常见排序算法

看云文档小组

目　录

（0）前言

> 来源：http://bubkoo.com/2014/01/17/sort-algorithm/archives/
> 作者：bubkoo

最近整理了一些常见的排序算法，资料基本上都来自网上，大部分参考了维基百科，分析了常见算法的原理，并举例分步说明，有的还给出了排序动画演示，但没有涉及算法复杂度等方面的概念，最后对每一种排序算法都给出了至少一种 JavaScript 的实现方法（因为我是做前端方面的，所以只给出了 JavaScript 代码）。

由于自己能力和经验有限，难免出现某些纰漏和错误，欢迎指正。

日本程序员 norahiko，写了一个排序算法的动画演示，非常有趣。另外，今天一同事告诉我有一个排序算法的舞蹈，请点击【程序员的艺术：排序算法舞蹈】。

（1）冒泡排序 (Bubble Sort)

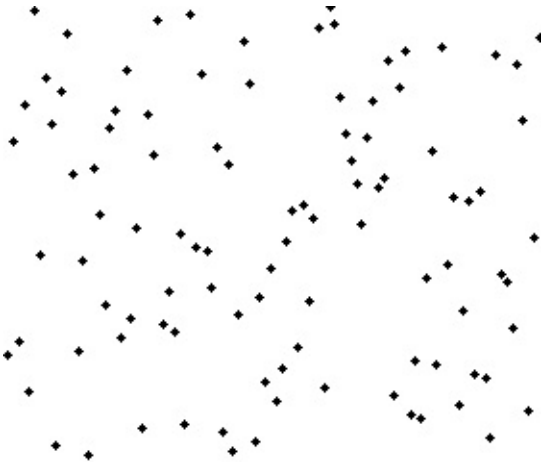
算法原理

> 冒泡排序（Bubble Sort，台湾译为：泡沫排序或气泡排序）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。

冒泡排序算法的流程如下：

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。在这一点，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. 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。

由于它的简洁，冒泡排序通常被用来对于程序设计入门的学生介绍算法的概念。



图片来自维基百科

实例分析

以数组 arr = [5, 1, 4, 2, 8] 为例说明，加粗的数字表示每次循环要比较的两个数字：

第一次外循环

(**5** **1** 4 2 8) → (**1** **5** 4 2 8)， 5 > 1 交换位置
(1 **5** **4** 2 8) → (1 **4** **5** 2 8)， 5 > 4 交换位置
(1 4 **5** **2** 8) → (1 4 **2** **5** 8)， 5 > 2 交换位置
(1 4 2 **5** **8**) → (1 4 2 **5** **8**)， 5 < 8 位置不变

第二次外循环（除开最后一个元素8，对剩余的序列）

(**1** **4** 2 5 8) → (**1** **4** 2 5 8)， 1 < 4 位置不变
(1 **4** **2** 5 8) → (1 **2** **4** 5 8)， 4 > 2 交换位置
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)， 4 < 5 位置不变

第三次外循环（除开已经排序好的最后两个元素，可以注意到上面的数组其实已经排序完成，但是程序本身并不知道，所以还要进行后续的循环，直到剩余的序列为 1）

(**1** **2** 4 5 8) → (**1** **2** 4 5 8)
(1 **2** **4** 5 8) → (1 **2** **4** 5 8)

第四次外循环（最后一次）
(**1** **2** 4 5 8) → (**1** **2** 4 5 8)

JavaScript 语言实现

```
function bubbleSort(array) {
    var length = array.length,
        i,
        j,
        temp;
    for (i = length - 1; 0 < i; i--) {
        for (j = 0; j < i; j++) {
            if (array[j] > array[j + 1]) {
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
    return array;
}
```

参考文章

- en.wikipedia.org
- 维基百科，自由的百科全书
- Bubble Sort
- 经典排序算法 - 冒泡排序Bubble sort
- 冒泡排序

（2）快速排序 (Quick Sort)

算法原理

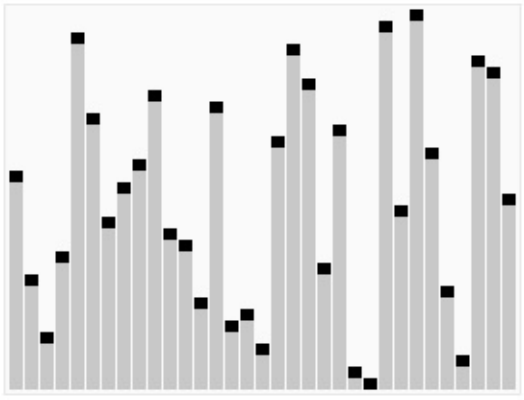
> 快速排序是图灵奖得主 C. R. A. Hoare 于 1960 年提出的一种划分交换排序。它采用了一种分治的策略，通常称其为分治法(Divide-and-ConquerMethod)。

C. R. A. Hoare

分治法的基本思想是：将原问题分解为若干个规模更小但结构与原问题相似的子问题。递归地解这些子问题，然后将这些子问题的解组合为原问题的解。

利用分治法可将快速排序的分为三步：

1. 在数据集之中，选择一个元素作为”基准”（pivot）。
2. 所有小于”基准”的元素，都移到”基准”的左边；所有大于”基准”的元素，都移到”基准”的右边。这个操作称为分区 (partition) 操作，分区操作结束后，基准元素所处的位置就是最终排序后它的位置。
3. 对”基准”左边和右边的两个子集，不断重复第一步和第二步，直到所有子集只剩下一个元素为止。



图片来自维基百科分区是快速排序的主要内容，用伪代码可以表示如下：

```
function partition(a, left, right, pivotIndex)
     pivotValue := a[pivotIndex]
     swap(a[pivotIndex], a[right]) // 把 pivot 移到結尾
     storeIndex := left
     for i from left to right-1
         if a[i] < pivotValue
             swap(a[storeIndex], a[i])
             storeIndex := storeIndex + 1
     swap(a[right], a[storeIndex]) // 把 pivot 移到它最後的地方
     return storeIndex // 返回 pivot 的最终位置
```

首先，把基准元素移到結尾（如果直接选择最后一个元素为基准元素，那就不用移动），然后从左到右（除了最后的基准元素），循环移动小于等于基准元素的元素到数组的开头，每次移动 storeIndex 自增 1，表示下一个小于基准元素将要移动到的位置。循环结束后 storeIndex 所代表的的位置就是基准元素的所有摆放的位置。所以最后将基准元素所在位置（这里是 right）与 storeIndex 所代表的的位置的元素交换位置。要注意的是，一个元素在到达它的最后位置前，可能会被交换很多次。

一旦我们有了这个分区算法，要写快速排列本身就很容易：

```
procedure quicksort(a, left, right)
    if right > left
        select a pivot value a[pivotIndex]
        pivotNewIndex := partition(a, left, right, pivotIndex)
        quicksort(a, left, pivotNewIndex-1)
        quicksort(a, pivotNewIndex+1, right)
```

实例分析

举例来说，现有数组 arr = [3,7,8,5,2,1,9,5,4]，分区可以分解成以下步骤：

1. 首先选定一个基准元素，这里我们元素 5 为基准元素（基准元素可以任意选择）：

```
          pivot
            ↓
3   7   8   5   2   1   9   5   4
```

1. 将基准元素与数组中最后一个元素交换位置，如果选择最后一个元素为基准元素可以省略该步：

```
                            pivot
                              ↓
3   7   8   4   2   1   9   5   5
```

1. 从左到右（除了最后的基准元素），循环移动小于基准元素 5 的所有元素到数组开头，留下大于等于基准元素的元素接在后面。在这个过程它也为基准元素找寻最后摆放的位置。循环流程如下：

循环 i == 0 时，storeIndex == 0，找到一个小于基准元素的元素 3，那么将其与 storeIndex 所在位置的元素交换位置，这里是 3 自身，交换后将 storeIndex 自增 1，storeIndex == 1：

```
                              pivot
                                ↓
  3   7   8   4   2   1   9   5   5
  ↑
storeIndex
```

循环 i == 3 时，storeIndex == 1，找到一个小于基准元素的元素 4：

```
     ┌─────────┐                pivot
     ↓         ↓                  ↓
 3   7   8   4   2   1   9   5   5
     ↑         ↑
storeIndex     i
```

交换位置后，storeIndex 自增 1，storeIndex == 2：

```
                            pivot
                              ↓
3   4   8   7   2   1   9   5   5
        ↑
```

```
    storeIndex
```

循环 i == 4 时，storeIndex == 2，找到一个小于基准元素的元素 2：

```
          ┌─────────┐              pivot
          ↓         ↓                ↓
3   4   8   7   2   1   9   5   5
          ↑         ↑
    storeIndex    i
```

交换位置后，storeIndex 自增 1，storeIndex == 3：

```
                              pivot
                                ↓
3   4   2   7   8   1   9   5   5
            ↑
        storeIndex
```

循环 i == 5 时，storeIndex == 3，找到一个小于基准元素的元素 1：

```
              ┌─────────┐         pivot
              ↓         ↓           ↓
3   4   2   7   8   1   9   5   5
              ↑         ↑
        storeIndex    i
```

交换后位置后，storeIndex 自增 1，storeIndex == 4：

```
                              pivot
                                ↓
3   4   2   1   8   7   9   5   5
                ↑
            storeIndex
```

循环 i == 7 时，storeIndex == 4，找到一个小于等于基准元素的元素 5：

```
                  ┌─────────────┐ pivot
                  ↓             ↓   ↓
3   4   2   1   8   7   9   5   5
                  ↑             ↑
            storeIndex        i
```

