

# 面试中的排序算法总结

16 回复 490 查看



(<https://www.shiyanlou.com/user/8490>) 实验楼管理员



(<https://www.shiyanlou.com/vip>)

2016-04-28 15:54

技术分享 (<https://www.shiyanlou.com/questions/?tag=技术分享>)

## 前言

查找和排序算法是算法的入门知识，其经典思想可以用于很多算法当中。因为其实现代码较短，应用较常见。所以在面试中经常会问到排序算法及其相关的问题。但万变不离其宗，只要熟悉了思想，灵活运用也不是难事。一般在面试中最常考的是快速排序和归并排序，并且经常有面试官要求现场写出这两种排序的代码。对这两种排序的代码一定要信手拈来才行。还有插入排序、冒泡排序、堆排序、基数排序、桶排序等。面试官对于这些排序可能会要求比较各自的优劣、各种算法的思想及其使用场景。还有要会分析算法的时间和空间复杂度。通常查找和排序算法的考察是面试的开始，如果这些问题回答不好，估计面试官都没有继续面试下去的兴趣都没了。所以想开个好头就要把常见的排序算法思想及其特点要熟练掌握，有必要时要熟练写出代码。

接下来我们就分析一下常见的排序算法及其使用场景。限于篇幅，某些算法的详细演示和图示请自行寻找详细的参考。



## 全部回答



实验楼管理员 (<https://www.shiyanlou.com/user/8490>)



(<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

### 冒泡排序

冒泡排序是最简单的排序之一了，其大体思想就是通过与相邻元素的比较和交换来把小的数交换到最前面。这个过程类似于水泡向上升一样，因此而得名。举个栗子，对5,3,8,6,4这个无序序列进行冒泡排序。首先从后向前冒泡，4和6比较，把4交换到前面，序列变成5,3,8,4,6。同理4和8交换，变成5,3,4,8,6,3和4无需交换。5和3交换，变成3,5,4,8,6,3.这样一次冒泡就完了，把最小的数3排到最前面了。对剩下的序列依次冒泡就会得到一个有序序列。冒泡排序的时间复杂度为  $O(n^2)$ 。

实现代码：

```
/**
 *@Description:<p>冒泡排序算法实现</p>
 *@author 王旭
 *@time 2016-3-3 下午8:54:27
 */
public class BubbleSort {

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

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

2016-04-28 15:55



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) 💖 (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

## 选择排序

选择排序的思想其实和冒泡排序有点类似，都是在一次排序后把最小的元素放到最前面。但是过程不同，冒泡排序是通过相邻的比较和交换。而选择排序是通过对整体的选择。举个栗子，对5,3,8,6,4这个无序序列进行简单选择排序，首先要选择5以外的最小数来和5交换，也就是选择3和5交换，一次排序后就变成了3,5,8,6,4.对剩下的序列一次进行选择 and 交换，最终就会得到一个有序序列。其实选择排序可以看成冒泡排序的优化，因为其目的相同，只是选择排序只有在确定了最小数的前提下才进行交换，大大减少了交换的次数。选择排序的时间复杂度为  $O(n^2)$

实现代码：

```

/**
 * @Description:<p>简单选择排序算法的实现</p>
 * @author 王旭
 * @time 2016-3-3 下午9:13:35
 */
public class SelectSort {

    public static void selectSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;
        int minIndex = 0;
        for(int i=0; i<arr.length-1; i++) { //只需要比较n-1次
            minIndex = i;
            for(int j=i+1; j<arr.length; j++) { //从i+1开始比较，因为minIndex默认为i了，i就没必要比了。
                if(arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }


            if(minIndex != i) { //如果minIndex不为i，说明找到了更小的值，交换之。
                swap(arr, i, minIndex);
            }
        }
    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}

```

2016-04-28 15:55



实验楼管理员 (<https://www.shiyanlou.com/user/8490>)  (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

## 插入排序

插入排序不是通过交换位置而是通过比较找到合适的位置插入元素来达到排序的目的。相信大家都有过打扑克牌的经历，特别是牌数较大的。在分牌时可能要整理自己的牌，牌多的时候怎么整理呢？就是拿到一张牌，找到一个合适的位置插入。这个原理其实和插入排序是一样的。举个例子，对5,3,8,6,4这个无序序列进行简单插入排序，首先假设第一个数的位置是正确的，想一下在拿到第一张牌的时候，没必要整理。然后3要插到5前面，把5后移一位，变成3,5,8,6,4.想一下整理牌的时候应该也是这样吧。然后8不用动，6插在8前面，8后移一位，4插在5前面，从5开始都向后移一位。注意在插入一个数的时候要保证这个数前面的数已经有序。简单插入排序的时间复杂度也是  $O(n^2)$ 。

实现代码：

