一、基础介绍

排序有内部排序和外部排序，内部排序是数据记录在内存中进行排序，而外部排序是因排序的数据很大，一次不能容纳全部的排序记录，在排序过程中需要访问外存。我们这里说的排序就是内部排序。

> 当n较大，则应采用时间复杂度为O(nlog2n)的排序方法：快速排序、堆排序或归并排序序。
> 快速排序：是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；

1.1 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'。

1.2 按需要的额外内存分类

- 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。

1.3 按排序是否稳定分类

排序算法的稳定性:若待排序的序列中，存在多个具有相同关键字的记录，经过排序，这些记录的相对次序保持不变，则称该算法是稳定的；若经排序后，记录的相对次序发生了改变，则称该算法是不稳定的。

- **稳定性的好处：**
  排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。基数排序就是这样，先按低位排序，逐次按高位排序，低位相同的元素其顺序再高位也相同时是不会改变的。另外，如果排序算法稳定，可以避免多余的比较；
- **stable sort：**
  > 插入排序、冒泡排序、归并排序、基数排序、桶排序、计数排序。
- **unstable sort：**
  > 希尔排序、选择排序、快速排序、堆排序。

1.4 时间复杂度

1. **平方阶(O(n2))排序**
   各类简单排序:直接插入、直接选择和冒泡排序；
2. **线性对数阶(O(nlog2n))排序**
   快速排序、堆排序和归并排序；
3. **O(n1+§))排序,§是介于0和1之间的常数。**

希尔排序

4. **线性阶(O(n))排序**

   基数排序，此外还有桶、箱排序。

说明：

> 当原表有序或基本有序时，直接插入排序和冒泡排序将大大减少比较次数和移动记录的次数，时间复杂度可降至O（n）；
> 而快速排序则相反，当原表基本有序时，将蜕化为冒泡排序，时间复杂度提高为O（n2）；
> 原表是否有序，对简单选择排序、堆排序、归并排序和基数排序的时间复杂度影响不大。

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均复杂度	最好情况	最坏情况	辅助存储	
插入排序	直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	希尔排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(n)$或$O(log_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
基数排序		O(nd))	O(nd))	O(nd))	O(r+n)	稳定
n为数据规模，基数排序中r为基数，表示r进制，d为位数						

1.5 选择排序算法准则

每种排序算法都各有优缺点。因此，在实用时需根据不同情况适当选用，甚至可以将多种方法结合起来使用。
影响排序的因素有很多，平均时间复杂度低的算法并不一定就是最优的。相反，有时平均时间复杂度高的算法可能更适合某些特殊情况。同时，选择算法时还得考虑它的可读性，以利于软件的维护。一般而言，**需要考虑的因素有以下四点：**

1. 待排序的记录数目n的大小；
2. 记录本身数据量的大小，也就是记录中除关键字外的其他信息量的大小；
3. 关键字的结构及其分布情况；

4. 对排序稳定性的要求。

- **设待排序元素的个数为n.**

    a. 当n较大，则应采用时间复杂度为O(nlog2n)的排序方法：快速排序、堆排序或归并排序序。
    快速排序：是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；
    堆排序 ： 如果内存空间允许且要求稳定性的，
    归并排序：它有一定数量的数据移动，所以我们可能过与插入排序组合，先获得一定长度的序列，然后再合并，在效率上将有所提高。

    b. 当n较大，内存空间允许，且要求稳定性 =》归并排序

    c. 当n较小，可采用直接插入或直接选择排序。
    直接插入排序：当元素分布有序，直接插入排序将大大减少比较次数和移动记录的次数。
    直接选择排序 ：元素分布有序，如果不要求稳定性，选择直接选择排序

    d. 一般不使用或不直接使用传统的冒泡排序。

    e. 基数排序
    它是一种稳定的排序算法，但有一定的局限性：
    1、关键字可分解。
    2、记录的关键字位数较少，如果密集更好
    3、如果是数字时，最好是无符号的，否则将增加相应的映射复杂度，可先将其正负分开排序。

