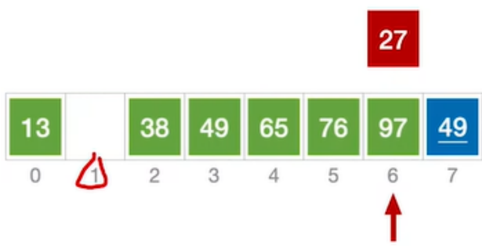
九.排序

- 排序：重新排列表中的元素，使表中元素满足按关键字有序的过程。
- 输入：n个记录$R_1, R_2, ..., R_n$，对应的关键字为 $k_1, k_2, ..., k_n$。
- 输出：输入序列的一个重排$R_1', R_2', ..., R_n'$，使得$k_1' \leq k_2' \leq ... \leq k_n'$(也可递减)。
- 排序算法的评价指标：时间复杂度、空间复杂度、稳定性。
- 算法的稳定性：若待排序表中有两个元素R_1和R_2，其对应的关键字相同即$key_i = key_j$，且在排序前R_1在R_2的前面，若使用某一排序算法排序后，R_1仍然在R_2的前面，则称这个排序算法是稳定的，否则称排序算法是不稳定的。
- 分类：
  内部排序：排序期间元素都在内存中——关注如何使时间、空间复杂度更低。
  外部排序：排序期间元素无法全部同时存在内存中，必须在排序的过程中根据要求不断地在内、外存之间移动——关注如何使时间、空间复杂度更低，如何使读/写磁盘次数更少

插入排序

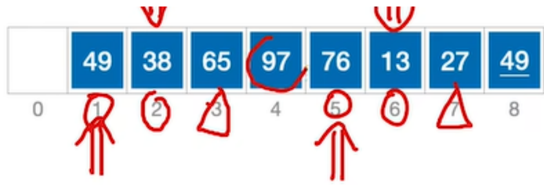
每次将一个待排序的记录按其关键字大小，插入到前面已经排好序的子序列中，直到全部记录插入完成。




```
// 对A[]数组中共n个元素进行插入排序
void InsertSort(int A[],int n){
    int i,j,temp;
    for(i=1; i<n; i++){ //将各元素插入已排好序的序列中
        if(A[i]<A[i-1]){        //如果A[i]关键字小于前驱
            temp=A[i];
            for(j=i-1; j>=0 && A[j]>temp; --j)
                A[j+1]=A[j];    //所有大于temp的元素都向后挪
            A[j+1]=temp;
        }
    }
}
算法效率分析：
时间复杂度（对比关键字、移动元素）：最好情况(原始表已经有序) O(n)，最差情况（原始表为逆序）O(n^{2})，平均情况 O(n^{2})。
空间复杂度：O(1)。
算法稳定性：稳定。
适用性：适用于顺序存储和链式存储的线性表。
```

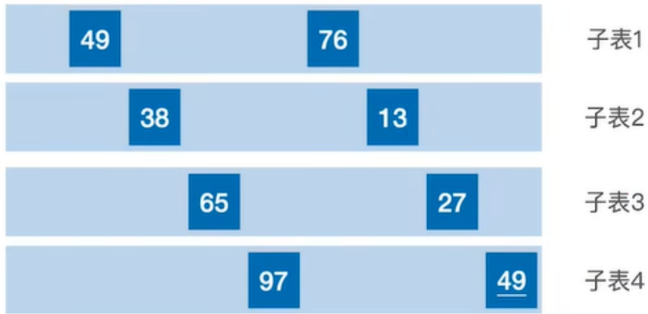
希尔排序

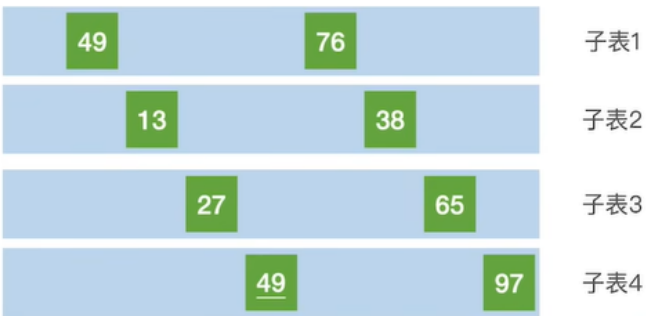
先追求表中元素的部分有序，再逐渐逼近全局有序，以减小插入排序算法的时间复杂度。
希尔排序:先将待排序表分割成若干形如 $L[i, i+d, i+ed, ..., i+kd]$的“特殊”子表，对各个子表分别进行直接插入排序。缩小增量d，重复上述过程，直到d=1为止。



