数据结构与算法（Python版）

顺序查找算法及分析

陈斌 北京大学 gischen@pku.edu.cn

顺序查找Sequential Search

- 如果数据项保存在如列表这样的集合中，我们会称这些数据项具有线性或者顺序关系。
- 在Python List中，这些数据项的存储位置称为下标（index），这些下标都是有序的整数。
- 通过下标，我们就可以按照顺序来访问和查找数据项，这种技术称为“顺序查找”

顺序查找Sequential Search

❖ **要确定列表中是否存在需要查找的数据项**

首先从列表的第1个数据项开始，

按照下标增长的顺序，逐个比对数据项，

如果到最后一个都未发现要查找的项，那么查找失败。

顺序查找：无序表查找代码


```
def sequentialSearch(alist, item):
    pos = 0
    found = False

    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos = pos+1

    return found

testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequentialSearch(testlist, 3))
print(sequentialSearch(testlist, 13))
```

下标顺序增长

顺序查找：算法分析

- 要对查找算法进行分析，首先要确定其中的**基本计算步骤**。
- 回顾第二章算法分析的要点，这种基本计算步骤必须要**足够简单**，并且在算法中**反复执行**
- 在查找算法中，这种基本计算步骤就是进行数据项的**比对**

  当前数据项等于还是不等于要查找的数据项，比对的次数决定了算法复杂度

顺序查找：算法分析

- 在顺序查找算法中，为了保证是讨论的一般情形，需要假定列表中的数据项并没有按值排列顺序，而是**随机放置**在列表中的各个位置

换句话说，数据项在列表中各处出现的概率是相同的

顺序查找：算法分析

- 数据项是否在列表中，比对次数是不一样的
- 如果数据项不在列表中，需要比对所有数据项才能得知，比对次数是n
- 如果数据项在列表中，要比对的次数，其情况就较为复杂

  最好的情况，第1次比对就找到

  最坏的情况，要n次比对

顺序查找：算法分析

- 数据项在列表中，比对的一般情形如何？

  因为数据项在列表中各个位置出现的概率是相同的；所以平均状况下，比对的次数是n/2；

- 所以，顺序查找的算法复杂度是O(n)

Case	Best Case	Worst Case	Average Case
item is present	1	n	n/2
item is not present	n	n	n

- 这里我们假定列表中的数据项是无序的，那么如果数据项排了序，顺序查找算法的效率又如何呢？

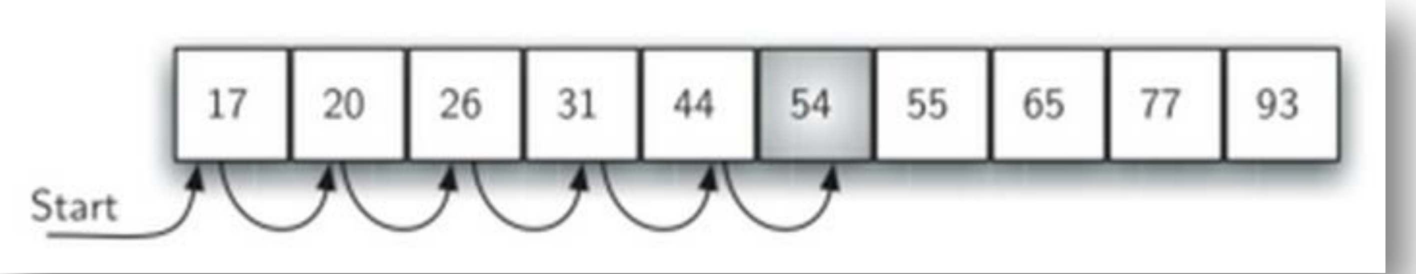
顺序查找：算法分析

- ❖ **实际上，我们在第三章的有序表Search方法实现中介绍过顺序查找**

  当数据项存在时，比对过程与无序表完全相同

  不同之处在于，如果数据项不存在，比对可以提前结束

  - 如下图中查找数据项50，当看到54时，可知道后面不可能存在50，可以提前退出查找



顺序查找：有序表查找代码

```
def orderedSequentialSearch(alist, item):
    pos = 0
    found = False
    stop = False
    while pos < len(alist) and not found and not stop:
        if alist[pos] == item:
            found = True
        else:
            if alist[pos] > item:
                stop = True
            else:
                pos = pos+1

    return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(orderedSequentialSearch(testlist, 3))
print(orderedSequentialSearch(testlist, 13))
```

提前退出

数据结构与算法（Python版）

二分查找算法及分析

陈斌 北京大学 gischen@pku.edu.cn

二分查找

- ❖ 那么对于有序表，有没有更好更快的查找算法?
- ❖ 在顺序查找中，如果第1个数据项不匹配查找项的话，那最多还有n-1个待比对的数据项
- ❖ 那么，有没有方法能利用**有序表**的特性，迅速缩小待比对数据项的范围呢?

#

二分查找

- **我们从列表中间开始比对!**

  如果列表中间的项匹配查找项，则查找结束

  如果不匹配，那么就有两种情况：

  - 列表中间项比查找项大，那么查找项只可能出现在前半部分
  - 列表中间项比查找项小，那么查找项只可能出现在后半部分

  无论如何，我们都会将比对范围缩小到原来的一半：n/2

- **继续采用上面的方法查找**

  每次都会将比对范围缩小一半

二分查找：代码

中间项比对

缩小比对范围

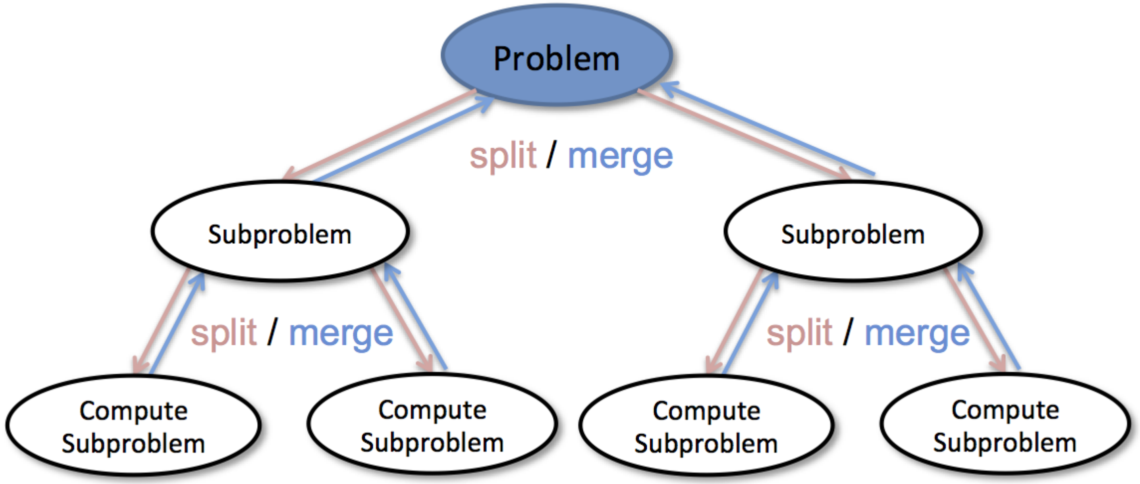
```python
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

二分查找：分而治之

❖ 二分查找算法实际上体现了解决问题的典型策略：**分而治之**

将问题分为若干更小规模的部分

通过解决每一个小规模部分问题，并将结果汇总得到原问题的解



二分查找：分而治之

❖ 显然，递归算法就是一种典型的分治策略算法，二分法也适合用递归算法来实现

