

SunIBAS m
70824e5 4 hours ago

1 contributor

Raw Blame History

396 lines (334 sloc) 14.5 KB

排序

内容摘自

复杂度对比

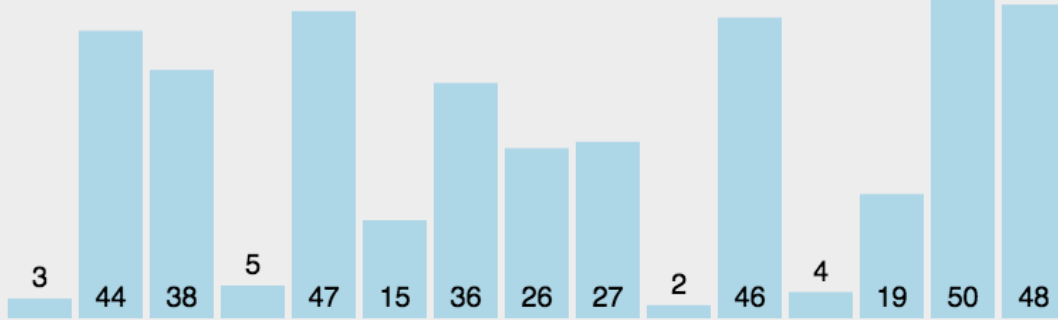
排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log_2n)$	$O(n\log_2n)$	$O(n\log_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n\log_2n)$	$O(n^2)$	$O(n\log_2n)$	$O(n\log_2n)$	不稳定
归并排序	$O(n\log_2n)$	$O(n\log_2n)$	$O(n\log_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

1、冒泡排序（Bubble Sort）

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

1.1 算法描述

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



```
function bubbleSort(arr) {
  var len = arr.length;
  for (var i = 0; i < len - 1; i++) {
    for (var j = 0; j < len - 1 - i; j++) {
      if (arr[j] > arr[j+1]) { // 相邻元素两两对比
        var temp = arr[j+1]; // 元素交换
        arr[j+1] = arr[j];
        arr[j] = temp;
      }
    }
  }
  return arr;
}
```

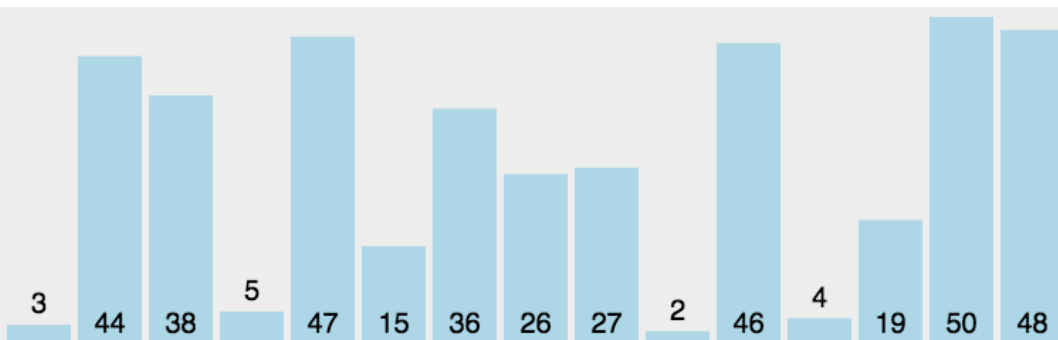
2、选择排序 (Selection Sort)

选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：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。

2.1 算法描述

n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。具体算法描述如下：

- 初始状态：无序区为R[1..n]，有序区为空；
- 第i趟排序(i=1,2,3...n-1)开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n)。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1..i]和R(i+1..n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
- n-1趟结束，数组有序化了。



```
function selectionSort(arr) {
  var len = arr.length;
  var minIndex, temp;
  for (var i = 0; i < len - 1; i++) {
    minIndex = i;
    for (var j = i + 1; j < len; j++) {
      if (arr[j] < arr[minIndex]) { // 寻找最小的数
        minIndex = j;             // 将最小数的索引保存
      }
    }
    temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

