# 排序

## 分类



### 内排序/外排序

所谓的内排序是指所有的数据已经读入**内存**，在内存中进行排序的算法。排序过程中不需要对磁盘进行读写。同时，内排序也一般假定所有用到的辅助空间也可以直接存在于内存中。与之对应地，另一类排序称作外排序，即内存中无法保存全部数据，需要进行**磁盘**访问，每次读入部分数据到内存进行排序。

外排序是指处理超过内存限度的数据的排序算法，通常将中间结果放在读写较慢的外存储器（通常是硬盘）上。外排序常采用“排序-归并”策略。

排序阶段，读入能放在内存中的数据量，将其排序输出到临时文件，依次进行，将待排序数据组织为多个有序的临时文件。

归并排序，将这些临时文件组合为大的有序文件。

下面给出它们的具体分类：

**内部排序算法：**插入排序（直接插入排序、希尔排序），选择排序（直接选择排序、堆排序），交换排序（冒泡排序、快速排序），归并排序，基数排序。

**外部排序算法：**计数排序、桶排序等。

注：原地排序（冒泡排序，选择排序，插入排序），非原地排序（快速排序，归并排序）。

### 线性时间比较类排序/线性时间非比较类排序

**非线性时间比较类排序：**通过比较来决定元素间的相对次序，由于其时间复杂度不能突破O(nlogn)，因此称为非线性时间比较类排序。

**线性时间非比较类排序：**不通过比较来决定元素间的相对次序，它可以突破基于比较排序的时间下界，以线性时间运行，因此称为线性时间非比较类排序。比如计数排序，桶排序，基数排序。

## 性能

### 稳定性

稳定性：如果i=j，排序前i在j的前面，排序后i仍然在j的前面，即相等的两个数字的相对位置在排序前后不变，则该算法是稳定的，否则不稳定。

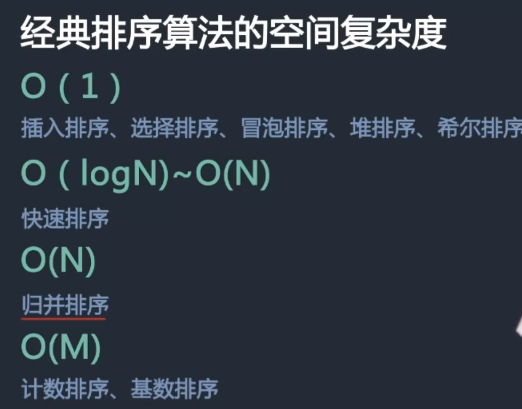
排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，前一个键排序的结果可以为后一个键排序所用。可能比较难理解，这里再举个例子方便理解，比如在基数排序中，先将低位排序，再逐次按高位排序，稳定的话就可以保证排序后低位元素的顺序在高位相同时是不会改变的。

### 时间复杂度

时间复杂度：指执行算法所需要的工作量，即对待排序数据的总操作次数，我们用它来描述算法的运行时间。

### 空间复杂度

空间复杂度：指执行算法所需的内存空间。



## 总结

排序算法无绝对优势：

通常不能随便说哪种排序算法好。这个和要排序的元素相关。例如对人的年龄排序或者身高排序，因为这种数据范围通常比较小，可以考虑采用计数排序。但是对于均匀分布的整数，计数排序就不合适了。除非特别说明，否则认为要排序的数据范围是均匀分布的。

为什么叫快速排序？

快速排序之所以叫快速排序，并不代表它比堆排序和归并排序优良。在最好情况下，它的渐进复杂度与堆排序和归并排序是相同的。只是快速排序的常量系数比较小而已。

工程上的排序：

1、工程上的排序是综合排序；

2、数组较小时，插入排序；

3、数组较大时，快速排序或其他O(N\*logN)的排序。

## 数据结构

在排序算法中，常用顺序表数据结构表示待排序的数据，其定义如下：

#define MAXSIZE 100 /\*用于要排序数组个数最大值，可根据需要修改\*/

typedef struct

{

int data[MAXSIZE+1]; /\*存储要排序数组，data[0]用作哨兵或临时变量\*/

int length; /\*用于记录顺序表的长度\*/

}SqList;

另外，排序操作中经常用到数组元素的交换，将这些操作封装为函数，如下所示：

/\*交换L中数组data的下标为i和j的值\*/

void swap(SqList \*L, int i, int j)

{

int temp = L->data[i];

L->data[i] = L->data[j];

L->data[j] = temp;

}

# 直接插入排序

直接插入排序是一种最为简单的排序方法，因此也被称为简单插入排序。

## 基本思想

对于未排序元素，在已排序序列中从后向前扫描，找到相应位置把它插入进去；在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为新元素提供插入空间。

## 具体步骤

1、从第一个元素开始，默认该元素已被排好序；

2、取出下一个元素，在已经排序的元素序列中从后向前扫描；

3、如果该元素（已排序）大于新元素，将该元素移到下一位置；

4、重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；

5、将新元素插入到该位置后；

6、重复步骤2~5。

## 代码实现

### C语言

**版本1：**

void InsertSort(int k[], int n)

{

int i,j,temp;

for(i=1;i<n;i++)

{

if(k[i]<k[i-1])

{

temp = k[i];

for(j=i-1;k[j]>temp;j--)

{

k[j+1] = k[j];

}

k[j+1] = temp;

}

}

}

**版本2：**

#include <iostream>

using namespace std;

//插入排序 （思想：在一个已经排序好的序列中，为下一个元素找合适的插入位置）

void InsertSort(int\* array,int n)

{

int i,j;

int temp;

//从第二个元素开始进行插入排序 为其在前面排序好的序列中找合适的位置

for(i=1;i<n;i++)

{

//为当前元素选择合适的位置

temp = array[i];

for(j=i-1;j>=0;j--)

if(array[j] > temp)

array[j+1] = array[j];

else

break;

//找到合适的位置后填充当前值

array[j+1] = temp;

}

for(i=0;i<n;i++)

cout<<array[i]<<endl;

}

int main()

{

int array[]={7,5,2,9,1,3};

InsertSort(array,6);

return 0;

}

### Python

## 性能分析

**稳定性：**稳定

**时间复杂度：**

平均情况：O(n^2)

最好情况：O(n)

最坏情况：O(n^2)

**空间复杂度：**O(1)

# 希尔排序

希尔排序其实是插入排序的改良算法，其关键在于步长的选择。

## 基本思想

## 具体步骤

## 代码实现

### C语言

void ShellSort(int k[], int n)

{

int i,j,temp;

int gap = n;

do

{

gap = gap/3 + 1;

for(i=gap;i<n;i++)