```python
def binarySearch(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)//2
        if alist[midpoint]==item:
          return True
        else:
          if item<alist[midpoint]:
            return binarySearch(alist[:midpoint],item)
          else:
            return binarySearch(alist[midpoint+1:],item)
```

结束条件

调用自身

缩小规模

二分查找：算法分析

- 由于二分查找，每次比对都将下一步的比对范围**缩小一半**
- 每次比对后剩余数据项如下表所示：

Comparisons	Approximate Number of Items Left
1	n/2
2	n/4
3	n/8
…	
i	$n/2^i$

二分查找：算法分析

- 当比对次数足够多以后，比对范围内就会仅剩余1个数据项
- 无论这个数据项是否匹配查找项，比对最终都会结束，解下列方程：

得到：$i=\log_2(n)$

$$\frac{n}{2^i} = 1$$

- 所以二分法查找的算法复杂度是O(log n)

二分查找：进一步的考虑

- 虽然我们根据比对的次数，得出二分查找的复杂度O(log n)
- 但本算法中除了比对，还有一个因素需要注意到：

```
binarySearch(alist[:midpoint],item)
```

这个递归调用使用了列表切片，而切片操作的复杂度是O(k)，这样会使整个算法的时间复杂度稍有增加；

当然，我们采用切片是为了程序可读性更好，实际上也可以不切片，而只是传入起始和结束的索引值即可，这样就不会有切片的时间开销了。

二分查找：进一步的考虑

- 另外，虽然二分查找在时间复杂度上优于顺序查找
- 但也要考虑到**对数据项进行排序**的开销

  如果一次排序后可以进行多次查找，那么排序的开销就可以摊薄

  但如果数据集经常变动，查找次数相对较少，那么可能还是直接用无序表加上顺序查找来得经济

- 所以，在算法选择的问题上，光看时间复杂度的优劣是不够的，还需要考虑到**实际应用**的情况。

数据结构与算法（Python版）

冒泡排序和选择排序算法及分析

陈斌 北京大学 gischen@pku.edu.cn

排序：冒泡排序Bubble Sort

- 冒泡排序的算法思路在于对无序表进行多趟比较交换，
- 每趟包括了多次两两相邻比较，并将逆序的数据项互换位置，最终能将本趟的最大项就位
- 经过n-1趟比较交换，实现整表排序
- 每趟的过程类似于“气泡”在水中不断上浮到水面的经过

排序：冒泡排序Bubble Sort

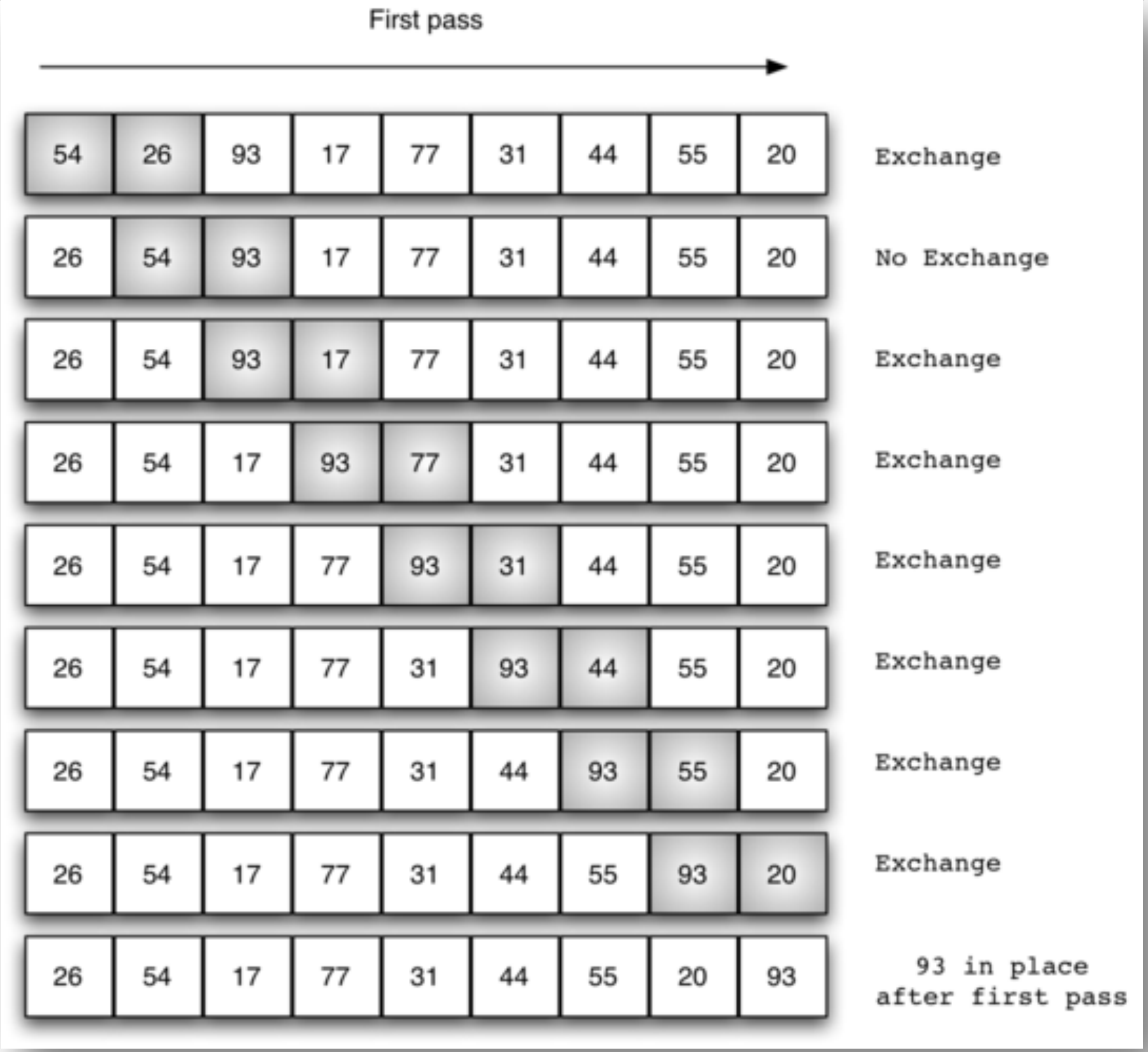