```
    }  
    return arr;  
}
```

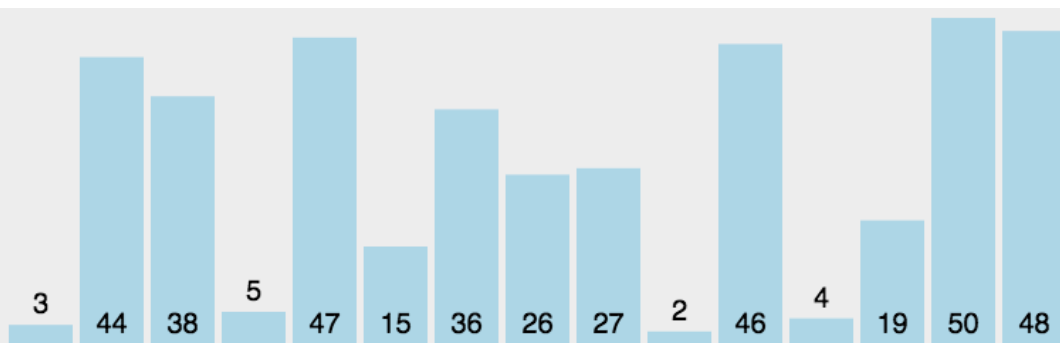
3、插入排序 (Insertion Sort)

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

3.1 算法描述

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

- 从第一个元素开始，该元素可以认为已经被排序；
- 取出下一个元素，在已经排序的元素序列中从后向前扫描；
- 如果该元素（已排序）大于新元素，将该元素移到下一位置；
- 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；
- 将新元素插入到该位置后；
- 重复步骤2~5。



```
function insertionSort(arr) {  
    var len = arr.length;  
    var preIndex, current;  
    for (var i = 1; i < len; i++) {  
        preIndex = i - 1;  
        current = arr[i];  
        while (preIndex >= 0 && arr[preIndex] > current) {  
            arr[preIndex + 1] = arr[preIndex];  
            preIndex--;  
        }  
        arr[preIndex + 1] = current;  
    }  
    return arr;  
}
```

4、希尔排序 (Shell Sort)

1959年Shell发明，第一个突破 $O(n^2)$ 的排序算法，是简单插入排序的改进版。它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫缩小增量排序。

4.1 算法描述

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

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

第一遍



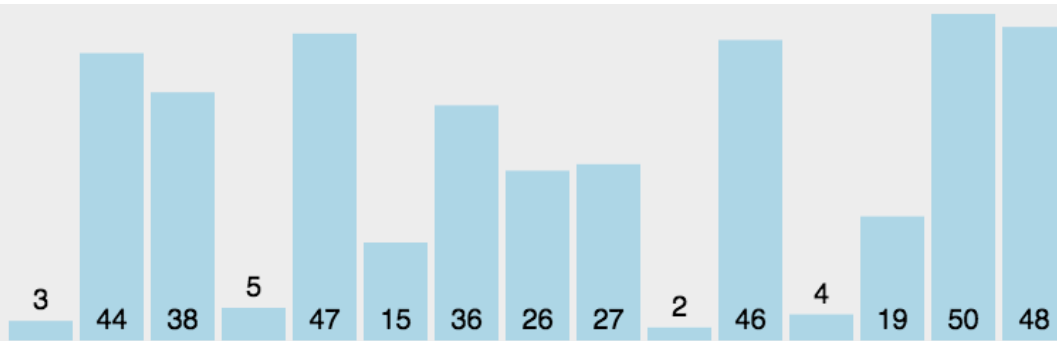
```
function shellSort(arr) {  
    var len = arr.length,  
        temp,  
        gap = 1;  
    while (gap < len / 3) {          // 动态定义间隔序列  
        gap = gap * 3 + 1;  
    }  
    for (gap; gap > 0; gap = Math.floor(gap / 3)) {  
        for (var i = gap; i < len; i++) {  
            temp = arr[i];  
            for (var j = i - gap; j > 0 && arr[j] > temp; j -= gap) {  
                arr[j + gap] = arr[j];  
            }  
            arr[j + gap] = temp;  
        }  
    }  
    return arr;  
}
```

5、归并排序 (Merge Sort)

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

5.1 算法描述

- 把长度为 n 的输入序列分成两个长度为 $n/2$ 的子序列；
- 对这两个子序列分别采用归并排序；
- 将两个排序好的子序列合并成一个最终的排序序列。



