排序

▼ 概念

数据表：待排序的记录的有限集合

关键字：记录中的属性域

排序：使数据表**按关键字线性有序**

稳定性：关键字相同的记录在排序过程中保持前后次序不变

- **稳定**：冒泡、插入、归并、基数
- **不稳定**：希尔、快排、选择、堆排
- **平均时间复杂度**
  - $O(n^2)$：冒泡、插入、选择
  - $O(n\log_2 n)$：快排、堆排、归并（且这三个最好最坏平均相同）
- 快排最好 $O(n\log_2 n)$，最坏 $O(n^2)$，在待排序列接近有序时最坏。
- 选择排序最好也只有 $O(n^2)$，冒插希最好有 $O(n)$
- **空间复杂度**
  - 快排 $O(\log_2 n)$，归并 $O(n)$

考察最好时间复杂度：

10．下列排序算法中，对初始状态为递增序列的表按递增顺序排序，最省时间的是（B）。

A．快速排序 $O(n^2)$　B．起泡排序 $O(n)$　C．归并排序 $O(n\log n)$　D．简单选择排序 $O(n^2)$

▼ 插入排序

▼ 直接插入排序

- 时间复杂度：$O(n^2)$

- 空间复杂度：$O(1)$
- 稳定

不带哨兵

```
void InsertSort(int arr[],int len){
    int i,j,tmp;
    for(i=1;i<len;i++){
        tmp=arr[i];
        for(j=i-1;j>=0 && arr[j]>tmp;j--){
            arr[j+1]=arr[j];
        }
        arr[j+1]=tmp;
    }
}
```

带哨兵

```
void InsertSort_2(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++){
        A[0]=A[i];
        for(j=i-1;A[0]<A[j];j--){
            A[j+1]=A[j];
        }
        A[j+1]=A[0];
    }
}
```

最好情况（正序）

比较次数：$\sum_{i=2}^{n} 1 = n-1$

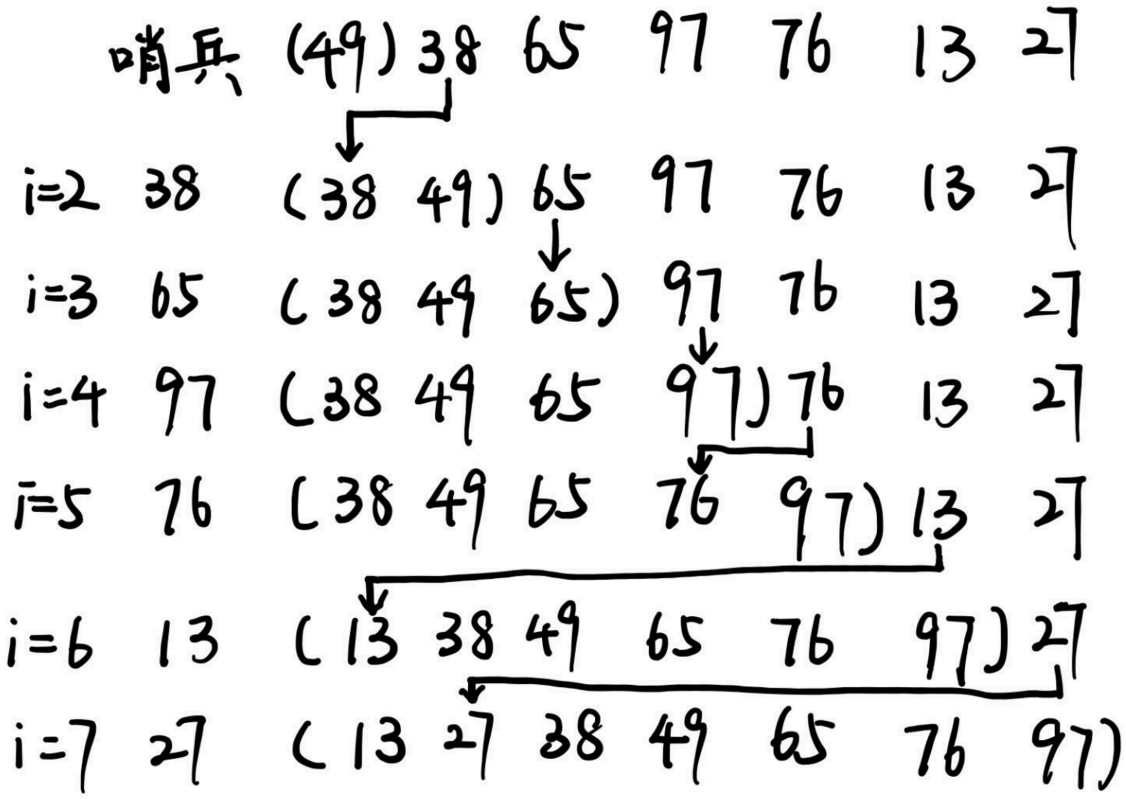
移动次数：0

最坏情况（逆序）

比较次数：$\sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}$ （跟前 $n-1$个数、哨兵比较）