第一趟：$d_1=n/2=4$

第一趟：$d_1=n/2=4$





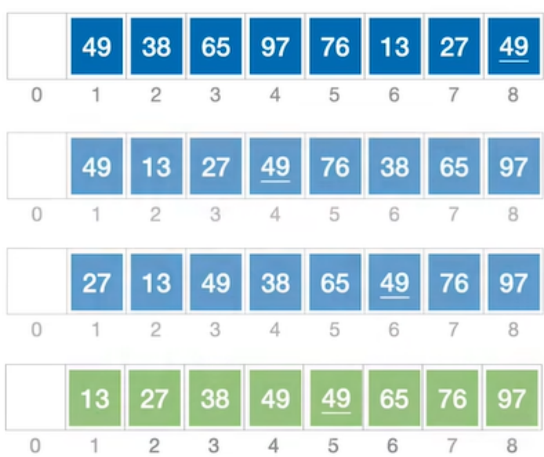
第二趟：$d_2=d_1/2=2$

希尔本人建议：每次将增量缩小一半

第一趟：$d_1=n/2=4$

第二趟：$d_2=d_1/2=2$

第三趟：$d_3=d_2/2=1$



```
// 对A[]数组共n个元素进行希尔排序
void ShellSort(ElemType A[], int n){
    int d,i,j;
    for(d=n/2; d>=1; d=d/2){    //步长d递减
        for(i=d+1; i<=n; ++i){
            if(A[i]<A[i-d]){
                A[0]=A[i];              //A[0]做暂存单元
                for(j=i-d; j>0 && A[0]<A[j]; j-=d)
                    A[j+d]=A[j];
                A[j+d]=A[0];
            }
                }
    }
}
```

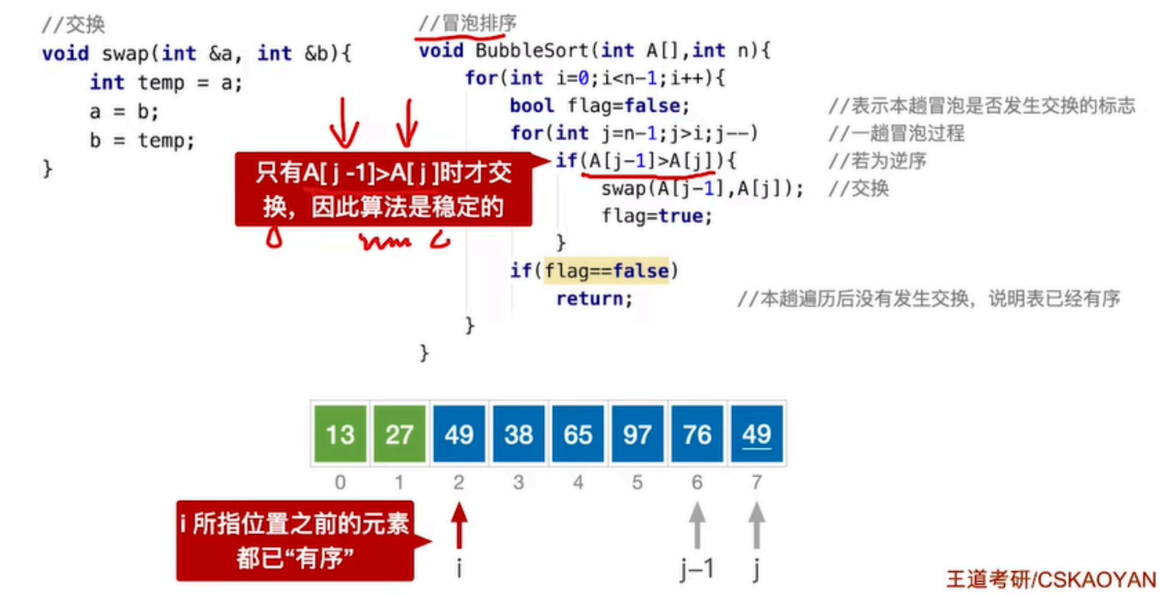
时间复杂度：希尔排序时间复杂度依赖于增量序列$d_1, d_2, \ldots$的选择有关。最差情况O(n^{2})，n在某个特顶范围时可达O($n^{1.3}$) 。
空间复杂度：O(1)
算法稳定性：不稳定
仅适用于顺序表