```

/**
 * @Description:<p>简单插入排序算法实现</p>
 * @author 王旭
 * @time 2016-3-3 下午9:38:55
 */
public class InsertSort {

    public static void insertSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;

        for(int i=1; i<arr.length; i++) { //假设第一个数位置时正确的；要往后移，必须要假设第一个。

            int j = i;
            int target = arr[i]; //待插入的

            //后移
            while(j > 0 && target < arr[j-1]) {
                arr[j] = arr[j-1];
                j --;
            }

            //插入
            arr[j] = target;
        }
    }
}

```

2016-04-28 15:55



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

## 快速排序

快速排序一听名字就觉得很高端，在实际应用当中快速排序确实也是表现最好的排序算法。快速排序虽然高端，但其实其思想是来自冒泡排序，冒泡排序是通过相邻元素的比较和交换把最小的冒泡到最顶端，而快速排序是比较和交换小数和大数，这样一来不仅把小数冒泡到上面同时也把大数沉到下面。

举个栗子：对5,3,8,6,4这个无序序列进行快速排序，思路是右指针找比基准数小的，左指针找比基准数大的，交换之。

5,3,8,6,4 用5作为比较的基准，最终会把5小的移动到5的左边，比5大的移动到5的右边。

5,3,8,6,4 首先设置i,j两个指针分别指向两端，j指针先扫描（思考一下为什么？）4比5小停止。然后i扫描，8比5大停止。交换i,j位置。

5,3,4,6,8 然后j指针再扫描，这时j扫描4时两指针相遇。停止。然后交换4和基准数。

4,3,5,6,8 一次划分后达到了左边比5小，右边比5大的目的。之后对左右子序列递归排序，最终得到有序序列。

上面留下来了一个问题为什么一定要j指针先动呢？首先这也不是绝对的，这取决于基准数的位置，因为在最后两个指针相遇的时候，要交换基准数到相遇的位置。一般选取第一个数作为基准数，那么就是在左边，所以最后相遇的数要和基准数交换，那么相遇的数一定要比基准数小。所以j指针先移动才能先找到比基准数小的数。

快速排序是不稳定的，其时间平均时间复杂度是  $O(n \lg n)$ 。

实现代码：

```

/**
 *@Description:<p>实现快速排序算法</p>
 *@author 王旭
 *@time 2016-3-3 下午5:07:29
 */
public class QuickSort {
    //一次划分
    public static int partition(int[] arr, int left, int right) {
        int pivotKey = arr[left];
        int pivotPointer = left;

        while(left < right) {
            while(left < right && arr[right] >= pivotKey)
                right--;
            while(left < right && arr[left] <= pivotKey)
                left++;
            swap(arr, left, right); //把大的交换到右边，把小的交换到左边。
        }
        swap(arr, pivotPointer, left); //最后把pivot交换到中间
        return left;
    }

    public static void quickSort(int[] arr, int left, int right) {
        if(left >= right)
            return ;
        int pivotPos = partition(arr, left, right);
        quickSort(arr, left, pivotPos-1);
        quickSort(arr, pivotPos+1, right);
    }


    public static void sort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;
        quickSort(arr, 0, arr.length-1);
    }

    public static void swap(int[] arr, int left, int right) {
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
    }
}

```

2016-04-28 15:56



实验楼管理员 (<https://www.shiyanlou.com/user/8490>)  (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

其实上面的代码还可以再优化，上面代码中基准数已经在pivotKey中保存了，所以不需要每次交换都设置一个temp变量，在交换左右指针的时候只需要先后覆盖就可以了。这样既能减少空间的使用还能降低赋值运算的次数。优化代码如下：

```

/**
 * @Description:<p>实现快速排序算法</p>
 * @author 王旭
 * @time 2016-3-3 下午5:07:29
 */
public class QuickSort {

    /**
     * 划分
     * @param arr
     * @param left
     * @param right
     * @return
     */
    public static int partition(int[] arr, int left, int right) {
        int pivotKey = arr[left];

        while(left < right) {
            while(left < right && arr[right] >= pivotKey)
                right--;
            arr[left] = arr[right]; //把小的移动到左边
            while(left < right && arr[left] <= pivotKey)
                left++;
            arr[right] = arr[left]; //把大的移动到右边
        }
        arr[left] = pivotKey; //最后把pivot赋值到中间
        return left;
    }

