浅谈算法和数据结构（2）：基本排序算法

2016-07-16 算法爱好者

（点击上方公众号，可快速关注）

> 来源：寒江独钓
>
> 链接：http://www.cnblogs.com/yangecnu/p/Introduction-Insertion-and-Selection-and-Shell-Sort.html

本篇开始学习排序算法。排序与我们日常生活中息息相关，比如，我们要从电话簿中找到某个联系人首先会按照姓氏排序、买火车票会按照出发时间或者时长排序、买东西会按照销量或者好评度排序、查找文件会按照修改时间排序等等。在计算机程序设计中，排序和查找也是最基本的算法，很多其他的算法都是以排序算法为基础，在一般的数据处理或分析中，通常第一步就是进行排序，比如说二分查找，首先要对数据进行排序。在Donald Knuth 的计算机程序设计的艺术这四卷书中，有一卷是专门介绍排序和查找的。

排序的算法有很多，在维基百科上有这么一个分类，另外大家有兴趣也可以直接上维基百科上看相关算法，本文也参考了上面的内容。

首先来看比较简单的选择排序(Selection sort)，插入排序(Insertion sort)，然后在分析插入排序的特征和缺点的基础上，介绍在插入排序基础上改进的希尔排序(Shell sort)。

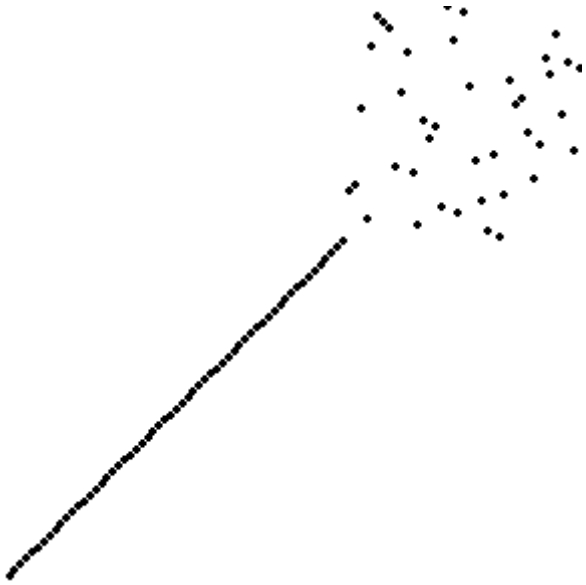
一 选择排序

原理：

选择排序很简单，他的步骤如下：

1. 从左至右遍历，找到最小(大)的元素，然后与第一个元素交换。
2. 从剩余未排序元素中继续寻找最小（大）元素，然后与第二个元素进行交换。
3. 以此类推，直到所有元素均排序完毕。

之所以称之为选择排序，是因为每一次遍历未排序的序列我们总是从中选择出最小的元素。下面是选择排序的动画演示：



实现：

算法实现起来也很简单，我们新建一个Sort泛型类，让该类型必须实现IComparable接口，然后我们定义SelectionSort方法，方法传入T数组，代码如下：

