湖南大學

排序算法
实验报告

姓名：杨杰
学号：201908010705
班级：计科 1907

实验 9 排序算法比较

一、排序的概述

定义：

一种常见操作，将一组“无序”的记录序列调整为按关键字“有序”的记录序列。

形式化定义：

设含n个记录的序列为{“R” _“1” ， “R” _**2** ，……， “R” ***n}***，*其相 应的关键字序列为{**K**_1 ， **K**_**2** ，……， **K**_***n***}*，则排序是确定 这个n个记录的一种排列 Rp1，Rp2，……Rpn，使得各记录按关键字有序，即：Kp1≤Kp2 ≤ … ≤ Kpn

分类：

（1）按排序涉及的存储器可分内部排序和外部排序

1.若待排序的记录数不很大，整个排序过程不需要访问外存便能完成，则称此类排序问题为内部排序。

2.若待排序的记录数很大，整个排序过程不可能都在内存中完成，需要访问外存，则称此类排序问题为外部排序。

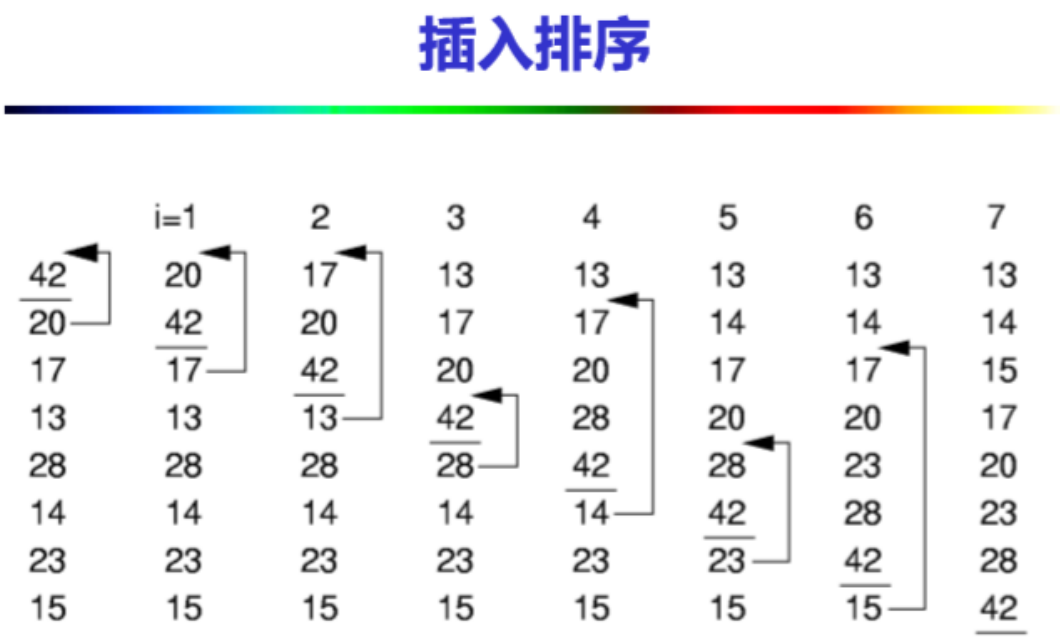
（2）按照排序性能可分为：

简单排序	选择排序	插入排序	冒泡排序	
经典排序	Shell 排序	归并排序	快速排序	堆排序
特殊排序	分配排序	基数排序		

二、排序算法

1. 简单排序:

（1）插入排序



https://blog.csdn.net/weixin_44307065

算法如下：[1]

```
    template <typename E, typename Comp> void inssort(E A[], int n)
    {
      for (int i=1; i<n; i++)//从第 2 个记录开始插入
          for (int j=i; (j>0)&&(Comp::prior(A[j], A[j-1])); j--)
          swap(A,j,j-1);//交换当前记录与前一记录
    }
```

优化：加入监视哨

```
    template < typename E, typename Comp > void inssort(E A[], int n)
    {
      for (int i=2; i<=n; i++)
      {
          A[0]=A[i];
          j=i-1;//设置监视哨 A[0]
          while (Comp::prior(x, A[j])) //把比 A[i]大的数据逐一后移
          {
              A[j+1]=A[j];
              j--;
          }
          A[j+1]=A[0];
```

```
        }
    }
```

性能分析：

最好情况下（关键字在记录中顺序有序）

比较次数：∑(i=2,n) 1=n-1

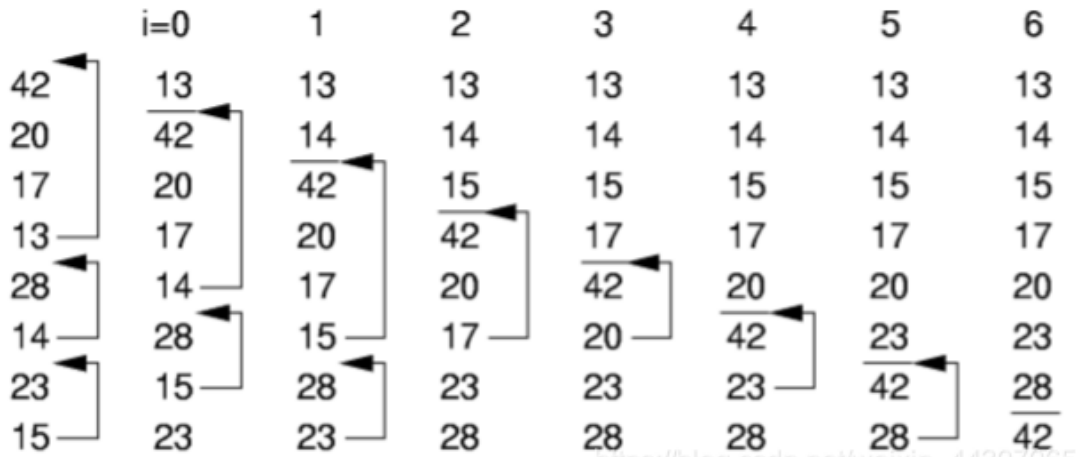
移动次数：0

最坏情况（关键字在记录中逆序有序）

比较次数：∑(i=2,n) （i-1）=n(n-1)/2

移动次数：∑(i=2,n) （i+1）=（n+4）(n-1)/2

（2）冒泡排序



https://blog.csdn.net/weixin_44307065

算法如下：

```
template < typename E, typename Comp >
void bubsort(E A[], int n)
{
    for (int i=0; i<n-1; i++)
        for (int j=n-1; j>i; j--)
            if (Comp::prior(A[j], A[j-1]))
            swap(A, j, j-1);
}
```

优化：[2]

```
template < typename E, typename Comp >
void bubsort(E A[], int n)
{
```

```
    int flag;
    for (int i=0; i<n-1; i++)
    {
        flag=FALSE; //加标志位
        for (int j=n-1; j>i; j--)
        if (Comp::prior(A[j], A[j-1]))
        {
            swap(A, j, j-1);
            flag=TRUE;
        }
        if(flag==FALSE) return;//发现已经全部有序了
    }
}
```

