## 选择排序（冒泡）

### 排序原理：

遍历所有的元素，min作为指针找到最小的元素的下标，结束后，替换第i个元素和最小的元素交换位置,但是arr[max]的时候就不需要比较了。

### 特点：

不需要额外的内存空间，交换的次数是所有的排序算法里面最少的。缺点是比较的次数太多了，每次排序了一个元素之后，第二次查询未用到第一次查询累计的结果。时间复杂度是平方级别的。

//数据源是一个不重复的数组

**public** **void** **sort**(**int**[] arr){

//从第一个数组开始和后面 (n-1）个数据比较，但是当i=最后一个的时候 arr[max]的时候就不需要比较了

**for**(**int** i=0;i<arr.length-1;i++){

**int** min=i;

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

**if**(*less*(arr[j],arr[min])) min=j;

}

*exch*(arr,i,min);

}

}

内存分析： 只多占用一个下标，int min。

比较次数： N(N-1)/2 次 是稳定的

交换次数: N-1 次 也是稳定的

### 示例

171 880 237 549 816 174 785 370 863 104 356 288 975 937 112 246 573 934 538 422

排序后数组

104 112 171 174 237 246 288 356 370 422 538 549 573 785 816 863 880 934 937 975

count number is: 190

exchange number is: 19

### 性能分析

//根据size 和 sort类型，system.out 耗时,每次都测试1000次，取1000次的sum耗时

**public** **static** **void** **outputCostByGrowSize**(**int** initSize,**int** grow,SortInterface sort){

**int** size=initSize;

//最大size 10000个

**while**(size<=(MAX\_SIZE/20)){

System.out.println("this time the arrSize is: "+size);

**long** cost=*testAveCost*(sort,size,1000);

size+=grow;

}

}

**public** **static** **void** **main**(String[] argu){

Selection exa=**new** Selection();

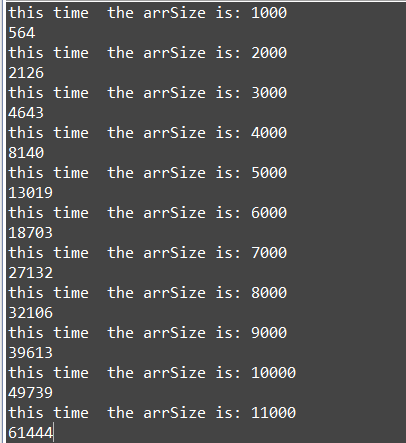
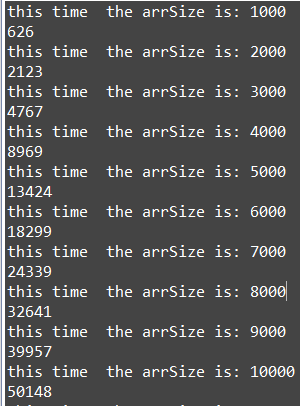
*outputCostByGrowSize*(1000,1000,exa);

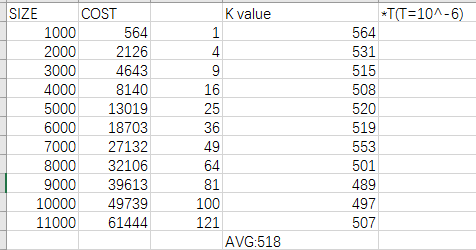
}

### 验证性能满足公式 K\*(N^2)，计算K值，K值在每次重复测试1000次的时候，近似等于5.18\*10^-4

cost= 5.18\*10^-4 \* (size)^2

### 测试耗时结果（2次结果相近）



## 插入排序

### 时间复杂度O(N^2) 平方级 最差； 最好的情况 O(N)；

### 排序原理：

桥牌打发，从二个元素开始，维护一个递增的数组，如果某元素小于 最大值，就和最大值互换位置，与次大值比较

### 特点：

已经有序或者基本有序的数组（数组中只有几个元素的位置不正确，每个元素距离它的最终位置都不远）

或者是数组的数量较少时，一般JDK1.7 认定数量小于47的时候默认就用插入排序了。

### 代码

//数据源是一个不重复的数组

//排序规则：桥牌打发，从二个元素开始，维护一个递增的数组，如果某元素小于 最大值，就和最大值互换位置，与次大值比较

**public** **static** **void** **sort**(**int**[] arr){

*count*=0;

*exchCount*=0;

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

**int** j=i;

**while**(j>0&&*less*(arr[j],arr[j-1])){

*exch*(arr,j,j-1);

j--;

}

}

}

### JDK1.7的插入排序实现

**private** **static** **void** **sort**(**int**[] a, **int** left, **int** right, **boolean** leftmost) {

**int** length = right - left + 1;

// Use insertion sort on tiny arrays

**if** (length < INSERTION\_SORT\_THRESHOLD) {

**if** (leftmost) {

/\*

\* Traditional (without sentinel) insertion sort,

\* optimized for server VM, is used in case of

\* the leftmost part.