{

if(k[i]<k[i-gap])

{

temp = k[i];

for(j=i-gap;k[j]>temp;j-=gap)

{

k[j+gap] = k[j];

}

k[j+gap] = temp;

}

}

}while(gap>1)

}

### Python

## 性能分析

**稳定性：不稳定**

**时间复杂度：**

**平均情况：O(n^1.25)**

最好情况：

最坏情况：

**空间复杂度：**O(1)

# 简单选择排序

## 基本思想

先在待排序列表中找到**最小（大）**的元素，把它放在起始位置作为已排序序列；然后，再从剩余待排序序列中找到最小（大）的元素放在已排序序列的末尾，以此类推，直至完毕。

这个有点像暴力解决的意思，所以它的**效率不高**（最坏、最好、平均都一样很差）。但是，其比较和元素移动次数要比冒泡排序少，效率略高。

## 具体步骤

1、初始状态整个待排序序列为无序序列，有序序列为空；

2、每次遍历无序序列将最小元素交换到有序序列之后；

3、n-1趟遍历后排序完成。

## 代码实现

### C语言

#include<stdio.h>

void SelectSort(int k[], int n)

{

int i,j,min,temp;

for(i=0;i<n-1;i++)

{

min = i;

for( j=i+1; j<n;j++)

{

min = i;

for(j=i+1;j<n;j++)

{

if(k[j] < k[min])

{

min = j;

}

}

if( min != i)

{

temp = k[min];

k[min] = k[i];

k[i] = temp;

}

}

}

}

### Python

function selectionSort(arr) {

var len = arr.length;

var minIndex, temp;

for (var i = 0; i < len - 1; i++) {

minIndex = i;

for (var j = i + 1; j < len; j++) {

if (arr[j] < arr[minIndex]) { // 寻找最小的数

minIndex = j; // 将最小数的索引保存

}

}

temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

return arr;

}

## 性能分析

**稳定性：**不稳定

**时间复杂度：**

**平均情况：O(n^2)**

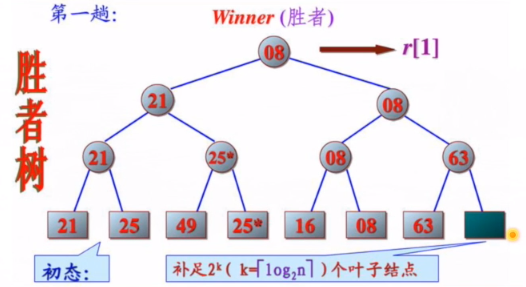
最好情况：O(n^2)

最坏情况：O(n^2)

注：比较和移动次数要比冒泡排序少。

**空间复杂度：**O(1)

# 锦标赛排序



# 堆排序

## 基本思想

堆的结构相当于一个完全二叉树，最大堆满足下面的性质：父结点的值总大于它的孩子结点的值。

注：如果构造从小到大的排序，使用大顶堆。将根节点元素与最后一个元素交换，则最后一个就是最大值，其双亲结点就是倒数第二大元素。

堆排序实际上是一种**快速选择排序**。

## 具体步骤

1、将待排序列表构造成一个**最大堆**，作为初始无序堆（即初始无序列表）；

2、将堆顶元素（最大值）与堆尾元素互换；

3、将该堆（无序区）尺寸缩小1，并对缩小后的堆重新调整为最大堆形式；

4、重复上述步骤，直至堆（无序区）的尺寸变为1，此时排序完成。

注：建堆的时间复杂度为O(N)，调整堆的时间复杂度为O(nlogn)，总体复杂度O(nlogn)。

## 代码实现

### C语言

#include <stdio.h>

#include <stdlib.h>

//调整堆

void HeapAdjust(int\* array,int s,int length)

{

int temp = array[s];//待调整的元素

int child = 2\*s+1;//待调整节点的左孩子的位置

//开始调整

while(child < length)

{

//如果右孩子大于左孩子，找到比当前待调整节点大的孩子节点

if(child+1 < length&& array[child] < array[child+1])

child++;

//如果较大的孩子大于待调整的节点

if(array[s] < array[child])

{

array[s] = array[child];//那么较大的节点向上移

s = child;//更新待调整节点的位置

child = 2\*s+1;//更新待调整节点的左孩子

}

else

break;

array[s] = temp;

}

}

//堆排序

void HeapSort(int\* array,int length)

{

int i;

//构造最大堆

for(i=(length-1)/2;i>=0;i--)

{

HeapAdjust(array,i,length);

}

//从最后一个元素开始对序列进行调整

for(i=length-1;i>0;i--)

{

//交换堆顶元素和堆中最后一个元素

int tmp = array[i];

array[i] = array[0];

array[0] = tmp;

//每次交换元素之后，就需要重新调整堆

HeapAdjust(array,0,i);

}

}

int main()

{

int array[]={3,4,5,1,9,8,6,2,7,10};

HeapSort(array,10);

int i;

for(i=0;i<10;i++)

printf("%d\n",array[i]);

return 0;

}

或：

void swap(int k[], int i, int j)

{

int temp;

temp = k[i];

k[i] = k[j];

k[j] = temp;

}

void HeapAdjust(int k[], int s, int n)

{

int i,temp;

temp = k[s]; //存储当前需要处理的双亲结点

for(i=2\*s;i<=n;i\*=2) //对于完全二叉树，2\*s是左子树，2\*s+1是右子树

//i\*=2表示指向下一个双亲

{

if(i<n && k[i]<k[i+1]) //k[i]是左孩子,k[i+1]是右孩子

{

i++;

}

if(temp>=k[i])

{

break;

}

k[s] = k[i];

s= i;

}

k[s] = temp;

}

//构造大顶堆

void HeapSort(int k[], int n)

{

int i;

for(i=n/2;i>0;i--)

//从低开始，即编号为n/2的节点开始构造

{

HeapAdjust(k,i,n); //(待处理数组,当前双亲结点,长度)

}

//调整位置

for(i=n;i>1;i--)

{

swap(k,1,i); //第一个元素与最后一个元素互换

HeapAdjust(k,1,i-1);

//根节点与最后一个元素互换后根节点不是最大的，需要调整

}

}

### Python

## 性能分析

**稳定性：**不稳定

**时间复杂度：**

**平均情况：O(nlogn)**

最好情况：O(nlogn)

最坏情况：O(nlogn)

