常见排序算法

排序是计算机内经常进行的一种操作，其目的是将一组“无序”的记录序列调整为“有序”的记录序列。

排序分内排序和外排序。
内排序：指在排序期间数据对象全部存放在内存的排序。
外排序：指在排序期间全部对象个数太多，不能同时存放在内存，必须根据排序过程的要求，不断在内、外存之间移动的排序。

内排序的方法有许多种，按所用策略不同，可归纳为五类：插入排序、选择排序、交换排序、归并排序和分配排序。
插入排序主要包括直接插入排序和希尔排序两种；
选择排序主要包括直接选择排序和堆排序；
交换排序主要包括冒泡排序和快速排序；
归并排序主要包括二路归并（常用的归并排序）和自然归并。
分配排序主要包括箱排序和基数排序。

稳定排序：假设在待排序的文件中，存在两个或两个以上的记录具有相同的关键字，在用某种排序法排序后，若这些相同关键字的元素的相对次序仍然不变，则这种排序方法是稳定的。
其中**冒泡，插入，基数，归并**属于稳定排序；
选择，快速，希尔，堆属于不稳定排序。

时间复杂度是衡量算法好坏的最重要的标志。
排序的时间复杂度与算法执行中的**数据比较次数**与**数据移动次数**密切相关。

以下给出介绍简单的排序方法：插入排序，选择排序，冒泡排序。
三种算法的时间复杂度都是 n^2 级的。

冒泡排序：

标准的冒泡排序过程如下：
首先比较 a[1]与 a[2]的值，若 a[1]大于 a[2]则交换两者的值，否则不变。
再比较 a[2]与 a[3]的值，若 a[2]大于 a[3]则交换两者的值，否则不变。
再比较 a[3]与 a[4]，以此类推，最后比较 a[n-1]与 a[n]的值。
这样处理一轮后，a[n]的值一定是这组数据中最大的。
再对 a[1]~a[n-1]以相同方法处理一轮。
共处理 n-1 轮后 a[1]、a[2]、……a[n]就以升序排列了。

过程举例：

初始元素序列：	**8**	**3**	**2**	**5**	**9**	**3***	**6**		
第一趟排序：	**3**	**2**	**5**	**8**	**3***	**6**	**【**	**9**	**】**
第二趟排序：	**2**	**3**	**5**	**3***	**6**	**【8**		**9**	**】**
第三趟排序：	**2**	**3**	**3***	**5**	**【**	**6**	**8**	**9**	**】**
第四趟排序：	**2**	**3**	**3***	**【5**	**6**	**8**		**9**	**】**
第五趟排序：	**2**	**3**	**【3***	**5**	**6**	**8**		**9**	**】**
第六趟排序：	**2**	**【3**	**3***	**5**	**6**	**8**		**9**	**】**

以下是冒泡排序的代码（模板），不过跟以上描述有些区别，做了些改进。

```
template <class T>
void BubbleSort(T a[],int n)
{
    int i,j,k;
    T t;

    for(i = n - 1; i > 0;i = k)    //将 i 设置为被交换的最后一对元素中较小的下标
        for(k = j = 0; j < i;j++)
            if(a[j] > a[j+1])
            {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                k = j; //如有交换发生，记录较小元素的下标
            }
}
```

这样，对于某段有序的序列，下一次遍历时就不用再比较了。最理想的是整个序列本就有序，如此，k = 0,只遍历一遍就完成了。

冒泡排序是经典的排序算法，因其过程像冒泡而得名。
也因容易理解，很多教科书都将其收入。
然而冒泡排序的效果确是各种算法里较为糟糕的，特别是当数据量大时。

直接选择排序

算法描述：

首先找出最大的元素，将其与 a[n-1]位置交换；

然后在余下的 n-1 个元素中寻找最大的元素，将其与 a[n-2]位置交换，

如此进行下去直至 n 个元素排序完毕。

过程举例：

初始元素序列：	**8**	**3**	**2**	**5**	**9**	**3***	**6**
第一趟排序：	**8**	**3**	**2**	**5**	**6**	**3***	**【9 】**
第二趟排序：	**3***	**3**	**2**	**5**	**6**	**【 8**	**9 】**
第三趟排序：	**3***	**3**	**2**	**5**	**【6**	**8**	**9 】**
第四趟排序：	**3***	**3**	**2**	**【5**	**6**	**8**	**9 】**
第五趟排序：	**2**	**3**	**【3***	**5**	**6**	**8**	**9 】**
第六趟排序：	**2**	**【 3**	**3***	**5**	**6**	**8**	**9 】**