冒泡排序

从后往前（或从前往后）两两比较相邻元素的值，若为逆序（即 $A[i-1] > A[i]$），则交换它们，直到序列比较完。如此重复最多 n-1 次冒泡就能将所有元素排好序。为保证稳定性，关键字相同的元素不交换。

```
// 交换a和b的值
void swap(int &a, int &b){
    int temp=a;
    a=b;
    b=temp;
}

// 对A[]数组共n个元素进行冒泡排序
void BubbleSort(int A[], int n){
    for(int i=0; i<n-1; i++){
        bool flag = false;                      //标识本趟冒泡是否发生交换
        for(int j=n-1; j>i; j--){
            if(A[j-1]>A[j]){
                swap(A[j-1],A[j]);
                flag=true;
            }
        }
        if(flag==false)
            return;       //若本趟遍历没有发生交换，说明已经有序
    }
}
```

第n趟结束后，最小的n个元素会到最前边



时间复杂度：最好情况O(n)，最差情况$O(n^{2})$，平均情况$O(n^{2})$。
空间复杂度：O(1)。
稳定性：稳定。
适用性：冒泡排序可以用于顺序表、链表（从前往后“冒泡”，每一趟将更大的元素“冒”到链尾）。

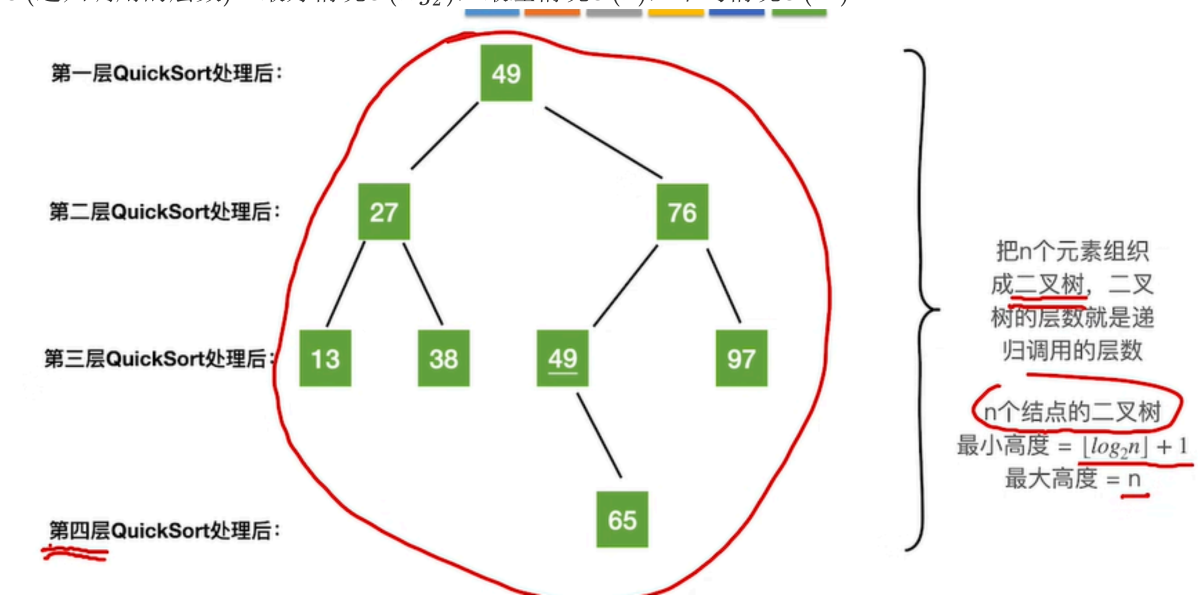
快速排序

- 在待排序表$L[1...n]$中任选一个元素$pivot$作为枢轴（通常取首元素），通过一趟排序将待排序表分为独立的两部分$L[1...k-1]$和$L[k+1...n]$。使得$L[1...k-1]$中的所有元素小于$pivot$，$L[k+1...n]$中的所有元素大于等于$pivot$，则$pivot$放在了其最终位置$L[k]$上。重复此过程直到每部分内只有一个元素或空为止。
- 快速排序是所有内部排序算法中性能最优的排序算法。
- 在快速排序算法中每一趟都会将枢轴元素放到其最终位置上。（可用来判断进行了几趟快速排序）
- 快速排序可以看作数组中n个元素组织成二叉树，每趟处理的枢轴是二叉树的根节点，递归调用的层数是二叉树的层数。

```
// 用第一个元素将数组A[]划分为两个部分
int Partition(int A[], int low, int high){
    int pivot = A[low]; //第一个元素作为枢轴
    while(low<high){    //用low、high搜索枢轴的最终位置
        while(low<high && A[high]>=pivot)
            --high;     //找到比枢轴小的元素，移到左端
        A[low] = A[high];
        while(low<high && A[low]<=pivot)
            ++low;      //找到比枢轴大的元素，移到右端
        A[high] = A[low];
    }
    A[low] = pivot;
    return low;
}

// 对A[]数组的low到high进行快速排序
void QuickSort(int A[], int low, int high){
    if(low<high){
        int pivotpos = Partition(A, low, high);  //划分
        QuickSort(A, low, pivotpos - 1);
        QuickSort(A, pivotpos + 1, high);
    }
}
```