二、排序算法介绍

2.1 插入排序

2.1.1 直接插入排序

- **基本思想**
  将一个记录插入到已排序好的有序表中，从而得到一个新，记录数增1的有序表。即：先将序列的第1个记录看成是一个有序的子序列，然后从第2个记录逐个进行插入，直至整个序列有序为止。**稳定的**排序算法。

  > 要点：设立哨兵，作为临时存储和判断数组边界之用。

- **算法实现**

```
void InsertSort(int a[], int n){
 for(int i= 1; i<n; i++){
  if(a[i] < a[i-1]){ //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入
    int j= i-1;
    int x = a[i];    //复制为哨兵，即存储待排序元素
    a[i] = a[i-1]; //先后移一个元素
    while(x < a[j]){ //查找在有序表的插入位置
     a[j+1] = a[j];
     j--;    //元素后移
    }
    a[j+1] = x;  //插入到正确位置
  }
 }
}
```

- **效率**
  时间复杂度：O(n*n)

> 其他的插入排序有二分插入排序，2-路插入排序。

2.1.2 希尔排序

- **基本思想**
  先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。不稳定的排序算法。
- **操作方法**
  - 选择一个增量序列t1，t2，…，tk，其中t1等于 n/2，tk=1；
  - 按增量序列个数k，对序列进行k 趟排序；
  - 每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。
    > 比如10个数，第一轮步长为5，则0,5为子表，第二轮步长为3，则0,3,6为字表进行插入排序
- 算法实现

增量序列d = {n/2 ,n/4, n/8 .....1} n为要排序数的个数

```
void ShellInsertSort(int a[], int n, int dk)
{
 for(int i= dk; i<n; ++i){
  if(a[i] < a[i-dk]){   //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入
   int j = i-dk;
   int x = a[i];    //复制为哨兵，即存储待排序元素
   a[i] = a[i-dk];  //首先后移一个元素
   while(x < a[j]){ //查找在有序表的插入位置
    a[j+dk] = a[j];
    j -= dk;    //元素后移
   }
   a[j+dk] = x; //插入到正确位置
  }
 }
}

/*先按增量d（n/2,n为要排序数的个数进行希尔排序 */
void shellSort(int a[], int n){
 int dk = n/2;
 while( dk >= 1 ){
  ShellInsertSort(a, n, dk);
  dk = dk/2;
 }
}
```

希尔排序时效分析很难，关键码的比较次数与记录移动次数依赖于增量因子序列d的选取，增量因子序列可以有各种取法，有取奇数的，也有取质数的，但需要注意：增量因子中除1 外没有公因子，且最后一个增量因子必须为1。

2.2 选择排序

2.2.1 直接选择排序

在要排序的一组数中，选出最小（或者最大）的一个数与第1个位置的数交换；然后在剩下的数当中再找最小（或者最大）的与第2个位置的数交换，依次类推，直到第n-1个元素（倒数第二个数）和第n个元素（最后一个数）比较为止。**不稳定的排序算法。**

- **操作方法：**
  - 第一趟，从n 个记录中找出关键码最小的记录与第一个记录交换；
  - 第二趟，从第二个记录开始的n-1 个记录中再选出关键码最小的记录与第二个记录交换；
  - 以此类推.....第i 趟，则从第i 个记录开始的n-i+1 个记录中选出关键码最小的记录与第i 个记录交换，直到整个序列按关键码有序。
- **算法实现**

```
int SelectMinKey(int a[], int n, int i){
 int k = i;
 for(int j=i+1 ;j< n; ++j)
  if(a[k] > a[j]) k = j;
 return k;
}
/* 选择排序 */
void selectSort(int a[], int n){
 int key, tmp;
 for(int i = 0; i< n; ++i) {
  key = SelectMinKey(a, n,i); //选择最小的元素
  if(key != i){
   tmp = a[i]; a[i] = a[key]; a[key] = tmp; //最小元素与第i位置元素互换
  }
 }
}
```

