**Java排序算法**

**1）分类：**

1）插入排序（直接插入排序、希尔排序）

2）交换排序（冒泡排序、快速排序）

3）选择排序（直接选择排序、堆排序）

4）归并排序

5）分配排序（箱排序、基数排序）

所需辅助空间最多：归并排序

所需辅助空间最少：堆排序

平均速度最快：快速排序

不稳定：快速排序，希尔排序，堆排序。

**1）选择排序算法的时候**

1.数据的规模 ； 2.数据的类型 ； 3.数据已有的顺序   
一般来说，当数据规模较小时，应选择直接插入排序或冒泡排序。任何排序算法在数据量小时基本体现不出来差距。考虑数据的类型，比如如果全部是正整数，那么考虑使用桶排序为最优。 考虑数据已有顺序，快排是一种不稳定的排序（当然可以改进），对于大部分排好的数据，快排会浪费大量不必要的步骤。数据量极小，而起已经基本排好序，冒泡是最佳选择。我们说快排好，是指大量随机数据下，快排效果最理想。而不是所有情况。

3）总结：

――按平均的时间性能来分：

1）时间复杂度为O(nlogn)的方法有：快速排序、堆排序和归并排序，其中以快速排序为最好；

2）时间复杂度为O(n2)的有：直接插入排序、起泡排序和简单选择排序，其中以直接插入为最好，特 别是对那些对关键字近似有序的记录序列尤为如此；

3）时间复杂度为O(n)的排序方法只有，基数排序。

当待排记录序列按关键字顺序有序时，直接插入排序和起泡排序能达到O(n)的时间复杂度;而对于快速排序而言，这是最不好的情况，此时的时间性能蜕化为O(n2)，因此是应该尽量避免的情况。简单选择排序、堆排序和归并排序的时间性能不随记录序列中关键字的分布而改变。

――按平均的空间性能来分（指的是排序过程中所需的辅助空间大小）：

1） 所有的简单排序方法(包括：直接插入、起泡和简单选择)和堆排序的空间复杂度为O(1)；

2） 快速排序为O(logn )，为栈所需的辅助空间;

3） 归并排序所需辅助空间最多，其空间复杂度为O(n );

4）链式基数排序需附设队列首尾指针，则空间复杂度为O(rd )。

――排序方法的稳定性能：

1） 稳定的排序方法指的是，对于两个关键字相等的记录，它们在序列中的相对位置，在排序之前和 经过排序之后，没有改变。

2） 当对多关键字的记录序列进行LSD方法排序时，必须采用稳定的排序方法。

3） 对于不稳定的排序方法，只要能举出一个实例说明即可。

4） 快速排序，希尔排序和堆排序是不稳定的排序方法。  
4）插入排序：

包括直接插入排序，希尔插入排序。

直接插入排序： 将一个记录插入到已经排序好的有序表中。

1, sorted数组的第0个位置没有放数据。

2，从sorted第二个数据开始处理：

如果该数据比它前面的数据要小，说明该数据要往前面移动。

首先将该数据备份放到 sorted的第0位置当哨兵。

然后将该数据前面那个数据后移。

然后往前搜索，找插入位置。

找到插入位置之后讲 第0位置的那个数据插入对应位置。

O(n\*n), 当待排记录序列为正序时，时间复杂度提高至O(n)。

希尔排序（缩小增量排序 diminishing increment sort）：先将整个待排记录序列分割成若干个子序列分别进行直接插入排序，待整个序列中的记录基本有序时，再对全体记录进行一次直接插入排序。

**插入排序Java代码：**

**public** **class** InsertionSort {

// 插入排序：直接插入排序，希尔排序

**public** **void** straightInsertionSort(**double** [] sorted){

**int** sortedLen= sorted.length;

**for**(**int** j=2;j<sortedLen;j++){

**if**(sorted[j]<sorted[j-1]){

sorted[0]= sorted[j];//先保存一下后面的那个

sorted[j]=sorted[j-1];// 前面的那个后移。

**int** insertPos=0;

**for**(**int** k=j-2;k>=0;k--){

**if**(sorted[k]>sorted[0]){

sorted[k+1]=sorted[k];

}**else**{

insertPos=k+1;

**break**;

}

}

sorted[insertPos]=sorted[0];

