九大排序算法

排序的定义：
输入：n 个数：a1,a2,a3,...,an
输出：n 个数的排列:a1',a2',a3',...,an'，使得 a1'<=a2'<=a3'<=...<=an'。

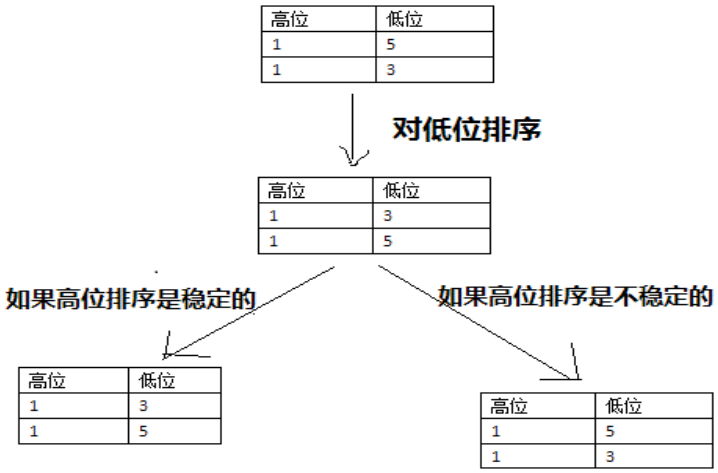
In-place sort（不占用额外内存或占用常数的内存）：插入排序、选择排序、冒泡排序、堆排序、快速排序。
Out-place sort：归并排序、计数排序、基数排序、桶排序。

当需要对大量数据进行排序时，In-place sort 就显示出优点，因为只需要占用常数的内存。
设想一下，如果要对 10000 个数据排序，如果使用了 Out-place sort，则假设需要用 200G 的额外空间，则一台老式电脑会吃不消，但是如果使用 In-place sort，则不需要花费额外内存。

stable sort：插入排序、冒泡排序、归并排序、计数排序、基数排序、桶排序。
unstable sort：选择排序(5 8 5 2 9)、快速排序、堆排序。

为何排序的稳定性很重要？

在初学排序时会觉得稳定性有这么重要吗？两个一样的元素的顺序有这么重要吗？其实很重要。在基数排序中显得尤为突出，如下：



算法导论习题 8.3-2 说：如果对于不稳定的算法进行改进，使得那些不稳定的算法也稳定？

其实很简单，只需要在每个输入元素加一个 index，表示初始时的数组索引，当不稳定的算法排好序后，对于相同的元素对 index 排序即可。

基于比较的排序都是遵循“决策树模型”，而在决策树模型中，我们能证明给予比较的排序算法最坏情况下的运行时间为Ω(nlgn)，证明的思路是因为将 n 个序列构成的决策树的叶子节点个数至少有 n!，因此高度至少为 nlgn。

线性时间排序虽然能够理想情况下能在线性时间排序，但是每个排序都需要对输入数组做一些假设，比如计数排序需要输入数组数字范围为[0,k]等。

在排序算法的正确性证明中介绍了”循环不变式“，他类似于数学归纳法，"初始"对应"n=1"，"保持"对应"假设 n=k 成立，当 n=k+1 时"。

一、插入排序

特点：stable sort、In-place sort
最优复杂度：当输入数组就是排好序的时候，复杂度为 O(n)，而快速排序在这种情况下会产生 O(n^2)的复杂度。
最差复杂度：当输入数组为倒序时，复杂度为 O(n^2)
插入排序比较适合用于“少量元素的数组”。

其实插入排序的复杂度和逆序对的个数一样，当数组倒序时，逆序对的个数为 n(n-1)/2，因此插入排序复杂度为 O(n^2)。
在算法导论 2-4 中有关于逆序对的介绍。

伪代码：

```
Insertion_Sort(A)
{
    for i=2 to n
        j = i-1
        key = A[i]
        while j>0 && A[j]>key
            A[j+1] = A[j]
            j--
        A[j+1] = key
}
```

证明算法正确性：

循环不变式：在每次循环开始前，A[1...i-1]包含了原来的 A[1...i-1]的元素，并且已排序。

初始：i=2，A[1...1]已排序，成立。
保持：在迭代开始前，A[1...i-1]已排序，而循环体的目的是将 A[i]插入 A[1...i-1]中，使得 A[1...i]排序，因此在下一轮迭代开 始前，i++，因此现在 A[1...i-1]排好序了，因此保持循环不变式。