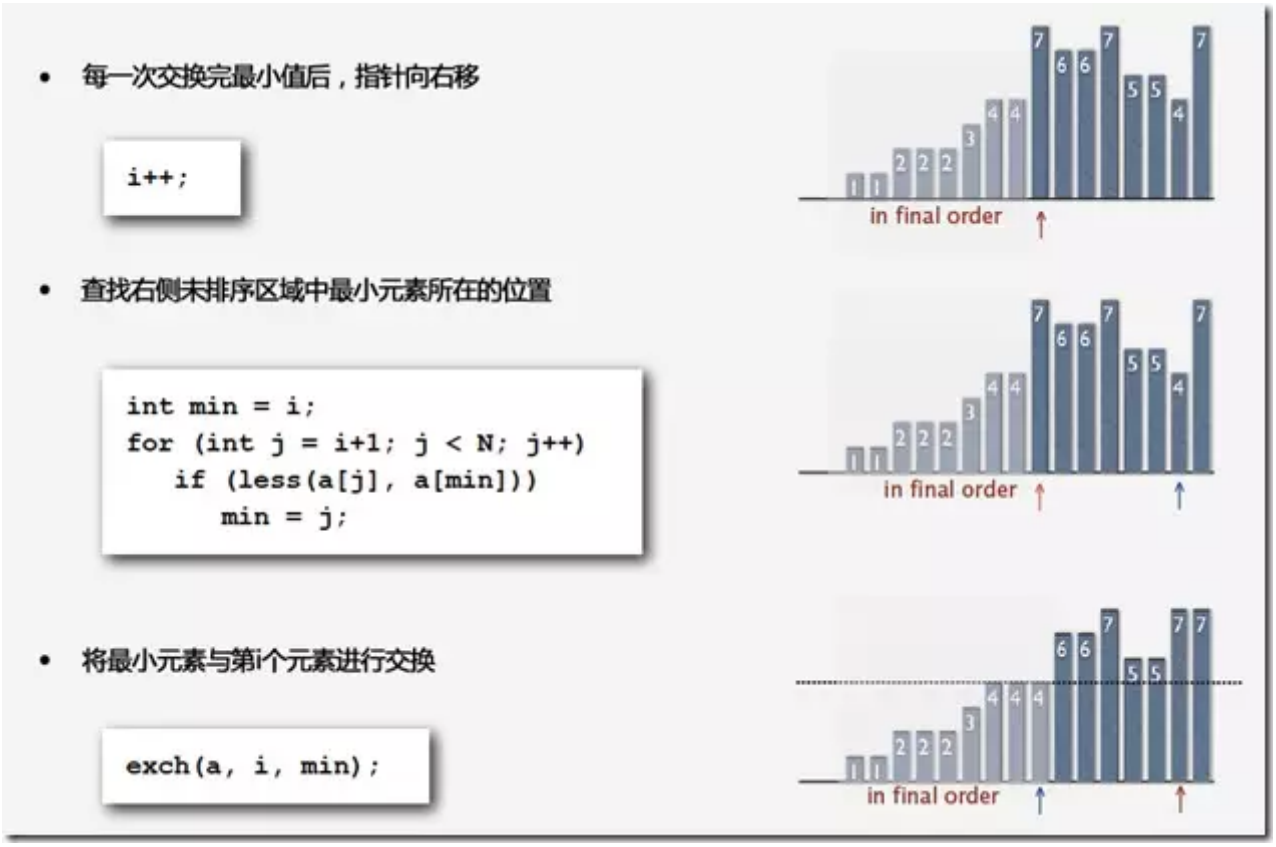
```
/// <summary>
/// 排序算法泛型类，要求类型实现IComparable接口
/// </summary>
/// <typeparam name="T"></typeparam>
public class Sort<T> where T : IComparable<T>
{
    /// <summary>
```

```
    /// 选择排序
    /// </summary>
    /// <param name="array"></param>
    public static void SelectionSort(T[] array)
    {
        int n = array.Length;

        for (int i = 0; i < n; i++)
        {
            int min = i;
            //从第i+1个元素开始，找最小值
            for (int j = i + 1; j < n; j++)
            {
                if (array[min].CompareTo(array[j]) > 0)
                    min = j;
            }
            //找到之后和第i个元素交换
            Swap(array, i, min);
        }
    }

    /// <summary>
    /// 元素交换
    /// </summary>
    /// <param name="array"></param>
    /// <param name="i"></param>
    /// <param name="min"></param>
    private static void Swap(T[] array, int i, int min)
    {
        T temp = array[i];
        array[i] = array[min];
        array[min] = temp;
    }
}
```