直接选择排序的模板

```
template <class T>
void SelectionSort(T a[],int n)
{
    int i,j,k;
    T t;

    for(i = 0;i < n - 1; i++)
    {
        for(k = i,j = i + 1;j < n;j++)
            if(a[k] > a[j])
                k = j;
        if(k != j)    //这行也可以不要
        {
            t = a[k];
            a[k] = a[i];
            a[i] = t;
        }
    }
}
```

直接选择排序交换少，比较多。元素比较费劲时(如字符串比较)不建议用此算法。

直接插入排序

算法描述：

每步将一个待排序元素，插入到前面已经排好序的一组元素的适当位置上，直到全部元素插入为止。

过程举例：

初始元素序列：	**【8 】**	**3**	**2**	**5**	**9**	**3***	**6**
第一趟排序：	**【3**	**8】**	**2**	**5**	**9**	**3***	**6**
第二趟排序：	**【2**	**3**	**8】**	**5**	**9**	**3***	**6**
第三趟排序：	**【2**	**3**	**5**	**8】**	**9**	**3***	**6**
第四趟排序：	**【2**	**3**	**5**	**8**	**9】**	**3***	**6**
第五趟排序：	**【2**	**3**	**3***	**5**	**8**	**9】**	**6**
第六趟排序：	**【2**	**3**	**3***	**5**	**6**	**8**	**9】**

直接插入排序模板

```
template <class T>
void InsertionSort(T a[],int n)
{
    int i,j;
    T t;

    for(i = 1;i < n;i++)
    {
        t = a[i];
        for(j = i - 1;j >= 0 && a[j] > t;j--)
          a[j+1] = a[j];
        a[j+1] = t;
    }
}
```

直接插入排序是稳定的排序算法，也是三种简单排序算法最快的。

下面介绍几种常见的高级的排序算法：希尔排序，堆排序，快速排序，二路归并排序。这几种排序比以上三种算法要难，但效率却要高许多(当要比较的元素较多时)。

希尔排序

希尔排序(Shell Sort)是插入排序的一种。是针对直接插入排序算法的改进。该方法又称缩小增量排序，因 DL．Shell 于 1959 年提出而得名。

希尔排序属于插入类排序,是将整个无序列分割成若干小的子序列分别进行插入排序

算法描述：

先取一个正整数 d1<n，把所有序号相隔 d1 的数组元素放一组，组内进行直接插入排序；然后取 d2<d1，重复上述分组和排序操作；直至 di=1，即所有记录放进一个组中排序为止 。

过程举例：

假设待排序文件有 10 个记录，其关键字分别是：

49，38，65，97，76，13，27，49，55，04。

增量序列 d 的取值依次为：

5，3，1

```
d=5
49 38 65 97 76 13 27 49* 55 04
49 13
|-------------------|
38 27
|-------------------|
65 49*
|-------------------|
97 55
|-------------------|
76 04
|-------------------|
一趟结果
13 27 49* 55 04 49 38 65 97 76
-------------------------------------------------------------------
```

d=3

13 27 49* 55 04 49 38 65 97 76

13 55 38 76

|------------|------------|------------|

27 04 65

|------------|------------|

49* 49 97

|------------|------------|

二趟结果

13 04 49* 38 27 49 55 65 97 76

---

d=1

13 04 49* 38 27 49 55 65 97 76

|----|----|----|----|----|----|----|----|----|

三趟结果

04 13 27 38 49* 49 55 65 76 97

希尔排序模板

```
template <class T>
void ShellSort(T a[],int n)
{
    int i,j,k;
    T t;

    k = n / 2;
    while(k > 0)
    {
        for(i = k;i < n;i++)
        {
            t = a[i];
            for(j = i - k;j >= 0 && a[j] > t;j -= k)
                a[j+k] = a[j];
            a[j+k] = t;
        }
        k /= 2;
    }
}
```

可以看到以上代码与直接插入排序极为相似。
希尔排序的增量 d 选取有很多方法，以上代码就是取 d =n/2；然后去 d=d/2 一直到 d=1。或许这不是最优的增量序列，但却是最简单的。

