[算法总结] 十大排序算法

*vivia*不想当科研狗的研究僧

本文首发于我的个人博客：尾尾部落

排序算法是最经典的算法知识。因为其实现代码短，应该广，在面试中经常会问到排序算法及其相关的问题。一般在面试中最常考的是快速排序和归并排序等基本的排序算法，并且经常要求现场手写基本的排序算法。如果这些问题回答不好，估计面试就凉凉了。所以熟练掌握排序算法思想及其特点并能够熟练地手写代码至关重要。

下面介绍几种常见的排序算法：冒泡排序、选择排序、插入排序、归并排序、快速排序、希尔排序、堆排序、计数排序、桶排序、基数排序的思想，其代码均采用Java实现。

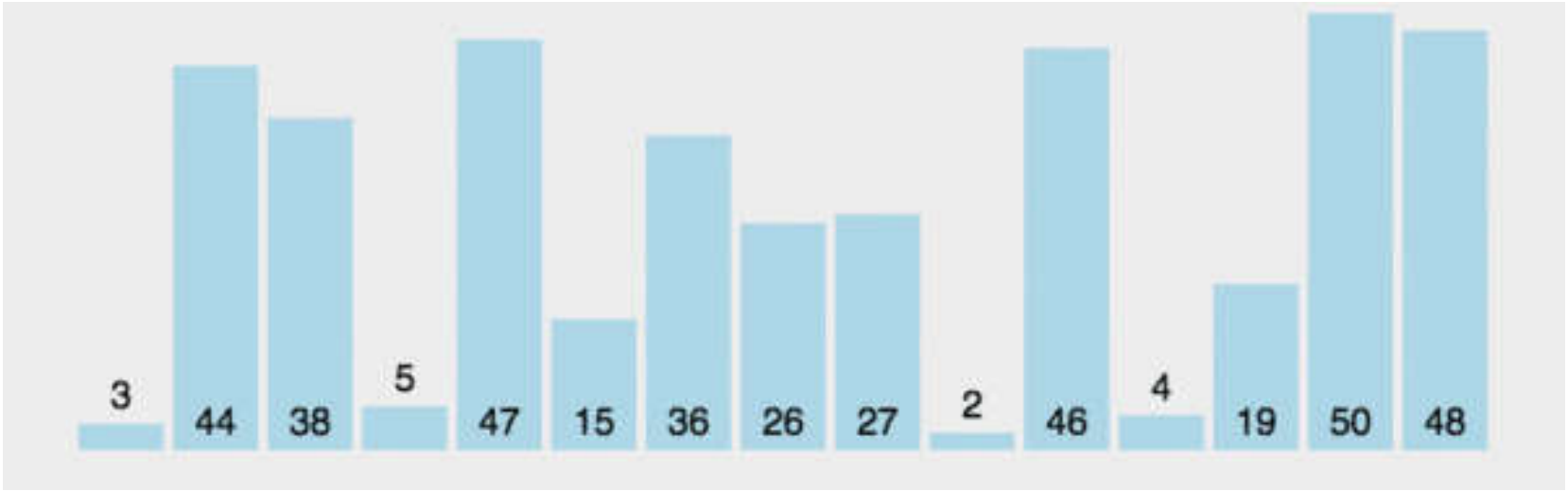
1. 冒泡排序

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

算法描述

1. 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
3. 针对所有的元素重复以上的步骤，除了最后一个；
4. 重复步骤1~3，直到排序完成。

动图演示



算法实现

```
public static void bubbleSort(int[] arr) {
    int temp = 0;
    for (int i = arr.length - 1; i > 0; i--) { // 每次需要排序的长度
        for (int j = 0; j < i; j++) { // 从第一个元素到第i个元素
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }//loop j
    }//loop i
}// method bubbleSort
```

稳定性

在相邻元素相等时，它们并不会交换位置，所以，冒泡排序是稳定排序。

适用场景

冒泡排序思路简单，代码也简单，特别适合小数据的排序。但是，由于算法复杂度较高，在数据量大的时候不适合使用。

代码优化

在数据完全有序的时候展现出最优时间复杂度，为O(n)。其他情况下，几乎总是O(n2)。因此，算法在数据基本有序的情况下，性能最好。
要使算法在最佳情况下有O(n)复杂度，需要做一些改进，增加一个`swap`的标志，当前一轮没有进行交换时，说明数组已经有序，没有必要再进行下一轮的循环了，直接退出。

```
public static void bubbleSort(int[] arr) {
    int temp = 0;
    boolean swap;
    for (int i = arr.length - 1; i > 0; i--) { // 每次需要排序的长度
        swap=false;
        for (int j = 0; j < i; j++) { // 从第一个元素到第i个元素
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swap=true;
            }
        }//loop j
        if (swap==false){
            break;
        }
    }//loop i
}// method bubbleSort
```

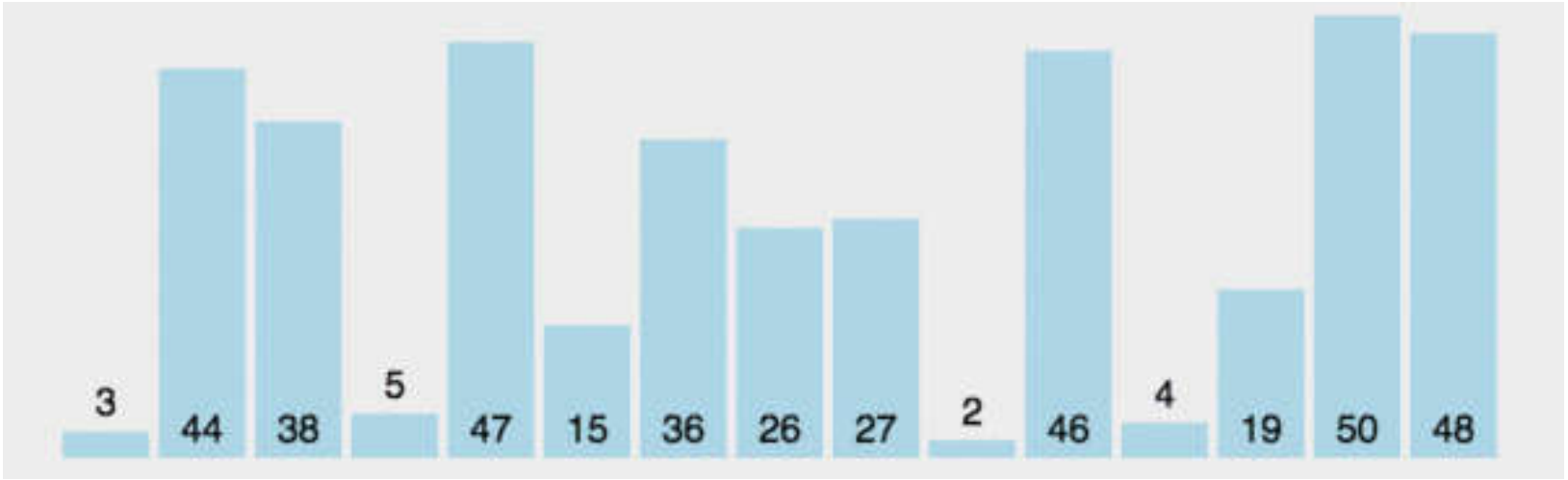
2. 选择排序

选择排序是一种简单直观的排序算法，它也是一种交换排序算法，和冒泡排序有一定的相似度，可以认为选择排序是冒泡排序的一种改进。

算法描述

1. 在未排序序列中找到最小（大）元素，存放到排序序列的起始位置
2. 从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
3. 重复第二步，直到所有元素均排序完毕。