- 第1趟比较交换，共有n-1对相邻数据进行比较

  一旦经过最大项，则最大项会一路交换到达最后一项

- 第2趟比较交换时，最大项已经就位，需要排序的数据减少为n-1，共有n-2对相邻数据进行比较

- 直到第n-1趟完成后，最小项一定在列表首位，就无需再处理了。

冒泡排序：第1趟


First pass
54 26 93 17 77 31 44 55 20
Exchange
26 54 93 17 77 31 44 55 20
No Exchange
26 54 93 17 77 31 44 55 20
Exchange
26 54 17 93 77 31 44 55 20
Exchange
26 54 17 77 93 31 44 55 20
Exchange
26 54 17 77 31 93 44 55 20
Exchange
26 54 17 77 31 44 93 55 20
Exchange
26 54 17 77 31 44 55 93 20
Exchange
26 54 17 77 31 44 55 20 93
93 in place after first pass

冒泡排序：代码

n-1趟

序错，交换

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

Python支持直接交换

`alist[i],alist[i+1]=alist[i+1],alist[i]`

https://zh.visualgo.net/sorting

冒泡排序：算法分析

- 无序表初始数据项的排列状况对冒泡排序没有影响
- 算法过程总需要n-1趟，随着趟数的增加，**比对**次数逐步从n-1减少到1，并包括可能发生的数据项**交换**。
- 比对次数是1~n-1的累加：

$$\frac{1}{2}n^2 - \frac{1}{2}n$$

- 比对的时间复杂度是$O(n^2)$

冒泡排序：算法分析

- 关于交换次数，时间复杂度也是$O(n^2)$，通常每次交换包括3次赋值
- **最好**的情况是列表在排序前已经有序，交换次数为0
- **最差**的情况是每次比对都要进行交换，交换次数等于比对次数
- **平均**情况则是最差情况的一半

冒泡排序：算法分析

- 冒泡排序通常作为时间效率较差的排序算法，来作为其它算法的对比基准。
- 其效率主要差在每个数据项在找到其最终位置之前，
- 必须要经过**多次**比对和交换，其中大部分的操作是**无效**的。
- 但有一点优势，就是**无需**任何额外的存储空间开销。

冒泡排序：性能改进

- 另外，通过监测每趟比对是否发生过交换，可以提前确定排序是否完成
- 这也是其它多数排序算法无法做到的
- 如果某趟比对没有发生任何交换，说明列表已经排好序，可以提前结束算法

冒泡排序：性能改进

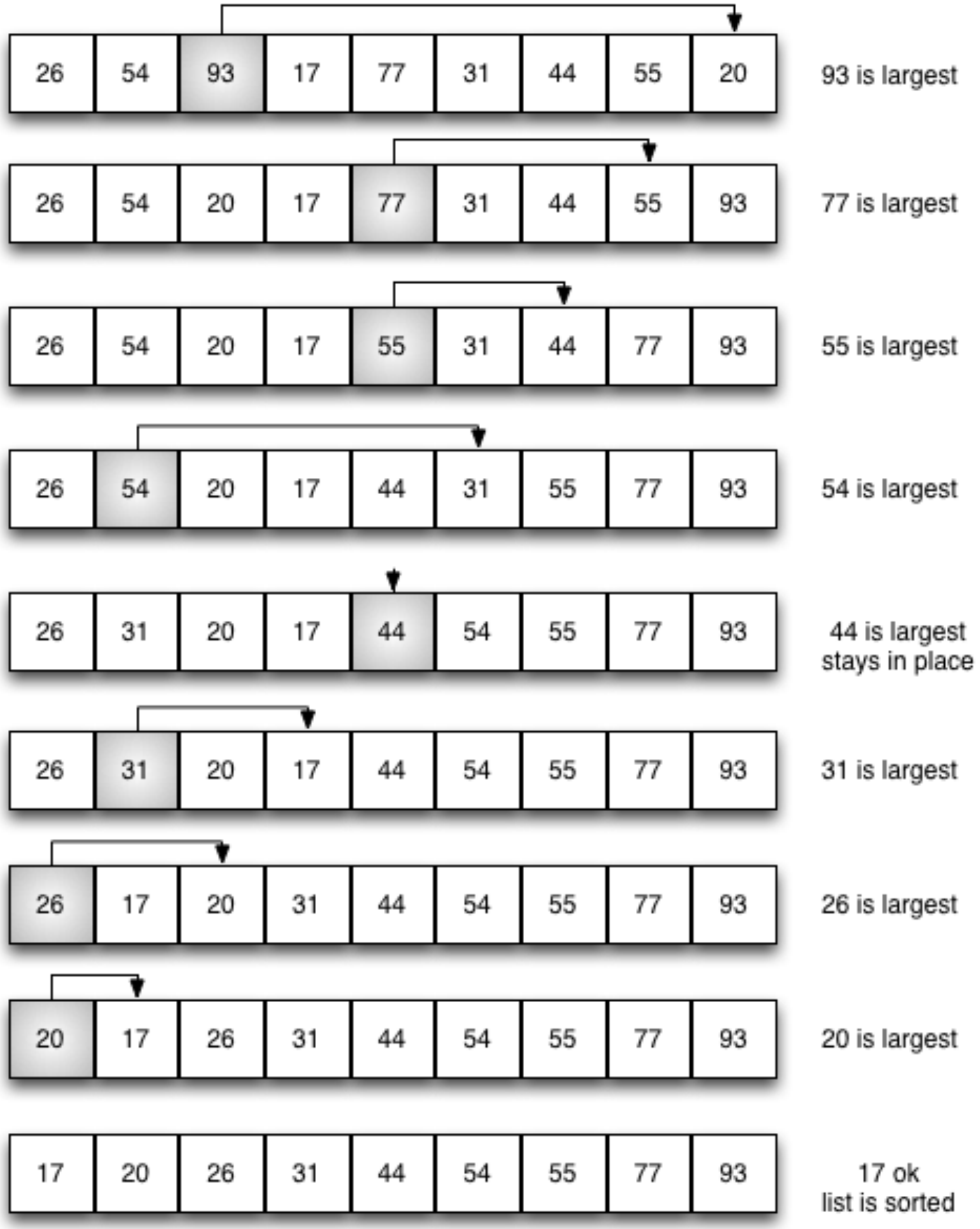

