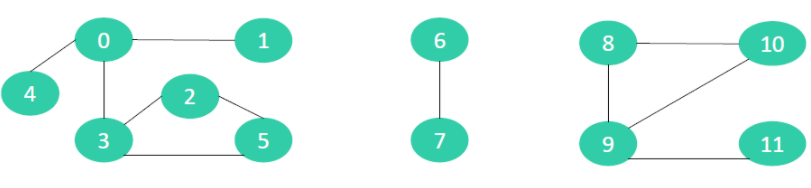


第四步：同样的方法，对4进行排序

# 并查集

## 需求分析

* 假设有 n 个村庄，有些村庄之间有连接的路，有些村庄之间并没有连接的路



设计一个数据结构，能够快速执行 2 个操作：

（1）查询 2 个村庄之间是否有连接的路

（2）连接 2 个村庄

* 使用数组、链表、平衡二叉树、集合(Set) 都可以完成需求：

一个集合代表一个村庄

1. 查询2个村庄之间是否有连接的路：

假设用数组来实现：查询数组1是否包含数组2的元素；时间复杂度：O(n)

链表，也是O(n)

平衡二叉树：由于村庄之间没有大小之分，因此也是O(n)

集合：一个set一个村庄，由于底层是hash表，查询是O(1),但是受村庄个数影响，n个村庄的话，就是查n个村庄是否有这个元素也是O(n)

1. 连接两个村庄：

就是将集合1和集合2进行合并，对于动态数组来说，时间复杂度也是O(n)

* 并查集能做到查询、连接的均摊时间复杂度都是 O(α(n))，α(n) < 5，非常适合解决这类“连接”相关的问题。

并查集能够实现这两个操作都在O(5)级别

## 并查集

* 并查集也叫作不相交集合（Disjoint Set）
* 并查集有2个核心操作:

1. 查找（Find）：查找元素所在的集合

(这里的集合并不是特指Set这种数据结构，是指广义的数据集合)

1. 合并（Union）：将两个元素所在的集合合并为一个集合

* 有 2 种常见的实现思路：

1. Quick Find

查找（Find）的时间复杂度：O(1)

合并（Union）的时间复杂度：O(n)

1. Quick Union

查找（Find）的时间复杂度：O(logn), 可以优化至 O(α(n)), α(n) < 5

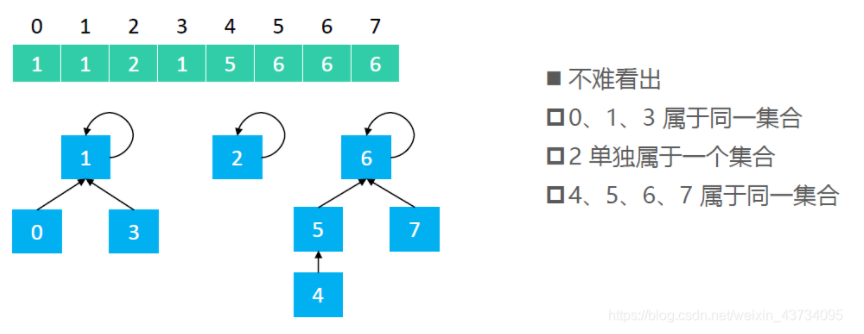
合并（Union）的时间复杂度：O(logn), 可以优化至 O(α(n)), α(n) < 5

## **如何存储数据？**

假设并查集处理的数据都是整型，那么可以用整型数组来存储数据。

* 数组索引代表村庄
* 索引对应的值代表村庄所在的集合

将｛0，1，2，3，4，5，6，7｝存储到数组中，如下图：



属于同一个集合的村庄是有相连的路的；

将索引中的元素看作是索引的父节点

* 因此，并查集是可以用数组实现的树形结构（二叉堆、优先级队列也是可以用数组实现的树形结构）

## **接口定义**

查找v所属的集合(根结点)

public abstract int find(int v);