移动次数：$\sum_{i=2}^{n}(i+1)=\frac{(n+4)(n-1)}{2}$（设置哨兵1+后移 i-1+移动哨兵至目标位置1=i+1）

手算模拟



▼ 折半插入排序

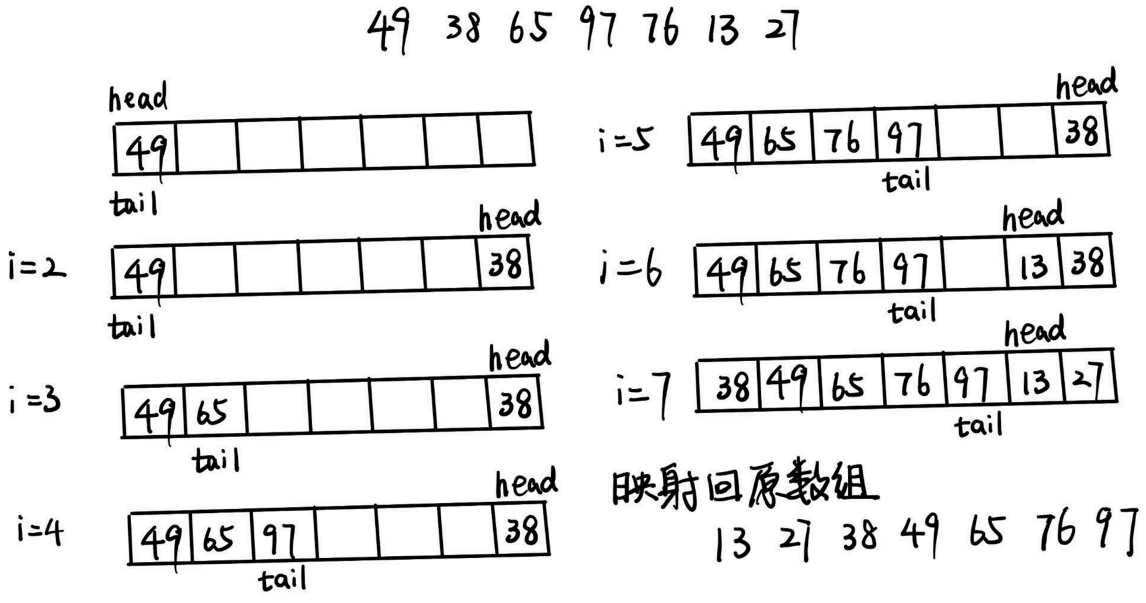
- 时间复杂度 $O(n^2)$
- 可减少比较次数，移动次数不变

```
void BiInsertSort(int A[],int n){
    int i,j,low,high,mid;
    for(i=2;i<=n;i++){
        A[0]=A[i];
        low=1;
        high=i-1;
        while(low<=high){
            mid=(low+high)/2;
            //找到第一个不小于key的位置(最右边的可插位置)
```

```
            if(A[mid]>A[0]) high=mid-1;
            else low=mid+1;
        }
        for(j=i-1;j>=high+1;j--){
            A[j+1]=A[j];
        }
        A[high+1]=A[0];
    }
}
```

▼ 2路插入排序

- 增加辅助循环队列
- 可减少移动次数
- 时间复杂度 $O(n^2)$




```
//下标从0开始
void InsertSort2(int arr[],int n){
    int TP[]={0};
    int i,j,head=0,tail=0;
    TP[0]=arr[0];
```

```
    for(i=1;i<n;i++){
        //小于最小，直接存
        if(arr[i]<TP[head]){
            head=(head-1+n)%n;
            TP[head]=arr[i];
        }
        //大于最大，直接存
        else if(arr[i]>TP[tail]){
            tail++;
            TP[tail]=arr[i];
        }
        else{
            tail++;
            TP[tail]=TP[tail-1];
            for(j=tail-1;arr[i]<TP[(j-1+n)%n];j=(j-1+n)%n){
                TP[j]=TP[(j-1+n)%n];
            }
            TP[j]=arr[i];
        }
    }
    for(i=0;i<n;i++){
        arr[i]=TP[head];
        head=(head+1)%n;
    }
}
```

