

《数据结构》上机报告

2018 年 1 月 6 日

姓名： 赵得泽 学号： 1753642 班级： 电子二班 得分： _____

实验题目	排序算法汇总	
问题描述	排序算法分为简单排序（时间复杂度为 $O(n^2)$ ）和高效排序（时间复杂度为 $O(n \log n)$ ）。	
基本要求	本题给定 N 个整数，要求输出从小到大排序后的结果。 请用不同的排序算法（直接插入排序，折半插入排序，希尔排序，冒泡排序，快速排序，选择排序，归并排序，堆排序）分别进行测试，查看每个算法的运行时间和通过组数，进行对比分析，总结，写入报告中。	
	已完成基本内容（序号）：	1
选做要求		
	已完成选做内容（序号）	
数据结构设计	<pre>typedef int KeyType; typedef struct { KeyType key; } RedType; typedef struct { RedType r[MAX_SIZE + 1]; int length; } SqList;</pre> 数据结构主要用到的就是线性表中的顺序表；	
归并排序		

算法源码

```

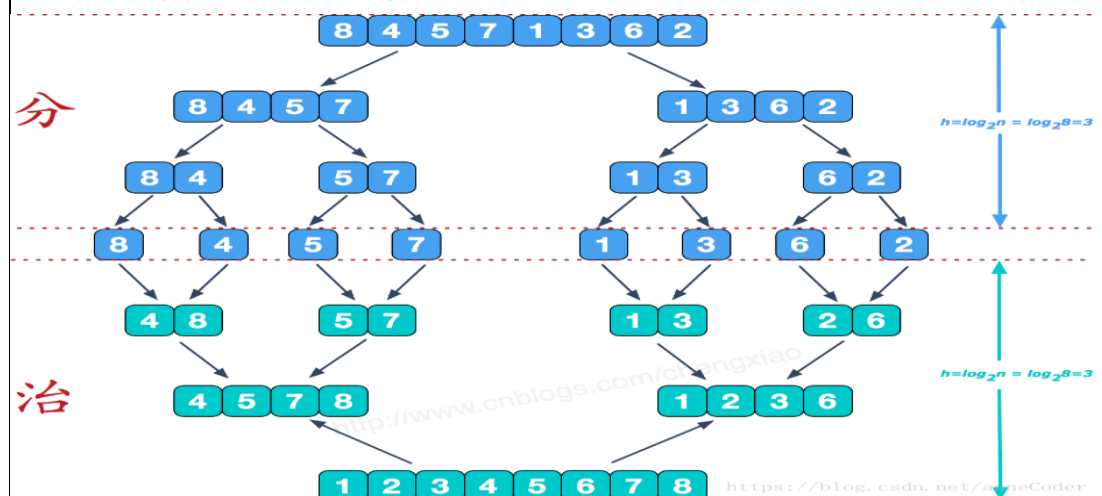
RedType L[MAX_SIZE / 2 + 2], R[MAX_SIZE / 2 + 2];
void Merge(Sqlist&S, int left, int mid, int right)
{
    int n1 = mid - left, n2 = right - mid;
    for (int i = 0; i < n1; i++)
        L[i] = S.r[left + i];
    for (int i = 0; i < n2; i++)
        R[i] = S.r[mid + i];
    int i = 0, j = 0;
    R[n2].key = L[n1].key = INF;
    for (int k = left; k < right; k++)
    {
        if (L[i].key <= R[j].key)
            S.r[k] = L[i++];
        else
            S.r[k] = R[j++];
    }
}

void MergeSort(Sqlist&S, int left, int right)
{
    if (left + 1 < right)
    {
        int mid = (left + right) / 2;
        MergeSort(S, left, mid);
        MergeSort(S, mid, right);
        Merge(S, left, mid, right);
    }
}