    /**
     * 递归划分子序列
     * @param arr
     * @param left
     * @param right
     */
    public static void quickSort(int[] arr, int left, int right) {
        if(left >= right)
            return ;
        int pivotPos = partition(arr, left, right);
        quickSort(arr, left, pivotPos-1);
        quickSort(arr, pivotPos+1, right);
    }


    public static void sort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;
        quickSort(arr, 0, arr.length-1);
    }
}

```

总结快速排序的思想：冒泡+二分+递归分治，慢慢体会。。。

2016-04-28 15:56



实验楼管理员 (<https://www.shiyanlou.com/user/8490>)  (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

## 堆排序

堆排序是借助堆来实现的选择排序，思想同简单的选择排序，以下以大顶堆为例。注意：如果想升序排序就使用大顶堆，反之使用小顶堆。原因是堆顶元素需要交换到序列尾部。

首先，实现堆排序需要解决两个问题：

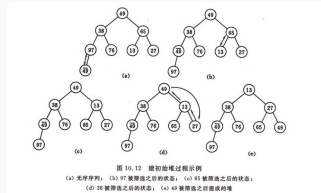
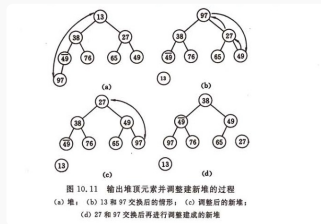
1. 如何由一个无序序列建成一个堆？
2. 如何在输出堆顶元素之后，调整剩余元素成为一个新的堆？

第一个问题，可以直接使用线性数组来表示一个堆，由初始的无序序列建成一个堆就需要自底向上从第一个非叶元素开始挨个调整成一个堆。

第二个问题，怎么调整成堆？首先是将堆顶元素和最后一个元素交换。然后比较当前堆顶元素的左右孩子节点，因为除了当前的堆顶元素，左右孩子堆均满足条件，这时需要选择当前堆顶元素与左右孩子节点的较大者（大顶堆）交换，直至叶子节点。我们称这个自堆顶自叶子的调整为筛选。

从一个无序序列建堆的过程就是一个反复筛选的过程。若将此序列看成是一个完全二叉树，则最后一个非终端节点是 $n/2$ 取底个元素，由此筛选即可。举个栗子：

49,38,65,97,76,13,27,49序列的堆排序建初始堆和调整的过程如下：



2016-04-28 15:57



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) <https://www.shiyanlou.com/vip>

(<https://www.shiyanlou.com/user/8490>)  
实现代码:

```

/**
 * @Description:<p>堆排序算法的实现，以大顶堆为例。</p>
 * @author 王旭
 * @time 2016-3-4 上午9:26:02
 */
public class HeapSort {

    /**
     * 堆筛选，除了start之外，start~end均满足大顶堆的定义。
     * 调整之后start~end称为一个大顶堆。
     * @param arr 待调整数组
     * @param start 起始指针
     * @param end 结束指针
     */
    public static void heapAdjust(int[] arr, int start, int end) {
        int temp = arr[start];

        for(int i=2*start+1; i<=end; i*=2) {
            //左右孩子的节点分别为2*i+1, 2*i+2

            //选择出左右孩子较小的下标
            if(i < end && arr[i] < arr[i+1]) {
                i ++;
            }
            if(temp >= arr[i]) {
                break; //已经为大顶堆，=保持稳定性。
            }
            arr[start] = arr[i]; //将子节点上移
            start = i; //下一轮筛选
        }

        arr[start] = temp; //插入正确的位置
    }

    public static void heapSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;

        //建立大顶堆
        for(int i=arr.length/2; i>=0; i--) {
            heapAdjust(arr, i, arr.length-1);
        }

        for(int i=arr.length-1; i>=0; i--) {
            swap(arr, 0, i);
            heapAdjust(arr, 0, i-1);
        }

    }


    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

}

```

2016-04-28 15:57



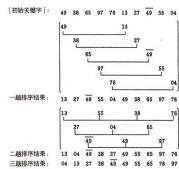
实验楼管理员 (<https://www.shiyanlou.com/user/8490>)  (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

## 希尔排序

希尔排序是插入排序的一种高效率的实现，也叫缩小增量排序。简单的插入排序中，如果待排序列是正序时，时间复杂度是  $O(n)$ ，如果序列是基本有序的，使用直接插入排序效率就非常高。希尔排序就利用了这个特点。基本思想是：先将整个待排序序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录基本有序时再对全体记录进行一次直接插入排序。举个例子：





从上述排序过程可见，希尔排序的特点是，子序列的构成不是简单的逐段分割，而是将某个相隔某个增量的记录组成一个子序列。如上面的例子，第一趟排序时的增量为5，第二趟排序的增量为3。由于前两趟的插入排序中记录的关键字是和同一子序列中的前一个记录的关键字进行比较，因此关键字较小的记录就不是一步一步地向前挪动，而是跳跃式地往前移，从而使得进行最后一趟排序时，整个序列已经做到基本有序，只要作记录的少量比较和移动即可。因此希尔排序的效率要比直接插入排序高。

希尔排序的分析是复杂的，时间复杂度是所取增量的函数，这涉及一些数学上的难题。但是在大量实验的基础上推出当 $n$ 在某个范围内时，时间复杂度可以达到  $O(n^{1.3})$ 。

实现代码：