▼ 表插入排序

- 使用链表
- 无需数据移动，但必须用线性查找
- 时间复杂度 $O(n^2)$

初始化

	0	1	2	3	4	5	6	7
key	M	49	38	65	97	76	13	27
next	1	0						

i=2

	0	1	2	3	4	5	6	7
key	M	49	38	65	97	76	13	27
next	2	0	1					

i=3

	0	1	2	3	4	5	6	7
key	M	49	38	65	97	76	13	27
next	2	3	1	0				

i=4

	0	1	2	3	4	5	6	7
key	M	49	38	65	97	76	13	27
next	2	3	1	4	0			

i=5

	0	1	2	3	4	5	6	7
key	M	49	38	65	97	76	13	27
next	2	3	1	5	0	4		

i=6

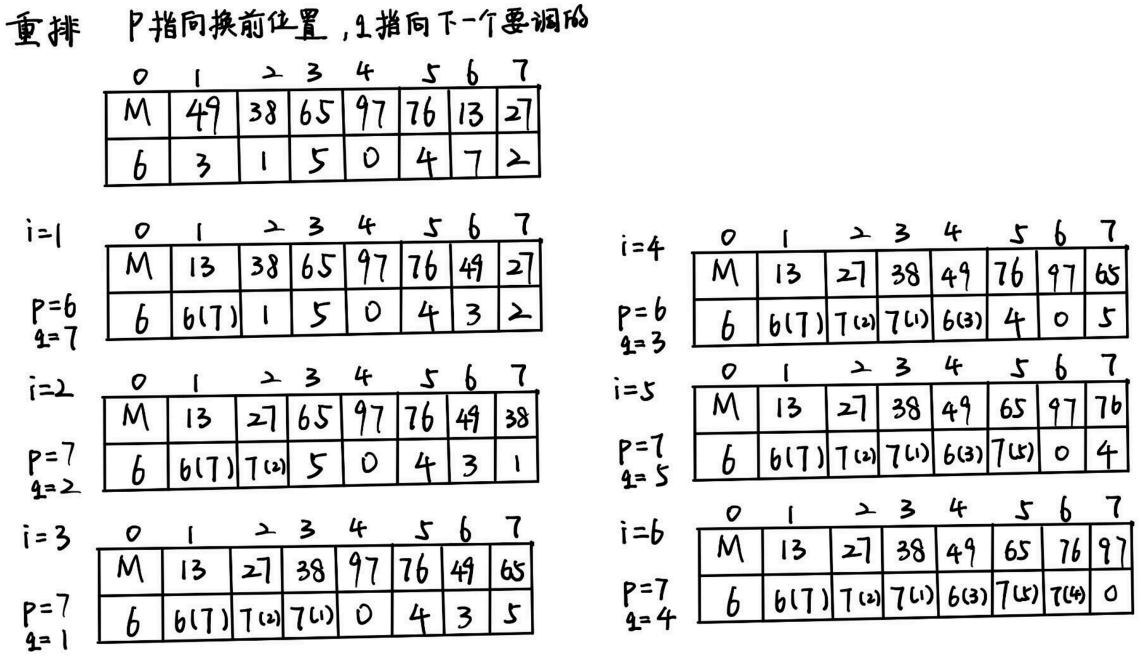
	0	1	2	3	4	5	6	7
key	M	49	38	65	97	76	13	27
next	6	3	1	5	0	4	2	

i=7

	0	1	2	3	4	5	6	7
key	M	49	38	65	97	76	13	27
next	6	3	1	5	0	4	7	2

```
void TableInsertSort(int tb[],int n){
    //初始化 表头与第一个元素构成循环链表
    tb[0].data=MAXVALUE;
    tb[0].link=1;
    tb[1].link=0;

    int p,q;
    //逐渐加入元素
    for(int i=2;i<n;i++){
        p=t[0].link;  //每次从头开始遍历
        q=0;        //保存前驱
        while(p!=0 && t[p].data<=t[i].data){
            q=p;
            p=t[p].link;
        }
        t[i].link=t[q].link; //在q后插入
        t[q].link=i;
    }
}
```




```
void Arrange(SLinkListType &SL){
    p=SL.r[0].next;   //p指向第一小的元素位置
    //将i小的元素放到i位置
    for(i=1;i<L.length-1;i++){
        //p<i说明要找的位置的元素已经被换走
        //通过next记录的调换位置寻找之前的元素
        while(p<i) p=SL.r[p].next;
        //q记录下一个待调整的元素
        q=SL.r[p].next;
        //需要调换
        if(p!=i){
            SL.r[p]←→SL.r[i]; //调换数据
            SL.r[i].next=p;    //next中保存被换走的元素的新位置
        }
        p=q;
    }
}
```

▼ 希尔排序

- 时间复杂度与增量序列dlta有关

- 使用Hibbard增量序列： $h_k = 2^k - 1$，时间复杂度为 $O(n^{\frac{3}{2}})$
- 空间复杂度：$O(1)$
- 不稳定

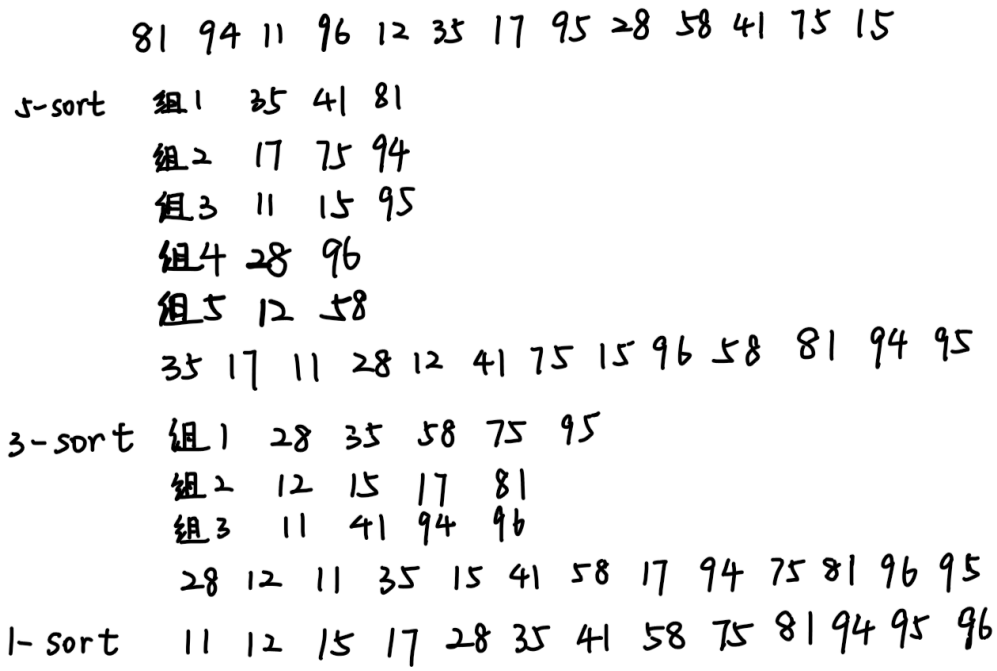
```
//下标从1开始
//增量为dk的一次插入排序
void ShellInsert(SqList &L,int dk){
    int i,j;
    for(i=dk+1;i<=L.length;i++){
        L.r[0]=L.r[i];
        for(j=i-dk;j>0 && L.r[0]<L.r[j];j-=dk){
            L.r[j+dk]=L.r[j];
        }
        L.r[j+dk]=L.r[0];
    }
}
//传入一组增量
void ShellSort(SqList &L,int dlta[],int t){
    for(int k=0;k<t;t++){
        ShellInsert(L,dlta[k]);
    }
}
```