交换后位置后，storeIndex 自增 1，storeIndex == 5：

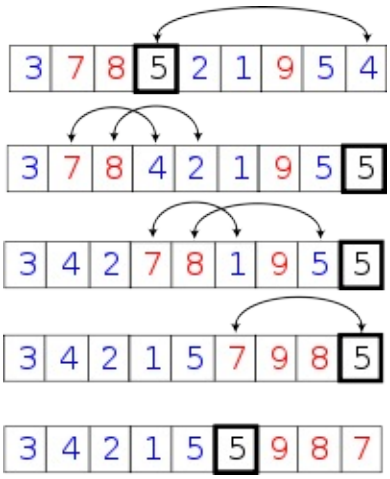
```
                              pivot
                                ↓
3   4   2   1   5   7   9   8   5
                    ↑
                storeIndex
```

1. 循环结束后交换基准元素和 storeIndex 位置的元素的位置：

```
                   pivot
                     ↓
3   4   2   1   5   5   9   8   7
                    ↑
                storeIndex
```

那么 storeIndex 的值就是基准元素的最终位置，这样整个分区过程就完成了。

引用维基百科上的一张图片：



图片来自维基百科

JavaScript 语言实现

查看了很多关于 JavaScript 实现快速排序方法的文章后，发现绝大多数实现方法如下：

```
function quickSort(arr) {
    if (arr.length <= 1) {
        return arr;
    }
    var pivotIndex = Math.floor(arr.length / 2);
    var pivot = arr.splice(pivotIndex, 1)[0];
    var left = [];
```

```
    var right = [];
    for (var i = 0; i < arr.length; i++) {
        if (arr[i] < pivot) {
            left.push(arr[i]);
        } else {
            right.push(arr[i]);
        }
    }
    return quickSort(left).concat([pivot], quickSort(right));
}
```

> 上面简单版本的缺点是，它需要Ω(n)的额外存储空间，也就跟归并排序一样不好。额外需要的存储器空间配置，在实际上的实现，也会极度影响速度和高速缓存的性能。
>
> 摘自维基百科

按照维基百科中的原地(in-place)分区版本，实现快速排序方法如下：

```
function quickSort(array) {
        // 交换元素位置
        function swap(array, i, k) {
                var temp = array[i];
                array[i] = array[k];
                array[k] = temp;
        }
        // 数组分区，左小右大
        function partition(array, left, right) {
                var storeIndex = left;
                var pivot = array[right]; // 直接选最右边的元素为基准元素
                for (var i = left; i < right; i++) {
                        if (array[i] < pivot) {
                                swap(array, storeIndex, i);
                                storeIndex++; // 交换位置后，storeIndex 自增
 1，代表下一个可能要交换的位置
                        }
                }
                swap(array, right, storeIndex); // 将基准元素放置到最后的正确
位置上
                return storeIndex;
        }

        function sort(array, left, right) {
                if (left > right) {
                        return;
                }
                var storeIndex = partition(array, left, right);
                sort(array, left, storeIndex - 1);
                sort(array, storeIndex + 1, right);
        }

        sort(array, 0, array.length - 1);
```

```
        return array;
}
```

另外一个版本，思路和上面的一样，代码逻辑没有上面的清晰

```
function quickSort(arr) {
    return sort(arr, 0, arr.length - 1);

    function swap(arr, i, k) {
        var temp = arr[i];
        arr[i] = arr[k];
        arr[k] = temp;
    }

    function sort(arr, start, end) {
        sort(arr, 0, arr.length - 1);
        return arr;

        function swap(arr, i, k) {
            var temp = arr[i];
            arr[i] = arr[k];
            arr[k] = temp;
        }

        function sort(arr, start, end) {
            if (start >= end) return;

            var pivot = arr[start],
                i = start + 1,
                k = end;

            while (true) {
                while (arr[i] < pivot) {
                    i++;
                }
                while (arr[k] > pivot) {
                    k--;
                }

                if (i >= k) {
                    break;
                }
                swap(arr, i, k);
            }

            swap(arr, start, k);
            sort(arr, start, Math.max(0, k - 1));
            sort(arr, Math.min(end, k + 1), end);
        }
    }
}
```

参考文章

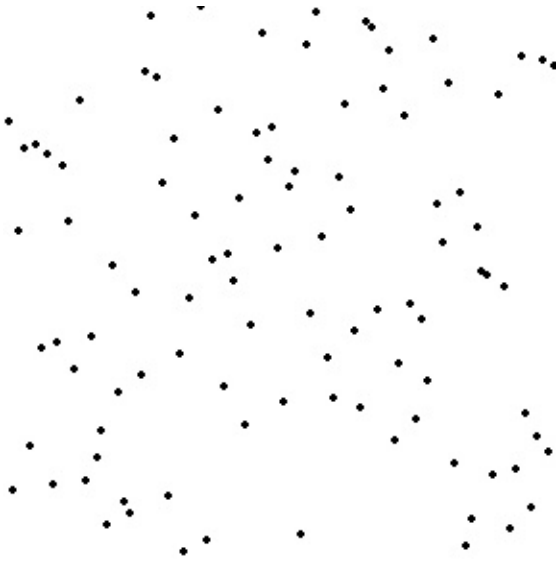
- wiki Quicksort
- 维基百科 - 快速排序
- 快速排序（Quicksort）的Javascript实现
- Quicksort in JavaScript
- 经典排序算法 - 快速排序Quick sort
- 快速排序(QuickSort)
- ソートアルゴリズムを映像化してみた
- Stable quicksort in Javascript
- Friday Algorithms: Quicksort – Difference Between PHP and JavaScript

（3）选择排序 (Selection Sort)

算法原理

> 选择排序（Selection Sort）是一种简单直观的排序算法。它的工作原理如下，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，然后，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。以此类推，直到所有元素均排序完毕。

选择排序的主要优点与数据移动有关。如果某个元素位于正确的最终位置上，则它不会被移动。选择排序每次交换一对元素，它们当中至少有一个将被移到其最终位置上，因此对n个元素的序列进行排序总共进行至多n-1次交换。在所有的完全依靠交换去移动元素的排序方法中，选择排序属于非常好的一种。



图片来源于维基百科

实例分析

以数组 arr = [8, 5, 2, 6, 9, 3, 1, 4, 0, 7] 为例，先直观看一下每一步的变化，后面再介绍细节

第一次从数组 [8, 5, 2, 6, 9, 3, 1, 4, 0, 7] 中找到最小的数 0，放到数组的最前面（与第一个元素进行交换）：

```
                                min
                                 ↓
8   5   2   6   9   3   1   4   0   7
↑                                ↑
└────────────────────────────────┘
```

交换后：

```
0   5   2   6   9   3   1   4   8   7
```

在剩余的序列中 [5, 2, 6, 9, 3, 1, 4, 8, 7] 中找到最小的数 1，与该序列的第一个个元素进行位置交换：

```
                        min
                         ↓
0   5   2   6   9   3   1   4   8   7
    ↑                   ↑
    └───────────────────┘
```

交换后：

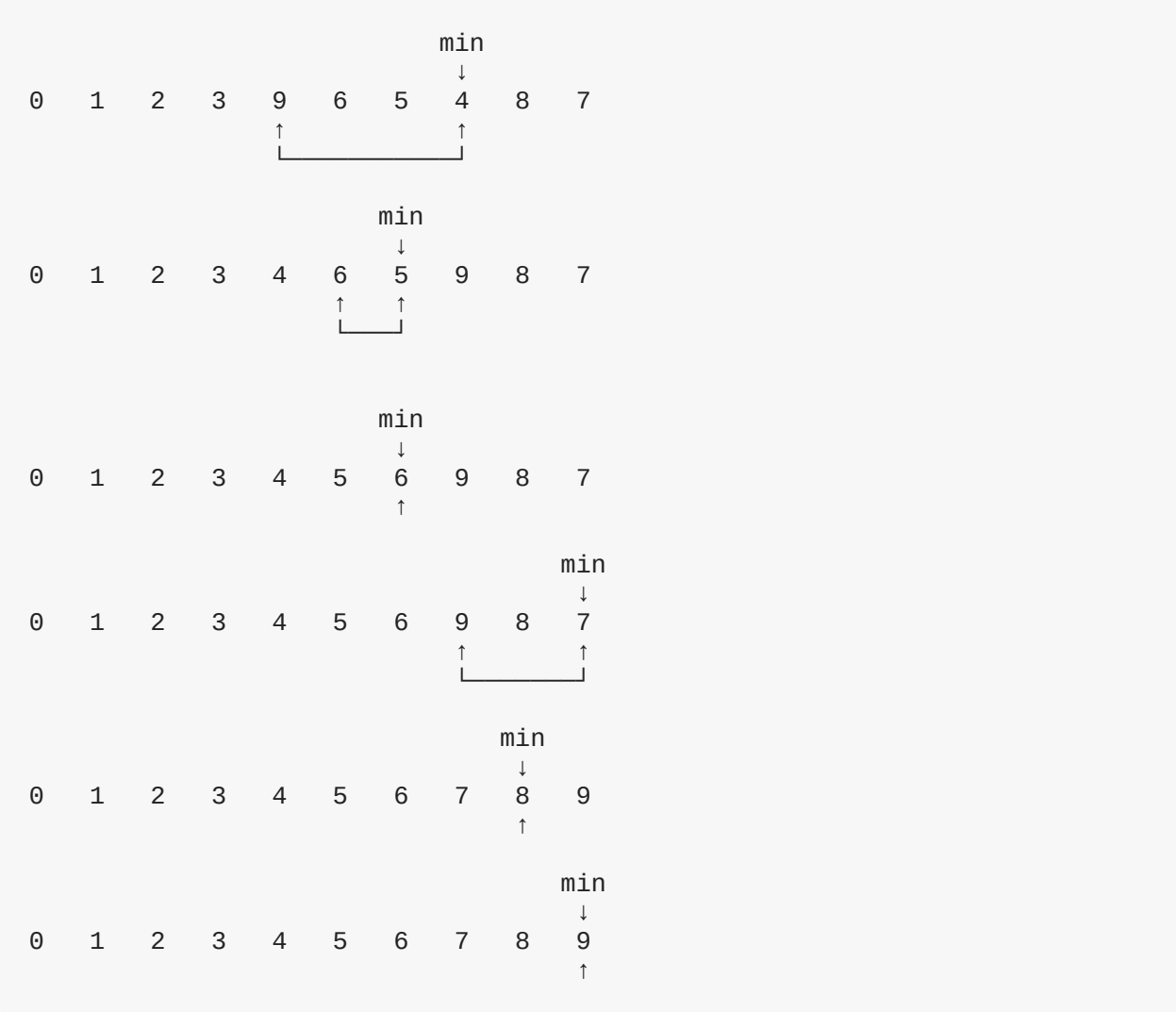
```
0   1   2   6   9   3   5   4   8   7
```