```
/**
 * @Description: <p>希尔排序算法实现</p>
 * @author 王旭
 * @time 2016-3-3 下午10:53:55
 */
public class ShellSort {

    /**
     * 希尔排序的一趟插入
     * @param arr 待排数组
     * @param d 增量
     */
    public static void shellInsert(int[] arr, int d) {
        for (int i = d; i < arr.length; i++) {
            int j = i - d;
            int temp = arr[i]; // 记录要插入的数据
            while (j >= 0 && arr[j] > temp) { // 从后向前，找到比其小的数的位置
                arr[j + d] = arr[j]; // 向后挪动
                j -= d;
            }

            if (j != i - d) // 存在比其小的数
                arr[j + d] = temp;
        }
    }

    public static void shellSort(int[] arr) {
        if (arr == null || arr.length == 0)
            return ;
        int d = arr.length / 2;
        while (d >= 1) {
            shellInsert(arr, d);
            d /= 2;
        }
    }
}
```

2016-04-28 15:58



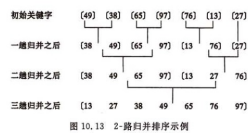
实验楼管理员 (<https://www.shiyanlou.com/user/8490>) <https://www.shiyanlou.com/vip>

(<https://www.shiyanlou.com/user/8490>)

## 归并排序

归并排序是另一种不同的排序方法，因为归并排序使用了递归分治的思想，所以理解起来比较容易。其基本思想是，先递归划分分子问题，然后合并结果。把待排序列看成由两个有序的子序列，然后合并两个子序列，然后把子序列看成由两个有序序列。。。倒着来看，其实就是先两两合并，然后四四合并。。。最终形成有序序列。空间复杂度为  $O(n)$ ，时间复杂度为  $O(n \log n)$ 。

举个栗子：



实现代码：

```
/**
 * @Description: <p>归并排序算法的实现</p>
 * @author 王旭
 * @time 2016-3-4 上午8:14:20
 */
public class MergeSort {

    public static void mergeSort(int[] arr) {
        mSort(arr, 0, arr.length-1);
    }

    /**
     * 递归分治
     * @param arr 待排数组
     * @param left 左指针
     * @param right 右指针
     */
    public static void mSort(int[] arr, int left, int right) {
        if(left >= right)
            return ;
        int mid = (left + right) / 2;

        mSort(arr, left, mid); //递归排序左边
        mSort(arr, mid+1, right); //递归排序右边
        merge(arr, left, mid, right); //合并
    }

    /**
     * 合并两个有序数组
     * @param arr 待合并数组
     * @param left 左指针
     * @param mid 中间指针
     * @param right 右指针
     */
    public static void merge(int[] arr, int left, int mid, int right) {
        //[left, mid] [mid+1, right]
        int[] temp = new int[right - left + 1]; //中间数组

        int i = left;
        int j = mid + 1;
        int k = 0;
        while(i <= mid && j <= right) {
            if(arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            }
            else {
                temp[k++] = arr[j++];
            }
        }

        while(i <= mid) {
            temp[k++] = arr[i++];
        }

        while(j <= right) {
            temp[k++] = arr[j++];
        }

        for(int p=0; p<temp.length; p++) {
            arr[left + p] = temp[p];
        }
    }
}
```



(<https://www.shiyanlou.com/user/8490>)

## 计数排序

如果在面试中有面试官要求你写一个  $O(n)$  时间复杂度的排序算法，你千万不要立刻说：这不可能！虽然前面基于比较的排序的下限是  $O(n \log n)$ 。但是确实也有线性时间复杂度的排序，只不过有前提条件，就是待排序的数要满足一定的范围的整数，而且计数排序需要比较多的辅助空间。其基本思想是，用待排序的数作为计数数组的下标，统计每个数字的个数。然后依次输出即可得到有序序列。

实现代码：

```
/**
 * @Description: <p>计数排序算法实现</p>
 * @author 王旭
 * @time 2016-3-4 下午4:52:02
 */
public class CountSort {

    public static void countSort(int[] arr) {
        if(arr == null || arr.length == 0)
            return ;

        int max = max(arr);

        int[] count = new int[max+1];
        Arrays.fill(count, 0);

        for(int i=0; i<arr.length; i++) {
            count[arr[i]] ++;
        }

        int k = 0;
        for(int i=0; i<=max; i++) {
            for(int j=0; j<count[i]; j++) {
                arr[k++] = i;
            }
        }

