

十大经典排序算法

十大排序中的稳定排序

十大排序中的非稳定排序

0 冒泡排序

1. 算法步骤
2. 动图演示
3. 什么时候最快
4. 什么时候最慢
5. 实现

1 选择排序

1. 算法步骤
2. 动图演示

2 插入排序

1. 算法步骤
2. 动图演示

3 希尔排序

1. 算法步骤

4 归并排序

2. 算法步骤
3. 动图演示

5 快速排序

1. 算法步骤
2. 动图演示

6 堆排序

1. 算法步骤
2. 动图演示

7 计数排序

1. 动图演示

8 桶排序

1. 什么时候最快

2. 什么时候最慢

9 基数排序

1. 基数排序 vs 计数排序 vs 桶排序

2. LSD 基数排序动图演示

排序算法	时间复杂度	空间复杂度	稳定性	备注
冒泡排序	$O(n^2)$	$O(1)$	稳定	简单
选择排序	$O(n^2)$	$O(1)$	不稳定	简单
插入排序	$O(n^2)$	$O(1)$	稳定	简单
希尔排序	$O(n^2)$	$O(1)$	不稳定	简单
归并排序	$O(n \log n)$	$O(n)$	稳定	递归
快速排序	$O(n \log n)$	$O(\log n)$	不稳定	递归
堆排序	$O(n \log n)$	$O(1)$	不稳定	递归
计数排序	$O(n)$	$O(n)$	稳定	非比较
桶排序	$O(n)$	$O(n)$	稳定	非比较
基数排序	$O(n \log r)$	$O(n)$	稳定	非比较

【JAVA十大经典排序算法】_Rita_zzf的博客-CSDN博客_java十大排序

目录排序算法说明排序的定义术语说明算法总结算法分类比较和非比较的区别冒泡排序 (Bubble Sort...

https://blog.csdn.net/Rita_zf/article/details/123843920

- 1、稳定排序：如果 a 原本在 b 的前面，且 $a == b$ ，排序之后 a 仍然在 b 的前面，则为稳定排序。
- 2、非稳定排序：如果 a 原本在 b 的前面，且 $a == b$ ，排序之后 a 可能不在 b 的前面，则为非稳定排序。
- 3、原地排序：原地排序就是指在排序过程中不申请多余的存储空间，只利用原来存储待排数据的存储空间进行比较和交换的数据排序。
- 4、非原地排序：需要利用额外的数组来辅助排序。
- 5、时间复杂度：一个算法执行所消耗的时间。
- 6、空间复杂度：运行完一个算法所需的内存大小

算法	时间复杂度			空间复杂度
	最佳	平均	最差	最差
快速排序	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(\log(n))$
归并排序	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$O(n)$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
堆排序	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
希尔排序	$O(n)$	$O((n \log(n))^2)$	$O((n \log(n))^2)$	$O(1)$
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n)$
基数排序	$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$

十大排序中的稳定排序

冒泡排序 (bubble sort) — $O(n^2)$ 插入排序 (insertion sort) — $O(n^2)$ 归并排序 (merge sort) — $O(n \log n)$

十大排序中的非稳定排序

面试考察中一般问快排，选择，希尔，堆这几种非稳定排序

选择排序 (selection sort) — $O(n^2)$ 希尔排序 (shell sort) — $O(n \log n)$ 堆排序 (heapsort) — $O(n \log n)$ 快速排序 (quicksort) — $O(n \log n)$

0 冒泡排序

冒泡排序 (Bubble Sort) 也是一种简单直观的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。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。

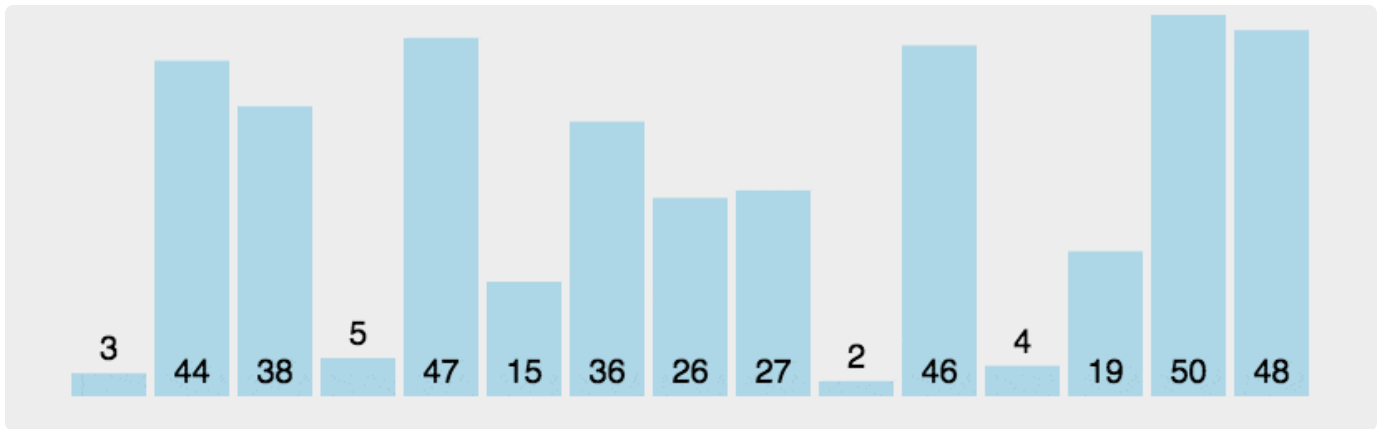
作为最简单的排序算法之一，冒泡排序给我的感觉就像 Abandon 在单词书里出现的感觉一样，每次都在第一页第一位，所以最熟悉。冒泡排序还有一种优化算法，就是立一个 flag，当在一趟

序列遍历中元素没有发生交换，则证明该序列已经有序。但这种改进对于提升性能来说并没有什么太大作用。

1. 算法步骤

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
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。这步做完后，最后的元素会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

