**插入排序**

**插入排序重复地将新的元素插入到一个排好序的子线性表中，直到整个线性表排好序。**

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

**将list[i]插入已排好序的子线性表中，这样list[0..i]也是排好的**

**}**

**为了将list[i]插入list[0..i-1]，需要将list[i]存储在一个名为currentElement的临时变量中。如果list[i-1]>currentElement，就将list[i-1]移到list[i]；如果list[i-2]>currentElement,就将list[i-2]移到list[i-1]，以此类推，直到list[i-k]<=currentElement或者k>i（传递的是排好序的数列的第一个元素）。将currentElement赋值给list[i-k+1]。**

**public class InsertionSort{**

**public static void insertionSort(int[] list){**

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

**int currentElement=list[i];**

**int k;**

**for(k=i-1;k>=0&&list[k]>currentElement;k--){**

**list[k+i]=list[k];**

**}**

**list[k+1]=currentElement;**

**}**

**}**

**}**

**外层循环（循环控制变量i）的迭代是为了获取已排好序的子线性表，其范围从list[0]到list[i]。内层循环（循环控制变量k）将list[i]插入从list[0]到list[i-1]的子线性表中。**

**冒泡排序**

**冒泡排序算法多次遍历数组，在每次遍历中连续比较相邻的元素，如果元素没有按照顺序排列，则互换它们的值。**

**第一次遍历之后，最后一个元素称为数组中的最大数。第二次遍历之后，倒数第二个元素称为数组中的第二大数。整个过程持续到所有元素都已排好序。**

**for(int k=1;k<list.length;k++){**

**for(int i=0;i<list.length-k;i++){**

**if(list[i]>list[i+1])**

**swap list[i] with list[i+1];**

**}**

**}**

**注意到如果在某次遍历中没有发生交换，那么就不必进行下一次遍历，因为所有的元素都已经排好序了。**

**public class BubbleSort{**

**public static void bubbleSort(int[] list){**

**boolean needNextPass=true;**

**for(int k=1;k<list.length&&needNextPass;k++){**

**needNextPass=false;**

**for(int i=0;i<list.length-k;i++){**

**if(list[i]>list[i+1]){**

**int temp=list[i];**

**list[i]=list[i+1];**

**list[i+1]=temp;**

**needNextPass=true;**

**}**

**}**

**}**

**}**

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

**int[] list={2,3,2,4,324,-1};**

**bubbleSort(list);**

**}**

**}**

**最佳情况下，冒泡排序的时间为O(n)。**

**最差情况下，冒泡排序的时间为O(n).**

**归并排序**

**归并排序算法将数组分为两半，对每部分递归地应用归并排序。在两部分都排好序后，对它们进行归并。**

**public static void mergeSort(int[] list){**

**if(list.length>1){**

**mergeSort(list[0...list.length/2]);**

**mergeSort(list[list.length/2+1...list.length]);**

**merge list[0..list.length/2] with list[list.length/2+1..list.length];**

**}**

**}**

**public class MergeSort{**

**public static void mergeSort(int[] list){**

**if(list.length>1){**

**int[] firstHalf=new int[list.lenght/2];**

**System.arraycopy(list,0,firstHalf,0,list.length/2);**

**mergeSort(firstHalf);**

**int secondHalfLength=list.length-list.length/2;**

**int[] secondHalf=new int[secondHalfLength];**

**System.arraycopy(list,list.length/2,secondHalf,0,secondHalfLength);**

**mergeSort(secondHalf);**

**merge(firstHalf,secondHalf);**

**}**

**}**

**public static void merge(int[] list1,int[] list2,int[] temp){**

**int current1=0;**

**int current2=0;**

**int current3=0;**

**while(current1<list1.length&&current2<list2.length){**

**if(list1[current1]<list2[current2])**

**temp[current3++]=list1[current1++];**

**else**

**temp[current3++]=list2[current2++];**

**}**

**while(current1<list1.length)**

**temp[current3++]=list1[current1++];**

**while(current2<list2.length)**

**temp[current3++]=list2[current2++];**

**}**

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

**int[] list={2,3,2,5,6,1,2};**

**mergeSort(list);**

**for(int i=0;i<list.length;i++)**

**System.out.print(list[i]+" ");**

**}**

**}**

**方法merge归并两个有序数组list1和list2为一个临时数组temp。**

**current1和current2指向list1和list2中要考虑的当前元素。该方法重复比较list1和list2中的当前元素，并将较小的一个元素移到temp中。如果较小元素在list1中，current1增加1；如果较小元素在list2中，current2增加1.最后，其中一个数组中的所有元素都被移动到temp中。如果list1中仍有未移动的元素，就将它们复制到temp中。如果list2中仍有未移动的元素，就将它们赋值到temp中。**

