QuickSort 实验报告

一、实验目的

通过实践，加深理解快速排序的性质、特点，熟悉其相关操作，提高编程能力。

二、实验内容

1. 编程实现快速排序与插入排序相结合的排序算法，即当对一个长度小于 k 的子数组调用快速排序算法时，让它不做任何处理就返回。上层的快速排序返回后，对整个数组运行一次插入排序来完成排序过程。

2. 随机生成大小为 1w, 10w, 100w, 1000w, …… 的数据集，测试算法的执行时间，并与库函数(例如：C++中 的 sort()函数)进行比较。、

三、实验步骤

1.快速排序算法介绍及实现

（1）快速排序算法介绍：

快速排序算法采用分治思想，平均情况运行时间为$\theta(nlgn)$，最坏情况运行时间为$\theta(n^2)$.下面是对一个典型子数组 A[p..r]进行快速排序的三步分治过程：

分解：数组 A[p..r]被划分为两个（可能为空）的子数组 A[p..q-1]和 A[q+1..r]，使得 A[p..q-1]中的每一个元素都小于等于 A[q]，而 A[q]也小于等于 A[q+1..r]中的每个元素。其中，计算下标也是划分的一部分。

解决：通过递归调用快速排序，对子数组 A[p..q-1]和 A[q+1..r]进行排序。

合并：因为子数组都是原址排序的，所以不需要合并操作，数组 A[p..r]已经有序。

（2）快速排序算法实现：

Quicksort 函数为快速排序算法的主过程，Input 数组为输入的待排序子数组，low 和 high 标记了数组的上下边界，k 为递归结束长度，当 high-low+1(子数组长度)小于 k 时停止递归，否则快排算法继续往深层递归：先调用 Partition 函数对子数组进行划分，再分别对划分之后的左右子子数组进行递归。

```
51  void QuickSort(vector<int>& Input, int low, int high, int k)  //快排
52  {
53      if(high-low+1 >= k) //判断输入数组的长度是否小于k
54      {
55          int pivot = 0;
56          pivot = Partition(Input, low, high);
57          QuickSort(Input, low, pivot-1, k);
58          QuickSort(Input, pivot+1, high, k);
59      }
60  }
```

Partition 函数为划分函数，它实现了对 Input[low..high]的原址重排，变量 x 存储着划分主元，迭代过程中 low 与指针 i 之间的元素均小于等于 x，i+1 与 j-1 之间的元素均大于 x，其实现如下：

```
32  int Partition(vector<int>& Input, int low, int high) //划分
33  {
34      int x = Input[high];
35      int i = low-1;
36      int j = low;
37      while(j<high)
38      {
39          if(Input[j] < x)
40          {
41              i++;
42              Exchange(Input[i], Input[j]);
43          }
44          j++;
45      }
46      Exchange(Input[i+1], Input[j]);
47      return i+1;
48  }
```

2. 插入排序算法介绍及实现

（1）插入排序算法介绍：

插入排序最好情况下运行时间为θ(n),平均和最坏情况下运行时间为θ(n^2).对输入子数组 Input 进行 n-1 次迭代（n 为数组长度），每次迭代过程中，将元素 A[i]移动到它在数组中的正确位置，并使得 A[0..i]子子数组中的元素均按递增排序。

（2）插入排序算法实现：

```
11  void InsertSort(vector<int>& Input) //插入排序
12  {
13      for(int i=1; i<Input.size(); i++)
14      {
15          if(Input[i]<Input[i-1])
16          {
17              int temp = Input[i];
18              int j;
19              for( j = i-1; j>=0&&Input[j]>temp; j--)
20                  Input[j+1] = Input[j];
21              Input[j+1] = temp;
22          }
23      }
24  }
```

3.计算排序时间

如下过程为对调用 STL 库中 sort 函数进行排序的运行时间的计算：QueryPerformanceCounter()函数返回高精确度性能计数器的值,它可以以微秒为单位计时。但是 QueryPerformanceCounter()确切的精确计时的最小单位是与系统有关的,所以,必须要查询系统以得到 QueryPerformanceCounter() 返回的嘀哒声的频率。QueryPerformanceFrequency()提供了这个频率值,返回每秒嘀哒声的个数.计算确切的时间是从第一次调用 QueryPerformanceCounter()开始的。假设得到的 LARGE_INTEGER 为 t1，过一段时间后再次调用该函数结束的，设得到 t2。两者之差除以 QueryPerformanceFrequency()的频率就是开始到结束之间的秒数。计时函数本身的开销很小，可忽略不计。

```
LARGE_INTEGER t11,t22,tcc;
QueryPerformanceFrequency(&tcc);
QueryPerformanceCounter(&t11);
/************排序*********/
sort(ite1, ite2);
/************************/
QueryPerformanceCounter(&t22);
time2=(double)(t22.QuadPart-t11.QuadPart)/(double)tc.QuadPart;
cout<<"library sorting-algorithm:"<<endl;
cout<<"time = "<<time2<<"s"<<endl;  //输出时间（单位：s）
```

四、实验结果

程序分别构建了大小为 1w、10w、100w、1000w、10000w 的数据集进行测试，与 STL 中的 sort 函数运行时间相比较，测试结果如下（其中 ArraySize 为待排序数组长度）：

ArraySize = 1w 时：

```
My sorting algorithm:
time = 0.0020669s
library sorting-algorithm:
time = 0.00246s
```

ArraySize = 10w 时：

```
My sorting algorithm:
time = 0.0366767s
library sorting-algorithm:
time = 0.0481808s
```

ArraySize = 100w 时：

```
My sorting algorithm:
time = 0.318147s
library sorting-algorithm:
time = 0.389379s
```

ArraySize = 1000w 时：

```
My sorting algorithm:
time = 7.03871s
library sorting-algorithm:
time = 3.94306s
```

ArraySize = 10000w 时：

```
My sorting algorithm:
time = 466.935s
library sorting-algorithm:
time = 44.7871s
```

五、实验心得

通过更改 ArraySize 的大小，并比较快速排序与插入排序相结合的排序算法与 STL 库中的 sort 算法的运行时间，不难发现当待排序数组大小小于等于10^6数量级时，使用前者排序较快，当待排序数组大小等于10^7数量级时，前者运行时间接近是后者的两倍多，当待排序数组大小等于10^8数量级时，前者运行时间已然是后者的 10 倍多！说明 STL 库中的 sort 算法在应用于实际场景中（待排序的数据量很大）性能确实很优良的，其源代码编写者在设计算法时肯定做了很多优化，其性能不是我们简简单单地通过将几个排序算法相结合就可以超过的。

这也告诉我们阅读一些优秀源代码的重要性，我们在学习过程中确实有必要自己动手去重复造一些轮子，但也不能闭门造车。