```
def shortBubbleSort(alist):
    exchanges = True
    passnum = len(alist)-1
    while passnum > 0 and exchanges:
       exchanges = False
       for i in range(passnum):
           if alist[i]>alist[i+1]:
               exchanges = True
               temp = alist[i]
               alist[i] = alist[i+1]
               alist[i+1] = temp
       passnum = passnum-1

alist=[20,30,40,90,50,60,70,80,100,110]
shortBubbleSort(alist)
print(alist)
```

选择排序Selection Sort

- 选择排序对冒泡排序进行了改进，保留了其基本的多趟比对思路，每趟都使当前最大项就位。
- 但选择排序对交换进行了削减，相比起冒泡排序进行多次交换，每趟仅进行1次交换，记录最大项的所在位置，最后再跟本趟最后一项交换
- 选择排序的时间复杂度比冒泡排序稍优

  比对次数不变，还是$O(n^2)$

  交换次数则减少为$O(n)$


26 54 93 17 77 31 44 55 20
93 is largest
26 54 20 17 77 31 44 55 93
77 is largest
26 54 20 17 55 31 44 77 93
55 is largest
26 54 20 17 44 31 55 77 93
54 is largest
26 31 20 17 44 54 55 77 93
44 is largest
stays in place
26 31 20 17 44 54 55 77 93
31 is largest
26 17 20 31 44 54 55 77 93
26 is largest
20 17 26 31 44 54 55 77 93
20 is largest
17 20 26 31 44 54 55 77 93
17 ok
list is sorted

选择排序：代码