\*/

**for** (**int** i = left, j = i; i < right; j = ++i) {

**int** ai = a[i + 1];

**while** (ai < a[j]) {

a[j + 1] = a[j];

**if** (j-- == left) {

**break**;

}

}

a[j + 1] = ai;

}

内存使用：不需要额外的内存

### 交换次数： 约等于 N(N-1)/4 次，和比较次数非常近似

最差的情况: N\*(N-1)/2

最好的情况: 0次 已经有序

### 比较次数：约等于 N(N-1)/4 次，和交换次数非常近似

1)最差的情况（逆序数组）: N\*(N-1)/2

在第一趟排序中，插入排序最多比较一次，第二趟最多比较两次，依次类推，最后一趟最多比较N-1次。因此有：1+2+3+...+N-1 =N\*(N-1)/2

2）最好的情况（已经有序）： N-1 次

### 示例

原始数组

578 442 338 884 77 891 620 492 257 968 974 188 490 871 93 484 901 124 208 757

排序后数组

77 93 124 188 208 257 338 442 484 490 492 578 620 757 871 884 891 901 968 974

count number is: 116

exchange number is: 100

check isOrder result: true

Best count number is: 19

Best exchange number is: 0

check isOrder result: true

974 968 901 891 884 871 757 620 578 492 490 484 442 338 257 208 188 124 93 77

77 93 124 188 208 257 338 442 484 490 492 578 620 757 871 884 891 901 968 974

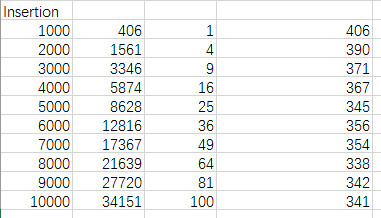
count number is: 190

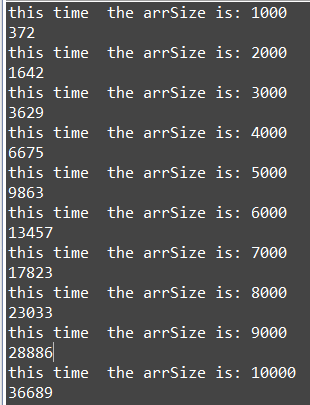
exchange number is: 190

check isOrder result: true

### 性能分析

算法同选择排序，也是重复1000次，1000位初始size，1000递增，算出在不重复随机数组中，耗时也满足 平方规律，K 大约 3.61\* 10^-4





## 希尔排序

### 时间复杂度 O(N^X) 1<X<2 快的话，4/3 5/4 6/5

小小的一个改变，速度提升了将近30倍，快成N 直线了，

//排序规则：基于插入排序的加强版，先大跨度的做插入排序（先13调，调完再按4来调），基本接近最后位置，再按照1的一次完整插入排序

//特点1）任意相隔 H 的元素都是有序的 2）由许许多个独立的有序的小数组组成

@Override

**public** **void** **sort**(**int**[] arr){

*count*=0;

*exchCount*=0;

**int** size=arr.length;

**int** h=1;

**while**(h<size/3) h=h\*3+1; // 1 4 13 40 grow with size ,多乘一次，就多一次跨度的希尔排序

**while**(h>=1){

**for**(**int** i=h;i<size;i++){

**for**(**int** j=i;j>=h&&*less*(arr[j],arr[j-h]);j-=h){

*exch*(arr,j,j-h);

}

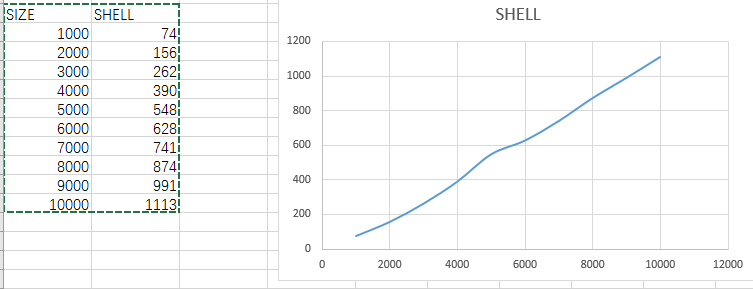
}

h=h/3;