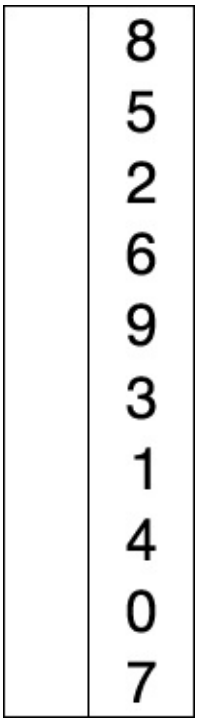
在剩余的序列中 [2, 6, 9, 3, 5, 4, 8, 7] 中找到最小的数 2，与该序列的第一个个元素进行位置交换（实际上不需要交换）：

```
        min
         ↓
0   1   2   6   9   3   5   4   8   7
        ↑
```

重复上述过程，直到最后一个元素就完成了排序。

```
                    min
                     ↓
0   1   2   6   9   3   5   4   8   7
            ↑       ↑
            └───────┘
```


min
0 1 2 3 9 6 5 4 8 7
min
0 1 2 3 4 6 5 9 8 7
min
0 1 2 3 4 5 6 9 8 7
min
0 1 2 3 4 5 6 9 8 7
min
0 1 2 3 4 5 6 7 8 9
min
0 1 2 3 4 5 6 7 8 9



图片来源于维基百科

JavaScript 语言实现

```
function selectionSort(array) {
  var length = array.length,
      i,
      j,
      minIndex,
      minValue,
      temp;
  for (i = 0; i < length - 1; i++) {
    minIndex = i;
    minValue = array[minIndex];
    for (j = i + 1; j < length; j++) {
      if (array[j] < minValue) {
        minIndex = j;
        minValue = array[minIndex];
      }
    }

    // 交换位置
    temp = array[i];
    array[i] = minValue;
    array[minIndex] = temp;
  }
  return array
}
```

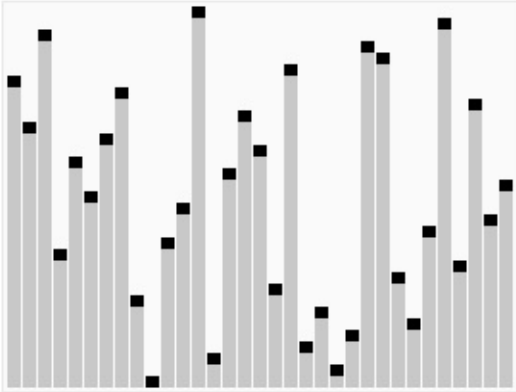
参考文章

- en.wikipedia.org
- wikibooks
- 维基百科
- Selection sort in JavaScript
- 直接选择排序(Straight Selection Sort)
- 经典排序算法 - 选择排序 Selection Sort
- 选择排序算法

（4）堆排序 (Heap Sort)

算法原理

先上一张堆排序动画演示图片：



图片来自维基百科

1. 不得不说说二叉树

要了解堆首先得了解一下二叉树，在计算机科学中，二叉树是每个节点最多有两个子树的树结构。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）。二叉树常被用于实现二叉查找树和二叉堆。

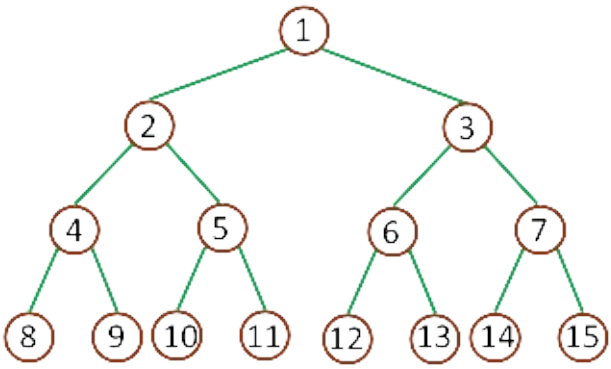
二叉树的每个结点至多只有二棵子树（不存在度大于 2 的结点），二叉树的子树有左右之分，次序不能颠倒。二叉树的第 i 层至多有 2i - 1 个结点；深度为 k 的二叉树至多有 2k - 1 个结点；对任何一棵二叉树 T，如果其终端结点数为 n0，度为 2 的结点数为 n2，则n0 = n2 + 1。

树和二叉树的三个主要差别：

- 树的结点个数至少为 1，而二叉树的结点个数可以为 0
- 树中结点的最大度数没有限制，而二叉树结点的最大度数为 2
- 树的结点无左、右之分，而二叉树的结点有左、右之分

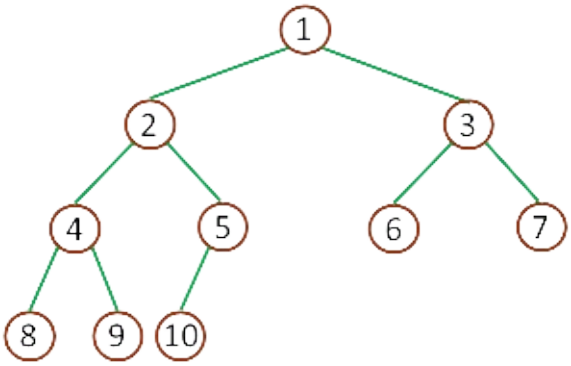
二叉树又分为完全二叉树（complete binary tree）和满二叉树（full binary tree）

满二叉树：一棵深度为 k，且有 2k - 1 个节点称之为满二叉树



深度为 3 的满二叉树 full binary tree

完全二叉树：深度为 k，有 n 个节点的二叉树，当且仅当其每一个节点都与深度为 k 的满二叉树中序号为 1 至 n 的节点对应时，称之为完全二叉树

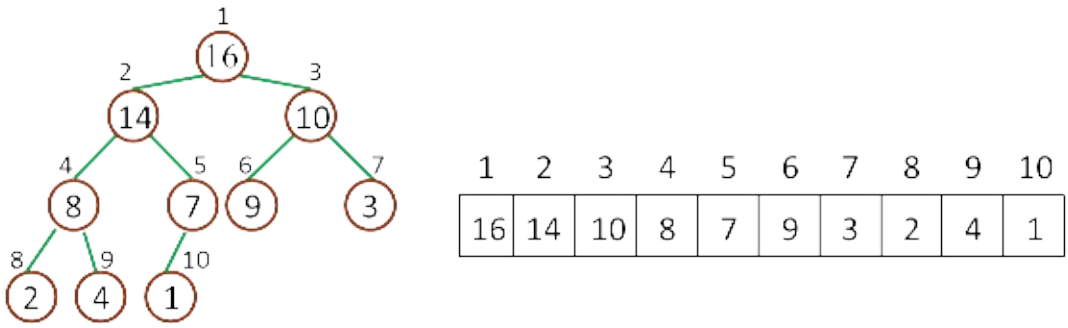


深度为 3 的完全二叉树 complete binary tree

2. 什么是堆？

堆（二叉堆）可以视为一棵完全的二叉树，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。

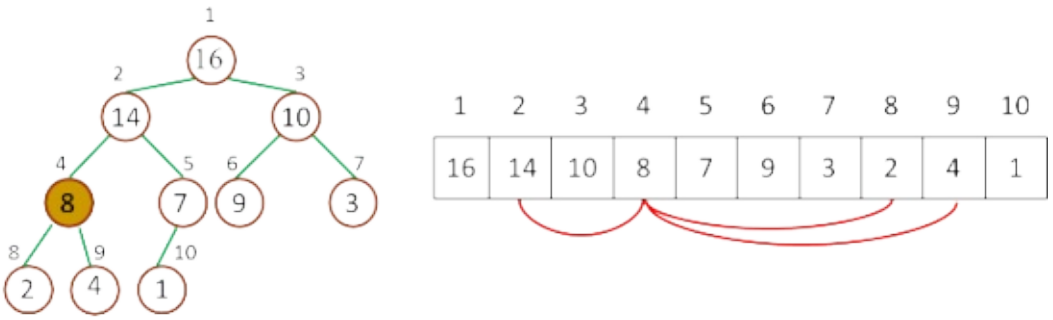
如下图，是一个堆和数组的相互关系



堆和数组的相互关系

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：

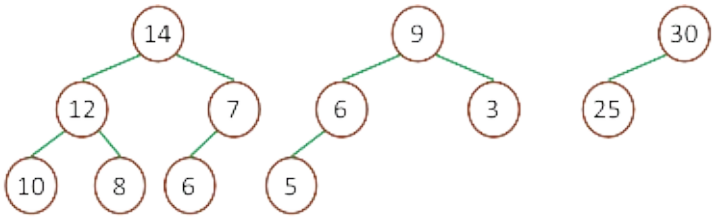
- Parent(i) = floor(i/2)，i 的父节点下标
- Left(i) = 2i，i 的左子节点下标
- Right(i) = 2i + 1，i 的右子节点下标



二叉堆一般分为两种：最大堆和最小堆。

最大堆：

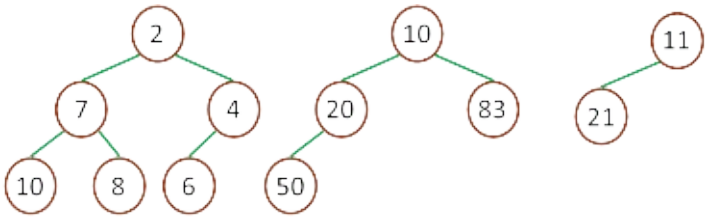
- 最大堆中的最大元素值出现在根结点（堆顶）
- 堆中每个父节点的元素值都大于等于其孩子结点（如果存在）