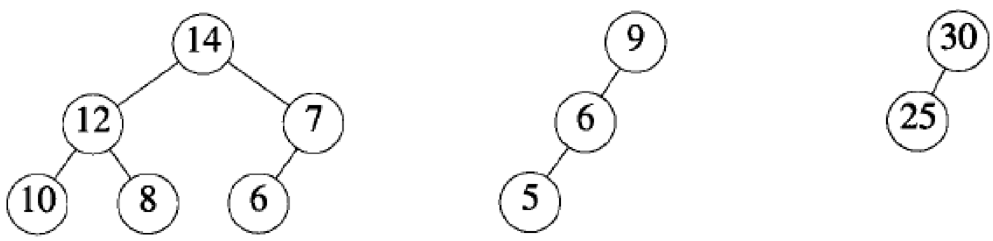
刚开始时，d 最大，每一组的元素个数很少，排序速度很快；
d 变小时，每一组元素变多，但元素基本有序了，插入排序对于有序的序列效率很高。
所以，希尔排序的时间复杂度会比直接插入排序好。
然而，在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，因而 shell 排序是不稳定的。

希尔排序是高级排序算法中最容易实现的，效率也不赖。考试或比赛时若需要排序，这是不错的选择。

堆排序

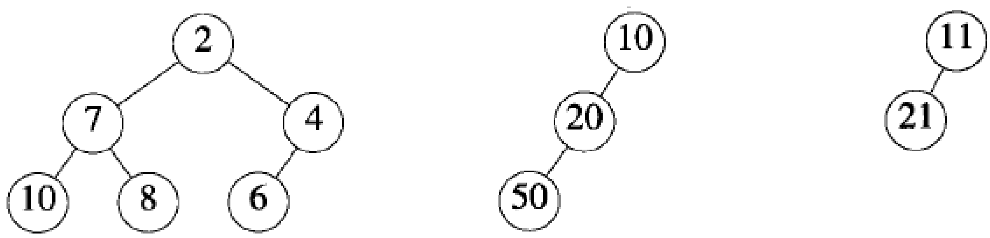
要了解堆排序，先得认识“堆”。这涉及到数据结构的知识，不清楚之处请查阅相关书籍或从网上查找相关资料。此处为介绍堆排序，简单介绍堆。

[最大树（最小树）]：每个节点的值都大于（小于）或等于其子节点（如果有的话）值的树。最大树（max tree）与最小树（ min tree）的例子分别如图9 – 1、9 – 2所示，虽然这些树都是二叉树，但最大树不必是二叉树，最大树或最小树节点的子节点个数可以大于2。



a) b) c)

图9-1 最大树



a) b) c)

图9-2 最小树

[**最大堆（最小堆）**]：最大（最小）的完全二叉树。
图9-1b 所示的最大树并不是最大堆（ max heap），另外两个最大树是最大堆。图9-2b 所示的最小树不是最小堆（min heap），而另外两个是。

最大堆（最小堆）有一个特性，顶端的节点（根节点）为最大（最小）元素。由此特性到一种排序算法——**堆排序**。做法是将一个无序序列转化为堆，然后依次取出根节点，所得序列是有序序列。

算法描述：
（用最大堆排序。）
先将初始文件 R[0:n-1]建成一个大根堆，此堆为初始的无序区 ；
再将关键字最大的记录 R[0]（即堆顶）和无序区的最后一个记录 R[n-1]交换，由此得到新的无序区 R[0:n-2]和有序区 R[n-1]，且满足 R[0:n-2].keys≤R[n-1].key ；
由于交换后新的根 R[0]可能违反堆性质，故应将当前无序区 R[0:n-2]调整为堆（重建堆）。
然后再次将 R[0:n-2]中关键字最大的记录 R[0]和该区间的最后一个记录 R[n-2]交换，由此得到新的无序区 R[0:n-3]和有序区 R[n-2:n-1]，且仍满足关系 R[0..n-3].keys≤R[n-2:n-1].keys，同样要将 R[0:n-3]调整为堆……
直到无序区只有一个元素 R[0]时，R[0:n-1]为有序序列。

