经典排序算法分析和代码-中篇

上篇文章中我们讨论插入排序、谢尔排序和堆排序，这章我们接着讲其他的几种排序。

4.归并排序

归并排序以$O(N \log N)$最坏情形的运行时间运行，而所使用的比较次数几乎是最优的。

它是递归算法的一个很好的实例。

这个算法中的基本操作是合并两个已排序的表。因为这两个表是已排序的，所以若将输出放到第三个表中，则该算法

可以通过对输入数据进行一趟排序来完成。

比如有两个数组　　1 3 5 7　　和 2 4 6 8将他们合并到其三个数组中，第一次1和2相比较，1比较小，则将1放到第三个数组中

1 3 5 7　　**2** 4 8 10　　1

再将将3和2比较得到

1 **3** 5 7　　**2** 4 8 10　　1 2

3和4开始比较得到

1 **3** 5 7　　2 **4** 8 10　　1 2 3

5和4比较得到

1 3 **5** 7　　2 **4** 8 10　　1 2 3 4

5和8比较得到

1 3 **5** 7　　2 4 **8** 10　　1 2 3 4 5

7和8比较得到

1 3 5 **7**　　2 4 **8** 10　　1 2 3 4 5 7

第一个数组使用完了，我们将第二个数组剩余的 8和10拷贝到第三个数组尾部得到

1 2 3 4 5 7 8 10

这样一次合并就完成了。

可以看到合并的时间显然是线性的，因为最多进行了 N-1次比较，其中 N 是元素总数。

我们可以看到每一次比较总会将小的数放大第三个数组中 ,但是最后一次比较至少可以直接添加两个元素到

第三个数组中。

因此归并算法很容易描述，如果 N=1，那么只有一个元素需要排序，我们排序直接完成，然后直接将前半部分和

后半部分进行合并,得到元素为2的有序数组，再对元素为2的相邻数组进行合并，如此递归进行，得到最后的有序数组。

下面直接看代码对照理解:

```
#include <stdio.h>

/**
*合并两个有序数组
*/
void Merge(int Array[], int tmpArray[], int left, int mid, int right)
{
    int i = left;
    int tmpleft = left;
    int tmpright = mid+1;
    //比较左右数组元素大小，放入第三个数组中
    while (tmpleft <= mid && tmpright <= right)
    {
        if (Array[tmpleft] < Array[tmpright])
            tmpArray[i++] = Array[tmpleft++];
        else
            tmpArray[i++] = Array[tmpright++];
    }
    //右边数组用完了进入此循环拷贝左边数组到第三个数组
    for (; tmpleft <= mid; tmpleft++)
    {
        tmpArray[i++] = Array[tmpleft];
    }
    //左边数组用完了进入此循环拷贝右边数组到第三个数组
    for (; tmpright <= right; tmpright++)
    {
        tmpArray[i++] = Array[tmpright];
    }
    //合并后的数组拷贝回原数组
    for (i = left; i <= right; i++)
```

```
    {
        Array[i] = tmpArray[i];
    }
}

/**
  *归并排序
  */
void MergeSort(int Array[], int tmpArray[], int left, int right)
{
    //基准条件，如果只有一个元素即为有序数组，直接返回
    if (left >= right)
        return;
    //递归数组排序左右两边数组
    int mid = (left + right) / 2;
    MergeSort(Array, tmpArray, left, mid);
    MergeSort(Array, tmpArray, mid + 1, right);
    //排序完成后进入和并数组
    Merge(Array, tmpArray, left, mid, right);
}

int main()
{
    int a[] = { 1, 0, 2, 9, 3, 8, 4, 7, 5, 6 };
    int tempa[10];
    MergeSort(a, tempa, 0, 9);
    for (int i = 0; i < 10; i++)
    {
        printf("%d \n", tempa[i]);
    }
    return 0;
}
```

虽然归并排序的运行时间是$O(N \log N)$，但是它很难用于主存排序，主要问题在于合并两个排序的表需要线性附加内存，

在整个算法中还要花费将数据复制到临时数组再复制回来这样的一些附加工作，其结果是严重减慢了排序的速度。这种复制可以

通过在递归的交替层面上审慎的交换 Array 和 tmpArray 的角色加以避免。归并排序的一种变形也可以非递归的实现。

与其他的$O(N \log N)$排序相比，归并排序的运行时间很大程度上依赖于在数组中进行元素的比较和移动所消耗的时间。

这些消耗是和编程语言相关的。

例如，在其他语言(例如 Java)中，当排序一般的对象时，元素的比较耗时很多，但是移动元素就快得多。在所有流行的排序算法中，归并排序使用最少次数比较。因此，在 Java 中，归并排序是一般目的的排序的最佳选择。事实上，在标致 Java 库中的一般排序就是用的这种算法。

另一方面，在 C++中，对于一般排序，当对象很大时，赋值对象的代价是很大的，二对象比较通常相对消耗小些。这是因为编译器在处理函数模板的扩展时具有强大的执行在线优化的能力。在本节中，如果我们可以使用很少的数据移动，那么即使使用稍微多一些比较的算法也是合理的。下面介绍的快速排序算法较好的平衡了这两者，而且也是 C++库中普遍使用的排序历程。