最大堆

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）



最小堆

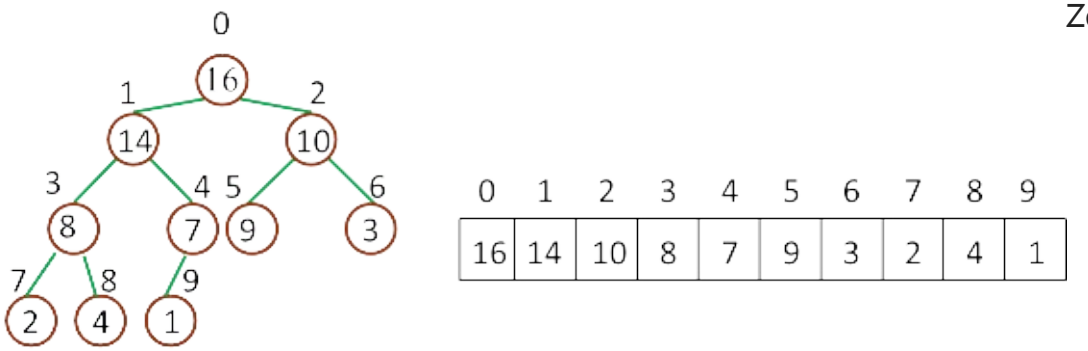
3. 堆排序原理

> 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

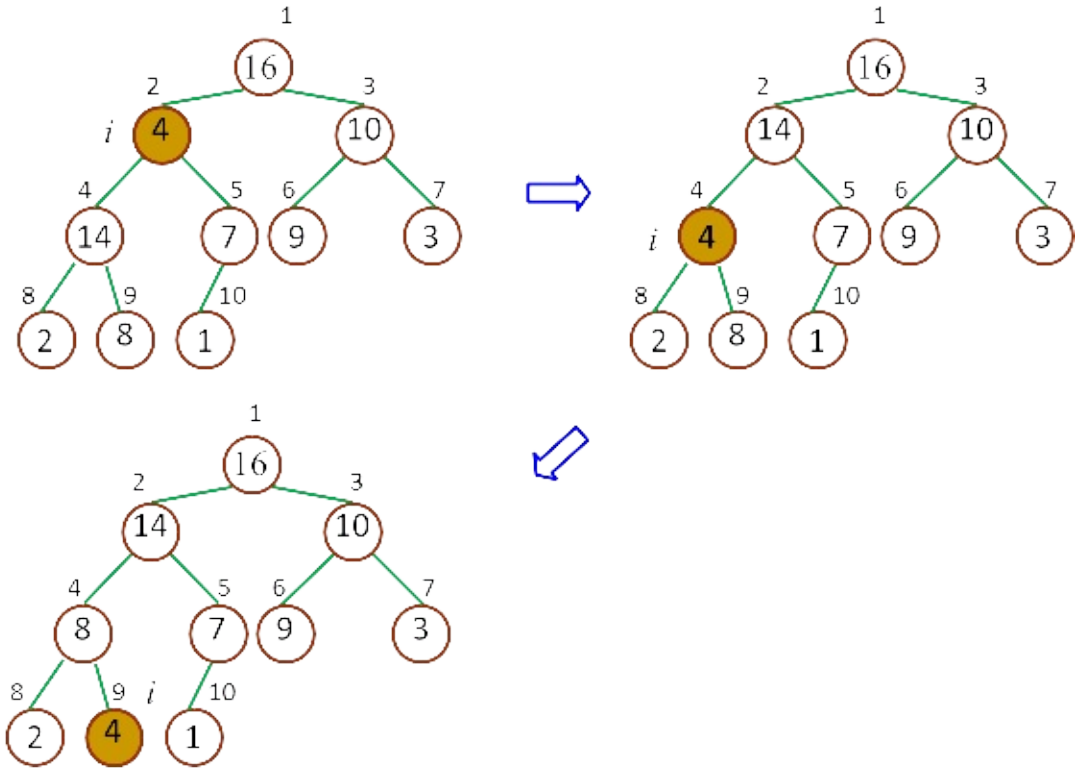
Zero-Based



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

最大堆调整（MAX-HEAPIFY）的作用是保持最大堆的性质，是创建最大堆的核心子程序，作用过程如图所示：



Max-Heapify

由于一次调整后，堆仍然违反堆性质，所以需要递归的测试，使得整个堆都满足堆性质，用 JavaScript 可以表示如下：

```
/**
 * 从 index 开始检查并保持最大堆性质
 *
 * @array
 *
 * @index 检查的起始下标
 *
 * @heapSize 堆大小
 *
 **/
function maxHeapify(array, index, heapSize) {
 var iMax = index,
 iLeft = 2 * index + 1,
 iRight = 2 * (index + 1);

 if (iLeft
 iMax = iLeft;
 }
```

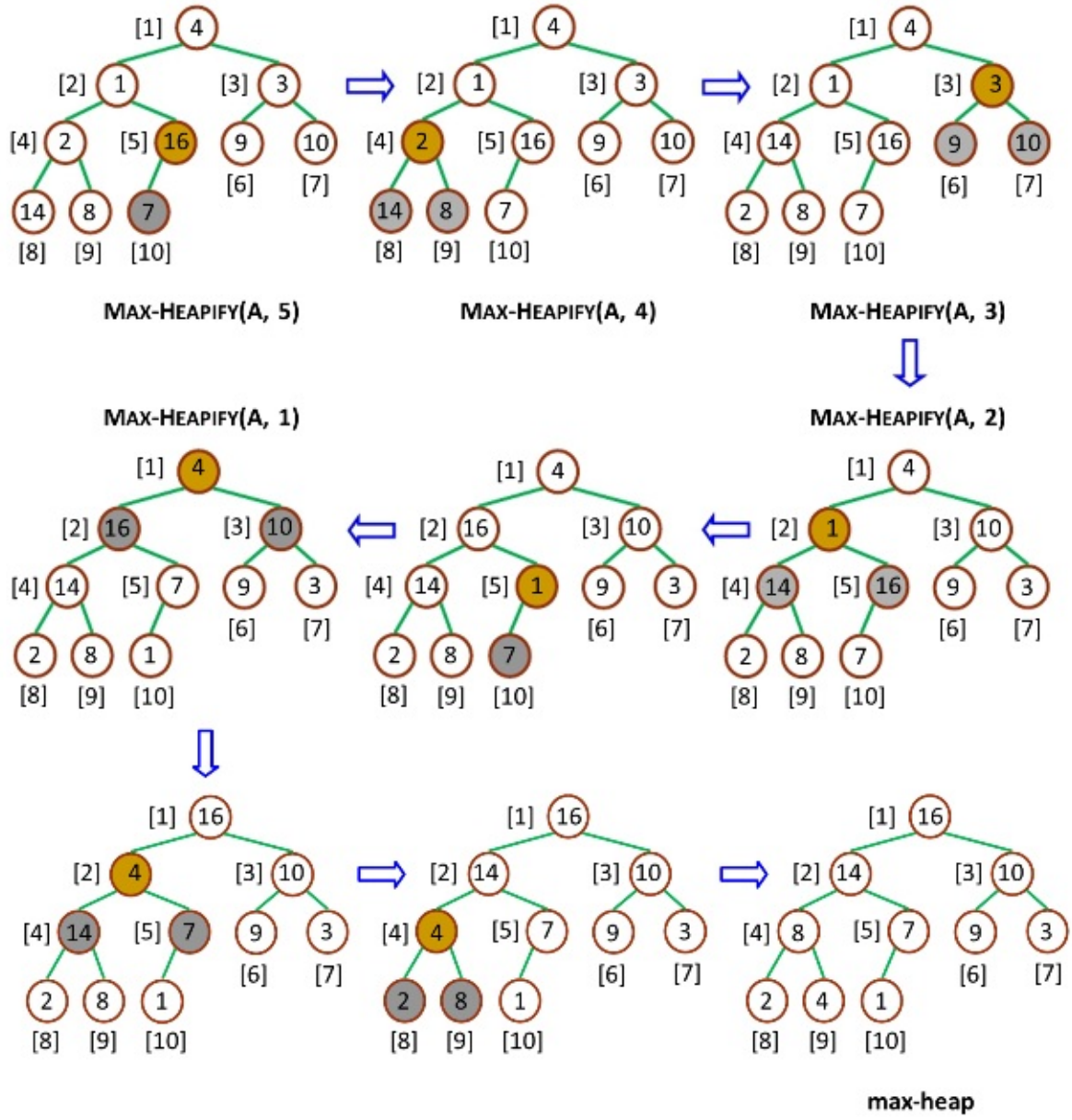
```
 if (iRight
 iMax = iRight;
 }

 if (iMax != index) {
 swap(array, iMax, index);
 maxHeapify(array, iMax, heapSize); // 递归调整
 }
}

function swap(array, i, j) {
 var temp = array[i];
 array[i] = array[j];
 array[j] = temp;
}
```

通常来说，递归主要用在分治法中，而这里并不需要分治。而且递归调用需要压栈/清栈，和迭代相比，性能上有略微的劣势。当然，按照20/80法则，这是可以忽略的。但是如果你觉得用递归会让自己心里过不去的话，也可以用迭代，比如下面这样：

```
/**
 * 从 index 开始检查并保持最大堆性质
 *
 * @array
 *
 * @index 检查的起始下标
 *
 * @heapSize 堆大小
 *
 **/
function maxHeapify(array, index, heapSize) {
 var iMax, iLeft, iRight;
 while (true) {
 iMax = index;
 iLeft = 2 * index + 1;
 iRight = 2 * (index + 1);
 if (iLeft
 iMax = iLeft;
 }

 if (iRight
 iMax = iRight;
 }

 if (iMax != index) {
 swap(array, iMax, index);
 index = iMax;
 } else {
 break;
 }
 }
```

```
}

function swap(array, i, j) {
 var temp = array[i];
 array[i] = array[j];
 array[j] = temp;
}
```