**快速排序**

**该算法在数组中选择一个称为主元（pivot）的元素，将数组分为两部分，使得第一部分中的所有元素都小于或等于主元，而第二部分中的所有元素都大于主元。对第一部分递归地应用快速排序算法，然后对第二部分递归地应用快速排序算法。**

**public static void quickSort(int[] list){**

**if(list.length>1){**

**select a pivot;**

**partition list into list1 and list2 such that**

**all elements in list1 <=pivot and**

**all elements in list2 >pivot;**

**quickSort(list1);**

**quickSort(list2);**

**}**

**}**

**该算法的每次划分都将主元放在了恰当的位置。主元的选择会影响算法的性能。**

**假定将数组的第一个元素选为主元。**

**public class QuickSort{**

**public static void quickSort(int[] list){**

**quickSort(list,0,list.length-1);**

**}**

**public static void quickSort(int[] list,int first,int last){**

**if(last>first){**

**int pivotIndex=partition(list,first,last);**

**quickSort(list,first,pivotIndex-1);**

**quickSort(list,pivotIndex+1,last);**

**}**

**}**

**public static int partition(int[] list,int first,int last){**

**int pivot=list[first];**

**int low=first+1;**

**int high=last;**

**while(high>low){**

**while(low<=high&&list[low]<=pivot)**

**low++;**

**while(low<=high&&list[high]>pivot)**

**high--;**

**if(high>low){**

**int temp=list[high];**

**list[high]=list[low];**

**list[low]=temp;**

**}**

**}**

**while(high>first && list[high]>=pivot)**

**high--;**

**if(pivot>list[high]){**

**list[first]=list[high];**

**list[high]=pivot;**

**return high;**

**}**

**else**

**return first;**

**}**

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

**int[] list={1,2,33,4,1};**

**quickSort(list);**

**}**

**}**

**方法partitioin使用主元划分数组list[first..last]。将子数组的第一个元素选为主元。在初始情况下，low指向子数组中的第二个元素，而high指向子数组中的最后一个元素。**

**方法在数组中从左侧开始查找第一个大于主元的元素，然后从数组右侧开始查找第一个小于或等于主元的元素，最后交换这两个元素。在while循环中重复相同的查找和交换操作，直到所有元素都查找完为止。**

**如果主元被移动，方法返回将子数组分为两部分的主元的新下标return high；否则，返回主元的原始下标return first**

**快速排序算法最差情况下，需要O(n^2)的时间**

**最佳情况下，需要O(nlogn)的时间**

**归并排序和快速排序都使用了分而治之法。**

**对于归并排序，大量的工作是将两个子线性表进行归并，归并是在子线性表都排好序后进行的。**

**对于快速排序，大量的工作是将线性表划分为两个子线性表，划分是在子线性表排好序前进行的。**

**在最差情况下，归并排序的效率高于快速排序，但是平均情况下，两者的效率相同。归并排序在归并两个数组时需要一个临时数组，而快速排序不需要额外的数组空间。因此，快速排序的空间效率高于归并排序。**

**堆排序**

**堆排序使用的是二叉堆。它首先将所有的元素添加到一个堆上，然后不断移除最大的元素以获得一个排好序的线性表。**

**堆排序使用的二叉堆，它是一棵完全二叉树。二叉树是一种层次体系结构。它可能是空的，也可能包含一个称为根（root）的元素以及称为左子树（left subtree）和右子树（right subtree）的两棵不同的二叉树。**

**堆属性：每个节点大于或等于它的任意一个孩子。**

**如果堆的大小是事先知道的，那么可以将堆存储在一个ArrayList或一个数组中。树根在位置0处，它的两个孩子在1和2处。对于位置i处的结点，它的左结点位置为2i+1处，右结点位置为2i+2处，它的父结点位置为（i-1）/2处。**

**为堆添加一个新结点：首先将它添加到堆的末尾，然后按如下方式重建这棵树：**

**将最后一个结点作为当前结点;**

**while（当前结点大于它的父结点）{**

**将当前结点和它的父结点交换;**

**现在当前结点往上面进了一个层次;**

**}**

**删除根结点：重建算法如下：**

**用最后一个结点替换根结点;**

**让根结点称为当前结点;**

**while（当前结点具有子结点并且当前结点小于它的子结点）{**

**将当前结点和它的较大子结点交换;**

**现在当前结点往下面退了一个层次;**

**}**

**Heap类**

public class Heap<E extends Comparable> {

