Sort in Java



1.Selection sort 选择

2.Insertion sort 插入

3.Heap sort 堆

4.Merge sort 合并

5.Quick sort 快速

6.Bubble sort 冒泡

7.Radix sort 基数

8.Shell sort 希尔

9.Bucket sort 桶

Out-place: Merge sort, Counting sort, Bucket sort, Radix sort

Comparison sorting algorithm是array of elements能相互比较，element是string也可以

比较排序有: Bubble sort, Heap sort, Merge sort, Quick sort,Insertion sort, Shell sort, Selection Sort适用一切排序

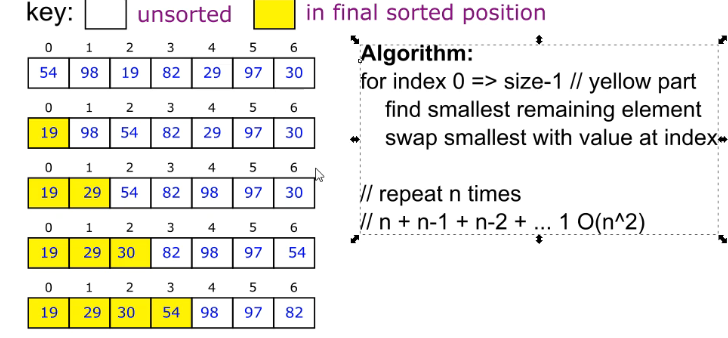
非比较排序有: Radix sort, bucket sort, counting sort，相对时间复杂度低，但要extra memory

Stable:如果a原本在b前面, 而a=b，排序之后a仍然在b前面

Unstable：如果a原本在b前面，而a=b，排序之后a可能会出现b的后面

1. **Selection sort 选择**

worst&best O(n^2) unstable(array时)， stable(linked list时)



54 98 19 82 29 97 30

把第一个数(54)作为sorted position，再往后search array里最小的数和54换位置，再把98作为放到sorted position，再重复搜索整个array里最小的数字和98换

public static void selectionSort(int[] array) {

for (int i = 0; i < array.length; i++) {

int min = array[i];

int minId = i;

for (int j = i+1; j < array.length; j++) {

if (array[j] < min) {

min = array[j];

minId = j;

}

}

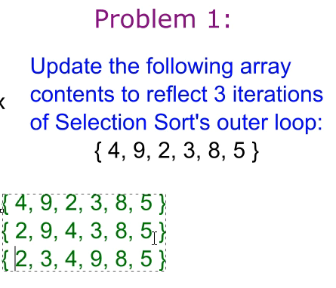
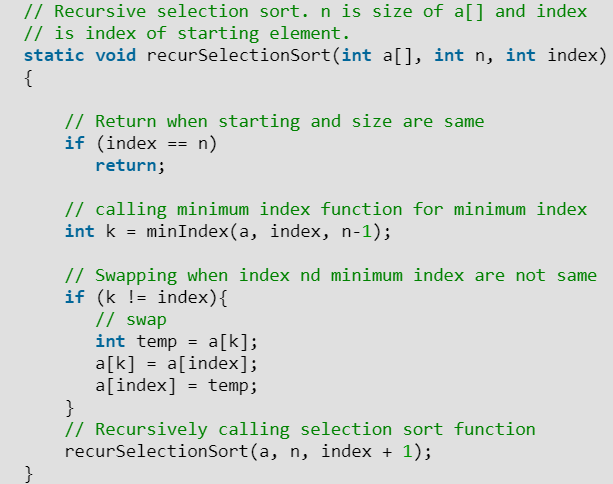
// swapping

int temp = array[i];

array[i] = min;

array[minId] = temp; } }

Recursive

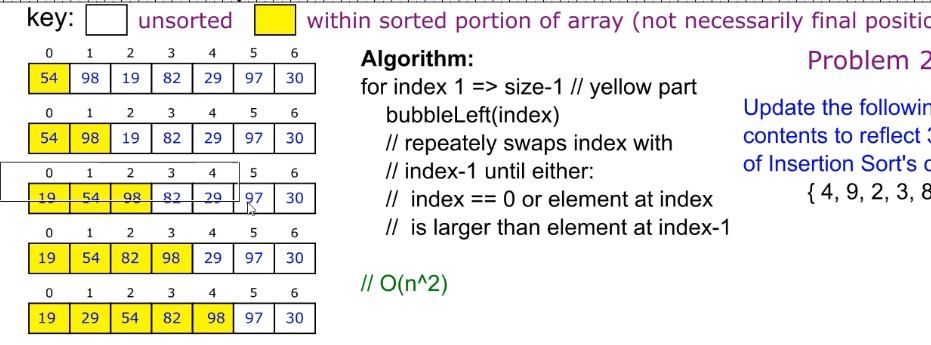


1. **Insertion sort 插入**

worst O(n^2)--每次都要把新元素放sorted position最前面 best O(n) stable

sorted position不是final position

每次sorted position多包括一个index，然后在sorted position里sort--把新的index元素放在合适的位置



public static void insertionSort(int[] array) {

for (int i = 1; i < array.length; i++) {

int current = array[i];

int j = i - 1;

while(j >= 0 && current < array[j]) {

array[j+1] = array[j];

j--; }

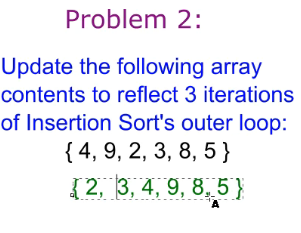
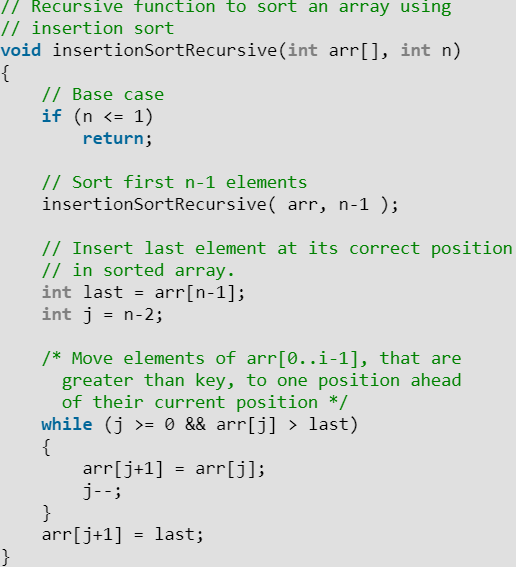
// at this point we've exited, so j is either -1