堆排序效率很好，所以大家一定要会熟练将堆排序应用到各种场景中～

**空间复杂度：**O(1)

# 冒泡排序

## 基本思想

冒泡排序（Bubble Sort）是一种**交换排序**，基本思想是：两两比较相邻记录的关键字，如果反序则交换，直到没有反序的记录为止。因为每遍历一次列表，最大（或最小）的元素会经过交换一点点“浮”到列表的一端（顶端），所以形象的称这个算法为冒泡算法。

冒泡排序的原理是：**每一趟只能确定将一个数归位**。即如果有n个数进行排序，只需将n-1个数归位，也就是说要进行n-1趟操作。

注：因为每次只能确定一个数的位置，确定整个数组的顺序需要多趟比较，因此其效率相对低。

## 具体步骤

1、比较两个相邻元素，如果前一个比后一个大，则交换这两个相邻元素；

2、从头至尾对每一对相邻元素进行步骤1的操作，完成1次对整个待排序数字列表的遍历后，最大的元素就放在了该列表的最后一个位置上了；

3、对除最后一个元素的所有元素重复上述步骤，这第二次遍历后第二大的元素就也放在了正确的位置（整个列表的倒数第二位置上）；

4、不断重复上述步骤，每次遍历都会将一个元素放在正确的位置上，从而下次遍历的元素也会随之减少一个，直至没有任何一对数字需要比较。

## 实现动图

## 代码实现

### C语言

#### 基本版

**版本1：**

/\*对顺序表L作交换排序（冒泡排序初级版）\*/

void BubbleSort0(SqList \*L)

{

int i,j;

for(i=1;i<L->length;i++)

{

for(j=i+1;j<=L->length;j++)

//这里是data[i]与data[i+1]->data[L->length]都比较，不是相邻的两两比较

{

if(L->data[i]>L-data[j])

{

swap(L.i.j); /\*交换L->data[i]和L->data[j]的值\*/

}

}

}

}

这段代码严格讲，不算是标准的冒泡排序算法，因为它不满足“两两比较相邻记录”的冒泡排序思想，应该算是最简单的交换排序算法。它的思想是让每一个关键字，都和它后面的每一个关键字进行比较，如果大则交换，这样第一位置的关键字在一次循环后就变成最小值。但是这种方法效率比较低下，采用如下的改进算法：

/\*对顺序表L作冒泡排序\*/

void BubbleSort(SqList \*L)

{

int i,j;

for(i=1;i<L->length;i++)

{

for(j=L->length;j>=i;j--) /\*注意j是从后往前循环\*/

{

if(L->data[j]>L->data[j+1])

/\*若前者大于后者（注意这里与上一算法差异）\*/

/\*算法核心部分是双重嵌套循环，因此时间复杂度是O(N2)\*/

{

swap(L,j,j+1);

}

}

}

}

对上述算法还可以做进一步的优化：

/\*对顺序表L作冒泡排序\*/

void BubbleSort(SqList \*L)

{

int i,j;

bool flag = true; /\*flag用作标记\*/

for(i=1;i<L->length && flag;i++) /\*若flag为true则退出循环\*/

{

flag = false; /\*初始化为false\*/

for(j=L->length;j>=i;j--) /\*注意j是从后往前循环\*/

{

if(L->data[j]>L->data[j+1])

{

swap(L,j,j+1);

flag = true; /\*如果有数据交换，则flag为true\*/

}

}

}

}

#### 改进版

**版本2：**

#include <iostream>

#include <vector>

using namespace std;

//冒泡排序 从大到小排序

void Bulldle(int\* array,int num)

{

int i,j,temp;

for(i=0;i<num-1;i++)

{

for(j = i+1;j<num;j++)

if(array[i] < array[j])

{

temp = array[i];

array[i] = array[j];

array[j] = temp;

}

}

}

// 冒泡排序的升级版 从小到大排序

void Bulldle2(int\* array,int num)

{

int i,j,temp;

int flag =1;

for(i=0;i<num-1 && flag;i++)

{

flag =0;

for(j=i+1;j<num;j++)

if(array[i] > array[j])

{

temp = array[i];

array[i] = array[j];

array[j] = temp;

flag =1;//增加标志位，即“哨兵”

}

}

}

int main()

{

int array[]={7,5,2,9,1,3};

Bulldle2(array,6);

int i;

for(i=0;i<sizeof(array)/sizeof(int);i++)

cout<<array[i]<<endl;

return 0;

}

### Python

## 性能分析

**稳定性：**稳定

**时间复杂度：**

**平均情况：O(n^2)**

最好情况：O(n)

最坏情况：O(n^2)

**空间复杂度：**O(1)

冒泡排序解决了桶排序空间复杂度高的问题，但是实际应用的很少，因为时间复杂度很高，不实用！因此引入快速排序。

# 快速排序

## 基本思想

通过一趟排序将待排序列表分割成独立的两部分，其中一部分的所有元素都比另一部分小，然后再按此方法将独立的两部分分别继续重复进行此操作，这个过程我们可以通过**递归**实现，从而达到最终将整个列表排序的目的。

**快速排序其实就是把冒泡排序中的相邻两数比较转换为与某一基准值的比较**（可以理解为冒泡排序升级版，快速排序是**交换排序**）。即，**冒泡排序是“邻居”的比较和交换，快速排序是与基准值的比较和交换**。

快速排序用到的算法思想是：递归与分治策略（二分思想）。

**快速排序与归并排序：**

1、都是分治的思想；

2、经过一次划分后，实现了对A的调整：其中一个子集合的所有元素均小于等于另外一个子集合的所有元素；

3、按同样的策略对两个子集合进行分类处理。当子集合分类完毕后，整个集合的分类也完成了。这一过程避免了子集合的归并操作。

## 具体步骤

1、从待排序列表（数组）中选择一个元素作为基准（pivot），这里我们选择最后一个元素元素；

2、遍历列表，将所有小于基准的元素放在其前面，这样就可以将待排序列表分成两部分了；

3、递归地对每个部分进行1、2操作，这里递归结束的条件是序列的大小为0或1，此时递归结束，排序就已经完成了。

## 代码实现

### C语言

#### 版本一

#include <stdio.h>

#include <stdlib.h>

//将一个数组以某一个基准点划分为两个子数组

int Partition(int\* array,int left,int right)

{

int key = array[right];//以最后一个元素为基准点

int i = left-1;

int temp;

//开始以基准点为标准分割序列

for(int j = left;j<right;j++)

{

if(array[j] < key)

{

i++;

temp = array[j];

array[j] = array[i];

array[i] = temp;

}

}

//将基准点放置到合适的位置上

temp = array[i+1];

array[i+1] = array[right];

array[right] = temp;

return i+1;

}

//快速排序

void quicksort(int\* array,int left,int right)

{

if(left < right)