动图演示



算法实现

```
public static void selectionSort(int[] arr) {
    int temp, min = 0;
    for (int i = 0; i < arr.length - 1; i++) {
        min = i;
        // 循环查找最小值
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[min] > arr[j]) {
                min = j;
            }
        }
        if (min != i) {
            temp = arr[i];
            arr[i] = arr[min];
            arr[min] = temp;
        }
    }
}
```

稳定性

用数组实现的选择排序是不稳定的，用链表实现的选择排序是稳定的。
不过，一般提到排序算法时，大家往往会默认是数组实现，所以选择排序是不稳定的。

适用场景

选择排序实现也比较简单，并且由于在各种情况下复杂度波动小，因此一般是优于冒泡排序的。在所有的完全交换排序中，选择排序也是比较不错的一种算法。但是，由于固有的O(n2)复杂度，选择排序在海量数据面前显得力不从心。因此，它适用于简单数据排序。

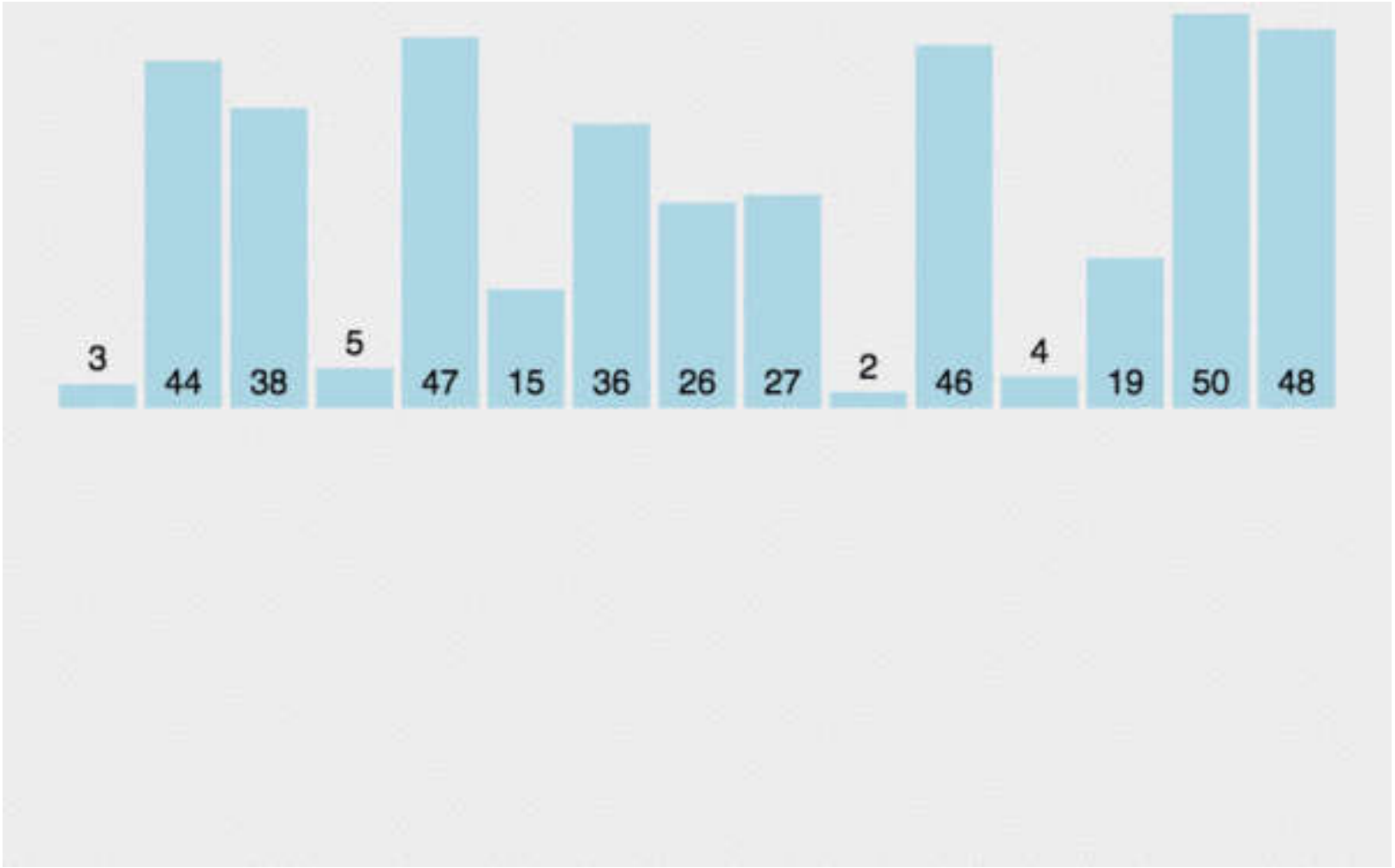
3. 插入排序

插入排序是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

算法描述

1. 把待排序的数组分成已排序和未排序两部分，初始的时候把第一个元素认为是已排好序的。
2. 从第二个元素开始，在已排好序的子数组中寻找到该元素合适的位置并插入该位置。
3. 重复上述过程直到最后一个元素被插入有序子数组中。

动图演示



算法实现

```
public static void insertionSort(int[] arr){
    for (int i=1; i<arr.length; ++i){
        int value = arr[i];
        int position=i;
        while (position>0 && arr[position-1]>value){
            arr[position] = arr[position-1];
            position--;
        }
        arr[position] = value;
```

```
    }//loop i
}
```

稳定性

由于只需要找到不大于当前数的位置而并不需要交换，因此，直接插入排序是稳定的排序方法。

适用场景

插入排序由于O(n2)的复杂度，在数组较大的时候不适用。但是，在数据比较少的时候，是一个不错的选择，一般做为快速排序的扩充。例如，在STL的sort算法和stdlib的qsort算法中，都将插入排序作为快速排序的补充，用于少量元素的排序。又如，在JDK 7 java.util.Arrays所用的sort方法的实现中，当待排数组长度小于47时，会使用插入排序。