    }

    public static int max(int[] arr) {
        int max = Integer.MIN_VALUE;
        for(int ele : arr) {
            if(ele > max)
                max = ele;
        }

        return max;
    }

}
```

2016-04-28 15:58



(<https://www.shiyanlou.com/user/8490>)

## 桶排序

桶排序算是计数排序的一种改进和推广，但是网上有许多资料把计数排序和桶排序混为一谈。其实桶排序要比计数排序复杂许多。

对桶排序的分析和解释借鉴这位兄弟的文章（有改动）：<http://hxraid.iteye.com/blog/647759>

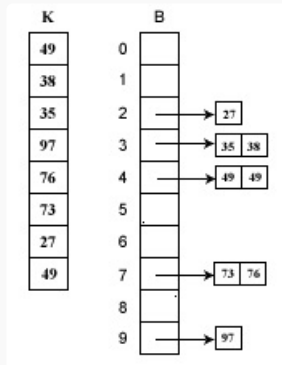
(<http://hxraid.iteye.com/blog/647759>)

**桶排序的基本思想：**

假设有一组长度为N的待排关键字序列  $K[1 \dots n]$ 。首先将这个序列划分成M个的子区间(桶)。然后基于某种映射函数，将待排序列的关键字k映射到第i个桶中(即桶数组B的下标i)，那么该关键字k就作为B[i]中的元素(每个桶B[i]都是一组大小为N/M的序列)。接着对每个桶B[i]中的所有元素进行比较排序(可以使用快排)。然后依次枚举输出  $B[0] \dots B[M]$  中的全部内容即是一个有序序列。  $\text{bindex} = f(\text{key})$  其中，bindex 为桶数组B的下标(即第bindex个桶), k为待排序列的关键字。桶排序之所以能够高效，其关键在于这个映射函数，它必须做到：如果关键字  $k_1 < k_2$ ，那么  $f(k_1) \leq f(k_2)$ 。

也就是说  $B(i)$  中的最小数据都要大于  $B(i-1)$  中最大数据。很显然，映射函数的确定与数据本身的特点有很大的关系。

举个栗子：



假如待排序列  $K = \{49, 38, 35, 97, 76, 73, 27, 49\}$ 。这些数据全部在1—100之间。因此我们定制10个桶，然后确定映射函数  $f(k) = k/10$ 。则第一个关键字49将定位到第4个桶中 ( $49/10=4$ )。依次将所有关键字全部堆入桶中，并在每个非空的桶中进行快速排序后得到如图所示。只要顺序输出每个  $B[i]$  中的数据就可以得到有序序列了。

### 桶排序分析：

桶排序利用函数的映射关系，减少了几乎所有的比较工作。实际上，桶排序的  $f(k)$  值的计算，其作用就相当于快排中划分，希尔排序中的子序列，归并排序中的子问题，已经把大量数据分割成了基本有序的数据块(桶)。然后只需要对桶中的少量数据做先进的比较排序即可。

对N个关键字进行桶排序的时间复杂度分为两个部分：

(1) 循环计算每个关键字的桶映射函数，这个时间复杂度是  $O(N)$ 。

(2) 利用先进的比较排序算法对每个桶内的所有数据进行排序，其时间复杂度为  $\sum O(N_i * \log N_i)$ 。其中  $N_i$  为第i个桶的数据量。

很显然，第(2)部分是桶排序性能好坏的决定因素。尽量减少桶内数据的数量是提高效率的唯一办法(因为基于比较排序的最好平均时间复杂度只能达到  $O(N * \log N)$  了)。因此，我们需要尽量做到下面两点：

(1) 映射函数  $f(k)$  能够将N个数据平均的分配到M个桶中，这样每个桶就有  $[N/M]$  个数据量。

(2) 尽量的增大桶的数量。极限情况下每个桶只能得到一个数据，这样就完全避开了桶内数据的“比较”排序操作。当然，做到这一点很不容易，数据量巨大的情况下， $f(k)$  函数会使得桶集合的数量巨大，空间浪费严重。这就是一个时间代价和空间代价的权衡问题了。

对于N个待排数据，M个桶，平均每个桶  $[N/M]$  个数据的桶排序平均时间复杂度为：

$$O(N) + O(M * (N/M) * \log(N/M)) = O(N + N * (\log N - \log M)) = O(N + N * \log N - N * \log M)$$

当  $N=M$  时，即极限情况下每个桶只有一个数据时。桶排序的最好效率能够达到  $O(N)$ 。

**总结：**桶排序的平均时间复杂度为线性的  $O(N+C)$ ，其中  $C = N * (\log N - \log M)$ 。如果相对于同样的N，桶数量M越大，其效率越高，最好的时间复杂度达到  $O(N)$ 。当然桶排序的空间复杂度为  $O(N+M)$ ，如果输入数据非常庞大，而桶的数量也非常多，则空间代价无疑是昂贵的。此外，桶排序是稳定的。

2016-04-28 15:59



实现代码：

(<https://www.shiyanlou.com/user/8490>)