}

}

}

**public** **void** shellInertionSort(**double** [] sorted, **int** inc){

**int** sortedLen= sorted.length;

**for**(**int** j=inc+1;j<sortedLen;j++ ){

**if**(sorted[j]<sorted[j-inc]){

sorted[0]= sorted[j];//先保存一下后面的那个

**int** insertPos=j;

**for**(**int** k=j-inc;k>=0;k-=inc){

**if**(sorted[k]>sorted[0]){

sorted[k+inc]=sorted[k];

//数据结构课本上这个地方没有给出判读，出错：

**if**(k-inc<=0){

insertPos = k;

}

}**else**{

insertPos=k+inc;

**break**;

}

}

sorted[insertPos]=sorted[0];

}

}

}

**public** **void** shellInsertionSort(**double** [] sorted){

**int**[] incs={7,5,3,1};

**int** num= incs.length;

**int** inc=0;

**for**(**int** j=0;j<num;j++){

inc= incs[j];

shellInertionSort(sorted,inc);

}

}

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

Random random= **new** Random(6);

**int** arraysize= 21;

**double** [] sorted=**new** **double**[arraysize];

System.*out*.print("Before Sort:");

**for**(**int** j=1;j<arraysize;j++){

sorted[j]= (**int**)(random.nextDouble()\* 100);

System.*out*.print((**int**)sorted[j]+" ");

}

System.*out*.println();

InsertionSort sorter=**new** InsertionSort();

// sorter.straightInsertionSort(sorted);

sorter.shellInsertionSort(sorted);

System.*out*.print("After Sort:");

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

System.*out*.print((**int**)sorted[j]+" ");

}

System.*out*.println();

}

}

**5）交换排序：**

**包括冒泡排序，快速排序。**

**冒泡排序法：**该算法是**专门针对已部分排序的数据**进行排序的一种排序算法。如果在你的数据清单中只有一两个数据是乱序的话，用这种算法就是最快的排序算法。如果你的数据清单中的数据是随机排列的，那么这种方法就成了最慢的算法了。因此在使用这种算法之前一定要慎重。这种算法的核心思想是扫描数据清单，寻找出现乱序的两个相邻的项目。当找到这两个项目后，交换项目的位置然后继续扫描。重复上面的操作直到所有的项目都按顺序排好。

**快速排序：**通过一趟排序，将待排序记录分割成独立的两个部分，其中一部分记录的关键字均比另一部分记录的关键字小，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。**具体做法是：**使用两个指针low,high, 初值分别设置为序列的头，和序列的尾，设置pivotkey为第一个记录，首先从high开始向前搜索第一个小于pivotkey的记录和pivotkey所在位置进行交换，然后从low开始向后搜索第一个大于pivotkey的记录和此时pivotkey所在位置进行交换，重复知道low=high了为止。

**交换排序Java代码：**

**public** **class** ExchangeSort {

**public** **void** BubbleExchangeSort(**double** [] sorted){

**int** sortedLen= sorted.length;

**for**(**int** j=sortedLen;j>0;j--){

**int** end= j;

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

**double** tempB= sorted[k];

sorted[k]= sorted[k]<sorted[k+1]?

sorted[k]:sorted[k+1];

**if**(Math.*abs*(sorted[k]-tempB)>10e-6){

sorted[k+1]=tempB;

}

}

}

}

**public** **void** QuickExchangeSortBackTrack(**double** [] sorted,

**int** low,**int** high){

**if**(low<high){

**int** pivot= findPivot(sorted,low,high);

QuickExchangeSortBackTrack(sorted,low,pivot-1);

QuickExchangeSortBackTrack(sorted,pivot+1,high);

}

}

**public** **int** findPivot(**double** [] sorted, **int** low, **int** high){

sorted[0]= sorted[low];

**while**(low<high){

**while**(low<high && sorted[high]>= sorted[0])--high;

sorted[low]= sorted[high];

**while**(low<high && sorted[low]<=sorted[0])++low;

sorted[high]= sorted[low];

}

sorted[low]=sorted[0];

**return** low;

}

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

Random random= **new** Random(6);

**int** arraysize= 21;

**double** [] sorted=**new** **double**[arraysize];

System.*out*.print("Before Sort:");

**for**(**int** j=1;j<arraysize;j++){

sorted[j]= (**int**)(random.nextDouble()\* 100);

System.*out*.print((**int**)sorted[j]+" ");

}

System.*out*.println();

ExchangeSort sorter=**new** ExchangeSort();

// sorter.BubbleExchangeSort(sorted);

sorter.QuickExchangeSortBackTrack(sorted, 1, arraysize-1);

System.*out*.print("After Sort:");