```python
def selectionSort(alist):
    for fillslot in range(len(alist)-1,0,-1):
        positionOfMax=0
        for location in range(1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax = location

        temp = alist[fillslot]
        alist[fillslot] = alist[positionOfMax]
        alist[positionOfMax] = temp
```

数据结构与算法（Python版）

插入排序算法及分析

陈斌 北京大学 gischen@pku.edu.cn

插入排序Insertion Sort

❖ 插入排序时间复杂度仍然是$O(n^2)$，但算法思路与冒泡排序、选择排序不同

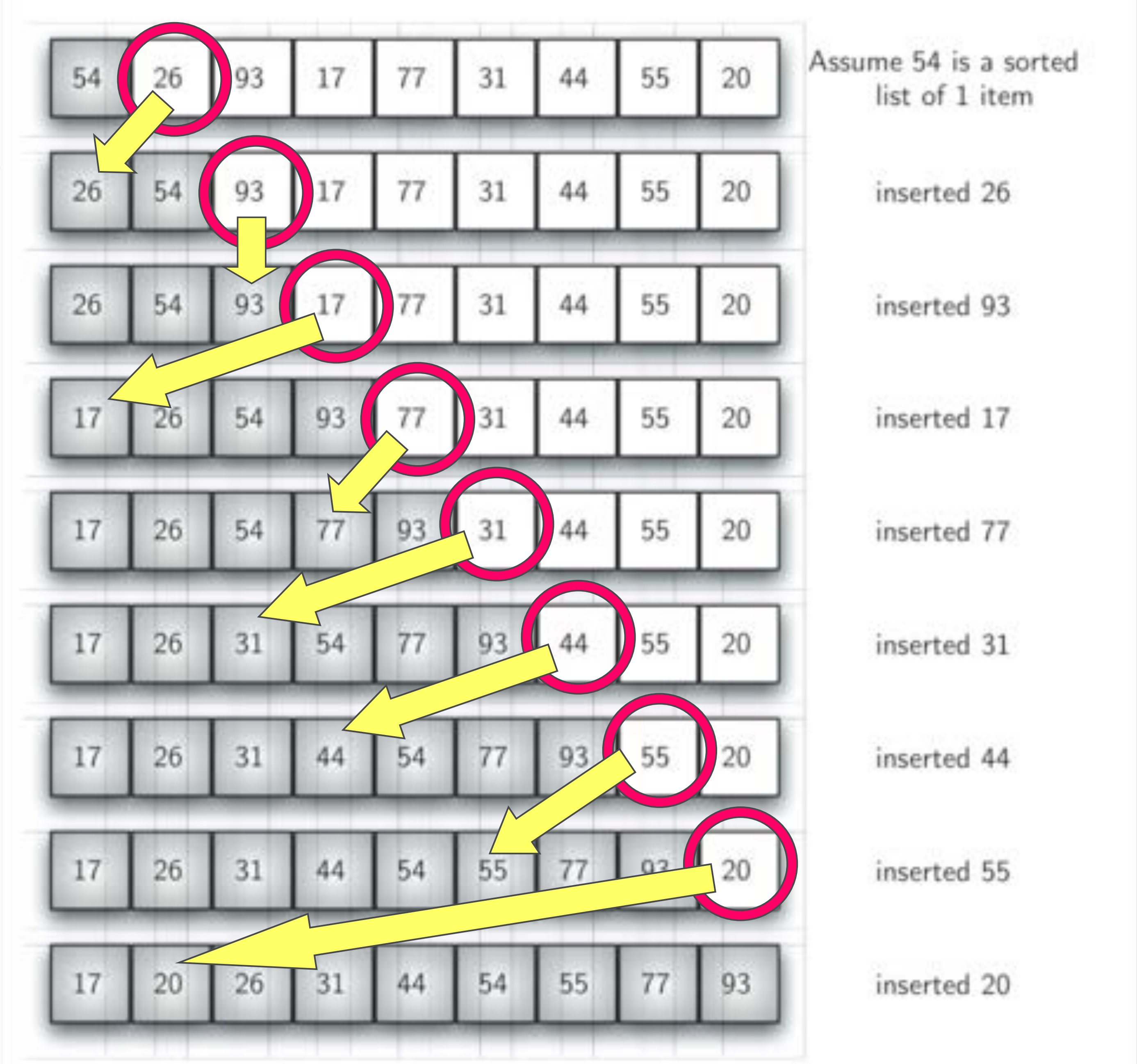
❖ 插入排序维持一个已排好序的子列表，其位置始终在列表的前部，然后逐步扩大这个子列表直到全表

插入排序Insertion Sort

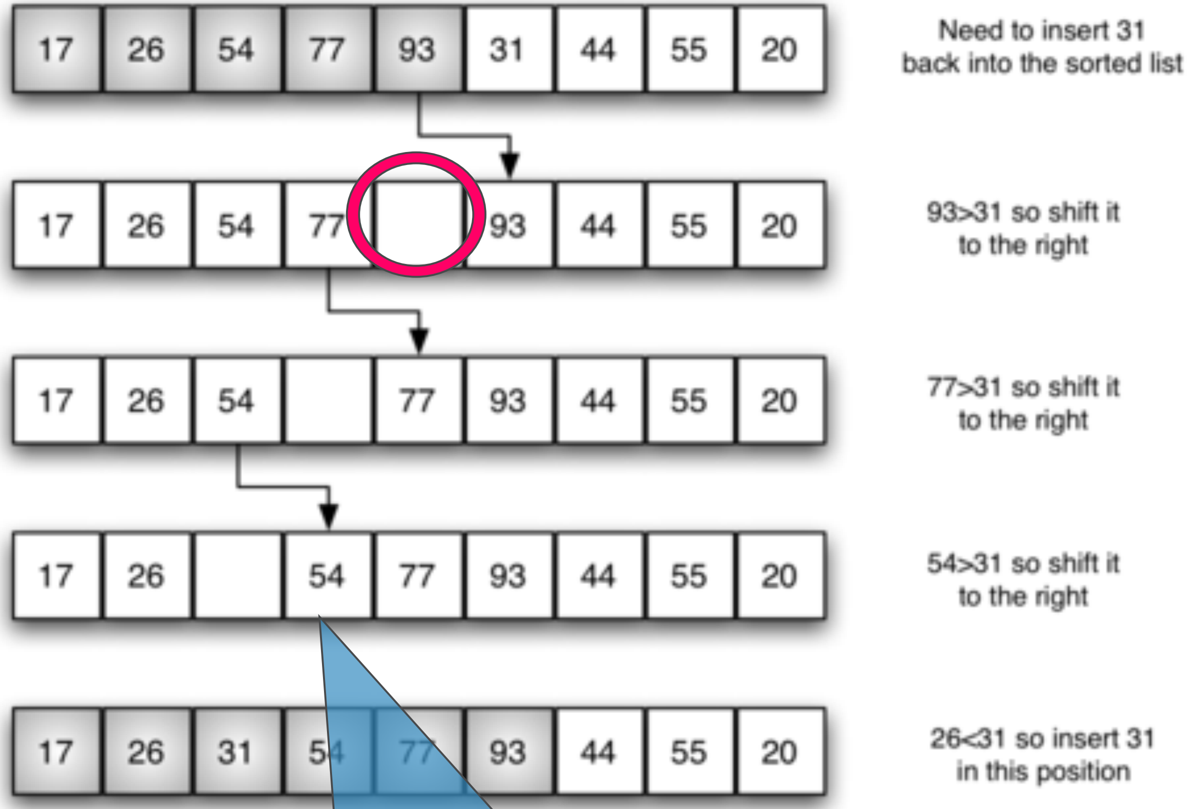
- 第1趟，子列表仅包含第1个数据项，将第2个数据项作为“新项”插入到子列表的合适位置中，这样已排序的子列表就包含了2个数据项
- 第2趟，再继续将第3个数据项跟前2个数据项比对，并移动比自身大的数据项，空出位置来，以便加入到子列表中
- 经过n-1趟比对和插入，子列表扩展到全表，排序完成

插入排序Insertion Sort

- 插入排序的比对主要用来寻找“**新项**”的**插入位置**
- 最差情况是每趟都与子列表中所有项进行比对，总比对次数与冒泡排序相同，数量级仍是$O(n^2)$
- 最好情况，列表已经排好序的时候，每趟仅需1次比对，总次数是$O(n)$


54 26 93 17 77 31 44 55 20
Assume 54 is a sorted list of 1 item
26 54 93 17 77 31 44 55 20
inserted 26
26 54 93 17 77 31 44 55 20
inserted 93
17 26 54 93 77 31 44 55 20
inserted 17
17 26 54 77 93 31 44 55 20
inserted 77
17 26 31 54 77 93 44 55 20
inserted 31
17 26 31 44 54 77 93 55 20
inserted 44
17 26 31 44 54 55 77 93 20
inserted 55
17 20 26 31 44 54 55 77 93
inserted 20

插入排序：思路



比对，移动所有
比“新项”大的数据项

插入排序：代码