终止：最后 i=n+1，并且 A[1...n]已排序，而 A[1...n]就是整个数组，因此证毕。

而在算法导论 2.3-6 中还问是否能将伪代码第 6-8 行用二分法实现？

实际上是不能的。因为第 6-8 行并不是单纯的线性查找，而是还要移出一个空位让 A[i]插入，因此就算二分查找用 O(lgn)查到了插入的位置，但是还是要用 O(n)的时间移出一个空位。

问：快速排序（不使用随机化）是否一定比插入排序快？

答：不一定，当输入数组已经排好序时，插入排序需要 O(n)时间，而快速排序需要 O(n^2)时间。

递归版插入排序

```
Recursive_InsertionSort(A,p,q)
{
    if p<q
        Recursive_InsertionSort(A,p,q-1);//递归将A[p...q-1]排序
        Insert(A,p,q-1);
}
Insert(A,p,q)
{
    key = A[q+1]
    j=q
    while j>0 && A[j]>key
        A[j+1]=A[j]
        j--
    A[j+1]=key
}
```

二、冒泡排序

特点：stable sort、In-place sort
思想：通过两两交换，像水中的泡泡一样，小的先冒出来，大的后冒出来。
最坏运行时间：O(n^2)
最佳运行时间：O(n^2)（当然，也可以进行改进使得最佳运行时间为 O(n)）

算法导论思考题 2-2 中介绍了冒泡排序。

伪代码：

```
1 Bubble_sort(A)
2 {
3     for i=1 to n
4         for j= n to i+1
5             if A[j]<A[j-1]
6                 swap A[j]<->A[j-1]
7 }
```

证明算法正确性：

运用两次循环不变式，先证明第 4-6 行的内循环，再证明外循环。

内循环不变式：在每次循环开始前，A[j]是 A[j...n]中最小的元素。

初始：j=n，因此 A[n]是 A[n...n]的最小元素。
保持：当循环开始时，已知 A[j]是 A[j...n]的最小元素，将 A[j]与 A[j-1]比较，并将较小者放在 j-1 位置，因此能够说明 A[j-1]是 A[j-1...n]的最小元素，因此循环不变式保持。
终止：j=i，已知 A[i]是 A[i...n]中最小的元素，证毕。

接下来证明外循环不变式：在每次循环之前，A[1...i-1]包含了 A 中最小的 i-1 个元素，且已排序：A[1]<=A[2]<=...<=A[i-1]。

初始：i=1，因此 A[1..0]=空，因此成立。
保持：当循环开始时，已知 A[1...i-1]是 A 中最小的 i-1 个元素，且 A[1]<=A[2]<=...<=A[i-1]，根据内循环不变式，终止时 A[i]是 A[i...n]中最小的元素，因此 A[1...i]包含了 A 中最小的 i 个元素，且 A[1]<=A[2]<=...<=A[i-1]<=A[i]
终止：i=n+1，已知 A[1...n]是 A 中最小的 n 个元素，且 A[1]<=A[2]<=...<=A[n]，得证。

在算法导论思考题 2-2 中又问了”冒泡排序和插入排序哪个更快“呢？

一般的人回答：“差不多吧，因为渐近时间都是 O(nˆ2)”。
但是事实上不是这样的，插入排序的速度直接是逆序对的个数，而冒泡排序中执行“交换“的次数是逆序对的个数，因此冒泡排序执行的时间至少是逆序对的个数，因此插入排序的执行时间至少比冒泡排序快。

递归版冒泡排序

```
recursive_bubblesort(A,p,q)
{
    if p<q
    {
        findmin(A,p,q); //Divide
        recursive_bubblesort(A,p+1,q);  //Conquer
    }
}
findmin(A,p,q)
{
    for i=q to p+1
        if A[i]<A[i-1]
            swap A[i]<->A[i-1]
}
```

改进版冒泡排序

最佳运行时间：O(n)
最坏运行时间：O(n^2)

```
improved_bubble_sort(A)
{
    for i=1 to n-1
        if flag==false return;
        flag=false;
        for j=n to i+1
            if A[j]<A[j-1]
                swap A[j]<->A[j-1]
                flag = true;
}
```