// or it's at the first element where current >= a[j]

array[j+1] = current;

} }

Recursive



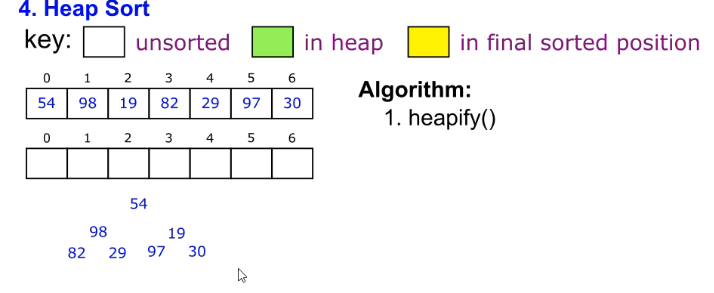
1. **Heap sort 堆**

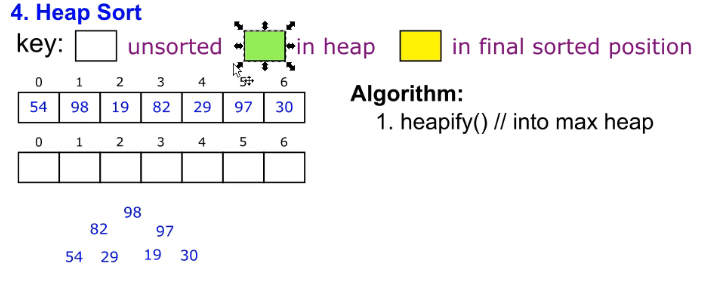
worst&best O(n\*log2n) unstable

简单的说就是把heap先heapify成max-heap，再把root(最大value)放到末尾处(sorted part)--和末尾处的element互换位置，（之后的操作不用考虑sorted part）对heap进行percolate down，这样root又是最大value，重复之前步骤

1.先转换成max-heap结构，再percolate down，再转换回array结构变成

98 82 97 54 29 19 30



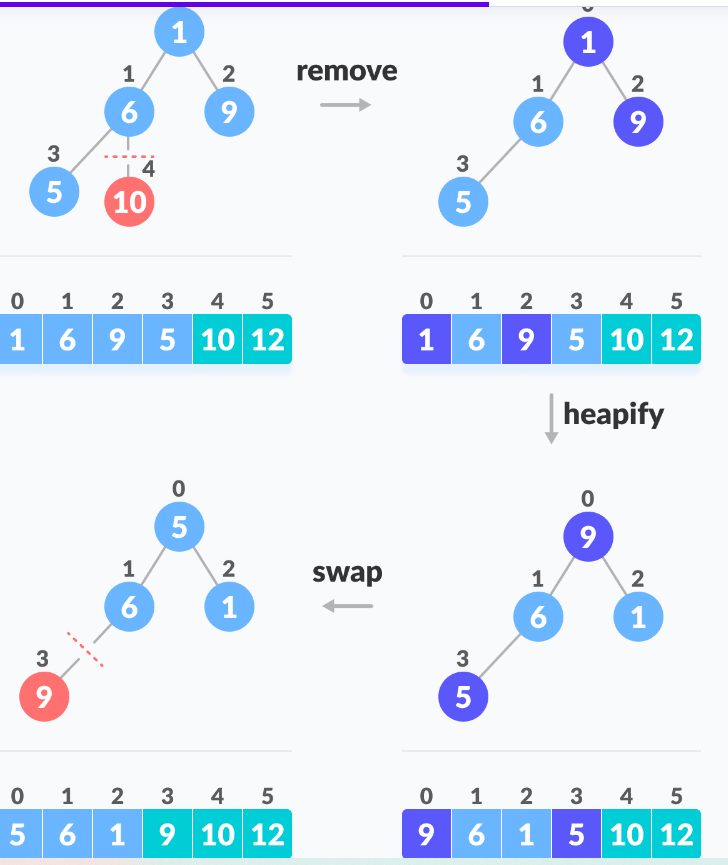
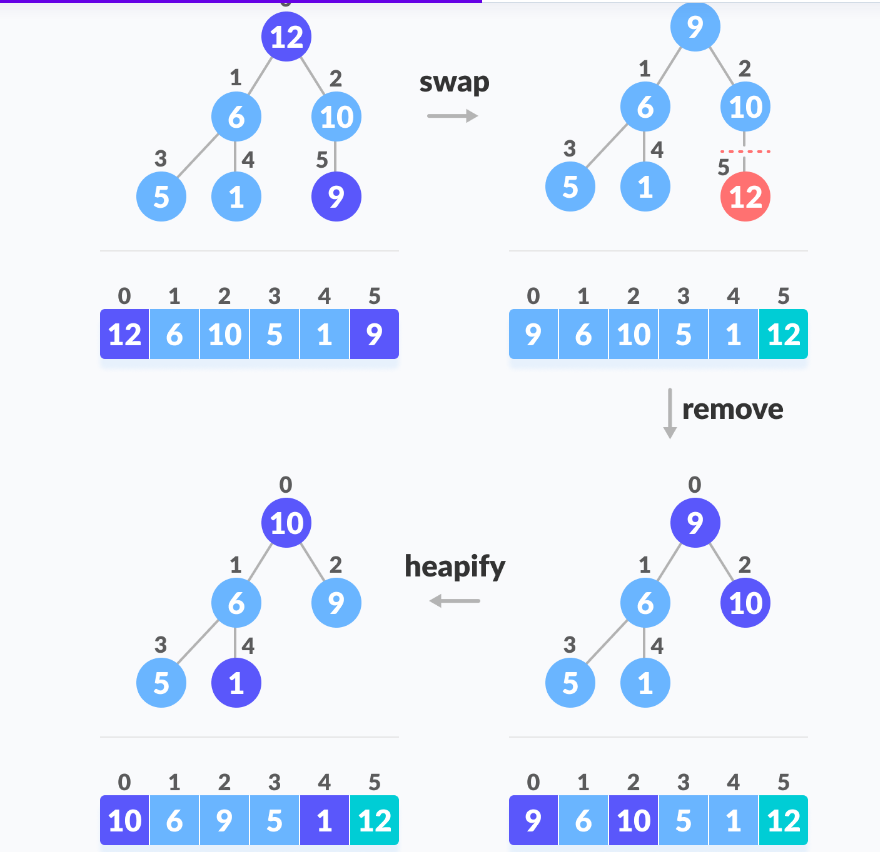