- **改进方案**

  二元选择排序：在直接选择排序中，每趟循环只能确定一个元素。所以我们可以考虑改进为每趟循环确定当前趟最大和最小值的位置,从而减少排序所需的循环次数。改进后对n个数据进行排序，最多只需进行[n/2]趟循环即可。

```
void SelectSort(int r[],int n) {
 int i ,j , min ,max, tmp;
 for (i=1 ;i <= n/2;i++) {
  // 做不超过n/2趟选择排序
  min = i; max = i ; //分别记录最大和最小关键字记录位置
  for (j= i+1; j<= n-i; j++) {
```

```
    if (r[j] > r[max]) {
     max = j ; continue ;
    }
    if (r[j]< r[min])
     min = j ;
    }
    //该交换操作还可分情况讨论以提高效率
    tmp = r[i-1]; r[i-1] = r[min]; r[min] = tmp;
    tmp = r[n-i]; r[n-i] = r[max]; r[max] = tmp;
  }
}
```

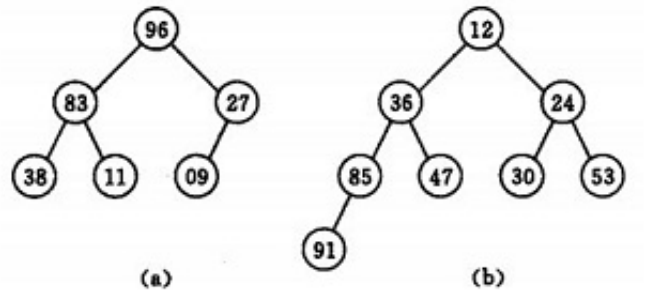
2.2.2 堆排序

堆排序是一种树形选择排序，是对直接选择排序的有效改进。

- **基本思想：**
  堆的定义如下：具有n个元素的序列（k1,k2,...,kn),当且仅当满足

$$\begin{cases} k_i \leqslant k_{2i} \\ k_i \leqslant k_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geqslant k_{2i} \\ k_i \geqslant k_{2i+1} \end{cases} \quad \left(i = 1,2,\cdots,\left\lfloor \frac{n}{2} \right\rfloor\right)$$

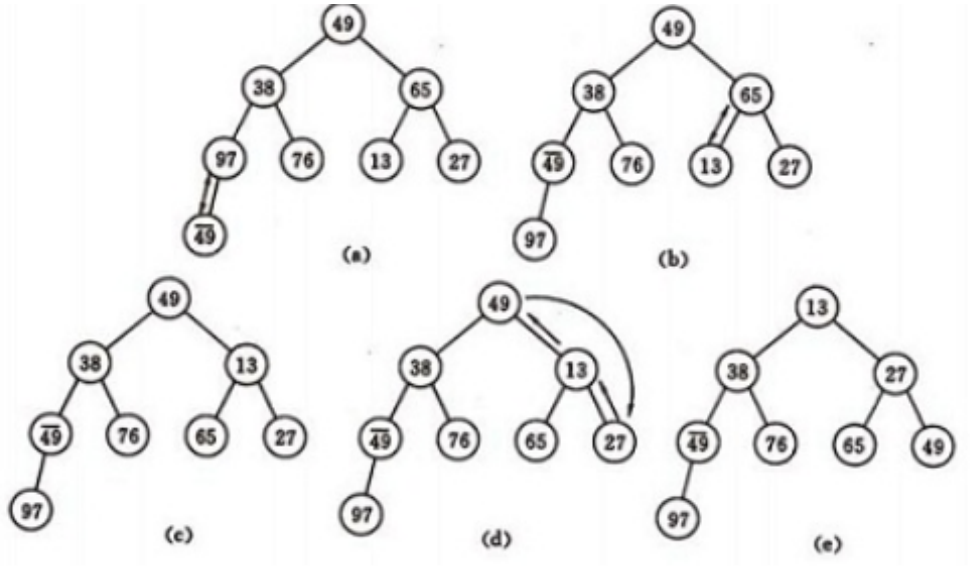
  时称之为堆。由堆的定义可以看出，堆顶元素（即第一个元素）必为最小项（小顶堆）。
  若以一维数组存储一个堆，则堆对应一棵完全二叉树，且所有非叶结点的值均不大于(或不小于)其子女的值，根结点（堆顶元素）的值是最小(或最大)的。如：
  （a）大顶堆序列：（96, 83,27,38,11,09)
  （b）小顶堆序列：（12，36，24，85，47，30，53，91）