```
def insertionSort(alist):
    for index in range(1,len(alist)):

        currentvalue = alist[index]
        position = index

        while position>0 and alist[position-1]>currentvalue:
            alist[position]=alist[position-1]
            position = position-1

        alist[position]=currentvalue
```

新项/插入项

比对、移动

插入新项

由于移动操作仅包含1次赋值，是交换操作的1/3，所以插入排序性能会较好一些。

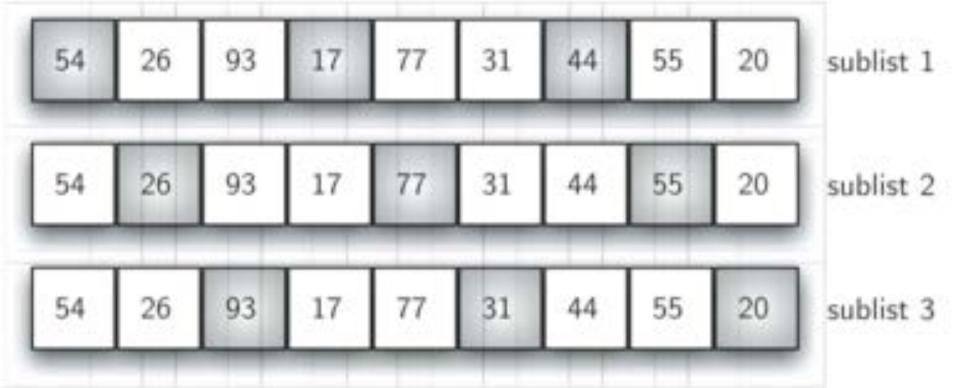
数据结构与算法（Python版）

谢尔排序算法及分析

陈斌 北京大学 gischen@pku.edu.cn

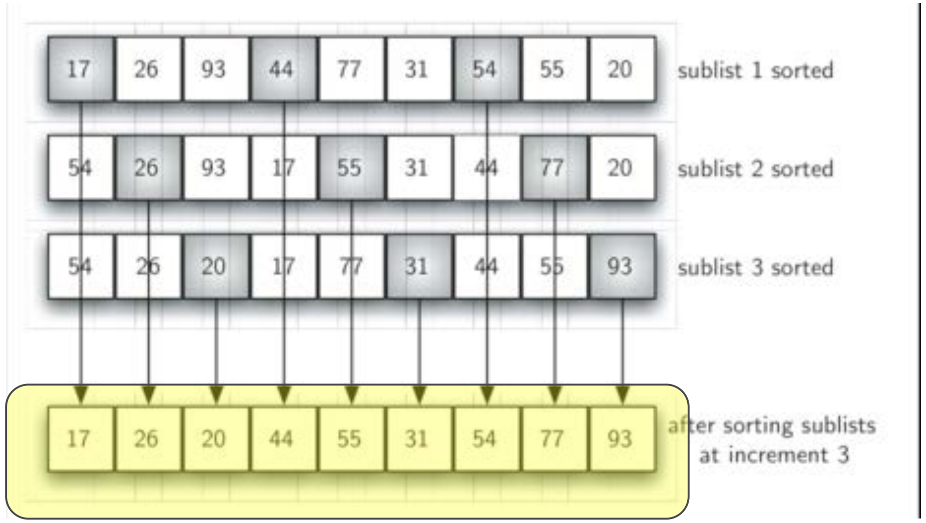
谢尔排序Shell Sort

- 我们注意到插入排序的比对次数，在最好的情况下是O(n)，这种情况发生在列表已是有序的情况下，实际上，**列表越接近有序，插入排序的比对次数就越少**
- 从这个情况入手，谢尔排序以插入排序作为基础，对无序表进行“间隔”划分子列表，每个子列表都执行插入排序



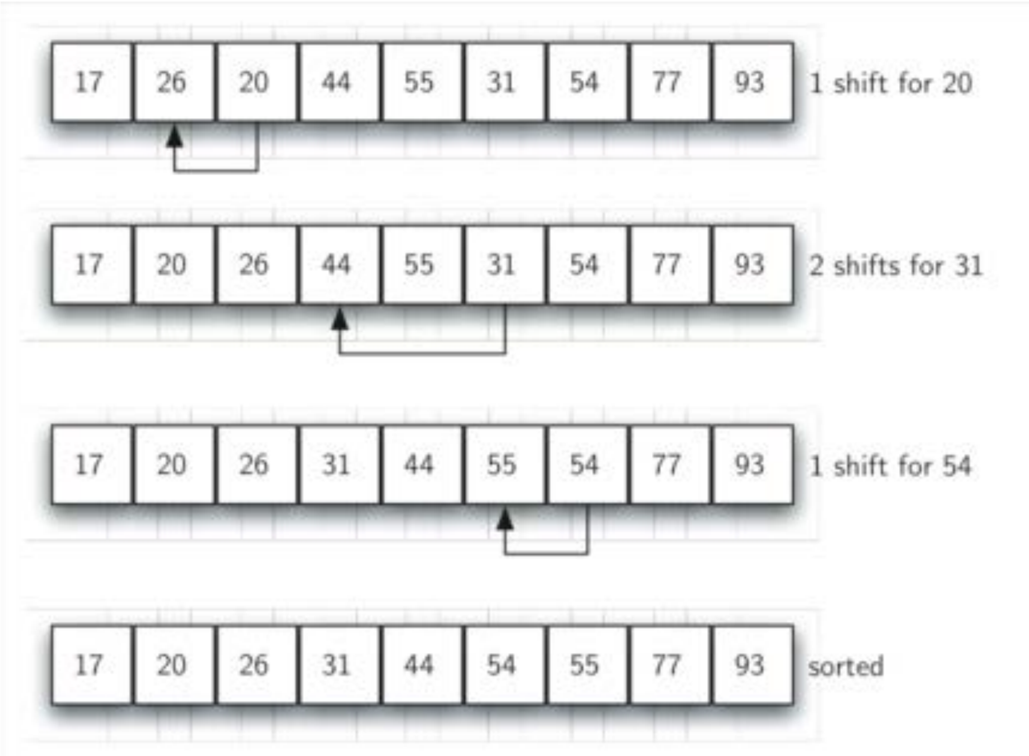
谢尔排序Shell Sort

- 随着子列表的数量越来越少，无序表的整体越来越接近有序，从而减少整体排序的比对次数
- 间隔为3的子列表，子列表分别插入排序后的整体状况更接近有序



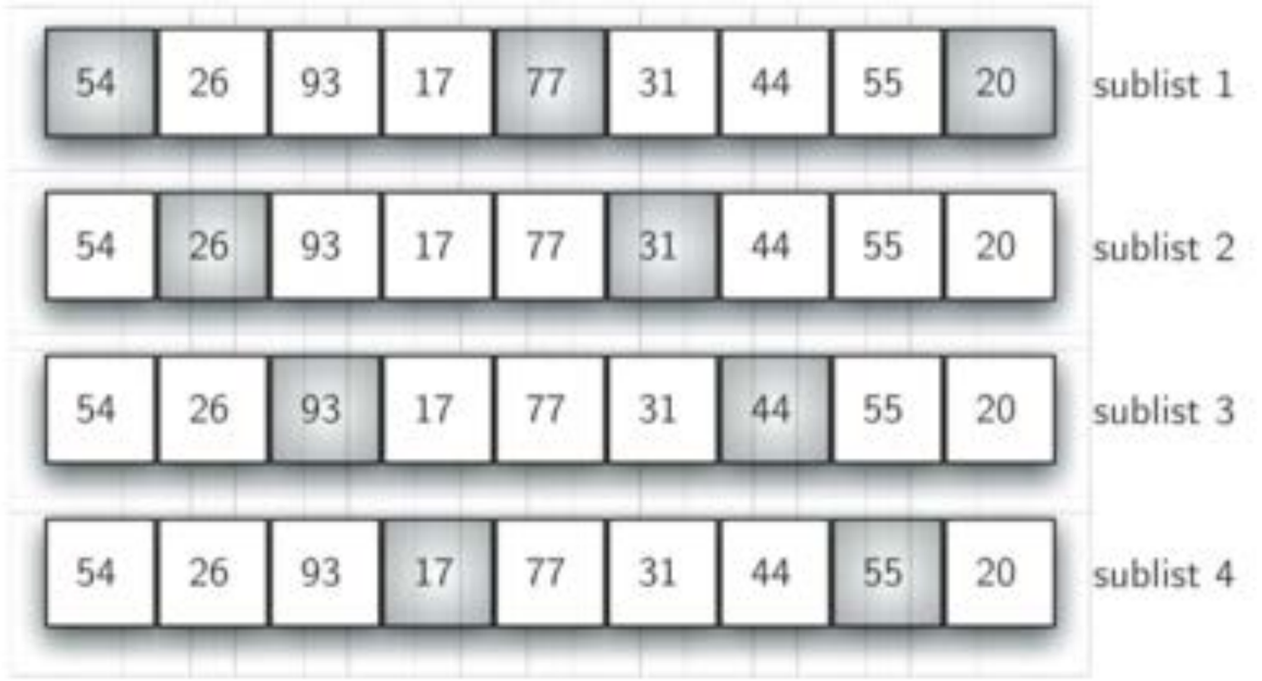
谢尔排序：思路

❖ 最后一趟是标准的插入排序，但由于前面几趟已经将列表处理到接近有序，这一趟仅需少数几次移动即可完成



谢尔排序：思路

❖ 子列表的间隔一般从n/2开始，每趟倍增：n/4, n/8……直到1



谢尔排序：代码