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

System.*out*.print((**int**)sorted[j]+" ");

}

System.*out*.println();

}

}  
**6）选择排序：**

**分为直接选择排序，堆排序**

**直接选择排序：**第i次选取 i到array.Length-1中间最小的值放在i位置。

**堆排序：**首先，数组里面用层次遍历的顺序放一棵完全二叉树。从最后一个非终端结点往前面调整，直到到达根结点，这个时候除根节点以外的所有非终端节点都已经满足堆得条件了，于是需要调整根节点使得整个树满足堆得条件，于是从根节点开始，沿着它的儿子们往下面走（最大堆沿着最大的儿子走，最小堆沿着最小的儿子走）。主程序里面，首先从最后一个非终端节点开始调整到根也调整完，形成一个heap， 然后将heap的根放到后面去（即：每次的树大小会变化，但是 root都是在1的位置，以方便计算儿子们的index，所以如果需要升序排列，则要逐步大顶堆。因为根节点被一个个放在后面去了。降序排列则要建立小顶堆）

**代码中的问题：有时候第2个和第3个顺序不对（原因还没搞明白到底代码哪里有错）**

**选择排序Java代码：**

**public** **class** SelectionSort {

**public** **void** straitSelectionSort(**double** [] sorted){

**int** sortedLen= sorted.length;

**for**(**int** j=1;j<sortedLen;j++){

**int** jMin= getMinIndex(sorted,j);

exchange(sorted,j,jMin);

}

}

**public** **void** exchange(**double** [] sorted,**int** i,**int** j){

**int** sortedLen= sorted.length;

**if**(i<sortedLen && j<sortedLen && i<j && i>=0 && j>=0){

**double** temp= sorted[i];

sorted[i]=sorted[j];

sorted[j]=temp;

}

}

**public** **int** getMinIndex(**double** [] sorted, **int** i){

**int** sortedLen= sorted.length;

**int** minJ=1;

**double** min= Double.*MAX\_VALUE*;

**for**(**int** j=i;j<sortedLen;j++){

**if**(sorted[j]<min){

min= sorted[j];

minJ= j;

}

}

**return** minJ;

}

**public** **void** heapAdjust(**double** [] sorted,**int** start,**int** end){

**if**(start<end){

**double** temp= sorted[start];

// 这个地方j<end与课本不同，j<=end会报错：

**for**(**int** j=2\*start;j<end;j \*=2){

**if**(j+1<end && sorted[j]-sorted[j+1]>10e-6){

++j;

}

**if**(temp<=sorted[j]){

**break**;

}

sorted[start]=sorted[j];

start=j;

}

sorted[start]=temp;

}

}

**public** **void** heapSelectionSort(**double** [] sorted){

**int** sortedLen = sorted.length;

**for**(**int** i=sortedLen/2;i>0;i--){

heapAdjust(sorted,i,sortedLen);

}

**for**(**int** i=sortedLen;i>1;--i){

exchange(sorted,1,i);

heapAdjust(sorted,1,i-1);

}

}

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

Random random= **new** Random(6);

**int** arraysize=9;

**double** [] sorted=**new** **double**[arraysize];

System.*out*.print("Before Sort:");

**for**(**int** j=1;j<arraysize;j++){

sorted[j]= (**int**)(random.nextDouble()\* 100);

System.*out*.print((**int**)sorted[j]+" ");

}

System.*out*.println();

SelectionSort sorter=**new** SelectionSort();

// sorter.straitSelectionSort(sorted);

sorter.heapSelectionSort(sorted);

System.*out*.print("After Sort:");

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

System.*out*.print((**int**)sorted[j]+" ");

}

System.*out*.println();

}

}

**7）归并排序：**

将两个或两个以上的有序表组合成一个新的有序表。归并排序要使用一个辅助数组，大小跟原数组相同，递归做法。每次将目标序列分解成两个序列，分别排序两个子序列之后，再将两个排序好的子序列merge到一起。

**归并排序Java代码：**

**public** **class** MergeSort {

**private** **double**[] bridge;//辅助数组

**public** **void** sort(**double**[] obj){

**if** (obj == **null**){

**throw** **new** NullPointerException("

The param can not be null!");

}

bridge = **new** **double**[obj.length]; // 初始化中间数组

mergeSort(obj, 0, obj.length - 1); // 归并排序

bridge = **null**;

}

**private** **void** mergeSort(**double**[] obj, **int** left, **int** right){

**if** (left < right){

**int** center = (left + right) / 2;

mergeSort(obj, left, center);

mergeSort(obj, center + 1, right);

merge(obj, left, center, right);