```
function mergeSort(arr) { // 采用自上而下的递归方法
  var len = arr.length;
  if (len < 2) {
    return arr;
  }
  var middle = Math.floor(len / 2),
      left = arr.slice(0, middle),
      right = arr.slice(middle);
  return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right) {
  var result = [];
  while (left.length > 0 && right.length > 0) {
    if (left[0] <= right[0]) {
      result.push(left.shift());
    } else {
      result.push(right.shift());
    }
  }
  while (left.length)
    result.push(left.shift());
  while (right.length)
    result.push(right.shift());
  return result;
}
```

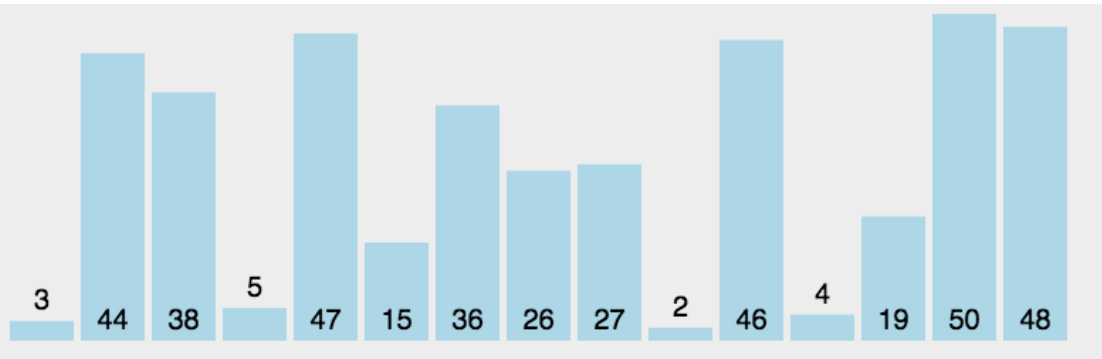
6、快速排序 (Quick Sort)

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

6.1 算法描述

快速排序使用分治法来把一个串 (list) 分为两个子串 (sub-lists)。具体算法描述如下：

- 从数列中挑出一个元素，称为“基准” (pivot)；
- 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；
- 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。



```
function quickSort(arr, left, right) {
    var len = arr.length,
        partitionIndex,
        left = typeof left !== 'number' ? 0 : left,
        right = typeof right !== 'number' ? len - 1 : right;
    if (left < right) {
        partitionIndex = partition(arr, left, right);
        quickSort(arr, left, partitionIndex-1);
        quickSort(arr, partitionIndex+1, right);
    }
    return arr;
}

function partition(arr, left, right) { // 分区操作
    var pivot = left, // 设定基准值 (pivot)
        index = pivot + 1;
    for (var i = index; i <= right; i++) {
        if (arr[i] < arr[pivot]) {
            swap(arr, i, index);
            index++;
        }
    }
    swap(arr, pivot, index - 1);
    return index-1;
}

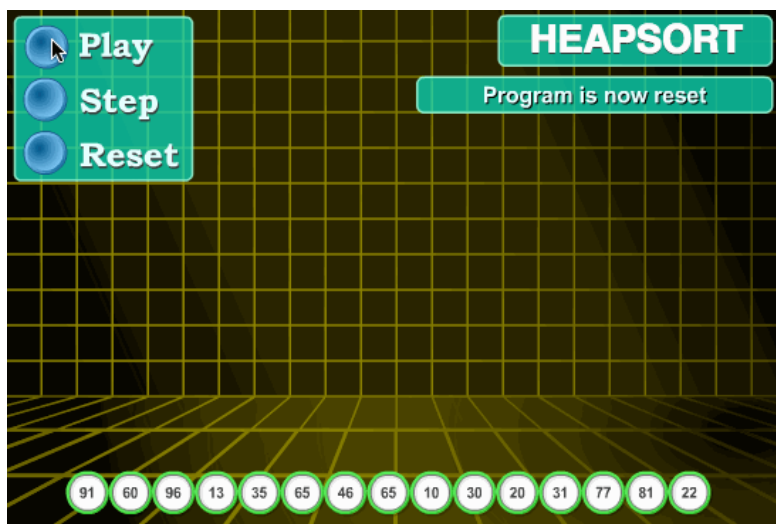
function swap(arr, i, j) {
    var temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

