## 如何用两个栈实现队列的功能：

两个栈形成一个队列的核心就是用栈先进后出的结构构造出队列先进先出的结构（确定队列的出队元素和入队元素即可）。

两个栈：栈S1、栈S2，栈S1用来收集元素，栈S2主要起调度缓冲区的作用。

做法：

队列入列时：直接将元素压到栈S1，S1的栈顶元素就是队列的入列元素；

队列出列时：先判断栈S2中是否有元素，若有元素，说明栈S1中的元素已经全都倒入S2了，这时S2的栈顶的元素肯定就是第一个入队列的元素，即出队元素。如果S2中没有元素，先将S1中的元素倒入S2中，再弹出S2的栈顶元素，即出队元素。

这种做法效率最高，避免了两个栈多次倒入倒出操作

## 怎么将一个数组中的奇数全都删掉

1. 遍历整个数组 ，然后将里面的基数删掉。（取余是否为0）
2. 采用另外一个新的数组，将里面的偶数元素复制到新数组中
3. 对数组，设置两个指针，一个在第一位i=0，一个在最后一位j=arr.length;

<1>如果 i<=j&&i<arr.length，否则执行完成。判断arr[i]%2==0?

如果为0，执行第<2>部,否则执行<3>;

<2> i++,然后执行<1>;

<3> arr[j]%2==1？

如果为1，则执行第<4>部，否则执行<5>

<4>j--,然后执行 <3> ；

<5>调换arr[i]与arr[j]，然后i++,j--;继续执行<1>

## 字符串【816】消除，不能使用字符串内置函数

例如：81881616被消除后，结果是 81 ，818[816]16 81[816] 81 输出81

|  |
| --- |
| **public static** String test(String str) {  **char**[] chars = str.toCharArray();  **int** num = -1;  **for** (**int** i = 0; i < chars.**length**; i++) {  **if** (i <= chars.**length** - 3) {  **if** (chars[i] - 48 == 8 && chars[i + 1] - 48 == 1 && chars[i + 2] - 48 == 6) {  num = i;  }  }  }  **if** (num != -1) {  String s1 = **new** String(chars, 0, num);  String s2 = **new** String(chars, num + 3, chars.**length** - num - 3);  **return** test2(s1 + s2);  }  **return** str; } |

## 字符串反转

## 数组内部奇数和偶数数量相同，顺序是随机的。请写一个程序将基数放在基数下标，偶数放在偶数下标。 要求：就在原始数组上进行操作，不能借助额外空间（定义的基础变量除外），时间复杂度要好.

|  |
| --- |
| */\*\**  \* 数组内部奇数和偶数数量相同，顺序是随机的。请写一个程序将基数放在基数下标，偶数放在偶数下标。  \* 要求：就在原始数组上进行操作，不能借助额外空间（定义的基础变量除外），时间复杂度要好. *\*/* **public class** Test4 {  **public static void** main(String[] args) {  **int**[] arr=**new int**[]{0,1,3,2,4,5};  *test*(arr);  }  **public static void** test(**int**[] arr){  **for**(**int** i=0, j=arr.**length**-1;i<arr.**length**-1&&j>0;){  **if**(arr[i]%2==0){  i=i+2;  }**else if**(arr[j]%2==1){  j=j-2;  }**else if**(arr[i]%2==1&&arr[j]%2==0){  **int** temp=arr[i];  arr[i]=arr[j];  arr[j]=temp;  i=i+2;  j=j-2;  }  }  System.***out***.println(Arrays.*toString*(arr));  } } |

## 从10万个数中找10个最大的数

对于这种题目，最普通的想法是先对这10万个数进行排序，然后再选取数组中前10个数，即为最后的答案，排序算法的时间复杂度不下于O(N lgN)。最好的方法是建立一个最小堆。   
算法描述：   
我们首先取10万个元素中的前10个元素来建立由10个元素组成的最小堆。这样堆顶元素便是当前已知元素的第10大的数；然后依次读取剩下的99990个元素，若读取的元素比堆顶元素大，则将堆顶元素和当前元素替换，并自堆顶至下调整堆；这样读取完所有元素后，堆中的10个元素即为这10万个数最大的10个数，同时堆顶元素为这10万个元素第10大元素。   
时间复杂度：   
设从N个数中找M个最大数   
每次重新恢复堆的时间复杂都为O(logM),最多供进行了（N-M）次恢复堆操作，顾时间复杂度为O(NlogM)

## 排序算法有几种，写 一下快速排序和堆排序。

[**十大经典排序算法**](http://www.cnblogs.com/onepixel/articles/7674659.html)