算法效率分析：
时间复杂度：快速排序的时间复杂度$=O(n*$

递归调用的层数)。最好情况$O(nlog_2^n)$,最差情况$O(n^2)$，平均情况$O(n^2)$。空间复杂度：快速排序的空间复杂度 $=$ O(递归调用的层数)。最好情况$O(log_2^n)$，最差情况$O(n)$，平均情况$O(n^2)$。



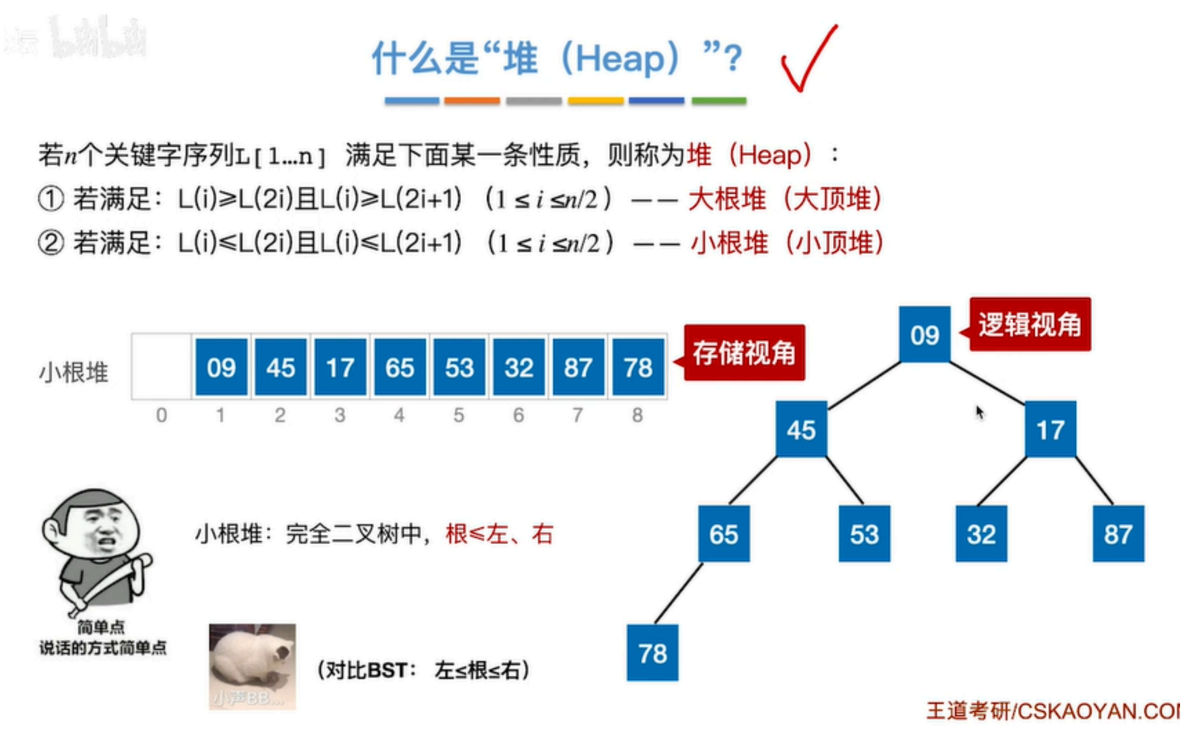
直接选择排序(选择排序)

选择排序： 每一趟在待排序元素中选取关键字最小（或最大）的元素加入有序子序列。
n个元素的简单选择排序需要n-1趟处理

```
// 交换a和b的值
void swap(int &a, int &b){
   int temp = a;
   a = b;
   b = temp;
}

// 对A[]数组共n个元素进行选择排序
void SelectSort(int A[], int n){
   for(int i=0; i<n-1; i++){            //一共进行n-1趟，i指向待排序序列中第一个元素
       int min = i;
       for(int j=i+1; j<n; j++){                //在A[i...n-1]中选择最小的元素
           if(A[j]<A[min])
               min = j;
       }
       if(min!=i)
           swap(A[i], A[min]);
   }
}
```