性能分析：

最好情况下（关键字在记录中顺序有序）

比较次数：n-1

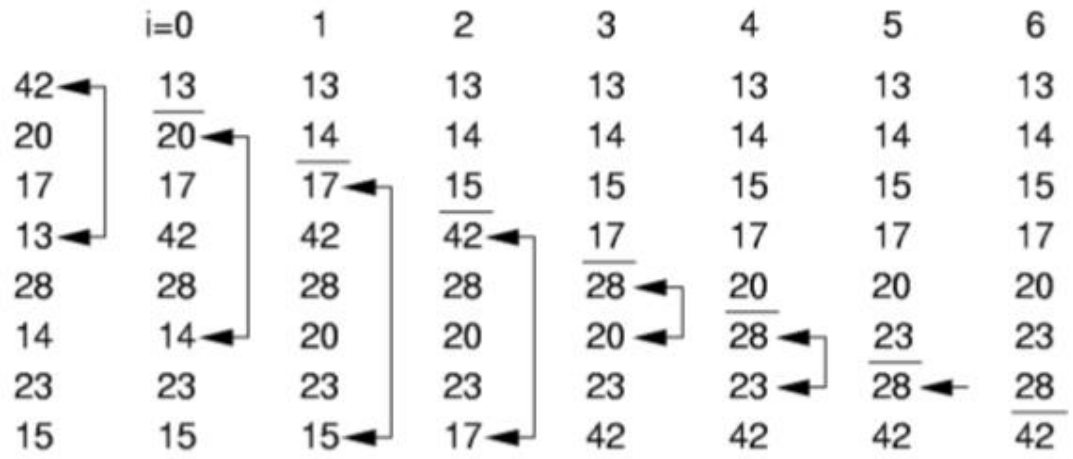
移动次数：0

最坏情况（关键字在记录中逆序有序）

比较次数：∑(i=n,2) （i-1）=n(n-1)/2

移动次数：3*∑(i=n,2) （i-1）=3n(n-1)/2

（3）直接选择排序



https://blog.csdn.net/weixin_44307065

算法如下：

```
template < typename E, typename Comp >
void selsort(E A[], int n)
{
    for (int i=0; i<n-1; i++)
    {
        int lowindex = i; // Remember its index
        for (int j=n-1; j>i; j--) // Find least
        if (Comp::prior(A[j], A[lowindex]))
        lowindex = j; // Put it in place
        swap(A, i, lowindex);
    }
```

性能分析：

比较次数：∑(i=1,n-1) （n-i）=n(n-1)/2

移动次数：最小为 0，最大为 3(n-1)

时间代价：

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

https://blog.csdn.net/weixin_44307065

2. 经典排序：

(1)Shell 排序

ν 又称缩小增量排序法

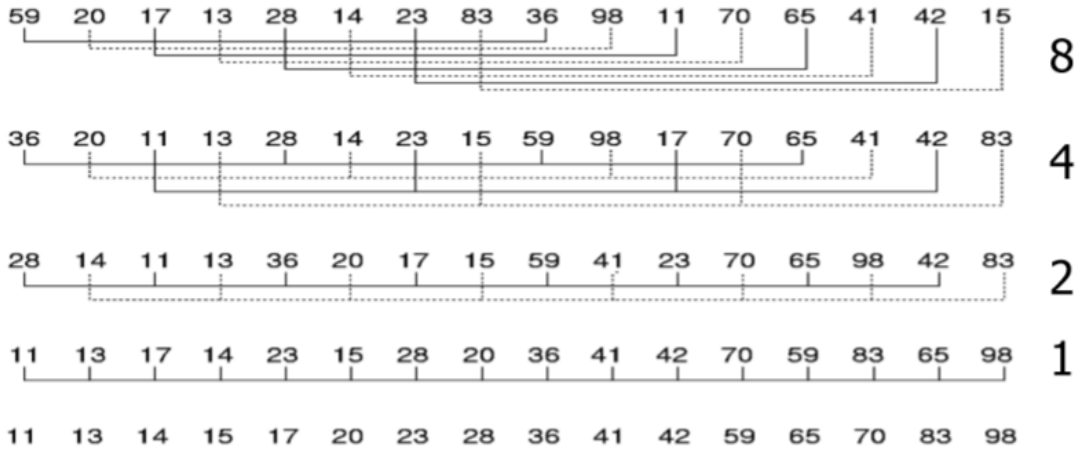
ν 与交换排序不同的是，shell 排序是在不相邻的记录之 间进行比较与交换

ν n 很小时，或基本有序时排序速度较快

ν 基本原理

ν 利用插入排序的最佳时间代价特性，先对所有记录按增量进 行分组，组内进行插入排序；

ν 减少增量重复上面步骤直至增量为 1 时止[3]



https://blog.csdn.net/weixin_44307065

算法实现：

```
template < typename E, typename Comp >
void inssort2(E A[], int n, int incr)
{//子序列插入排序
    for (int i=incr; i<n; i+=incr)
        for (int j=i; (j>=incr) && (Comp::prior(A[j], A[j-incr])); j-=incr)
        swap(A, j, j-incr);
}
template < typename E, typename Comp >
void shellsort(E A[], int n)
{ // Shellsort
    for (int i=n/2; i>2; i/=2)    // For each incr
        for (int j=0; j<i; j++)      // Sort sublists
        inssort2<E,Comp>(&A[j], n-j, i);
        inssort2<E,Comp>(A, n, 1);
}
```

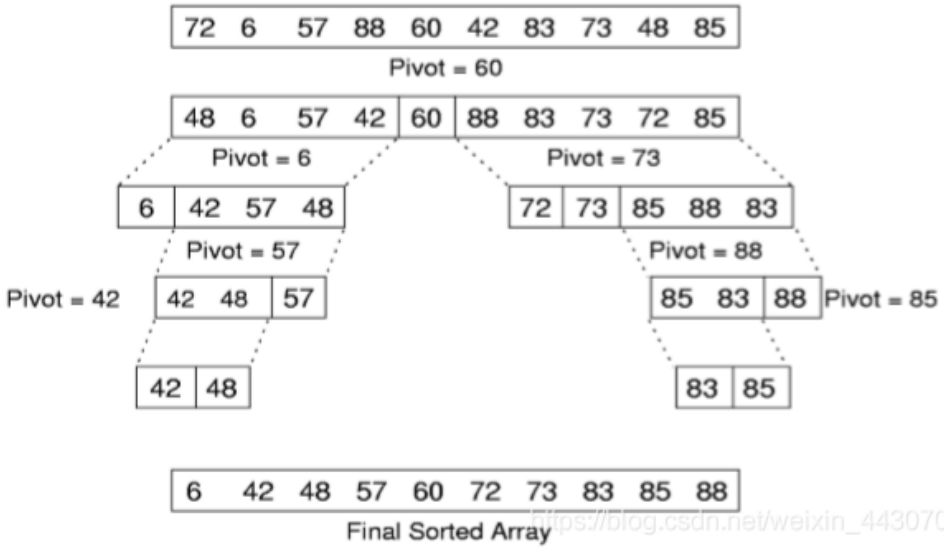
算法分析：

是不稳定算法，时间复杂度 O(n^1.5)