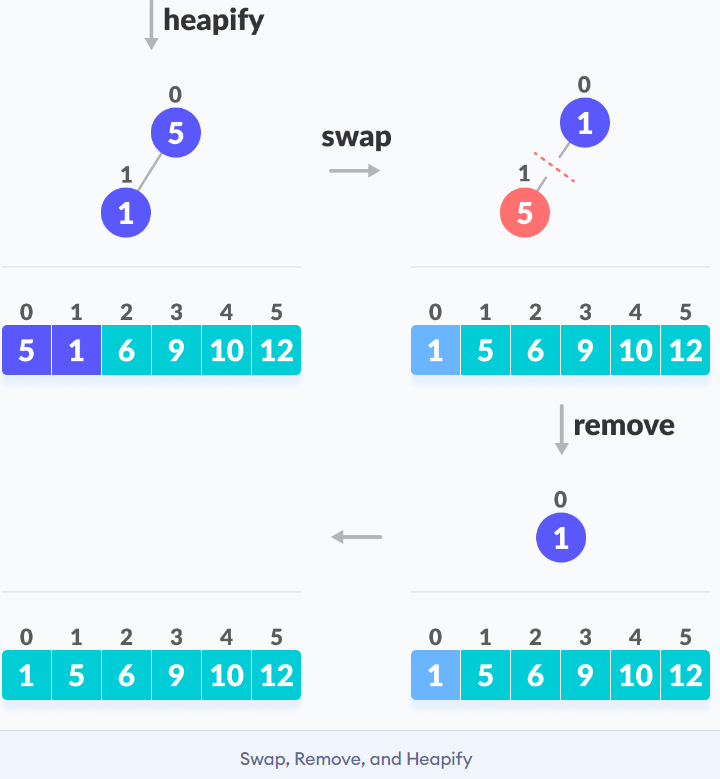
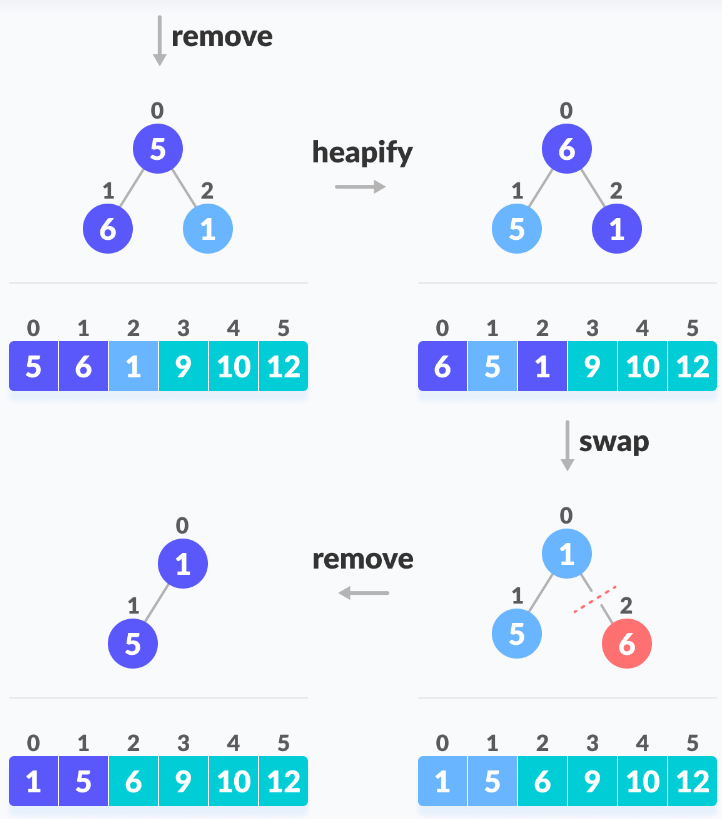
1. 开始heap sort，把最大的value放最后