创建最大堆（Build-Max-Heap）的作用是将一个数组改造成一个最大堆，接受数组和堆大小两个参数，Build-Max-Heap 将自下而上的调用 Max-Heapify 来改造数组，建立最大堆。因为 Max-Heapify 能够保证下标 i 的结点之后结点都满足最大堆的性质，所以自下而上的调用 Max-Heapify 能够在改造过程中保持这一性质。如果最大堆的数量元素是 n，那么 Build-Max-Heap 从 Parent(n) 开始，往上依次调用 Max-Heapify。流程如下：

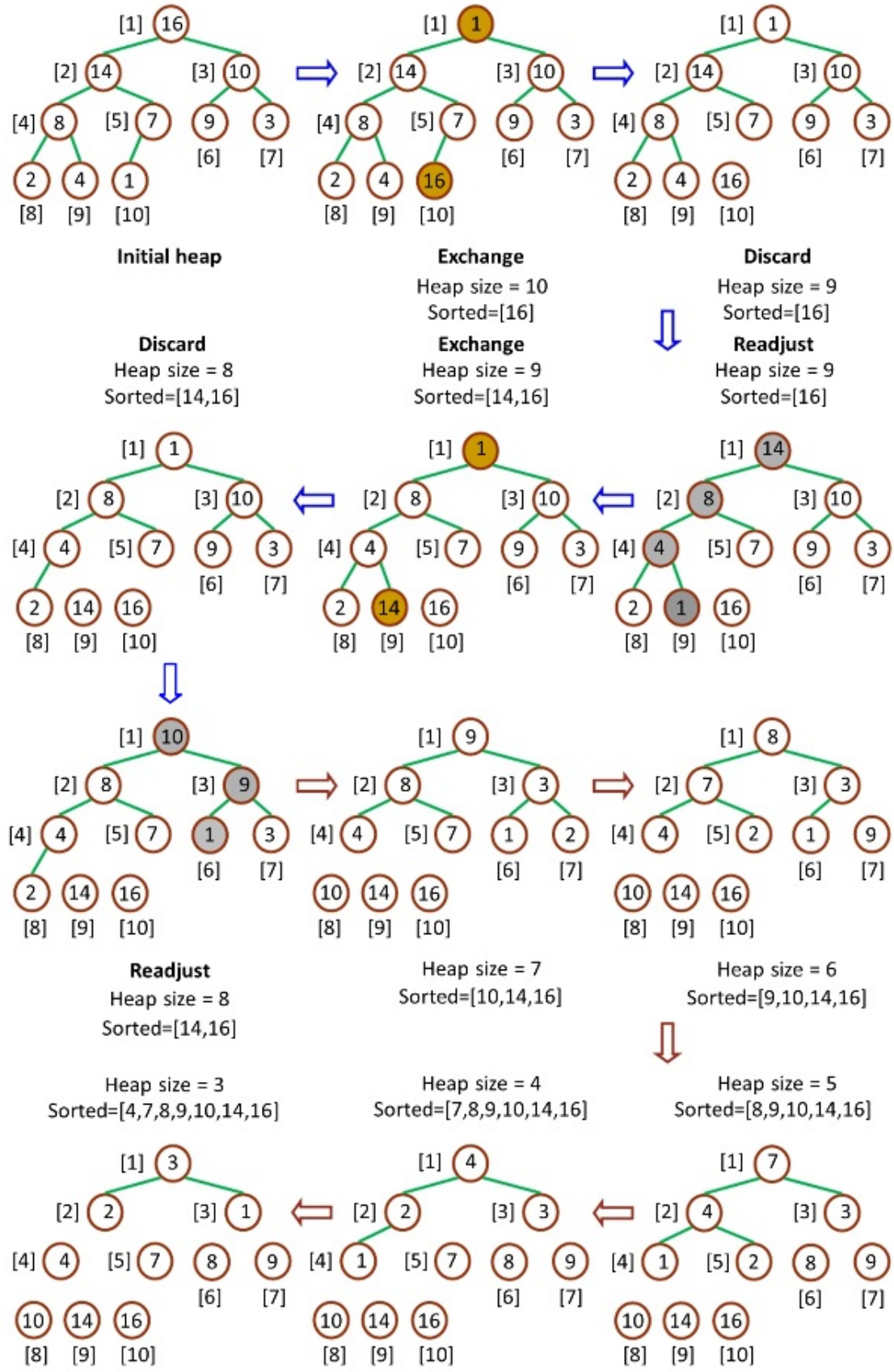


Build-Max-Heap

用 JavaScript 描述如下：

```
function buildMaxHeap(array, heapSize) {
 var i,
 iParent = Math.floor((heapSize - 1) / 2);

 for (i = iParent; i >= 0; i--) {
 maxHeapify(array, i, heapSize);
 }
}
```

堆排序（Heap-Sort）是堆排序的接口算法，Heap-Sort先调用Build-Max-Heap将数组改造为最大堆，然后将堆顶和堆底元素交换，之后将底部上升，最后重新调用Max-Heapify保持最大堆性质。由于堆顶元素必然是堆中最大的元素，所以一次操作之后，堆中存在的最大元素被分离出堆，重复n-1次之后，数组排列完毕。整个流程如下：


Initial heap
Exchange
Heap size = 10
Sorted=[16]
Discard
Heap size = 9
Sorted=[16]
Readjust
Heap size = 9
Sorted=[16]
Exchange
Heap size = 9
Sorted=[14,16]
Discard
Heap size = 8
Sorted=[14,16]
Readjust
Heap size = 8
Sorted=[14,16]
Heap size = 7
Sorted=[10,14,16]
Heap size = 6
Sorted=[9,10,14,16]
Heap size = 5
Sorted=[8,9,10,14,16]
Heap size = 4
Sorted=[7,8,9,10,14,16]
Heap size = 3
Sorted=[4,7,8,9,10,14,16]

Heap-Sort

用 JavaScript 描述如下：

```
function heapSort(array, heapSize) {

 buildMaxHeap(array, heapSize);

 for (int i = heapSize - 1; i > 0; i--) {
 swap(array, 0, i);
 maxHeapify(array, 0, i);
 }
}
```

JavaScript 语言实现

最后，把上面的整理为完整的 javascript 代码如下：

```
function heapSort(array) {

  function swap(array, i, j) {
    var temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }

  function maxHeapify(array, index, heapSize) {
    var iMax,
      iLeft,
      iRight;
    while (true) {
      iMax = index;
      iLeft = 2 * index + 1;
      iRight = 2 * (index + 1);

      if (iLeft < heapSize && array[index] < array[iLeft]) {
        iMax = iLeft;
      }

      if (iRight < heapSize && array[iMax] < array[iRight]) {
        iMax = iRight;
      }

      if (iMax != index) {
        swap(array, iMax, index);
        index = iMax;
      } else {
        break;
      }
    }
```

```
  }

  function buildMaxHeap(array) {
    var i,
      iParent = Math.floor(array.length / 2) - 1;

    for (i = iParent; i >= 0; i--) {
      maxHeapify(array, i, array.length);
    }
  }

  function sort(array) {
    buildMaxHeap(array);

    for (var i = array.length - 1; i > 0; i--) {
      swap(array, 0, i);
      maxHeapify(array, 0, i);
    }
    return array;
  }

  return sort(array);
}
```

参考文章

- Wikipedia
- 维基百科，堆排序
- 维基百科，二叉树
- Algorithms Chapter 6 Heapsort
- Heap Sort
- 堆与堆排序
- 堆排序
- 堆排序(Heap Sort)算法学习
- Sorting Algorithm Animations

（5）插入排序 (Insertion Sort)

算法原理

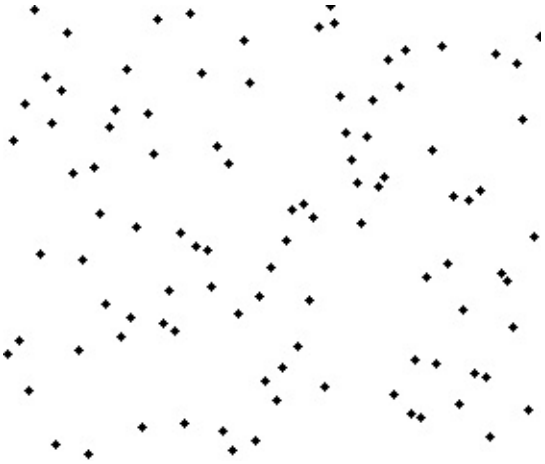
设有一组关键字{ K1， K2，…， Kn }；排序开始就认为 K1 是一个有序序列；让 K2 插入上述表长为 1 的有序序列，使之成为一个表长为 2 的有序序列；然后让 K3 插入上述表长为 2 的有序序列，使之成为一个表长为 3 的有序序列；依次类推，最后让 Kn 插入上述表长为 n-1 的有序序列，得一个表长为 n 的有序序列。

具体算法描述如下：

1. 从第一个元素开始，该元素可以认为已经被排序
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

如果比较操作的代价比交换操作大的话，可以采用二分查找法来减少比较操作的数目。该算法可以认为是插入排序的一个变种，称为二分查找排序。

> 二分查找法，是一种在有序数组中查找某一特定元素的搜索算法。搜素过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组为空，则代表找不到。这种搜索算法每一次比较都使搜索范围缩小一半。



图片来自维基百科

实例分析

现有一组数组 arr = [5, 6, 3, 1, 8, 7, 2, 4]，共有八个记录，排序过程如下：

```
[5]   6   3   1   8   7   2   4
  ↑
  └──┘

[5, 6]   3   1   8   7   2   4
↑        │
└────────┘

[3, 5, 6]  1   8   7   2   4
↑          │
└──────────┘

[1, 3, 5, 6]  8   7   2   4
           ↑  │
           └──┘

[1, 3, 5, 6, 8]  7   2   4
              ↑  │
              └──┘

[1, 3, 5, 6, 7, 8]  2   4
   ↑                │
   └────────────────┘

[1, 2, 3, 5, 6, 7, 8]  4
         ↑             │
         └─────────────┘

[1, 2, 3, 4, 5, 6, 7, 8]
```