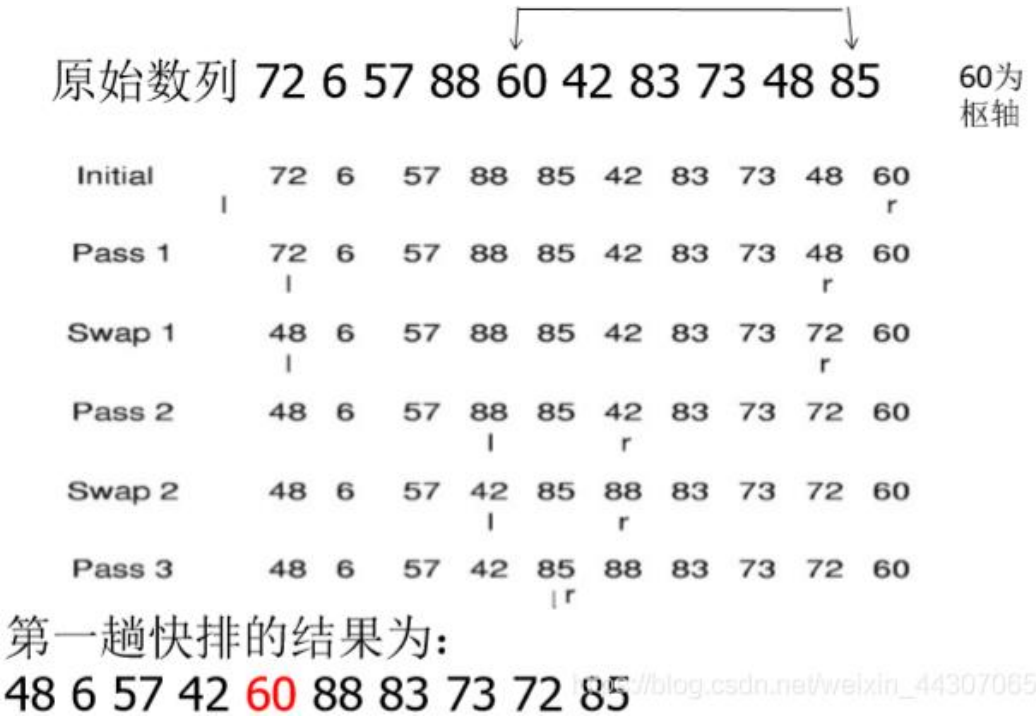
(2)快速排序

目前所有内排序算法中在平均情况下最快的一种，如：qsort 函数，体现了“分治法”的思想

目标：找一个记录，以它的关键字作为“枢轴”，凡其关键字小于枢轴的记录均移动至该记录之前，反之，凡关键字大于枢轴的记录均移动至该记录之后。致使一趟排序之后，记录的无序序列 A[s...t]将分割成两部分：A[s...i-1]和 A[i+1...t]，且 A[j1].key≤A[i].key≤A[j].key(s≤j1≤i-1)枢轴(i+1≤j≤t)。[4]



https://blog.csdn.net/weixin_44307065



算法实现：

```
template < typename E, typename Comp >
void qsort(E A[], int i, int j)
{
    if (j <= i) return; // List too small
    int pivotindex = findpivot(A, i, j);
    swap(A, pivotindex, j);    // Put pivot at end // k will be first
position on right side
    int k = partition<E,Comp>(A, i-1, j, A[j]);
    swap(A, k, j);                // Put pivot in place
    qsort<E,Comp>(A, i, k-1);
    qsort<E,Comp>(A, k+1, j);
}
template < typename E>
inline int findpivot(E A[], int i, int j)
{ return (i+j)/2; }//取中间为轴
template < typename E, typename Comp >
```

```
inline int partition(E A[], int l, int r,E& pivot)
{
    do { // Move the bounds in until they meet
    while (Comp::prior(A[++l], pivot));
    while ((l<r) && Comp::prior(pivot ,A[--r]));
    swap(A, l, r); // Swap out-of-place values
    } while (l < r); // Stop when they cross //swap(A, l, r); //
Reverse last swap  在第三版已删除
    return l;              // Return first pos on right
}
```

算法分析：

假设一次划分所得枢轴位置

i=k ，则对 n 个记录进

行快排所需时间：

其中 Tpass(n)为对

n 个记录进行一次划分所需时间：

T(n)=Tpass(n)+T(k-1)+T(n-k),其中 Tpass 是为对 n 个记录进行划分的时间

若待排序列中记录的关键字是随机分布的，则

k 取 1 至 n 中任意一值的可能性相同。由此可得快速排序所需时间的平均值为：

$$T_{avg}(n) = Cn + \frac{1}{n}\sum_{k=1}^{n}\left[T_{avg}(k-1) + T_{avg}(n-k)\right]$$

设 $T_{avg}(1) \le b$

则可得结果：

$$T_{avg}(n) < (\frac{b}{2} + 2c)(n+1)\ln(n+1)$$

结论：快速排序的时间复杂度为O(nlogn)

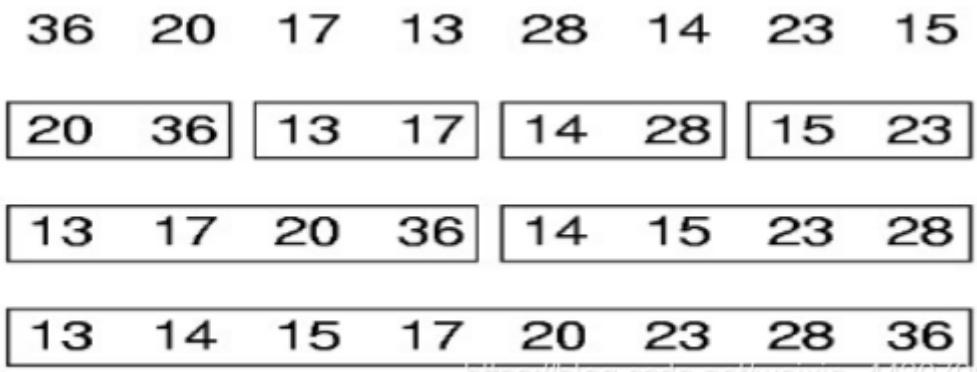
若待排记录的初始状态为按关键字有序 时，快速排序将蜕化为起泡排序，其时间复杂度为 O(n2)。

（3）归并排序

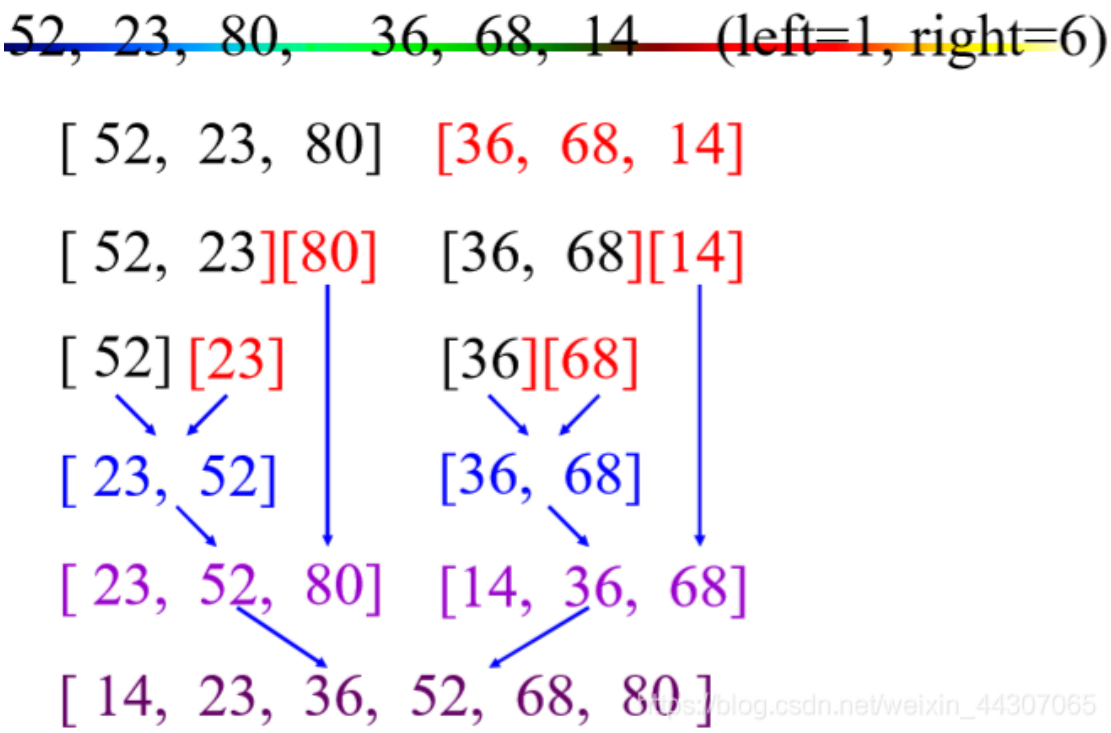
通常采用的是 2-路归并排序。基本思想也是基于分治法，即：将一个序列分成两个等长的子序列，分别对这两个子序列递归地调用归并排序算法，再将两个位置相邻的记录有序子序列。[5]