三、选择排序

特性：In-place sort，unstable sort。
思想：每次找一个最小值。
最好情况时间：O(n^2)。
最坏情况时间：O(n^2)。

伪代码：

```
selection_sort(A)
{
    for i=1 to n-1
        min=i;
        for j=i+1 to n
            if A[min]>A[j]
                min = j;
        swap A[min]<->A[i]
}
```

证明算法正确性：

循环不变式：A[1...i-1]包含了 A 中最小的 i-1 个元素，且已排序。

初始：i=1，A[1...0]=空，因此成立。
保持： 在某次迭代开始之前，保持循环不变式，即 A[1...i-1]包含了 A 中最小的 i-1 个元素，且已排序，则进入循环体后，程序从 A[i...n]中找出最小值放在 A[i]处，因此 A[1...i]包含了 A 中最小的 i 个元素，且已排序，而 i++，因此下一次循环之前，保持 循环不变式：A[1..i-1]包含了 A 中最小的 i-1 个元素，且已排序。
终止：i=n，已知 A[1...n-1]包含了 A 中最小的 i-1 个元素，且已排序，因此 A[n]中的元素是最大的，因此 A[1...n]已排序，证毕。

算法导论 2.2-2 中问了"为什么伪代码中第 3 行只有循环 n-1 次而不是 n 次"？

在循环不变式证明中也提到了，如果 A[1...n-1]已排序，且包含了 A 中最小的 n-1 个元素，则 A[n]肯定是最大的，因此肯定是已排序的。

递归版选择排序

```
recursive_selectionsort(A,p,q)
{
    if p<q
    {
        min=find_min(A,p,q);    //从A[p...q]中找出最小的与A[p]交换
        swap A[p]<->A[min]
        recursive_selectionsort(A,p+1,q);   //对A[p+1...q]排序
    }
}
find_min(A,p,q)
{
    min = p
    for i=p+1 to q
        if A[min]>A[i]
            min = i
    return min
}
```

递归式：

T(n)=T(n-1)+O(n)
=> T(n)=O(n^2)

四、归并排序

特点：stable sort、Out-place sort
思想：运用分治法思想解决排序问题。
最坏情况运行时间：O(nlgn)
最佳运行时间：O(nlgn)

分治法介绍：分治法就是将原问题分解为多个独立的子问题，且这些子问题的形式和原问题相似，只是规模上减少了，求解完子问题后合并结果构成原问题的解。分治法通常有 3 步：Divide（分解子问题的步骤） 、 Conquer（递归解决子问题的步骤）、 Combine（子问题解求出来后合并成原问题解的步骤）。
假设 Divide 需要 f(n)时间，Conquer 分解为 b 个子问题，且子问题大小为 a，Combine 需要 g(n)时间，则递归式为：
T(n)=bT(n/a)+f(n)+g(n)

算法导论思考题 4-3（参数传递）能够很好的考察对于分治法的理解。

就如归并排序，Divide 的步骤为 m=(p+q)/2，因此为 O(1)，Combine 步骤为 merge()函数，Conquer 步骤为分解为 2 个子问题，子问题大小为 n/2，因此：
归并排序的递归式：T(n)=2T(n/2)+O(n)

而求解递归式的三种方法有：
(1)替换法：主要用于验证递归式的复杂度。
(2)递归树：能够大致估算递归式的复杂度，估算完后可以用替换法验证。
(3)主定理：用于解一些常见的递归式。

