不同条件下，排序方法的选择

(1)若n较小(如n≤50)，可采用直接插入或直接选择排序。

当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插人，应选直接选择排序为宜。

(2)若文件初始状态基本有序(指正序)，则应选用直接插人、冒泡或随机的快速排序为宜；

(3)若n较大，则应采用时间复杂度为O(nlgn)的排序方法：快速排序、堆排序或归并排序。

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；

堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。这两种排序都是不稳定的。

若要求排序稳定，则可选用归并排序。但本章介绍的从单个记录起进行两两归并的 排序算法并不值得提倡，通常可以将它和直接插入排序结合在一起使用。先利用直接插入排序求得较长的有序子文件，然后再两两归并之。因为直接插入排序是稳定 的，所以改进后的归并排序仍是稳定的。

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

冒泡排序

1.比较相邻的元素。如果第一个比第二个大，就交换他们两个。

2.对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。

3.针对所有的元素重复以上的步骤，除了最后一个。

4.持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

```
def bubbleSort(arr):
    for i in range(1, len(arr)):
        for j in range(0, len(arr)-i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

选择排序

1.首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置

2.再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。

3.重复第二步，直到所有元素均排序完毕。

```python
def selectionSort(arr):
    for i in range(len(arr) - 1):
        # 记录最小数的索引
        minIndex = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[minIndex]:
                minIndex = j
        # i 不是最小数时，将 i 和最小数进行交换
        if i != minIndex:
            arr[i], arr[minIndex] = arr[minIndex], arr[i]
    return arr
```

插入排序

1.将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。

2.从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置。（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）

```python
def select(arr):
    for i in range(1,len(arr)):
        item=arr[i]
        j=i-1
        while j>=0:
            if item<arr[j]:
                arr[j+1]=arr[j]
                j-=1
            else:
                break
        arr[j+1]=item

```

快速排序：

1 从数列中挑出一个元素，称为 “基准”（pivot）；

2 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；

3 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序；

```python
def quicksort(arr):
    if not arr:
        return []
    l=quicksort([arr[i] for i in range(1,len(arr)) if arr[i]<arr[0]])
    r=quicksort([arr[i] for i in range(1,len(arr)) if arr[i]>arr[0]])
    return l+[arr[0]]+r
```

```python

def quicksort(arr,l,r):
    if l<r:
        pi=partition(arr,l,r)
        quicksort(arr,l,pi)
        quicksort(arr,pi+1,r)

```

```
def partition(arr,l,r):
    index=l-1
     pivot=arr[r]
     for i in range(l,r):
         if arr[i]<pivot:
             i+=1
             arr[j],arr[i]=arr[j],arr[j]
     arr[i+1],pivot=pivot,arr[i+1]
     return i+1
```

堆排序：

适用场景： 堆排序可以在N个元素中找到top K，时间复杂度是O(N log K)，空间复杂的是O(K)；另外一个适合用heap的场合是优先队列，需要在一组不停更新的数据中不停地找最大/小元素

```

def heapfy(arr,n,i):
    large=i
    l=2*i+1
    r=2*i+2
    if l<n and arr[l]>arr[large]:
        large=l
    if r<n and arr[r]>arr[large]:
        large=r
     if large!=i:
         arr[large],arr[i]=arr[i],arr[large]
         heapfy(arr,n,large)

def heapsort(arr):
     n=len(arr)
     for i in range((n-2)//2,-1,-1):
         heapfy(arr,n,i)
     for i in range(n-1,-1,-1):
         arr[i],arr[0]=arr[0],arr[i]
         heapfy(arr,i,0)
```

希尔排序：

基本思想是：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。

1 选择一个增量序列 t1，t2，……，tk，其中 ti > tj, tk = 1；

2 按增量序列个数 k，对序列进行 k 趟排序；

3 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。

```
def shell_sort(alist):
    n = len(alist)
    gap = n // 2
    while gap > 0:
        for i in range(gap,n):
            j = i
            while j>=gap and alist[j-gap] > alist[j]:
```

```
                alist[j-gap], alist[j] = alist[j], alist[j-gap]
                j -= gap
        gap = gap //2
```

归并排序

```
def mergesort(arr):
    if len(arr)<=1:
        return arr
    mid=len(arr)//2
    left=mergesort(arr[:mid])
    right=mergesort(arr[mid:])
    return merge(left,right)

def merge(left,right):
    c=[]
    l=r=0
    while l<len(left) and r<len(right):
        if left[l]<right[r]:
            c.append(left[l])
            l+=1
        else:
            c.append(right[r])
            r+=1
    if l==len(left):
        c+=right[r:]
    if r==len(right):
        c+=left[l:]
    return c
```

计数排序

```
def countingSort(arr, maxValue):
    bucketLen = maxValue+1
    bucket = [0]*bucketLen
    sortedIndex =0
    arrLen = len(arr)
    for i in range(arrLen):
        if not bucket[arr[i]]:
            bucket[arr[i]]=0
        bucket[arr[i]]+=1
    for j in range(bucketLen):
        while bucket[j]>0:
            arr[sortedIndex] = j
            sortedIndex+=1
            bucket[j]-=1
    return arr
```