}

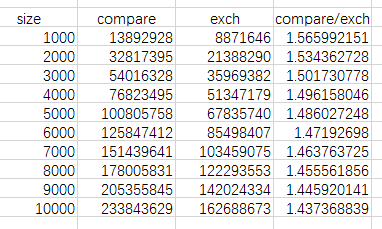
}

### 性能分析



### 比较和交换次数：

根据实现方式不同而不同，类似于 性能函数，交换次数小于比较次数，数组越大，需要交换的次数增长更快。



### 备注：

网上的版本不一样，有的是按照size/2 的递减，不是按照 h=h\*3+1 这种递增。

## 算法比较

### 总则：比较算法的优劣一定要结合具体的情况，不能说选择排序就是垃圾，因为他交换的次数相对来说是最少的，当移动的成本很高 而查询的成本很低的情况下，选择排序就是最好的排序方法；又或者数组的数量不大的情况下，其实比较算法的优劣根本没有意义；又或者内存非常充裕的时候，最佳的算法又不同了；又或者数组已经是一个接近有序的数组。

## 快速排序

### 排序逻辑：

获取一个随机且不重复的数组（如果不随机的话，需要随机打乱）；每次把首位元素作为哨兵，通过遍历，找到该哨兵在这个数组中的准确位置，交换位置后；把数组拦截成3部分: 所有的比哨兵小的数组、哨兵、所有的比哨兵大的数组；然后递归，直至全部有序。

// 5 1 3 8 4 6 9 2 11 --> 1 3 8 i=2 arr[2]=8

// 5 1 3 8 4 6 9 2 11 -->2 11 j=7 arr[7]=2

// 5 1 3 2 4 6 9 8 11 --> 8 和 2 交换位置

### 代码:

@Override

**public** **void** **sort**(**int**[] arr){

sortPart(arr,0,arr.length-1);

}

//递归按照切分段 给数组排序

**public** **void** **sortPart**(**int**[] arr,**int** lo,**int** hi){

**if**(lo>=hi) **return**;

**int** index=*partition*(arr,lo,hi);

// 注意这里把数组分成3部分， 是 index-1 ； index；index+1 三部分

sortPart(arr,lo,index-1);

sortPart(arr,index+1,hi);

}

// 功能： 1）移动数组 2）返回切分后的切分下标

**private** **static** **int** **partition**(**int**[] arr,**int** lo,**int** hi){

**int** i=lo,j=hi+1;

**int** v=arr[lo];

**while**(**true**){

// 5 1 3 8 4 6 9 2 11 --> 1 3 8 i=2 arr[2]=8

//标杆值 从初始位置挨个和后续元素比较，直到后续元素比标杆元素大，或者是后续元素到了最后的位置

//下标i 移动，直到找到比标杆元素大的下标为止，获取到i的值，要交换

**while**(*less*(arr[++i],v)){

**if**(i>=hi) **break**;

}

// 5 1 3 8 4 6 9 2 11 -->2 11 j=7 arr[7]=2

**while**(*less*(v,arr[--j])){

**if**(j<=lo) **break**;

}

**if**(i>=j) **break**;

//交换之后 因为i 和 j的值是在外面申明过的，所以不会重复比较之前比较过的对象 从 arr[i] -- arr[j] 区间继续比较

// 5 1 3 2 4 6 9 8 11 --> 8 和 2 交换位置

*exch*(arr,i,j);