初始时把要排序的n个数的序列看作是一棵顺序存储的二叉树（一维数组存储二叉树），调整它们的存储序，使之成为一个堆，将堆顶元素输出，得到n 个元素中最小(或最大)的元素，这时堆的根节点的数最小（或者最大）。然后对前面(n-1)个元素重新调整使之成为堆，输出堆顶元素，得到n 个元素中次小(或次大)的元素。

依此类推，直到只有两个节点的堆，并对它们作交换，最后得到有n个节点的有序序列。称这个过程为堆排序。**是一个不稳定的排序算法。**

- 调整方法
  - **再讨论对n 个元素初始建堆的过程。**
    建堆方法：对初始序列建堆的过程，就是一个反复进行筛选的过程。
    1）n 个结点的完全二叉树，则最后一个结点是第个结点的子树。
    2）筛选从第个结点为根的子树开始，该子树成为堆。
    3）之后向前依次对各结点为根的子树进行筛选，使之成为堆，直到根结点。
    如图建堆初始过程：无序序列：（49，38，65，97，76，13，27，49）

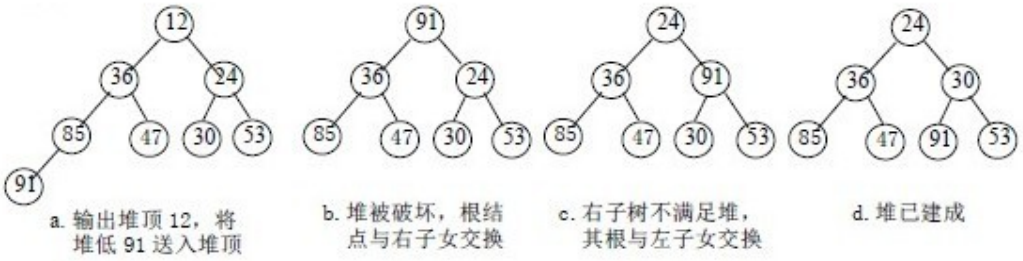


（a）无序序列；（b）97 被筛选之后的状态；（c）65 被筛选之后的状态；
（d）38 被筛选之后的状态；（e）49 被筛选之后建成的堆

  - 输出堆顶元素后，对剩余n-1元素重新建成堆的调整过程。即调整小顶堆的方法：
    1）设有m 个元素的堆，输出堆顶元素后，剩下m-1 个元素。将堆底元素送入堆顶（（最后一个元素与堆顶进行交换），堆被破坏，其原因仅是根结点不满足堆的性质。
    2）将根结点与左、右子树中较小元素的进行交换。
    3）若与左子树交换：如果左子树堆被破坏，即左子树的根结点不满足堆的性质，则重复方法（2）.
    4）若与右子树交换，如果右子树堆被破坏，即右子树的根结点不满足堆的性质。则重复方法（2）.
    5）继续对不满足堆性质的子树进行上述交换操作，直到叶子结点，堆被建