```

算法说明 及图示

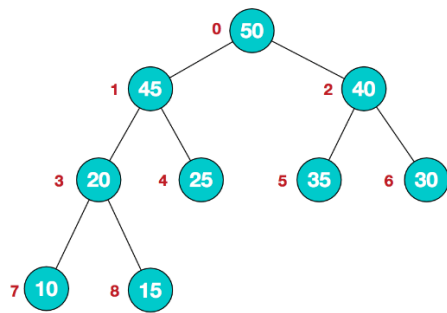
这种方法也就是分治的策略，将有序的子序列合并，得到完全有序的序列。若将两个有序表合并成一个有序表，称为二路归并。



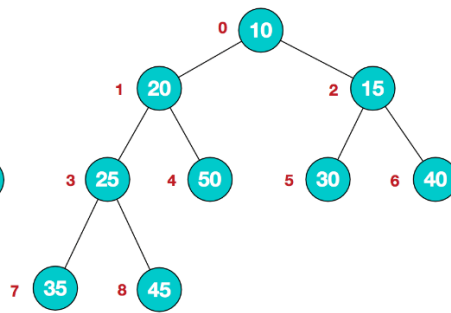
堆排序

算法源码	<pre> RedType temp; typedef SqList HeapType; int pivotkey; void HeapAdjust(HeapType &H, int s, int m) { int j; RedType rc = H.r[s]; for (j = 2 * s; j <= m; j *= 2) { if (j < m && H.r[j].key < H.r[j + 1].key) j++; if (!(rc.key < H.r[j].key)) break; H.r[s] = H.r[j]; s = j; } H.r[s] = rc; } void HeapSort(HeapType&H) { RedType temp; for (int i = H.length / 2; i > 0; i--) HeapAdjust(H, i, H.length); //建立初始大顶堆; for (int i = H.length; i > 1; i--) { temp = H.r[1]; H.r[1] = H.r[i]; H.r[i] = temp; HeapAdjust(H, 1, i - 1); //将1~i-1重新调整为大顶堆; } } </pre>
算法说明及图示	<p>堆的说明：</p> <ul style="list-style-type: none"> (1) 性质：堆是一棵完全二叉树，完全二叉树不一定是堆； (2) 分类：大顶堆：父节点不小于子节点键值，小顶堆：父节点不大于子节点键值；上述算法中我们采用的是大顶堆； (3) 左右孩子：没有大小的顺序。 (4) 堆的存储：一般都用数组来存储堆，i结点的父结点下标就为$(i-1)/2$。它的左右子结点下标分别为$2*i+1$和$2*i+2$。如第0个结点左右子结点下标分别为1和2。 <p>算法描述：</p> <p>将初始的待排序关键字序列建立成大顶堆，次堆为初始的无序区；再将堆顶元素$r[1]$与最后一个元素$r[n]$交换，此事得到新的无序区$r[1, 2\cdots, n-1]$，且满足$r[1, 2\cdots, n-1] \leq r[n]$；由于交换后的新堆顶可能违反堆的性质，所以对当前$r[1, 2\cdots, n-1]$继续进行调整为大顶堆，然后再将$r[1]$与无序区的最后一个元素交换，以此类推，不断重复直到有序取得元素个数为$n-1$，则整个排序过程完成。</p> <p>算法图示：</p>

