



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”

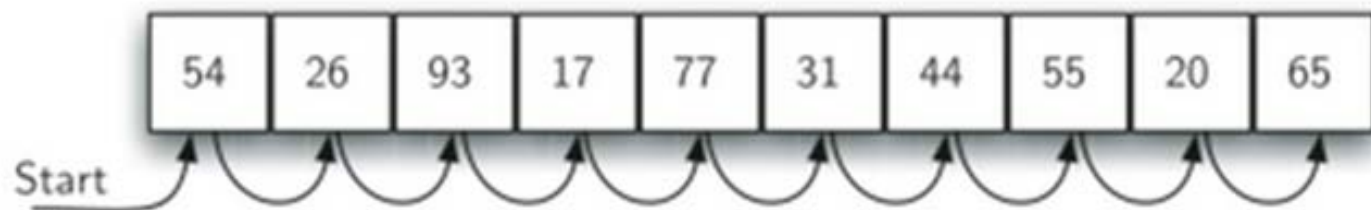
顺序查找Sequential Search

❖ 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
```

下标顺序增长

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

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？

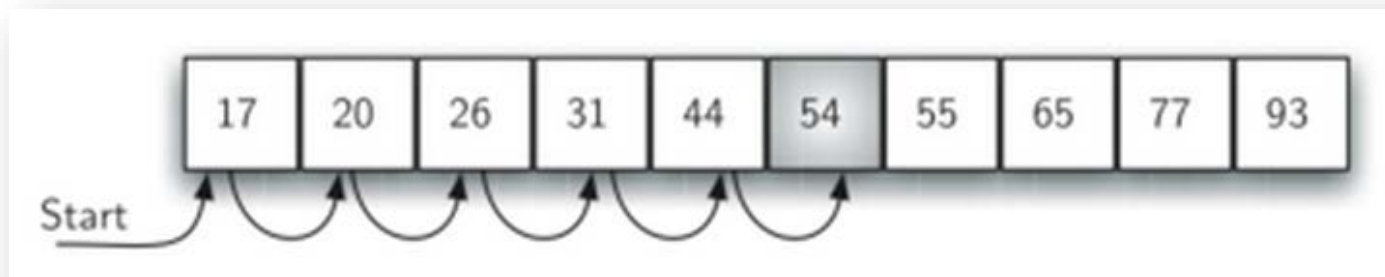
顺序查找：算法分析

❖ 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
```



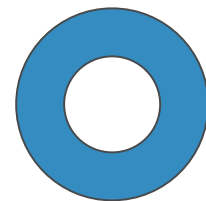
```
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]  
print(orderedSequentialSearch(testlist, 3))  
print(orderedSequentialSearch(testlist, 13))
```

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。





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

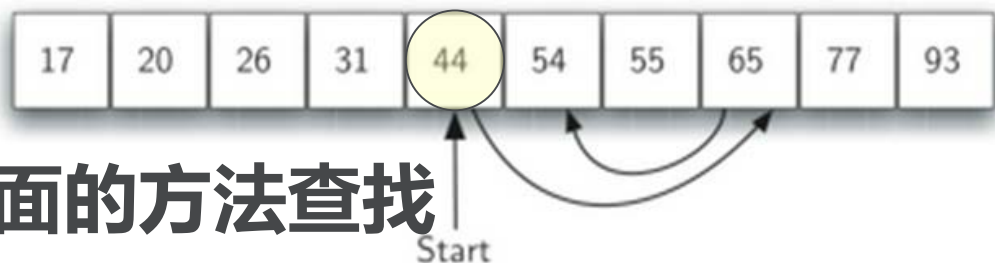
❖ 我们从列表中间开始比对!

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

```
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

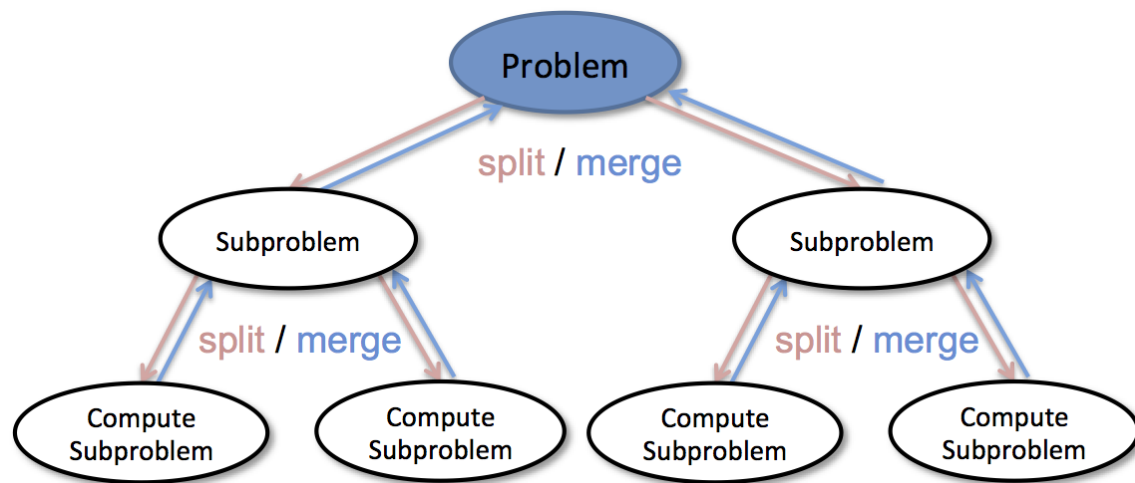
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```


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