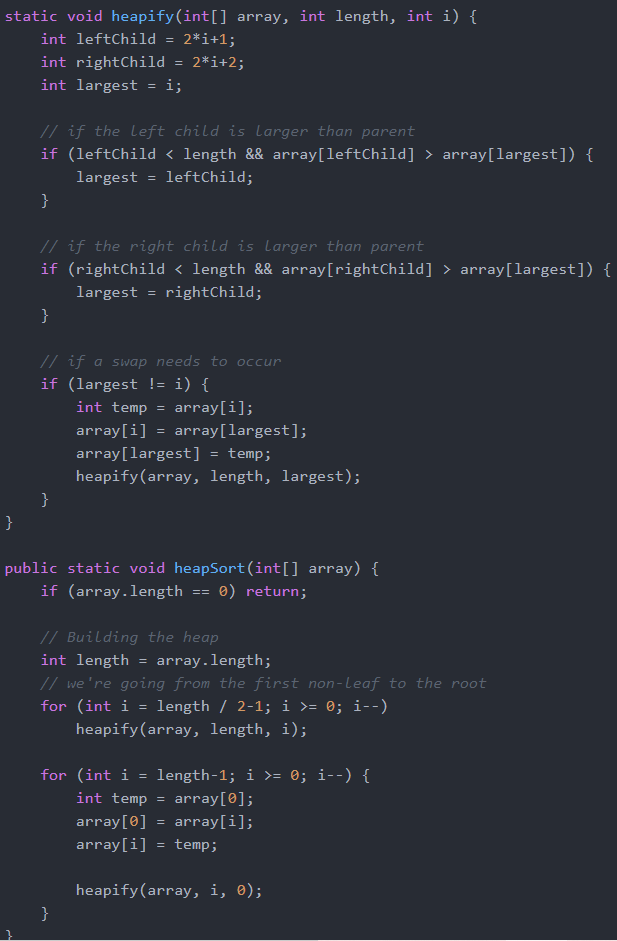
图从左到右

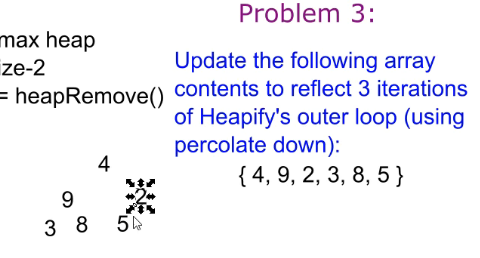
12

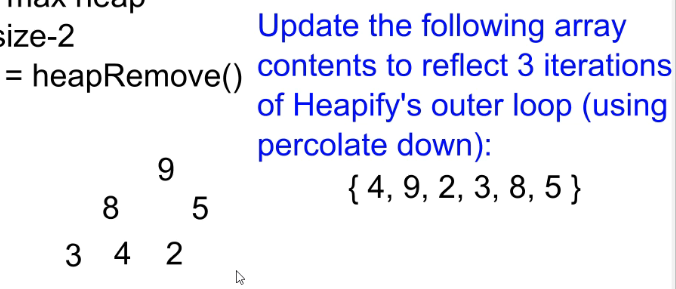
34











9 8 5 3 4 2 三次swap

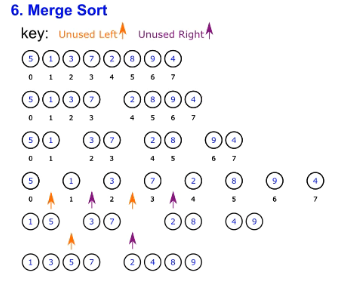
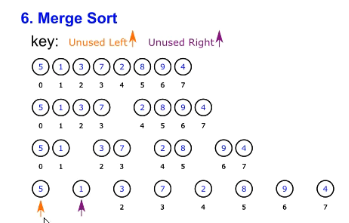
1. **Merge sort 合并**

worst/best O(n\*log2n)--outer recursive call O(log2n) inner loop--O(n),

如果是两个sorted array则是O(n)