- 使用Shell增量
- 时间复杂度 $O(n^2)$

```
//下标从0开始
void ShellSort(int arr[],int len){
    int gap,i,j;
    for(gap=len/2;gap>0;gap/=2){//步长变化
        for(i=gap;i<len;i++){
            int tmp=arr[i];
            for(j=i-gap;j>=0&&arr[j]>tmp;j-=gap){
                arr[j+gap]=arr[j];
```

```
        }
        arr[j+gap]=tmp;
      }
    }
  }
```

手算模拟



▼ 交换排序

▼ 冒泡排序

- 时间复杂度 $O(n^2)$
- 稳定

最好情况（正序）：

比较次数： $n-1$

移动次数： 0

最坏情况（逆序）：

比较次数： $\sum_{i=1}^{n}(i-1)=\frac{n(n-1)}{2}$

移动次数： $3 \times \sum_{i=1}^{n}(i-1) = \frac{3n(n-1)}{2}$ 一次交换需要3次移动

```
//从前往后冒（i从0开始）
void BubbleSort(int arr[], int len) {
    for(int i=0;i<len-1;i++){
        for(int j=0;j<len-1-i;j++) {
            if(arr[j]>arr[j+1]) {
                swap(arr+j,arr+j+1);
            }
        }
    }
}
//从前往后冒（i从len-1开始）
void BubbleSort(int arr[], int len) {
    for(int i=len-1;i>=1;i--){
        for(int j=0;j<i;j++) {
            if(arr[j]>arr[j+1]) {
                swap(arr+j,arr+j+1);
            }
        }
    }
}
```

优化方案：

1. 外层优化（减少遍数）：增加flag判断，若没有交换则提前退出
2. 内层优化（减少比较次数）：记录每次最后交换的位置last，表明last之后已经有序

```
//从后往前冒（排序遍数优化）
void BubbleSort(int arr[], int len) {
    int flag;
    for(int i=0;i<len-1;i++){
        flag=0;
        for(int j=len-1;j>i;j--) {
            if(arr[j-1]>arr[j]) {
```

```
            swap(arr+j-1,arr+j);
            flag=1;
        }
    }
    if(!flag) break;
  }
}
//从前往后冒（内层优化）
void BubbleSort(int arr[], int len) {
  int i,j,last;
  i=len-1;
  while(i>0){
    last=0;
    for(j=0;j<i;j++){
      if(arr[j+1]<arr[j]){
        swap(arr+j,arr+j+1);
        last=j;  //记录最后一次交换的位置
              //last后面的已有序
      }
    }
    i=last;  //last后的不再处理
  }
}
```

▼ 快速排序

- 时间复杂度 $O(n \times$ 递归层数$)$

  平均、最好 $O(n\log_2 n)$ 最坏 $O(n^2)$

  最坏情况在待排序列已接近有序时出现

- 空间复杂度 $O($递归层数$)$

  最好 $O(\log_2 n)$ 最坏 $O(n)$

- 不稳定

```
int part(int arr[],int left,int right){
    int pivot=arr[left];
    while(left<right){
        while(left<right&&arr[right]>=pivot) right--;
        if(left<right) arr[left++]=arr[right];
        while(left<right&&arr[left]<=pivot) left++;
        if(left<right) arr[right--]=arr[left];
    }
    arr[left]=pivot;
    return left;
}
void QuickSort(int arr[],int left,int right){
    if(left<right){
        int divide=part(arr,left,right);
        QuickSort(arr,left,divide-1);
        QuickSort(arr,divide+1,right);
    }
}
```

三值取中优化

```
    int mid=(left+right)/2;
    if(arr[left]>arr[mid]&&arr[left]>arr[right])   //left是最大的
        arr[mid]>arr[right]? swap(arr+mid,arr+left):swap(arr+right,arr+left);
    if(arr[left]<arr[mid]&&arr[left]<arr[right])   //left是最小的
        arr[mid]<arr[right]? swap(arr+mid,arr+left):swap(arr+right,arr+left);
```

手算模拟

1. 请使用快速排序算法对序列(52, 49, 80, 36, 14, 75, 58, 97, 23, 61) 由小到大排序，写出排序过程。排序中选择子序列第一个元素为枢轴。

第一轮：枢轴 52

61>52　　58>52

23<52 交换　　75>52

49<52　　14<52 交换

80>52 交换　　36<52

97>52

(23 49 14 36)52(75 58 97 80 61)

第二轮：枢轴 23, 75

52左：36>23　　52右：61<75 交换

14<23 交换　　58<75

49>23 交换　　97>75 交换

得 14 23 49 36　　80>75

得 61 58 75 80 97

(14)23(49 36)52(61 58)75(80 97)