其中有一点比较有意思的是，在每次比较操作发现新元素小于等于已排序的元素时，可以将

已排序的元素移到下一位置，然后再将新元素插入该位置，接着再与前面的已排序的元素进行比较，这样做交换操作代价比较大。还有一个做法是，将新元素取出，从左到右依次与已排序的元素比较，如果已排序的元素大于新元素，那么将该元素移动到下一个位置，接着再与前面的已排序的元素比较，直到找到已排序的元素小于等于新元素的位置，这时再将新元素插入进去，就像下面这样：

图片来自维基百科

JavaScript 语言实现

直接插入排序 JavaScript 实现代码：

```
function insertionSort(array) {

  function swap(array, i, j) {
    var temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }

  var length = array.length,
      i,
      j;
  for (i = 1; i < length; i++) {
    for (j = i; j > 0; j--) {
      if (array[j - 1] > array[j]) {
        swap(array, j - 1, j);
      } else {
        break;
      }
    }
  }
  return array;

}
```

下面这种方式可以减少交换次数：

```
function insertionSort(array) {

  var length = array.length,
    i,
    j,
    temp;
  for (i = 1; i < length; i++) {
    temp = array[i];
    for (j = i; j >= 0; j--) {
      if (array[j - 1] > temp) {
        array[j] = array[j - 1];
      } else {
        array[j] = temp;
        break;
      }
    }
  }
  return array;

}
```

利用二分查找法实现的插入排序，二分查找排序：

```
function insertionSort2(array) {

  function binarySearch(array, start, end, temp) {
    var middle;
    while (start <= end) {
      middle = Math.floor((start + end) / 2);
      if (array[middle] < temp) {
        if (temp <= array[middle + 1]) {
          return middle + 1;
        } else {
          start = middle + 1;
        }
      } else {
        if (end === 0) {
          return 0;
        } else {
          end = middle;
        }
      }
    }
  }

  function binarySort(array) {
    var length = array.length,
        i,
        j,
        k,
        temp;
    for (i = 1; i < length; i++) {
      temp = array[i];
```

```
        if (array[i - 1] <= temp) {
          k = i;
        } else {
          k = binarySearch(array, 0, i - 1, temp);
          for (j = i; j > k; j--) {
            array[j] = array[j - 1];
          }
        }
        array[k] = temp;
      }
      return array;
    }

    return binarySort(array);

  }
```

参考文章

- Wikipedia
- 维基百科 - 插入排序
- 维基百科 - 二分查找法
- 排序算法—折半插入排序（二分查找排序）
- 直接插入排序
- 直接插入排序基本思想

（6）希尔排序 (Shell Sort)

算法原理

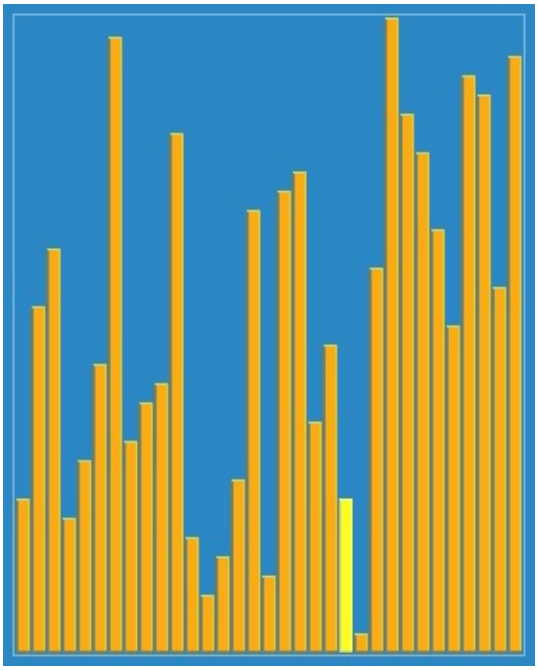
> 希尔排序算法是按其设计者希尔（Donald Shell）的名字命名，该算法由1959年公布，是插入排序的一种更高效的改进版本。它的作法不是每次一个元素挨一个元素的比较。而是初期选用大跨步（增量较大）间隔比较，使记录跳跃式接近它的排序位置；然后增量缩小；最后增量为 1 ，这样记录移动次数大大减少，提高了排序效率。希尔排序对增量序列的选择没有严格规定。

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

算法思路：

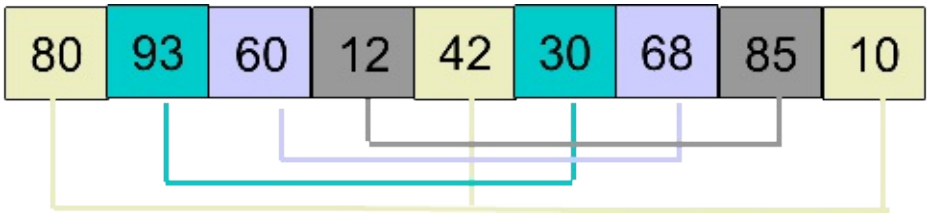
1. 先取一个正整数 d1(d1 1 个组，所有距离为 d1 的倍数的记录看成一组，然后在各组内进行插入排序
2. 然后取 d2(d2 1)
3. 重复上述分组和排序操作；直到取 di = 1(i >= 1) 位置，即所有记录成为一个组，最后对这个组进行插入排序。一般选 d1 约为 n/2，d2 为 d1 /2， d3 为 d2/2 ，... ， di = 1。



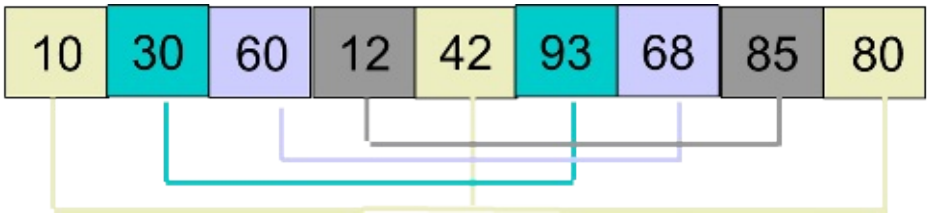
图片来自维基百科

实例分析

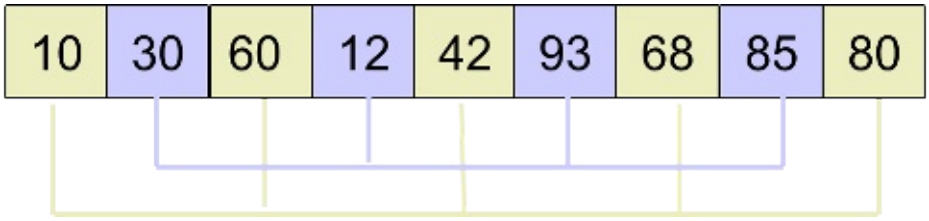
假设有数组 array = [80, 93, 60, 12, 42, 30, 68, 85, 10]，首先取 d1 = 4，将数组分为 4 组，如下图中相同颜色代表一组：



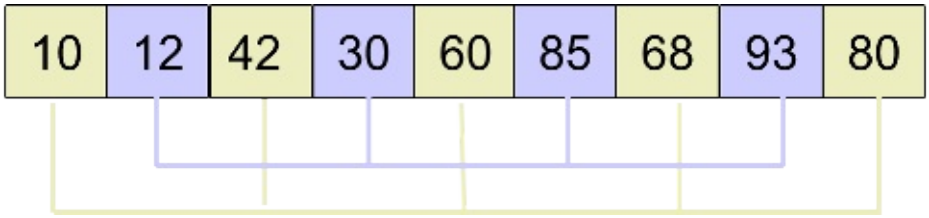
然后分别对 4 个小组进行插入排序，排序后的结果为：



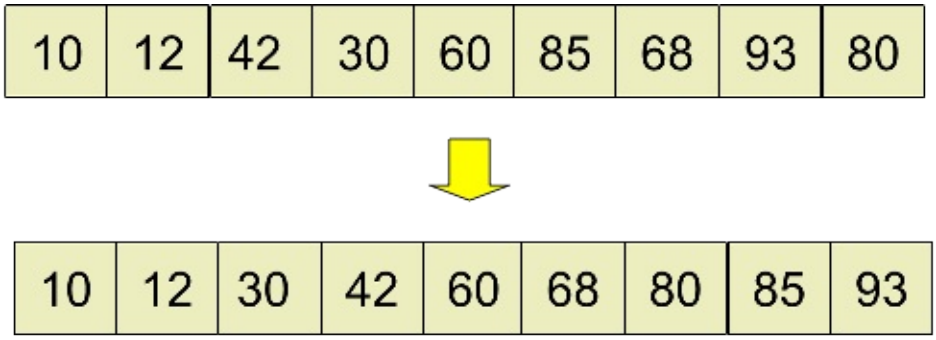
然后，取 d2 = 2，将原数组分为 2 小组，如下图：



然后分别对 2 个小组进行插入排序，排序后的结果为：



最后，取 d3 = 1，进行插入排序后得到最终结果：



JavaScript 语言实现

按照惯例，下面给出了 JavaScript 的算法实现：

```
function shellSort(array) {

    function swap(array, i, k) {
        var temp = array[i];
        array[i] = array[k];
        array[k] = temp;
    }

    var length = array.length,
        gap = Math.floor(length / 2);

    while (gap > 0) {
        for (var i = gap; i < length; i++) {
            for (var j = i; 0 < j; j -= gap) {
                if (array[j - gap] > array[j]) {
                    swap(array, j - gap, j);
                } else {
                    break;
```

```
                }
            }
        }

        gap = Math.floor(gap / 2);
    }

    return array;
}
```