0、排序算法说明

0.1 排序的定义

对一序列对象根据某个关键字进行排序。

0.2 术语说明

稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；

不稳定：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；

内排序：所有排序操作都在内存中完成；

外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；

时间复杂度： 一个算法执行所耗费的时间。

空间复杂度：运行完一个程序所需内存的大小。

0.3 算法总结



图片名词解释：

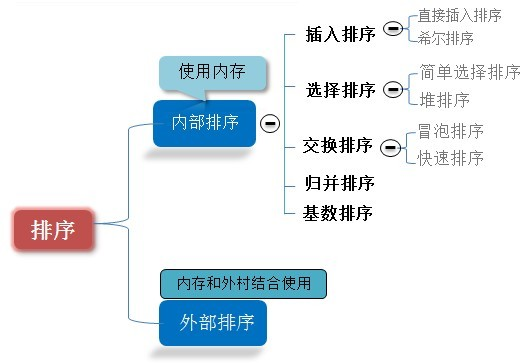
n: 数据规模

k: “桶”的个数

In-place: 占用常数内存，不占用额外内存

Out-place: 占用额外内存

0.5 算法分类



### 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

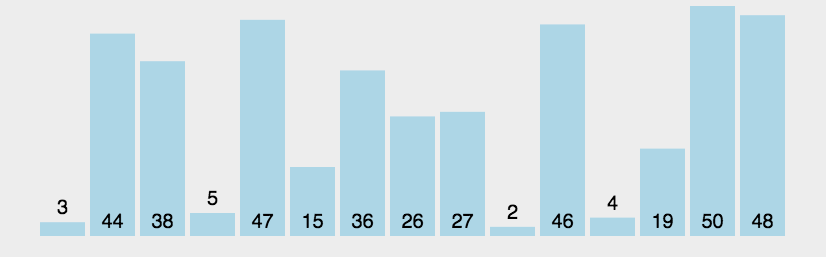
比较相邻的元素。如果第一个比第二个大，就交换它们两个；

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；

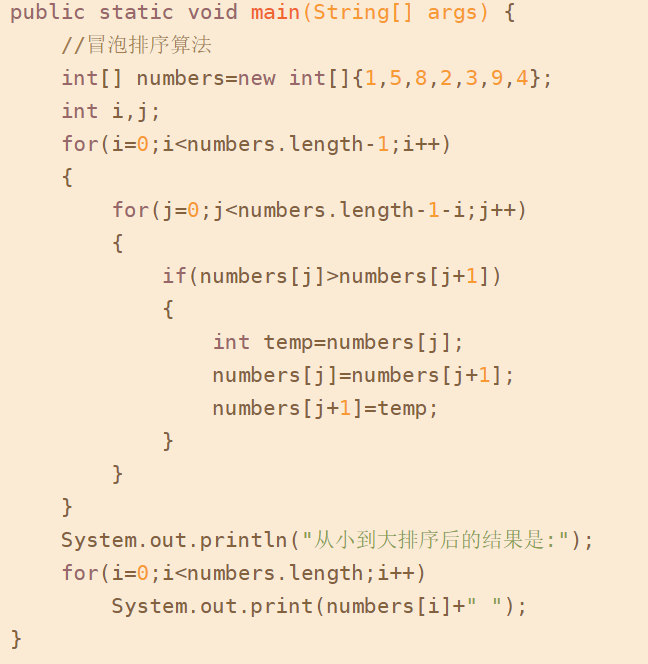
针对所有的元素重复以上的步骤，除了最后一个；

重复步骤1~3，直到排序完成。

1.2 动图演示



1.3 代码实现



1.4 算法分析

最佳情况：T(n) = O(n)   最差情况：T(n) = O(n2)   平均情况：T(n) = O(n2)

### 2、选择排序（Selection Sort）

表现最稳定的排序算法之一，因为无论什么数据进去都是O(n2)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。理论上讲，选择排序可能也是平时排序一般人想到的最多的排序方法了吧。

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。

2.2 动图演示



2.3 代码实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | 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;  } |

2.4 算法分析

最佳情况：T(n) = O(n2)  最差情况：T(n) = O(n2)  平均情况：T(n) = O(n2)