2. 动图演示



3. 什么时候最快

当输入的数据已经是正序时（都已经是正序了，我还要你冒泡排序有何用啊）。

4. 什么时候最慢

当输入的数据是反序时（写一个 for 循环反序输出数据不就行了，干嘛要用你冒泡排序呢，我是闲的吗）。

5. 实现

```
1 public class BubbleSort implements IArraySort {
2
3     @Override
4     public int[] sort(int[] sourceArray) throws Exception {
5         // 对 arr 进行拷贝, 不改变参数内容
6         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
7
8         for (int i = 1; i < arr.length; i++) {
9             // 设定一个标记, 若为true, 则表示此次循环没有进行交换, 也就是待排序列已经
            有序, 排序已经完成。
10             boolean flag = true;
11
12             for (int j = 0; j < arr.length - i; j++) {
13                 if (arr[j] > arr[j + 1]) {
14                     int tmp = arr[j];
15                     arr[j] = arr[j + 1];
16                     arr[j + 1] = tmp;
17
18                     flag = false;
19                 }
20             }
21
22             if (flag) {
23                 break;
24             }
25         }
26         return arr;
27     }
28 }
29
```

```
1 def bubbleSort(arr):
2     for i in range(1, len(arr)):
3         for j in range(0, len(arr)-i):
4             if arr[j] > arr[j+1]:
5                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
6     return arr
```

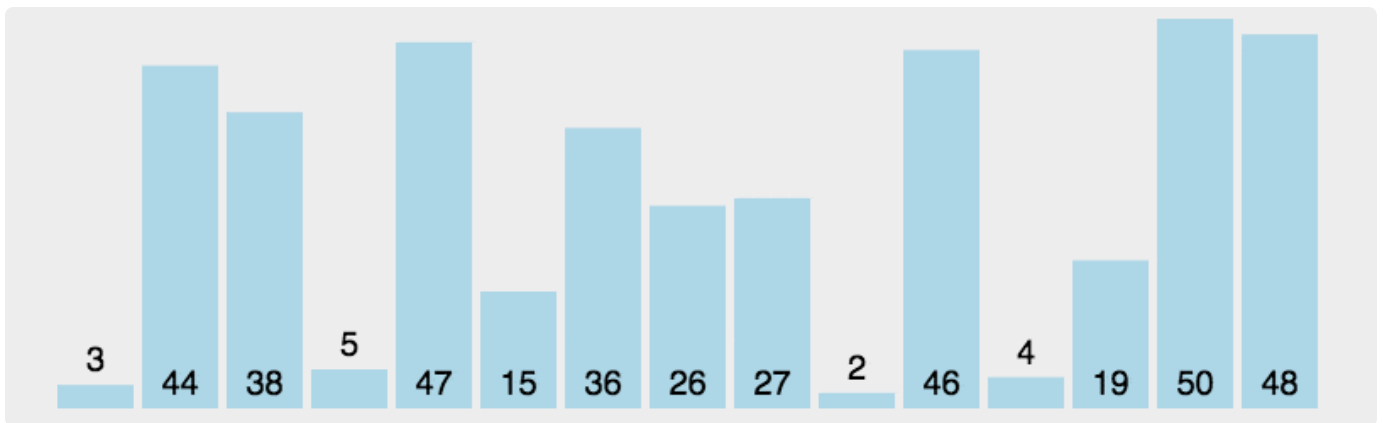
1 选择排序

选择排序是一种简单直观的排序算法，无论什么数据进去都是 $O(n^2)$ 的时间复杂度。所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。

1. 算法步骤

1. 首先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. 再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。

2. 动图演示



```
1 public class SelectionSort implements IArraySort {
2
3     @Override
4     public int[] sort(int[] sourceArray) throws Exception {
5         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
6
7         // 总共要经过 N-1 轮比较
8         for (int i = 0; i < arr.length - 1; i++) {
9             int min = i;
10
11             // 每轮需要比较的次数 N-i
12             for (int j = i + 1; j < arr.length; j++) {
13                 if (arr[j] < arr[min]) {
14                     // 记录目前能找到的最小值元素的下标
15                     min = j;
16                 }
17             }
18
19             // 将找到的最小值和i位置所在的值进行交换
20             if (i != min) {
21                 int tmp = arr[i];
22                 arr[i] = arr[min];
23                 arr[min] = tmp;
24             }
25
26         }
27         return arr;
28     }
29 }
```

```
1 def selectionSort(arr):
2     for i in range(len(arr) - 1):
3         # 记录最小数的索引
4         minIndex = i
5         for j in range(i + 1, len(arr)):
6             if arr[j] < arr[minIndex]:
7                 minIndex = j
8         # i 不是最小数时，将 i 和最小数进行交换
9         if i != minIndex:
10             arr[i], arr[minIndex] = arr[minIndex], arr[i]
11     return arr
```