参考文章

- 维基百科，自由的百科全书
- 希尔排序基本思想
- [演算法] 希爾排序法(Shell Sort)
- 算法系列15天速成——第三天 七大经典排序【下】
- Algorithm Implementation/Sorting/Shell sort

（7）归并排序 (Merge Sort)

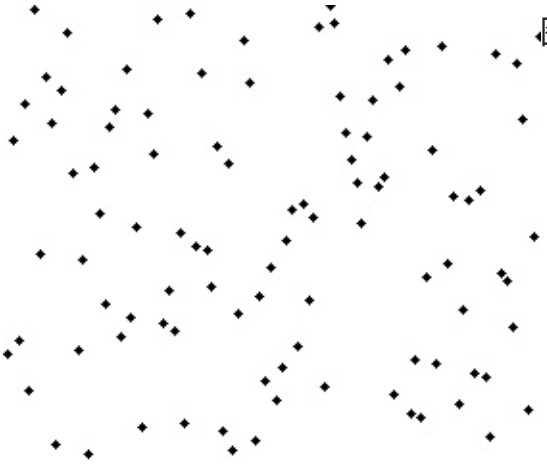
算法原理

> 归并排序（Merge Sort，台湾译作：合并排序）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（Divide and Conquer）的一个非常典型的应用。

> 归并操作(Merge)，也叫归并算法，指的是将两个已经排序的序列合并成一个序列的操作。归并排序算法依赖归并操作。归并排序有多路归并排序、两路归并排序，可用于内排序，也可以用于外排序。这里仅对内排序的两路归并方法进行讨论。

算法思路：

1. 把 n 个记录看成 n 个长度为 l 的有序子表
2. 进行两两归并使记录关键字有序，得到 n/2 个长度为 2 的有序子表
3. 重复第 2 步直到所有记录归并成一个长度为 n 的有序表为止。



图片来自维基百科

实例分析

以数组 array = [6, 5, 3, 1, 8, 7, 2, 4] 为例，首先将数组分为长度为 2 的子数组，并使每个子数组有序：

```
[6, 5]  [3, 1]  [8, 7]  [2, 4]
   ↓       ↓       ↓       ↓
[5, 6]  [1, 3]  [7, 8]  [2, 4]
```

然后再两两合并：

```
[6, 5, 3, 1]  [8, 7, 2, 4]
      ↓             ↓
[1, 3, 5, 6]  [2, 4, 7, 8]
```

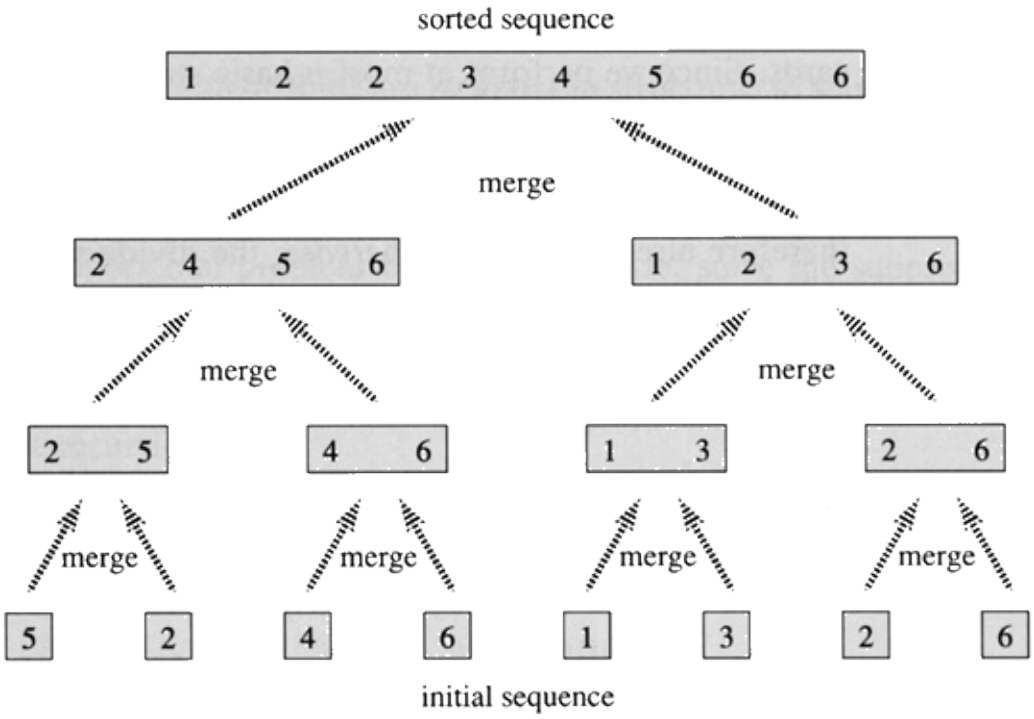
最后将两个子数组合并：

```
[6, 5, 3, 1, 8, 7, 2, 4]
            ↓
[1, 2, 3, 4, 5, 6, 7, 8]
```

排序过程动画演示如下：

图片来自维基百科

再有数组 array = [5, 2, 4, 6, 1, 3, 2, 6]，归并排序流程也可以如下表示：



JavaScript 语言实现

屌丝的惯例，上代码，由于要两两归并的子数组都是有序的数组，同时我们在希尔排序中提到过“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” ，所以我们可以将其中一个子数组中的元素依次插入到另一个数组当中，使其归并后成为一个有序的数组。代码如下：

```
function mergeSort(array) {

    function sort(array, first, last) {
        first = (first === undefined) ? 0 : first
        last = (last === undefined) ? array.length - 1 : last
        if (last - first < 1) {
            return;
        }
        var middle = Math.floor((first + last) / 2);
        sort(array, first, middle);
        sort(array, middle + 1, last);

        var f = first,
            m = middle,
            i,
            temp;

        while (f <= m && m + 1 <= last) {
            if (array[f] >= array[m + 1]) { // 这里使用了插入排序的思想
                temp = array[m + 1];
                for (i = m; i >= f; i--) {
                    array[i + 1] = array[i];
```

```
                }
                array[f] = temp;
                m++
            } else {
                f++
            }
        }

        return array;
    }

    return sort(array);
}
```

参考文章

- Wikipedia
- 维基百科，自由的百科全书
- Merge Sort
- 两路归并算法
- MERGE SORT 动画演示

（8）鸡尾酒排序 (Cocktail Sort/Shaker Sort)

算法原理

为什么叫鸡尾酒排序？其实我也不知道，知道的小伙伴请告诉我。

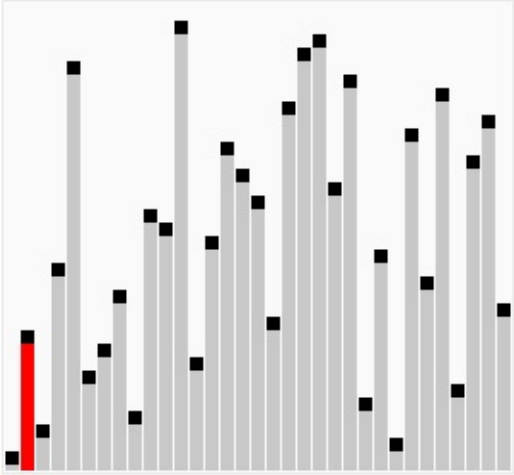
其实它还有很多奇怪的名称，比如双向冒泡排序 (Bidirectional Bubble Sort)、波浪排序 (Ripple Sort)、摇曳排序 (Shuffle Sort)、飞梭排序 (Shuttle Sort) 和欢乐时光排序 (Happy Hour Sort)。本文中就以鸡尾酒排序来称呼它。

> 鸡尾酒排序是冒泡排序的轻微变形。不同的地方在于，鸡尾酒排序是从低到高然后从高到低来回排序，而冒泡排序则仅从低到高去比较序列里的每个元素。他可比冒泡排序的效率稍微好一点，原因是冒泡排序只从一个方向进行比对(由低到高)，每次循环只移动一个项目。

以序列(2,3,4,5,1)为例，鸡尾酒排序只需要访问一次序列就可以完成排序，但如果使用冒泡排序则需要四次。但是在乱数序列状态下，鸡尾酒排序与冒泡排序的效率都很差劲，优点只有原理简单这一点。

排序过程：

1. 先对数组从左到右进行冒泡排序（升序），则最大的元素去到最右端
2. 再对数组从右到左进行冒泡排序（降序），则最小的元素去到最左端
3. 以此类推，依次改变冒泡的方向，并不断缩小未排序元素的范围，直到最后一个元素结束



图片来自维基百科

实例分析

以数组 array = [45, 19, 77, 81, 13, 28, 18, 19, 77] 为例，排序过程如下：

从左到右，找到最大的数 81，放到数组末尾：

```
| 19  45  77  13  28  18  19  77 |  81
```

从右到左，找到剩余数组（先框中的部分）中最小的数 ，放到数组开头：

```
13  | 19  45  77  18  28  19  77 |  81
```

从左到右，在剩余数组中找到最大数，放在剩余数组的末尾：

```
13  | 19  45  18  28  18  77 |  77  81
```

从右到左

```
13  18  | 19  45  18  28  77 |  77  81
```

从左到右

```
13   18   19   18   28   45   77   77   81
```

从右到左

```
13   18   18   19   28   45   77   77   81
```

从左到右

```
13   18   18   19   28   45   77   77   81
```

从右到左

```
13   18   18   19   28   45   77   77   81
```

JavaScript 语言实现

惯例，看代码：

```
function shakerSort(array) {

    function swap(array, i, j) {
        var temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    var length = array.length,
        left = 0,
        right = length - 1,
        lastSwappedLeft = left,
        lastSwappedRight = right,
        i,
        j;

    while (left < right) {
```

```
        // 从左到右
        lastSwappedRight = 0;
        for (i = left; i < right; i++) {
            if (array[i] > array[i + 1]) {
                swap(array, i, i + 1);
                lastSwappedRight = i;
            }
        }
        right = lastSwappedRight;
        // 从右到左
        lastSwappedLeft = length - 1;
        for (j = right; left < j; j--) {
            if (array[j - 1] > array[j]) {
                swap(array, j - 1, j)
                lastSwappedLeft = j
            }
        }
        left = lastSwappedLeft;
    }
}
```