堆排序涉及两个堆的操作:**初始化堆**；**堆元素的删除**。堆的操作比较复杂，而且有很多版本，建议读者自己找堆的资料研读。以下代码仅为说明堆排序的关键操作。

```
template<class T>
void DeleteMax(T& x,T heap[],int &size)
{//把最大元素取出，赋值给 x，并从堆中删除

    x = heap[0]; //取出最大元素

    //重建堆
    T y = heap[--size];//堆中最后面的元素
    //从根开始，为 y 寻找合适的位置
    int i = 0,
        ci = 1; //节点 heap[i]的子节点
    while (ci <= size)
    {
        //令 heap[ci]为子节点中较大者
        if (ci < size && heap[ci] < heap[ci+1])
            ci++;
```

```
            //若 y 比两子节点都大，跳出循环（插入父节点(heap[i])的位置）
            if (y >= heap[ci])
                break;

            heap[i] = heap[ci]; //将较大节点上移
            i = ci;             //到下一层（与较大节点的子节点比较）
            ci *= 2;
        }
        heap[i] = y;
}


template<class T>
void Initialize(T a[], int size)
{//把无序的序列 a[0:n-1]转换为堆序(最大堆）
  //初始化过程与删除操作有相似的地方

        T *heap = a;

        for (int i = size/2; i >= 0; i--)
        {
            T y = heap[i];

            int c = 2*i;
            while (c <= size)
            {
                if (c < size && heap[c] < heap[c+1])
                    c++;
                if (y >= heap[c])
                    break;
                heap[c/2] = heap[c];
                c *= 2;
            }
            heap[c/2] = y;
        }
}
```

```
template <class T>
void HeapSort(T a[], int n)
{
	int size = n;
	T x;

	Initialize(a,size);//初始化堆

	for (int    i= n-1; i > 0; i--)
	{
		DeleteMax(x,a,size);
		a[i] = x;
	}
}
```

堆排序的时间，主要由建立初始化堆和反复重建堆这两部分的时间开销构成。初始化堆可在 O(n)的时间内完成，而重建堆可在 O(logn)的时间内完成。易知堆排序的最坏时间复杂度为 O(nlogn)。

快速排序

快速排序又叫分区交换排序，它是对起泡排序方法的一种改进。
由 C.A.R. Hoare 于 1962 年提出。

算法思想：
通过一次排序将待排序对象分割成独立的两部分，
其中一部分对象的关键码均比另一部分对象的关键码小，
再分别对这两部分子序列对象继续进行排序，以达到整个序列有序。

排序过程描述
假设有 n 个待排序的对象{a[0]，a[1]，…，a[n-1]}
首先任选一个对象 a[0]（通常选取序列中得第一个对象）作为支点（pivot）；
然后调整序列中各个对象的位置；
将所有关键码小于或等于 a[0]的对象排在 a[0]的前面；
将所有关键码大于 a[0]的对象排在 a[0]的后面，即{≤a[0]}：a[0]：{>a[0]}；
以上过程称作一次快速排序。

初始序列：

12	2	16	30	28	10	16*	20	6	18

一次快速排序后：

0	1	2	3	4	5	6	7	8	9	
10	2	6	12	28	30	16*	20	16	18	
i	i	i	i j	i j	j	j	j	j	j	j

pivot=12

在第一次快速排序中，确定了所选取的支点对象 a[0]最终在序列中的排列位置，同时也把剩余的对象分成两个子序列。

对两个子序列分别进行快速排序，又确定了两个对象在序列中应处的位置，并将剩余对象分成了四个子序列：

即{≤a[i]}：a[i]：{a[i]<且≤a[0]}：a[0]：{a[0]<且≤a[j]}:a[j] ：{>a[j]}

如此重复下去，当各子序列的长度为 1 时，全部对象排序完毕。

完整过程举例：

	0	1	2	3	4	5	6	7	8	9
初始状态	12	2	16	30	28	10	16*	20	6	18
第一次快速排序	10	2	6	12	28	30	16*	20	16	18
第二次快速排序	6	2	10	12	16	18	16*	20	28	30
第三次快速排序	2	6	10	12	16*	16	18	20	28	30
第四次快速排序	2	6	10	12	16*	16	18	20	28	30

快速排序模板

```
#ifndef QuickSort_
#define QuickSort_

template<class T>
void quickSort(T a[], int l, int r)
{//对 a[l:r]排序
    if (l >= r) return;

    int i = l,       //从左到右的游标
        j = r + 1;  //从右到左的游标
    T t, pivot = a[l]; //以 a[l]为支点

    while (true)
    {
        while (a[++i] < pivot && i < r);  //从左边找>=pivot 的元素
        while (a[--j] > pivot && j > l);  //从右边找<=pivot 的元素

        if (i >= j) break;  //如未找到交换对,退出循环

        t = a[i];
        a[i] = a[j];
        a[j] = t;
    }

    //将支点 a[l]与 a[j]交换
    a[l] = a[j];
    a[j] = pivot;

    quickSort(a, l, j-1); // 左段快速排序
    quickSort(a, j+1, r); // 右段快速排序
}

template<class T>
void QuickSort(T *a, int n)
{
    quickSort(a, 0, n-1);
}
```