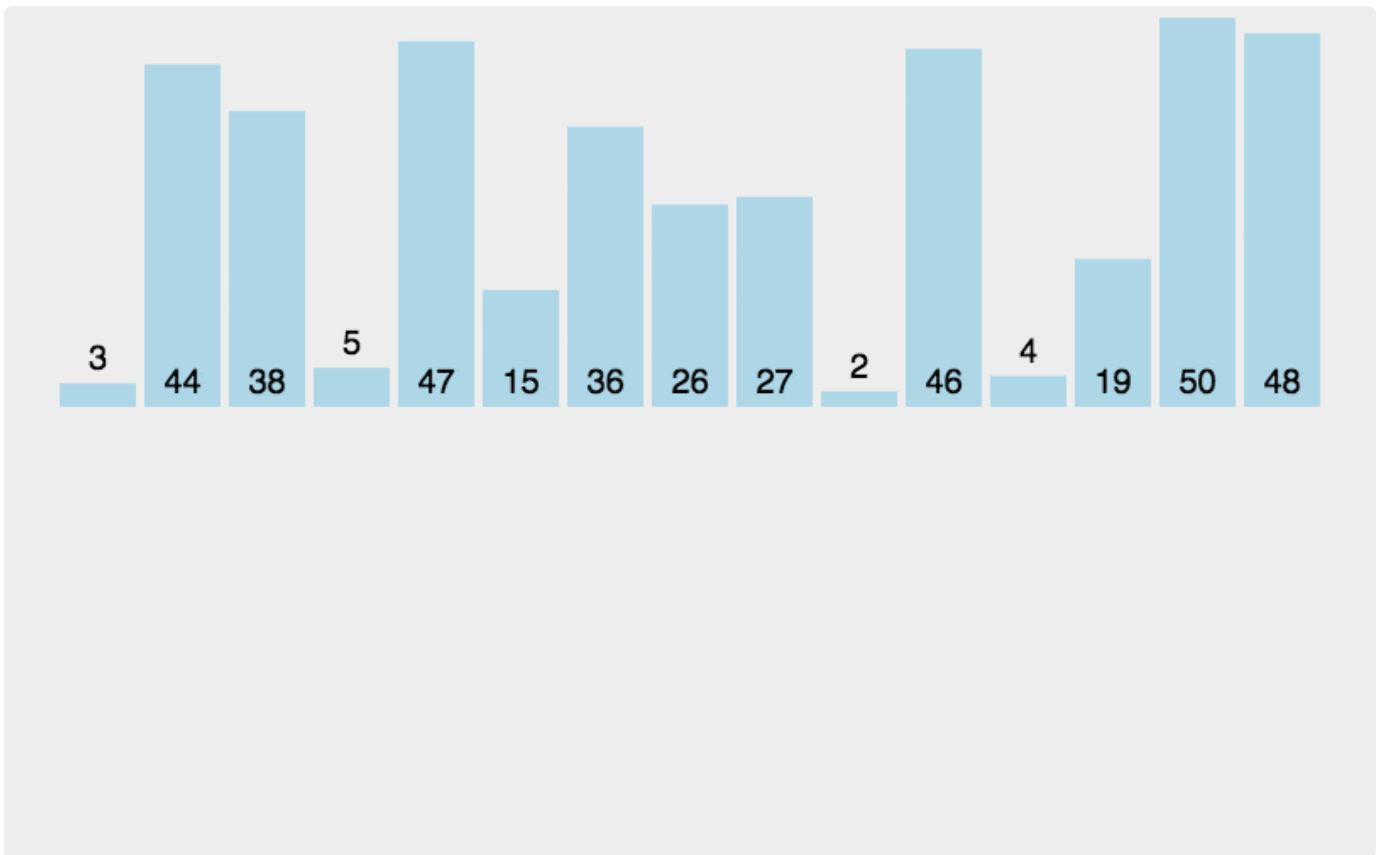
2 插入排序

插入排序的代码实现虽然没有冒泡排序和选择排序那么简单粗暴，但它的原理应该是最容易理解的了，因为只要打过扑克牌的人都应该能够秒懂。插入排序是一种最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。插入排序和冒泡排序一样，也有一种优化算法，叫做拆半插入。

1. 算法步骤

1. 将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。
2. 从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置。（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）

2. 动图演示




```
1 public class InsertSort implements IArraySort {
2
3     @Override
4     public int[] sort(int[] sourceArray) throws Exception {
5         // 对 arr 进行拷贝，不改变参数内容
6         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
7
8         // 从下标为1的元素开始选择合适的位置插入，因为下标为0的只有一个元素，默认是有
        序的
9         for (int i = 1; i < arr.length; i++) {
10
11             // 记录要插入的数据
12             int tmp = arr[i];
13
14             // 从已经排序的序列最右边的开始比较，找到比其小的数
15             int j = i;
16             while (j > 0 && tmp < arr[j - 1]) {
17                 arr[j] = arr[j - 1];
18                 j--;
19             }
20
21             // 存在比其小的数，插入
22             if (j != i) {
23                 arr[j] = tmp;
24             }
25
26         }
27         return arr;
28     }
29 }
```

```
1 def insertionSort(arr):
2     for i in range(len(arr)):
3         preIndex = i-1
4         current = arr[i]
5         while preIndex >= 0 and arr[preIndex] > current:
6             arr[preIndex+1] = arr[preIndex]
7             preIndex-=1
8         arr[preIndex+1] = current
9     return arr
```

3 希尔排序

希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本。但希尔排序是非稳定排序算法。

希尔排序是基于插入排序的以下两点性质而提出改进方法的：

- 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率；
- 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位；

希尔排序的基本思想是：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，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

1. 算法步骤

1. 选择一个增量序列 t_1, t_2, \dots, t_k ，其中 $t_i > t_j, t_k = 1$ ；
2. 按增量序列个数 k ，对序列进行 k 趟排序；
3. 每趟排序，根据对应的增量 t_i ，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。

```
1 public class ShellSort implements IArraySort {
2
3     @Override
4     public int[] sort(int[] sourceArray) throws Exception {
5         // 对 arr 进行拷贝, 不改变参数内容
6         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
7
8         int gap = 1;
9         while (gap < arr.length/3) {
10             gap = gap * 3 + 1;
11         }
12
13         while (gap > 0) {
14             for (int i = gap; i < arr.length; i++) {
15                 int tmp = arr[i];
16                 int j = i - gap;
17                 while (j >= 0 && arr[j] > tmp) {
18                     arr[j + gap] = arr[j];
19                     j -= gap;
20                 }
21                 arr[j + gap] = tmp;
22             }
23             gap = (int) Math.floor(gap / 3);
24         }
25
26         return arr;
27     }
28 }
```

```
1 def shellSort(arr):
2     import math
3     gap=1
4     while(gap < len(arr)/3):
5         gap = gap*3+1
6         while gap > 0:
7             for i in range(gap,len(arr)):
8                 temp = arr[i]
9                 j = i-gap
10                while j >=0 and arr[j] > temp:
11                    arr[j+gap]=arr[j]
12                    j-=gap
13                arr[j+gap] = temp
14                gap = math.floor(gap/3)
15            return arr
16 }
```

4 归并排序

归并排序（Merge sort）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（Divide and Conquer）的一个非常典型的应用。

作为一种典型的分而治之思想的算法应用，归并排序的实现由两种方法：

- 自上而下的递归（所有递归的方法都可以用迭代重写，所以就有了第 2 种方法）；
- 自下而上的迭代；

在《数据结构与算法 JavaScript 描述》中，作者给出了自下而上的迭代方法。但是对于递归法，作者却认为：

However, it is not possible to do so in JavaScript, as the recursion goes too deep for the language to handle.