时间复杂度：无论待排序序列有序、逆序还是乱序，都需要进行$n-1$次处理，总共需要对比关键字$(n-1)+(n-2)+...+1=n(n-1)/2$次，因此时间复杂度始终是$O(n^2)$。
空间复杂度：O(1)。
稳定性：不稳定。
适用性：适用于顺序存储和链式存储的线性表。

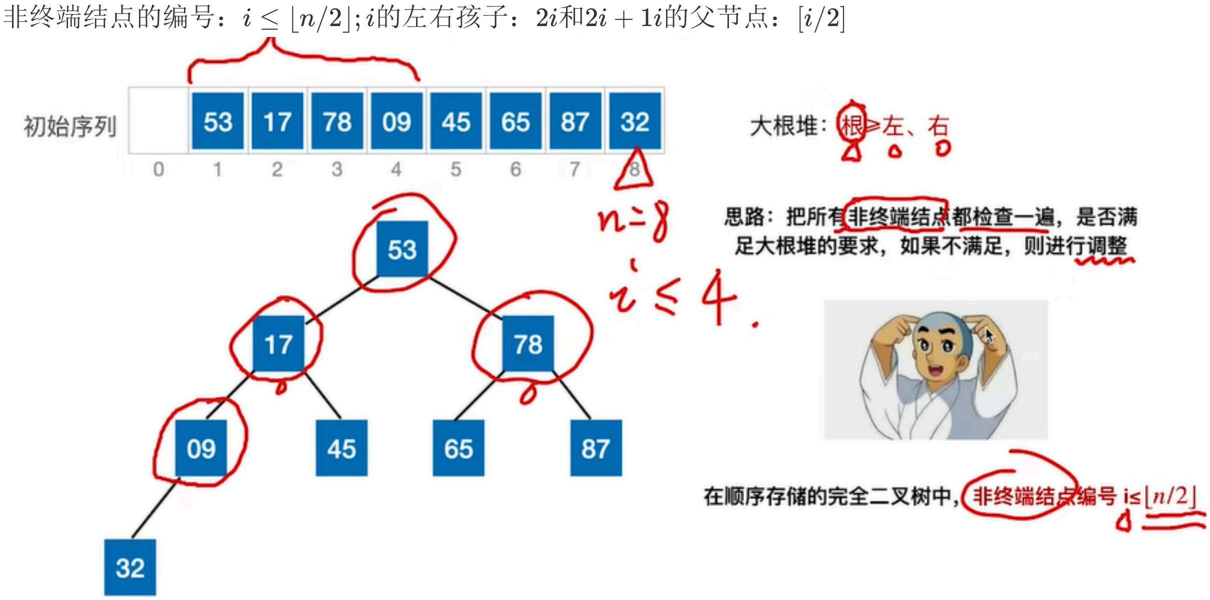
堆排序（选择排序）

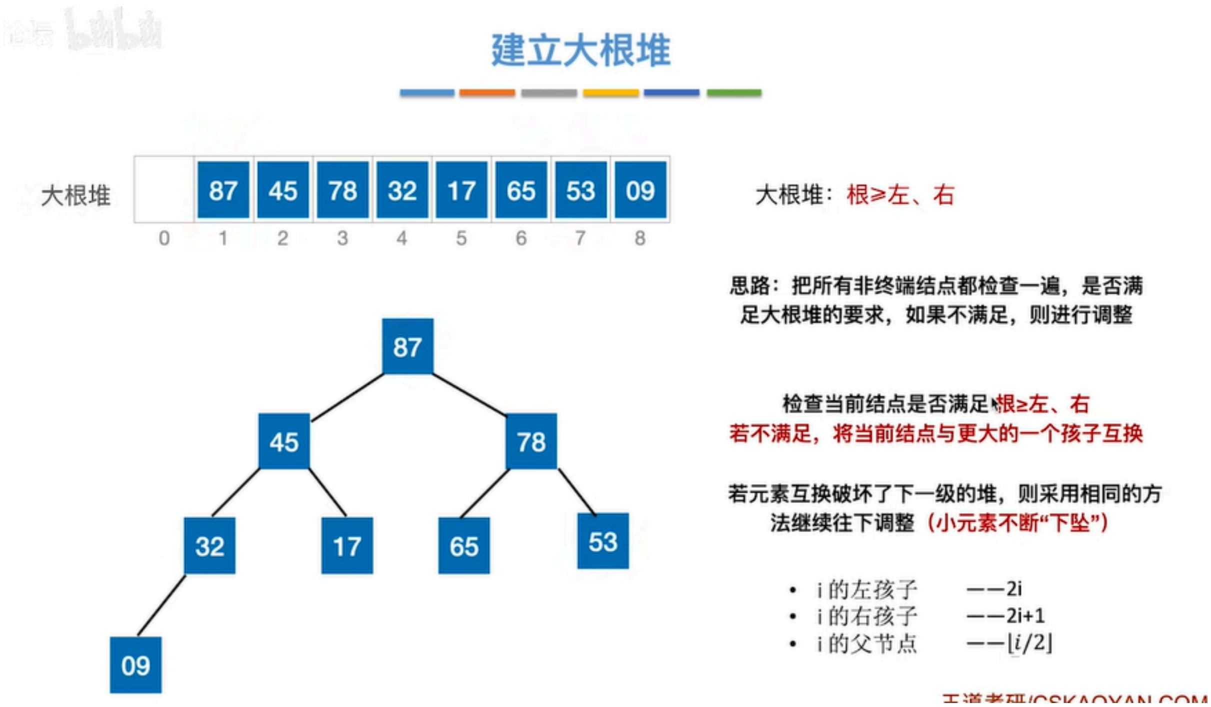


首先将存放在 L [1... n] 中的n个元素建成初始堆，由于堆本身的特点，堆顶元素就是最大值。将堆顶元素与堆底元素交换，这样待排序列的最大元素已经找到了排序后的位置。此时剩下的元素已不满足大根堆的性质，堆被破坏，将堆顶元素下坠使其继续保持大根堆的性质，如此重复直到堆中仅剩一个元素为止。（建堆、排序）

在顺序存储的完全二叉树中：

非终端结点的编号：$i \le \lfloor n/2 \rfloor$；$i$的左右孩子：$2i$和$2i+1$ i的父节点：$\lfloor i/2 \rfloor$





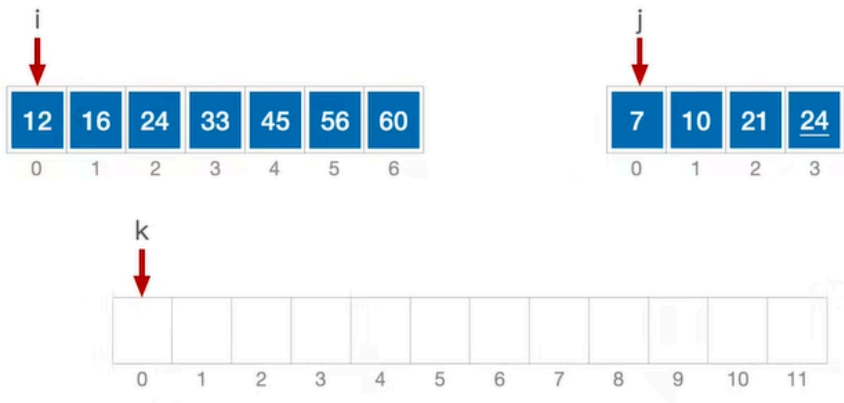

```
// 对初始序列建立大根堆
void BuildMaxHeap(int A[], int len){
    for(int i=len/2; i>0; i--)          //从后往前调整所有非终端结点
        HeadAdjust(A, i, len);
}

// 将以k为根的子树调整为大根堆
void HeadAdjust(int A[], int k, int len){
    A[0] = A[k];                    //A[0]暂存子树的根结点
    for(int i=2*k; i<=len; i*=2){       //i初始化当前结点的左孩子
        if(i<len && A[i]<A[i+1])        //沿k较大的子结点向下调整
            i++;                    //取key较大的子节点的下标
        if(A[0] >= A[i])
            break;                  //筛选结束
        else{
            A[k] = A[i];                        //将A[i]调整至双亲结点上
            k=i;                                    //修改k值，以便继续向下筛选
        }
    }
    A[k] = A[0]                     //被筛选结点的值放入最终位置
}

// 交换a和b的值
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

// 对长为len的数组A[]进行堆排序
void HeapSort(int A[], int len){
    BuildMaxHeap(A, len);               //初始建立大根堆
    for(int i=len; i>1; i--){           //n-1趟的交换和建堆过程
        swap(A[i], A[1]);
        HeadAdjust(A,1,i-1);
    }
}
```