### 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。插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

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；

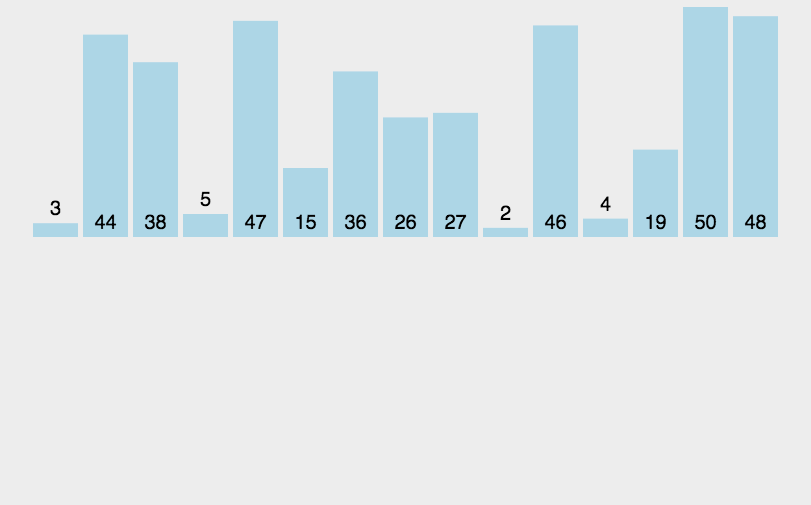
如果该元素（已排序）大于新元素，将该元素移到下一位置；

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。

3.2 动图演示



3.2 代码实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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;  } |

3.4 算法分析

最佳情况：T(n) = O(n)   最坏情况：T(n) = O(n2)   平均情况：T(n) = O(n2)

### 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。

希尔排序的核心在于间隔序列的设定。既可以提前设定好间隔序列，也可以动态的定义间隔序列。动态定义间隔序列的算法是《算法（第4版》的合著者Robert Sedgewick提出的。

4.1 算法描述

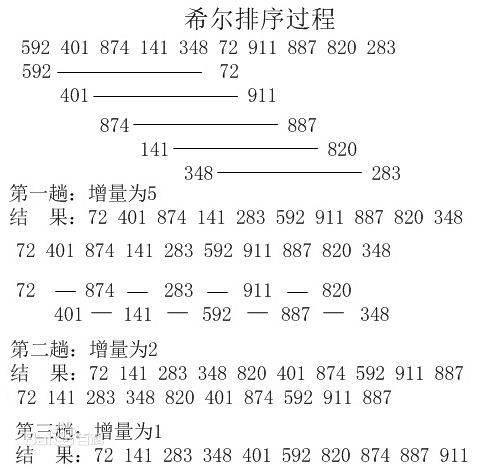
先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=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ti，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。

4.2 过程演示



4.3 代码实现

|  |
| --- |
| 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;  } |

4.4 算法分析

最佳情况：T(n) = O(nlog2 n)  最坏情况：T(n) = O(nlog2 n)  平均情况：T(n) =O(nlog n)

5、归并排序（Merge Sort）

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。

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。归并排序是一种稳定的排序方法。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。

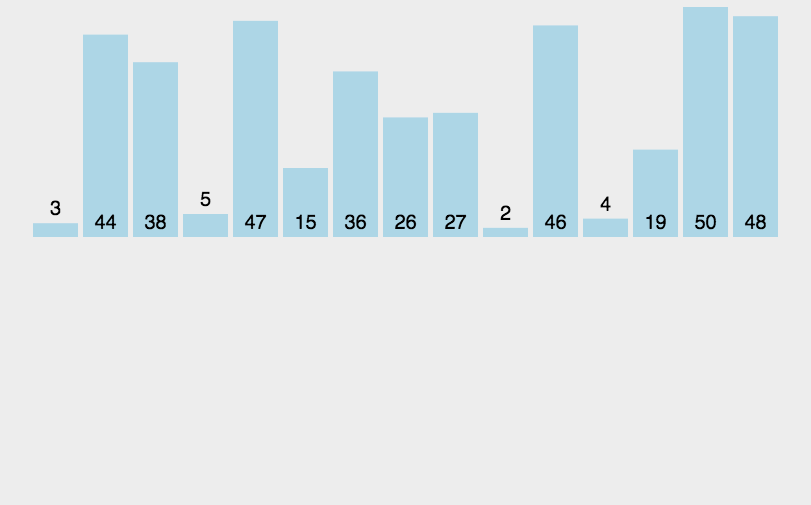
5.1 算法描述

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。

5.2 动图演示



5.3 代码实现

|  |
| --- |
| 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;  } |

5. 4 算法分析

最佳情况：T(n) = O(n)  最差情况：T(n) = O(nlogn)  平均情况：T(n) = O(nlogn)