成。

称这个自根结点到叶子结点的调整过程为筛选。如图：



- **算法实现**

  堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数实现排序的函数。

```
void HeapAdjust(int H[],int s, int length){
 int tmp = H[s];
 int child = 2*s+1; //左孩子结点的位置。(i+1 为当前调整结点的右孩子结点的位置)
    while (child < length) {
  if(child+1 <length && H[child]<H[child+1]) { // 如果右孩子大于左孩子(找到比当前待调整结点大的孩子结点)
   ++child ;
  }
  if(H[s]<H[child]) { // 如果较大的子结点大于父结点
   H[s] = H[child]; // 那么把较大的子结点往上移动，替换它的父结点
   s = child;   // 重新设置s ,即待调整的下一个结点的位置
   child = 2*s+1;
  } else {  // 如果当前待调整结点大于它的左右孩子，则不需要调整，直接退出
    break;
  }
  H[s] = tmp;   // 当前待调整的结点放到比其大的孩子结点位置上
 }
 print(H,length);
}
/**
 * 初始堆进行调整
 * 将H[0..length-1]建成堆
 * 调整完之后第一个元素是序列的最小的元素
 */
void BuildingHeap(int H[], int length){
```

```
 //最后一个有孩子的节点的位置 i= (length -1) / 2
 for (int i = (length -1) / 2 ; i >= 0; --i)
  HeapAdjust(H,i,length);
}
/**堆排序算法 */
void HeapSort(int H[],int length){
    //初始堆
 BuildingHeap(H, length);
 //从最后一个元素开始对序列进行调整
 for (int i = length - 1; i > 0; --i) {
  //交换堆顶元素H[0]和堆中最后一个元素
  int temp = H[i]; H[i] = H[0]; H[0] = temp;
  //每次交换堆顶元素和堆中最后一个元素之后，都要对堆进行调整
  HeapAdjust(H,0,i);
  }
}
```

时间主要在堆的建立上，设树的深度为$k=\lfloor \log_2 n \rfloor+1$，从根到叶的筛选，元素的比较次数最多为2*（k-1）,建堆完成后把根节点和最左页节点交换。需要置换n个元素，所以最大时间复杂度为nlogn。

2.3 交换排序

2.3.1 冒泡排序

- **基本思想：**
  在要排序的一组数中，对当前还未排好序的范围内的全部数，自上而下对相邻的两个数依次进行比较和调整，让较大的数往下沉，较小的往上冒。即：每当两相邻的数比较后发现它们的排序与排序要求相反时，就将它们互换。
- **冒泡排序的改进算法**
  - 设置一标志性变量pos,用于记录每趟排序中最后一次进行交换的位置。由于pos位置之后的记录均已交换到位,故在进行下一趟排序时只要扫描到pos位置即可。
  - 传统冒泡排序中每一趟排序操作只能找到一个最大值或最小值,我们考虑利用在每趟排序中进行正向和反向两遍冒泡的方法一次可以得到两个最终值(最大者和最小者) , 从而使排序趟数几乎减少了一半。
- **算法实现：**

```
void Bubble_1 ( int r[], int n) {
 int i= n -1; //初始时,最后位置保持不变
 while ( i> 0) {
  int pos= 0; //每趟开始时,无记录交换
  for (int j= 0; j< i; j++)
   if (r[j]> r[j+1]) {
    pos= j; //记录交换的位置
    int tmp = r[j]; r[j]=r[j+1];r[j+1]=tmp;
   }
  i= pos; //为下一趟排序作准备
  }
}

void Bubble_2 ( int r[], int n){
 int low = 0;
 int high= n -1; //设置变量的初始值
 int tmp,j;
 while (low < high) {
  for (j= low; j< high; ++j) //正向冒泡,找到最大者
   if (r[j]> r[j+1]) {
    tmp = r[j]; r[j]=r[j+1];r[j+1]=tmp;
   }
  --high;   //修改high值, 前移一位
  for ( j=high; j>low; --j) //反向冒泡,找到最小者
   if (r[j]<r[j-1]) {
    tmp = r[j]; r[j]=r[j-1];r[j-1]=tmp;
   }
  ++low;    //修改low值,后移一位
 }
}
```

