常见排序算法总结

1. **插入排序**
2. **直接插入排序**

（1）工作原理

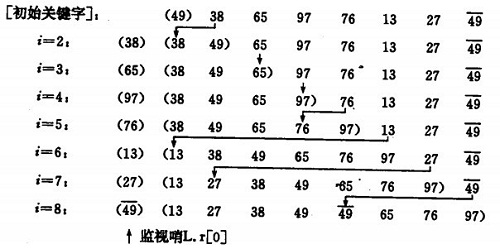
插入排序是一种最简单直观的排序算法，它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

（2）算法步骤

①将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。

②从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置。（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）

直接插入排序示例：



（3）核心代码

|  |
| --- |
| public void sort(){  int temp;  for(int i = 1; i<arraytoSort.length; i++){  for(int j = i-1; j>=0; j--){  if( arraytoSort[j+1] < arraytoSort[j] ){  temp = arraytoSort[j+1];  arraytoSort[j+1] = arraytoSort[j];  arraytoSort[j] = temp;  }  }  }  } |

（4）性能分析

①时间复杂度O(n^2), 空间复杂度O(1)：

最好的情况下：正序有序(从小到大)，这样只需要比较n次，不需要移动。因此时间复杂度为O(n)；

最坏的情况下：逆序有序,这样每一个元素就需要比较n次，共有n个元素，因此实际复杂度为O(n^2)。平均情况下：O(n^2).

②稳定性：稳定。

③排序时间与输入有关：输入的元素个数；元素已排序的程度。

1. **希尔排序**

（1）工作原理

希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本。但希尔排序是非稳定排序算法。

希尔排序是基于插入排序的以下两点性质而提出改进方法的：

①插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率。

②但插入排序一般来说是低效的， 因为插入排序每次只能将数据移动一位。

希尔排序的基本思想是：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，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

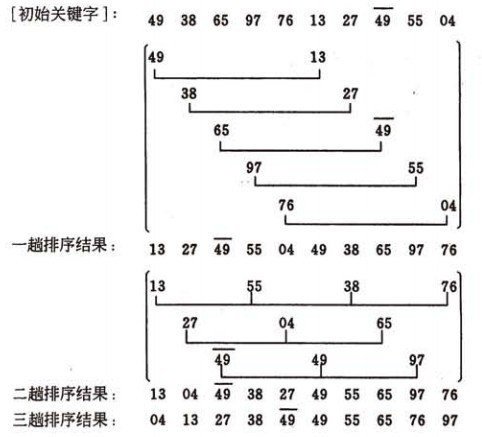
（2）算法步骤

①选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

②按增量序列个数k，对序列进行k 趟排序；

③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。

希尔排序的示例：



（3）核心代码

|  |
| --- |
| public class ShellSort {  /\*\*  \* 希尔排序的一趟插入  \* @param arr 待排数组  \* @param d 增量  \*/  public static void shellInsert(int[] arr, int d) {  for(int i=d; i<arr.length; i++) {  int j = i - d;  int temp = arr[i]; //记录要插入的数据  while (j>=0 && arr[j]>temp) { //从后向前，找到比其小的数的位置  arr[j+d] = arr[j]; //向后挪动  j -= d;  }  if (j != i - d) //存在比其小的数  arr[j+d] = temp;  }  }  public static void shellSort(int[] arr) {  if(arr == null || arr.length == 0)  return ;  int d = arr.length / 2;  while(d >= 1) {  shellInsert(arr, d);  d /= 2;  }  }  } |

（4）性能分析

①希尔排序的时间复杂度与增量(即，步长gap)的选取有关。例如，当增量为1时，希尔排序退化成了直接插入排序，此时的时间复杂度为O(N²)，而Hibbard增量的希尔排序的时间复杂度为O(N^**3/2**)。