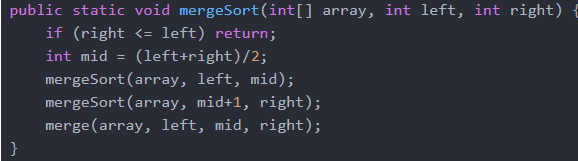
stable

把array二分直到每个子序只有一个element，再合并，每次合并都伴随sort



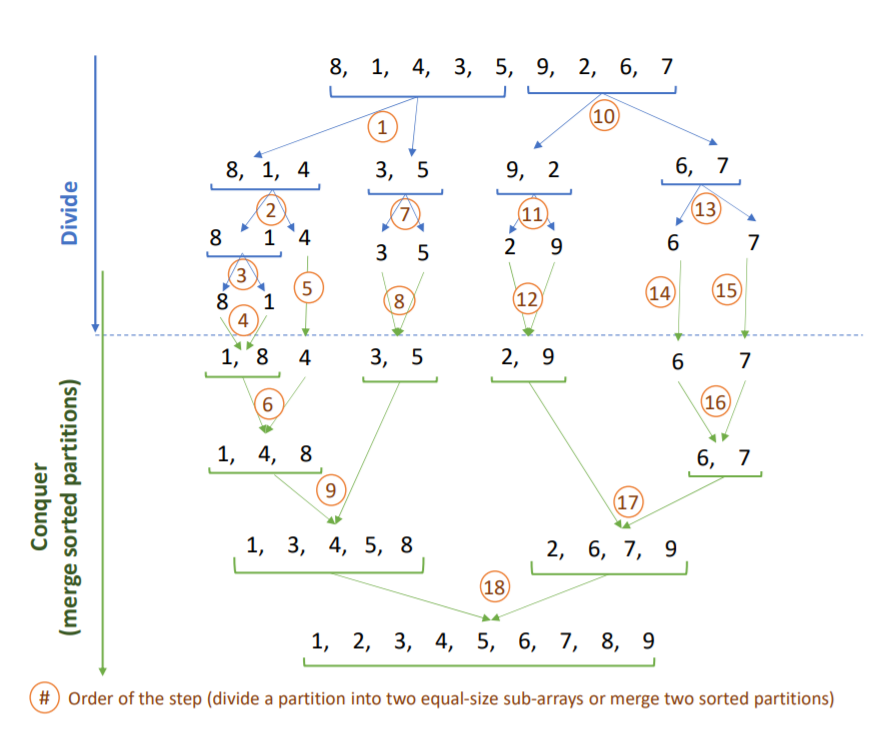
先合并element第1和2 ，3和4，再合并12和34，右边同理



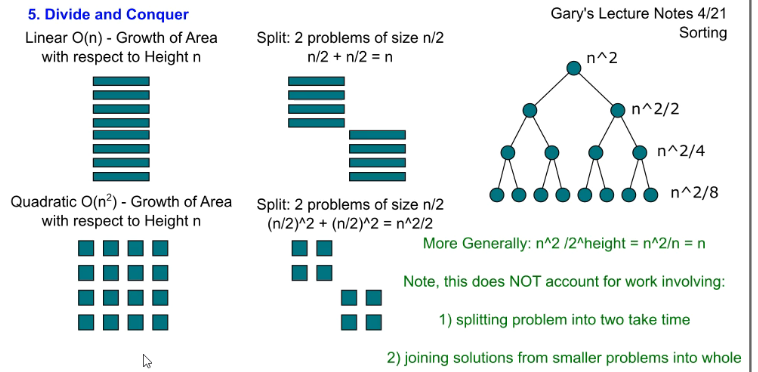


9 4 2 3 8 5 1 7

4 9 2是最先copy到temporary array的values



Merge sort是分治法(Divide and Conquer)的一种经典应用



1. **Quick sort 快速**

worst O(n^2)把最大或最小的element作为pivot--inner loop O(n),recursive O(n)

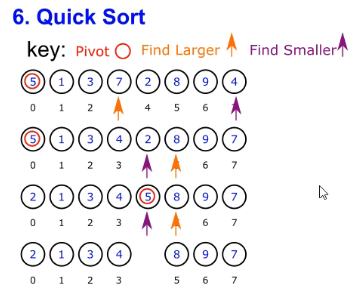
所以element很多的时候用quick好，不容易是O(n^2)

best/average O(n\*log2n) 选pivot时候正好是中位数median

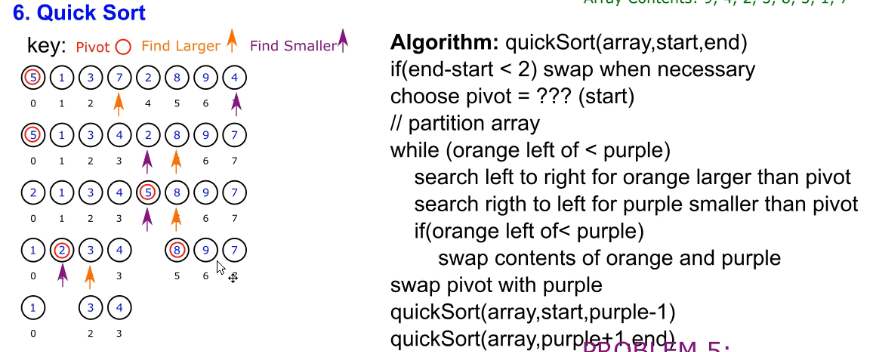
unstable

也运用了divide and conquer--分治法

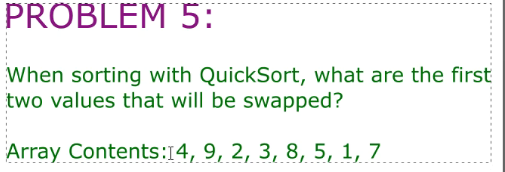
先选择一个数作为pivot开始partition，把比pivot小的element放pivot左边，大的放右边，再进行recursive call来排序



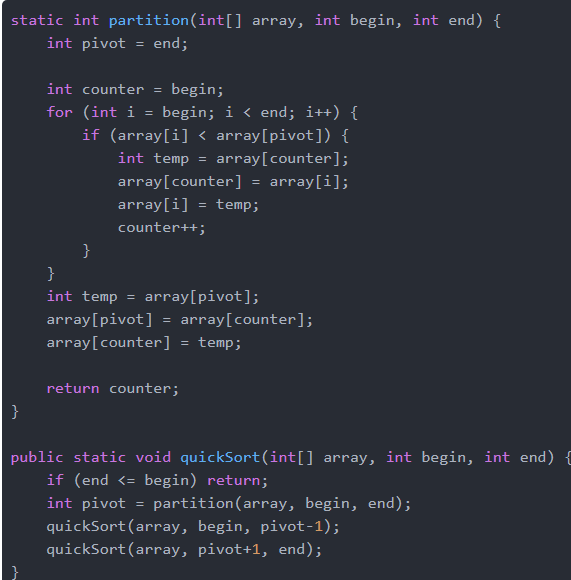
1. 任意选择一个数作为pivot--5
2. 从左到右开始找比5大的数--7设置成Find Larger
3. 在7的右边从右往左找有没有比5小的数(如果没有则partition完成),把比5小的数设置成Find Smaller
4. 把Find Larger和Find Smaller的数互换位置
5. 之后重复上面从左往右找比5大的第一个数作为Find Larger--8，从8的右边从右往左找比5小的数--这次没有了--也就是Find Smaller cross over Find Larger
6. partition完成，准备开始recursive
7. 在recursive之前再做一次交换，把pivot和Find Smaller的element互换
8. 开始recursive call on 2 1 3 4，另一个recursive call on 8 9 7--对2134设置pivot重复上面步骤



base case: if(end-start<2)