然而，在 JavaScript 中这种方式不太可行，因为这个算法的递归深度对它来讲太深了。

说实话，我不太理解这句话。意思是 JavaScript 编译器内存太小，递归太深容易造成内存溢出吗？还望有大神能够指教。

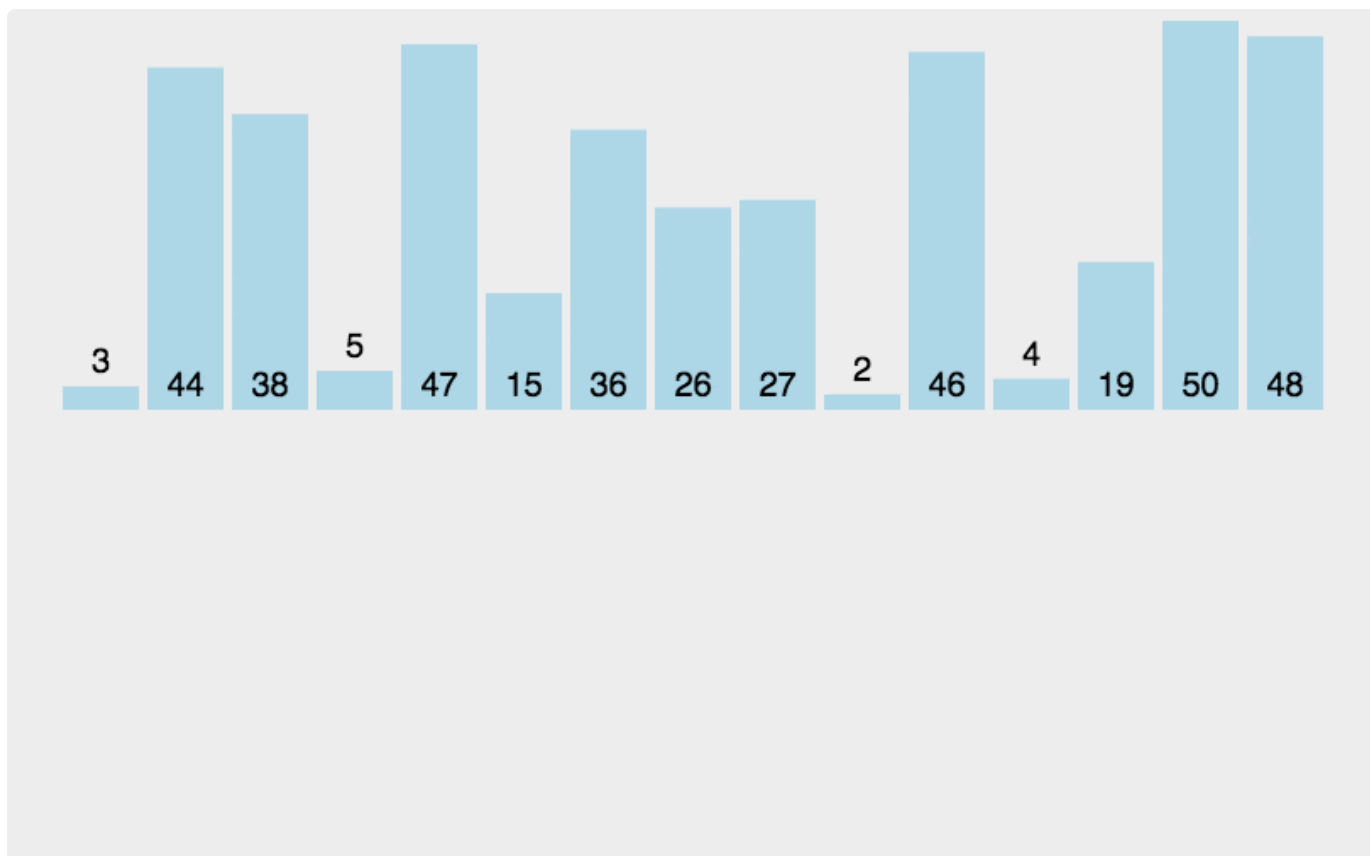
和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是 $O(n\log n)$ 的时间复杂度。代价是需要额外的内存空间。

2. 算法步骤

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 直到某一指针达到序列尾；
5. 将另一序列剩下的所有元素直接复制到合并序列尾。

3. 动图演示



```
1 public class MergeSort implements IArraySort {
2
3     @Override
4     public int[] sort(int[] sourceArray) throws Exception {
5         // 对 arr 进行拷贝, 不改变参数内容
6         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
7
8         if (arr.length < 2) {
9             return arr;
10        }
11        int middle = (int) Math.floor(arr.length / 2);
12
13        int[] left = Arrays.copyOfRange(arr, 0, middle);
14        int[] right = Arrays.copyOfRange(arr, middle, arr.length);
15
16        return merge(sort(left), sort(right));
17    }
18
19    protected int[] merge(int[] left, int[] right) {
20        int[] result = new int[left.length + right.length];
21        int i = 0;
22        while (left.length > 0 && right.length > 0) {
23            if (left[0] <= right[0]) {
24                result[i++] = left[0];
25                left = Arrays.copyOfRange(left, 1, left.length);
26            } else {
27                result[i++] = right[0];
28                right = Arrays.copyOfRange(right, 1, right.length);
29            }
30        }
31
32        while (left.length > 0) {
33            result[i++] = left[0];
34            left = Arrays.copyOfRange(left, 1, left.length);
35        }
36
37        while (right.length > 0) {
38            result[i++] = right[0];
39            right = Arrays.copyOfRange(right, 1, right.length);
40        }
41
42        return result;
43    }
44
45 }
```

```
1 def mergeSort(arr):
2     import math
3     if(len(arr)<2):
4         return arr
5     middle = math.floor(len(arr)/2)
6     left, right = arr[0:middle], arr[middle:]
7     return merge(mergeSort(left), mergeSort(right))
8
9 def merge(left, right):
10    result = []
11    while left and right:
12        if left[0] <= right[0]:
13            result.append(left.pop(0));
14        else:
15            result.append(right.pop(0));
16    while left:
17        result.append(left.pop(0));
18    while right:
19        result.append(right.pop(0));
20    return result
```