算法效率分析：时间复杂度：$O(n\log_2 n)$。建堆时间$O(n)$，之后进行$n-1$次向下调整操作，每次调整时间复杂度为$O(\log_2 n)$。空间复杂度：$O(1)$。稳定性：不稳定。

归并排序

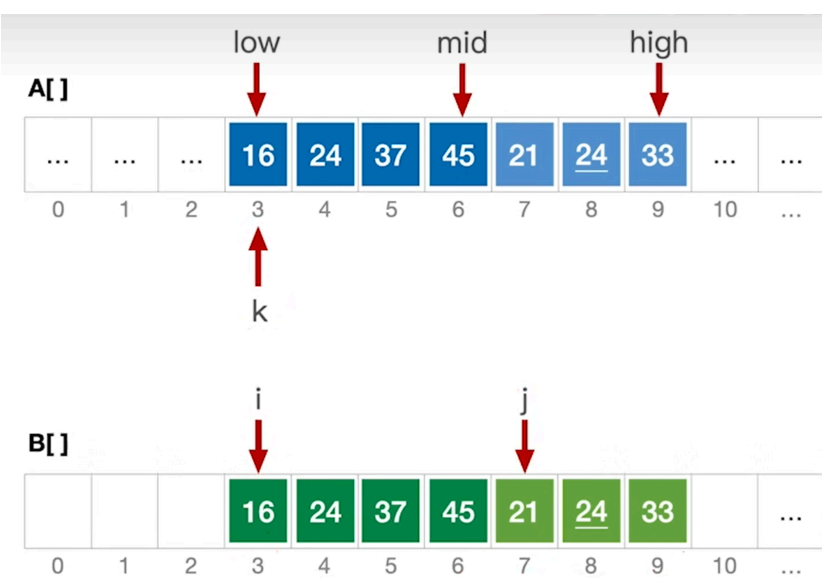
归并（Merge）：把两个或多个已经有序的序列合并成一个新的有序表。k路归并每选出一个元素，需对比关键字k-1次。

