



本文内容是对桶排序的梳理和总结，本文内容包括：

## 桶排序(Bucket Sort)

算法步骤

举例说明

图解算法

代码实现

算法分析

## 排序算法十大经典方法

1. 冒泡排序：比较相邻元素并交换，每一轮将最大的元素移动到最后。
2. 选择排序：每一轮选出未排序部分中最小的元素，并将其放在已排序部分的末尾。
3. 插入排序：将未排序部分的第一个元素插入到已排序部分的合适位置
4. 希尔排序：改进的插入排序，将数据分组排序，最终合并排序
5. 归并排序：将序列拆分成子序列，分别排序后合并，递归完成
6. 快速排序：选定一个基准值，将小于基准值的元素放在左边，大于基准值的元素放在右边，递归排序
7. 堆排序：将序列构建成一个堆，然后一次将堆顶元素取出并调整堆
8. 计数排序：统计每个元素出现的次数，再按照元素大小输出
9. **桶排序：将数据分到一个或多个桶中，对每个桶进行排序，最后输出**
10. 基数排序：按照元素的位数从低到高进行排序，每次排序只考虑一位

# 1 桶排序(Bucket Sort)

---

- 桶排序 (Bucket sort) 是计数排序算法的升级版，将数据分到有限数量的桶子里，然后每个桶再分别排序

## 1.1 算法步骤

---

1. 第一步：确定桶的个数和每个桶装的数据范围

## 1. 确定每个桶存储的范围

- 首先找出所有数据中的最大值max和最小值min，根据max和min确定每个桶所装的数据的范围 range
- $\text{range} = (\text{max} - \text{min}) / \text{len} + 1$ ，len为数据的个数，需要保证至少有一个桶，故而需要加个1

## 2. 确定桶的数量

- $\text{bucketCount} = (\text{max} - \text{min}) / \text{range} + 1$ ，需要保证每个桶至少能装1个数，故而需要加个1

2. 第二步：将待排序的数据放入到各自的桶中

3. 第三步：对各个桶中的数据进行排序，可以使用其他的排序算法排序，例如快速排序

4. 第四步：依次将各个桶中的数据放入返回数组中，最后得到的数据即为最终有序数据

## 1.2 举例说明

**需求描述：**现在有一组待排序的数字（如下图所示），要求通过桶排序升序排序。

11 38 8 34 27 19 26 13

### 第一步：创建桶

- 先创建5个桶，桶的区间跨度=(最大值-最小值)/桶的数量，如图下所示



### 第二步：数据放入桶中

- 遍历原始序列，将序列放入桶中，如下图所示



### 第三步：桶内排序

- 每个桶内部的元素分别排序，如下图所示



此时，元素已是有序的了

## 1.3 图解算法

如果下图不动，点击[这里](#)查看在线的图解

0.6 0.85 0.9 0.5 0.35 0.2 0.1 0.85 0.8 0.5

@稀土掘金技术社区

## 1.4 代码实现

```
1 public static void main(String[] args) {
```

```
2     int[] arr = {11, 38, 8, 34, 19, 26, 13};
3     bucketSort(arr);
4     System.out.println("排序后: " + Arrays.toString(arr));
5 }
6
7 public static void bucketSort(int[] arr) {
8     // 找出数组中的最大最小值
9     int len = arr.length;
10    int min = arr[0], max = arr[0];
11    for (int i = 1; i < len; i++) {
12        min = Math.min(min, arr[i]);
13        max = Math.max(max, arr[i]);
14    }
15
16    // 确定每个桶存储的范围
17    int range = (max - min) / len + 1; // 保证最少存储1个
18
19    // 确定桶的数量
20    int bucketCount = (max - min) / range + 1; // 保证桶的个数
    至少为1
21
22    // 初始化桶
23    int [][] buckets = new int[bucketCount][0]; // 声明
    bucketCount个桶
24
25    // 将待排序的数据放入到各自的桶中
26    for (int i = 0; i < len; i++) {
27        int index = (arr[i] - min) / range;
28        buckets[index] = arrAppend(buckets[index], arr[i]);
29    }
30    // 对各个桶中的数据进行排序
31    for (int i = 0; i < bucketCount; i++) {
32        insertSort(buckets[i]);
33    }
34
35    // 依次将各个桶中的数据放入返回数组中
36    int index = 0;
37    for (int i = 0; i < bucketCount; i++) {
38        for (int j = 0; j < buckets[i].length; j++) {
39            arr[index++] = buckets[i][j];
40        }
41    }
42 }
43
44 // 插入排序算法
```

```

45 public static int[] insertSort(int arr[]) {
46     for (int i = 1; i < arr.length; i++) {
47         int tmp = arr[i];
48         int j = i;
49         while (j > 0 && tmp < arr[j - 1]) {
50             arr[j] = arr[j - 1];
51             j--;
52         }
53         if (j != i) {
54             arr[j] = tmp;
55         }
56     }
57     return arr;
58 }
59
60 //自动扩容，并保存数据
61 public static int[] arrAppend(int arr[], int value) {
62     arr = Arrays.copyOf(arr, arr.length + 1);
63     arr[arr.length - 1] = value;
64     return arr;
65 }

```

## 1.5 算法分析

---

- **稳定性**：稳定
- **时间复杂度**：最佳： $O(n+k)$ ，最差： $O(n^2)$ ，平均： $O(n+k)$
- **空间复杂度**： $O(n+k)$