5 快速排序

快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序 n 个项目要 $O(n \log n)$ 次比较。在最坏状况下则需要 $O(n^2)$ 次比较，但这种状况并不常见。事实上，快速排序通常明显比其他 $O(n \log n)$ 算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来。

快速排序使用分治法（Divide and conquer）策略来把一个串行（list）分为两个子串行（sub-lists）。

快速排序又是一种分而治之思想在排序算法上的典型应用。本质上来看，快速排序应该算是在冒泡排序基础上的递归分治法。

快速排序的名字起的是简单粗暴，因为一听到这个名字你就知道它存在的意义，就是快，而且效率高！它是处理大数据最快的排序算法之一了。虽然 Worst Case 的时间复杂度达到了 $O(n^2)$ ，但是人家就是优秀，在大多数情况下都比平均时间复杂度为 $O(n \log n)$ 的排序算法表现要更好，可是这是为什么呢，我也不知道。好在我的强迫症又犯了，查了 N 多资料终于在《算法艺术与信息学竞赛》上找到了满意的答案：

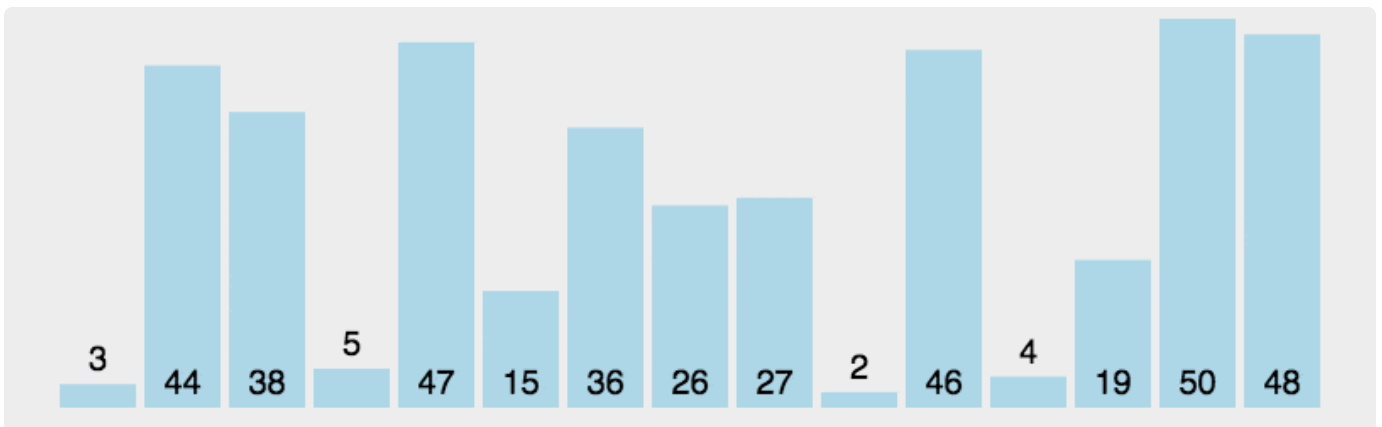
快速排序的最坏运行情况是 $O(n^2)$ ，比如说顺序数列的快排。但它的平摊期望时间是 $O(n \log n)$ ，且 $O(n \log n)$ 记号中隐含的常数因子很小，比复杂度稳定等于 $O(n \log n)$ 的归并排序要小很多。所以，对绝大多数顺序性较弱的随机数列而言，快速排序总是优于归并排序。

1. 算法步骤

1. 从数列中挑出一个元素，称为“基准” (pivot)；
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，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 (recursive) 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；

递归的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递归下去，但是这个算法总会退出，因为在每次的迭代 (iteration) 中，它至少会把一个元素摆到它最后的位置去。

2. 动图演示




```
1 public static void quickSort(int[] array, int low, int high) {
2
3     /**
4      * 分析:
5      * 1. 选定一个基准值, array[low]
6      * 2. 右指针从右向左遍历high--, 查找比基准值小的数据, 左指针从左向右low++, 查
       找比基准值大的数据
7      * 3. 如果指针未相遇, 交换左右两值位置, 如果指针相遇, 则替换基准值的位置
8      * 4. 左递归, 右递归
9      */
10    // 方法退出条件, 指针相遇或错过
11    if (low >= high) {
12        return;
13    }
14    // 1. 指定基准值和左右指针记录位置
15    int pivot = array[low];
16    int l = low;
17    int r = high;
18    int temp = 0;
19    // 2. 遍历条件, 左右指针位置
20    while (l < r) {
21        // 2.1 右侧遍历
22        while (l < r && array[r] >= pivot) {
23            r--;
24        }
25        // 2.2 左侧遍历
26        while (l < r && array[l] <= pivot) {
27            l++;
28        }
29        // 2.3 l指针还在r指针右侧, 尚未相遇
30        if (l < r) {
31            temp = array[l];
32            array[l] = array[r];
33            array[r] = temp;
34        }
35    }
36    // 3. 当左右指针相遇, 交换基准值位置
37    array[low] = array[l];
38    array[l] = pivot;
39    // 4. 根据条件左侧递归遍历
40    if (low < l) {
41        quickSort(array, low, l - 1);
42    }
43    // 5. 根据条件右侧递归遍历
44    if (r < high) {
```

```

45         quickSort(array, r + 1, high);
46     }
47
48 }
49
50 //测试
51 public static void main(String[] args) {
52     int[] arr = {-9, 78, 0, 0, 1, 0, 3, -1, 23, -56, 7};
53     quickSort(arr, 0, arr.length - 1);
54     System.out.println(Arrays.toString(arr));
55 }
56
57