5.快速排序

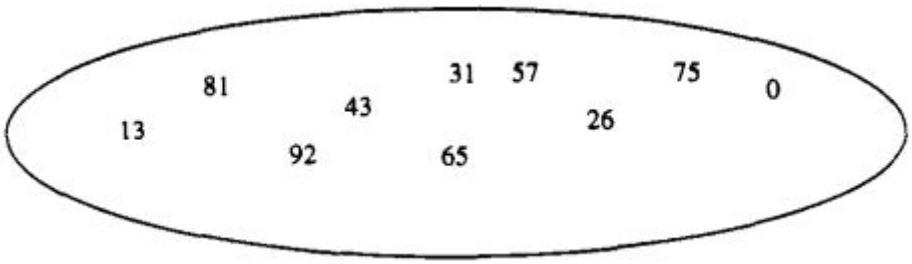
顾名思义，快速排序(quicksort)是在实践中最快的一直排序算法，它的平均运行时间是$O(N \log N)$。改算法之所以特别快，主要是由于非常精炼和高度优化的内部循环。它的最坏情形的性能为$O(N^2)$，但稍加努力就可避免这种情形。通过将堆排序与快速排序结

合起来，就可以在堆排序的$O(N \log N)$最坏运行时间下，得到对几乎所有输入的最快运行时间。

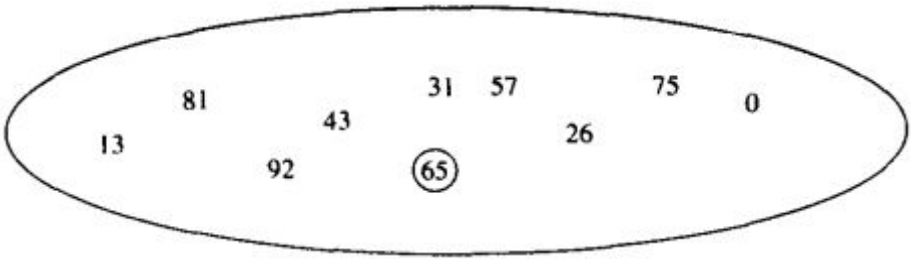
虽然多年来快速排序算法曾被认为是理论上高度优化而在实践中不可能正确编程的一种算法，但是该算法简单易懂而且不难证明。想归并排序一样，快速排序也是一种分治的递归算法。

下面通过实例分析快速排序:

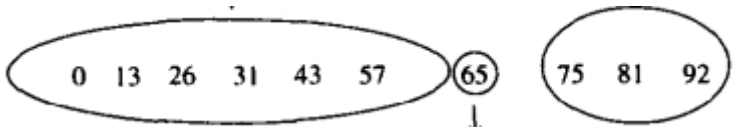
在待排序的数组中：



随机的选择一个枢纽元(pivot) 65:

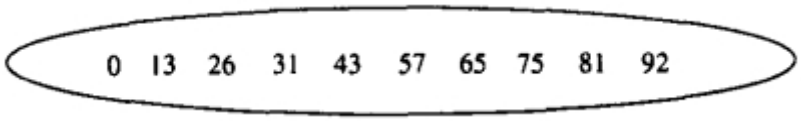


然后进行划分，将比枢纽元65小的放到左边，其他的放到右边:



那么一次快速排序完成，然后递归的对左边数组和右边数组分别进行快速排序。

最后完成后数组即成为有序:



下面我们给出一种快速排序的算法代码:

```
63 #include <stdio.h>
64
65 //分为两个区，左边的数都比右边的数小，
66 //返回值为中间数所在的位置
67 int Partition(int Nums[], int left, int right)
68 {
69     int midNum = Nums[right];//定数组最后一个数位中间数
70     int j = left;
71     for (int i = left; i < right; i++)//循环比较第 i 个数和中间数
72     {
73         if (Nums[i] < midNum)//如果小于中间数的，j 指针就向后移动，
74             //j 指针之前的数都小于中间数
75         {
76             if (i != j)
77             {
78                 int temp = Nums[i];
79                 Nums[i] = Nums[j];
80                 Nums[j] = temp;
81             }
82             j++;
83         }
```

```
84      }
85      //收尾工作，将中间数和 j 指向的中间位置的数相调换
86      int temp = Nums[j];
87      Nums[j] = midNum;
88      Nums[right] = temp;
89
90      return j;//返回中间数的下标
91  }
92
93  void QuickSort(int Nums[], int left, int right)
94  {
95      if (left < right)
96      {
97          int mid = Partition(Nums, left, right);//分区
98          QuickSort(Nums, left, mid - 1);//左部分递归排序
99          QuickSort(Nums, mid + 1, right);//右部分递归排序
100     }
101 }
102
103 int main()
104 {
105     int a[] = { 1, 0, 2, 9, 3, 8, 4, 7, 5, 6 };
106     QuickSort(a, 0, 9);
107     for (int i = 0; i < 10; i++)
108     {
109         printf("%d \n", a[i]);
110     }
111     return 0;
112 }
```

任何只使用比较的一般排序算法在最坏情形下需要$O(N \log N)$时间，但是在某些特殊情况下以线性时间进行

排序仍然是可能的，下一篇我们将介绍不是基于比较的排序。