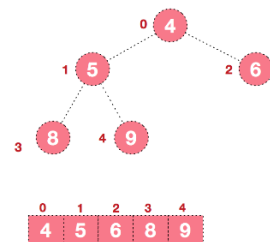
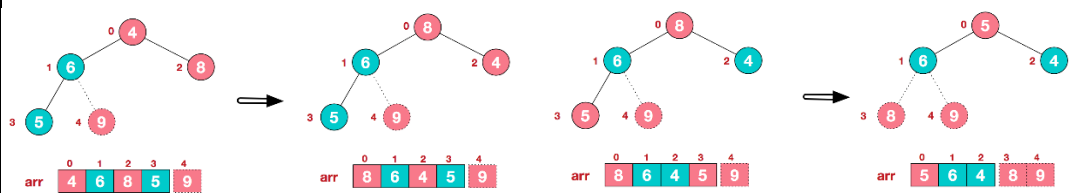
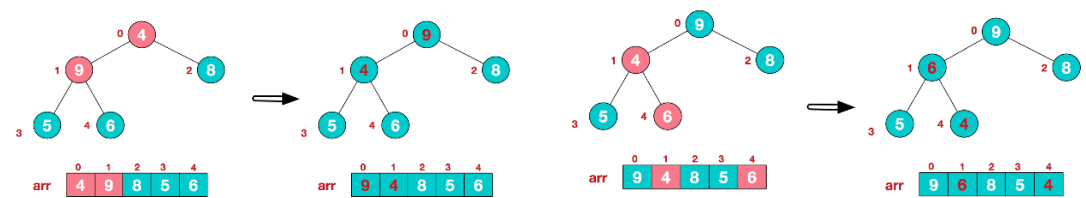
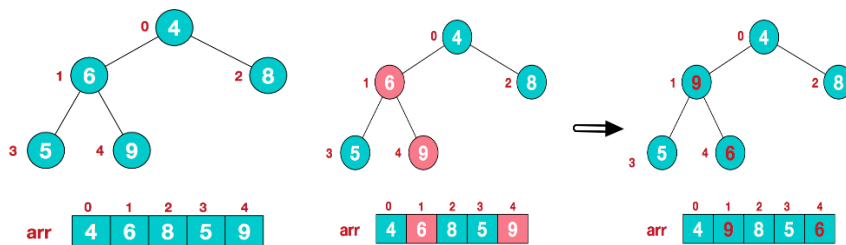
大顶堆