合并v1、v2所在的集合

public abstract void union(int v1, int v2);

检查v1、v2是否属于同一集合

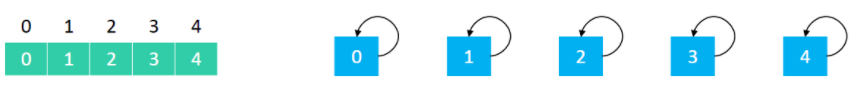
public boolean isSame(int v1, int v2);

isSame() 的实现十分简单：

public boolean isSame(int v1, int v2){ return find(v1) == find(v2); }

## **元素的初始化**

初始化时，每个元素各自属于一个****单元素集合****

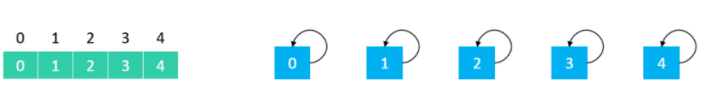


# **Quick Find**

Quick Find 的 union(v1, v2)：让 ****v1 所在集合的所有元素****都指向 ****v2 的根节点****。(这意味着将村庄v1放进v2所属的集合中，连接两个村庄)  
 并且 Quick Find 的高度永远保持 <= 2。

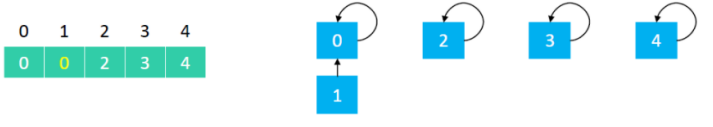
## **union 示例及实现**

例如：  
将｛0，1，2，3，4，5｝初始化为并查集，每个元素各自属于一个单元素集合：{0}, {1}, {2}, {3}, {4} 。

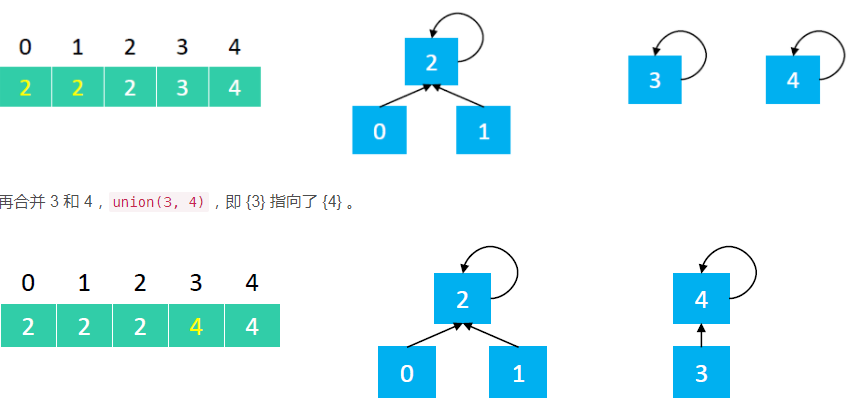


规定Union(V1,V2) 的规则是：将V1村庄，放进V2的集合中：

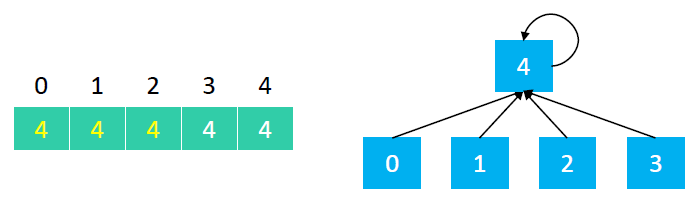
* Union(0,1):将0和1放进集合0，因此1的元素变为0



* 同理，union(1, 2)，1 所在集合有 {0, 1}，即 {0, 1} 指向了 2 。



* union(0, 3)，0 所在集合为 {0, 1, 2}，3 所在集合为 {3,4}，如下：



Union(3,2):