}

}

**private** **void** merge(**double**[] obj, **int** left,

**int** center, **int** right){

**int** mid = center + 1;

**int** third = left;

**int** tmp = left;

**while** (left <= center && mid <= right){

// 从两个数组中取出小的放入中间数组

**if** (obj[left]-obj[mid]<=10e-6){

bridge[third++] = obj[left++];

} **else**{

bridge[third++] = obj[mid++];

}

}

// 剩余部分依次置入中间数组

**while** (mid <= right){

bridge[third++] = obj[mid++];

}

**while** (left <= center){

bridge[third++] = obj[left++];

}

// 将中间数组的内容拷贝回原数组

copy(obj, tmp, right);

}

**private** **void** copy(**double**[] obj, **int** left, **int** right)

{

**while** (left <= right){

obj[left] = bridge[left];

left++;

}

}

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

Random random = **new** Random(6);

**int** arraysize = 10;

**double**[] sorted = **new** **double**[arraysize];

System.*out*.print("Before Sort:");

**for** (**int** j = 0; j < arraysize; j++) {

sorted[j] = (**int**) (random.nextDouble() \* 100);

System.*out*.print((**int**) sorted[j] + " ");

}

System.*out*.println();

MergeSort sorter = **new** MergeSort();

sorter.sort(sorted);

System.*out*.print("After Sort:");

**for** (**int** j = 0; j < sorted.length; j++) {

System.*out*.print((**int**) sorted[j] + " ");

}

System.*out*.println();

}

}

**8）基数排序：**

使用10个辅助队列，假设最大数的数字位数为 x， 则一共做 x次，从个位数开始往前，以第i位数字的大小为依据，将数据放进辅助队列，搞定之后回收。下次再以高一位开始的数字位为依据。

**以Vector作辅助队列，基数排序的Java代码：**

**public** **class** RadixSort {

**private** **int** keyNum=-1;

**private** Vector<Vector<Double>> util;

**public** **void** distribute(**double** [] sorted, **int** nth){

**if**(nth<=keyNum && nth>0){

util=**new** Vector<Vector<Double>>();

**for**(**int** j=0;j<10;j++){

Vector <Double> temp= **new** Vector <Double>();

util.add(temp);

}

**for**(**int** j=0;j<sorted.length;j++){

**int** index= getNthDigit(sorted[j],nth);

util.get(index).add(sorted[j]);

}

}

}

**public** **int** getNthDigit(**double** num,**int** nth){

String nn= Integer.*toString*((**int**)num);

**int** len= nn.length();

**if**(len>=nth){

**return** Character.*getNumericValue*(nn.charAt(len-nth));

}**else**{

**return** 0;

}

}

**public** **void** collect(**double** [] sorted){

**int** k=0;

**for**(**int** j=0;j<10;j++){

**int** len= util.get(j).size();

**if**(len>0){

**for**(**int** i=0;i<len;i++){

sorted[k++]= util.get(j).get(i);

}

}

}

util=**null**;

}

**public** **int** getKeyNum(**double** [] sorted){

**double** max= Double.*MIN\_VALUE*;

**for**(**int** j=0;j<sorted.length;j++){

**if**(sorted[j]>max){

max= sorted[j];

}

}

**return** Integer.*toString*((**int**)max).length();

}

**public** **void** radixSort(**double** [] sorted){

**if**(keyNum==-1){

keyNum= getKeyNum(sorted);

}

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

distribute(sorted,i);

collect(sorted);

}

}

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

Random random = **new** Random(6);

**int** arraysize = 21;

**double**[] sorted = **new** **double**[arraysize];

System.*out*.print("Before Sort:");

**for** (**int** j = 0; j < arraysize; j++) {

sorted[j] = (**int**) (random.nextDouble() \* 100);

System.*out*.print((**int**) sorted[j] + " ");

}

System.*out*.println();

RadixSort sorter = **new** RadixSort();

sorter.radixSort(sorted);

System.*out*.print("After Sort:");

**for** (**int** j = 0; j < sorted.length; j++) {

System.*out*.print((**int**) sorted[j] + " ");

}

System.*out*.println();

}

}

**=====》总结：上述Java代码中，基本上用的都是double数组，如果想要应用其他的数组，只需要将double数组改成 Comparable接口数组，凡是实现了Comparable接口的都可以用。而在C++中，是用模板类来解决这个问题。**