```python
def shellSort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:

      for startposition in range(sublistcount):
        gapInsertionSort(alist,startposition,sublistcount)

      print("After increments of size",sublistcount,
                                   "The list is",alist)

      sublistcount = sublistcount // 2

def gapInsertionSort(alist,start,gap):
    for i in range(start+gap,len(alist),gap):

        currentvalue = alist[i]
        position = i

        while position>=gap and alist[position-gap]>currentvalue:
            alist[position]=alist[position-gap]
            position = position-gap

        alist[position]=currentvalue
```

间隔设定

子列表排序

间隔缩小

谢尔排序：算法分析

- **粗看上去，谢尔排序以插入排序为基础，可能并不会比插入排序好**
- **但由于每趟都使得列表更加接近有序，这过程会减少很多原先需要的“无效”比对**

  对谢尔排序的详尽分析比较复杂，大致说是介于$O(n)$和$O(n^2)$之间

- **如果将间隔保持在2^k-1(1、3、5、7、15、31等等)，谢尔排序的时间复杂度约为O（$n^{3/2}$)**

$$O(n^{\frac{3}{2}}).$$

数据结构与算法（Python版）

归并排序算法及分析

陈斌 北京大学 gischen@pku.edu.cn

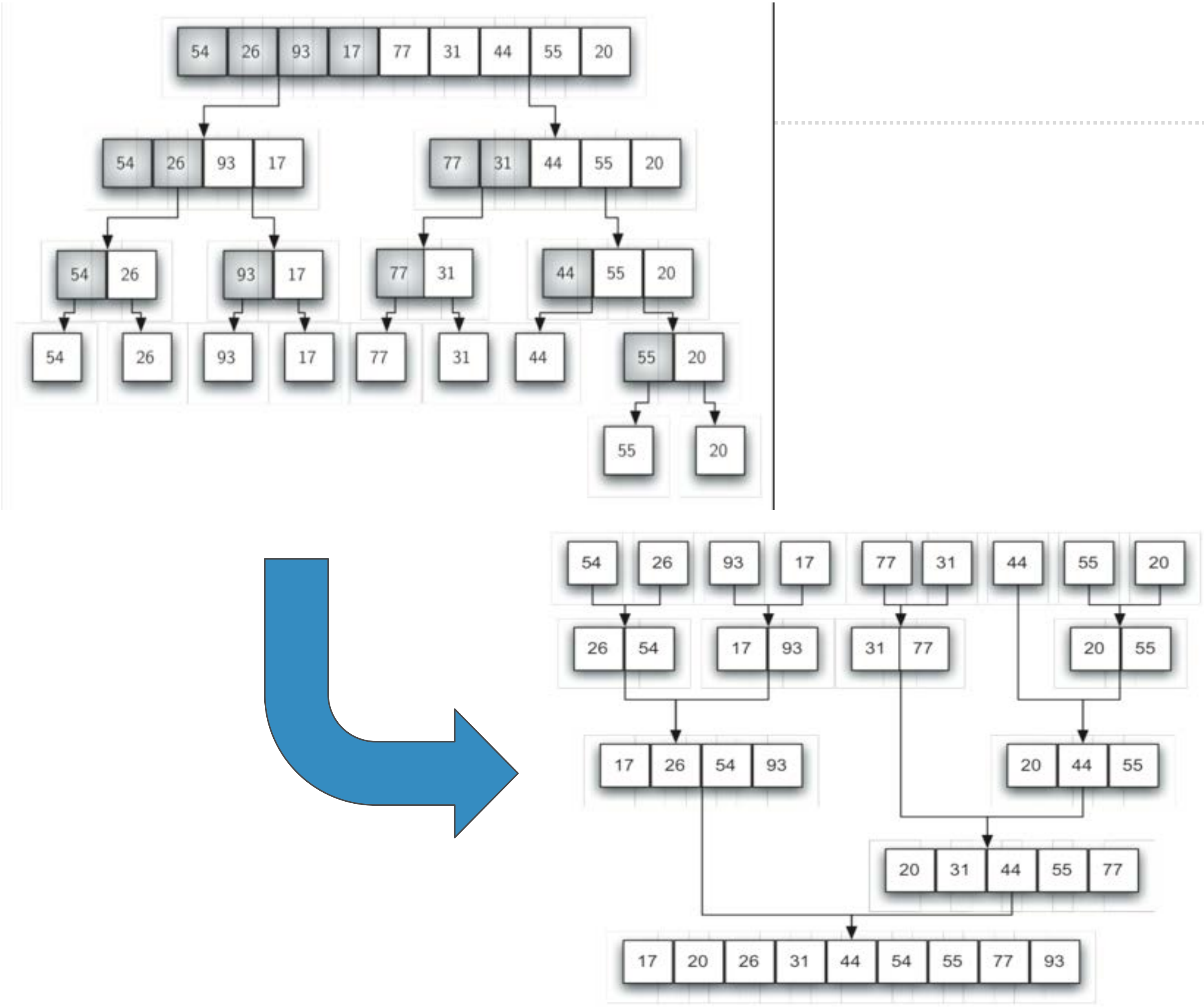
归并排序Merge Sort

- 下面我们来看看**分治策略**在排序中的应用
- 归并排序是递归算法，思路是将数据表持续分裂为两半，对两半分别进行归并排序

  递归的**基本结束条件**是：数据表仅有1个数据项，自然是排好序的；

  缩小规模：将数据表分裂为相等的两半，规模减为原来的二分之一；

  调用自身：将两半分别调用自身排序，然后将分别排好序的两半进行归并，得到排好序的数据表



54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20
54 26 93 17 77 31 44 55 20
26 54
17 93
31 77
20 55
17 26 54 93
20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93