4. 归并排序

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

算法描述

两种方法

- 递归法（Top-down）

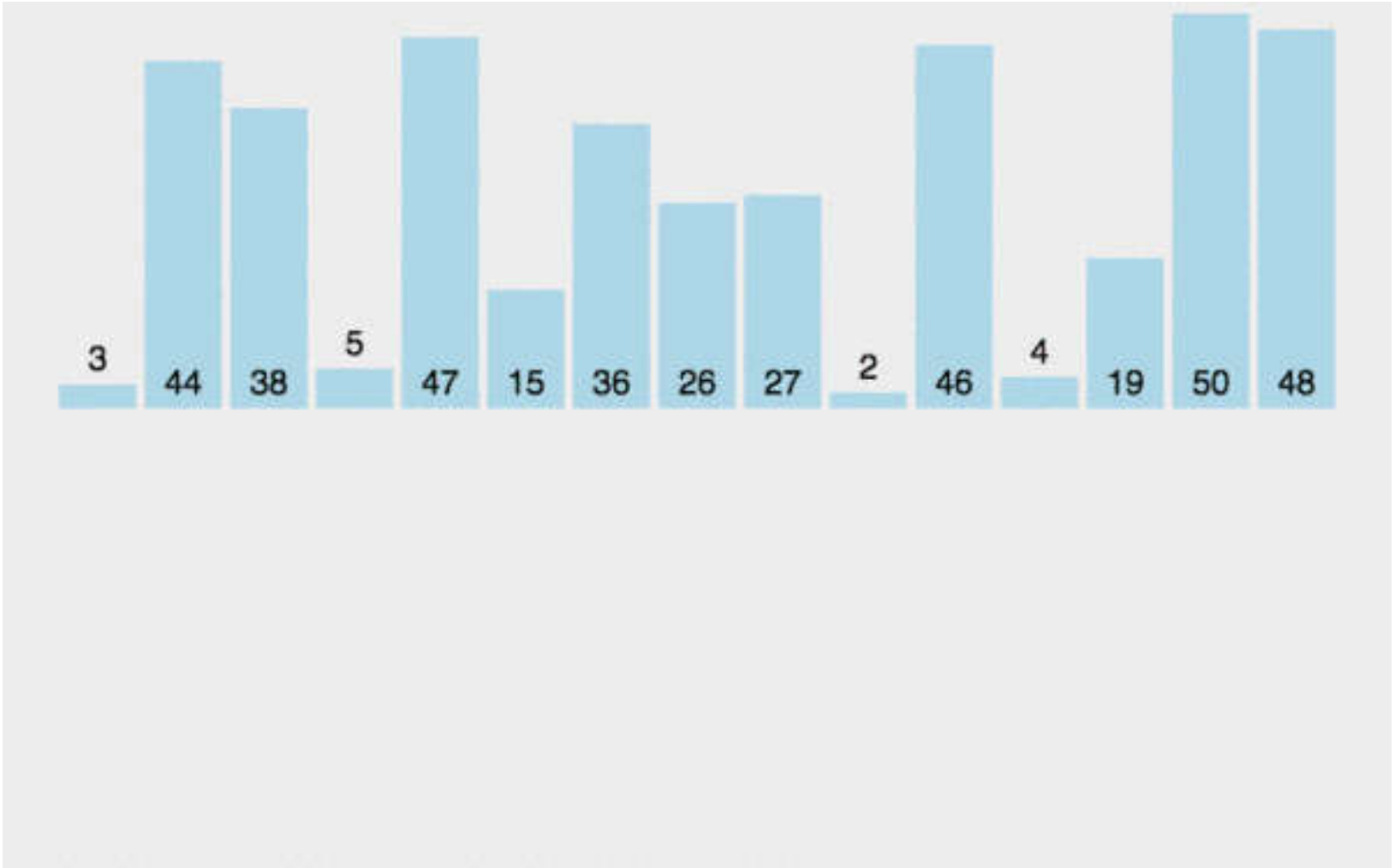
1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
4. 重复步骤3直到某一指针到达序列尾
5. 将另一序列剩下的所有元素直接复制到合并序列尾

- 迭代法（Bottom-up）

原理如下（假设序列共有n个元素）：

1. 将序列每相邻两个数字进行归并操作，形成ceil(n/2)个序列，排序后每个序列包含两/一个元素
2. 若此时序列数不是1个则将上述序列再次归并，形成ceil(n/4)个序列，每个序列包含四/三个元素
3. 重复步骤2，直到所有元素排序完毕，即序列数为1

动图演示



算法实现

```
public static void mergeSort(int[] arr){
    int[] temp =new int[arr.length];
    internalMergeSort(arr, temp, 0, arr.length-1);
}
private static void internalMergeSort(int[] arr, int[] temp, int left, int right){
    //当left==right的时，已经不需要再划分了
    if (left<right){
        int middle = (left+right)/2;
        internalMergeSort(arr, temp, left, middle);          //左子数组
        internalMergeSort(arr, temp, middle+1, right);       //右子数组
        mergeSortedArray(arr, temp, left, middle, right);    //合并两个子数组
    }
}
// 合并两个有序子序列
private static void mergeSortedArray(int arr[], int temp[], int left, int middle, int right){
    int i=left;
    int j=middle+1;
    int k=0;
    while (i<=middle && j<=right){
        temp[k++] = arr[i] <= arr[j] ? arr[i++] : arr[j++];
    }
    while (i <=middle){
        temp[k++] = arr[i++];
    }
    while ( j<=right){
        temp[k++] = arr[j++];
    }
    //把数据复制回原数组
    for (i=0; i<k; ++i){
        arr[left+i] = temp[i];
    }
}
```

稳定性

因为我们在遇到相等的数据的时候必然是按顺序“抄写”到辅助数组上的，所以，归并排序同样是稳定算法。

适用场景

归并排序在数据量比较大的时候也有较为出色的表现（效率上），但是，其空间复杂度O(n)使得在数据量特别大的时候（例如，1千万数据）几乎不可接受。而且，考虑到有的机器内存本身就比较小，因此，采用归并排序一定要注意。