```
/**
 * @Description:<p>桶排序算法实现</p>
 * @author 王旭
 * @time 2016-3-4 下午7:39:31
 */
public class BucketSort {

    public static void bucketSort(int[] arr) {
        if(arr == null && arr.length == 0)
            return ;

        int bucketNums = 10; //这里默认为10, 规定待排数[0,100)
        List<List<Integer>> buckets = new ArrayList<List<Integer>>(); //桶的索引

        for(int i=0; i<10; i++) {
            buckets.add(new LinkedList<Integer>()); //用链表比较合适
        }

        //划分桶
        for(int i=0; i<arr.length; i++) {
            buckets.get(f(arr[i])).add(arr[i]);
        }


        //对每个桶进行排序
        for(int i=0; i<buckets.size(); i++) {
            if(!buckets.get(i).isEmpty()) {
                Collections.sort(buckets.get(i)); //对每个桶进行快排
            }
        }

        //还原排好序的数组
        int k = 0;
        for(List<Integer> bucket : buckets) {
            for(int ele : bucket) {
                arr[k++] = ele;
            }
        }
    }

    /**
     * 映射函数
     * @param x
     * @return
     */
    public static int f(int x) {
        return x / 10;
    }
}
```

2016-04-28 15:59



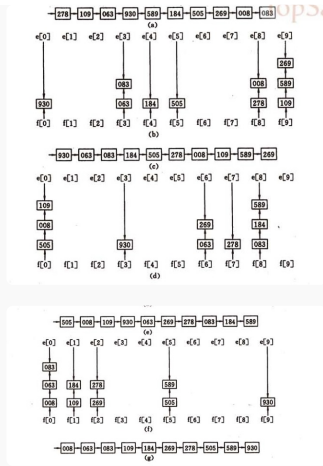
实验楼管理员 (<https://www.shiyanlou.com/user/8490>)  (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

## 基数排序

基数排序又是一种和前面排序方式不同的排序方式，基数排序不需要进行记录关键字之间的比较。基数排序是一种借助多关键字排序思想对单逻辑关键字进行排序的方法。所谓的多关键字排序就是有多多个优先级不同的关键字。比如说成绩的排序，如果两个人总分相同，则语文高的排在前面，语文成绩也相同则数学高的排在前面。。。如果对数字进行排序，那么个位、十位、百位就是不同优先级的关键字，如果要进行升序排序，那么个位、十位、百位优先级一次增加。基数排序是通过多次的收分配和收集来实现的，关键字优先级低的先进行分配和收集。

举个栗子：



2016-04-28 16:00



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) [vip](https://www.shiyanlou.com/vip)

(<https://www.shiyanlou.com/user/8490>)

实现代码：

```
/**
 * @Description: <p>基数排序算法实现</p>
 * @author 王旭
 * @time 2016-3-4 下午8:29:52
 */
public class RadixSort {

    public static void radixSort(int[] arr) {
        if(arr == null && arr.length == 0)
            return ;

        int maxBit = getMaxBit(arr);

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

            List<List<Integer>> buf = distribute(arr, i); //分配
            collecte(arr, buf); //收集
        }

    }

    /**
     * 分配
     * @param arr 待分配数组
     * @param iBit 要分配第几位
     * @return
     */
    public static List<List<Integer>> distribute(int[] arr, int iBit) {
        List<List<Integer>> buf = new ArrayList<List<Integer>>();
        for(int j=0; j<10; j++) {
            buf.add(new LinkedList<Integer>());
        }
        for(int i=0; i<arr.length; i++) {
            buf.get(getNBit(arr[i], iBit)).add(arr[i]);
        }
        return buf;
    }

    /**
     * 收集
     * @param arr 把分配的数据收集到arr中
     * @param buf
     */
    public static void collecte(int[] arr, List<List<Integer>> buf) {
        int k = 0;
        for(List<Integer> bucket : buf) {
            for(int ele : bucket) {
                arr[k++] = ele;
            }
        }
    }
}
```

```

    }

    /**
     * 获取最大位数
     * @param x
     * @return
     */
    public static int getMaxBit(int[] arr) {
        int max = Integer.MIN_VALUE;
        for(int ele : arr) {
            int len = (ele+"").length();
            if(len > max)
                max = len;
        }
        return max;
    }

    /**
     * 获取x的第n位，如果没有则为0。
     * @param x
     * @param n
     * @return
     */
    public static int getNBit(int x, int n) {