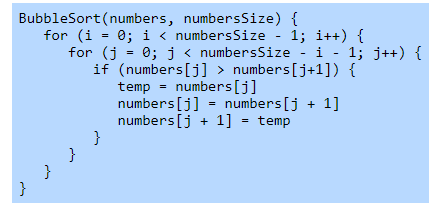


4是pivot，9和1是first two values被swapped



1. **Bubble sort 冒泡**

best/worst O(N)--用nested loop stable



index012345，0和1先比较，再1和2比较，2和3比较，每次比较都会sort，这样一对对比较到最后一个index，最后一个index就是最大的element(sorted part)再从头开始开始一对对比较，每次比较都会放一个unsorted part里最大element到sorted part



1. **Radix sort 基数**排序--为Integer设计

**排序大数据，比如100万个手机号，10G的订单号**

best/worst:O(k\*n) k是数字的最大位数 stable out-place

把一个array里按element的数字的最大位数来放到一个0-9 buckets里

比如要sort 1 2 34 53 56 88 91

1.

则放到0-9 buckets则是(按照个位数放)

91

1 2 53 34 56 88

0 1 2 3 4 5 6 7 8 9---这行为不同的bucket

2.再放回去

91 1 2 53 34 56 88 --同一个bucket的数会先把底下的数放回去，这样保证stable

3.再按照十位数放

56

1 2 34 53 88 91

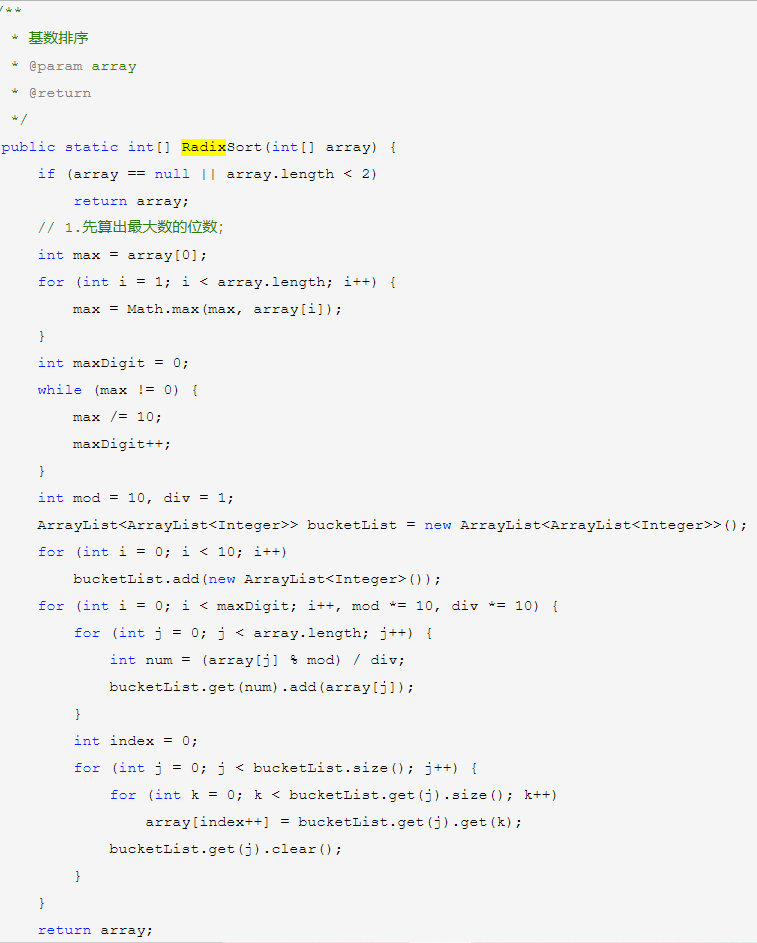
0 1 2 3 4 5 6 7 8 9---这行为不同的bucket

4.再放回去

1 2 34 53 56 88 91 sort完成

从1’ digit开始，再10’s digit, 100’s digit.....

