最近在忙着准备找工作，对于码农来说，找工作之前必备的就是各种的排序算法，其中包括算法实现、复杂度分析。于是我也开始研究各种排序算法，但是看完几遍之后发现，其原理并不复杂，于是就在思考，这些算法这么重要，那么它们在实际解决问题时如何来使用呢？这篇文章我就个人的理解，尽量形象、简单的描述各种基本排序算法的原理，并对其复杂度进行了相应的分析，最后说明其在实际使用中的应用。我希望我的这篇文章尽量的通俗形象，让我们展开想象的空间吧！

一、算法原理分析：

1、冒泡排序：

首先从它的名字上来理解它的原理，假设一个湖底从左到右并排了 10(n)个泡泡，我们在湖底从左向右游动，每次只比较相邻的两个泡泡，如果左边的比右边的小则交换其位置，这样游玩了一趟之后，即比较了 9（n-1）次，最大的那个就得到了，这时我们将这个泡泡冒出水面，以此类推，我们需要这样重复 9（n-1）次即可（考虑最后只剩下两个），每一次需要比较（n-i）次,j 为跑了的次数。

伪代码如下：

```
for i = 1:n-1 (n-1 趟)
    for j = 1:n-i（比较 n-i 次）
        if a[j] > a[j + 1]
            swap(a[j],a[j+1])
        endif
    endfor
endfor
```

当然由于数组元素从 0 开始，则编程时只需要将循环的首尾均减一即可。

复杂度分析：本算法共有两个循环，外循环共(n-1)个，内循环共（n-i）个，则总数为（n-1）+(n-2)+...+1 = [n*（n-1)]/2=O(n^2).

2、插入排序：

本算法也从其名字来理解，这次我们想象一下学生们排队好了，每个学生的身高都不同，现在希望他们能够快速的按照从矮到高的顺序排好队。我们的策略是插入的人都和队伍中的人挨个比，如果比队伍中的人矮，那就将队伍中的人向后移动一个位置，一直到合适的位置将其插入进去。这时候我们假设第一个元素已经在数组中了（因为没人和他比），这时我们只需要将剩下的 n-1 个人插入到队伍中就好了。假设队伍中已经插入了 i 个人，则第 i+1 个人需要和队伍中的 i 个人比才可以。

```
for i = 2:n  //插入第 i 个人
    key = a[i];  //设其为关键字用来跟每一个位置进行比较
    j = i-1;
    for j = i-1:1 //关键字和前面 i-1 个人比较
       if a[j] > key //大于关键字则将其向后移动一个，空一个空出来
        a[j+1] = a[j];
      else   //如果 key 大于队伍中的值时就退出
        break;
```

```
        endif
      endfor
      a[j+1] = key;//将其插入到该值的后面
   endfor
```

复杂度分析：该算法也包含两层循环，外层有 n-1 个元素，内层有 i-1 个元素，则共有 1+2+...+n-1 = [n*(n-1)]/2 = O(n^2)

稍微休息一下，纵观以上两个算法，虽然算法不同，但是都得需要循环比较，在相对有序时，快速排序的内层循环可以快速退出，因此其效果相对好一点，而冒泡排序还是从头比较。

3、归并排序：

看了两个循环嵌套的算法，下面看看递归的算法吧。归并排序是典型的具有归并性质的算法。先来说说它的思想，归并，换言之就是合并，既然要合并那就得先把原来的数组分开吧。咱们还是接着上面的例子，我们先将队伍分成两组，然后对两组排序，排序后进行合并。分成的两组其实又可以继续分为两组并合并，因此此处就可以使用递归的方法。

```
Merge://合并的代码
input:a[]
n1 = med；
n2 = n-med;
L[] = a[1:n1];
R[] = a[med+1:n];
L[n1+1] = 65535;//哨兵牌，每次到这之后只将剩下的那一摞放到数组中即可
R[n2+1] = 65535；

i = 1:n1;
j = 1:n2;
 for k = 1:n
   if(L[i] < R[j])
      a[k] = L[i]
      i++;
   else
      a[k] = R[j]
      j++;
   endif
endfor
```

```
MergeSort:
    q = n/2;
    MergeSort(a,1,q);
    MergeSort(a,q+1,n);
    Merge(a,1,q,n);
```