{

int k = Partition(array,left,right);

quicksort(array,left,k-1);

quicksort(array,k+1,right);

}

}

#### 版本二

//第二种分割的方法

void swap(int\* a,int\* b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int partition2(int\* array,int low,int high)

{

int key = array[low];

while(low < high)

{

//从后面找到一个合适的值和前面的交换

while(low<high && array[high] >= key)

high--;

swap(&array[low],&array[high]);

//从前面找到一个合适的值和后面的交换

while(low< high && array[low] <= key)

++low;

swap(&array[low],&array[high]);

}

return low;

}

//第二种分割方法下的快速排序

void quicksort2(int\* array,int left,int right)

{

int i,j,key;

if(left < right)

{

i = left;

j = right;

key = array[i];//以最左边的元素作为划分的基准点

do

{

while(array[j] > key && i<j)

j--;//从右向左找第一个小于基准值的位置j

if(i<j) //找到了位置j

{

array[i] = array[j];

i++;

} //将第J个元素置于左端并重置i

while(array[i] < key && i<j)

i++;//从左向右找第一个大于标准的位置i

if(i<j)

{

array[j] = array[i];

j--;

} //将第I个元素置于右端并重置j

} while(i!= j);

array[i] = key;// 分割完成

quicksort2(array,left,i-1);

quicksort2(array,i+1,right);

}

}

int main()

{

int array[]={9,4,5,2,1,3,7,8,6};

quicksort2(array,0,8);

int i;

for(i=0;i<9;i++)

printf("%d\n",array[i]);

return 0;

}

### Python

## 性能分析

**稳定性：**不稳定

**时间复杂度：**

平均情况：O(nlogn)

最好情况：O(nlong)

最坏情况：O(n^2)

快速排序之所以比较快，是因为相比冒泡排序，每次交换都是跳跃式的。这样在每次交换的时候就不会像冒泡排序一样只能在相邻的数之间进行交换，交换的距离就大得多了。其实不难理解，快排的最坏情况就已经退化为冒泡排序了！

**空间复杂度：**

平均情况：O(logn)

最好情况：O(logn)

最坏情况：O(n)

快速排序的最直接竞争者是堆排序。堆排序通常比快速排序稍微慢，但是最坏情况的运行时间总是O(nlogn)。快速排序是经常比较快，但仍然有最坏性能的机会。

堆排序拥有重要的特点：仅使用固定额外的空间，即堆排序是原地排序，而快速排序需要O(nlogn)的空间。

# 归并排序

## 基本思想

归并排序的两点改进：

1、在数组长度比较短的情况下，不进行递归，而是选择其他排序方案，如插入排序；

2、归并过程中，可以用记录数组下标的方式代替申请新内存空间，从而避免A和辅助数组间的频繁数据移动。

注：基于关键字比较的排序算法的平均时间复杂度的下界为O(nlogn)。

## 具体步骤

1、分治

2、合并

## 代码实现

### C语言

**版本1：**

#include <stdio.h>

#include <stdlib.h>

//合并两个有序子数组

void Mergearray(int\* a,int left,int mid,int right,int\* temp)

{

int i = left,j = mid+1;

int m = mid,n = right;

int k=0;

//二路归并

while(i<=m && j<=n)

{

if(a[i]<=a[j])

temp[k++] = a[i++];

else

temp[k++] = a[j++];

}

//处理子数组中剩余的元素

while(i<=m)

temp[k++] = a[i++];

while(j<=n)

temp[k++] = a[j++];

//从临时数组中拷贝到目标数组

for(i=0;i<k;i++)

a[left+i] = temp[i];

}

//归并排序的核心工作

int MergeSort(int\* a,int left,int right,int\* temp)

{

if(left < right)

{

int mid = left + (right-left)/2;

//归并排序 使得左边序列有序

MergeSort(a,left,mid,temp);

//归并排序 使得右边序列有序

MergeSort(a,mid+1,right,temp);

//合并两个有序序列

Mergearray(a,left,mid,right,temp);

}

}

int main()

{

int array[]={6,3,2,1,4,10,9,8,7,5};

int b[10];

MergeSort(array,0,9,b);

int i;

for(i=0;i<10;i++)

printf("%d\n",array[i]);

return 0;

}

**版本2：**

原地归并排序

#include <iostream>

#include <vector>

using namespace std;

/\*

题目要求：

两个有序子序列a[0,mid-1]和a[mid,num-1]， 将两个子数组归并，使其整体有序空间复杂度为O(1)

\*/

/\*

思路：将同一个数组中两段有序序列合并为一个 可以使用插入的排序方式

将后半部分的元素插入到前半部分内 对后半部分的每一个元素 都在前半部分找到合适的位置然后插入（类似于插入排序）

\*/

void Merge2(int\* array,int begin,int end)

{

int middle = begin + (end-begin)/2;

int i= begin;

int temp;

//将后半部分中的元素依序插入前面有序子序列中

while(middle <= end)

{

temp = array[middle];//待插入的元素

if(array[i] < array[middle]) //找到合适的位置

{

i++;

}

else //进行插入的操作

{

int index = middle;

while(index != i)

{

array[index] = array[index-1];

index--;

}

array[i++] = temp;

middle++;

}

}

} //这个函数能否再精简？

/\*

原地归并的思想

假如有一个数组，里面有两段已经排序好的数组不使用额外的空间将这两段有序序列进行合并有两段有序空间分别为A B，从A中找第一个大于B[0]的元素并记录为A[i] 从B中找第一个大于A[i]的元素并记录为B[j] 在这段空间中进行右移 然后就变得有序

\*/

//对称交换

void Revere(int\* array,int begin,int end)

{

int temp;

while(begin < end)

{

temp = array[begin];

array[begin]=array[end];

array[end] = temp;

begin++;

end--;

}

}

//右旋转

/\*

array为待旋转的数组

begin 为开始位置

middle是旋转中心

end 为后半段

\*/

void Rotate\_right(int\* array,int begin,int middle,int end)

{

Revere(array,begin,middle);

Revere(array,middle+1,end);

Revere(array,begin,end);

}

//原地合并

void Merge\_second(int\* array,int begin,int end)

{

//[begin,mid-1] [mid,end]分别有序

int mid = begin + (end-begin)/2+1;

int i = begin;

int index;

//直到将两个有序子序列其中的元素排序完成 那么两个有序子序列就合

并完成

while(mid <= end && i<mid)

{

//从前面的有序序列中找到一个元素，其大于后序有序序列中的第一

个元素

while(array[i] <= array[mid]&& i < mid)

{

i++;

}

//记录当前两个有序子序列的分界点

index = mid;