②稳定性：不稳定。

<http://www.cnblogs.com/skywang12345/p/3597597.html>

1. **交换排序**
2. **冒泡排序**

（1）工作原理

相邻的数据进行两两比较，小数放在前面，大数放在后面，这样一趟下来，最小的数就被排在了第一位，第二趟也是如此，如此类推，直到所有的数据排序完成。

（2）算法步骤（从后往前）

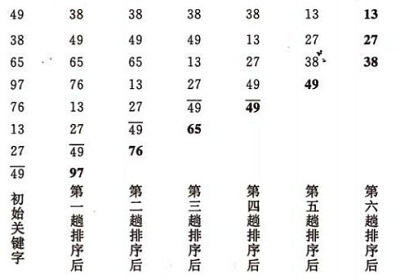
①比较相邻的元素。如果第一个比第二个大，就交换他们两个。

②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。这步做完后，最后的元素会是最大的数。

③针对所有的元素重复以上的步骤，除了最后一个。

④持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

冒泡排序的示例：



（3）核心代码

|  |
| --- |
| void bubbleSort(int a[]){  for(int i =0 ; i< a.length-1; ++i){  for(int j = 0; j < a.length-1-i; ++j){  if(a[j] > a[j+1])  {  int tmp = a[j];  a[j] = a[j+1];  a[j+1] = tmp;  }  }  }  } |

（4）性能分析

①时间复杂度O(n^2), 空间复杂度O(1):

若记录序列的初始状态为"正序"，则冒泡排序过程只需进行一趟排序，在排序过程中只需进行n-1次比较，且不移动记录；

反之，若记录序列的初始状态为"逆序"，则需进行n(n-1）/2次比较和记录移动。因此冒泡排序总的时间复杂度为O(n\*n)。

②稳定性：稳定，因为存在两两比较，不存在跳跃。

③排序时间与输入无关，最好，最差，平均都是O(n^2)