下图分析了选择排序中每一次排序的过程，您可以对照图中右边的柱状图来看。



测试如下：

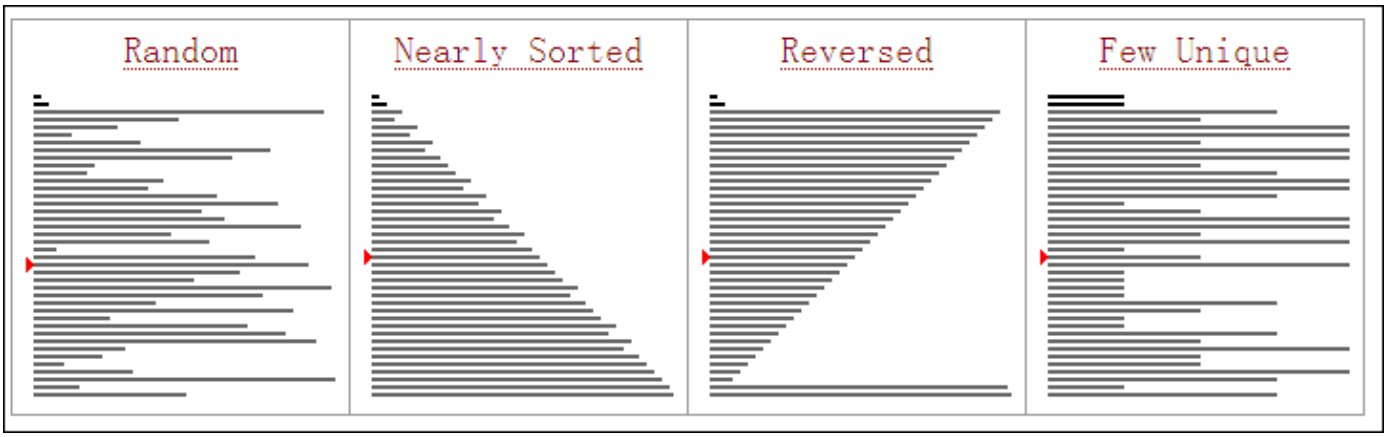
```
static void Main(string[] args)
{
    Int32[] array = new Int32[] { 1, 3, 1, 4, 2, 4, 2, 3, 2, 4, 7, 6, 6, 7, 5, 5, 7, 7 };
    Console.WriteLine("Before SelectionSort:");
    PrintArray(array);
    Sort<Int32>.SelectionSort(array);
    Console.WriteLine("After SelectionSort:");
    PrintArray(array);
    Console.ReadKey();
}
```

输出结果：

```
Before SelectionSort:
1,3,1,4,2,4,2,3,2,4,7,6,6,7,5,5,7,7,
After SelectionSort:
1,1,2,2,2,3,3,4,4,4,5,5,6,6,7,7,7,7,
```

分析：

选择排序的在各种初始条件下的排序效果如下：



1. 选择排序需要花费 (N – 1) + (N – 2) + … + 1 + 0 = N(N- 1) / 2 ~ N2/2次比较 和 N-1次交换操作。
2. 对初始数据不敏感，不管初始的数据有没有排好序，都需要经历N2/2次比较，这对于一些原本排好序，或者近似排好序的序列来说并不具有优势。在最好的情况下，即所有的排好序，需要0次交换，最差的情况，倒序，需要N-1次交换。
3. 数据交换的次数较少，如果某个元素位于正确的最终位置上，则它不会被移动。在最差情况下也只需要进行N-1次数据交换，在所有的完全依靠交换去移动元素的排序方法中，选择排序属于比较好的一种。

二 插入排序

原理：

插入排序也是一种比较直观的排序方式。可以以我们平常打扑克牌为例来说明，假设我们那在手上的牌都是排好序的，那么插入排序可以理解为我们每一次将摸到的牌，和手中的牌从左到右依次进行对比，如果找到合适的位置则直接插入。具体的步骤为：

1. 从第一个元素开始，该元素可以认为已经被排序
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描
3. 如果该元素小于前面的元素（已排序），则依次与前面元素进行比较如果小于则交换，直到找到大于该元素的就则停止；
4. 如果该元素大于前面的元素（已排序），则重复步骤2
5. 重复步骤2~4 直到所有元素都排好序 。

下面是插入排序的动画演示：

实现：

在Sort泛型方法中，我们添加如下方法，下面的方法和上面的定义一样

```
/// <summary>
/// 插入排序
/// </summary>
/// <param name="array"></param>
public static void InsertionSort(T[] array)
{
    int n = array.Length;
    //从第二个元素开始
    for (int i = 1; i < n; i++)
    {
        //从第i个元素开始，一次和前面已经排好序的i-1个元素比较，如果小于，则交换
        for (int j = i; j > 0; j--)
        {
            if (array[j].CompareTo(array[j - 1]) < 0)
            {
                Swap(array, j, j - 1);
            }
            else//如果大于，则不用继续往前比较了，因为前面的元素已经排好序，比较大的大就是教大的了。
                break;
        }
    }
}
```