```

Java | 复制代码

```

1  def quickSort(arr, left=None, right=None):
2      left = 0 if not isinstance(left,(int, float)) else left
3      right = len(arr)-1 if not isinstance(right,(int, float)) else right
4      if left < right:
5          partitionIndex = partition(arr, left, right)
6          quickSort(arr, left, partitionIndex-1)
7          quickSort(arr, partitionIndex+1, right)
8      return arr
9
10 def partition(arr, left, right):
11     pivot = left
12     index = pivot+1
13     i = index
14     while i <= right:
15         if arr[i] < arr[pivot]:
16             swap(arr, i, index)
17             index+=1
18         i+=1
19     swap(arr,pivot,index-1)
20     return index-1
21
22 def swap(arr, i, j):
23     arr[i], arr[j] = arr[j], arr[i]

```

6 堆排序

堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

堆排序可以说是一种利用堆的概念来排序的选择排序。分为两种方法：

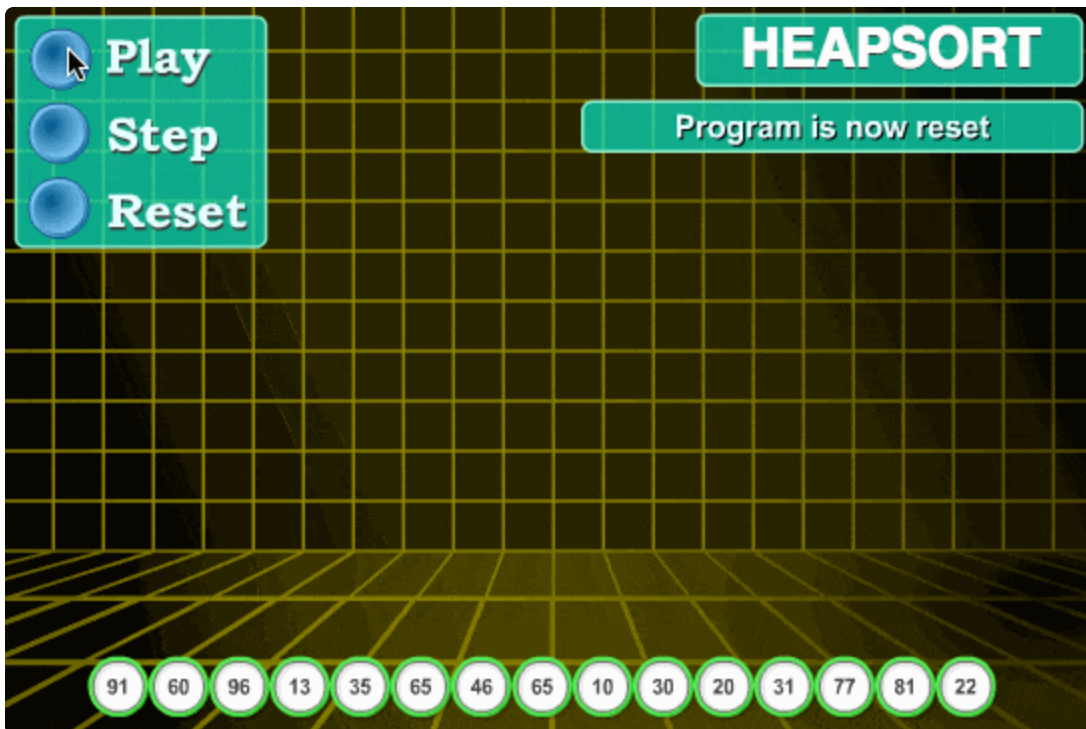
1. 大顶堆：每个节点的值都大于或等于其子节点的值，在堆排序算法中用于升序排列；
2. 小顶堆：每个节点的值都小于或等于其子节点的值，在堆排序算法中用于降序排列；

堆排序的平均时间复杂度为 $O(n\log n)$ 。

1. 算法步骤

1. 将待排序序列构建成为一个堆 $H[0.....n-1]$ ，根据（升序降序需求）选择大顶堆或小顶堆；
2. 把堆首（最大值）和堆尾互换；
3. 把堆的尺寸缩小 1，并调用 $\text{shift_down}(0)$ ，目的是把新的数组顶端数据调整到相应位置；
4. 重复步骤 2，直到堆的尺寸为 1。

2. 动图演示



```
1 public class HeapSort implements IArraySort {
2
3     @Override
4     public int[] sort(int[] sourceArray) throws Exception {
5         // 对 arr 进行拷贝, 不改变参数内容
6         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
7
8         int len = arr.length;
9
10        buildMaxHeap(arr, len);
11
12        for (int i = len - 1; i > 0; i--) {
13            swap(arr, 0, i);
14            len--;
15            heapify(arr, 0, len);
16        }
17        return arr;
18    }
19
20    private void buildMaxHeap(int[] arr, int len) {
21        for (int i = (int) Math.floor(len / 2); i >= 0; i--) {
22            heapify(arr, i, len);
23        }
24    }
25
26    private void heapify(int[] arr, int i, int len) {
27        int left = 2 * i + 1;
28        int right = 2 * i + 2;
29        int largest = i;
30
31        if (left < len && arr[left] > arr[largest]) {
32            largest = left;
33        }
34
35        if (right < len && arr[right] > arr[largest]) {
36            largest = right;
37        }
38
39        if (largest != i) {
40            swap(arr, i, largest);
41            heapify(arr, largest, len);
42        }
43    }
44
45    private void swap(int[] arr, int i, int j) {
```

```

46         int temp = arr[i];
47         arr[i] = arr[j];
48         arr[j] = temp;
49     }
50 }
51 }