<http://www.cnblogs.com/kkun/archive/2011/11/23/bubble_sort.html>

1. **快速排序**

（1）工作原理

①选择一个基准元素,通常选择第一个元素或者最后一个元素；

②通过一趟排序讲待排序的记录分割成独立的两部分，其中一部分记录的元素值均比基准元素值小。另一部分记录的 元素值比基准值大；

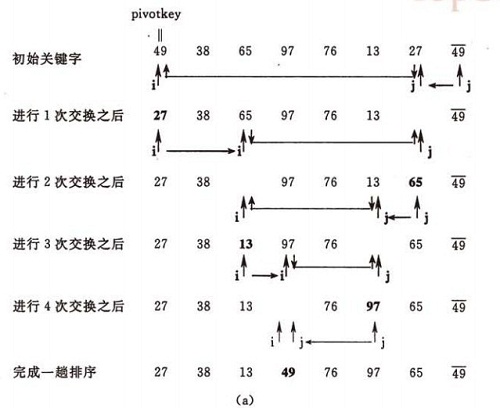
③此时基准元素在其排好序后的正确位置；

④然后分别对这两部分记录用同样的方法继续进行排序，直到整个序列有序。

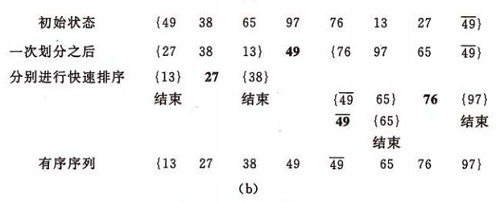
（2）算法步骤

快速排序的示例：

①一趟排序的过程：



②排序的全过程：



（3）核心代码（递归实现）

|  |
| --- |
| void print(int a[], int n){  for(int j= 0; j<n; j++){  cout<<a[j] <<" ";  }  cout<<endl;  }  void swap(int \*a, int \*b)  {  int tmp = \*a;  \*a = \*b;  \*b = tmp;  }  int partition(int a[], int low, int high)  {  int privotKey = a[low]; //基准元素  while(low < high){ //从表的两端交替地向中间扫描  while(low < high && a[high] >= privotKey) --high; //从high 所指位置向前搜索，至多到low+1 位置。将比基准元素小的交换到低端  swap(&a[low], &a[high]);  while(low < high && a[low] <= privotKey ) ++low;  swap(&a[low], &a[high]);  }  print(a,10);  return low;  }  void quickSort(int a[], int low, int high){  if(low < high){  int privotLoc = partition(a, low, high); //将表一分为二  quickSort(a, low, privotLoc -1); //递归对低子表递归排序  quickSort(a, privotLoc + 1, high); //递归对高子表递归排序  }  }  int main(){  int a[10] = {3,1,5,7,2,4,9,6,10,8};  cout<<"初始值：";  print(a,10);  quickSort(a,0,9);  cout<<"结果：";  print(a,10);  } |

（4）性能分析

①时间复杂度 O(nlogn) 空间复杂度O(logn)：

时间复杂度：

最坏O（n^2）====当划分不均匀时候 逆序and排好序都是最坏情况；

最好O（n）======当划分均匀；

partition的时间复杂度: O（n）一共需要logn次partition.

空间复杂度：(递归造成的栈空间的使用)

最好情况，递归树的深度logn 空间复杂的logn，最坏情况，需要进行n‐1 递归调用，其空间复杂度为 O(n)，平均情况，空间复杂度也为O(log2n)。

②稳定性：不稳定 （两个时间复杂度O(nlogn) 的排序算法都不稳定）

由于关键字的比较和交换是跳跃进行的，因此，快速排序是一种不稳定的排序方法。

1. **选择排序**
2. **简单选择排序**

（1）工作原理

在要排序的一组数中，选出最小（或者最大）的一个数与第1个位置的数交换；然后在剩下的数当中再找最小（或者最大）的与第2个位置的数交换，依次类推，直到第n-1个元素（倒数第二个数）和第n个元素（最后一个数）比较为止。

（2）算法步骤

①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

②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。

③重复第二步，直到所有元素均排序完毕。

（3）核心代码

|  |
| --- |
| public void sort(){  for(int i = 0; i<arraytoSort.length-1; i++){  int min = i;  int temp;  //find min  for(int j = i+1; j<arraytoSort.length ;j++){  if(arraytoSort[j] <arraytoSort[min]){  min = j;  }  }  //swap the min with the ith element  temp = arraytoSort[min];  arraytoSort[min] = arraytoSort[i];  arraytoSort[i] = temp;  }  } |

（4）性能分析

①时间复杂度O(n^2), 空间复杂度O(1)

②排序时间与输入无关，最佳情况，最坏情况都是如此,

③稳定性：不稳定。

1. **堆排序**

（1）堆的定义

n 个关键字序列 K1，K2，…，Kn 称为堆，当且仅当该序列满足如下性质(简称为堆性质)：



若将此序列所存储的向量 R[1..n]看做是一棵完全二叉树的存储结构，则**堆实质上是满足如下性质的完全二叉树：**树中任一非叶结点的关键字均不大于(或不小于)其左右孩子(若存在)结点的关键字。