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。

5.2 动图演示



5.3 代码实现

|  |
| --- |
| 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;  } |

5.4 算法分析

最佳情况：T(n) = O(nlogn)   最差情况：T(n) = O(n2)   平均情况：T(n) = O(nlogn)

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。

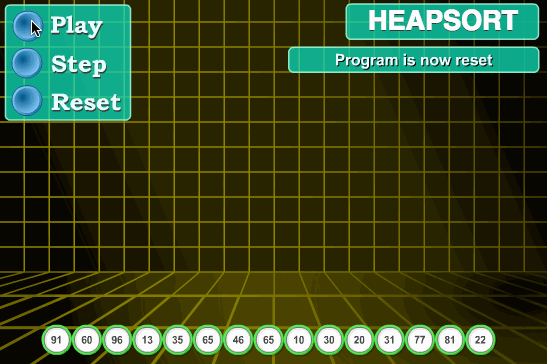
7.1 算法描述

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。

7.2 动图演示



7.3 代码实现

|  |
| --- |
| 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;  } |

7.4 算法分析

最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)

8、计数排序（Counting Sort）

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。

计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数。然后根据数组C来将A中的元素排到正确的位置。它只能对整数进行排序。

8.1 算法描述

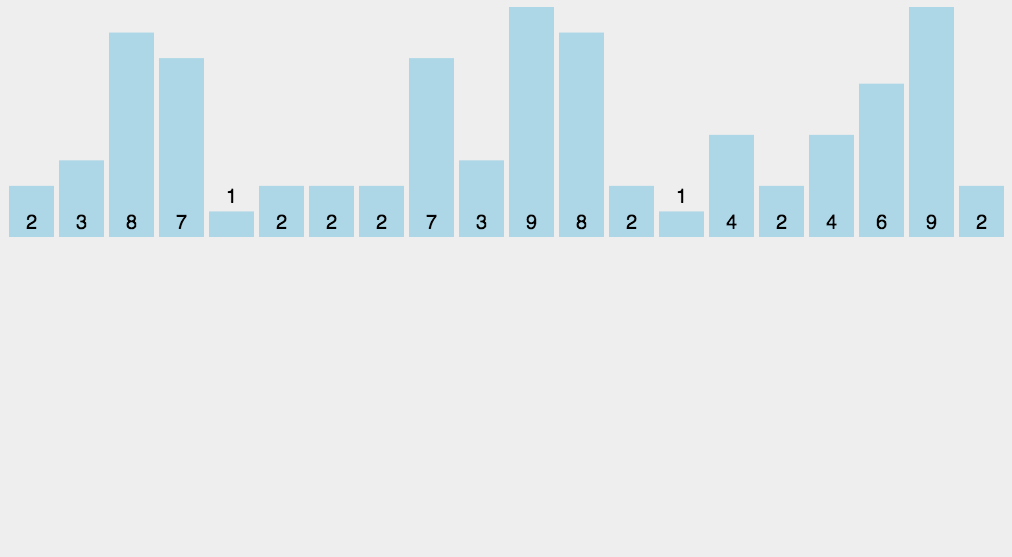
找出待排序的数组中最大和最小的元素；

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。

8.2 动图演示



8.3 代码实现

|  |
| --- |
| 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;  } |

8.4 算法分析

当输入的元素是n 个0到k之间的整数时，它的运行时间是 O(n + k)。计数排序不是比较排序，排序的速度快于任何比较排序算法。由于用来计数的数组C的长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上1），这使得计数排序对于数据范围很大的数组，需要大量时间和内存。

最佳情况：T(n) = O(n+k)  最差情况：T(n) = O(n+k)  平均情况：T(n) = O(n+k)

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。

9.2 图片演示



9.3 代码实现

|  |
| --- |
| 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;  } |

9.4 算法分析

桶排序最好情况下使用线性时间O(n)，桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，因为其它部分的时间复杂度都为O(n)。很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。

最佳情况：T(n) = O(n+k)   最差情况：T(n) = O(n+k)   平均情况：T(n) = O(n2)