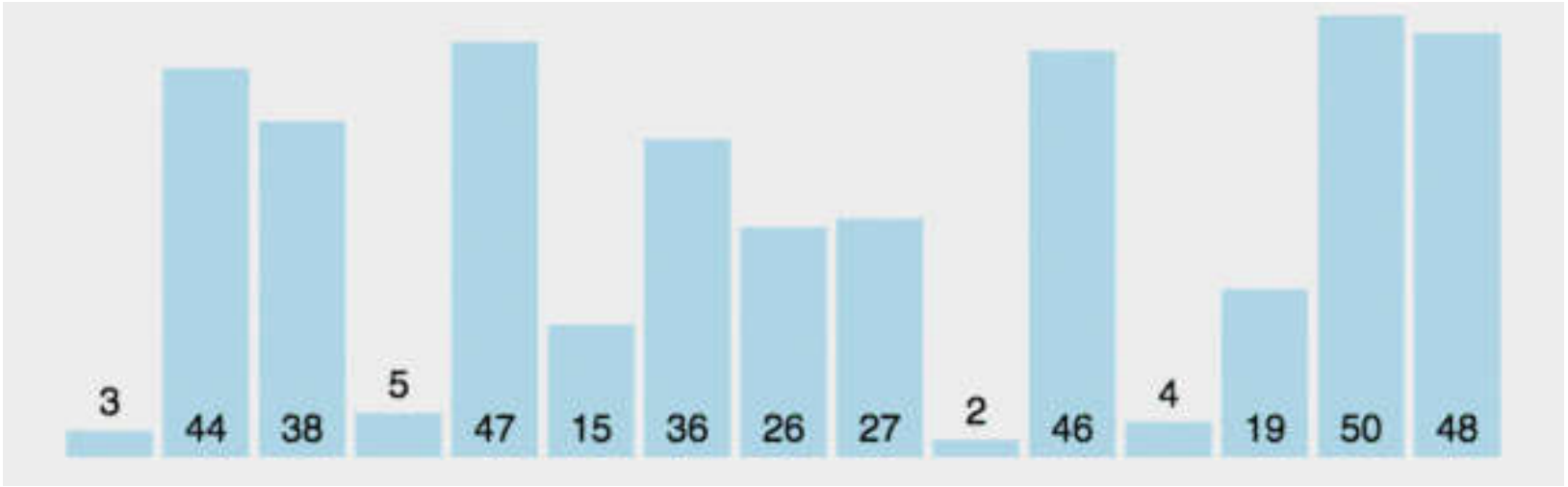
5. 快速排序

快速排序是一个知名度极高的排序算法，其对于大数据的优秀排序性能和相同复杂度算法中相对简单的实现使它注定得到比其他算法更多的宠爱。

算法描述

1. 从数列中挑出一个元素，称为"基准"（pivot），
2. 重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（相同的数可以到任何一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。
3. 递归地（recursively）把小于基准值元素的子数列和大于基准值元素的子数列排序。

动图演示



算法实现

```
public static void quickSort(int[] arr){
    qsort(arr, 0, arr.length-1);
}
private static void qsort(int[] arr, int low, int high){
    if (low >= high)
```

```
        return;
    int pivot = partition(arr, low, high);        //将数组分为两部分
    qsort(arr, low, pivot-1);                    //递归排序左子数组
    qsort(arr, pivot+1, high);                   //递归排序右子数组
}
private static int partition(int[] arr, int low, int high){
    int pivot = arr[low];     //基准
    while (low < high){
        while (low < high && arr[high] >= pivot) --high;
        arr[low]=arr[high];             //交换比基准大的记录到左端
        while (low < high && arr[low] <= pivot) ++low;
        arr[high] = arr[low];           //交换比基准小的记录到右端
    }
    //扫描完成，基准到位
    arr[low] = pivot;
    //返回的是基准的位置
    return low;
}
```

稳定性

快速排序并不是稳定的。这是因为我们无法保证相等的数据按顺序被扫描到和按顺序存放。

适用场景

快速排序在大多数情况下都是适用的，尤其在数据量大的时候性能优越性更加明显。但是在必要的时候，需要考虑下优化以提高其在最坏情况下的性能。

6. 堆排序

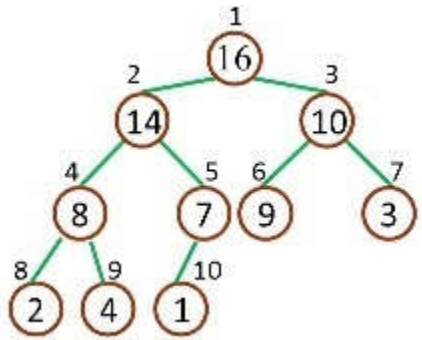
堆排序(Heapsort)是指利用堆积树（堆）这种数据结构所设计的一种排序算法，它是选择排序的一种。可以利用数组的特点快速定位指定索引的元素。堆排序就是把最大堆堆顶的最大数取出，将剩余的堆继续调整为最大堆，再次将堆顶的最大数取出，这个过程持续到剩余数只有一个时结束。

树的概念

关于树的概念请参考：[算法总结] 二叉树

堆的概念

堆是一种特殊的完全二叉树（complete binary tree）。完全二叉树的一个“优秀”的性质是，除了最底层之外，每一层都是满的，这使得堆可以利用数组来表示（普通的一般的二叉树通常用链表作为基本容器表示），每一个结点对应数组中的一个元素。
如下图，是一个堆和数组的相互关系：

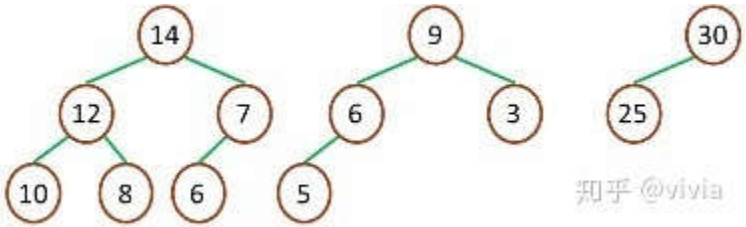


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

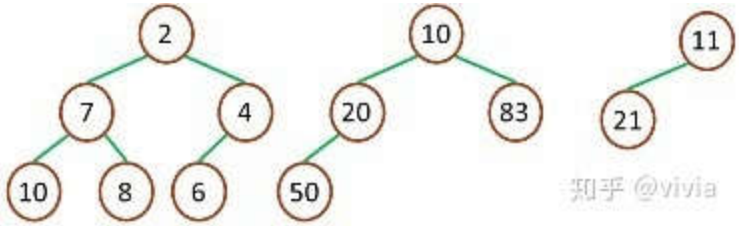
对于给定的某个结点的下标 i，可以很容易的计算出这个结点的父结点、孩子结点的下标：

- Parent(i) = floor(i/2)，i 的父节点下标
- Left(i) = 2i，i 的左子节点下标
- Right(i) = 2i + 1，i 的右子节点下标