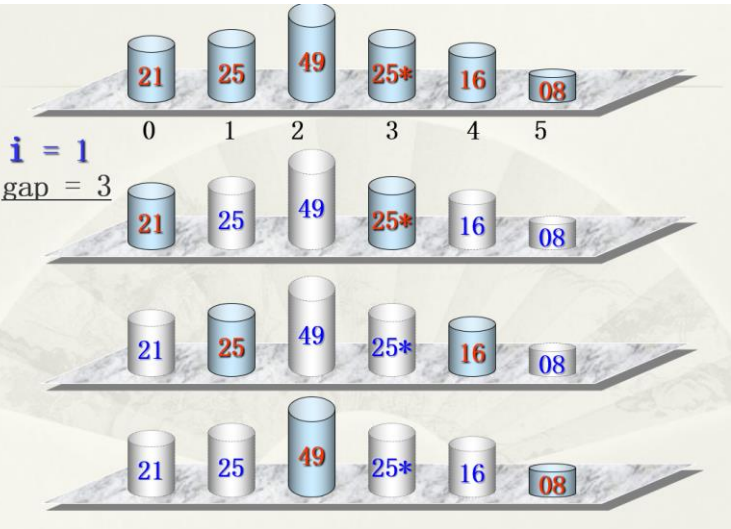
小顶堆

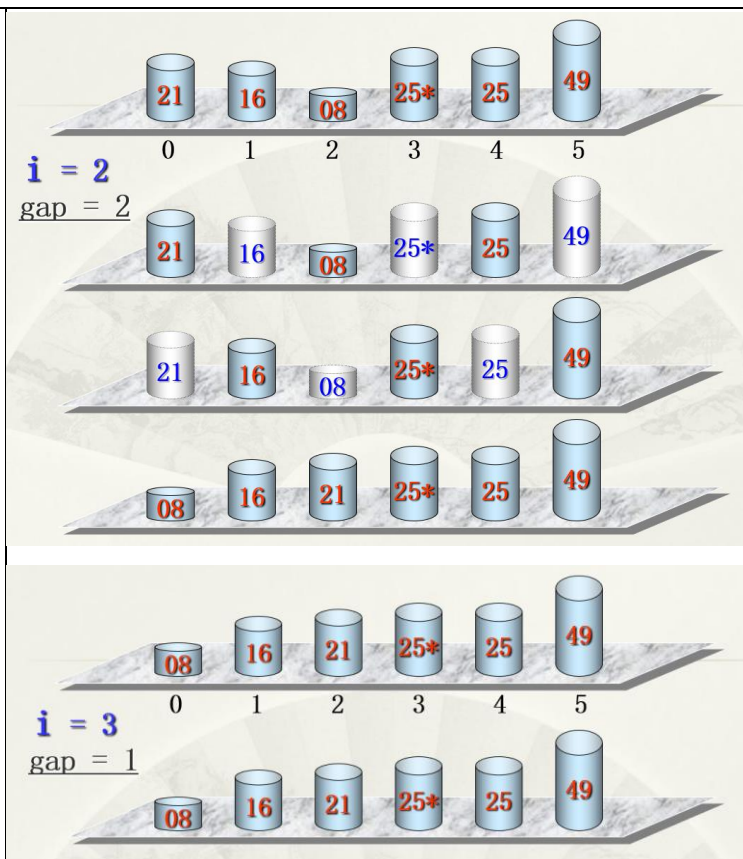


	0	1	2	3	4	5	6	7	8
arr	50	45	40	20	25	35	30	10	15



希尔排序

算法源码	<pre>void ShellInsert(SqList &L, int gap) { int j; for (int i = gap + 1; i <= L.length; i++) { if (L.r[i].key < L.r[i - gap].key) { L.r[0] = L.r[i]; for (j = i - gap; j > 0 && L.r[0].key < L.r[j].key; j = j - gap) L.r[j + gap] = L.r[j]; L.r[j + gap] = L.r[0]; } } } void ShellSort(SqList&L, int Delta[], int t) { for (int k = 0; k < t; k++) ShellInsert(L, Delta[k]); }</pre>
算法说明及图示	<p>希尔排序方法又称为“缩小增量”排序。</p> <p>算法描述：</p> <p>先将整个待排对象序列按照一定间隔分割成为若干子序列，分别进行直接插入排序 然后缩小间隔，对整个对象序列重复以上的划分子序列和分别排序工作，直到最后间隔为1</p> <p>此时整个对象序列已“基本有序”，最后，进行一次直接插入排序。</p> <p>算法图示：</p>  <p>The diagram shows four horizontal rows representing the array [21, 25, 49, 25*, 16, 08] at different stages of the Shell Sort process. The indices 0 through 5 are labeled below each row. The first row is labeled with $i = 1$ and $gap = 3$. The second row shows the array after the first pass with $gap = 3$. The third row shows the array after the second pass with $gap = 2$. The fourth row shows the array after the third pass with $gap = 1$.</p>



快速排序

算法源码

第一种:

```
RedType temp;
int pivotloc;
int pivotkey;
int ppp(SqList&L, int high, int low)
{
    pivotkey = L.r[low].key;//枢轴记录关键字
    while (low < high)
    {
        while (low < high&&L.r[high].key >= pivotkey)
            high--;
        temp = L.r[low];
        L.r[low] = L.r[high];
        L.r[high] = temp;
        while (low < high&&L.r[low].key <= pivotkey)
            low++;
        temp = L.r[low];
        L.r[low] = L.r[high];
        L.r[high] = temp;
    }
}
```

```

        return low;
    }
    int QuickSort(SqList &L, int low, int high)
    {
        if (low < high)
        {
            pivotloc = ppp(L, high, low);
            QuickSort(L, low, pivotloc - 1); //低子表进行递归
            QuickSort(L, pivotloc + 1, high); //高子表进行递归
        }
    }
}