堆分为大顶堆和小顶堆：

满足**Key[i]>=Key[2i+1] && key[i]>=key[2i+2]**称为大顶堆；

满足**Key[i]<=key[2i+1] && Key[i]<=key[2i+2]**称为小顶堆。

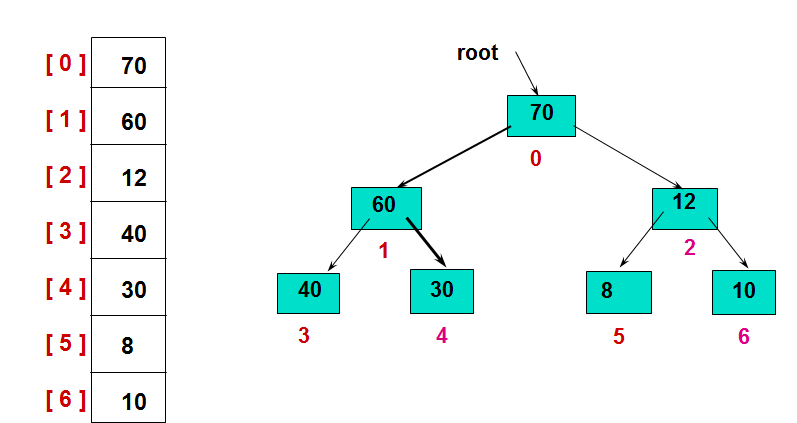
由上述性质可知大顶堆的堆顶的关键字肯定是所有关键字中最大的，小顶堆的堆顶的关键字是所有关键字中最小的。

（2）堆的存储

一般用数组来表示堆，若根结点存在序号0处，i结点的父结点下标就为(i-1)/2。i结点的左右子结点下标分别为2\*i+1和2\*i+2。

注：如果根结点是从1开始，则左右孩子结点分别是2i和2i+1。如第0个结点左右子结点下标分别为1和2。

如最大化堆如下（左图为其存储结构，右图为其逻辑结构）：



（3）算法步骤

①创建一个堆H[0..n-1]；

②把堆首（最大值）和堆尾互换；

③把堆的尺寸缩小1，并调用shift\_down(0),目的是把新的数组顶端数据调整到相应位置；

④重复步骤2，直到堆的尺寸为1。

（4）核心代码

|  |
| --- |
| public class HeapSort {  public void buildheap(int array[]){  int length = array.length;  int heapsize = length;  int nonleaf = length / 2 - 1;  for(int i = nonleaf; i>=0;i--){  heapify(array,i,heapsize);  }  }  public void heapify(int array[], int i,int heapsize){  int smallest = i;  int left = 2\*i+1;  int right = 2\*i+2;  if(left<heapsize){  if(array[i]>array[left]){  smallest = left;  }  else smallest = i;  }  if(right<heapsize){  if(array[smallest]>array[right]){  smallest = right;  }  }  if(smallest != i){  int temp;  temp = array[i];  array[i] = array[smallest];  array[smallest] = temp;  heapify(array,smallest,heapsize);  }  }  public void heapsort(int array[]){  int heapsize = array.length;  buildheap(array);  for(int i=0;i<array.length-1;i++){  // swap the first and the last  int temp;  temp = array[0];  array[0] = array[heapsize-1];  array[heapsize-1] = temp;  // heapify the array  heapsize = heapsize - 1;  heapify(array,0,heapsize);  }  } |

（5）性能分析

①时间复杂度 O(nlogn), 空间复杂度O(1):

排序时间与输入无关，最好，最差，平均都是O(nlogn)。 堆排序借助了堆这个数据结构，堆类似二叉树，又具有堆积的性质（子节点的关键值总小于（大于）父节点）

堆排序包括两个主要操作：

保持堆的性质 heapify(A,i)======时间复杂度O(logn)；

建堆 buildmaxheap(A)===========时间复杂度O(n)线性时间建堆。

堆排序是就地排序，辅助空间为 O(1)。

②稳定性：不稳定。

**对于大数据的处理：** 如果对100亿条数据选择Topk数据，选择快速排序好还是堆排序好？ 答案是只能用**堆排序**。 堆排序只需要维护一个k大小的空间，即在内存开辟k大小的空间。而快速排序需要开辟能存储100亿条数据的空间，which is impossible。