思想如下：

```
List mergesort(List inlist)
{
    if (inlist.length() <= 1) return inlist;
    List l1 = half of the items from inlist;
    List l2 = other half of items from inlist;
    return merge(mergesort(l1),mergesort(l2));
}
```



https://blog.csdn.net/weixin_44307065



https://blog.csdn.net/weixin_44307065

算法实现：

```
template < typename E, typename Comp >
void mergesort(E A[], E temp[], int left, int right)
{
    if (left == right) return;
    int mid = (left+right)/2;
    mergesort<E,Comp>(A, temp, left, mid);
    mergesort<E,Comp>(A, temp, mid+1, right);
    for (int i=left; i<=right; i++) // Copy temp[i] = A[i];
    int i1 = left,i2 = mid + 1;
```

```
    for (int curr=left; curr<=right; curr++)
    {
        if (i1 == mid+1)// Left exhausted
        A[curr] = temp[i2++];
        else if (i2 > right)    // Right exhausted
        A[curr] = temp[i1++];
        else if (Comp::prior(temp[i1], temp[i2]))
        A[curr] = temp[i1++];
        else A[curr] = temp[i2++];
    }
}
```

R.Sedgewich 提出的优化：[6]

```
template < typename E, typename Comp >
void mergesort(E A[], E temp[],int left, int right)
{
    if ((right-left) <= THRESHOLD)
    {
        inssort<E,Comp>(&A[left],right-left+1);
        return;
    }
    int i, j, k, mid = (left+right)/2;
    if (left == right) return;
    mergesort<E,Comp>(A, temp, left, mid);
    mergesort<E,Comp>(A, temp, mid+1, right);
    for (i=mid; i>=left; i--) temp[i] = A[i];//正序复制前半个表
    for (j=1; j<=right-mid; j++) temp[right-j+1] = A[j+mid];//逆
序复制后半个表
    for (i=left,j=right,k=left; k<=right; k++)//从前后两半个表相
向比较来复制较小值
    if (temp[i] < temp[j]) A[k] = temp[i++]; else A[k] = temp[j--];
```

算法分析：

设需排序元素的数目为 n，递归的深度为 logn （简单起见，设 n 是 2 的幂），第一层递归是对一 个长度为 n 的数组排序，下一层是对两个长度为 n/2 的子数组排序，…，最后一层对 n 个长度为 1 的子数组排序。

时间复杂度：T(n)=O(nlog2n)

空间复杂度：S(n)=O(n)

（4）堆排序

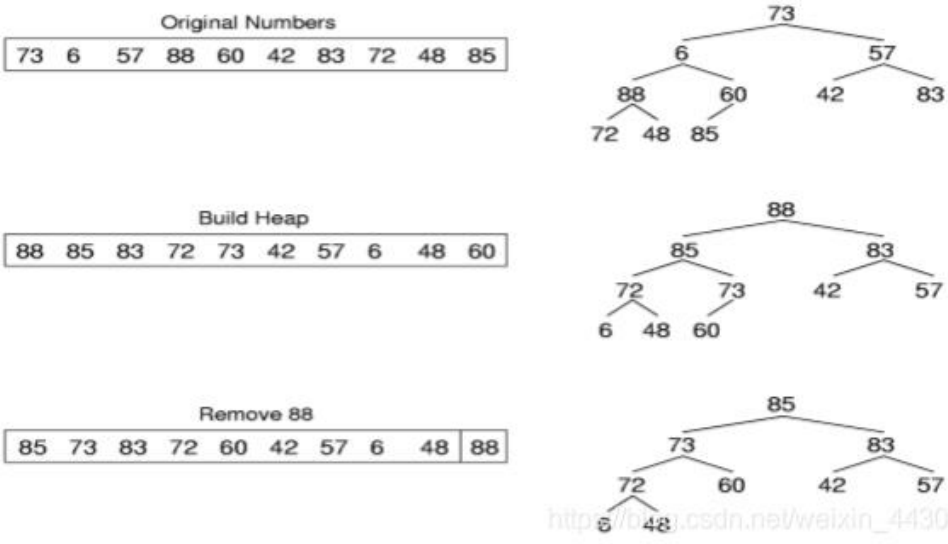
堆排序： 将无序序列建成一个堆，得到关键字最小（或最 大）的记录；输出堆顶的最小（大）值后，使剩 余的 n-1 个元素重又建成一个堆，则可得到 n 个元素的次小值；重复执行，得到一个有序序列，这 个过程叫堆排序。

堆排序需解决的两个问题：

如何由一个无序序列建成一个堆？

如何在输出堆顶元素之后，调整剩余元素，使之 成为一个新的堆？

堆在 MAXHeap 部分已经分析过，建 heap 是从 n/2-1 的位置开始往上 shiftdown（）而删除堆顶则是直接将堆顶元素与最后一个元素交换，然后 shiftdown（）一次



算法实现：

```
template < typename E, typename Comp >
void heapsort(E A[], int n)
{ // Heapsort
    E maxval;
    maxheap<E,Comp> H(A, n, n);
    for (int i=0; i<n; i++)// Now sort
    maxval=H.removefirst(); // Put max at end
}
```

算法分析：

1. 对深度为k 的堆，“筛选”所需进行的关键字 比较的次数至多为2(k-1)；
2. 对n 个关键字，建成深度为 h (=⌊ log 2 n ⌋+1)的堆， 所需进行的关键字比较的次数至多 4 n ；
3. 调整“堆顶”n-1 次，总共进行的关键字比较的次数不超过 2 (⌊log2(n-1)⌋+ [log2(n-2)]+ …+(log2)2) < 2n([og2n]) 因此，堆排序的时间复杂度为0(n log n)。

3. 特殊排序方法：

（1）分配排序

动机：按关键码分配存储位置，不进行比较；

基本思想：根据记录的关键码来确定其排序的最终位置

分配排序:

for (i=0; i<n; i++) B[A[i]] = A[i]; //根据关键码确定每条记录的位置

优点：O（n）

缺点：适用范围窄

扩展:

1.允许关键码重复，让每个盒子成为一个链表的 头结点；

2.允许关键码的范围大于 n，即比记录多，但还要使得每 条记录有个盒子。

算法实现：[7]

```
template < typename E, class getKey >
void binsort(E A[], int n)
{
```