private java.util.ArrayList<E> list = new java.util.ArrayList<E>();

/\*\* Create a default heap \*/

public Heap() {

}

/\*\* Create a heap from an array of objects \*/

public Heap(E[] objects) {

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

add(objects[i]);

}

/\*\* Add a new object into the heap \*/

public void add(E newObject) {

list.add(newObject); // Append to the heap

int currentIndex = list.size() - 1; // The index of the last node

while (currentIndex > 0) {

int parentIndex = (currentIndex - 1) / 2;

// Swap if the current object is greater than its parent

if (list.get(currentIndex).compareTo(

list.get(parentIndex)) > 0) {

E temp = list.get(currentIndex);

list.set(currentIndex, list.get(parentIndex));

list.set(parentIndex, temp);

}

else

break; // the tree is a heap now

currentIndex = parentIndex;

}

}

/\*\* Remove the root from the heap \*/

public E remove() {

if (list.size() == 0) return null;

E removedObject = list.get(0);

list.set(0, list.get(list.size() - 1));

list.remove(list.size() - 1);

int currentIndex = 0;

while (currentIndex < list.size()) {

int leftChildIndex = 2 \* currentIndex + 1;

int rightChildIndex = 2 \* currentIndex + 2;

// Find the maximum between two children

if (leftChildIndex >= list.size()) break; // The tree is a heap

int maxIndex = leftChildIndex;

if (rightChildIndex < list.size()) {

if (list.get(maxIndex).compareTo(

list.get(rightChildIndex)) < 0) {

maxIndex = rightChildIndex;

}

}

// Swap if the current node is less than the maximum

if (list.get(currentIndex).compareTo(

list.get(maxIndex)) < 0) {

E temp = list.get(maxIndex);

list.set(maxIndex, list.get(currentIndex));

list.set(currentIndex, temp);

currentIndex = maxIndex;

}

else

break; // The tree is a heap

}

return removedObject;

}

/\*\* Get the number of nodes in the tree \*/

public int getSize() {

return list.size();

}

}

**方法add（E newObject）将一个对象追加到树中，如果该对象大于它的父结点，就互换它们。此过程持续到该新对象成为根结点，或者新对象不大于它的父结点。**

**方法remove（）删除并返回根结点。为保持堆的特性，该方法将最后的对象移到根结点处，如果该对象小于它的较大的子结点，就互换它们。此过程持续到最后一个对象成为子结点，或者该对象不小于它的子结点。**

**使用Heap类进行排序**

public class HeapSort {

/\*\* Heap sort method \*/

public static <E extends Comparable> void heapSort(E[] list) {

// Create a Heap of integers

Heap<E> heap = new Heap<E>();

// Add elements to the heap

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

heap.add(list[i]);

// Remove elements from the heap

for (int i = list.length - 1; i >= 0; i--)

list[i] = heap.remove();