如果所有数都是两位数，但用100’s digit来排，则每个数都在bucket 0

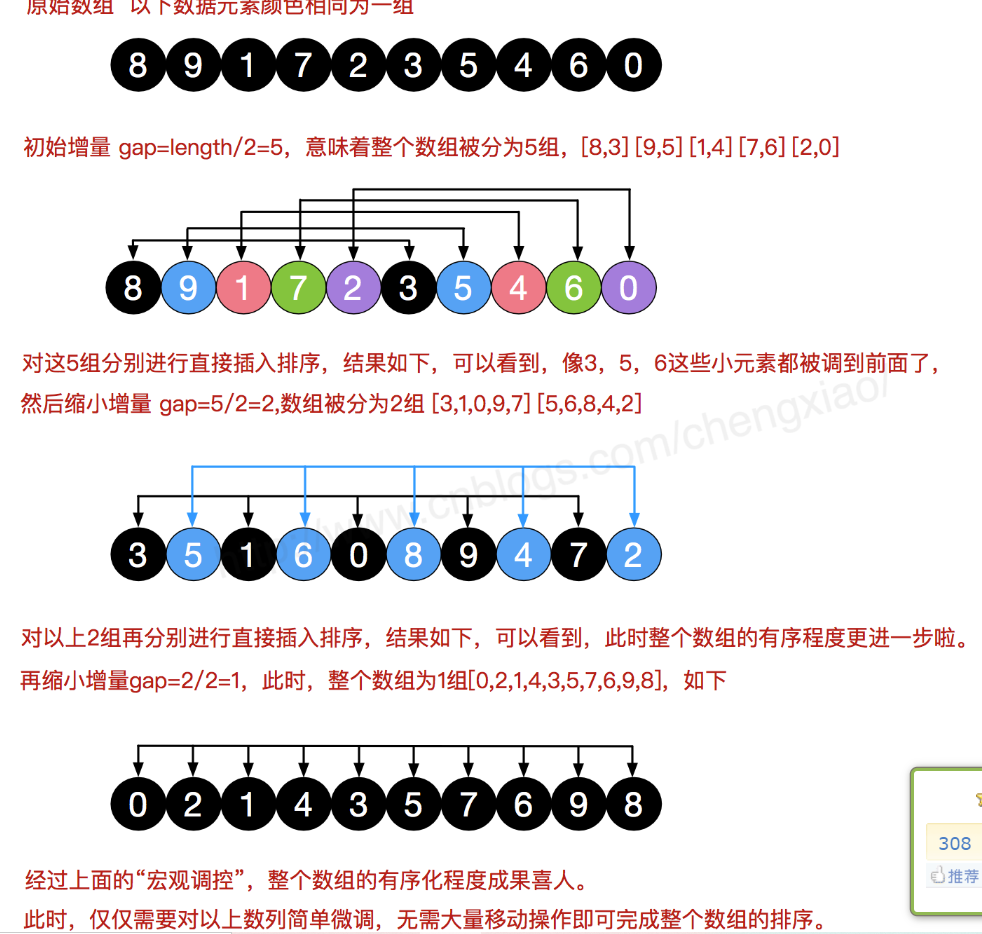


Sorting signed integers

array里有negative integer则要2个bucket分别放negative integer和non-negative integer，再把negative integer用reverse order和non-negative integer放到一起

1. **Shell sort 希尔**

best/worst O(nlog n) unstable

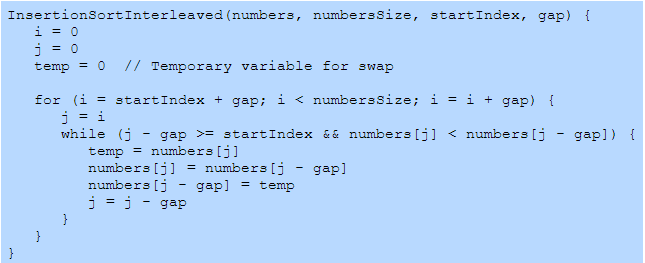


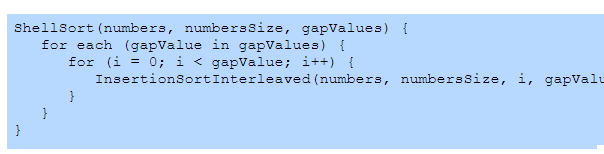
先选一个gap value，之后每次除以2

array size=6

gap value=2，则有2个interleaved list，每个list有3个element

gap value=4，则是2个有2个element的list和2个只有1个element的list





1. **Bucket Sort 桶**

**根据年龄给100万用户排序时候用，空间换时间**

best：O(n+k) worst：O(n+k) Average：O(n^2) stable

**为什么是O(n)**

如果要排序的数据有n 个，我们将其均分到 m 个桶内，每个桶里就有 k = n/m 个元素。每个桶内部使用快速排序，时间复杂度为O(klogk)。m个桶排序的时间复杂度就是 O(mklogk)，把 k = n/m 代入可得时间复杂度为O(nlog(n/m))。当桶的个数m接近数据个数n时，log(n/m)就是一个非常小的常量。这时候桶排序的时间复杂度就接近于O(n)

**桶排序的缺点：**

1. 要排序的数据需要很容易能划分到 m 个桶，并且，桶与桶之间有着天然的大小顺序。这样每个桶内的数据排序完之后，桶与桶之间就不需要再进行排序了
2. 需要数据在各个桶之间分布是比较均匀的。如果数据经过桶的划分后，有些桶的数据非常多，有的非常少，很不均匀，那桶内排序的时间复杂度就不是常量级了，比如**如果数据都被划分到一个桶里，那就退化成O(nlogn)**的[排序算法](https://so.csdn.net/so/search?q=%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/every__day/article/details/_blank)了

**把桶排序用于外部排序(大数据排序)**：

**问题**：比如我们有10G 的订单数据/手机号码，我们希望根据订单金额（假设金额过都是正整数）进行排序，但我们的内存有限，只有几百MB，没办法一次性把10GB 的数据加载到内存中，这个时候怎么办呢？

**解决**：我们可以先扫描一遍文件，确定订单金额的数据范围。假定订单最小值是1元，最大值是10万元。把订单划分到100个桶里，第一个桶里订单金额的范围是1——1000，第二个桶的范围是1001——2000，依次类推。一个桶对应一个文件，文件编号从 00——99。

理想情况下，10GB的订单数据比较平均的分到100个文件中，每个文字里有约100MB的订单数据。然后把这100个文件依次放入内存中，用快排来排序。当所有文件里的订单数据排好序之后。再按照文件编号，从小到大把这100个的数据写入到一个文件中，就可以得到这10G订单排序好的数据了。

但数据不大可能均匀分布，若某个桶内的数据特别多，比如订单金额5001——6000的特别多，可以将这个区间的订单数据再细分到十个桶内（5001——5100，5101——5200……），其它的操作同上，就可以解决问题了。总之某个桶里的数据，内存放下下，就再细分

每个bucket都有额外的sorting algorithm

non-negative数字的array才能用bucket sort

bucket index = [number\*(N/(M+1))]