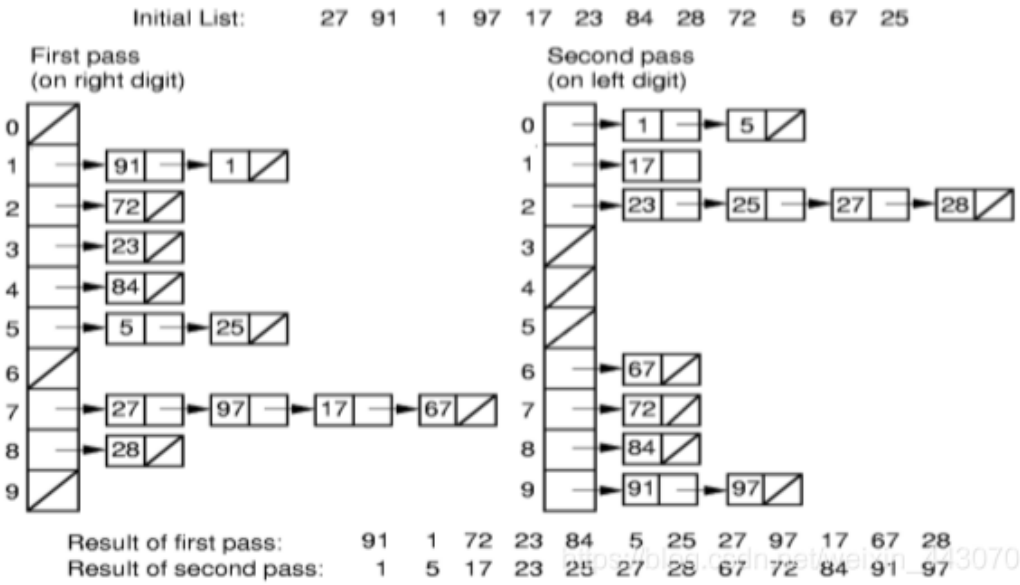
```
    List<E> B[MaxKeyValue];
    E item;
    for (i=0; i<n; i++) B[A[i]].append(getKey::key(A[i]));
    for (i=0; i<MaxKeyValue; i++)
    for (B[i].setStart(); B[i].getValue(item); B[i].next())
    output(item);
}
```

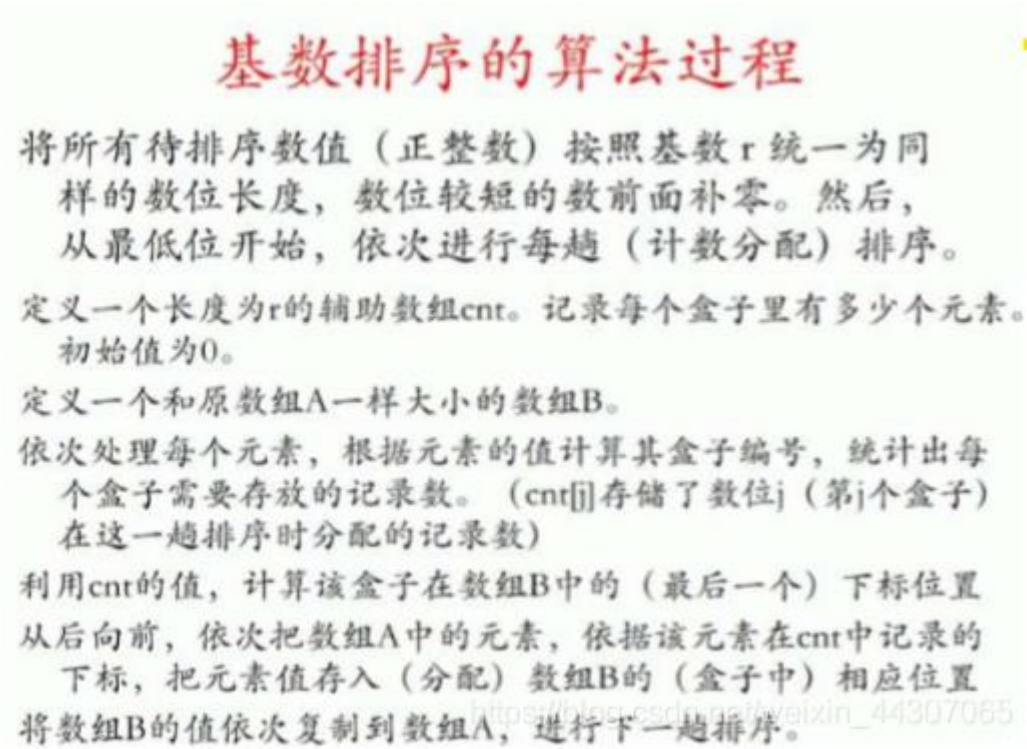
性能分析：

时间代价：O(n+MaxkeyValue),如果 MaxkeyValue 很大那么时间代价可能会达到 O(n^2)或者更差.

空间代价：O(n+MaxkeyValue)

(2)基数排序

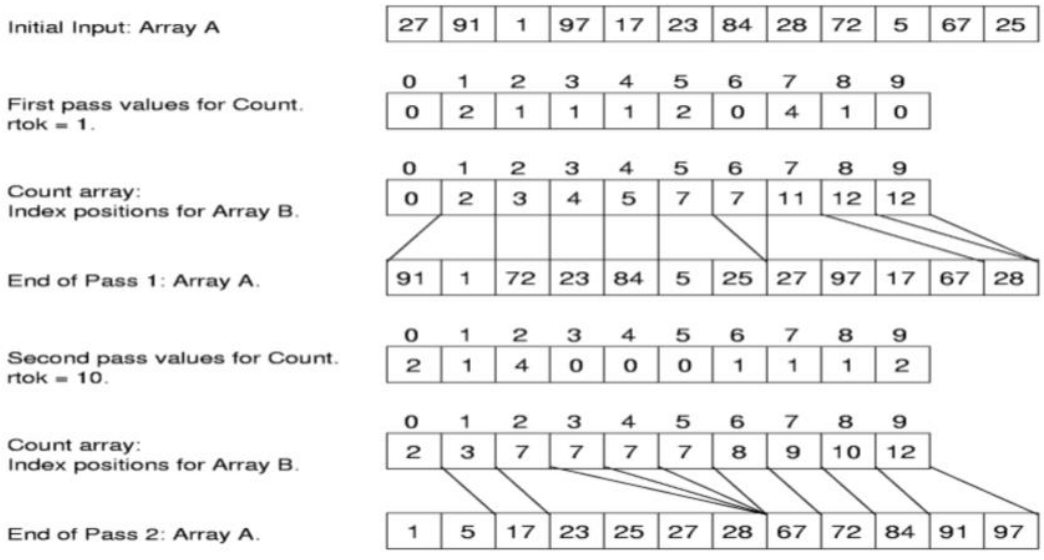





```
template < typename E, class getKey >
void radix(E A[], E B[], int n, int k, int r, int cnt[])
    { //k 是位数（关键字个数），r 是盒子数（基数） // cnt[i]
stores # of records in
    bin[i]
    int j;
```

```
    for (int i=0, rtoi=1; i<k; i++, rtoi*=r)
    {
        for (j=0; j<r; j++) cnt[j] = 0;//初始化计数值
        // Count # of records for each bin，计算每个盒子的记录数
        for(j=0; j<n; j++) cnt[(getKey ::key(A[j])/rtoi)%r]++;
        // cnt[j] will be last slot of bin j.计算每个盒子最后一个槽的位置
        for (j=1; j<r; j++) cnt[j] = cnt[j-1] + cnt[j];
        for (j=n-1; j>=0; j--)//从后向前，将每个记录放入盒子相应位置
        B[--cnt[getKey ::key(A[j])/rtoi)%r]] = A[j];
        for (j=0; j<n; j++)
        A[j] = B[j];//将数组 B 复制回去
    }
}
```