第三轮：枢轴 49, 61, 80

23右：36<49 交换　　75左：58<61 交换　　75右：97>80

得(36)49　　得(58)61　　得 80(97)

14 23 36 49 52 58 61 75 80 97

排序结果 14 23 36 49 52 58 61 75 80 97

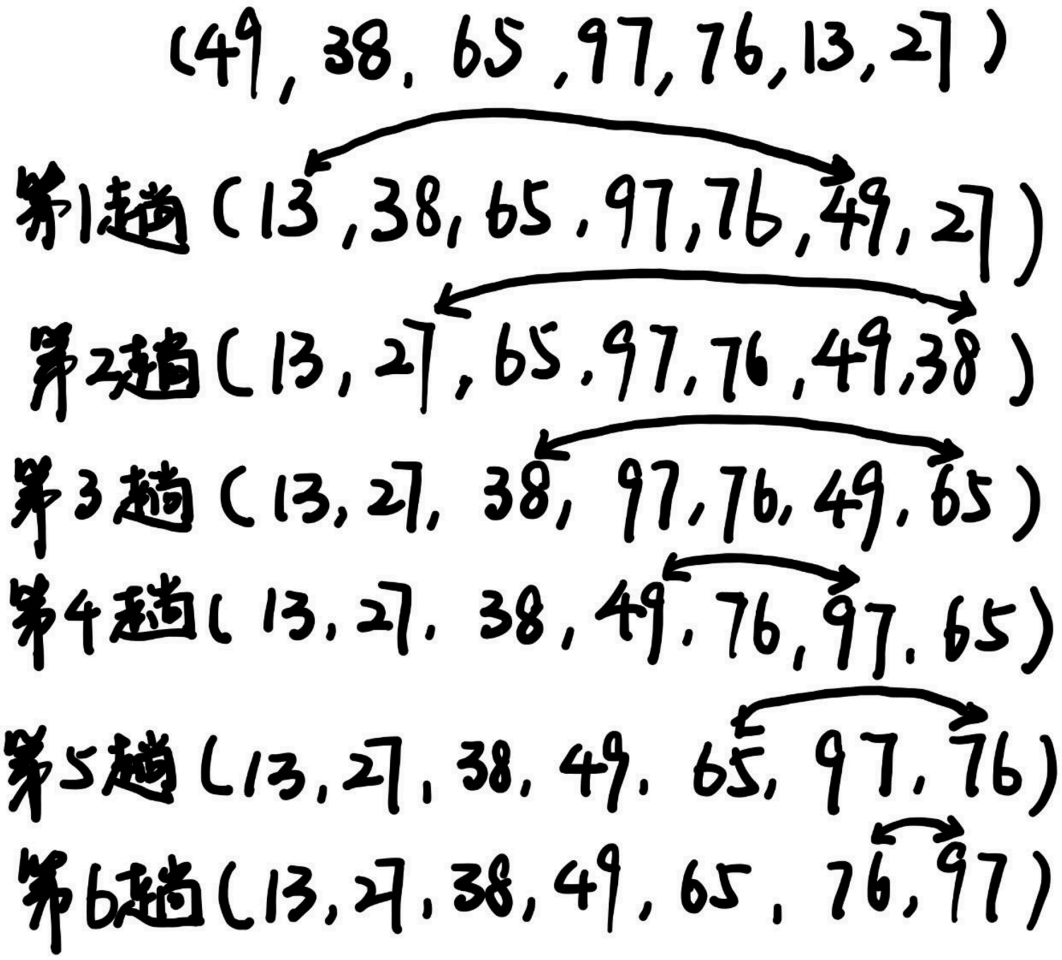
▼ 选择排序

▼ 简单选择排序

- 每趟选出第 i 小的数，与 i 位置的数交换
- 比较次数： $\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

- 移动次数：
  - 最好（正序）：0
  - 最坏（逆序）：$3(n-1)$
- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(1)$
- 不稳定

```
void SelectSort(int A[],int n){
    for(int i=0;i<n-1;i++){
        int min=i;
        for(int j=i+1;j<n;j++){
            if(A[j]<A[min]) min=j;
        }
        if(min!=i) swap(A+i,A+min);
    }
}
```