//从后面有序子序列中找到一个元素，其大于前面有序序列的第i个

元素

while(mid <= end && array[mid]<= array[i])

{

mid++;

}

//以index为分界点，也就是以当前两个有序子序列的分界点为旋转

中心,右旋转

Rotate\_right(array,i,index-1,mid-1);

//更新前面子序列中未排序元素的位置

i += (mid-index);

}

}

//归并排序总体函数

void Inplace\_MergeSort(int\* arr,int beg,int end)

{

if(beg < end)

{

int mid = (beg+end)/2;

Inplace\_MergeSort(arr,beg,mid);

Inplace\_MergeSort(arr,mid+1,end);

Merge\_second(arr,beg,end);

}

}

int main()

{

int array[]= {2,4,6,8,10,1,3,5,7,9};

int i;

Inplace\_MergeSort(array,0,9);

for(i=0;i<10;i++)

cout<<array[i]<<endl;

return 0;

}

### Python

## 性能分析

**稳定性：**稳定

**时间复杂度：**

平均情况：O(nlogn)

最好情况：O(nlogn)

最坏情况：O(nlogn)

**空间复杂度：**O(n)

## 适用场景

# 桶排序

## 基本思想

桶排序其实就是拿空间换时间。

桶排序（Bucket Sort）是一种排序算法，它将元素分散到几个有限数量的桶中，对每个桶中的元素分别进行排序，然后按顺序将各个桶中的元素合并成最终的排序结果。

原理：

1、分桶：将待排序数组中的元素分散到若干个桶中。每个桶内的元素应该具有一定的范围，并且相互独立。

2、桶内排序：对每个非空桶中的元素进行单独的排序。可以使用其他排序算法（比如插入排序、快速排序等）对桶内元素进行排序。

3、合并桶：最后将各个桶中的元素按照顺序合并成最终的排序结果。

虽然桶排序（Bucket Sort）和计数排序（Counting Sort）都属于线性时间复杂度的排序算法，但它们有一些关键的区别点：

1、数据分布

- 桶排序：桶排序适用于元素分布较为均匀的情况。它将元素根据范围分散到不同的桶中，每个桶内再使用其他排序算法进行排序，要求桶内元素分布相对均匀。

- 计数排序：计数排序针对的是待排序数组中元素的范围较小且都是非负整数的情况。它统计每个元素出现的次数，并根据统计信息进行排序，不需要比较元素之间的大小关系。

2、桶的使用

-桶排序：桶排序中需要确定桶的数量，并将元素分散到不同的桶中。通常需要根据待排序数组的特性合理选择桶的数量和范围。

- 计数排序：计数排序中并不涉及将元素分散到多个桶中，而是通过统计元素出现的次数来实现排序，不需要明确的桶结构。

3、稳定性

- 桶排序：桶排序通常是稳定的排序算法，可以保持相同元素在排序后的相对位置不变。

- 计数排序：计数排序也是稳定的排序算法，它不涉及元素的比较，仅统计元素出现的次数，所以能够保持相同元素的相对位置不变。

4. 排序算法复杂度

- 桶排序：在最理想情况下（每个桶内的元素分布均匀），桶排序的时间复杂度为 O(m + nlogn)，其中m是待排序数组的长度，n 是桶的数量。

- 计数排序：计数排序的时间复杂度为 O(m + n)，其中 m 是待排序数组的长度，n 是待排序数组中元素的范围。

总体来说，桶排序适用于元素分布相对均匀的情况，需要先将元素分散到桶中再对桶内元素进行排序；而计数排序适用于元素范围较小、都是非负整数且分布较为集中的情况，通过统计元素出现次数直接得到排序结果。

## 具体步骤

实现步骤：

1、确定待排序数组中的最大值maxValue和最小值minValue。

2、根据元素范围确定桶的数量，并创建对应数量的桶。

3、将待排序数组中的元素分散到各个桶中。

4、对每个非空桶中的元素进行排序，可以使用其他排序算法。

5、最后按照桶的顺序将各个桶中的元素合并成最终的排序结果。

桶排序（Bucket Sort）和计数排序（Counting Sort）是两种常见的线性时间复杂度的排序算法，它们之间有一些关键的区别：

1、主要思想：

- 桶排序：将元素分散到不同的桶中，每个桶内使用其他排序算法（通常是插入排序）进行排序，最后合并桶的结果。桶排序适用于元素均匀分布的情况，可以有效地处理大规模数据。

- 计数排序：统计数组中每个元素出现的次数，然后根据统计信息对原数组进行重构，得到有序结果。计数排序适用于元素范围较小的情况，但需要整数元素非负且范围不太大。

2、适用范围：

- 桶排序：适用于元素分布均匀，而且元素之间的差值较小的情况。在元素分布不均匀的情况下，桶排序的效果可能会受到影响。

- 计数排序：适用于元素范围较小，且元素非负的情况。计数排序的性能受到元素范围的限制，如果元素范围过大，可能会占用较多的空间。

3、稳定性：

- 桶排序：可以是稳定的或非稳定的，具体取决于每个桶内排序算法的稳定性。

- 计数排序：是稳定的，因为对于相同元素，按照统计顺序放置，保持了相对顺序。

4、空间复杂度：

- 桶排序：需要额外的空间来存储桶。

- 计数排序：需要额外的空间来存储计数数组，空间复杂度相对较小。

5、时间复杂度：

- 桶排序：平均时间复杂度为O(n + k)，其中 n 是元素个数，k 是桶的数量。

-计数排序：平均时间复杂度为O(n + k)，其中 n 是元素个数，k 是元素范围。

总的来说，桶排序适用于元素分布均匀的情况，而计数排序适用于元素范围较小的情况。选择哪种排序算法取决于实际情况和数据的特点。

## 代码实现

void bucketSort(vector<float>& arr) {

int n = arr.size();

// 创建桶

vector<vector<float>> buckets(n);

// 将元素分散到各个桶中

for (int i = 0; i < n; ++i) {

int bucketIndex = n \* arr[i];

buckets[bucketIndex].push\_back(arr[i]);

}

// 对每个非空桶中的元素进行排序

for (int i = 0; i < n; ++i) {

if (!buckets[i].empty()) {

sort(buckets[i].begin(), buckets[i].end());

}

}

// 合并桶中的元素作为最终结果

int index = 0;

for (int i = 0; i < n; ++i) {

for (int j = 0; j < buckets[i].size(); ++j) {

arr[index++] = buckets[i][j];

}

}

}

另外一种实现：

void bucketSort(vector<int>& nums) {

int minVal = \*min\_element(nums.begin(), nums.end());

int maxVal = \*max\_element(nums.begin(), nums.end());