}

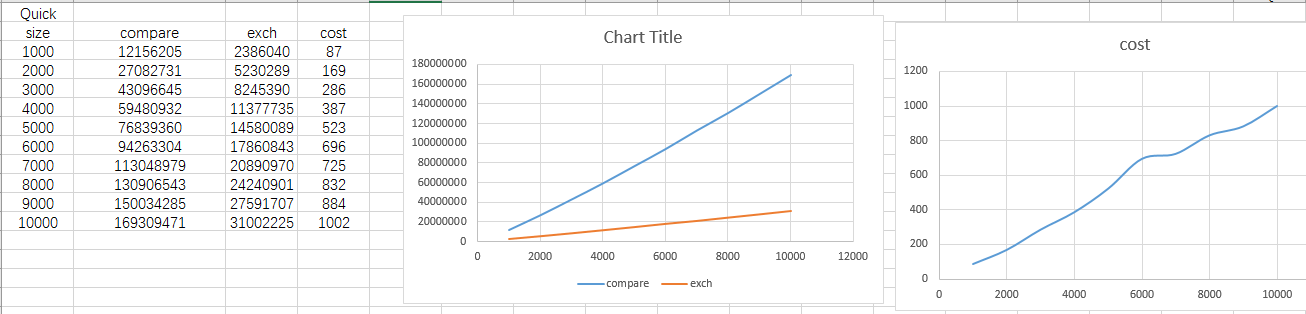
//一定要记得 最后一次要把标杆值 和新的标杆值交换位置啊

*exch*(arr,lo,j);

**return** j;

}

### 性能分析：



### 排序算法的改进版

**public** **void** **sortPartAdvance**(**int**[] arr,**int** lo,**int** hi,**boolean** isCount,**int** argu){

//当数组为10个以内大小的时候，改用插入排序，会更快

argu=10;

**if**(lo+argu>=hi) {

Insertion exa=**new** Insertion();

**int**[] shortArr=**new** **int**[argu];

System.*arraycopy*(arr,lo,shortArr,0,argu);

exa.sort(shortArr);

**return**;

}

**int** index=0;

**if**(isCount){

index=*partitionCount*(arr,lo,hi);

}**else**{

index=*partition*(arr,lo,hi);

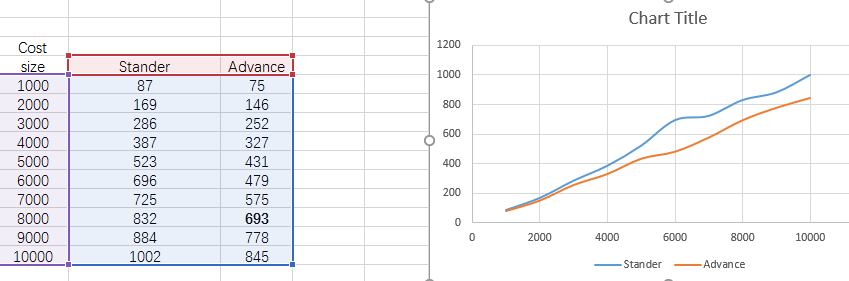
}

sortPart(arr,lo,index-1,isCount);

sortPart(arr,index+1,hi,isCount);

}

性能提升



### 允许有重复元素的快速排序

//该方法和 标准的快速排序其实不同，标准的是 左右开弓比较，而下面的比较是仅仅和下一个元素比较，比哨兵小的统统扔到数组的后面去

**public** **void** **sortRepeatArr**(**int**[] arr,**int** lo,**int** hi){

**if**(lo>=hi) **return**;

//lt 是哨兵位置，目标是整理一个 数组，找到哨兵的准确位置，哨兵左边的元素都比哨兵小，哨兵右边的元素都比哨兵大

**int** lt=lo;

//比较好的最后一个末位下标， 表示gt前的元素 要么没比较过，要么比哨兵小，gt 后面的所有元素都比哨兵大

//区分开来，gt不是hi，hi是末位元素，而gt是比较后结果位置

**int** gt=hi;

//i是和哨兵比较的元素下标

**int** i=lo+1;

//哨兵值

**int** v=arr[lt];

**while**(i<=gt){

//如果比较元素比 哨兵小 ，则交换 比较元素和哨兵元素的位置

//同时比较后一个元素 i ++

//同时哨兵的位置往后调整 1，因为交换了元素，所以哨兵的下标也发生了 变化，但是有2种情况，

// 1 2 5 4 6 3 4 8 9 假设当前哨兵是 5. 和4比较， 4比5小 ，交换位置，

// 1 2 |4 5| 6 3 4 8 9 此时哨兵元素还是5，交换前 哨兵下边是 arr[2]，交换后 哨兵下标 lt++； 值就是3了，同时比较元素下边i 也要变大

//第二种情况有多个相同的哨兵的话， lt ++ 就是让下一个哨兵 顶上

// 1 2 5 5 4 6 3 4 8 9 当前哨兵下标是arr[2] lt=2 比较元素i=4 ，交换之后

// 1 2 4 5 5 6 3 4 8 9 交换之后 哨兵位置是3 而老的哨兵 在 arr[4]的 位置，最新的哨兵 之前的arr[3]