▼ 树形选择排序

- 时间复杂度：$O(n \log n)$
- 辅助空间过多

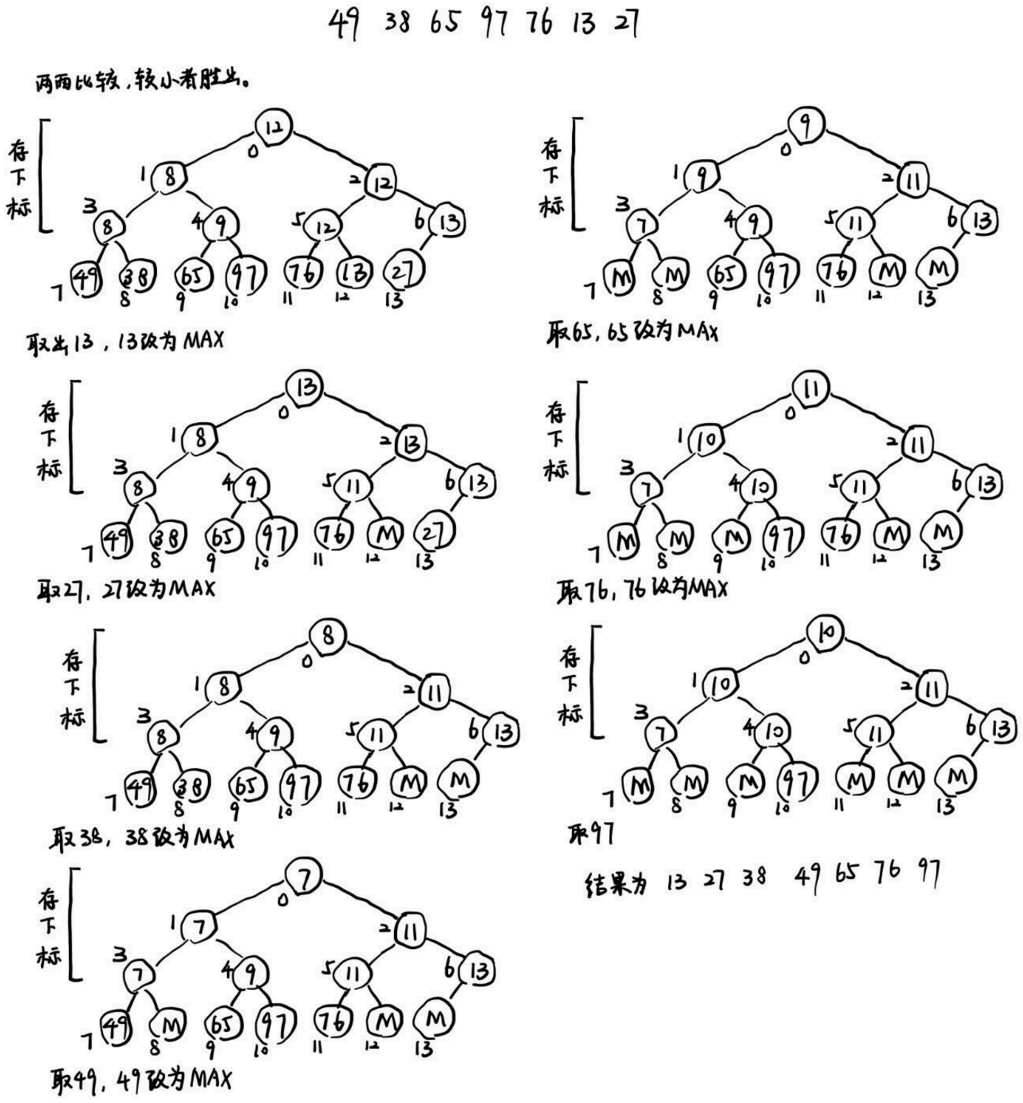
过程：

建树：两两比较，较小者胜出

取出A[0]，设置为MAX

调整：只需调整A[0]所经历的比较分支

手算模拟




```
int GetValue(T e[],int n,int p){
    //结点没有值
    if(p>=n) return MAXVALUE;
    T value;
    //非叶子结点上的分支结点，从下标寻值
    if(p<n/2){
        value=e[e[p]];
    }
    //叶子结点上的分支结点 直接赋值
    else{
        value=e[p];
```

```
    }
    return value;
}
void Play(T e[],int n,int p){
    int left,right;
    int leftvalue,rightvalue;
    //遍历所有分支结点
    while(p>=0){
        left=2*p+1;
        right=2*p+2;
        leftvalue=GetValue(T e[],int n,int p);
        rightvalue=GetValue(T e[],int n,int p);
        if(leftvalue<=rightvalue){
            //非叶子结点上的分支结点，已经是下标
            if(left<n/2){
                e[p]=e[left];
            }
            //叶子结点上的分支结点，要保存其下标
            else{
                e[p]=left;
            }
        }
        else{
            if(right<n/2){
                e[p]=e[right];
            }
            else{
                e[p]=right;
            }
        }
        --p;
    }
}
void Select(T e[],int n,int p){
    int i=p;    //父结点
    int j=2*p+1; //左孩子结点
```

```
    T leftvalue,rightvalue;
    int flag=-1;
    while(flag && i>=0){
        leftvalue=GetValue(e,n,j);
        rightvalue=GetValue(e,n,j+1);
        //小者胜出
        if(leftvalue<=rightvalue){
            if(j<n/2){
                e[i]=e[j];
            }
            else{
                e[i]=j;
            }
        }
        else{
            if((j+1)<n/2){
                e[i]=e[j+1];
            }
            else{
                e[i]=j+1;
            }
        }
        if(i==0) flag++;
        i=(i-1)/2;  //到上一层父结点
        j=2*i+1;
    }
}
void TreeSelectSort(int A[],int n){
    //建树
    int size=2*n-1,i;
    T *e=(T *)malloc(sizeof(T)*size);
    int k=size/2;   //叶子结点开始的下标
    //将待比较序列存到叶子结点
    for(i=0;i<n;i++){
        e[k++]=A[i];
    }
```

```
    int curpos=size/2-1;  //最后一个分支结点
    Play(e,size,curpos);
    i=0;
    //下标为0保存的是最小的数
    A[i]=e[e[0]];
    //将已经取出的位置改为MAX
    e[e[0]]=MAXVALUE;
    //只需要重新比较取出值所比较过的分支
    for(i=1;i<n;i++){
        //当前结点的父结点
        curpos=(e[0]-1)/2;
        Select(e,size,curpos);
    }
}
```