```
def mergeSort(alist):
##    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i= j= k= 0
        while i<len(lefthalf) and j<len(righthalf):
            if lefthalf[i]<righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i<len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j<len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
##    print("Merging ",alist)
```

基本结束条件

递归调用

拉链式交错把左右半部
从小到大归并到结果列表中

归并左半部剩余项

归并右半部剩余项

另一个归并排序代码（更Pythonic）


```python
# merge sort
# 归并排序


def merge_sort(lst):
    # 递归结束条件
    if len(lst) <= 1:
        return lst

    # 分解问题，并递归调用
    middle = len(lst) // 2
    left = merge_sort(lst[:middle])  # 左半部排好序
    right = merge_sort(lst[middle:])  # 右半部排好序

    # 合并左右半部，完成排序
    merged = []
    while left and right:
        if left[0] <= right[0]:
            merged.append(left.pop(0))
        else:
            merged.append(right.pop(0))

    merged.extend(right if right else left)
    return merged
```

归并排序：算法分析

- 将归并排序分为两个过程来分析：**分裂**和**归并**
- 分裂的过程，借鉴二分查找中的分析结果，是对数复杂度，时间复杂度为O(log n)
- 归并的过程，相对于分裂的每个部分，其所有数据项都会被比较和放置一次，所以是线性复杂度，其时间复杂度是O(n)

  综合考虑，每次分裂的部分都进行一次O(n)的数据项归并，总的时间复杂度是O(nlog n)

归并排序：算法分析

- **最后，我们还是注意到两个切片操作**

  为了时间复杂度分析精确起见，

  可以通过**取消切片**操作，改为传递两个分裂部分的起始点和终止点，也是没问题的，

  只是算法可读性稍微牺牲一点点。

- **我们注意到归并排序算法使用了额外1倍的存储空间用于归并**

- **这个特性在对特大数据集进行排序的时候要考虑进去**

数据结构与算法（Python版）

快速排序算法及分析

陈斌 北京大学 gischen@pku.edu.cn

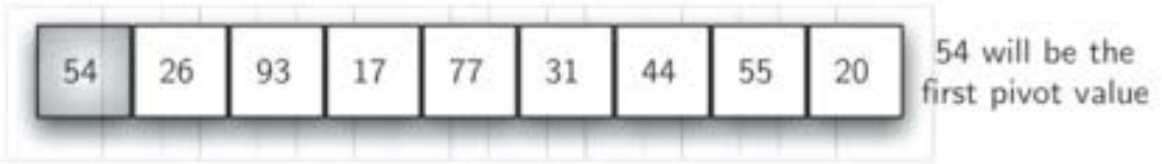
快速排序Quick Sort

- 快速排序的思路是依据一个“**中值**”数据项来把数据表分为**两半**：小于中值的一半和大于中值的一半，然后每部分分别进行快速排序（递归）

如果希望这两半拥有相等数量的数据项，则应该找到数据表的“中位数”

但找中位数需要计算开销！要想没有开销，只能随意找一个数来充当“中值”

比如，第1个数。



快速排序Quick Sort

- 快速排序的递归算法“递归三要素”如下
- **基本结束条件**：数据表仅有1个数据项，自然是排好序的
- **缩小规模**：根据“中值”，将数据表分为两半，最好情况是相等规模的两半
- **调用自身**：将两半分别调用自身进行排序（排序基本操作在分裂过程中）

快速排序：图示

- 分裂数据表的目标：找到“中值”的位置
- 分裂数据表的手段

  设置左右标（left/rightmark）

  左标向右移动，右标向左移动

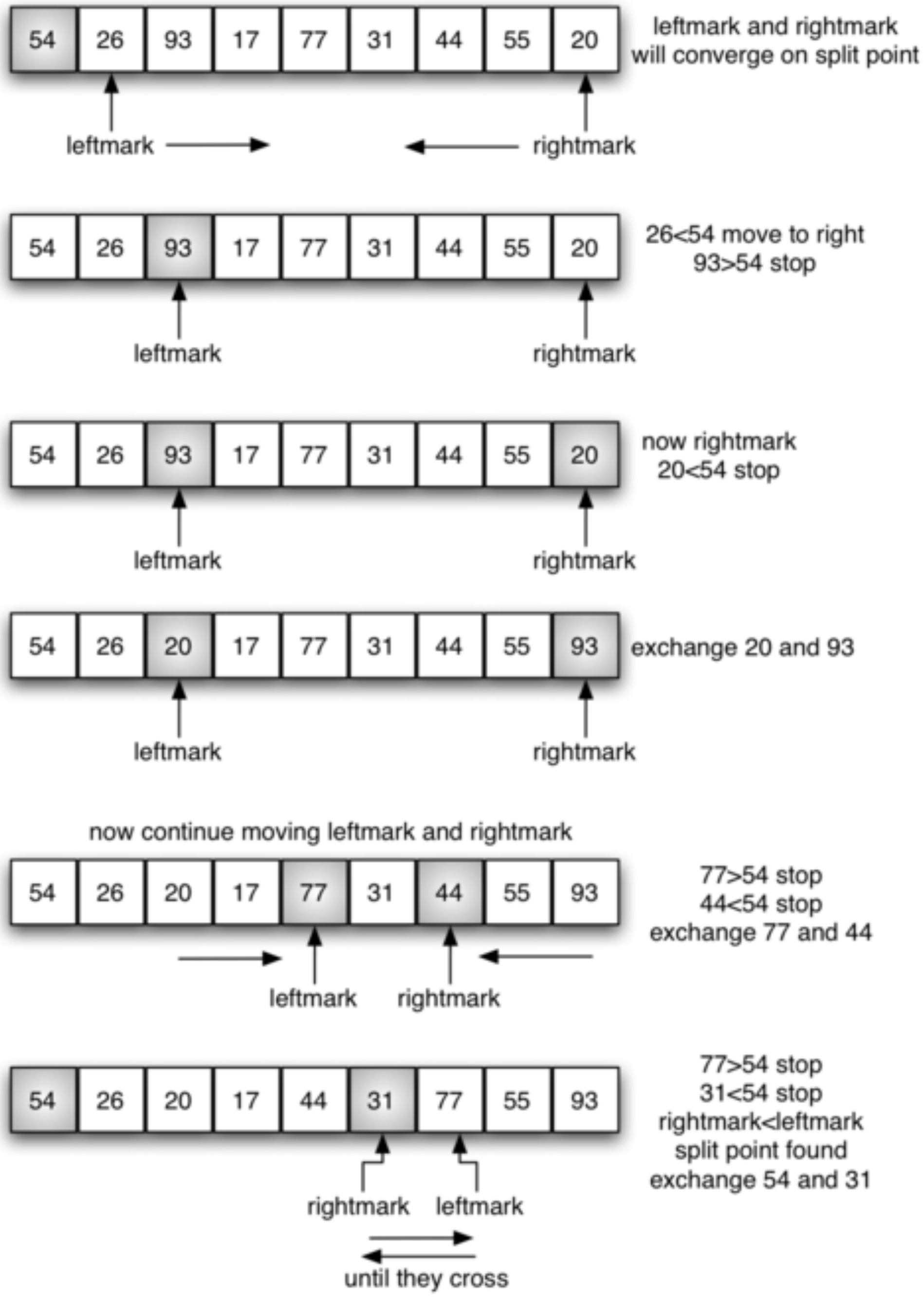
  - 左标一直向右移动，碰到比中值大的就停止
  - 右标一直向左移动，碰到比中值小的就停止
  - 然后把左右标所指的数据项交换

  继续移动，直到左标移到右标的右侧，停止移动

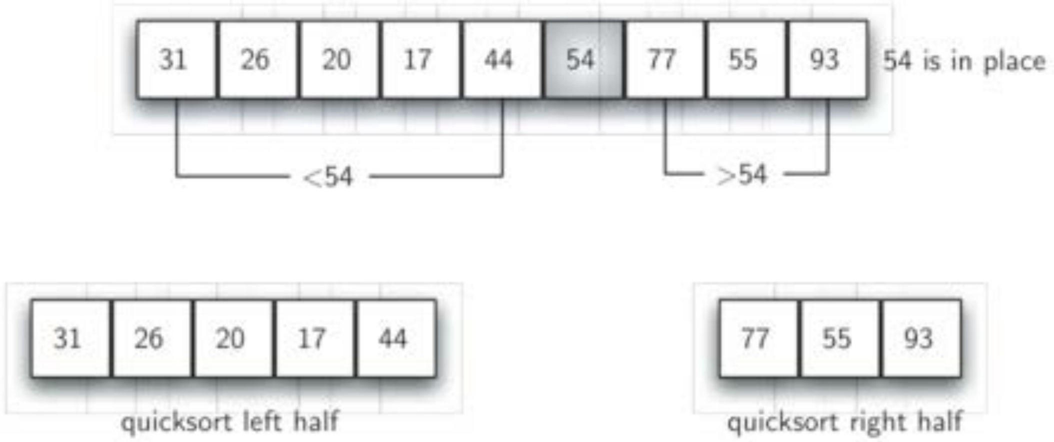
  这时右标所指位置就是“中值”应处的位置

  将中值和这个位置交换

  分裂完成，左半部比中值小，右半部比中值大



54 26 93 17 77 31 44 55 20
leftmark and rightmark will converge on split point
leftmark
rightmark
54 26 93 17 77 31 44 55 20
26<54 move to right
93>54 stop
leftmark
rightmark
54 26 93 17 77 31 44 55 20
now rightmark
20<54 stop
leftmark
rightmark
54 26 20 17 77 31 44 55 93
exchange 20 and 93
leftmark
rightmark
now continue moving leftmark and rightmark
54 26 20 17 77 31 44 55 93
77>54 stop
44<54 stop
exchange 77 and 44
leftmark
rightmark
54 26 20 17 44 31 77 55 93
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31
rightmark
leftmark
until they cross



31 26 20 17 44 54 77 55 93
54 is in place
<54
>54
31 26 20 17 44
quicksort left half
77 55 93
quicksort right half

快速排序：代码


```
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
    if first<last:

        splitpoint = partition(alist,first,last)

        quickSortHelper(alist,first,splitpoint-1)
        quickSortHelper(alist,splitpoint+1,last)

alist = [54,26,93,17,77,31,44,55,20]
quickSort(alist)
print(alist)
```

基本结束条件

分裂

递归调用

```python
def partition(alist,first,last):
    pivotvalue = alist[first]

    leftmark = first+1
    rightmark = last

    done = False
    while not done:

        while leftmark <= rightmark and \
                alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1

        while alist[rightmark] >= pivotvalue and \
                rightmark >= leftmark:
            rightmark = rightmark -1

        if rightmark < leftmark:
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark] = alist[rightmark]
            alist[rightmark] = temp

    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp


    return rightmark
```

选定“中值”

左右标初值

向右移动左标

向左移动右标

两标相错就结束移动

左右标的值交换

中值就位

中值点，也是分裂点

快速排序：算法分析

- 快速排序过程分为两部分：**分裂**和**移动**

  如果分裂总能把数据表分为相等的两部分，那么就是O(log n)的复杂度；

  而移动需要将每项都与中值进行比对，还是O(n)

- 综合起来就是O(nlog n)；
- 而且，算法运行过程中不需要额外的存储空间。

快速排序：算法分析

- 但是，如果不那么幸运的话，中值所在的分裂点过于偏离中部，造成左右两部分数量不平衡
- 极端情况，有一部分始终没有数据，这样时间复杂度就退化到$O(n^2)$

还要加上递归调用的开销（比冒泡排序还糟糕）

快速排序：算法分析

- 可以适当改进下**中值**的选取方法，让中值更具有代表性

  比如“三点取样”，从数据表的头、尾、中间选出中值

  会产生额外计算开销，仍然不能排除极端情况

- 还有什么采样具有代表性?

顺序查找：算法分析

- 顺序查找有序表的各种情况分析

Case	Best Case	Worst Case	Average Case
item is present	1	n	n/2
item is not present	1	n	n/2

- 实际上，就算法复杂度而言，仍然是O(n)
- 只是在数据项不存在的时候，有序表的查找能节省一些比对次数，但并不改变其数量级。