2.3.2 快速排序

- **基本思想：**

  1）选择一个基准元素,通常选择第一个元素或者最后一个元素,

  2）通过一趟排序讲待排序的记录分割成独立的两部分，其中一部分记录的元素值均比基准元素值小。另一部分记录的 元素值比基准值大。

  3）此时基准元素在其排好序后的正确位置

4）然后分别对这两部分记录用同样的方法继续进行排序，直到整个序列有序。

- 算法的实现

  1、递归实现

```
void swap(int *a, int *b){
 int tmp = *a; *a = *b; *b = tmp;
}

int partition(int a[], int low, int high){
 int privotKey = a[low];     //基准元素
 while(low < high){ //从表的两端交替地向中间扫描
  while(low < high && a[high] >= privotKey) --high; //从high 所指位置向前搜索，至多到low+1 位置。将比基准元素小的交换到低端
  swap(&a[low], &a[high]);
  while(low < high && a[low] <= privotKey ) ++low;
  swap(&a[low], &a[high]);
 }
 return low;
}
void quickSort(int a[], int low, int high){
 if(low < high){
  int privotLoc = partition(a, low, high); //将表一分为二
  quickSort(a, low, privotLoc -1);  //递归对低子表递归排序
  quickSort(a, privotLoc + 1, high);     //递归对高子表递归排序
 }
}
```

快速排序是通常被认为在同数量级（O(nlog2n)）的排序方法中**平均性能最好的**。但若初始序列按关键码有序或基本有序时，快排序反而蜕化为冒泡排序。为改进之，通常以**“三者取中法”**来选取基准记录，即将排序区间的两个端点与中点三个记录关键码居中的调整为支点记录。快速排序是一个**不稳定的排序方法**。

- **快速排序算法的改进**

  在本改进算法中,只对长度大于k的子序列递归调用快速排序,让原序列基本有序，然后再对整个基本有序序列用插入排序算法排序。

  实践证明，改进后的算法时间复杂度有所降低，且当k取值为 8 左右时,改进算法的性能最佳。

  - 算法实现如下：

```
void swap(int *a, int *b){
 int tmp = *a;*a = *b; *b = tmp;
}
int partition(int a[], int low, int high){
 int privotKey = a[low];    //基准元素
 while(low < high){ //从表的两端交替地向中间扫描
  while(low < high && a[high] >= privotKey) --high; //从high 所指位置向前搜索，至多到low+1 位置。将比基准元素小的交换到低端
  swap(&a[low], &a[high]);
  while(low < high && a[low] <= privotKey ) ++low;
  swap(&a[low], &a[high]);
 }
 return low;
}


void qsort_improve(int r[ ],int low,int high, int k){
 if( high -low > k ) { //长度大于k时递归, k为指定的数
  int pivot = partition(r, low, high); // 调用的Partition算法保持不变
  qsort_improve(r, low, pivot - 1,k);
  qsort_improve(r, pivot + 1, high,k);
 }
}
void quickSort(int r[], int n, int k){
 qsort_improve(r,0,n,k);//先调用改进算法Qsort使之基本有序
 //再用插入排序对基本有序序列排序
 for(int i=1; i<=n;i ++){
  int tmp = r[i];
  int j=i-1;
  while(tmp < r[j]){
   r[j+1]=r[j]; j=j-1;
  }
  r[j+1] = tmp;
 }
}
```

2.4 归并排序

- **基本思想：**

归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

- **合并方法：**
  设r[i…n]由两个有序子表r[i…m]和r[m+1…n]组成，两个子表长度分别为m-i +1、n-m。

  a. j=m+1；k=i；i=i; //置两个子表的起始下标及辅助数组的起始下标

  b. 若i>m 或j>n，转(4) //其中一个子表已合并完，比较选取结束

  c. //选取r[i]和r[j]较小的存入辅助数组rf
  如果r[i]<r[j]，rf[k]=r[i]； i++； k++； 转(2)
  否则，rf[k]=r[j]； j++； k++； 转(2)

  d. //将尚未处理完的子表中元素存入rf
  如果i<=m，将r[i…m]存入rf[k…n] //前一子表非空
  如果j<=n，将r[j…n] 存入rf[k…n] //后一子表非空

  e. 合并结束。

```
//将r[i…m]和r[m +1 …n]归并到辅助数组rf[i…n]
void Merge(ElemType *r,ElemType *rf, int i, int m, int n){
 int j,k;
 for(j=m+1,k=i; i<=m && j <=n ; ++k){
  if(r[j] < r[i]) rf[k] = r[j++];
  else rf[k] = r[i++];
 }
 while(i <= m) rf[k++] = r[i++];
 while(j <= n) rf[k++] = r[j++];
}
```