```

第二种:

```

RedType temp;
int pivotkey;
void QuickSort(SqList&L, int low, int high)
{
    if (low < high)
    {
        int pl = low, ph = high;
        pivotkey = L.r[low].key; //枢轴记录关键字
        RedType p = L.r[low];
        while (low < high)
        {
            while (low < high && L.r[high].key >= pivotkey)
                high--;
            if (low < high)
                L.r[low++] = L.r[high];
            while (low < high && L.r[low].key <= pivotkey)
                low++;
            if (low < high)
                L.r[high--] = L.r[low];
        }
        L.r[low] = p;
        QuickSort(L, pl, low - 1); //低子表进行递归
        QuickSort(L, low + 1, ph); //高子表进行递归
    }
}

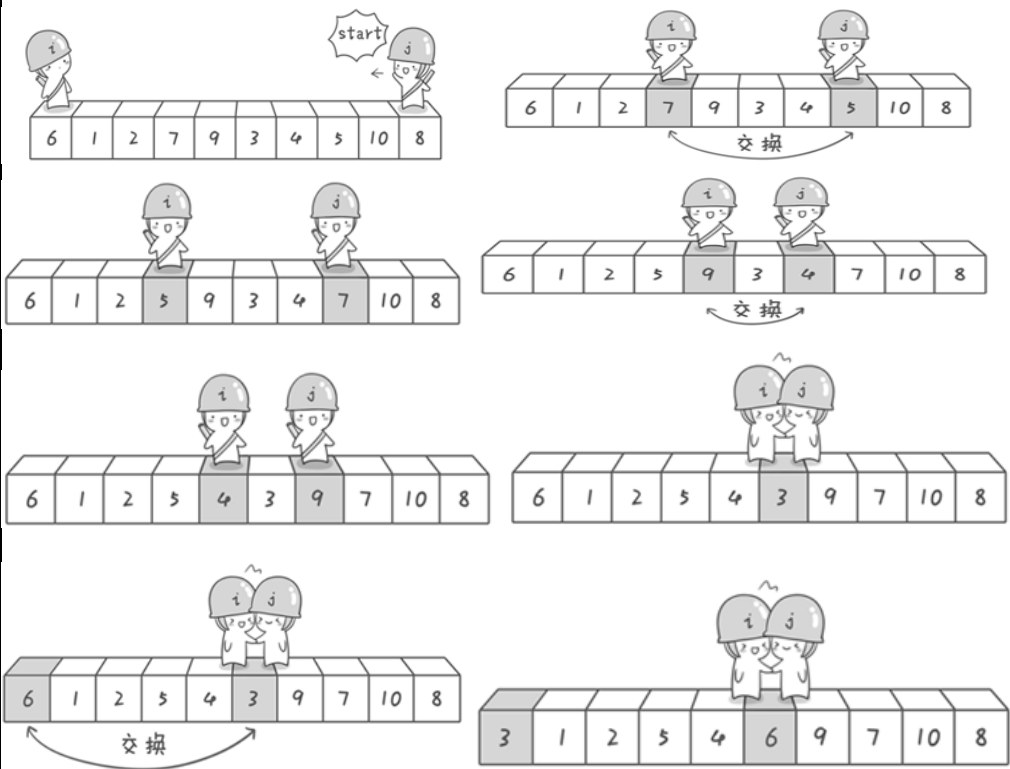
```

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

算法图示：

以 3 1 2 5 4 6 9 7 10 8 为例，取中间数6为参照数

算法说明及
图示



选择排序

算法源码

```
void SelectSort(Sqlist &L)
{
    int k;
    for (int i = 1; i < L.length; i++)
    {
        k = i;
        for (int j = i + 1; j <= L.length; j++)//选出剩余元素中最小值
            if (L.r[j].key < L.r[k].key)
                k = j;
        if (k != i)//将第i个与剩余元素中的最小值交换
        {
            RedType temp = L.r[k];
            L.r[k] = L.r[i];
            L.r[i] = temp;
        }
    }
}
```