二叉堆一般分为两种：最大堆和最小堆。
最大堆：
最大堆中的最大元素值出现在根结点（堆顶）
堆中每个父节点的元素值都大于等于其孩子结点（如果存在）



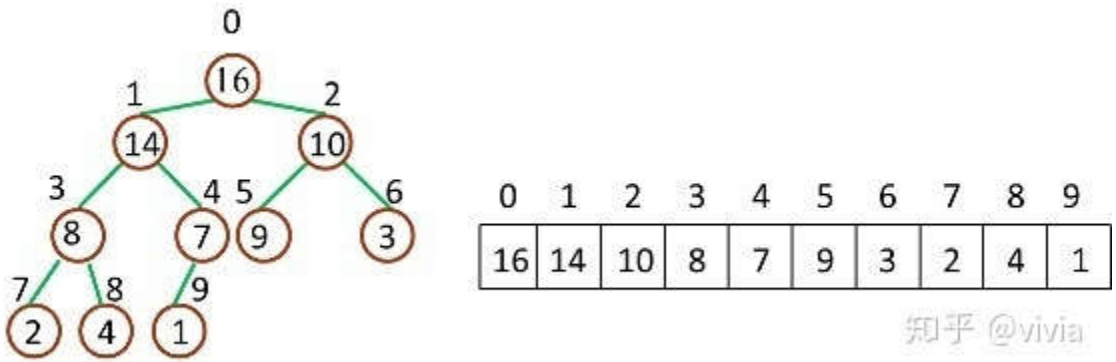
最小堆：
最小堆中的最小元素值出现在根结点（堆顶）
堆中每个父节点的元素值都小于等于其孩子结点（如果存在）



堆排序原理

堆排序就是把最大堆堆顶的最大数取出，将剩余的堆继续调整为最大堆，再次将堆顶的最大数取出，这个过程持续到剩余数只有一个时结束。在堆中定义以下几种操作：

- 最大堆调整（Max-Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点
- 创建最大堆（Build-Max-Heap）：将堆所有数据重新排序，使其成为最大堆
- 堆排序（Heap-Sort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算 继续进行下面的讨论前，需要注意的一个问题是：数组都是 Zero-Based，这就意味着我们的堆数据结构模型要发生改变



相应的，几个计算公式也要作出相应调整：

- Parent(i) = floor((i-1)/2)，i 的父节点下标
- Left(i) = 2i + 1，i 的左子节点下标
- Right(i) = 2(i + 1)，i 的右子节点下标

堆的建立和维护

堆可以支持多种操作，但现在我们关心的只有两个问题：

1. 给定一个无序数组，如何建立为堆？
2. 删除堆顶元素后，如何调整数组成为新堆？

先看第二个问题。假定我们已经有一个现成的大根堆。现在我们删除了根元素，但并没有移动别的元素。想想发生了什么：根元素空了，但其它元素还保持着堆的性质。我们可以把**最后一个元素**（代号A）移动到根元素的位置。如果不是特殊情况，则堆的性质被破坏。但这仅仅是由于A小于其某个子元素。于是，我们可以把A和这个子元素调换位置。如果A大于其所有子元素，则堆调整好了；否则，重复上述过程，A元素在树形结构中不断“下沉”，直到合适的位置，数组重新恢复堆的性质。上述过程一般称为“筛选”，方向显然是自上而下。

> 删除后的调整，是把最后一个元素放到堆顶，自上而下比较

删除一个元素是如此，插入一个新元素也是如此。不同的是，我们把新元素放在**末尾**，然后和其父节点做比较，即自下而上筛选。

> 插入是把新元素放在末尾，自下而上比较

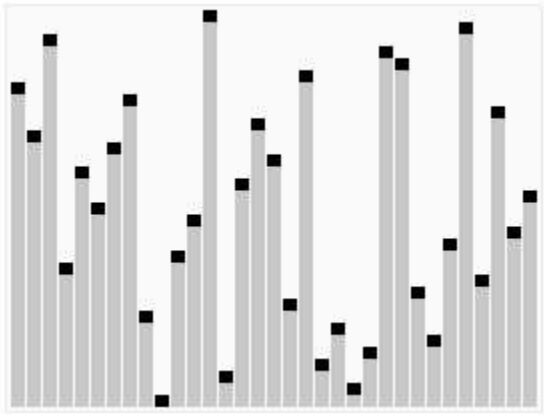
那么，第一个问题怎么解决呢？

常规方法是从第一个非叶子结点向下筛选，直到根元素筛选完毕。这个方法叫“筛选法”，需要循环筛选n/2个元素。

但我们还可以借鉴“插入排序”的思路。我们可以视第一个元素为一个堆，然后不断向其中添加新元素。这个方法叫做“插入法”，需要循环插入(n-1)个元素。

由于筛选法和插入法的方式不同，所以，相同的数据，它们建立的堆一般不同。大致了解堆之后，堆排序就是水到渠成的事情了。

动图演示



算法描述

我们需要一个升序的序列，怎么办呢？我们可以建立一个最小堆，然后每次输出根元素。但是，这个方法需要额外的空间（否则将造成大量的元素移动，其复杂度会飙升到O(n2)）。如果我们需要就地排序（即不允许有O(n)空间复杂度），怎么办？

有办法。我们可以建立最大堆，然后我们倒着输出，在最后一个位置输出最大值，次末位置输出次大值……由于每次输出的最大元素会腾出第一个空间，因此，我们恰好可以放置这样的元素而不需要额外空间。很漂亮的想法，是不是？

算法实现

```
public class ArrayHeap {
    private int[] arr;
    public ArrayHeap(int[] arr) {
        this.arr = arr;
    }
    private int getParentIndex(int child) {
        return (child - 1) / 2;
    }
    private int getLeftChildIndex(int parent) {
        return 2 * parent + 1;
    }
    private void swap(int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
```

```
    /**
     * 调整堆。
     */
    private void adjustHeap(int i, int len) {
        int left, right, j;
        left = getLeftChildIndex(i);
        while (left <= len) {
            right = left + 1;
            j = left;
            if (j < len && arr[left] < arr[right]) {
                j++;
            }
            if (arr[i] < arr[j]) {
                swap(array, i, j);
                i = j;
                left = getLeftChildIndex(i);
            } else {
                break; // 停止筛选
            }
        }
    }
    /**
     * 堆排序。
     * */
    public void sort() {
        int last = arr.length - 1;
        // 初始化最大堆
        for (int i = getParentIndex(last); i >= 0; --i) {
            adjustHeap(i, last);
        }
        // 堆调整
        while (last >= 0) {
            swap(0, last--);
            adjustHeap(0, last);
        }
    }

}
```

稳定性

堆排序存在大量的筛选和移动过程，属于不稳定的排序算法。

适用场景

堆排序在建立堆和调整堆的过程中会产生比较大的开销，在元素少的时候并不适用。但是，在元素比较多的情况下，还是不错的一个选择。尤其是在解决诸如“前n大的数”一类问题时，几乎是首选算法。

7. 希尔排序（插入排序的改良版）

在希尔排序出现之前，计算机界普遍存在“排序算法不可能突破O(n2)”的观点。希尔排序是第一个突破O(n2)的排序算法，它是简单插入排序的改进版。希尔排序的提出，主要基于以下两点：