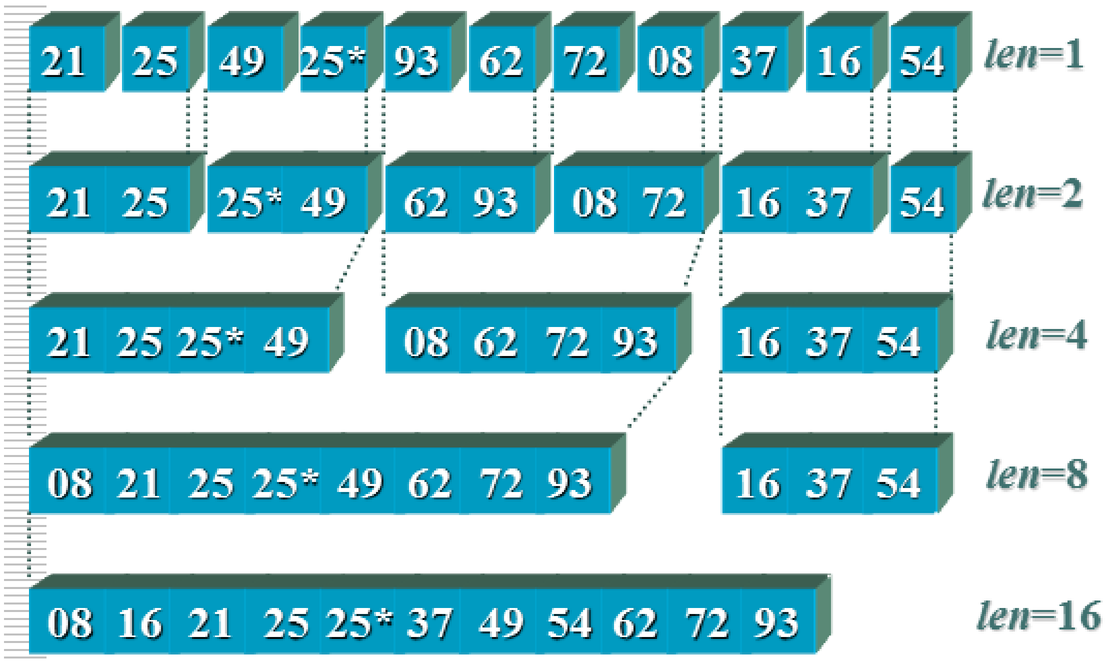
快速排序被认为是最快的内排序算法。当然，当数据量少时，快速排序甚至不及简单的排序算法；此外，当数据本身已有序时，快速排序的效率极低。
一般情况下，快速排序的时间复杂度为 O(nlogn)。
快速排序因其递归需要额外空间，数据量大时有可能会造成堆栈溢出，使程序会崩掉（还好现在操作系统做得好，不容易死机）。想办法转为非递归可避免这问题。

二路归并排序

归并排序是分治法思想运用的一个典范。二路归并排序是常用归并排序。

算法描述：
将有 n 个对象的原始序列看作 n 个有序子序列，每个序列的长度为 1；
从第一个子序列开始，把相邻的子序列两两合并，得到⌈n / 2⌉ 个长度为 2 或 1 的归并项 （如果 n 为奇数，则最后一个有序子序列的长度为 1），称这一过程为一次归并排序；
再对第一次归并排序后⌈n / 2⌉个子序列采用上述方法继续顺序成对归并，…，如此重复，直至得到一个长度为 n 的子序列为止。该子序列就是原始序列归并排序后的有序序列。