举例：



https://blog.csdn.net/weixin_44307065

基数排序算法效率

代价分析: 对于 n 个数据的序列，假设基数为 r，这个算法需要 k 趟 分配工

作。每趟分配的时间为 Θ(n+r)，因此总的时间开 销为 Θ(nk+rk)。因为 r 是基数，它一般是比较小的。可 以把它看成是一个常数。变量 k 与关键码长度有关，它是 以 r 为基数时关键码可能具有的最大位数。在一些应用中 我们可以认为 k 是有限的，因此也可以把它看成是常数。 **在这种假设下，基数排序的最佳、平均、最差时间代价 都是 Θ(n)，这使得基数排序成为我们所讨论过的具有最 好渐近复杂性的排序算法。**

各算法性能分析

	排序	平均时间性能	平均空间性能	是否稳定	备注
简单排序	插入排序	O(n^2)	O(1)	稳定	顺序有序时O（n）
	冒泡排序	O(n^2)	O(1)	稳定	顺序有序时O（n）
	选择排序	O(n^2)	O(1)	不稳定	顺序有序时退化O（n^2）
经典排序	Shell排序	O(n^1.5)	O(1)	不稳定	
	归并排序	O(nlogn)	O(n)	稳定	
	快速排序	O(nlogn)	log(n)	不稳定	
	堆排序	O(nlogn)	O(n)	不稳定	
特殊排序	分配排序	O(n+maxkeyvalue)	O(maxkeysize)	稳定	
	基数排序	Θ(n)	O(1)	稳定	

https://blog.csdn.net/weixin_44307065

基于“比较关键字”进行排序 的排序方法，可能达到的最快的时间复杂度为 O(nlogn)。

三、实验进程

1 月 1 日：

1. 结合课本及 PPT 回顾理解快速排序以及归并排序的的相关知识。
2. 通过网络学习如何使用 fstream 文件流实现文件的读写功能。

1 月 2 日：

1. 实现了快速排序以及归并排序的基本排序功能。
2. 通过网络了解 windows.h 头文件中包含的函数如何获取查找所需的时间，使用其中的QueryPerformanceCounter(&frequency)函数来获取系统频率，使用QueryPerformanceCounter(&nBeginTime)函数和QueryPerformanceCounter(&nEndTime)函数来获取开始查找时间和结束查找时间。

1 月 3 日：

1. 通过网络了解 time.h 头文件中包含的函数如何生成随机数，使用其中的 rand()函数来获取随机数。

2. 将文件的读写加入，实现了不同数据规模文件的创建、写入，以及排序时间的计算。
3. 完成 cpp 文件。

1 月 4 日：

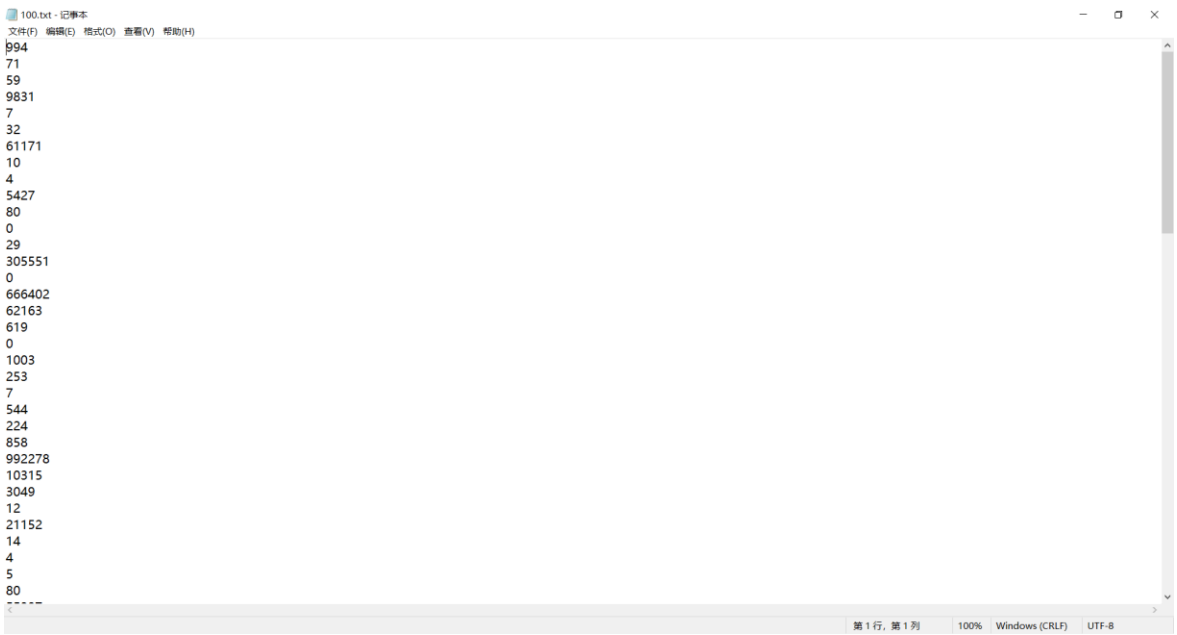
1. 通过 CSDN 进一步学习排序的相关知识。
2. 修改并完善实验日志，提交到学习通平台。

四、遇到的问题及解决方法

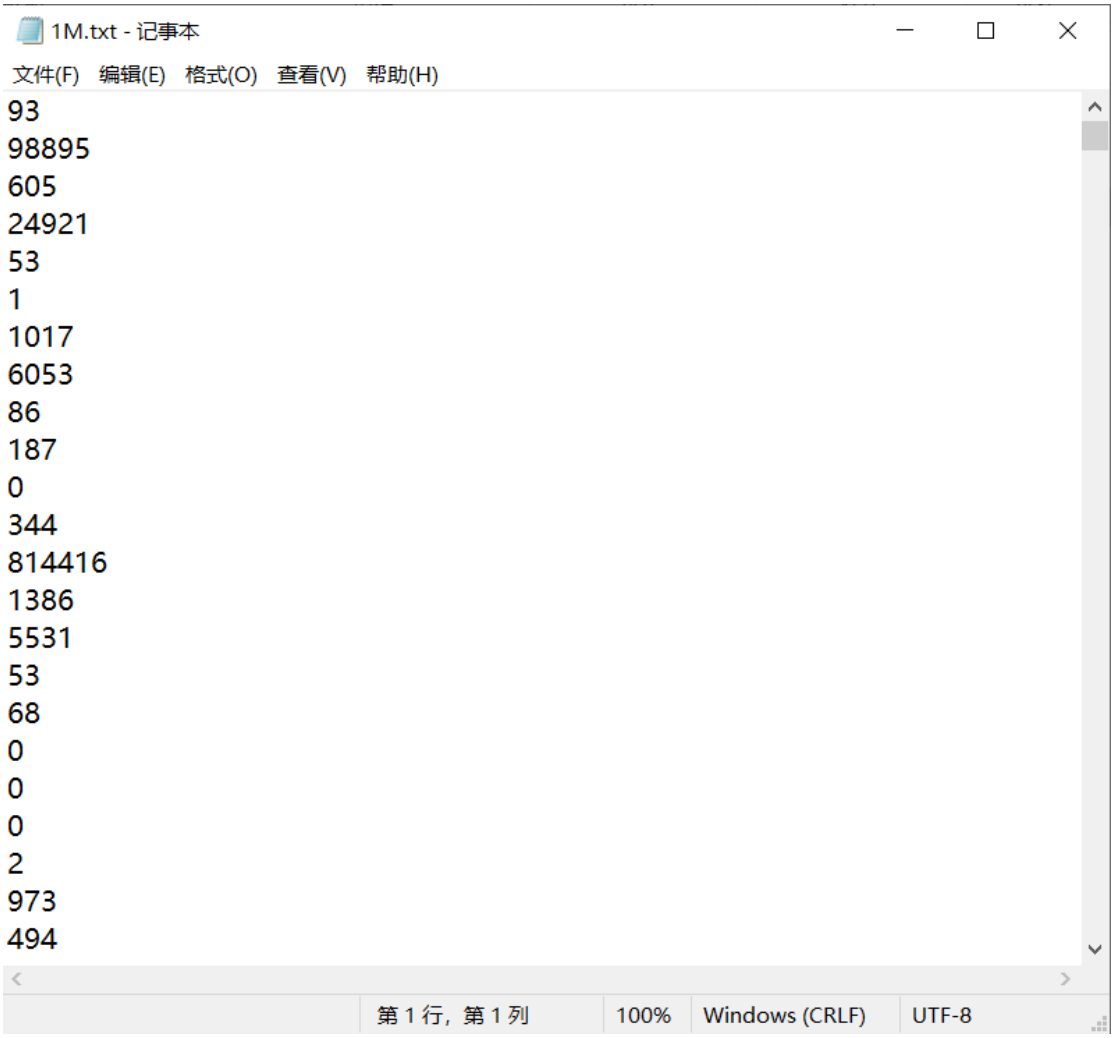
最初调用 rand()函数生成随机数时，并没有指定随机数的种子，导致每次生成的随机数都是相同的，后来调用 srand((int)time(NULL))函数，将随机数种子设置为系统时间，解决了这个问题。