// 计算桶的数量

int bucketCount = maxVal - minVal + 1;

// 初始化桶

vector<int> buckets(bucketCount, 0);

// 统计每个元素的出现次数

for (int num : nums) {

buckets[num - minVal]++;

}

// 将桶中的元素按顺序放回原数组

int index = 0;

for (int i = 0; i < bucketCount; ++i) {

while (buckets[i] > 0) {

nums[index++] = i + minVal;

buckets[i]--;

}

}

}

分析：

// 统计每个元素的出现次数

for (int num : nums) {

buckets[num - minVal]++;

}

而不是：

for (int num : nums) {

buckets[num]++;

}

这是因为在桶排序的实现中，我们是要统计每个元素的出现次数，而num - minVal用于将元素的值映射到桶数组的索引上。具体原因如下：

1、确定桶的数量：在初始化桶数组时，我们计算了bucketCount = maxVal - minVal + 1，这样桶数组的大小就能够容纳从minVal到maxVal之间的所有可能的元素值。

2、映射到桶的索引：在统计每个元素的出现次数时，通过 buckets[num - minVal]++，我们将元素的值映射到桶数组的索引上。这是因为num - minVal可以保证桶数组的索引从0开始，每个桶对应一个元素的值。

3、避免数组越界：如果使用buckets[num]++，而不考虑minVal，那么可能会导致数组越界。因为数组的索引应该从 0 开始，而元素的值num可能小于minVal，这样计算的索引就会小于 0。

通过映射到桶数组的索引，我们能够统计每个元素的出现次数，并在后续的步骤中按顺序将桶中的元素放回原数组。

另一种写法：

void bucketSort(vector<int>& nums) {

int minVal = \*min\_element(nums.begin(), nums.end());

int maxVal = \*max\_element(nums.begin(), nums.end());

int bucketSize = 5; // 可根据情况调整桶的大小

int bucketCount = (maxVal - minVal) / bucketSize + 1;

vector<vector<int>> buckets(bucketCount);

for (int num : nums) {

int index = (num - minVal) / bucketSize;

buckets[index].push\_back(num);

}

nums.clear();

for (auto& bucket : buckets) {

sort(bucket.begin(), bucket.end());

nums.insert(nums.end(), bucket.begin(), bucket.end());

}

}

## 性能分析

**稳定性：**稳定

**时间复杂度：**

平均情况：O(m+n)

最好情况：

最坏情况：

**空间复杂度：**O(n)

桶排序的时间和空间复杂度取决于桶的数量、桶内排序算法的选择以及待排序数据的特性。

以下是桶排序的时间和空间复杂度分析：

**时间复杂度：**

- 最优时间复杂度：O(m + n)，其中m是待排序数组的长度，n是桶的数量。

-平均时间复杂度：平均情况下的时间复杂度取决于桶内排序算法的选择。如果每个桶内元素分布均匀且桶内使用的排序算法效率较高，则平均时间复杂度为 O(m + nlogn)，其中n是平均每个桶内的元素数量。

- 最坏时间复杂度：如果每个元素都分散到一个桶内，需要进行单独的桶内排序，时间复杂度为 O(m^2)。

**空间复杂度：**

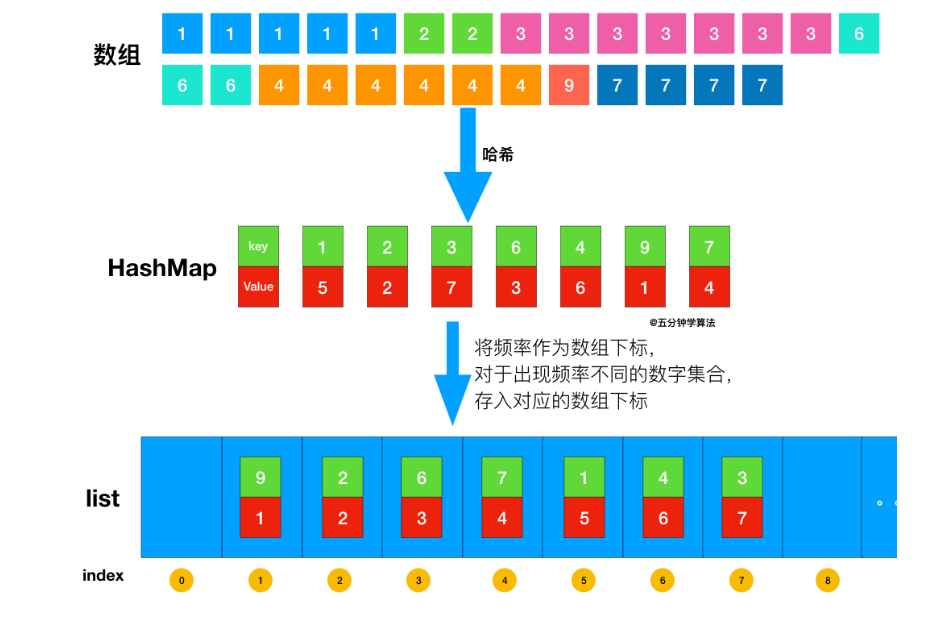
- 空间复杂度：O(m + n)，其中m是待排序数组的长度，n是桶的数量。需要额外的空间存储桶和桶内元素。

桶排序的时间复杂度在理想情况下（桶内元素分布均匀）是线性的，但是它对桶的数量和桶内元素的分布情况较为敏感。当元素分布不均匀时，桶内元素较多或较少时，桶排序的性能可能会下降。因此，在实际应用中，需要根据具体情况选择合适的桶数量和桶内排序算法，以获得较好的性能。

## 适用场景

[347. 前K个高频元素](https://leetcode-cn.com/problems/top-k-frequent-elements/)

首先依旧使用哈希表统计频率，统计完成后，创建一个数组，将频率作为数组下标，对于出现频率不同的数字集合，存入对应的数组下标即可。



[451. 根据字符出现频率排序](https://leetcode-cn.com/problems/sort-characters-by-frequency/)

[692. 前K个高频单词](https://leetcode-cn.com/problems/top-k-frequent-words/)

[912. 排序数组](https://leetcode-cn.com/problems/sort-an-array/)

# 计数排序

## 基本思想

对于给定的输入序列中的每一个元素x，确定该序列中值小于x的元素的个数（此处并非比较各元素的大小，而是通过对元素值的计数和计数值的累加来确定）。一旦有了这个信息，就可以将x直接存放到最终的输出序列的正确位置上。