过程演示：



二路归并排序模板

```
template <class T>
void Merge(T c[], T d[], int l, int m, int r)
{// 合并 c[l:m] 和 c[m:r] 到 d[l:r].
    int i = l,
        j = m+1,
        k = l;

    while ((i <= m) && (j <= r))
    {
        if (c[i] <= c[j]) d[k++] = c[i++];
        else d[k++] = c[j++];
    }

    if (i > m)
        for (int q = j; q <= r; q++)
            d[k++] = c[q];
    else
        for (int q = i; q <= m; q++)
            d[k++] = c[q];
}


template <class T>
void MergePass(T x[], T y[], int s, int n)
{ // 归并大小为 s 的相邻段.

    int i = 0,
        t = s + s;

    while (i <= n - t)
    { // 归并两个大小为 s 的相邻段
        Merge(x, y, i, i+s-1, i+t-1);
        i = i + t;
    }

    // 剩下元素不足 2s
```

```
    if (i + s < n)
        Merge(x, y, i, i+s-1, n-1);
    else
        for(int j = i; j <= n-1; j++)  // 把最后一段复制到 y
            y[j] = x[j];
}

template <class T>
void MergeSort(T a[],int n)
{//使用归并排序算法对 a[0:n-1]进行排序
    T *b = new T [n];
    int s = 1;

    while (s < n)
    {
        MergePass(a, b, s, n);
        s += s;
        MergePass(b, a, s, n);
        s += s;
    }

    if(b) delete[] b;
}
```

函数 MergeSort( )调用 MergePass( )$\lceil \log_2 n \rceil$ 次；
函数 MergePass( ) 做一趟两路归并排序，要调用 merge ( )函数 $O(n/s)$ 次；
Merge( )每次要执行比较 $O(s)$次；
算法总的时间复杂度为 $O(n\log_2 n)$。

由于两路归并排序中，每两个有序表合并成一个有序表时，若分别在两个有序表中出现有相同关键码，则会使前一个有序表中相同关键码先复制，后一有序表中相同关键码后复制，从而保持它们的相对次序不会改变。
所以，两路归并排序是一种稳定的排序方法。

至此，介绍了七种通用的排序算法。
这些通用算法均是基于关键字之间的比较来实现的，而从理论上已经证明：对于以上的排序，无论用何种方法都至少进行「logn「 次比较，因而最优的排序算法时间复杂度为 nlogn。
不需要比较的排序方法可使时间复杂度为线性级别：O(n)。
分配排序就是基于这种不需要比较的排序算法。分配排序包括箱子排序和基数排序。

箱子排序

箱排序又称桶排序，其基本思想是：设置若干个箱子,依次扫描待排序的记录 R[0],R[1],…,R[range-1]，把关键字等于 k 的记录全部都装入到第 k 个箱子里(分配)，然后按序号依次将各非空的箱子首尾连接起来。

箱子排序仅适用于对象的关键字是一定范围内的整数（或可映射至一定范围的整数)，比如某人群的年龄，或学生的分数（不考虑小数时）等。
比如，假若学生分数为 0~100 的整数，今要给这些学生的分数排序，可分配 100 个箱子，k 分就分配到第 k 个箱子，最后从第一个箱子起，逐一收起箱子，所得序列就是非递减序列。

箱子排序模板

```
template <class T>
void BinSort(T a[],int n,int range,int(*value)(T& x))
{//range 是元素的范围
    int *b = new int[range+1];
    T *t = new T[n];

    int i,temp,sum = 0;

    //将箱子置空
    for(i = 0;i <= range;i++)
        b[i] = 0;

    //统计每个箱子该有多大
    for(i = 0;i < n;i++)
        b[value(a[i])]++;//value 是一个以 T 类型数据为参数,返回 T 的关键字(整型)的函数

    //根据箱子大小，计算每个非空箱子的元素该放的位置（放在 t[]中）
    for(i = 0;i <= range;i++)
    {
```

```
        if(b[i] != 0)
        {
            temp = b[i];
            b[i] = sum;
            sum += temp;
        }
    }

    //根据a[i]的关键字找到所在的箱子
    //再根据b[x](0<=x<n)确定a[i]在t[]中的位置
    for(i = 0;i < n;i++)
        t[ b[value(a[i])]++ ] = a[i];

    //把值赋回数组a
    for(i = 0;i < n;i++)
        a[i] = t[i];

    delete[] b,t;
}
```