```

Python | 复制代码

```

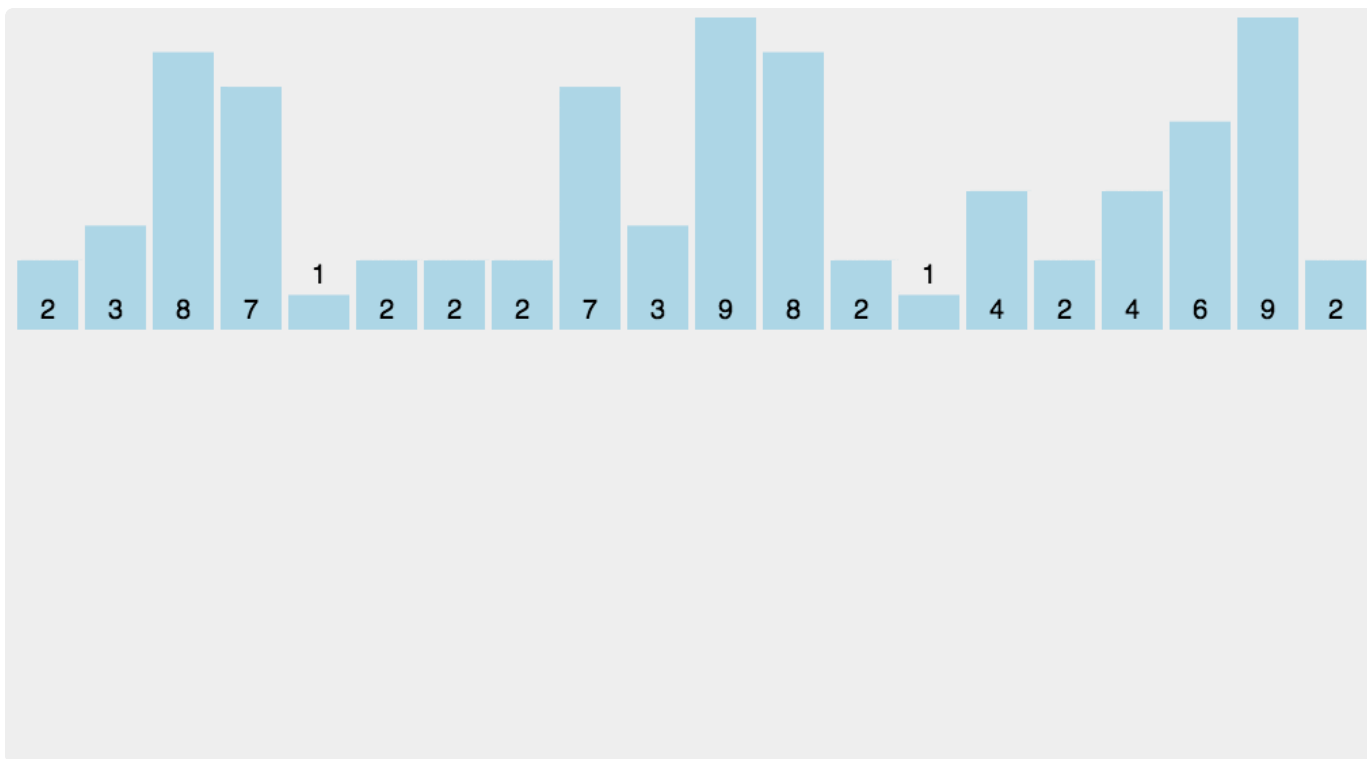
1  def buildMaxHeap(arr):
2      import math
3      for i in range(math.floor(len(arr)/2), -1, -1):
4          heapify(arr, i)
5
6  def heapify(arr, i):
7      left = 2*i+1
8      right = 2*i+2
9      largest = i
10     if left < arrLen and arr[left] > arr[largest]:
11         largest = left
12     if right < arrLen and arr[right] > arr[largest]:
13         largest = right
14
15     if largest != i:
16         swap(arr, i, largest)
17         heapify(arr, largest)
18
19  def swap(arr, i, j):
20      arr[i], arr[j] = arr[j], arr[i]
21
22  def heapSort(arr):
23      global arrLen
24      arrLen = len(arr)
25      buildMaxHeap(arr)
26      for i in range(len(arr)-1, 0, -1):
27          swap(arr, 0, i)
28          arrLen -= 1
29          heapify(arr, 0)
30      return arr

```

7 计数排序

计数排序的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。

1. 动图演示



```
1 public class CountingSort implements IArraySort {
2
3     @Override
4     public int[] sort(int[] sourceArray) throws Exception {
5         // 对 arr 进行拷贝, 不改变参数内容
6         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
7
8         int maxValue = getMaxValue(arr);
9
10        return countingSort(arr, maxValue);
11    }
12
13    private int[] countingSort(int[] arr, int maxValue) {
14        int bucketLen = maxValue + 1;
15        int[] bucket = new int[bucketLen];
16
17        for (int value : arr) {
18            bucket[value]++;
19        }
20
21        int sortedIndex = 0;
22        for (int j = 0; j < bucketLen; j++) {
23            while (bucket[j] > 0) {
24                arr[sortedIndex++] = j;
25                bucket[j]--;
26            }
27        }
28        return arr;
29    }
30
31    private int getMaxValue(int[] arr) {
32        int maxValue = arr[0];
33        for (int value : arr) {
34            if (maxValue < value) {
35                maxValue = value;
36            }
37        }
38        return maxValue;
39    }
40
41 }
42
```

```
1 def countingSort(arr, maxValue):
2     bucketLen = maxValue+1
3     bucket = [0]*bucketLen
4     sortedIndex = 0
5     arrLen = len(arr)
6     for i in range(arrLen):
7         if not bucket[arr[i]]:
8             bucket[arr[i]]=0
9             bucket[arr[i]]+=1
10    for j in range(bucketLen):
11        while bucket[j]>0:
12            arr[sortedIndex] = j
13            sortedIndex+=1
14            bucket[j]-=1
15    return arr
```