7、堆排序 (Heap Sort)

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。

7.1 算法描述

- 将初始待排序关键字序列(R1,R2....Rn)构建成大顶堆，此堆为初始的无序区；
- 将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,.....Rn-1)和新的有序区(Rn),且满足R[1,2...n-1] ≤ R[n]；
- 由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,.....Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2....Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为 n-1，则整个排序过程完成。



```
var len; // 因为声明的多个函数都需要数据长度，所以把len设置成为全局变量
```

```
function buildMaxHeap(arr) { // 建立大顶堆
    len = arr.length;
    for (var i = Math.floor(len/2); i >= 0; i--) {
        heapify(arr, i);
    }
}
```

```
function heapify(arr, i) { // 堆调整
    var left = 2 * i + 1,
        right = 2 * i + 2,
        largest = i;
    if (left < len && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < len && arr[right] > arr[largest]) {
        largest = right;
    }
    if (largest != i) {
        swap(arr, i, largest);
        heapify(arr, largest);
    }
}
```

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

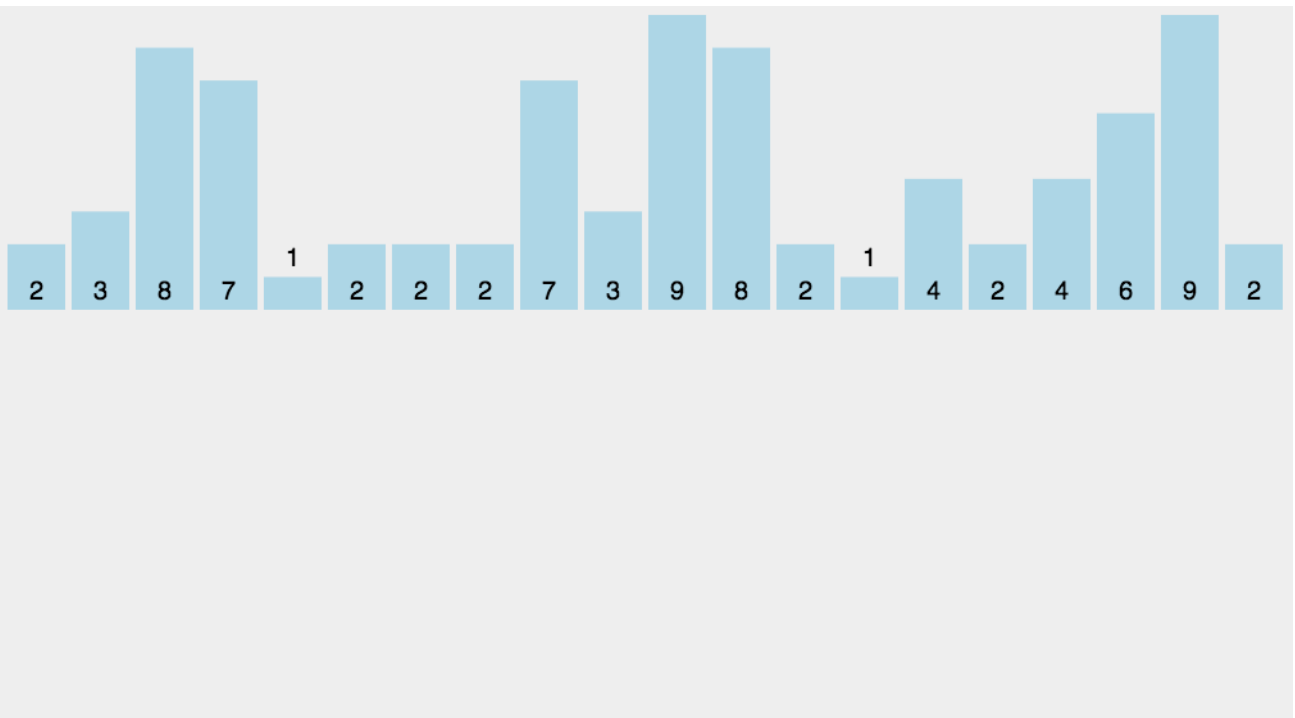
```
function heapSort(arr) {
    buildMaxHeap(arr);
    for (var i = arr.length - 1; i > 0; i--) {
        swap(arr, 0, i);
        len--;
        heapify(arr, 0);
    }
    return arr;
}
```