        String sx = x + "";
        if(sx.length() < n)
            return 0;
        else
            return sx.charAt(sx.length()-n) - '0';
    }
}

```

2016-04-28 16:00



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) <https://www.shiyanlou.com/vip>

(<https://www.shiyanlou.com/user/8490>)

## 总结

在前面的介绍和分析中我们提到了冒泡排序、选择排序、插入排序三种简单的排序及其变种快速排序、堆排序、希尔排序三种比较高效的排序。后面我们又分析了基于分治递归思想的归并排序还有计数排序、桶排序、基数排序三种线性排序。我们可以知道排序算法要么简单有效，要么是利用简单排序的特点加以改进，要么是以空间换取时间在特定情况下的高效排序。但是这些排序方法都不是固定不变的，需要结合具体的需求和场景来选择甚至组合使用。才能达到高效稳定的目的。没有最好的排序，只有最适合的排序。

下面就总结一下排序算法的各自的使用场景和适用场合。

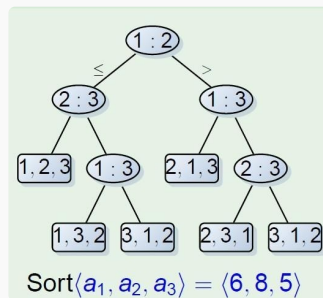
排序方法	平均时间	最坏情况	辅助空间
简单排序	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(n \log n)$	$O(n^2)$	$O(\log n)$
堆排序	$O(n \log n)$	$O(n \log n)$	$O(1)$
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n)$
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(n)$

1. 从平均时间来看，快速排序是效率最高的，但快速排序在最坏情况下的时间性能不如堆排序和归并排序。而后者相比较的结果是，在n较大时归并排序使用时间较少，但使用辅助空间较多。
2. 上面说的简单排序包括除希尔排序之外的所有冒泡排序、插入排序、简单选择排序。其中直接插入排序最简单，但序列基本有序或者n较小时，直接插入排序是好的方法，因此常将它和其他的排序方法，如快速排序、归并排序等结合在一起使用。
3. 基数排序的时间复杂度也可以写成  $O(d \cdot n)$ 。因此它最适用于n值很大而关键字较小的序列。若关键字也很大，而序列中大多数记录的最高关键字均不同，则亦可先按最高关键字不同，将序列分成若干小的子序列，而后进行直接插入排序。
4. 从方法的稳定性来比较，基数排序是稳定的内排方法，所有时间复杂度为  $O(n^2)$  的简单排序也是稳定的。但是快速排序、堆排序、希尔排序等时间性能较好的排序方法都是不稳定的。稳定性需要根据具体需求选择。

5. 上面的算法实现大多数是使用线性存储结构，像插入排序这种算法用链表实现更好，省去了移动元素的时间。具体的存储结构在具体的实现版本中也是不同的。

附：基于比较排序算法时间下限为  $O(n \log n)$  的证明：


基于比较排序下限的证明是通过决策树证明的，决策树的高度  $\Omega(n \lg n)$ ，这样就得出了比较排序的下限。



首先要引入决策树。首先决策树是一颗二叉树，每个节点表示元素之间一组可能的排序，它予以京进行的比较相一致，比较的结果是树的边。先来说明一些二叉树的性质，令T是深度为d的二叉树，则T最多有 $2^d$ 片树叶。具有L片树叶的二叉树的深度至少是 $\log L$ 。所以，对n个元素排序的决策树必然有 $n!$ 片树叶（因为n个数有 $n!$ 种不同的大小关系），所以决策树的深度至少是 $\log(n!)$ ，即至少需要 $\log(n!)$ 次比较。而 $\log(n!) = \log n + \log(n-1) + \log(n-2) + \dots + \log 2 + \log 1 \geq \log n + \log(n-1) + \log(n-2) + \dots + \log(n/2) \geq (n/2) \log(n/2) \geq (n/2) \log n - n/2 = O(n \log n)$  所以只用到比较的排序算法最低时间复杂度是 $O(n \log n)$ 。

2016-04-28 16:01



实验楼管理员 (<https://www.shiyanlou.com/user/8490>)  (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

参考资料：

- 《数据结构》 严蔚敏 吴伟民 编著

- 桶排序分析：<http://hxraid.iteye.com/blog/647759> (<http://hxraid.iteye.com/blog/647759>)
- 部分排序算法分析与介绍：<http://www.cnblogs.com/weixliu/archive/2012/12/23/2829671.html> (<http://www.cnblogs.com/weixliu/archive/2012/12/23/2829671.html>)

作者：大海里的太阳 (<http://www.cnblogs.com/wxisme/>)

出处：<http://www.cnblogs.com/wxisme/> (<http://www.cnblogs.com/wxisme/>)

本文地址：<http://www.cnblogs.com/wxisme/p/5243631.html> (<http://www.cnblogs.com/wxisme/p/5243631.html>)

声明：本文版权归作者和博客园共有，欢迎转载，但未经作者同意必须保留该声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。

2016-04-28 16:02

登录 (<https://www.shiyanlou.com/login?next=/questions/3931>)后回答问题

我要提问

标签

Linux (<https://www.shiyanlou.com/questions/?tag=Linux>)

课程相关 (<https://www.shiyanlou.com/questions/?tag=课程相关>)



Python (<https://www.shiyanlou.com/questions/?tag=Python>)    实验环境 (<https://www.shiyanlou.com/questions/?tag=实验环境>)

C/C++ (<https://www.shiyanlou.com/questions/?tag=C/C++>)    技术分享 (<https://www.shiyanlou.com/questions/?tag=技术分享>)