<div>算法说明及 图示</div>	<div><p>算法描述： 每次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，知道全部待排序的数据元素排完。</p><p>算法图示：</p><div><div></div><div>排序前： 9 1 2 5 7 4 8 6 3 5</div><div>第 1 趟： 1 9 2 5 7 4 8 6 3 5</div><div>第 2 趟： 1 2 9 5 7 4 8 6 3 5</div><div>第 3 趟： 1 2 3 5 7 4 8 6 9 5</div><div>第 4 趟： 1 2 3 4 7 5 8 6 9 5</div><div>第 5 趟： 1 2 3 4 5 7 8 6 9 5</div><div>第 6 趟： 1 2 3 4 5 5 8 6 9 7</div><div>第 7 趟： 1 2 3 4 5 5 6 8 9 7</div><div>第 8 趟： 1 2 3 4 5 5 6 7 9 8</div><div>第 9 趟： 1 2 3 4 5 5 6 7 8 9</div><div>排序后： 1 2 3 4 5 5 6 7 8 9</div></div><div>红色粗体表示位置发生变化的元素</div></div>
-------------------------	---

插入排序	
算法源码	<pre>void InsertSort(Sqlist &L) { int j; for (int i = 2; i <= L.length; i++) { if (L.r[i].key < L.r[i - 1].key) { L.r[0] = L.r[i]; L.r[i] = L.r[i - 1]; for (j = i - 2; L.r[0].key < L.r[j].key; j--) L.r[j + 1] = L.r[j]; L.r[j + 1] = L.r[0]; } } }</pre>
算法说明及图示	<p>算法描述： 工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用 in-place 排序（即只需用到 $O(1)$ 的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。</p> <p>算法图示：</p> <p>The diagram illustrates the Insertion Sort algorithm through a series of steps, divided into two columns: '有序' (Sorted) and '待排序' (Unsorted).</p> <ul style="list-style-type: none">Initial State: The '有序' column contains [3, 9] and the '待排序' column contains [3, 1, 4, 2, 7, 8, 6, 5].Step 1: The element 1 is moved from '待排序' to '有序', resulting in [1, 3, 9] and [4, 2, 7, 8, 6, 5].Step 2: The element 4 is moved from '待排序' to '有序', resulting in [1, 3, 4, 9] and [2, 7, 8, 6, 5].Text: 对待排序的序列逐个插入，直到插完为止 (Insert elements from the unsorted sequence one by one until all are inserted).Step 3: The element 5 is moved from '待排序' to '有序', resulting in [1, 2, 3, 4, 5, 6, 7, 8, 9] and an empty '待排序' column.Text: 所有元素全部插入，排序完成 (All elements are inserted, sorting is complete).Final State: The '有序' column contains the fully sorted sequence [1, 2, 3, 4, 5, 6, 7, 8, 9].

折半插入排序	
算法源码	<pre>void BInsertSort(SqList &L) { int j; for (int i = 2; i <= L.length; i++) { L.r[0] = L.r[i]; int low = 1; int high = i - 1; while (low <= high) { int m = (low + high) / 2; if (L.r[0].key < L.r[m].key) high = m - 1; else low = m + 1; } for (j = i - 1; j >= high + 1; j--) L.r[j + 1] = L.r[j]; L.r[high + 1] = L.r[0]; } }</pre>
算法说明及图示	<p>算法描述： 设在顺序表中有一个对象序列$V[0], V[1], \dots, v[n-1]$。其中，$v[0], V[1], \dots, v[i-1]$是已经排好序的对象。在插入 $v[i]$ 时，利用折半搜索法寻找 $v[i]$ 的插入位置。</p> <p>算法图示：</p> <p>初始关键字: 32 46 21 6 8</p> <p>i=1: (32) 46 21 6 8</p> <p>i=2: (32 46) 21 6 8</p> <p>i=3: (21 32 46) 6 8</p> <p>i=4: (6 21 32 46) 8</p> <p>i=5: (6 8 21 32 46)</p>