8、计数排序 (Counting Sort)

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

8.1 算法描述

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



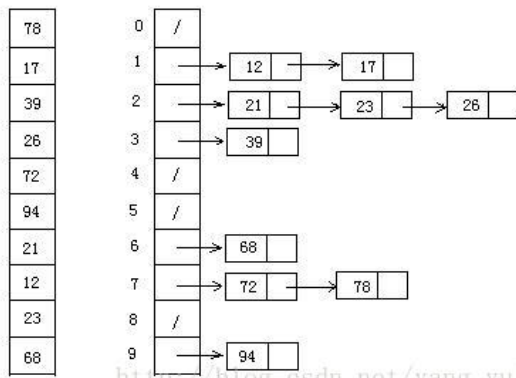
```
function countingSort(arr, maxValue) {  
  var bucket = new Array(maxValue + 1),  
      sortedIndex = 0;  
  arrLen = arr.length,  
  bucketLen = maxValue + 1;  
  for (var i = 0; i < arrLen; i++) {  
    if (!bucket[arr[i]]) {  
      bucket[arr[i]] = 0;  
    }  
    bucket[arr[i]]++;  
  }  
  for (var j = 0; j < bucketLen; j++) {  
    while(bucket[j] > 0) {  
      arr[sortedIndex++] = j;  
      bucket[j]--;  
    }  
  }  
  return arr;  
}
```

9、桶排序 (Bucket Sort)

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。桶排序 (Bucket sort)的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（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）。

9.1 算法描述

- 设置一个定量的数组当作空桶；
- 遍历输入数据，并且把数据一个一个放到对应的桶里去；
- 对每个不是空的桶进行排序；
- 从不是空的桶里把排好序的数据拼接起来。



```
function bucketSort(arr, bucketSize) {
    if (arr.length === 0) {
        return arr;
    }
    var i;
    var minValue = arr[0];
    var maxValue = arr[0];
    for (i = 1; i < arr.length; i++) {
        if (arr[i] < minValue) {
            minValue = arr[i];           // 输入数据的最小值
        } else if (arr[i] > maxValue) {
            maxValue = arr[i];           // 输入数据的最大值
        }
    }
    // 桶的初始化
    var DEFAULT_BUCKET_SIZE = 5;         // 设置桶的默认数量为5
    bucketSize = bucketSize || DEFAULT_BUCKET_SIZE;
    var bucketCount = Math.floor((maxValue - minValue) / bucketSize) + 1;
    var buckets = new Array(bucketCount);
    for (i = 0; i < buckets.length; i++) {
        buckets[i] = [];
    }
    // 利用映射函数将数据分配到各个桶中
    for (i = 0; i < arr.length; i++) {
        buckets[Math.floor((arr[i] - minValue) / bucketSize)].push(arr[i]);
    }
    arr.length = 0;
    for (i = 0; i < buckets.length; i++) {
        insertionSort(buckets[i]);         // 对每个桶进行排序，这里使用了插入排序
        for (var j = 0; j < buckets[i].length; j++) {
            arr.push(buckets[i][j]);
        }
    }
    return arr;
}
```

10、基数排序 (Radix Sort)

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。

10.1 算法描述

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

3

44

38

5

47

15

36

26

27

2

46

4

19

50

48

```
// LSD Radix Sort
var counter = [];
function radixSort(arr, maxDigit) {
    var mod = 10;
    var dev = 1;
    for (var i = 0; i < maxDigit; i++, dev *= 10, mod *= 10) {
        for(var j = 0; j < arr.length; j++) {
            var bucket = parseInt((arr[j] % mod) / dev);
            if(counter[bucket]==null) {
                counter[bucket] = [];
            }
            counter[bucket].push(arr[j]);
        }
        var pos = 0;
        for(var j = 0; j < counter.length; j++) {
            var value = null;
            if(counter[j]!=null) {
                while ((value = counter[j].shift()) !=null) {
                    arr[pos++] = value;
                }
            }
        }
    }
    return arr;
}
```