伪代码：

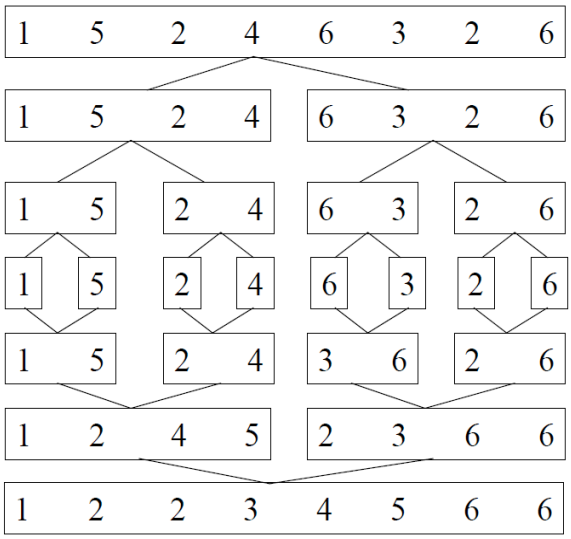
```
1  Merge_sort(A)
2  {
3      recursive_mergesort(A,1,length[A]);
4  }
5  recursive_mergesort(A,p,q)
6  {
7      if p<q
8          m = (p+q)/2
9          recursive_mergesort(A,p,m)
10         recursive_mergesort(A,m+1,q)
11         merge(A,p,m,q)
12 }
13 merge(A,p,m,q)
14 {
15     a = m-p+1
16     b = q-m
17     create array L[a+1] & R[b+1]
18     for i=1 to a
19         L[i] = A[p+i-1]
20     for i=1 to b
21         R[i] = A[m+i]
22     L[a+1] = INFINITY
23     R[b+1] = INFINITY
24     i = j = 1
25     for k = p to q
26         if L[i]<R[j]
27             A[k] = L[i]
28             i++
29         else if L[i]>R[j]
30             A[k] = R[j]
31             j++
32 }
```

证明算法正确性：

其实我们只要证明 merge()函数的正确性即可。
merge 函数的主要步骤在第 25~31 行，可以看出是由一个循环构成。

循环不变式：每次循环之前，A[p...k-1]已排序，且 L[i]和 R[j]是 L 和 R 中剩下的元素中最小的两个元素。
初始：k=p，A[p...p-1]为空，因此已排序，成立。
保持：在第 k 次迭代之 前，A[p...k-1]已经排序，而因为 L[i]和 R[j]是 L 和 R 中剩下的元素中最小的两个元素，因此只需要将 L[i]和 R[j]中最小的元素放到 A[k]即可，在第 k+1 次迭代之前 A[p...k]已排序，且 L[i]和 R[j]为剩下的最小的两个元素。
终止：k=q+1，且 A[p...q]已排序，这就是我们想要的，因此证毕。

归并排序的例子：



问：归并排序的缺点是什么？

答：他是 Out-place sort，因此相比快排，需要很多额外的空间。

问：为什么归并排序比快速排序慢？

答：虽然渐近复杂度一样，但是归并排序的系数比快排大。

问：对于归并排序有什么改进？

答：就是在数组长度为 k 时，用插入排序，因为插入排序适合对小数组排序。在算法导论思考题 2-1 中介绍了。复杂度为 O(nk+nlg(n/k))，当 k=O(lgn)时，复杂度为 O(nlgn)

五、快速排序

Tony Hoare 爵士在 1962 年发明，被誉为“20 世纪十大经典算法之一”。
算法导论中讲解的快速排序的 PARTITION 是 Lomuto 提出的，是对 Hoare 的算法进行一些改变的，而算法导论 7-1 介绍了 Hoare 的快排。
特性：unstable sort、In-place sort。
最坏运行时间：当输入数组已排序时，时间为 O(n^2)，当然可以通过随机化来改进（shuffle array 或者 randomized select pivot），使得期望运行时间为 O(nlgn)。
最佳运行时间：O(nlgn)
快速排序的思想也是分治法。
当输入数组的所有元素都一样时，不管是快速排序还是随机化快速排序的复杂度都为 O(n^2)，而在算法导论第三版的思考题 7-2 中通过改变 Partition 函数，从而改进复杂度为 O(n)。

注意：只要 partition 的划分比例是常数的，则快排的效率就是 O(nlgn)，比如当 partition 的划分比例为 10000:1 时（足够不平衡了），快排的效率还是 O(nlgn)

“A killer adversary for quicksort”这篇文章很有趣的介绍了怎么样设计一个输入数组，使得 quicksort 运行时间为 O(n^2)。

伪代码：

```
quick_sort(A)
{
    recursive_quicksort(A,1,length[A])
}
recursive_quicksort(A,p,q)
{
    if p<q
        r = partition(A,p,q)
        recursive_quicksort(A,p,r-1)
        recursive_quicksort(A,r+1,q)
}
partition(A,p,q)
{
    i = p-1
    pivot = A[q]
    for j=p to q-1
        if A[j]<=pivot
            i++
            swap A[i]<->A[j]
    swap A[i+1]<->A[q]
    return i+1
}
```