- **归并的迭代算法**
  1 个元素的表总是有序的。所以对n 个元素的待排序列，每个元素可看成1 个有序子表。对子表两两合并生成n/2个子表，所得子表除最后一个子表长度可能为1外，其余子表长度均为2。再进行两两合并，直到生成n 个元素按关键码有序的表。

```
//将r[i…m]和r[m +1 …n]归并到辅助数组rf[i…n]
void Merge(ElemType *r,ElemType *rf, int i, int m, int n){
```

```
 int j,k;
 for(j=m+1,k=i; i<=m && j <=n ; ++k){
  if(r[j] < r[i]) rf[k] = r[j++];
  else rf[k] = r[i++];
 }
 while(i <= m) rf[k++] = r[i++];
 while(j <= n) rf[k++] = r[j++];
 print(rf,n+1);
}

void MergeSort(ElemType *r, ElemType *rf, int lenght){
 int len = 1;
 ElemType *q = r ;
 ElemType *tmp ;
 while(len < lenght) {
  int s = len;
  len = 2 * s ;
  int i = 0;
  while(i+ len <lenght){
   Merge(q, rf, i, i+ s-1, i+ len-1 ); //对等长的两个子表合并
   i = i+ len;
  }
  if(i + s < lenght){
   Merge(q, rf, i, i+ s -1, lenght -1); //对不等长的两个子表合并
  }
  tmp = q; q = rf; rf = tmp; //交换q,rf，以保证下一趟归并时，仍从q 归并到rf
 }
}
```

- **两路归并的递归算法**

```
void MSort(ElemType *r, ElemType *rf,int s, int t){
 ElemType *rf2;
 if(s==t) r[s] = rf[s];
 else{
  int m=(s+t)/2;     /*平分*p 表*/
  MSort(r, rf2, s, m);  /*递归地将p[s…m]归并为有序的p2[s…m]*/
  MSort(r, rf2, m+1, t);     /*递归地将p[m+1…t]归并为有序的p2[m+1…t]*/
  Merge(rf2, rf, s, m+1,t); /*将p2[s…m]和p2[m+1…t]归并到p1[s…t]*/
```

```
 }
}
void MergeSort_recursive(ElemType *r, ElemType *rf, int n){
 /*对顺序表*p 作归并排序*/
 MSort(r, rf,0, n-1);
}
```

2.5 桶排序/基数排序

2.5.1 桶排序

- **基本思想**
  是将阵列分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法或是以递回方式继续使用桶排序进行排序）。
  桶排序是鸽巢排序的一种归纳结果。当要被排序的阵列内的数值是均匀分配的时候，桶排序使用线性时间（Θ（n））。但桶排序并不是比较排序，他不受到 O(n log n) 下限的影响。
  简单来说，就是把数据分组，放在一个个的桶中，然后对每个桶里面的在进行排序。
- **实现过程**
  例如要对大小为[1..1000]范围内的n个整数A[1..n]排序
  - 首先，可以把桶设为大小为10的范围，具体而言，设集合B[1]存储[1..10]的整数，集合B[2]存储 (10..20]的整数，……集合B[i]存储((i-1)*10, i*10]的整数，i = 1,2,..100。总共有 100个桶。
  - 然后，对A[1..n]从头到尾扫描一遍，把每个A[i]放入对应的桶B[j]中。 再对这100个桶中每个桶里的数字排序，这时可用冒泡，选择，乃至快排，一般来说任 何排序法都可以。
  - 最后，依次输出每个桶里面的数字，且每个桶中的数字从小到大输出，这 样就得到所有数字排好序的一个序列了。

> 假设有n个数字，有m个桶，如果数字是平均分布的，则每个桶里面平均有n/m个数字。如果对每个桶中的数字采用快速排序，那么整个算法的复杂度是O(n + m * n/m*log(n/m)) = O(n + nlogn - nlogm)，从上式看出，当m接近n的时候，桶排序复杂度接近O(n) 。
> 当然，以上复杂度的计算是基于输入的n个数字是平均分布这个假设的。实际应用中效果并没有这么好。如果所有的数字都落在同一个桶中，那就退化成一般的排