测试如下：

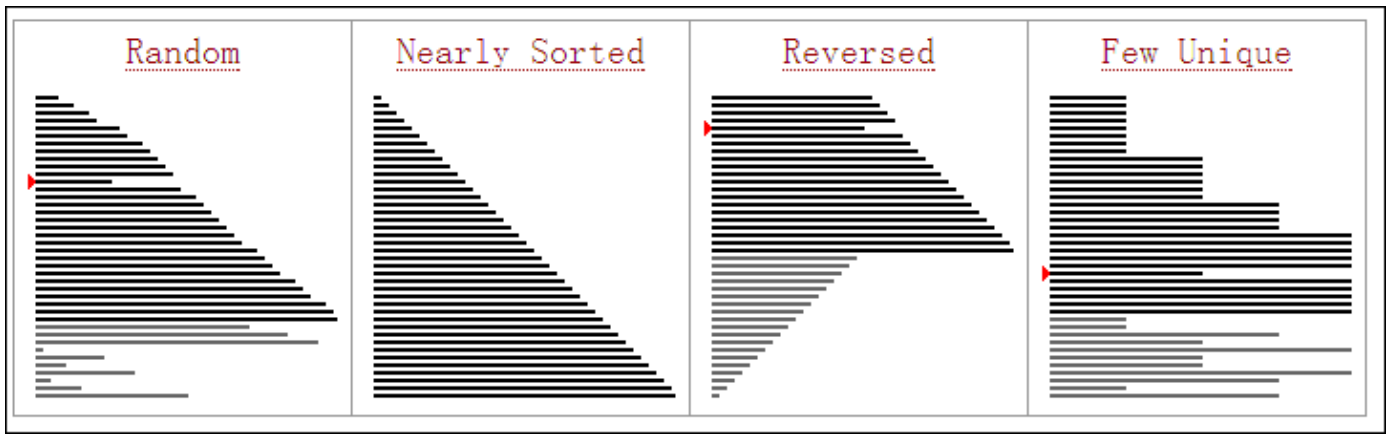
```
Int32[] array1 = new Int32[] { 1, 3, 1, 4, 2, 4, 2, 3, 2, 4, 7, 6, 6, 7, 5, 5, 7, 7 };
Console.WriteLine("Before InsertionSort:");
PrintArray(array1);
Sort<Int32>.InsertionSort(array1);
Console.WriteLine("After InsertionSort:");
PrintArray(array1);
Console.ReadKey();
```

输出结果：

```
Before InsertionSort:
1,3,1,4,2,4,2,3,2,4,7,6,6,7,5,5,7,7,
After InsertionSort:
1,1,2,2,2,3,3,4,4,4,5,5,6,6,7,7,7,7,
```

分析：

插入排序的在各种初始条件下的排序效果如下：



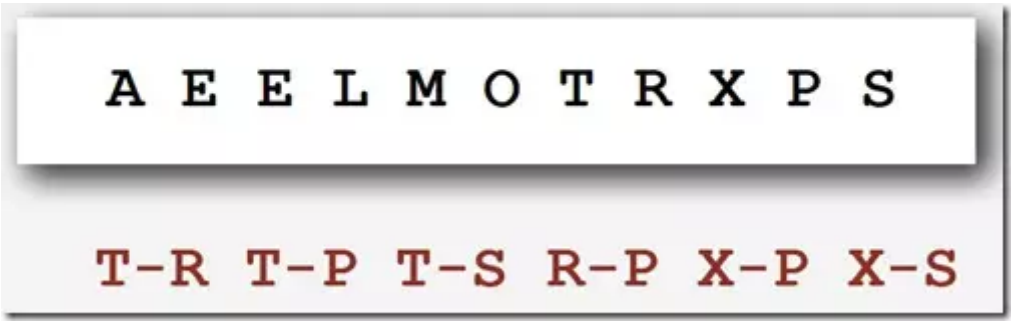
1. 插入排序平均需要N2/4次比较和N2/4 次交换。在最坏的情况下需要N2/2 次比较和交换；在最好的情况下只需要N-1次比较和0次交换。