计数排序核心思想，是用**空间换取时间**，本质是建立了基于元素的**Hash表**。**可以看成特殊的位图bitmap或哈希**。

计数排序（Counting Sort）是一种**非比较性**的排序算法，适用于**一定范围内整数排序**的场景。它的核心思想是统计每个元素出现的次数，然后根据元素的值以及它在统计数组中的位置，确定每个元素在排序后的位置。

**原理：**

统计元素出现次数： 遍历待排序数组，统计每个元素出现的次数，并将统计结果存储在额外的计数数组中。

计算元素位置：累加计数数组中每个元素之前的出现次数，得到每个元素在排序后的位置信息。

排序：根据元素在计数数组中的位置信息，将元素放置到正确的位置上。

## 具体步骤

实现步骤：

1、确定待排序数组中最大值maxValue和最小值minValue。

2、创建一个大小为maxValue - minValue + 1的计数数组countArray，用于统计每个元素出现的次数。

3、遍历待排序数组，统计每个元素出现的次数，并存储在countArray中。

4、对countArray进行累加操作，得到每个元素在排序后的位置信息。

5、创建一个临时数组sortedArray，根据元素在countArray中的位置信息，将元素按顺序放置到sortedArray中。

6、将sortedArray复制回原始数组，完成排序。

## 代码实现

void countingSort(vector<int>& arr) {

int minValue = arr[0], maxValue = arr[0];

for (int num : arr) {

minValue = min(minValue, num);

maxValue = max(maxValue, num);

}

int range = maxValue - minValue + 1;

vector<int> countArray(range, 0);

for (int num : arr) {

countArray[num - minValue]++;

}

for (int i = 1; i < range; ++i) {

countArray[i] += countArray[i - 1];

}

vector<int> sortedArray(arr.size());

for (int i = arr.size() - 1; i >= 0; --i) {

sortedArray[countArray[arr[i] - minValue] - 1] = arr[i];

countArray[arr[i] - minValue]--;

}

arr = sortedArray;

}

另一种写法：

void countingSort(vector<int>& nums) {

if (nums.empty()) {

// 如果数组为空，则无需排序

return;

}

// 找到数组中的最大值和最小值，以确定计数数组的大小

int minVal = \*min\_element(nums.begin(), nums.end());

int maxVal = \*max\_element(nums.begin(), nums.end());

// 创建计数数组，大小为值范围大小

vector<int> count(maxVal - minVal + 1, 0);

// 统计每个元素出现的次数

for (int num : nums) {

count[num - minVal]++; //避免溢出

}

// 根据计数数组的信息，原地修改输入数组

int index = 0;

for (int i = minVal; i <= maxVal; ++i) {

while (count[i - minVal] > 0) {

nums[index++] = i;

count[i - minVal]--;

}

}

}

或：

vector<int> countingSort(vector<int>& nums) {

// 寻找数组中的最小值和最大值

int minVal = \*min\_element(nums.begin(), nums.end());

int maxVal = \*max\_element(nums.begin(), nums.end());

// 计算计数数组的大小，范围是 [minVal, maxVal]

int range = maxVal - minVal + 1;

// 创建计数数组并初始化为 0

vector<int> count(range, 0);

// 统计每个元素的出现次数

for (int num : nums) {

count[num - minVal]++;

}

// 重构有序数组

vector<int> result(nums.size());

int index = 0;

for (int i = 0; i < range; ++i) {

// 将计数数组中的元素按照出现次数放入结果数组

while (count[i] > 0) {

result[index++] = i + minVal;

count[i]--;

}

}

return result;

}

## 性能分析

计数排序的时间复杂度和空间复杂度如下：

**时间复杂度：**

- 最优时间复杂度：O(n + k)，其中n是待排序数组的长度，k是待排序数组中元素的范围（最大值和最小值的差加一）。

- 平均时间复杂度：O(n + k)。

- 最坏时间复杂度：O(n + k)。

计数排序不涉及元素的比较操作，其时间复杂度主要取决于两个因素：

1、待排序数组的长度n。

2、待排序数组中元素的范围k。

当k远小于n时，计数排序的效率很高，因为它对数组进行了线性遍历并统计出现次数，然后根据统计信息排序。

**空间复杂度：**

* 空间复杂度：O(n + k)。

计数排序的空间复杂度也主要取决于两个因素：

1、待排序数组的长度n。

2、待排序数组中元素的范围k。

需要额外的空间来存储计数数组，其大小与元素范围有关。因此，当k远小于n时，计数排序的空间复杂度较低，是其优势之一。

## 适用场景

计数排序适用于以下场景：

1、元素范围较小：当待排序数组中的元素范围（最大值和最小值的差加一）相对较小且不是特别大时，计数排序是一种高效的排序算法。

2、元素为整数：计数排序适用于元素为整数的场景，特别是当待排序数组中的元素范围相对较小且为非负整数时，计数排序效果更好。

3、性能要求不高，但数据量大：对于数据量较大且性能要求不是特别高的情况下，计数排序可以提供较高的效率。由于计数排序的时间复杂度与元素的范围相关，当元素范围较小时，其性能较好。

4、不是稳定排序算法的限制：虽然计数排序通常是稳定的排序算法，但在一些特殊情况下可能不满足稳定性要求（例如，元素在排序后相对位置可能会改变），这需要考虑具体的情况。

总体来说，计数排序适用于元素范围较小、元素为整数且性能要求不是特别高的场景。它是一种简单、高效的排序算法，但在元素范围过大或负数存在的情况下，需要考虑其他更适合的排序算法。