参考文章

- 维基百科，自由的百科全书
- Cocktail Sort Algorithm or Shaker Sort Algorithm
- Sorting Algorithms: The Cocktail Sort
- [演算法]摇晃排序法(Shaker Sort)
- 冒泡排序与鸡尾酒排序

（9）猴子排序 (Bogo Sort)

算法原理

> 猴子排序 (Bogo Sort) 是个既不实用又原始的排序算法，其原理等同将一堆卡片抛起，落在桌上后检查卡片是否已整齐排列好，若非就再抛一次。其名字源自 Quantum bogodynamics，又称 bozo sort、blort sort 或猴子排序（参见无限猴子定理）。并且在最坏的情况下所需时间是无限的。

伪代码：

```
while not InOrder(list) do
 Shuffle(list)
done
```

这个排序方法没有办法给出实例分析，下面直接看代码。

JavaScript 语言实现

```
function bogoSort(array) {

    function swap(array, i, j) {
        var temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    // 随机交换顺序
    function shuffle(array) {
        var i,
            l = array.length;
        for (var i = 0; i < l; i++) {
            var j = Math.floor(Math.random() * l)
            swap(array, i, j)
        }
    }
    // 判断是否已经排好序
    function isSorted(array) {
        var i,
            l = array.length;
        for (var i = 1; i < l; i++) {
            if (array[i - 1] > array[i]) {
                return false;
            }
        }
```

```
            return true;
        }

        var sorted = false;
        while (sorted == false) { // 效率低下的位置
            v = shuffle(array);
            sorted = isSorted(array);
        }
        return array;
    }
```

参考文章

- 维基百科，自由的百科全书
- Sorting algorithms/Bogosort

（10）桶排序 (Bucket Sort)

算法原理

> 桶排序 (Bucket sort)或所谓的箱排序的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。

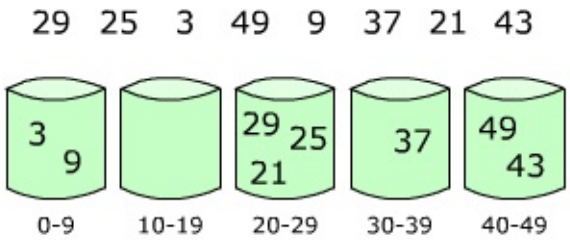
排序过程：

1. 假设待排序的一组数统一的分布在一个范围中，并将这一范围划分成几个子范围，也就是桶
2. 将待排序的一组数，分档规入这些子桶，并将桶中的数据进行排序
3. 将各个桶中的数据有序的合并起来

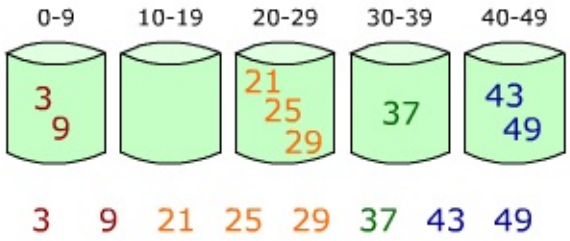
Data Structure Visualizations 提供了一个桶排序的分步动画演示。

实例分析

设有数组 array = [29, 25, 3, 49, 9, 37, 21, 43]，那么数组中最大数为 49，先设置 5 个桶，那么每个桶可存放数的范围为：0~9、10~19、20~29、30~39、40~49，然后分别将这些数放人自己所属的桶，如下图：



然后，分别对每个桶里面的数进行排序，或者在将数放入桶的同时用插入排序进行排序。最后，将各个桶中的数据有序的合并起来，如下图：



JavaScript 语言实现

首先用最笨的方法，每一个桶只能放相同的数字，最大桶的数量为数组中的正数最大值加上负数最小值的绝对值。

```
function bucketSort(array) {
    var bucket = [], // 正数桶
        negativeBucket = [], // 负数桶
        result = [],
        l = array.length,
        i,
        j,
        k,
        abs;

    // 入桶
    for (i = 0; i < l; i++) {
        if (array[i] < 0) {
            abs = Math.abs(array[i]);
            if (!negativeBucket[abs]) {
                negativeBucket[abs] = [];
            }
            negativeBucket[abs].push(array[i]);

        } else {
            if (!bucket[array[i]]) {
                bucket[array[i]] = [];
            }
            bucket[array[i]].push(array[i]);
        }
    }
    // 出桶
    l = negativeBucket.length;
    for (i = l - 1; i >= 0; i--) {
        if (negativeBucket[i]) {
            k = negativeBucket[i].length;
            for (j = 0; j < k; j++) {
                result.push(negativeBucket[i][j]);
            }
        }
    }

    l = bucket.length;
    for (i = 0; i < l; i++) {
        if (bucket[i]) {
```

```
            k = bucket[i].length;
            for (j = 0; j < k; j++) {
                result.push(bucket[i][j]);
            }
        }
    }

    return result;
}
```

下面这种方式就是文中举例分析的那样，每个桶存放一定范围的数字，用 step 参数来设置该范围，取 step 为 1 就退化成前一种实现方式。关键部位代码有注释，慢慢看，逻辑稍微有点复杂。

```
/*
* @array 将要排序的数组
*
* @step 划分桶的步长，比如 step = 5，表示每个桶存放的数字的范围是 5，像 -4~1、0~5
、6~11
*/
function bucketSort(array, step) {
    var result = [],
        bucket = [],
        bucketCount,
        l = array.length,
        i,
        j,
        k,
        s,
        max = array[0],
        min = array[0],
        temp;

    for (i = 1; i < l; i++) {
        if (array[i] > max) {
            max = array[i]
        }
        if (array[i] < min) {
            min = array[i];
        }
    }
    min = min - 1;

    bucketCount = Math.ceil((max - min) / step); // 需要桶的数量

    for (i = 0; i < l; i++) {
        temp = array[i];
        for (j = 0; j < bucketCount; j++) {
            if (temp > (min + step * j) && temp <= (min + step * (j + 1))
) { // 判断放入哪个桶
                if (!bucket[j]) {
                    bucket[j] = [];
```

```
                }
                // 通过插入排序将数字插入到桶中的合适位置
                s = bucket[j].length;
                if (s > 0) {
                    for (k = s - 1; k >= 0; k--) {
                        if (bucket[j][k] > temp) {
                            bucket[j][k + 1] = bucket[j][k];
                        } else {
                            break;
                        }
                    }
                    bucket[j][k + 1] = temp;
                } else {
                    bucket[j].push(temp);
                }
            }
        }
    }

    for (i = 0; i < bucketCount; i++) { // 循环取出桶中数据
        if (bucket[i]) {
            k = bucket[i].length;
            for (j = 0; j < k; j++) {
                result.push(bucket[i][j]);
            }
        }
    }

    return result;
}
```

参考文章

- 维基百科，自由的百科全书
- Programming Contest Central
- 桶排序（Bucket sort）
- Data Structure Visualizations
- 箱排序(Bin Sort)

（11）基数排序 (Radix Sort)

算法原理

> 基数排序 (Radix Sort) 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。基数排序的发明可以追溯到 1887 年赫尔曼·何乐礼在打孔卡片制表机 (Tabulation Machine)上的贡献。

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。

基数排序法会使用到桶 (Bucket)，顾名思义，通过将要比较的位（个位、十位、百位...），将要排序的元素分配至 0~9 个桶中，借以达到排序的作用，在某些时候，基数排序法的效率高于其它的比较性排序法。

Data Structure Visualizations 提供了一个基数排序的分步动画演示。

实例分析

基数排序的方式可以采用 LSD (Least sgnificant digital) 或 MSD (Most sgnificant digital)，LSD 的排序方式由键值的最右边开始，而 MSD 则相反，由键值的最左边开始。以 LSD 为例，假设原来有一串数值如下所示：

```
36   9   0   25   1   49   64   16   81   4
```

首先根据个位数的数值，按照个位置等于桶编号的方式，将它们分配至编号0到9的桶子中：

编号	0	1	2	3	4	5	6	7	8	9
	0	1			64	25	36			9
		81			4		16			49

然后，将这些数字按照桶以及桶内部的排序连接起来：

```
0   1   81   64   4   25   36   16   9   49
```

接着按照十位的数值，分别对号入座：

编号	0	1	2	3	4	5	6	7	8	9
	0	16	25	36	49		64		81	
	1									
	4									
	9									

最后按照次序重现连接，完成排序：

```
0   1   4   9   16   25   36   49   64   81
```

JavaScript 语言实现

暴力上代码：

```
function radixSort(array) {
    var bucket = [],
        l = array.length,
        loop,
        str,
        i,
        j,
        k,
        t,
        max = array[0];

    for (i = 1; i < l; i++) {
        if (array[i] > max) {
            max = array[i]
        }
    }

    loop = (max + '').length;

    for (i = 0; i < 10; i++) {
        bucket[i] = [];
    }

    for (i = 0; i < loop; i++) {
        for (j = 0; j < l; j++) {
            str = array[j] + '';
```

```
            if (str.length >= i + 1) {
                k = parseInt(str[str.length - i - 1]);
                bucket[k].push(array[j]);
            } else { // 高位为 0
                bucket[0].push(array[j]);
            }
        }

        array.splice(0, l);
        for (j = 0; j < 10; j++) {
            t = bucket[j].length;
            for (k = 0; k < t; k++) {
                array.push(bucket[j][k]);
            }
            bucket[j] = [];
        }
    }
    return array;
}
```

参考文章

- 维基百科，自由的百科全书
- Data Structure Visualizations
- Algorithm Gossip: 基数排序法
- Radix Sorting