先考虑最坏情况，那就是所有的元素逆序排列，那么第i个元素需要与前面的i-1个元素进行i-1次比较和交换，所有的加起来大概等于N(N- 1) / 2 ~ N2 / 2，在数组随机排列的情况下，只需要和前面一半的元素进行比较和交换，所以平均需要N2/4次比较和N2/4次交换。

在最好的情况下，所有元素都排好序，只需要从第二个元素开始都和前面的元素比较一次即可，不需要交换，所以为N-1次比较和0次交换。

2. 插入排序中，元素交换的次数等于序列中逆序元素的对数。元素比较的次数最少为元素逆序元素的对数，最多为元素逆序的对数 加上数组的个数减1。

3.总体来说，插入排序对于部分有序序列以及元素个数比较小的序列是一种比较有效的方式。

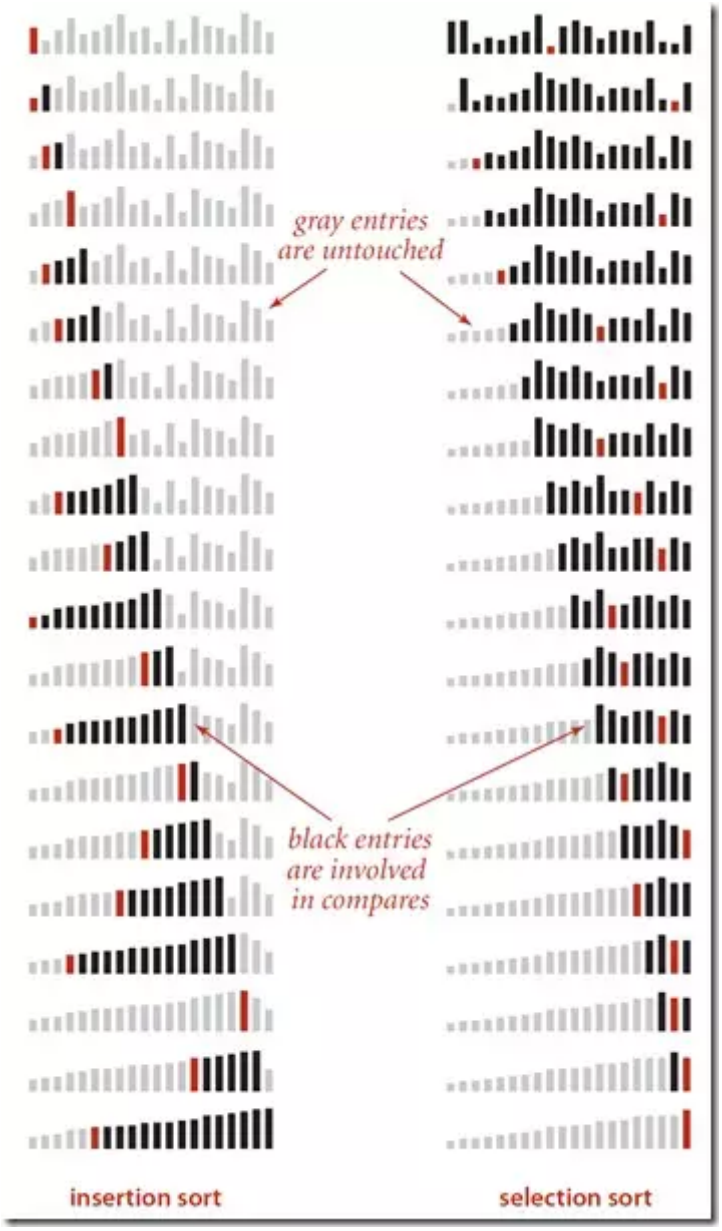


如上图中，序列AEELMOTRXPS，中逆序的对数为T-R，T-P，T-S，R-P，X-S 6对。典型的部分有序队列的特征有：

- 数组中每个元素离最终排好序后的位置不太远
- 小的未排序的数组添加到大的已排好序的数组后面
- 数组中只有个别元素未排好序

对于部分有序数组，插入排序是比较有效的。当数组中逆元素的对数越低，插入排序要比其他排序方法要高效的多。

选择排序和插入排序的比较：



上图展示了插入排序和选择排序的动画效果。图中灰色的柱子是不用动的，黑色的是需要参与到比较中的，红色的是参与交换的。图中可以看出：