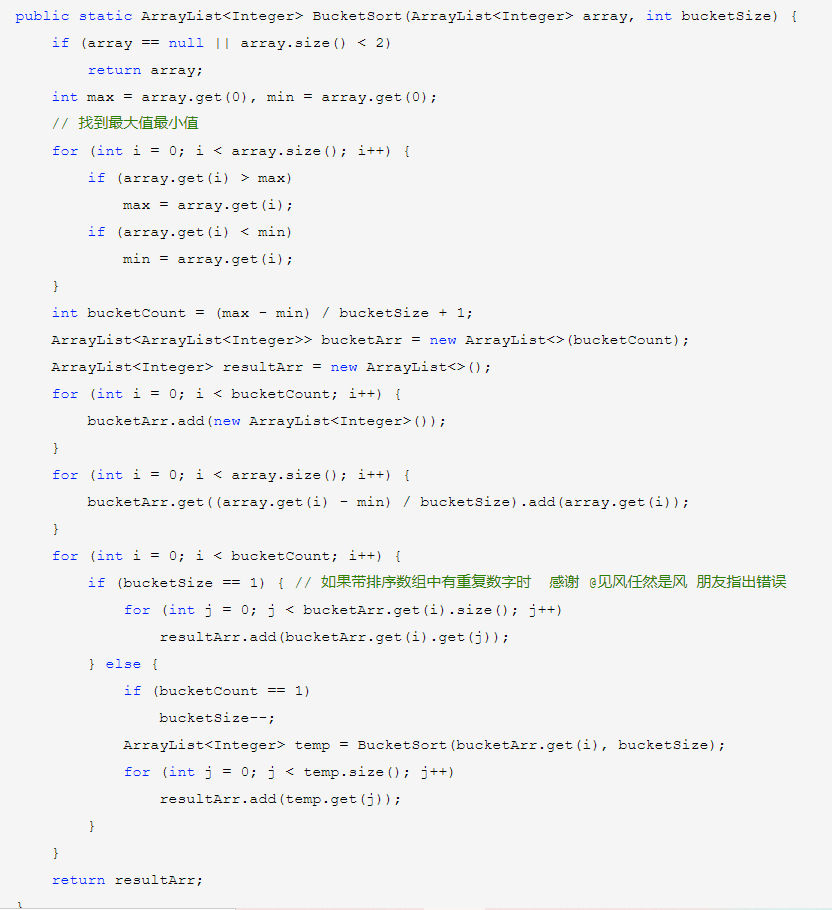
number是array里的element，N是bucket的数量，M=99？

list里有(71,22,99,7,14)

其中 7\*5/(99+1)=0

14\*5/(99+1)=0

则7和14都放在bucket 0--index0



1. **Counting Sort计数排序**

**时间复杂度都是O(n+k)** 空间复杂度O(k) out-place 稳定 空间换时间

**原理：**它不需要去跟其他元素比来比去，而是一开始就知道自己的位置，所以直接归位，在计数的该元素出现的词频数组里面，出现一次，就直接+1一次即可，如果没有出现改位置就是0，最后该位置的词频，就是代表其在原始数组里面出现的次数，由于词频数组的index是从0开始，所以最后直接遍历输出这个数组里面的每一个大于0的元素值即可。

获取数组最大数，创建大小为最大值+1的数组，元素出现一次就元素值的index+1

最后从index0开始遍历数组，每个位置打印元素值次index就行

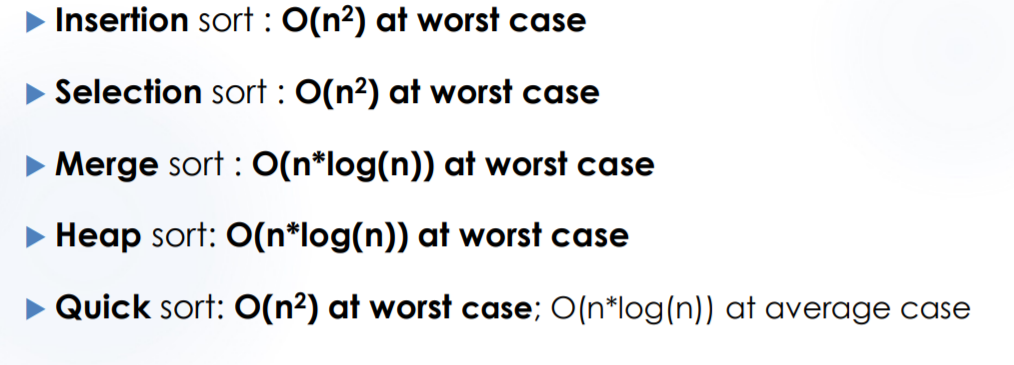


**Comparison of sorting algorithm**

1.Merge sort比Quick sort需要extra memory，因为每次call merge都会create新的array去放

2.selection sort的swap次数最少

3.对于mostly sorted array，insertion sort最好

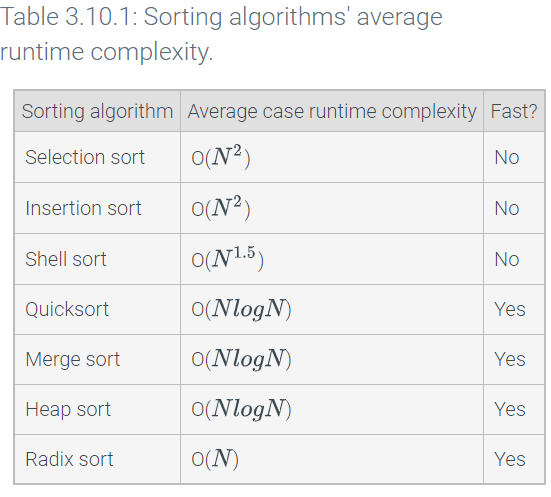


3.只有merge sort不是in place，因为要extra memory

4.只有Insertion和merge sort是stable

Overview：

1.fast sorting algorithm是average runtime complexity为O(N\*logN)



fast: Radix, Quick, Merge, Heap