10、基数排序（Radix Sort）

基数排序也是非比较的排序算法，对每一位进行排序，从最低位开始排序，复杂度为O(kn),为数组长度，k为数组中的数的最大的位数；

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。基数排序基于分别排序，分别收集，所以是稳定的。

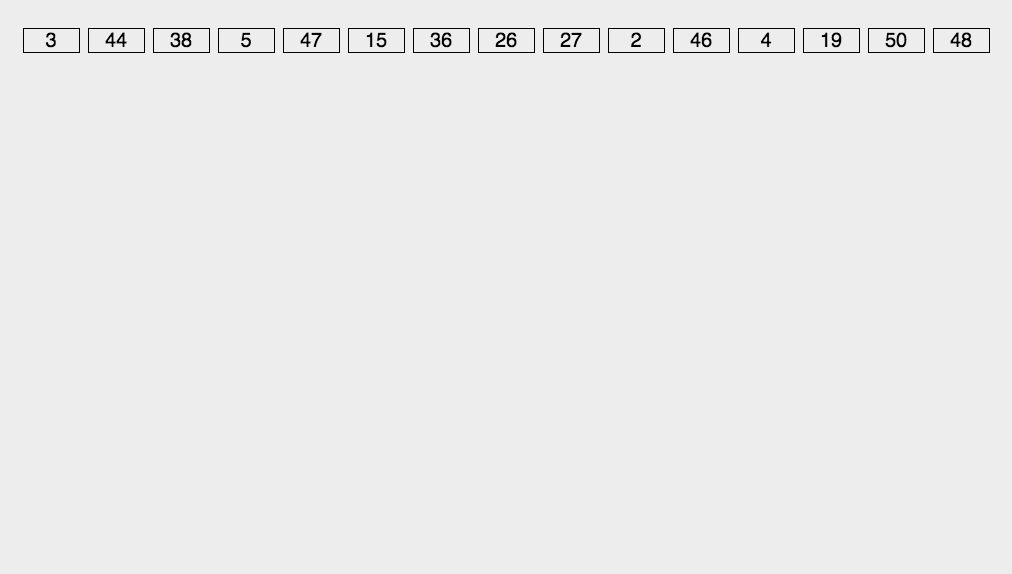
10.1 算法描述

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）；

10.2 动图演示



10.3 代码实现

|  |
| --- |
| //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;  } |

10.4 算法分析

最佳情况：T(n) = O(n \* k)   最差情况：T(n) = O(n \* k)   平均情况：T(n) = O(n \* k)

基数排序有两种方法：

MSD 从高位开始进行排序 LSD 从低位开始进行排序

基数排序 vs 计数排序 vs 桶排序

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：

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

## 最大堆排序

## 最小堆排序

## 从100g 中找到前100个出现最多的ip

## 封装一个接口，这个接口调用3个第三方，返回最快返回的那个第三方。（这3个第三方实现的功能一样）

## 二分查找算法

|  |
| --- |
| **public static void** main(String[] args) {  **int**[] arr = **new int**[]{9, 8, 7, 7, 6, 6, 5, 4, 3, 3, 3, 3, 3, 2, 1};  **int** num = 7;  System.***out***.println(*search*(arr, num, 0, arr.**length** - 1)); } */\*\*  \* 9, 8, 7, 7, 6, 6, 5, 4, 3, 2, 1  \* 在有序数组中(倒叙)，查找指定元素第一次出行的位置  \*  \** ***@param arr*** *\** ***@param num*** *\** ***@return*** *\*/* **public static int** search(**int**[] arr, **int** num, **int** left, **int** right) {  **if** (left > right) {  **return** -1;  }  **int** mid = (left + right) / 2;  **if** (arr[mid] == num) {  **return** mid;  } **else if** (arr[mid] > num) {  left = mid + 1;  } **else** {  right = mid - 1;  }  **return** *search*(arr, num, left, right); } |

## 快速排序

## 将两个有序链表连接在一起

## 二分查找的升级版本-在有序数组中(倒叙)，查找指定元素的位置第一次出行的位置

|  |
| --- |
| **public static void** main(String[] args) {  **int**[] arr = **new int**[]{9, 8, 7, 7, 6, 6, 5, 4, 3, 3, 3, 3, 3, 2, 1};  **int** num = 7;  System.***out***.println(*search*(arr, num, 0, arr.**length** - 1)); } */\*\*  \* 9, 8, 7, 7, 6, 6, 5, 4, 3, 2, 1  \* 在有序数组中(倒叙)，查找指定元素第一次出行的位置  \*  \** ***@param arr*** *\** ***@param num*** *\** ***@return*** *\*/* **public static int** search(**int**[] arr, **int** num, **int** left, **int** right) {  **if** (left > right) {  **return** -1;  }  **int** mid = (left + right) / 2;  **if** (arr[mid] == num) {  **if** (mid == 0 || arr[mid - 1] != arr[mid]) {  **return** mid;  }**else**{  right=mid-1;  }  } **else if** (arr[mid] > num) {  left = mid + 1;  } **else** {  right = mid - 1;  }  **return** *search*(arr, num, left, right); } |