笔者见过的箱子排序是用链表实现的，根据其思想，我用数组实现了以上代码，谨供参考。

其中，**int(*value)(T& x)**是一以T类型数据为参数，返回值为int型的函数。这样做是为了提高代码的通用性。

比如当T类型为结构体

```
struct  Student
{
char name[20];
int age;
};
```

而函数是

```
int fun(Student& A)
{
return A.age;
}
```

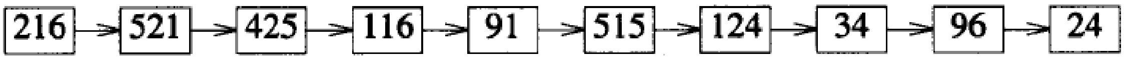
时，

可以这样调用箱子排序：BinSort（a,n,20,fun）;

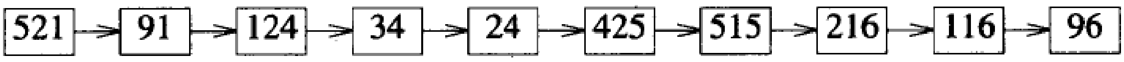
其中a是Student数组，20是学生最大年龄。

从上面的代码可以看出箱子排序的时间复杂度是 O(n+range),是线性级别的。

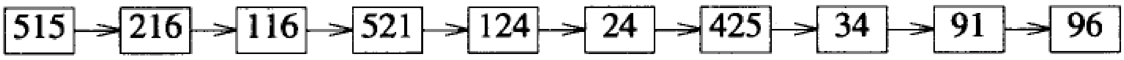
基数排序是箱子排序的扩展，用于数据范围大的序列，可节省“箱子”。
比如 10 个整数，范围是 0~999，用 1000 个箱子去装就太浪费了。可以把这些数分 3 段，个位，十位和百位。此时，基数是 10。
过程举例：



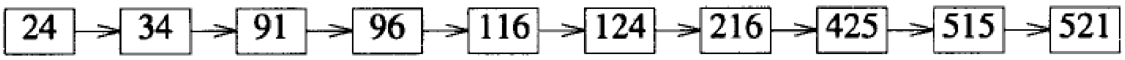
a)



b)



c)



d)

第一行是初始序列；后面依次根据个位，十位，百位对元素排序。
最终，序列化为有序序列。

以上例子可以体现了稳定排序的用途：分级排序。
箱子排序是稳定排序，排序后，次低位数字相同的节点，其相对次序保持不变。正是由于这个特性，使得数据可以分段分级排序。
基数排序也是稳定的排序算法。

文章的编写引用了许多资料，包括百度百科，机械工业出版社的《数据结构、算法与应用——C++语言描述》，以及老师的课件等。
在此鸣谢！

附录：

笔者编写了一个程序测试了以上一些排序函数的性能，鉴于测试函数的科学性及计算机的稳定性等原因，数据仅定性地反映各函数性能。

测试数据为整型时：

C:\windows\system32\cmd.exe

	插入	选择	冒泡	希尔	堆	快速	归并
n=10	0.000	0.000	0.001	0.000	0.000	0.001	0.001
n=50	0.005	0.002	0.007	0.003	0.005	0.002	0.004
n=100	0.018	0.018	0.032	0.009	0.015	0.008	0.017
n=500	0.189	0.408	0.610	0.087	0.062	0.040	0.093
n=1000	0.975	1.858	2.824	0.259	0.192	0.124	0.239
n=5000	18.175	36.518	66.064	1.539	0.832	1.064	0.859
n=10000	87.455	174.438	361.224	4.019	1.472	3.544	2.099
n=20000	358.855	728.038	1639.024	7.219	5.972	6.644	5.299
n=50000	2062.854	4178.838	9978.624	19.619	18.572	15.844	11.499

请按任意键继续. . .

测试数据为实型的时：

C:\windows\system32\cmd.exe

	插入	选择	冒泡	希尔	堆	快速	归并
n=10	0.000	0.001	0.000	0.000	0.000	0.001	0.001
n=50	0.003	0.006	0.005	0.004	0.003	0.006	0.004
n=100	0.013	0.021	0.023	0.010	0.009	0.015	0.010
n=500	0.232	0.443	0.770	0.089	0.055	0.061	0.074
n=1000	1.042	1.939	3.142	0.245	0.177	0.157	0.200
n=5000	19.742	37.219	70.162	1.825	0.497	1.437	0.520
n=10000	92.142	179.459	391.562	4.945	1.737	3.277	3.040
n=20000	391.642	747.259	1779.862	11.345	4.837	7.877	6.240
n=50000	2279.442	4288.259	10821.662	27.145	20.437	14.077	15.440

请按任意键继续. . .

备注：时间单位是毫秒。