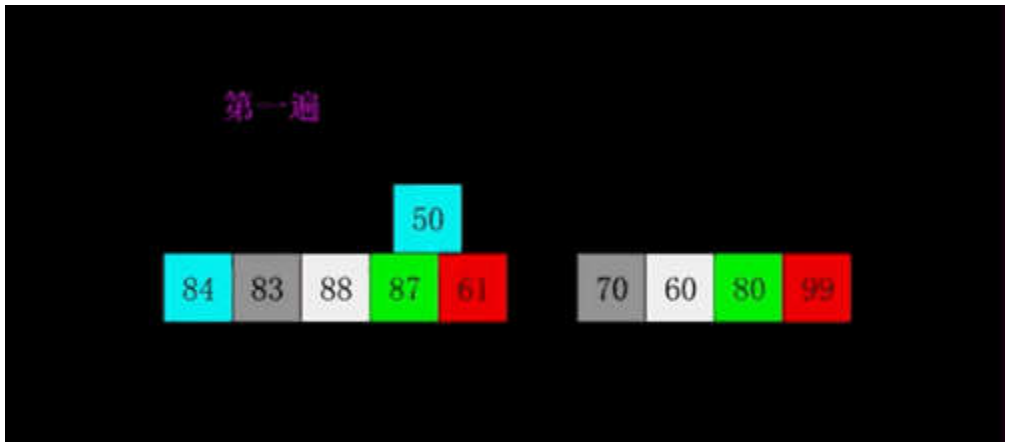
1. 插入排序算法在数组基本有序的情况下，可以近似达到O(n)复杂度，效率极高。
2. 但插入排序每次只能将数据移动一位，在数组较大且基本无序的情况下性能会迅速恶化。

算法描述

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

- 选择一个增量序列t1，t2，...，tk，其中ti>tj，tk=1；
- 按增量序列个数k，对序列进行 k 趟排序；
- 每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

动图演示



算法实现

Donald Shell增量

```
public static void shellSort(int[] arr){
    int temp;
    for (int delta = arr.length/2; delta>=1; delta/=2){                    //对每个增量进行一次排序
        for (int i=delta; i<arr.length; i++){
            for (int j=i; j>=delta && arr[j]<arr[j-delta]; j-=delta){ //注意每个地方增量和差值都是delta
                temp = arr[j-delta];
                arr[j-delta] = arr[j];
                arr[j] = temp;
            }
        }//loop i
    }//loop delta
}
```

O(n3/2) by Knuth

```
public static void shellSort2(int[] arr){
    int delta = 1;
    while (delta < arr.length/3){//generate delta
        delta=delta*3+1;    // <O(n^(3/2)) by Knuth,1973>: 1, 4, 13, 40, 121, ...
    }
    int temp;
    for (; delta>=1; delta/=3){
        for (int i=delta; i<arr.length; i++){
            for (int j=i; j>=delta && arr[j]<arr[j-delta]; j-=delta){
                temp = arr[j-delta];
                arr[j-delta] = arr[j];
                arr[j] = temp;
            }
        }//loop i
    }//loop delta
}
```

希尔排序的增量

希尔排序的增量数列可以任取，需要的唯一条件是最后一个一定为1（因为要保证按1有序）。但是，不同的数列选取会对算法的性能造成极大的影响。上面的代码演示了两种增量。
切记：增量序列中每两个元素最好不要出现1以外的公因子！（很显然，按4有序的数列再去按2排序意义并不大）。
下面是一些常见的增量序列。

- 第一种增量是最初Donald Shell提出的增量，即折半降低直到1。据研究，使用希尔增量，其时间复杂度还是O(n2)。

第二种增量Hibbard：{1, 3, ..., 2k-1}。该增量序列的时间复杂度大约是O(n1.5)。

第三种增量Sedgewick增量：(1, 5, 19, 41, 109,...)，其生成序列或者是9*4i* - 92i + 1或者是4i - 3*2i + 1。

稳定性

我们都知道插入排序是稳定算法。但是，Shell排序是一个多次插入的过程。在一次插入中我们能确保不移动相同元素的顺序，但在多次的插入中，相同元素完全有可能在不同的插入轮次被移动，最后稳定性被破坏，因此，Shell排序不是一个稳定的算法。

适用场景

Shell排序虽然快，但是毕竟是插入排序，其数量级并没有后起之秀--快速排序O(nlogn)快。在大量数据面前，Shell排序不是一个好的算法。但是，中小型规模的数据完全可以使用它。