随机化 partition 的实现：

```
randomized_partition(A,p,q)
{
    n = q-p+1
    gap = new Random().next(n) //[0~n-1]中随机取一个数
    swap A[q] <->A[p+gap]
    return partition(A,p,q)
}
```

改进当所有元素相同时的效率的 Partition 实现：

```
partition(A,p,q)
{
    i = p-1
    k = q
    pivot = A[q]
    for j=p to q-1
        if A[j]<pivot
            i++
            swap A[i]<->A[j]
        else if A[j]==pivot
            k--
            swap A[j]<->A[k]
    i = i + 1
    for t = q to k
        if i<k
            swap A[i]<->A[t]
            i++
    return i
}
```

证明算法正确性：

对 partition 函数证明循环不变式：A[p...i]的所有元素小于等于 pivot，A[i+1...j-1]的所有元素大于 pivot。
初始：i=p-1,j=p，因此 A[p...p-1]=空，A[p...p-1]=空，因此成立。
保持：当循环开始前，已知 A[p...i]的所有元素小于等于 pivot，A[i+1...j-1]的所有元素大于 pivot，在循环体中，

- 如果 A[j]>pivot，那么不动，j++，此时 A[p...i]的所有元素小于等于 pivot，A[i+1...j-1]的所有元素大于 pivot。
- 如果 A[j]<=pivot，则 i++，A[i+1]>pivot，将 A[i+1]和 A[j]交换后，A[P...i]保持所有元素小于等于 pivot，而 A[i+1...j-1]的所有元素大于 pivot。

终止：j=r，因此 A[p...i]的所有元素小于等于 pivot，A[i+1...r-1]的所有元素大于 pivot。

六、堆排序

1964 年 Williams 提出。

特性：unstable sort、In-place sort。
最优时间：O(nlgn)
最差时间：O(nlgn)
此篇文章介绍了堆排序的最优时间和最差时间的证明：
http://blog.csdn.net/xiazdong/article/details/8193625
思想：运用了最小堆、最大堆这个数据结构，而堆还能用于构建优先队列。

优先队列应用于进程间调度、任务调度等。
堆数据结构应用于 Dijkstra、Prim 算法。

```
MAX_HEAPIFY(A,i)
{
    heapsize = heapsize[A];
    largest = i;
    if left(i)<= heapsize && A[largest]<A[left(i)]
        largest = left(i);
    else if right(i)<= heapsize && A[largest]<A[right(i)]
        largest = right(i);
    if(largest!=i)
        swap A[i]<->A[largest];
        MAX_HEAPIFY(A,largest);
}
build_max_heap(A)
{
    for i=floor(n/2) to 1
        MAX_HEAPIFY(A,i);
}
heapsort(A)
{
    build_max_heap(A);
    for i=n to 2
        swap A[1]<->A[heapsize]
        heapsize--;
        MAX_HEAPIFY(A,1);
}
```

证明算法正确性：

（1）证明 build_max_heap 的正确性：
循环不变式：每次循环开始前，A[i+1]、A[i+2]、...、A[n]分别为最大堆的根。

初始：i=floor(n/2)，则 A[i+1]、...、A[n]都是叶子，因此成立。
保持：每次迭代开始前，已知 A[i+1]、A[i+2]、...、A[n]分别为最大堆的根，在循环体中，因为 A[i]的孩子的子树都是最大堆，因此执行完 MAX_HEAPIFY(A,i)后，A[i]也是最大堆的根，因此保持循环不变式。
终止：i=0，已知 A[1]、...、A[n]都是最大堆的根，得到了 A[1]是最大堆的根，因此证毕。

（2）证明 heapsort 的正确性：
循环不变式：每次迭代前，A[i+1]、...、A[n]包含了 A 中最大的 n-i 个元素，且 A[i+1]<=A[i+2]<=...<=A[n]，且 A[1]是堆中最大的。

初始：i=n，A[n+1]...A[n]为空，成立。
保 持：每次迭代开始前，A[i+1]、...、A[n]包含了 A 中最大的 n-i 个元素，且 A[i+1]<=A[i+2]<=...& lt;=A[n]，循环体内将 A[1]与 A[i]交换，因为 A[1]是