**if**(compare(arr[i],v)<0){

*exch*(arr,lt++,i++);

}**else** **if**(compare(arr[i],v)>0){

//如果比较元素比哨兵大的话， 则用gt 来维护一群比哨兵大的元素，和gt互换位置，且gt下标左移一次

// 1 2 5 5 6 7 47 8 3 假设当前哨兵是5 lt=2 arr[2]=5, 当前比较元素 arr[4]=6, 当前gt 为hi

// 1 2 5 5 3 7 47 8 6 <--交换之后的结果 gt -- ，所以gt就从末位下标 arr[8] 变成7了

*exch*(arr,gt--,i);

}**else**{

i++;

}

}

sortRepeatArr(arr,lo,lt-1);

sortRepeatArr(arr,gt+1,hi);}

## 归并排序

### 排序逻辑：把一个数组两两切分，直到变成只有1个元素为止，再把每个有序的数组（只有1个元素肯定是有序的），两两归并，最后的数组就是一个有序的数组了。

### 特点分析

1. 每次都要new 一个长度一样的 aux[] 数组浪费内存
2. 性能不一定高才是关键
3. 不需要交换元素，直接重新赋值即可，从辅助数组中取数，赋值到原数组中

//数据源是一个不重复的数组

@Override

**public** **void** **sort**(**int**[] arr){

sort(arr,0,arr.length-1);

}

//递归按照切分段 给数组排序

**public** **void** **sort**(**int**[] arr,**int** lo,**int** hi){

**if**(lo>=hi) **return**;

**int** mid=lo+(hi-lo)/2;

sort(arr,lo,mid);

sort(arr,mid+1,hi);

merge(arr,lo,mid,hi);

}

// arr数组 arr[0] -- arr[mid] 是有序的数组； arr[mid+1] -- arr[hi] 也是有序的数组，当mid=0的时候，就只有一个元素的时候肯定就是有序的数组了

**public** **void** **merge**(**int**[] arr, **int** lo,**int** mid,**int** hi){

//i 表示 arr1 的轮询到的下标位置， lo<=i<=mid ； i++

**int** i=lo;

//j 表示 arr2 的轮询到的下标位置，mid+1<=j<=hi ； j++

**int** j=mid+1;

//申明一个数组作为辅助数组，拷贝出

**int**[] aux =**new** **int**[arr.length];

**for**(**int** k=lo;k<=hi;k++){

aux[k]=arr[k];

}

**for**(**int** k=lo;k<=hi;k++){

//当第一个数组元素用尽的时候，后续就不需要调整了，因为后面的数组本来就是有序的

**if**(i>mid) **return**;

//当第二个数组元素用尽的时候，就需要把前面的i之后的元素，填充到后面当中

**else** **if**(j>hi) arr[k]=aux[i++];

//记住这里是 用不变数值的 aux数组进行比较啊，千万不能用 arr 原数组比较啊，坑爹

**else** **if**(*less*(aux[j],aux[i])) arr[k]=aux[j++];

**else** arr[k]=aux[i++];

}

}

### 加强版的归并排序

**public** **void** **sortAdvance**(**int**[] arr,**int** lo,**int** hi){

**if**(lo+10>=hi) {

**new** Insertion().sortParty(arr,lo,hi);

**return**;

}

**int** mid=lo+(hi-lo)/2;

sortAdvance(arr,lo,mid);

sortAdvance(arr,mid+1,hi);

merge(arr,lo,mid,hi);

}

Insertion

**public** **void** **sortParty**(**int**[] arr,**int** startIndex,**int** endIndex){

**int** size=endIndex-startIndex+1;

//注意插入排序的i 是小于startIndex+size

**for**(**int** i=startIndex+1;i<startIndex+size;i++){

**int** j=i;

**while**(j>startIndex&&*less*(arr[j],arr[j-1])){

*exch*(arr,j,j-1);

j--;