}

/\*\* A test method \*/

public static void main(String[] args) {

Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};

heapSort(list);

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

System.out.print(list[i] + " ");

}

}

**要使用堆对数组排序，应首先使用Heap类创建一个对象，使用add方法将所有元素添加到堆中，然后使用remove方法从堆中删除所有元素。以降序删除这些元素。**

**堆排序的时间复杂度**

**设h表示包含n个元素的堆的高度。由于，h<log(n+1)和log(n+1)<=h,因此，log(n+1)<=h<log(n+1)+1。所以堆的高度为O(logn)。**

**由于add方法会追踪从叶子结点到根结点的路径，因此向堆中添加一个新元素最多需要h步。所以，建立一个包含n个元素的数组的初始堆需要O(nlogn)时间。因为remove方法要追踪从根结点到叶子结点的路径，因此从堆中删除根结点后，重建堆最多需要h步。由于要调用n次remove方法，所以由堆产生一个有序数组需要的总时间为O(nlogn)。**

**归并排序和堆排序需要的时间都为O(nlogn)。为归并两个子数组，归并排序需要一个临时数组，而堆排序不需要额外的数组空间。因此，堆排序的空间效率高于归并排序。**

**桶排序和基数排序**

**桶排序和基数排序是对整数进行排序的高效算法。**

**桶排序算法的工作方式如下。假设键值的范围是从0到t。我们需要t+1个标记为0,1,2...t的桶。如果元素的键值为i，那么久将该元素放入桶i中。每个桶中都放着具有相同键值的元素。 可以使用ArrayList来实现一个桶。**

**void bucketSort(E[] list){**

**E[] bucket=(E[])new java.util.ArrayList[t+1];**

**//Distribute the elements from list to buckets**

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

**int key=list[i].getKey();//Assume element has the getKey() method**

**if(bucket[key]==null)**

**bucket[key]=new java.util.ArrayList<>();**

**bucket[key].add(list[i]);**

**}**

**//Now move the elements from the buckets back to list**

**int k=0;**

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

**if(bucket[i]!=null){**

**for(int j=0;j<bucket[i].size();j++){**

**list[k++]=bucket[i].get(j);**

**}**

**}**

**}**

**}**

**它需要耗费O(n+t)的时间来对线性表排序，使用的空间是O(n+t)，其中n是指线性表的大小。桶排序是稳定的。**

**注意，如果t太大，那么桶排序不是很可取。此时，可以使用基数排序。基数排序是基于桶排序的，但是它只使用10个桶。**

**假定键值是正整数。基数排序（radix sort）的思路就是将这些键值基于它们的基数位置分为子组。然后反复地从最小的基数位置开始，对其上的键值应用桶排序。**

**通常基数排序需要耗费O(dn)时间对带整数键值的n个元素排序，其中d是所有键值中基数位数目的最大值。**

**外部排序**

**可以使用外部排序对大容量数据进行排序。**

**前面讨论的所有排序算法，都假定要排序的所有数据在内存中都同时可用，如数组。要对存储在外部文件中的数据排序，首先要将数据送入内存，然后对它们进行内部排序。然而，如果文件太大，那么文件中的所有数据不能同时送入内存。因此需要外部排序（external sort）。**

**外部排序的复杂度为O(nlogn)**

**小结：**

**1.选择排序、插入排序、冒泡排序和快速排序的最差时间复杂度为O(n^2)。**

**2.归并排序的平均情况和最差情况的复杂度为O(nlogn)。快速排序的平均时间也是O(nlogn)。**

**3.对于设计排序这样的高效算法，堆是一个很有用的数据结构。**

**4.堆排序的时间复杂度为O(nlogn)。**

**5.桶排序和基数排序都是针对整数键值的特定排序算法。这些算法不是通过比较键值而是使用桶来对键值排序的，它们会比一般的排序算法效率更高。**

**6.可以使用归并排序的一种变体——称为外部排序——对外部文件中的大型数据进行排序。**