堆中最大的，因此 A[i]、...、A[n]包含了 A 中最大的 n-i+1 个元素且 A[i]<=A[i+1]<=A[i+2]<=...<=A[n]，因此保持循环不变式。
终止：i=1，已知 A[2]、...、A[n]包含了 A 中最大的 n-1 个元素，且 A[2]<=A[3]<=...<=A[n]，因此 A[1]<=A[2]<=A[3]<=...<=A[n]，证毕。

七、计数排序

特性：stable sort、out-place sort。
最坏情况运行时间：O(n+k)
最好情况运行时间：O(n+k)

当 k=O(n)时，计数排序时间为 O(n)

伪代码：

```
counting_sort(A,k)
{
    n = length[A]
    create array B[n] & C[k+1]
    for i=0 to k
        C[i] = 0
    for i=1 to n
        C[A[i]]++
    for i=1 to k
        C[i] = C[i] + C[i-1]
    for i=n to 1
        B[C[A[i]]] = A[i]
        C[A[i]]--
}
```

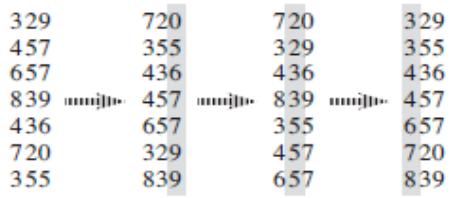
八、基数排序

本文假定每位的排序是计数排序。
特性：stable sort、Out-place sort。
最坏情况运行时间：O((n+k)d)
最好情况运行时间：O((n+k)d)

当 d 为常数、k=O(n)时，效率为 O(n)
我们也不一定要一位一位排序，我们可以多位多位排序，比如一共 10 位，我们可以先对低 5 位排序，再对高 5 位排序。
引理：假设 n 个 b 位数，将 b 位数分为多个单元，且每个单元为 r 位，那么基数排序的效率为 O[(b/r)(n+2^r)]。
当 b=O(nlgn)，r=lgn 时，基数排序效率 O(n)

比如**算法导论习题 8.3-4**：说明如何在 O(n)时间内，对 0~n^2-1 之间的 n 个整数排序？
答案：将这些数化为 2 进制，位数为 lg(n^2)=2lgn=O(lgn)，因此利用引理，b=O(lgn)，而我们设 r=lgn，则基数排序可以在 O(n)内排序。

基数排序的例子：




```
radix_sort(A,d,k)
{
    for i=1 to d
        counting_sort(A,i,k)
}
```

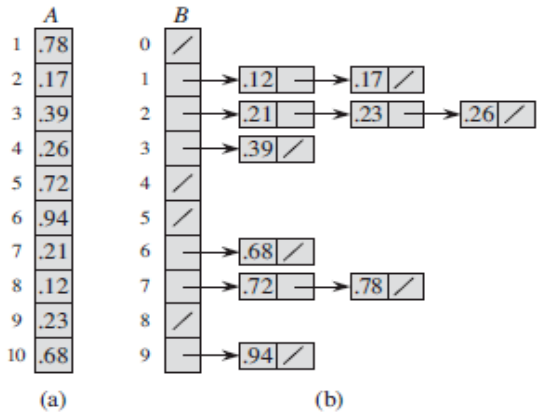
证明算法正确性：

通过循环不变式可证，证明略。

九、桶排序

假设输入数组的元素都在[0,1)之间。
特性：out-place sort、stable sort。
最坏情况运行时间：当分布不均匀时，全部元素都分到一个桶中，则 O(n^2)，当然[算法导论 8.4-2]也可以将插入排序换成堆排序、快速排序等，这样最坏情况就是 O(nlgn)。
最好情况运行时间：O(n)

桶排序的例子：



伪代码：

```
bucket_sort(A)
{
    n = length[A]
    create buckets B[n]
    for i=1 to n
        insert A[i] to B[nA[i]]
    for i=0 to n-1
        sort B[i] with insertion sort (can also by quicksort)
    concatenate B[]
}
```

证明算法正确性：

对于任意 A[i]<=A[j]，且 A[i]落在 B[a]，A[j]落在 B[b]，我们可以看出 a<=b，因此得证。