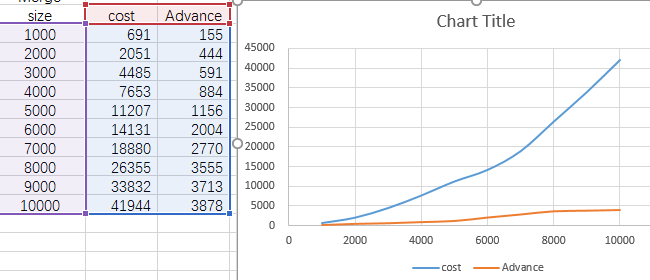
}

}

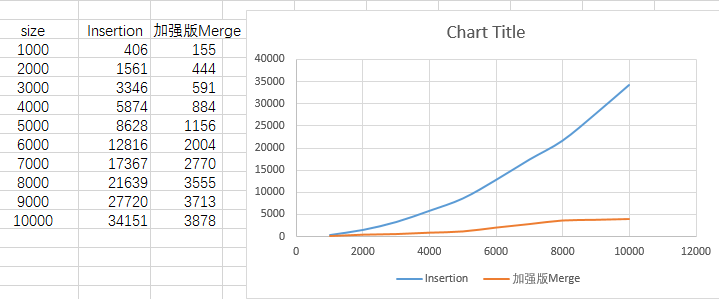
}

### 性能分析

加强版的 归并排序比 标准的归并排序快了 十多倍啊， 因为归并排序小的时候非常多，几乎每个元素都要从1个数组起步的。



加强版的归并排序和插入排序，远远的甩开了 插入排序，但是内存的开销变得很大了



## 堆排序

### 排序原理：基于优先队列的实现

1. 先把数组变成一个 首位是null， 后续挨个的数组，此时为无序堆
2. 从N/2的位置开始，挨个sink排序，使无序堆有序
3. 挨个获取到堆顶元素 ，获取最大值，交换到数组末位，然后缩小长度，轮询操作

### 代码

//不同于 MaxMQ，已知一个无序的堆，进行排序成 数组有序

**public** **void** **sort**(List<Integer> list){

**int** N=list.size()-1;

//先把数组构建成堆结构

**for**(**int** k=N/2;k>=1;k--){

sink(k,N);

}

**int** copySize=N;

//需要重写 或者扩展 sink 方法，要让 sink 是根据 size大小进行下沉

//这里递减的要用 copySize啊，因为要循环N 次，如果直接用N--的话， k每增加一次 ，N就减少1次，只能做到一半的数据排序了

**for**(**int** k=1;k<=N;k++){

exch(1,copySize);

sink(1,--copySize);

}

}

## 优先队列

解决的问题：利用堆有序，他能在插入操作和删除最大元素操作混合的动态常见中保证对数级别的运行时间。

**public** **void** **swim**(**int** index){

**while**(index>1&&less(list.get(index/2),list.get(index))){

exch(index/2,index);

index=index/2;

}

}

//递归的三个条件

//1)如果有子节点的话，就继续比较，可以是1个子节点，也可以是2个子节点

//2)如果有2个子节点且第二个子节点更大的话，最大子节点的下标就要++ ，也就是取第二个节点

//3)判断 父节点 和最大的子节点的大小

**public** **void** **sink**(**int** index){

//如果有子节点的话，就继续比较，可以是1个子节点，也可以是2个子节点

**while**(2\*index<=size()){

**int** childmax=2\*index;

//如果有2个子节点且第二个子节点更大的话，最大子节点的下标就要++ ，也就是取第二个节点

**if**(childmax+1<=size()&&less(list.get(childmax),list.get(childmax+1))){

childmax++;

}

**if**(less(list.get(childmax),list.get(index))) **break**;

exch(childmax,index);

index=childmax;

}

}

## 选择排序 插入排序 希尔排序比较

次数分析

选择排序：

比较次数： N(N-1)/2 次 是稳定的 ;交换次数: N-1 次 也是稳定的

插入排序：

比较次数： N(N-1)/4 次 近似 ;比较次数： N(N-1)/4 次 近似

Shell排序:

比较次数： Sn= K\*(N^X) 1<X<2 次 ;比较次数： 0.68\*Sn

性能分析：

希尔排序碾压 插入排序，在随机不重复的数组的排序中。

区别：对于随机不重复的1000大小的数组，选择排序比插入排序比较次数多，但是交换次数少，试验1000次结果，插入排序 是选择排序的50%。左右。

在JDK 1.7版本中是这样的。比例大约为1.9。

1000个随机不重复的数组中，插入排序比选择排序性能快50%。

long costAveCostByCount=testAveCost(exa,1000,1000);

System.out.println("insertion cost:1000 size,1000times: "+costAveCostByCount);

395

insertion cost:1000 size,1000times: 395

769

selection cost:1000 size,1000times: 769

从下面的图来看，随机不重复数组的 插入排序要优于 选择排序