冒泡排序	
算法源码	<pre>void BubbleSort(SqList &L) { for (int i = 1; i < L.length; i++) for (int j = 1; j <= L.length - i; j++) { if (L.r[j + 1].key < L.r[j].key) { RedType temp = L.r[j + 1]; L.r[j + 1] = L.r[j]; L.r[j] = temp; } } }</pre>
算法说明及图示	<p>算法描述： 设待排序对象序列中的对象个数为 n。最多作 $n-1$ 趟排序。在第 i 趟中顺次两两比较 $r[j-1].Key$和$r[j].Key$, $j = i, i+1, \dots, n-i-1$。如果发生逆序, 则交换$r[j-1]$和$r[j]$。</p> <p>算法图示：</p> <p>相邻元素两两比较，反序则交换</p> <p>第一轮完毕，将最大元素9浮到数组顶端</p> <p>同理,第二轮将第二大元素8浮到数组顶端</p> <p>排序完成</p>

算法结果分析	属性 算法		平均时间复杂度	最坏时间复杂度	本实验运行时间	通过数据组数	
	InsertSort		$O(n^2)$	$O(n^2)$	3.182 seconds	10	
	BInsertSort		$O(n^2)$	$O(n^2)$	3.515 seconds	10	
	ShellSort		$O(n\log n)$	$O(n^{1.3})$	5.678 seconds	15	
	BubbleSort		$O(n^2)$	$O(n^2)$	3.157 seconds	8	
	QuickSort		$O(n\log n)$	$O(n^2)$	6.736 seconds	12	
	HeapSort		$O(n\log n)$	$O(n\log n)$	4.05 seconds	15	
	MergeSort		$O(n\log n)$	$O(n\log n)$	1.802 seconds	15	
	SelectSort		$O(n^2)$	$O(n^2)$	4.146 seconds	8	
各种常用排序算法							
	类别	排序方法	时间复杂度			空间复杂度	稳定性
			平均情况	最好情况	最坏情况	辅助存储	
	插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
		shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
	选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
		堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
	交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
		快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
	归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	稳定
	调试分析	排序结果如图：					
<div>13</div> <div>81 68 33 40 89 72 75 21 75 3 63 15 54</div> <div>3 15 21 33 40 54 63 68 72 75 75 81 89</div>							
各算法的分析总结	QuickSort	优点：极快数据移动少； 缺点：不稳定； 此排序算法的效率在序列越乱的时候，效率越高。在数据有序时，会退化成冒泡排序；					
	BubbleSort	优点：稳定 缺点：慢，每次只能移动两个相邻的数据；					
	SelectSort	交换次数比冒泡排序少多了，由于交换所需 CPU 时间比较所需的 CPU 时间多，n 值较小时，选择排序比冒泡排序快。					
	ShellSort	比较在希尔排序中是最主要的操作，而不是交换。用已知最好的步长序列的希尔排序比直接插入排序要快，甚至在小数组中比快速排序和堆排序还快，但在涉及大量数据时希尔排序还是不如快排；					
	MergeSort	若 n 较大，并且要求排序稳定，则可以选择归并排序；					

	InsertSort	<p>优点:稳定, 快</p> <p>缺点: 比较次数不一定, 比较次数越少, 插入点后的数据移动越多, 特别是当数据总量庞大的时候</p> <p>最好情况下, 排序前对象已经按关键字大小从小到大有序, 每趟只需与前面的有序对象序列的最后一个对象的关键字比较 1 次, 总的关键字比较次数为 $n-1$, 不需要移动元素。</p>
	BInsertSort	<p>折半查找比顺序查找快, 所以折半插入排序就平均性能来说比直接插入排序要快。</p> <p>它所需要的关键字比较次数与待排序对象序列的初始排列无关, 仅依赖于对象个数。</p>
	HeapSort	<p>由于它在直接选择排序的基础上利用了比较结果形成。效率提高很大。它完成排序的总比较次数为 $O(n\log_2 n)$。它是对数据的有序性不敏感的一种算法。但堆排序将需要做两个步骤: 一是建堆, 二是排序 (调整堆)。所以一般在小规模序列中不合适, 但对于较大的序列, 将表现出优越的性能。</p>