插入排序不会动右边的元素，选择排序不会动左边的元素；由于插入排序涉及到的未触及的元素要比插入的元素要少，涉及到的比较操作平均要比选择排序少一半。

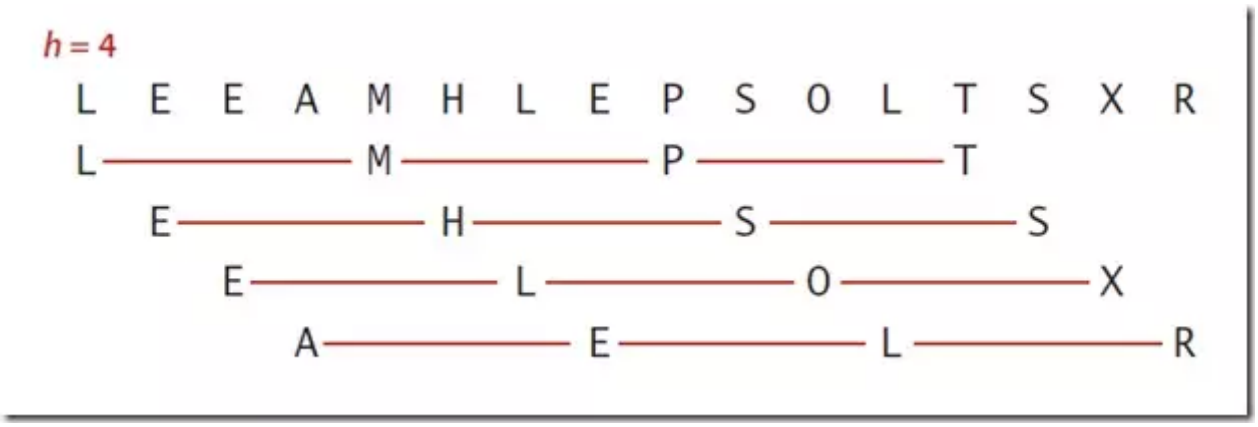
三 希尔排序(Shell Sort)

原理：

希尔排序也称之为递减增量排序，他是对插入排序的改进。在第二部插入排序中，我们知道，插入排序对于近似已排好序的序列来说，效率很高，可以达到线性排序的效率。但是插入排序效率也是比较低的，他一次只能将数据向前移一位。比如如果一个长度为N的序列，最小的元素如果恰巧在末尾，那么使用插入排序仍需一步一步的向前移动和比较，要N-1次比较和交换。

希尔排序通过将待比较的元素划分为几个区域来提升插入排序的效率。这样可以让元素可以一次性的朝最终位置迈进一大步，然后算法再取越来越小的步长进行排序，最后一步就是步长为1的普通的插入排序的，但是这个时候，整个序列已经是近似排好序的，所以效率高。

如下图，我们对下面数组进行排序的时候，首先以4为步长，这是元素分为了LMPT，EHSS，ELOX，AELR几个序列，我们对这几个独立的序列进行插入排序，排序完成之后，我们减小步长继续排序，最后直到步长为1，步长为1即为一般的插入排序，他保证了元素一定会被排序。