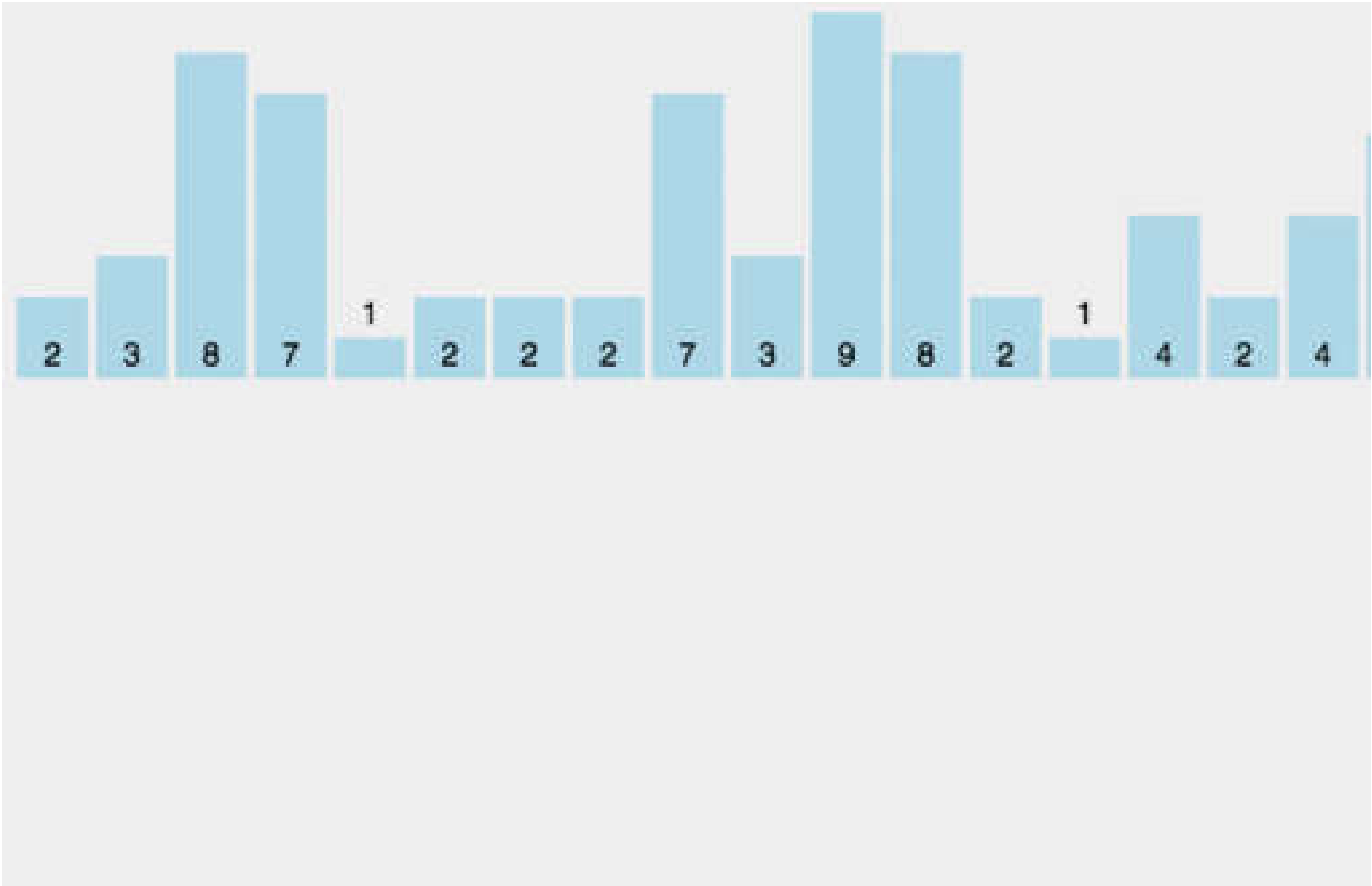
计数排序

计数排序不是基于比较的排序算法，其核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。 作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

算法描述

1. 找出待排序的数组中最大和最小的元素；
2. 统计数组中每个值为i的元素出现的次数，存入数组C的第i项；
3. 对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；
4. 反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。

动图演示



算法实现

```
public static void countSort(int[] a, int max, int min) {
     int[] b = new int[a.length];//存储数组
     int[] count = new int[max - min + 1];//计数数组

     for (int num = min; num <= max; num++) {
```

```
            //初始化各元素值为0，数组下标从0开始因此减min
            count[num - min] = 0;
        }

        for (int i = 0; i < a.length; i++) {
            int num = a[i];
            count[num - min]++;//每出现一个值，计数数组对应元素的值+1
        }

        for (int num = min + 1; num <= max; num++) {
            //加总数组元素的值为计数数组对应元素及左边所有元素的值的总和
            count[num - min] += sum[num - min - 1]
        }

        for (int i = 0; i < a.length; i++) {
             int num = a[i];//源数组第i位的值
             int index = count[num - min] - 1;//加总数组中对应元素的下标
             b[index] = num;//将该值存入存储数组对应下标中
             count[num - min]--;//加总数组中，该值的总和减少1。
        }

        //将存储数组的值一一替换给源数组
        for(int i=0;i<a.length;i++){
            a[i] = b[i];
        }
    }
```

稳定性

最后给 b 数组赋值是倒着遍历的，而且放进去一个就将C数组对应的值（表示前面有多少元素小于或等于A[i]）减去一。如果有相同的数x1,x2，那么相对位置后面那个元素x2放在（比如下标为4的位置），相对位置前面那个元素x1下次进循环就会被放在x2前面的位置3。从而保证了稳定性。

适用场景

排序目标要能够映射到整数域，其最大值最小值应当容易辨别。例如高中生考试的总分数，显然用0-750就OK啦；又比如一群人的年龄，用个0-150应该就可以了，再不济就用0-200喽。另外，计数排序需要占用大量空间，它比较适用于数据比较集中的情况。

桶排序

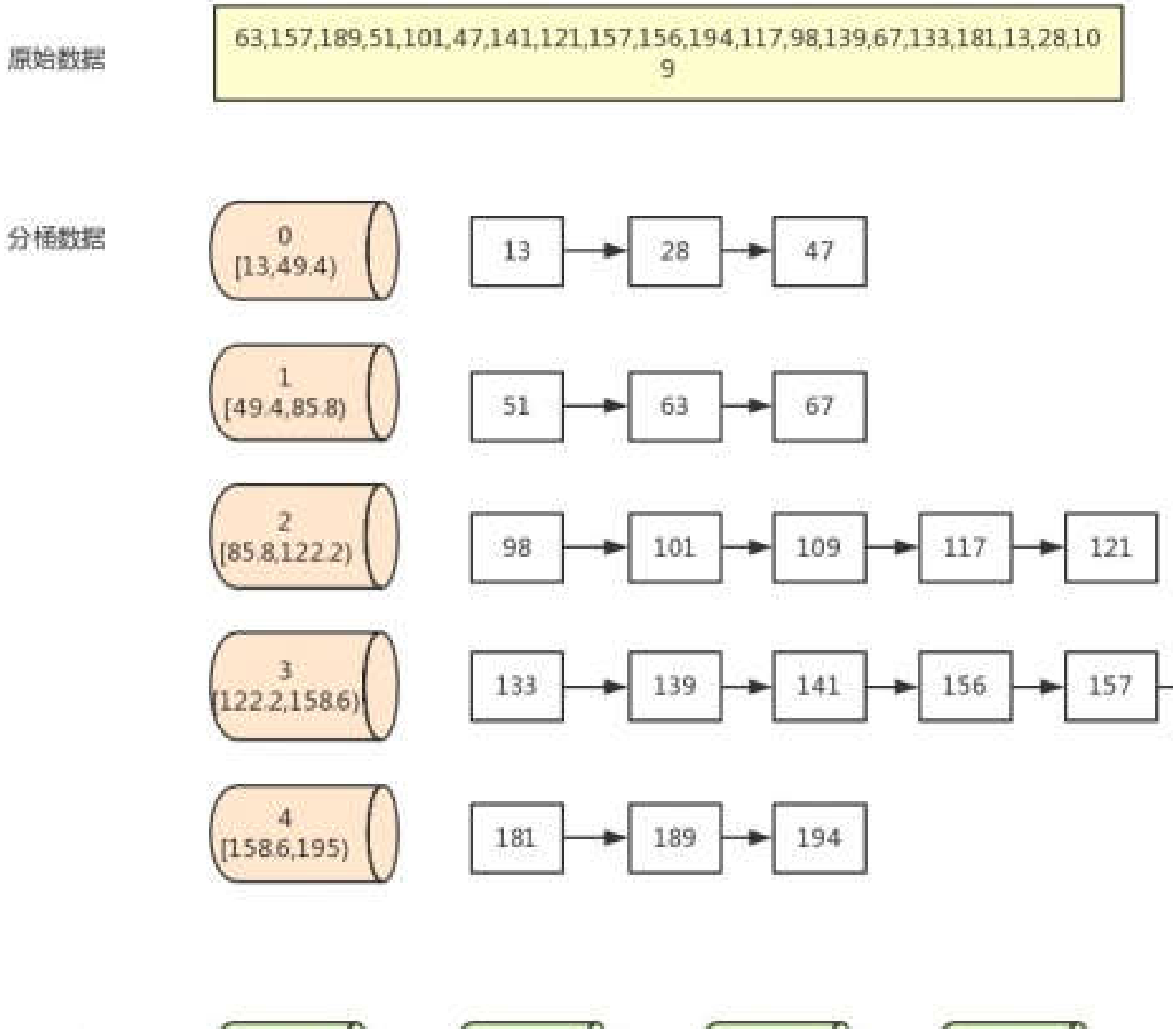
桶排序又叫箱排序，是计数排序的升级版，它的工作原理是将数组分到有限数量的桶子里，然后对每个桶子再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排序），最后将各个桶中的数据有序的合并起来。