由于建初始堆所需的比较次数较多，所以堆排序不适宜于记录数较少的文件。

1. **归并排序**

（1）工作原理

归并排序（Merge）是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

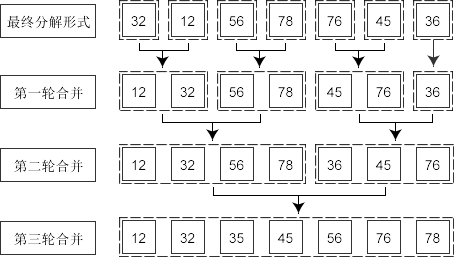
（2）算法步骤（以二路归并为例）

①把 n 个记录看成 n 个长度为 l 的有序子表；

②进行两两归并使记录关键字有序，得到 n/2 个长度为 2 的有序子表；

③重复第 2 步直到所有记录归并成一个长度为 n 的有序表为止。

归并排序的操作流程：



（3）核心代码

|  |
| --- |
| public int merge( int p, int q, int r ){  int count = 0;  int[] right = assignlist(p,q);  //赋值左半部分数组（赋值就是用for循环，还有一个哨兵：最后一个元素设置为maxvalue）  int[] left = assignlist(q+1,r); //赋值有半部分数组  int i = 0;  int j = 0;  for(int k = p; k<=r; k++){  if(right[i] <= left[j]){  arraytoSort[k] = right[i];  i++;  }  else if(right[i] > left[j]){  arraytoSort[k] = left[j];  j++;  count = count + (q - p + 1) -i;  }  }  return count;  }  void MergeSort(int arry[],int start,int end)  {  if(start<end)  {  int mid=(start+end)/2;//数组重点  MergeSort(arry,start,mid);//递归调用，排序前半段arry[start...mid]  MergeSort(arry,mid+1,end);//递归调用，排序后半段arry[mid+1,end]  MergeArry(arry,start,mid,end);//归并上述两段有序数组。  }  } |

（4）性能分析

①时间复杂度：O(nlogn), 空间复杂度：O(n) + O(logn)：

对长度为 n 的文件，需进行lgn趟二路归并，每趟归并的时间为 O(n)，故其时间复杂度无论是在最好情况下还是在最坏情况下均是 O(nlgn)。

需要一个辅助向量来暂存两有序子文件归并的结果，故其辅助空间复杂度为 O(n)，显然它不是就地排序。

②稳定性：稳定。

（5）算法适用场景

归并排序在数据量比较大的时候也有较为出色的表现（效率上），但是，其空间复杂度O(n)使得在数据量特别大的时候（例如，1千万数据）几乎不可接受。而且，考虑到有的机器内存本身就比较小，因此，采用归并排序一定要注意。

归并排序比较占用内存，但却是一种效率高且稳定的算法。

1. **基数排序**

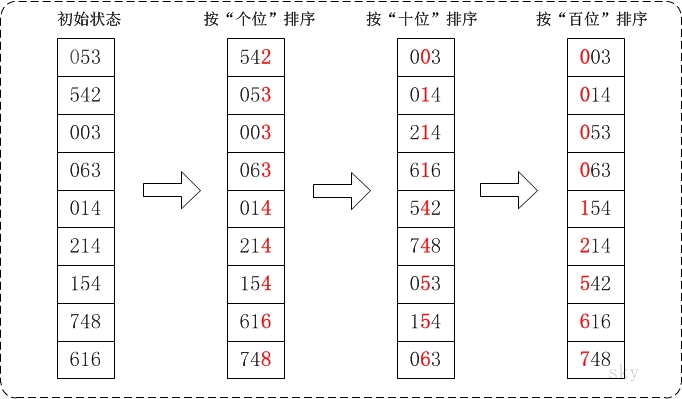
（1）工作原理

基数排序(Radix Sort)是桶排序的扩展，它的基本思想是：将整数(必须是正整数)按位数切割成不同的数字，然后按每个位数分别比较。

具体做法是：将所有待比较数值统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后, 数列就变成一个有序序列。