希尔排序的增量递减算法可以随意指定，可以以N/2递减，只要保证最后的步长为1即可。

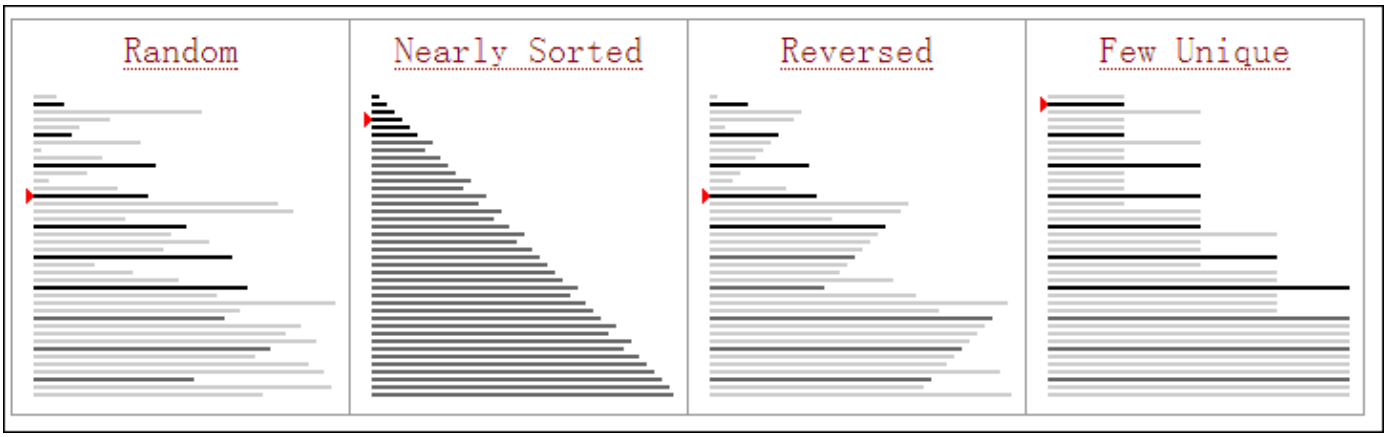
实现：

```
/// <summary>
/// 希尔排序
/// </summary>
/// <param name="array"></param>
public static void ShellSort(T[] array)
{
    int n = array.Length;
```

```
    int h = 1;
    //初始最大步长
    while (h < n / 3) h = h * 3 + 1;
    while (h >= 1)
    {
        //从第二个元素开始
        for (int i = 1; i < n; i++)
        {
            //从第i个元素开始，依次次和前面已经排好序的i-h个元素比较，如果小于，则交换
            for (int j = i; j >= h; j = j - h)
            {
                if (array[j].CompareTo(array[j - h]) < 0)
                {
                    Swap(array, j, j - h);
                }
                else//如果大于，则不用继续往前比较了，因为前面的元素已经排好序，比较大的大就是教大的了。
                    break;
            }
        }
        //步长除3递减
        h = h / 3;
    }
}
```

可以看到，希尔排序的实现是在插入排序的基础上改进的，插入排序的步长为1，每一次递减1，希尔排序的步长为我们定义的h，然后每一次和前面的-h位置上的元素进行比较。算法中，我们首先获取小于N/3 的最大的步长，然后逐步长递减至步长为1的一般的插入排序。

下面是希尔排序在各种情况下的排序动画：



分析：

1. 希尔排序的关键在于步长递减序列的确定，任何递减至1步长的序列都可以，目前已知的比较好的序列有：

- Shell’s 序列: N/2 , N/4 , …, 1 (重复除以2);
- Hibbard’s 序列: 1, 3, 7, …, 2k – 1 ;
- Knuth’s 序列: 1, 4, 13, …, (3k – 1) / 2 ;该序列是本文代码中使用的序列。
- 已知最好的序列是 Sedgewick’s (Knuth的学生，Algorithems的作者)的序列: 1, 5, 19, 41, 109, ….

该序列由下面两个表达式交互获得:

- 1, 19, 109, 505, 2161,….., 9(4k – 2k) + 1, k = 0, 1, 2, 3,…
- 5, 41, 209, 929, 3905,…..2k+2 (2k+2 – 3) + 1, k = 0, 1, 2, 3, …

“比较在希尔排序中是最主要的操作，而不是交换。”用这样步长的希尔排序比插入排序和堆排序都要快，甚至在小数组中比快速排序还快，但是在涉及大量数据时希尔排序还是比快速排序慢。

2. 希尔排序的分析比较复杂，使用Hibbard’s 递减步长序列的时间复杂度为O(N3/2)，平均时间复杂度大约为O(N5/4) ,具体的复杂度目前仍存在争议。

3. 实验表明，对于中型的序列(万)，希尔排序的时间复杂度接近最快的排序算法的时间复杂度nlogn。

四 总结

最后总结一下本文介绍的三种排序算法的最好最坏和平均时间复杂度。

名称	最好	平均	最坏	内存占用	稳定排序
插入排序	n	n2	n2	1	是
选择排序	n2	n2	n2	1	否
希尔排序	n	nlog2n 或 n3/2	依赖于增量递减序列目前最好的是 nlog2n	1	否

希望本文对您了解以上三个基本的排序算法有所帮助，后面将会介绍合并排序和快速排序。