> 计数排序是桶排序的一种特殊情况，可以把计数排序当成每个桶里只有一个元素的情况。网络中很多博文写的桶排序实际上都是计数排序，并非标准的桶排序，要注意辨别。

算法描述

1. 找出待排序数组中的最大值max、最小值min
2. 我们使用 动态数组ArrayList 作为桶，桶里放的元素也用 ArrayList 存储。桶的数量为(max-min)/arr.length+1
3. 遍历数组 arr，计算每个元素 arr[i] 放的桶
4. 每个桶各自排序
5. 遍历桶数组，把排序好的元素放进输出数组

图片演示


原始数据
63,157,189,51,101,47,141,121,157,156,194,117,98,139,67,133,181,13,28,109
分桶数据
0
[13,49.4)
13
28
47
1
[49.4,85.8)
51
63
67
2
[85.8,122.2)
98
101
109
117
121
3
[122.2,158.6)
133
139
141
156
157
4
[158.6,195)
181
189
194

算法实现

```
public static void bucketSort(int[] arr){
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;
    for(int i = 0; i < arr.length; i++){
        max = Math.max(max, arr[i]);
        min = Math.min(min, arr[i]);
    }
    //桶数
    int bucketNum = (max - min) / arr.length + 1;
    ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketNum);
    for(int i = 0; i < bucketNum; i++){
        bucketArr.add(new ArrayList<Integer>());
    }
    //将每个元素放入桶
    for(int i = 0; i < arr.length; i++){
        int num = (arr[i] - min) / (arr.length);
        bucketArr.get(num).add(arr[i]);
    }
    //对每个桶进行排序
    for(int i = 0; i < bucketArr.size(); i++){
        Collections.sort(bucketArr.get(i));
    }
    System.out.println(bucketArr.toString());
}
```

稳定性

可以看出，在分桶和从桶依次输出的过程是稳定的。但是，由于我们在对每个桶进行排序时使用了其他算法，所以，桶排序的稳定性依赖于这一步。如果我们使用了快排，显然，算法是不稳定的。

适用场景

桶排序可用于最大最小值相差较大的数据情况，但桶排序要求数据的分布必须均匀，否则可能导致数据都集中到一个桶中。比如[104,150,123,132,20000], 这种数据会导致前4个数都集中到同一个桶中。导致桶排序失效。

基数排序

基数排序(Radix Sort)是桶排序的扩展，它的基本思想是：将整数按位数切割成不同的数字，然后按每个位数分别比较。排序过程：将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后, 数列就变成一个有序序列。

算法描述

1. 取得数组中的最大数，并取得位数；
2. arr为原始数组，从最低位开始取每个位组成radix数组；
3. 对radix进行计数排序（利用计数排序适用于小范围数的特点）；

动图

算法实现

```
public abstract class Sorter {
     public abstract void sort(int[] array);
}

public class RadixSorter extends Sorter {
```

```
private int radix;

public RadixSorter() {
     radix = 10;
}

@Override
public void sort(int[] array) {
     // 数组的第一维表示可能的余数0-radix，第二维表示array中的等于该余数的元素
     // 如：十进制123的个位为3，则bucket[3][] = {123}
     int[][] bucket = new int[radix][array.length];
     int distance = getDistance(array); // 表示最大的数有多少位
     int temp = 1;
     int round = 1; // 控制键值排序依据在哪一位
     while (round <= distance) {
          // 用来计数：数组counter[i]用来表示该位是i的数的个数
          int[] counter = new int[radix];
          // 将array中元素分布填充到bucket中，并进行计数
          for (int i = 0; i < array.length; i++) {
               int which = (array[i] / temp) % radix;
               bucket[which][counter[which]] = array[i];
               counter[which]++;
          }
          int index = 0;
          // 根据bucket中收集到的array中的元素，根据统计计数，在array中重新排列
          for (int i = 0; i < radix; i++) {
               if (counter[i] != 0)
                    for (int j = 0; j < counter[i]; j++) {
                         array[index] = bucket[i][j];
                         index++;
                    }
               counter[i] = 0;
          }
          temp *= radix;
          round++;
     }
}

private int getDistance(int[] array) {
     int max = computeMax(array);
     int digits = 0;
     int temp = max / radix;
     while(temp != 0) {
          digits++;
          temp = temp / radix;
     }
```

```
            return digits + 1;
        }

        private int computeMax(int[] array) {
            int max = array[0];
            for(int i=1; i<array.length; i++) {
                if(array[i]>max) {
                    max = array[i];
                }
            }
            return max;
        }
}
```

稳定性

通过上面的排序过程，我们可以看到，每一轮映射和收集操作，都保持从左到右的顺序进行，如果出现相同的元素，则保持他们在原始数组中的顺序。可见，基数排序是一种稳定的排序。

适用场景

基数排序要求较高，元素必须是整数，整数时长度10W以上，最大值100W以下效率较好，但是基数排序比其他排序好在可以适用字符串，或者其他需要根据多个条件进行排序的场景，例如日期，先排序日，再排序月，最后排序年 ，其它排序算法可是做不了的。

总结

排序算法	平均时间复杂度	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(1)$	不稳定
插入排序	$O(n^2)$	$O(1)$	稳定
归并排序	$O(nlogn)$	$O(n)$	稳定
快速排序	$O(nlogn)$	$O(nlogn)$	不稳定
堆排序	$O(nlogn)$	$O(1)$	不稳定
希尔排序	$O(nlogn)$	$O(nlogn)$	不稳定
计数排序	$O(n+k)$	$O(n+k)$	稳定

参考

1. LeetCode领扣:面试 | 常用的排序算法总结
2. 飞翔的猫咪: 用Java写算法
3. bubkoo: 常见排序算法