（2）算法步骤

通过基数排序对数组{53, 3, 542, 748, 14, 214, 154, 63, 616}，它的示意图如下：



在上图中，首先将所有待比较树脂统一为统一位数长度，接着从最低位开始，依次进行排序。

①按照个位数进行排序。

②按照十位数进行排序。

③按照百位数进行排序。

排序后，数列就变成了一个有序序列。

（3）核心代码

|  |
| --- |
| public class RadixSort {  /\*  \* 获取数组a中最大值  \* 参数说明：  \* a -- 数组  \* n -- 数组长度  \*/  private static int getMax(int[] a) {  int max;  max = a[0];  for (int i = 1; i < a.length; i++)  if (a[i] > max)  max = a[i];  return max;  }  /\*  \* 对数组按照"某个位数"进行排序(桶排序)  \* 参数说明：  \* a -- 数组  \* exp -- 指数。对数组a按照该指数进行排序。  \* 例如，对于数组a={50, 3, 542, 745, 2014, 154, 63, 616}；  \* (01) 当exp=1表示按照"个位"对数组a进行排序  \* (02) 当exp=10表示按照"十位"对数组a进行排序  \* (03) 当exp=100表示按照"百位"对数组a进行排序  \* ...  \*/  private static void countSort(int[] a, int exp) {  //int output[a.length]; // 存储"被排序数据"的临时数组  int[] output = new int[a.length]; // 存储"被排序数据"的临时数组  int[] buckets = new int[10];  // 将数据出现的次数存储在buckets[]中  for (int i = 0; i < a.length; i++)  buckets[ (a[i]/exp)%10 ]++;  // 更改buckets[i]。目的是让更改后的buckets[i]的值，是该数据在output[]中的位置。  for (int i = 1; i < 10; i++)  buckets[i] += buckets[i - 1];  // 将数据存储到临时数组output[]中  for (int i = a.length - 1; i >= 0; i--) {  output[buckets[ (a[i]/exp)%10 ] - 1] = a[i];  buckets[ (a[i]/exp)%10 ]--;  }  // 将排序好的数据赋值给a[]  for (int i = 0; i < a.length; i++)  a[i] = output[i];  output = null;  buckets = null;  }  /\*  \* 基数排序  \* 参数说明：  \* a -- 数组  \*/  public static void radixSort(int[] a) {  int exp; // 指数。当对数组按各位进行排序时，exp=1；按十位进行排序时，exp=10；...  int max = getMax(a); // 数组a中的最大值  // 从个位开始，对数组a按"指数"进行排序  for (exp = 1; max/exp > 0; exp \*= 10)  countSort(a, exp);  }  public static void main(String[] args) {  int i;  int a[] = {53, 3, 542, 748, 14, 214, 154, 63, 616};  System.out.printf("before sort:");  for (i=0; i<a.length; i++)  System.out.printf("%d ", a[i]);  System.out.printf("\n");  radixSort(a); // 基数排序  System.out.printf("after sort:");  for (i=0; i<a.length; i++)  System.out.printf("%d ", a[i]);  System.out.printf("\n");  }  } |

（4）性能分析

①时间复杂度

设待排序的数组R[1..n]，数组中最大的数是d位数，基数为r（如基数为10，即10进制，最大有10种可能，即最多需要10个桶来映射数组元素）。处理一位数，需要将数组元素映射到r个桶中，映射完成后还需要收集，相当于遍历数组一遍，最多元素书为n，则时间复杂度为O(n+r)。所以，总的时间复杂度为O(d\*(n+r))。

②空间复杂度

设待排序的数组R[1..n]，数组中最大的数是d位数，基数为r。基数排序过程中，用到一个计数器数组，长度为r，还用到一个r\*n的二位数组来做为桶，所以空间复杂度为O(r\*n)。