> 序了。

前面说的几大排序算法 ，大部分时间复杂度都是O（n2），也有部分排序算法时间复杂度是O(nlogn)。而桶式排序却能实现O（n）的时间复杂度。但桶排序的缺点是：
1）首先是空间复杂度比较高，需要的额外开销大。排序有两个数组的空间开销，一个存放待排序数组，一个就是所谓的桶，比如待排序值是从0到m-1，那就需要m个桶，这个桶数组就要至少m个空间。
2）其次待排序的元素都要在一定的范围内等等。
桶式排序是一种分配排序。分配排序的特定是不需要进行关键码的比较，但前提是要知道待排序列的一些具体情况。

> 分配排序的基本思想：说白了就是进行多次的桶式排序。

2.5.2 基数排序

基数排序过程无须比较关键字，而是通过“分配”和“收集”过程来实现排序。它们的时间复杂度可达到线性阶：O(n)。
设n 个元素的待排序列包含d 个关键码{k1，k2，…，kd}，则称序列对关键码{k1，k2，…，kd}有序是指：对于序列中任两个记录r[i]和rj都满足下列有序关系：

$$(k_i^1, k_i^2, \quad , k_i^d) < (k_j^1, k_j^2, \quad , k_j^d)$$

其中k1 称为最主位关键码，kd 称为最次位关键码 。

- **两种多关键码排序方法：**
  多关键码排序按照从最主位关键码到最次位关键码或从最次位到最主位关键码的顺序逐次排序，分两种方法：
  - 最高位优先(Most Significant Digit first)法，简称MSD 法：
    1）先按k1 排序分组，将序列分成若干子序列，同一组序列的记录中，关键码k1 相等。
    2）再对各组按k2 排序分成子组，之后，对后面的关键码继续这样的排序分组，直到按最次位关键码kd 对各子组排序后。
    3）再将各组连接起来，便得到一个有序序列。扑克牌按花色、面值排序中介绍的方法一即是MSD 法。
  - 最低位优先(Least Significant Digit first)法，简称LSD 法：
    1) 先从kd 开始排序，再对kd-1进行排序，依次重复，直到按k1排序分组分成最小的子序列后。
    2) 最后将各个子序列连接起来，便可得到一个有序的序列, 扑克牌按花色、面值排序中介绍的方法二即是LSD 法。

- **基于LSD方法的链式基数排序的基本思想**
  “多关键字排序”的思想实现“单关键字排序”。对数字型或字符型的单关键字，可以看作由多个数位或多个字符构成的多关键字，此时可以采用“分配-收集”的方法进行排序，这一过程称作基数排序法，其中每个数字或字符可能的取值个数称为基数。比如，扑克牌的花色基数为4，面值基数为13。在整理扑克牌时，既可以先按花色整理，也可以先按面值整理。按花色整理时，先按红、黑、方、花的顺序分成4摞（分配），再按此顺序再叠放在一起（收集），然后按面值的顺序分成13摞（分配），再按此顺序叠放在一起（收集），如此进行二次分配和收集即可将扑克牌排列有序。
- **基数排序:**
  是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。
- 算法实现：

```
Void RadixSort(Node L[],length,maxradix)
{
   int m,n,k,lsp;
   k=1;m=1;
   int temp[10][length-1];
   Empty(temp); //清空临时空间
   while(k<maxradix) //遍历所有关键字
   {
     for(int i=0;i<length;i++) //分配过程
    {
       if(L[i]<m)
          Temp[0][n]=L[i];
       else
          Lsp=(L[i]/m)%10; //确定关键字
       Temp[lsp][n]=L[i];
       n++;
   }
   CollectElement(L,Temp); //收集
   n=0;
   m=m*10;
  k++;
 }
```

```
23 }
```

参考链接

https://blog.csdn.net/xiazdong/article/details/8462393
https://blog.csdn.net/hguisu/article/details/7776068