课程需求 (<https://www.shiyanlou.com/questions/?tag=课程需求>)    功能建议 (<https://www.shiyanlou.com/questions/?tag=功能建议>)

Java (<https://www.shiyanlou.com/questions/?tag=Java>)    其他 (<https://www.shiyanlou.com/questions/?tag=其他>)

Web (<https://www.shiyanlou.com/questions/?tag=Web>)    Hadoop (<https://www.shiyanlou.com/questions/?tag=Hadoop>)

NodeJS (<https://www.shiyanlou.com/questions/?tag=NodeJS>)    SQL (<https://www.shiyanlou.com/questions/?tag=SQL>)

PHP (<https://www.shiyanlou.com/questions/?tag=PHP>)    Shell (<https://www.shiyanlou.com/questions/?tag=Shell>)

常见问题 (<https://www.shiyanlou.com/questions/?tag=常见问题>)    Git (<https://www.shiyanlou.com/questions/?tag=Git>)

HTML (<https://www.shiyanlou.com/questions/?tag=HTML>)    网络 (<https://www.shiyanlou.com/questions/?tag=网络>)

HTML5 (<https://www.shiyanlou.com/questions/?tag=HTML5>)    信息安全 (<https://www.shiyanlou.com/questions/?tag=信息安全>)

Android (<https://www.shiyanlou.com/questions/?tag=Android>)    GO (<https://www.shiyanlou.com/questions/?tag=GO>)

NoSQL (<https://www.shiyanlou.com/questions/?tag=NoSQL>)    Ruby (<https://www.shiyanlou.com/questions/?tag=Ruby>)

训练营 (<https://www.shiyanlou.com/questions/?tag=训练营>)    Perl (<https://www.shiyanlou.com/questions/?tag=Perl>)



实验楼客户端  
即开即用  
会员专属

(<https://www.shiyanlou.com/vip>)

## 相关问题

C/C++的mem函数和strcpy函数的区别和应用 (<https://www.shiyanlou.com/questions/5274>)

python之线程、进程和协程 (<https://www.shiyanlou.com/questions/5107>)

从底层理解Python的执行 (<https://www.shiyanlou.com/questions/5247>)

20个为前端开发者准备的文档和指南（2） (<https://www.shiyanlou.com/questions/5215>)

震惊小伙伴的单行代码—Python篇 (<https://www.shiyanlou.com/questions/4157>)

九个Console命令，让js调试更简单 (<https://www.shiyanlou.com/questions/5178>)

10款最佳PHP自动化测试框架 (<https://www.shiyanlou.com/questions/2211>)

Git 远程操作的正确姿势 (<https://www.shiyanlou.com/questions/5134>)

JavaScript异步编程解决方案笔记 (<https://www.shiyanlou.com/questions/5094>)

Vim 起步的五个技巧 (<https://www.shiyanlou.com/questions/5072>)



动手做实验，轻松学IT。

(<http://weibo.com/shiyanlou2013>)

## 公司

关于我们 (<https://www.shiyanlou.com/aboutus>)

联系我们 (<https://www.shiyanlou.com/contact>)

加入我们 (<http://www.simplecloud.cn/jobs.html>)

技术博客 (<https://blog.shiyanlou.com/>)

## 合作

我要投稿 (<https://www.shiyanlou.com/contribute>)

教师合作 (<https://www.shiyanlou.com/labs>)

高校合作 (<https://www.shiyanlou.com/edu/>)

友情链接 (<https://www.shiyanlou.com/friends>)

## 服务

实战训练营 (<https://www.shiyanlou.com/bootcamp/>)

会员服务 (<https://www.shiyanlou.com/vip>)

实验报告 (<https://www.shiyanlou.com/courses/reports>)

常见问题 (<https://www.shiyanlou.com/questions/?tag=常见问题>)

隐私条款 (<https://www.shiyanlou.com/privacy>)

## 学习路径

Python学习路径 (<https://www.shiyanlou.com/paths/python>)

Linux学习路径 (<https://www.shiyanlou.com/paths/linuxdev>)

大数据学习路径 (<https://www.shiyanlou.com/paths/bigdata>)

Java学习路径 (<https://www.shiyanlou.com/paths/java>)

PHP学习路径 (<https://www.shiyanlou.com/paths/php>)

全部 (<https://www.shiyanlou.com/paths/>)

Copyright @2013-2016 实验楼在线教育

蜀ICP备13019762号 (<http://www.miibeian.gov.cn/>) 站长统计 ([http://www.cnzz.com/stat/website.php?web\\_id=5902315](http://www.cnzz.com/stat/website.php?web_id=5902315))