8 桶排序

桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。为了使桶排序更加高效，我们需要做到这两点：

1. 在额外空间充足的情况下，尽量增大桶的数量
2. 使用的映射函数能够将输入的 N 个数据均匀的分配到 K 个桶中

同时，对于桶中元素的排序，选择何种比较排序算法对于性能的影响至关重要。

1. 什么时候最快

当输入的数据可以均匀的分配到每一个桶中。

2. 什么时候最慢

当输入的数据被分配到了同一个桶中。


```
1 public class BucketSort implements IArraySort {
2
3     private static final InsertSort insertSort = new InsertSort();
4
5     @Override
6     public int[] sort(int[] sourceArray) throws Exception {
7         // 对 arr 进行拷贝, 不改变参数内容
8         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
9
10        return bucketSort(arr, 5);
11    }
12
13    private int[] bucketSort(int[] arr, int bucketSize) throws Exception {
14        if (arr.length == 0) {
15            return arr;
16        }
17
18        int minValue = arr[0];
19        int maxValue = arr[0];
20        for (int value : arr) {
21            if (value < minValue) {
22                minValue = value;
23            } else if (value > maxValue) {
24                maxValue = value;
25            }
26        }
27
28        int bucketCount = (int) Math.floor((maxValue - minValue) / bucketS
29        ize) + 1;
30        int[][] buckets = new int[bucketCount][0];
31
32        // 利用映射函数将数据分配到各个桶中
33        for (int i = 0; i < arr.length; i++) {
34            int index = (int) Math.floor((arr[i] - minValue) / bucketSize)
35            ;
36            buckets[index] = arrAppend(buckets[index], arr[i]);
37        }
38
39        int arrIndex = 0;
40        for (int[] bucket : buckets) {
41            if (bucket.length <= 0) {
42                continue;
43            }
44            // 对每个桶进行排序, 这里使用了插入排序
45            bucket = insertSort.sort(bucket);
46        }
47    }
48}
```

```

44         for (int value : bucket) {
45             arr[arrIndex++] = value;
46         }
47     }
48
49     return arr;
50 }
51
52 /**
53  * 自动扩容，并保存数据
54  *
55  * @param arr
56  * @param value
57  */
58 private int[] arrAppend(int[] arr, int value) {
59     arr = Arrays.copyOf(arr, arr.length + 1);
60     arr[arr.length - 1] = value;
61     return arr;
62 }
63
64 }

```

9 基数排序

基数排序是一种非比较型整数排序算法，其原理是将整数按**位数**切割成不同的数字，然后按每个位数分别比较。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。

1. 基数排序 vs 计数排序 vs 桶排序

基数排序有两种方法：

这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异案例看大家发的：

- 基数排序：根据键值的每位数字来分配桶；
- 计数排序：每个桶只存储单一键值；
- 桶排序：每个桶存储一定范围的数值；

2. LSD 基数排序动图演示

3

44

38

5

47

15

36

26

27

2

46

4

19

50

48

```
1  /**
2   * 基数排序
3   * 考虑负数的情况还可以参考: https://code.i-harness.com/zh-CN/q/e98fa9
4   */
5  public class RadixSort implements IArraySort {
6
7      @Override
8      public int[] sort(int[] sourceArray) throws Exception {
9          // 对 arr 进行拷贝, 不改变参数内容
10         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
11
12         int maxDigit = getMaxDigit(arr);
13         return radixSort(arr, maxDigit);
14     }
15
16     /**
17     * 获取最高位数
18     */
19     private int getMaxDigit(int[] arr) {
20         int maxValue = getMaxValue(arr);
21         return getNumLenght(maxValue);
22     }
23
24     private int getMaxValue(int[] arr) {
25         int maxValue = arr[0];
26         for (int value : arr) {
27             if (maxValue < value) {
28                 maxValue = value;
29             }
30         }
31         return maxValue;
32     }
33
34     protected int getNumLenght(long num) {
35         if (num == 0) {
36             return 1;
37         }
38         int lenght = 0;
39         for (long temp = num; temp != 0; temp /= 10) {
40             lenght++;
41         }
42         return lenght;
43     }
44
45     private int[] radixSort(int[] arr, int maxDigit) {
```

```

46         int mod = 10;
47         int dev = 1;
48
49         for (int i = 0; i < maxDigit; i++, dev *= 10, mod *= 10) {
50             // 考虑负数的情况，这里扩展一倍队列数，其中 [0-9]对应负数，[10-19]对应
正数 (bucket + 10)
51             int[][] counter = new int[mod * 2][0];
52
53             for (int j = 0; j < arr.length; j++) {
54                 int bucket = ((arr[j] % mod) / dev) + mod;
55                 counter[bucket] = arrayAppend(counter[bucket], arr[j]);
56             }
57
58             int pos = 0;
59             for (int[] bucket : counter) {
60                 for (int value : bucket) {
61                     arr[pos++] = value;
62                 }
63             }
64         }
65
66         return arr;
67     }
68
69     /**
70      * 自动扩容，并保存数据
71      *
72      * @param arr
73      * @param value
74      */
75     private int[] arrayAppend(int[] arr, int value) {
76         arr = Arrays.copyOf(arr, arr.length + 1);
77         arr[arr.length - 1] = value;
78         return arr;
79     }
80 }
81

```

```
1 def radix(arr):
2
3     digit = 0
4     max_digit = 1
5     max_value = max(arr)
6     #找出列表中最大的位数
7     while 10**max_digit < max_value:
8         max_digit = max_digit + 1
9
10    while digit < max_digit:
11        temp = [[] for i in range(10)]
12        for i in arr:
13            #求出每一个元素的个、十、百位的值
14            t = int((i/10**digit)%10)
15            temp[t].append(i)
16
17        coll = []
18        for bucket in temp:
19            for i in bucket:
20                coll.append(i)
21
22        arr = coll
23        digit = digit + 1
24
25    return arr
```