clock()函数、GetTickCount()函数、timeGetTime()函数、Boost 库中的 timer、高精度时控函数 QueryPerformanceFrequency()和 QueryPerformanceCounter()，以上函数都可以记录程序运行时间，但由于数据量较小时，程序运行时间很短，前三个函数的精度不足以计时，这时应该选用高精度时控函数 QueryPerformanceFrequency()和 QueryPerformanceCounter()。

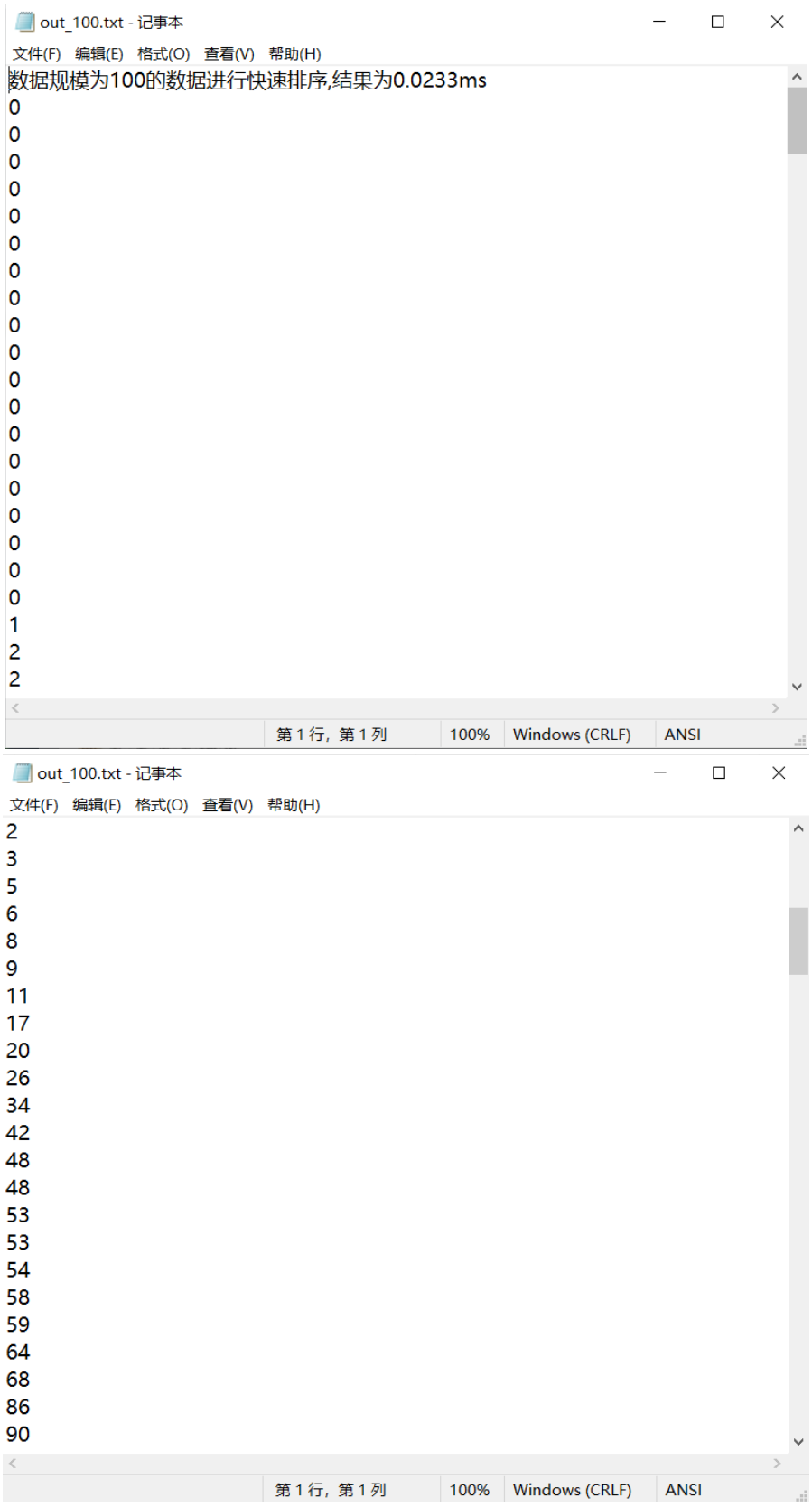
五、测试用例



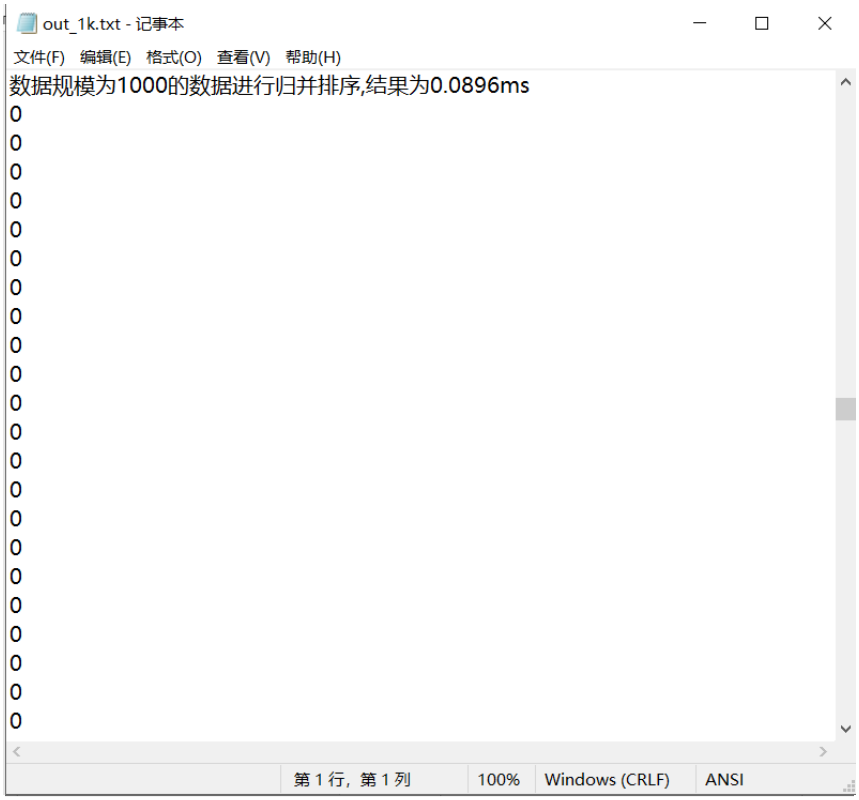
数据规模为100

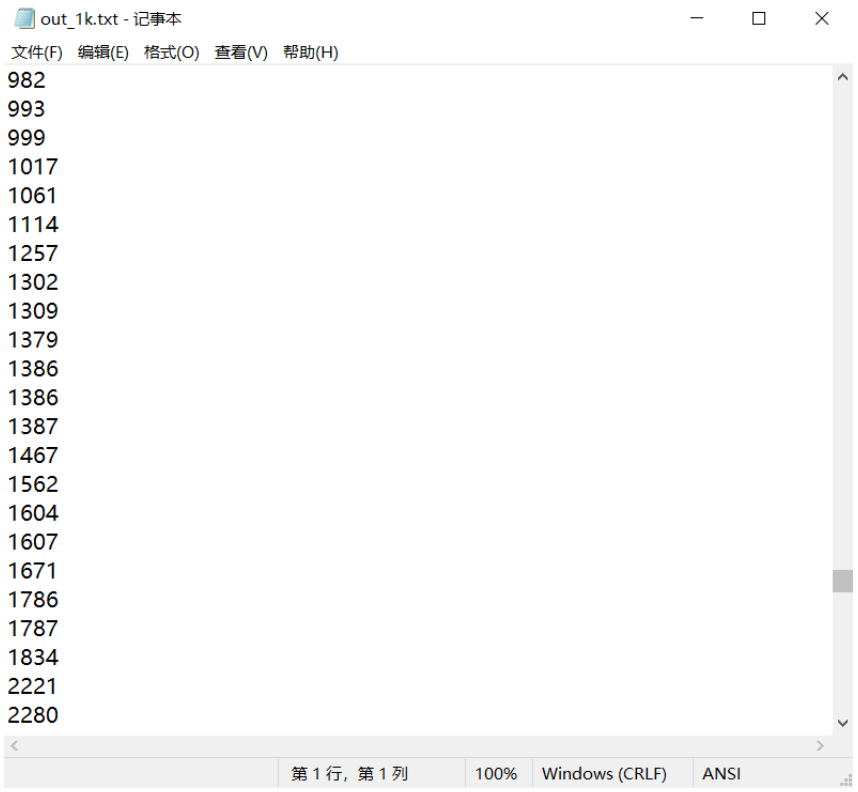


数据规模为1M的测试数据

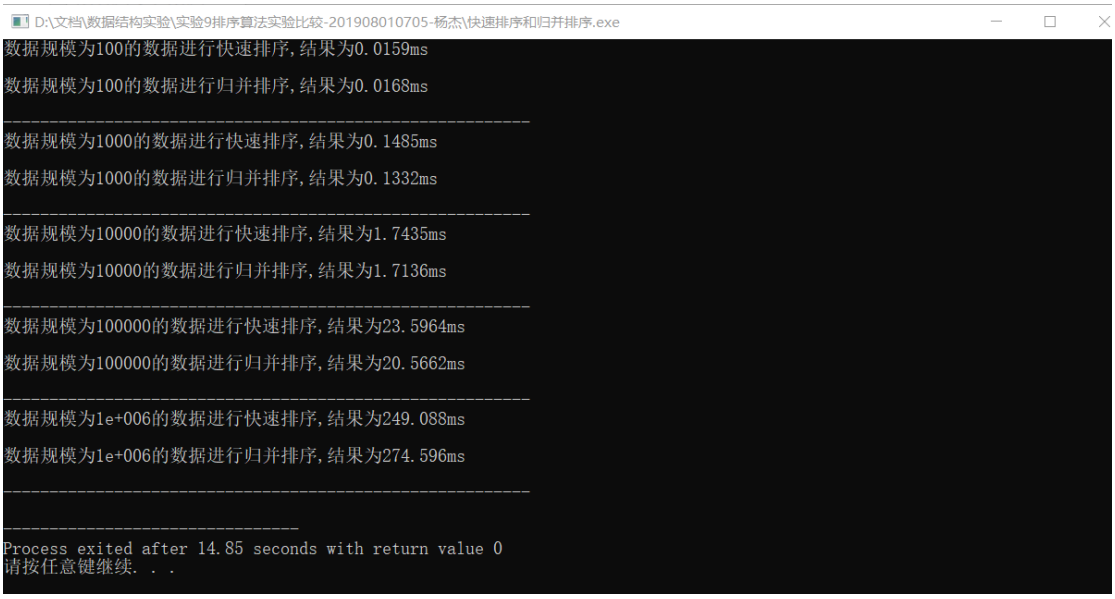


数据规模为100的数据快速排序结果





数据规模为1k的数据归并排序结果



各数据规模的数据快速排序与归并排序结果

六、实验结果

排序算法	数据规模	100	1K	10K	100K	1M
快速排序	排序时间	0.0156	0.1457	1.7441	20.6637	242.645
归并排序		0.0185	0.1294	2.7951	24.0444	285.919

（时间单位：ms）

七、实验分析

从上述实验数据中可以看出，当数据量较小时，快速排序和归并排序的时间消耗相当；但是随着实验数据的增多，不难发现快速排序所用的时间较少。