二路归并（二合一）



对比 i、j 所指元素，选择更小的一个放入 k 所指位置

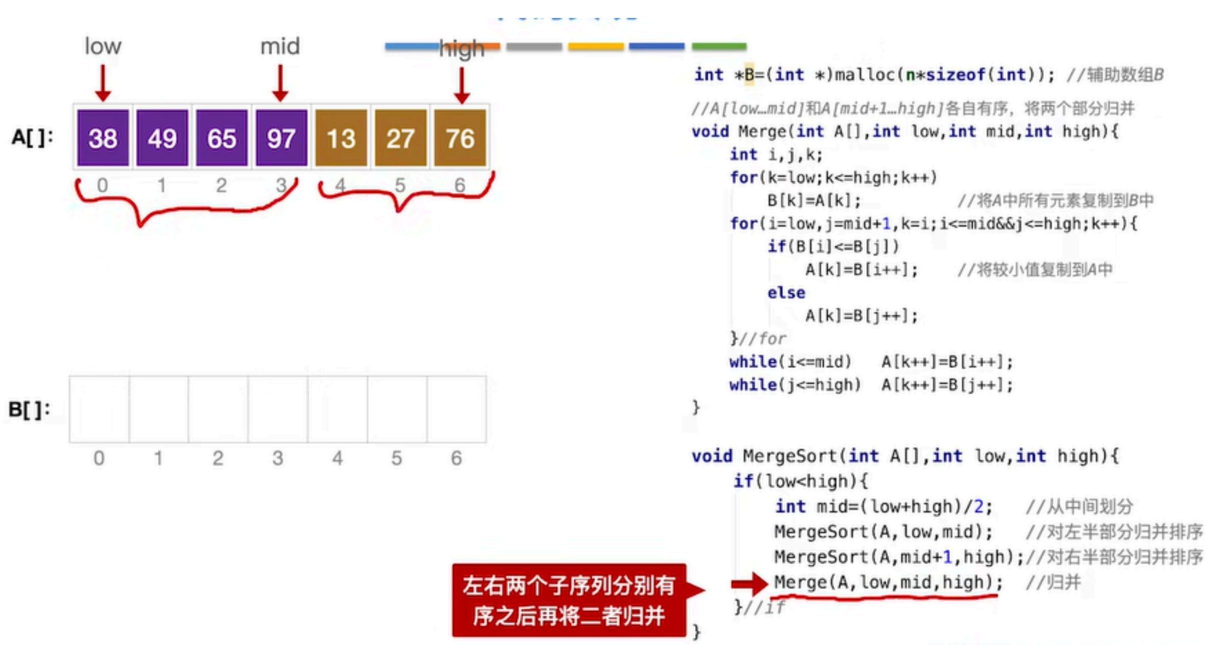
对比 i、j 所指元素，选择更小的一个放入 k 所指位置

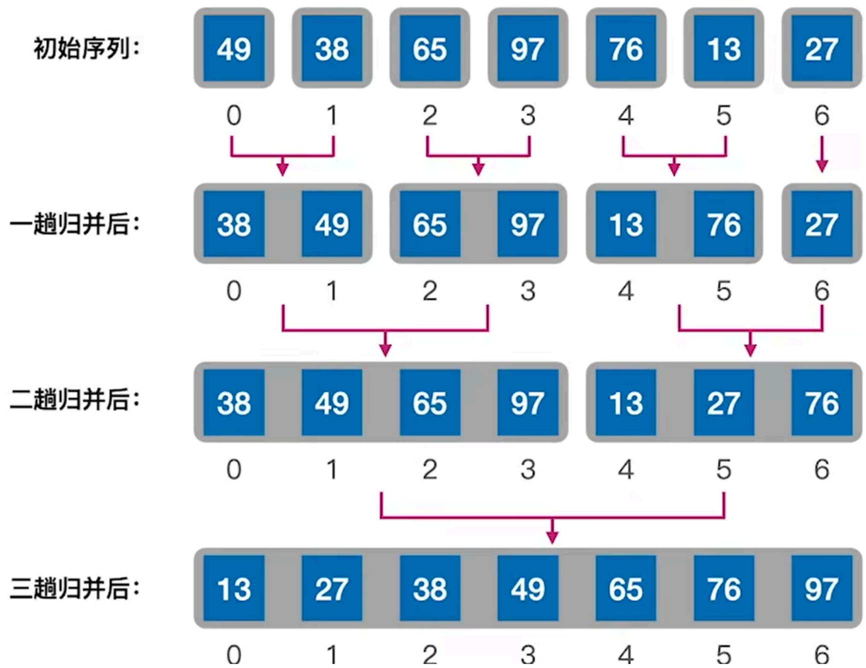



```
int *B=(int *)malloc(n*sizeof(int)); //辅助数组B

//A[low…mid]和A[mid+1…high]各自有序，将两个部分归并
void Merge(int A[],int low,int mid,int high){
    int i,j,k;
    for(k=low;k<=high;k++)
        B[k]=A[k];            //将A中所有元素复制到B中
    for(i=low,j=mid+1,k=i;i<=mid&&j<=high;k++){
        if(B[i]<=B[j])
            A[k]=B[i++];      //将较小值复制到A中
        else
            A[k]=B[j++];
    }//for
    while(i<=mid)    A[k++]=B[i++];
    while(j<=high)   A[k++]=B[j++];
}
```

归并

```
// 辅助数组B
int *B= new int[n];

// A[low,...,mid]，A[mid+1,...,high]各自有序，将这两个部分归并
void Merge(int A[], int low, int mid, int high){
    int i,j,k;
    for(k=low; k<=high; k++)
        B[k]=A[k];
    for(i=low, j=mid+1, k=i; i<=mid && j<= high; k++){
        if(B[i]<=B[j])
            A[k]=B[i++];
        else
            A[k]=B[j++];
    }
    //当某一序列已经遍历完了，将没有归并完的部分复制到尾部
    while(i<=mid)
        A[k++]=B[i++];
    while(j<=high)
        A[k++]=B[j++];
}

// 归并排序
void MergeSort(int A[], int low, int high){
    if(low<high){
        int mid = (low+high)/2;
        MergeSort(A, low, mid);
        MergeSort(A, mid+1, high);
        Merge(A,low,mid,high);     //归并
    }
}
```





2路归并的"归并树"——形态上就是一棵倒立的二叉树

二叉树的第h层最多有 2^{h-1}个结点

若树高为h，则应满足 $n \leq 2^{h-1}$

即 $h-1=\lceil log_2 n \rceil$

结论：n个元素进行2路归并排序，归并趟数=$\lceil log_2 n \rceil$

每趟归并时间复杂度为 $O(n)$，则算法时间复杂度为 $O(nlog_2 n)$

空间复杂度=$O(n)$，来自于辅助数组B

算法是稳定的