复杂度分析：对于递归调用的复杂度分析，要明白主方法：T（n）=aT(n/b)+f(n) 要掌握三种情况：1）如果 f(n)比 n^(log(b)a)小，则 T（n） = Ot(n^(log(b)a));2）如果 f(n)比 n^(log(b)a)大，则 T（n）= ot(f(n));3）如果 f(n)和 n^(log(b)a)差不多大，则 T（n）= ot(n^(log(b)a) * lgn)。

在此算法中，a = 2,b = 2;故 n^(log(b)a) = n = f(n),所以满足 3），故 T（n）= O(nlgn)。虽然该算法时间很快，但是需要使用两个数组来存储 L 和 R，典型的空间换取时间的手法。

4、快速排序：

听到这个名字，相必大家很快就觉得这个算法肯定很快，事实上也确实如此。该算法也采用了分而治之以及递归的思想。其主要的思想是：先随机选择一个关键人物作为标准，将比他矮的放到左边，比他高的放到右边。然后再在剩下的两组中按照此种方法进行递归。

```
Partition://分开确定 pivot 的位置，方便将其分开递归排序
    pivote = a[1];//将第一个人作为标准
    i = n+1;
    for j = tail:1
        if a[j]>=pivote
          i--;
          swap(a[i],a[j]);
          p_low++;
      endif
    endfor
    swap(a[1],a[i -1]);
return i -1;
QuickSort:
    p = Partition(a,1,n);
    QuickSort(a,1,p-1);
    QuickSort(a,p+1,n);
```

复杂度分析：类似于归并排序，该算法也将原问题分成两部分，T（n）= T(n/p)+T(n/(1-p))+n,其最好情况 T（n）=2T(n/2)+n=O(nlgn),最坏的情况是 T（n）= T(n-1)+O(n),T(n) = O(n^2);而对于不均分的情况，分析得出 T（n）=O(nlgn)。

该算法虽然和快速排序算法的复杂度一样，但是该算法在基本排好序的情况下属于最坏的情况，因此使用时需谨慎。

5、堆排序：

下面就来说一说堆排序，堆排序相对于上面几种排序稍微麻烦一点，但是只要掌握其精髓，也是很容易理解的。这个举实际的例子确实不太好举，但是大家可以想象出一棵二叉树。

堆排序，主要利用了最大堆和最小堆，其本质就是棵二叉树。

该排序过程大概可以分成如下三步：1）建立一个二叉树　2）将堆变为最大堆（即将最大值转移到堆顶），从最后一个非叶子节点开始　3）将堆顶元素和最后一个元素互换，然后调整堆，使其满足最大堆（堆长度在变）

1）此处不需要什么操作，只是提醒大家数组中的元素在树中的位置为前序遍历存储（根左右）

2）

```
BuildMaxHeap(a,n):
    第一个非叶子节点为：n/2
    for i = n/2:1
        HeapAjust(a,i,n);
    end
HeapAjust(a,i):调整为最大堆
    l=LEFT(i);//左子树
    r = RIGHT(i);//右子树
    if l<n && a[l] > a[i]
        largest = l;
    endif
    if r<n && a[r] >a[largest]
        largest = r;
    endif
    if largest != i
        swap(a[i],a[largest]);
        HeapAjust(a,largest);
    endif
HeapSort(a,n)
    BuildMaxHeap(a,n);
    for i  = n:2
        swap(a[1],a[i];
        HeapAjust(a,1,i-1);
    endfor
```

复杂度分析：该算法主要包括两部分，第一部分是堆调整 O（lgn）,建最大堆时，共调用（n/2）次，故其为 O（nlgn）.堆排序时主要包括 O（nlgn）+ O(nlgn) = O(nlgn)

堆排序不需要大量的递归或者多维的暂存数组。这对于数据量非常巨大的序列是合适的。比如超过数百万条记录，因为快速排序，归并排序都使用递归来设计算法，在数据量非常大的时候，可能会发生堆栈溢出错误。但是快速的时间一般线性增长，但是堆排序为 nlgn 的速度。