从实验结果可以看出，实验存在误差，经过分析，发现原因：一是时间函数的精确度不高，导致小数据的快速排序和归并排序所消耗的时间判定几乎为0。二是求取随机数的函数，当数据量较大时，最终结果偏小。

总体来讲，快速排序所耗时间要比归并排序少。

八、总结与心得

经过这次实验，我对于快速排序和归并排序的相关代码已基本熟悉，算法知识得到了复习与巩固。在写代码与调试的过程中，在解决问题过程中，丰富了个人编程的经历和经验，提高了个人解决问题的能力。

通过本次报告实验，我对排序的概念有了一个新的认识，由于这段时间临近期末事情比较多，故在实验上未花很多的精力，用了三天抽空把排序的两种算法做了，有些不尽人意的地方也没有加以修正，但是实验的过程之中我又把排序的知识重新温习了一遍，获益匪浅。这是数据结构课堂的最后一个实验，但学习的脚步永远不会停止，我会多写代码多实践，这样我才能发现自己在实际操作中的不足并加以改正。

九、实验说明

开发语言：C++

开发平台：Dev-C++

参考资料

1. 许卓群.数据结构与算法[M].高等教育出版社,2004.
2. 彭军、向毅.数据结构与算法[M].人民邮电出版社,2013.
3. Budd T.经典数据结构（Java 语言版）[M].清华大学出版社,2005.
4. 王晓东.算法设计与分析[M].清华大学出版社,2003.5.
5. 严蔚敏.数据结构 C 语言版[M].清华大学出版社,2007.
6. 王广芳.数据结构算法与应用-C++语言描述[M].机械工业出版社,2006.
7. 严蔚敏等.数据结构题集(C 语言版)[M].清华大学出版社,1999.2.