③稳定性

每一轮映射和收集操作，都保持从左到右的顺序进行，如果出现相同的元素，则保持他们在原始数组中的顺序。

可见，基数排序是一种稳定的排序。

1. **总结**

以上排序算法的时间、空间与稳定性的总结如下：

（1）当原表有序或基本有序时，直接插入排序和冒泡排序将大大减少比较次数和移动记录的次数，时间复杂度可降至O（n）；

（2）而快速排序则相反，当原表基本有序时，将蜕化为冒泡排序，时间复杂度提高为O（n^2）；

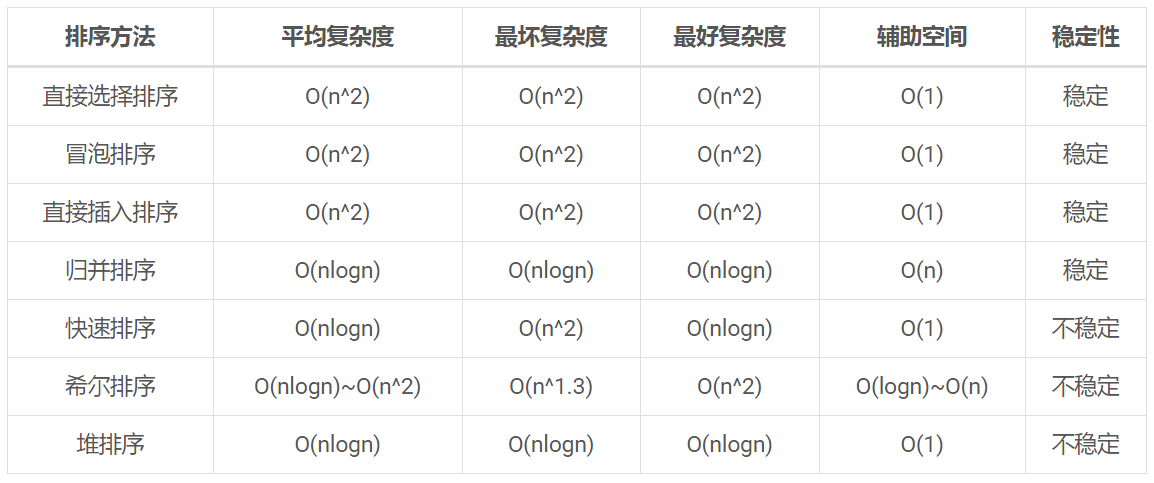
（3）原表是否有序，对简单选择排序、堆排序、归并排序和基数排序的时间复杂度影响不大。

（4）排序算法的稳定性：若待排序的序列中，存在多个具有相同关键字的记录，经过排序， 这些记录的相对次序保持不变，则称该算法是稳定的；反之则不稳定的。

稳定性的好处：排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。基数排序就是这样，先按低位排序，逐次按高位排序，低位相同的元素其顺序再高位也相同时是不会改变的。另外，如果排序算法稳定，可以避免多余的比较。

①稳定的排序算法：冒泡排序、插入排序、归并排序、选择排序。

②不是稳定的排序算法：快速排序、希尔排序、堆排序。



（5）选择排序算法准则：

每种排序算法都各有优缺点。因此，在实用时需根据不同情况适当选用，甚至可以将多种方法结合起来使用。一般而言，需要考虑的因素有以下四点：

①待排序的记录数目n的大小；

②记录本身数据量的大小，也就是记录中除关键字外的其他信息量的大小；

③关键字的结构及其分布情况；

④对排序稳定性的要求。

（6）设待排序元素的个数为n.

①当n较大，则应采用时间复杂度为O(n\*logn)的排序方法：快速排序、堆排序或归并排序。

快速排序：是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；

堆排序：如果内存空间有要求但没有稳定性要求；

归并排序：它有一定数量的数据移动，所以我们可能过与插入排序组合，先获得一定长度的序列，然后再合并，在效率上将有所提高。

②当n较大，内存空间允许，且要求稳定性：归并排序

③当n较小，可采用直接插入或直接选择排序。

直接插入排序：当元素分布有序，直接插入排序将大大减少比较次数和移动记录的次数。

直接选择排序：当元素分布有序，如果不要求稳定性，选择直接选择排序。

④一般不使用或不直接使用传统的冒泡排序。