▼ 堆排序

最大堆（大顶堆）：父结点大于等于孩子结点

最小堆（小顶堆）：父结点小于等于孩子结点

- 时间复杂度：$O(n\log n)$
- 空间复杂度：$O(1)$
- 不稳定

过程：

调整为大根堆

交换堆首和堆尾元素

基于大根堆排序

```
//调整以k为根的子树为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];//A[0]暂存根结点
    for(int i=2*k;i<=len;i*=2){
        //i<len才有右孩子
        //取左右孩子中较大者
```

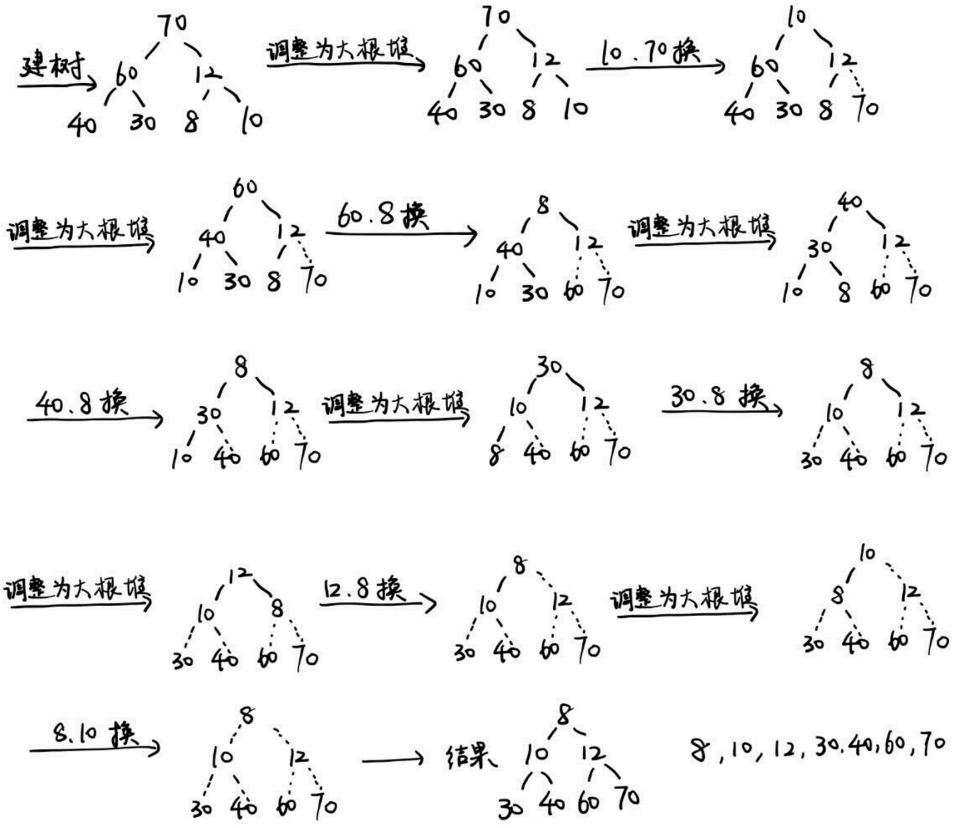
```
        if(i<len&&A[i]<A[i+1])
            i++;
        //根大于孩子,退出
        if(A[0]>=A[i]) break;
        else{
            //根由孩子代替
            A[k]=A[i];
            //取i位置为根可能存放的位置
            //进入下一轮循环,小元素"下坠"
            k=i;
        }
    }
    //放回根
    A[k]=A[0];
}
void BuildMaxHeap(int A[],int len){
    //从后往前调整所有非终端结点
    for(int i=len/2;i>0;i--){
        HeadAdjust(A,i,len);
    }
}

//基于大根堆进行排序
void HeapSort(int A[],int len){
    BuildMaxHeap(A,len);
    for(int i=len;i>1;i--){
        swap(A+i,A+1);//堆顶和堆底交换
        HeadAdjust(A,1,i-1);
    }
}
```

手算模拟

- **从小到大排用大根堆，从大到小排用小根堆**

2. 已知序列（70,60,12,40,30,8,10），请按照堆排序算法对其进行由小到大排序，并写出排序过程。



▼ 归并排序

- 2-路归并排序时间复杂度：$O(n\log n)$

可用于给一个表排序，也可用于合并两个有序表。

```
int *B=(int *)malloc(MAXSIZE*sizeof(int));

//将[low,mid]和[mid+1,high]两部分归并
void Merge(int A[],int low,int mid,int high){
    int i,j,k;
    //复制一份到B
    for(k=low;k<=high;k++){
        B[k]=A[k];
    }
    //i遍历B中第一个部分,j遍历B中第二个部分
```