# 基数排序

## 基本思想

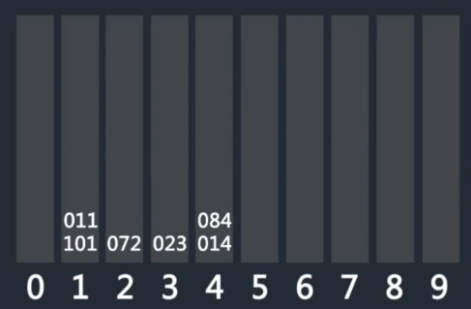
基数排序（radix sort）是一种非线性时间的排序算法。

## 具体步骤

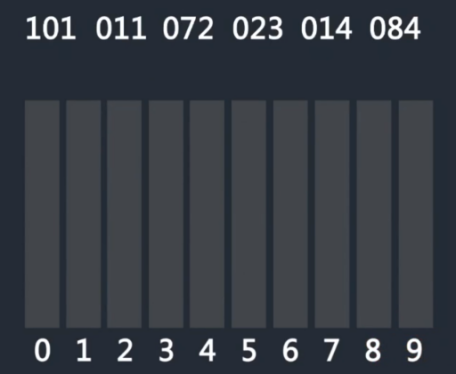
假设被排序的数都是十进制的，准备0~9十个桶：



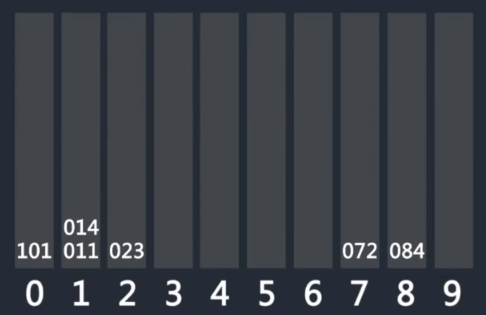
先根据个位数确定数字进入哪个桶：



依次从0~9号桶导出所有数，组成一个序列：



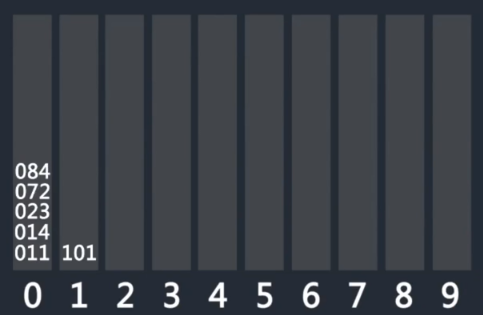
然后根据现在这个顺序，依据十位数字确定0~9号桶位置：



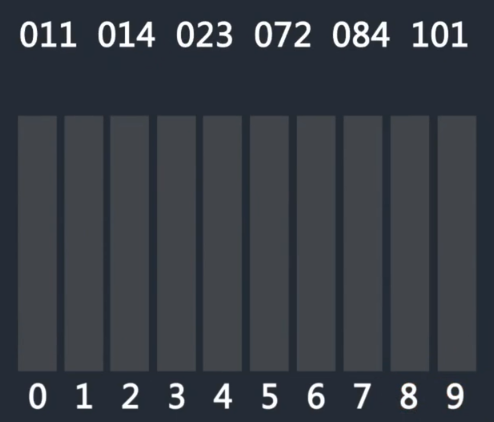
然后将数组导出组成一个新的序列：



然后根据百位数字确定桶位置：



最后一次倒出的顺序就是我们需要的顺序：



## 代码实现

## 性能分析

## 适用场景

# 应用

排序类问题的应用主要包括：

1. 数组排成最小数 剑指offer45
2. 数组的交集问题Leetcode349、Leetcode350
3. 奇偶数组Leetcode922
4. 合并区间Leetcode56
5. 链表（插入）排序Leetcode147、Leetcode148

## 几乎有序数组排序

题目：已知一个几乎有序的数组，几乎有序是指，如果把数组排好顺序的话，每个元素移动的距离不超过k，并且k相对于数组长度来说很小。请问选择什么方法对其排序比较好。

分析：

1、时间复杂度为O(N)的排序算法

计数排序、基数排序：

不基于比较的排序算法的限制：不适用所有情况

2、时间复杂度为O(N2)的排序算法

冒泡排序、选择排序：

这两个排序算法与数组原始序列无关

插入排序：

插入排序的过程与原始顺序相关

每个元素移动距离不超过k

时间复杂度O(N\*k)

3、时间复杂度为O(N\*logN)的排序算法

快速排序、归并排序：

与数组原始顺序无关，时间复杂度O(N\*logN)

4、改进后的堆排序（最佳答案）

## 按奇偶排序数组

## 重复值

题目：判断数组中是否有重复值，必须保证额外空间复杂度为O(1)。

分析：

如果没有空间复杂度的限制，用哈希表实现。哈希表实现，时间复杂度为O(N)，空间复杂度为O(N)。

非递归的堆排序

## 有序数组合并

题目：把两个有序数组合并为一个数组，第一个数组空间正好可以容纳两个数组的元素。

分析：

有序数组A有6个空间，其中3个存储数据，先将数组A最大值与数组B最大值，将大的拷贝到后面的多余空间：



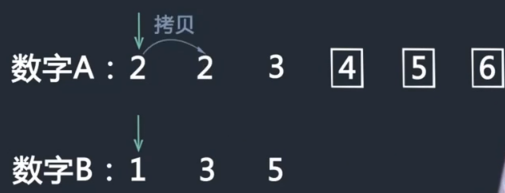
然后4和5比较，继续拷贝：

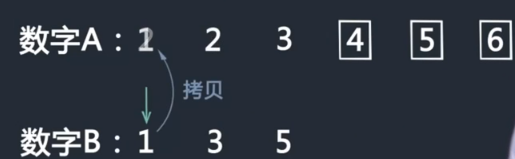


依次比较拷贝：









## 逆序数问题

题目：给定一个数组A[0…N-1]，若对于某两个元素a[i]、a[j]，若i<j且a[i]>a[j]，则称(a[i],a[j]) 为逆序对。一个数组中包含的逆序对的数目为该数组的逆序数。设计一个算法，求一个数组的逆序数。

如：3,56,2,7的逆序数为3。

分析：归并排序

代码：

## 最大的K个数

**题目：**

**分析：**

**小顶堆方案：**

1、建立一个小顶堆，小顶堆的大小为K

2、for每个数：

if这个数比小顶堆的堆顶元素大

弹出小顶堆的最小元素

把这个数插入到小顶堆

3、小顶堆中的k个元素就是所要求的元素

小顶堆的作用：

1、保持始终有k个最大元素——利于最后的输出

2、K个元素中最小的元素在堆顶——利于后续元素的比较

时间复杂度：O(N\*logk)

大顶堆方案：

1、建立全部n个元素的大顶堆；

2、利用堆排序，但得到前k个元素后即完成算法。

时间复杂度分析：

1、建堆O(N)；

2、选择1个元素的时间是O(logN)，所以，第二步的总时间复杂度为O(klogN)

该算法时间复杂度为O(N+logk)

综上，小顶堆方案好。

## 900MB数据排序

使用100MB内存对900MB的数据进行排序：

1、读入100MB数据至内存，用常规方式（如堆排序）排序；

2、将排序后的数据写入磁盘；

3、重复前两个步骤，得到9个100MB的块（临时文件）；

4、将100MB内存划分为10份，前9份中输入缓冲区，第10份为输出缓冲区：

如前9份各8M，第10份18M；或10份大小同时为10M

5、执行九路归并算法，将结果输出到输出缓冲区：

若输出缓冲区满，将数据写至目标文件，清空缓冲区；

若输入缓冲区空，读入相应文件的下一份数据。