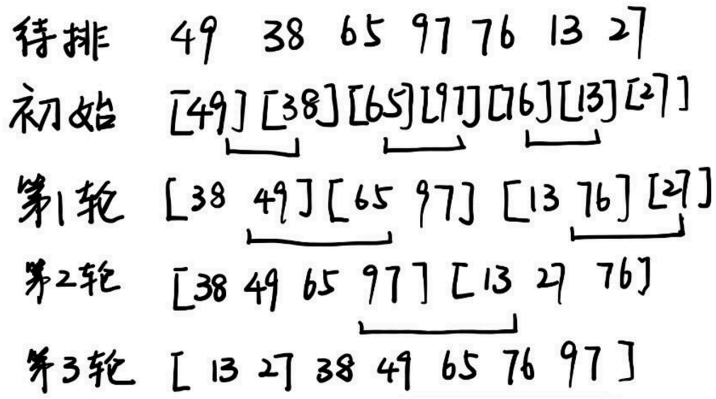
```
    //k为当前要确定的A的元素的下标
    for(i=low,j=mid+1,k=i;i<=mid&&j<=high;k++){
        //B中两个部分,哪个小取哪个
        //相等时优先选靠前的
        if(B[i]<=B[j])
            A[k]=B[i++];
        else
            A[k]=B[j++];
    }
    //处理剩下的
    while(i<=mid) A[k++]=B[i++];
    while(j<=high) A[k++]=B[j++];
}

void MergeSort(int A[],int low,int high){
    if(low<high){
        int mid=(low+high)/2;//从中间划分
        MergeSort(A,low,mid);//对左半部分排序
        MergeSort(A,mid+1,high);
        Merge(A,low,mid,high);
    }

}
```

手算模拟

3. 已知初始序列（49,38,65,97,76,13,27），进行由小到大排序，请写出各趟归并排序结果



▼ 基数排序

按类分配再收集

多关键字排序

例：将扑克牌按关键字 k_0花色、 k_1面值排序

MSD（最高位优先法）：先按花色分成四堆，每堆按面值排序，再收集。

LSD（最低位优先法）：先按面值分成13堆，每堆按花色排序，再收集。

链式基数排序

初始化：设置r个空队列。

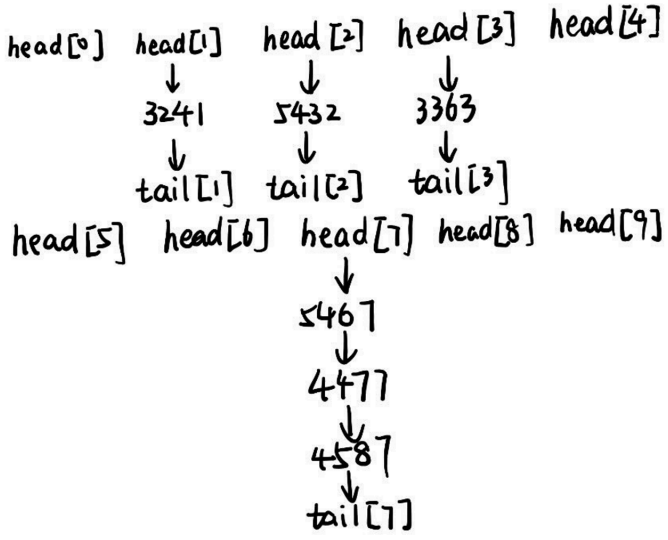
按照各个关键字位权重递增的次序（个、十、百），对d个关键字位做分配和收集。

分配：顺序扫描各个袁旭，若当前处理的关键字位=x，则将元素插入 Q_x队尾。

收集：把 $Q_{r-1}, Q_{r-2}, ..., Q_0$各个队列中的结点依次出队并链接

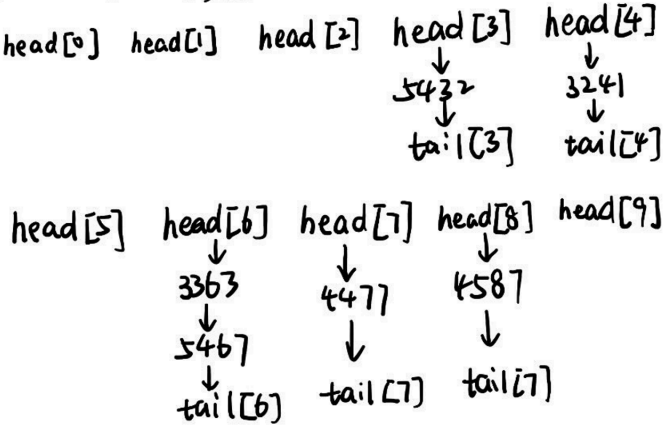
L→5467→3241→4477→5432→3363→4587

个位分配：



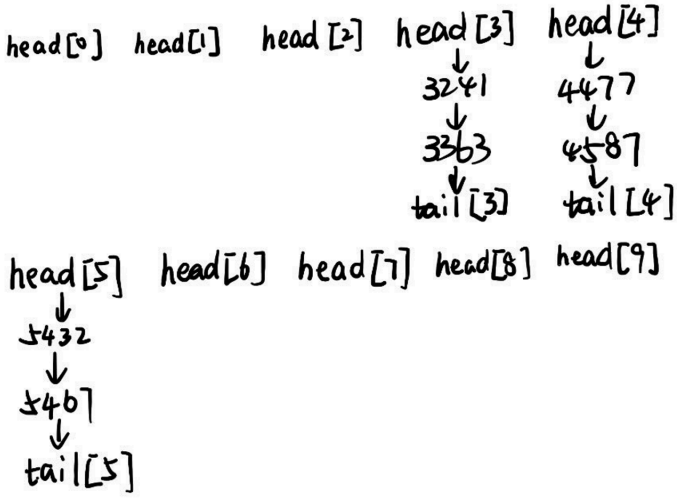
收集：L→3241→5432→3363→5467→4477→4587

十位分配：



收集：L→5432→3241→3363→5467→4477→4587

百位分配：



收集：L→3241→3363→4477→4587→5432→5467