```
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：

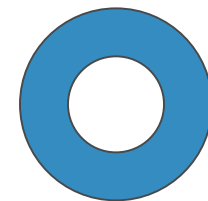
`binarySearch(alist[:midpoint], item)`

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 gisichen@pku.edu.cn

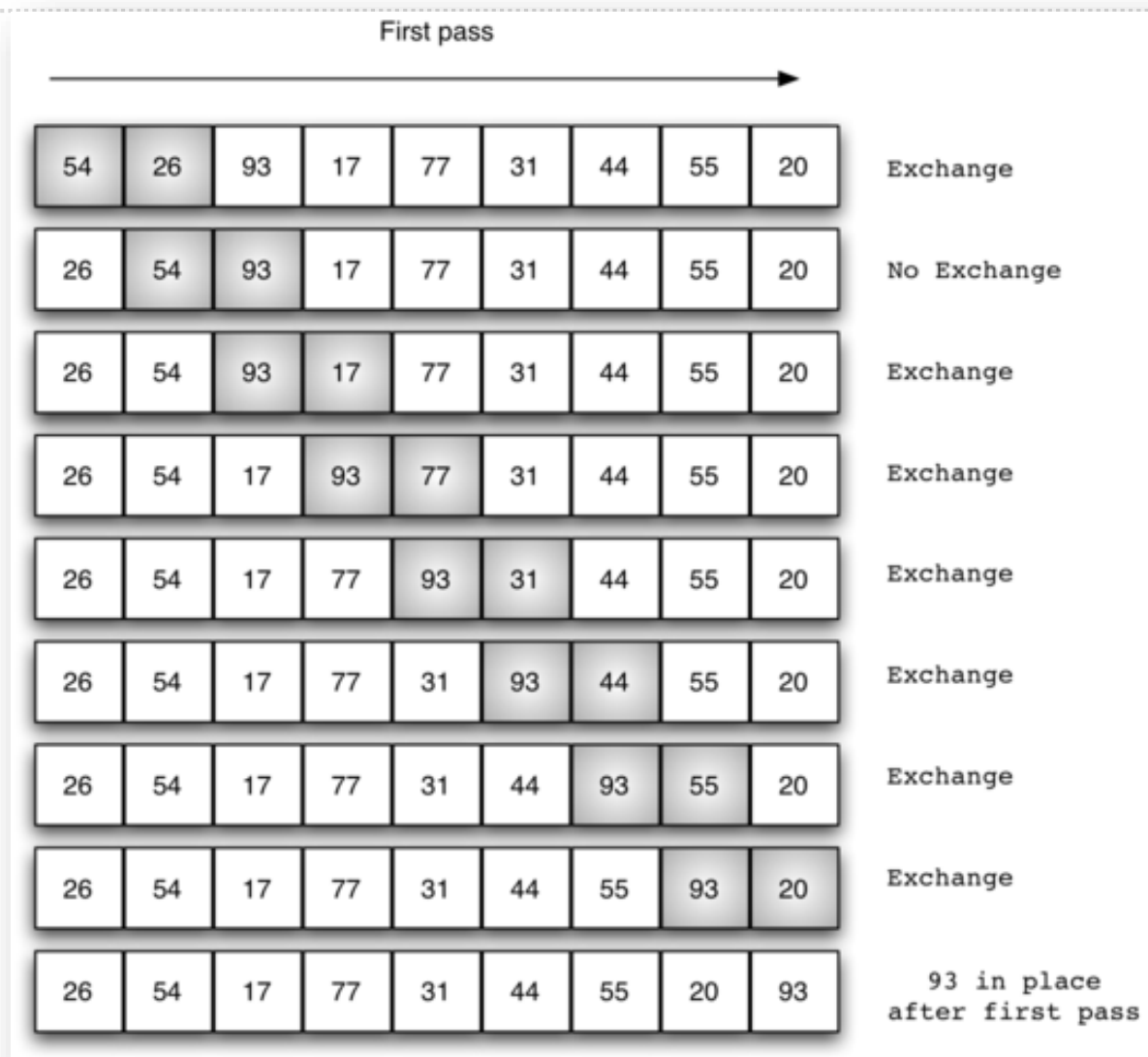
排序：冒泡排序Bubble Sort

- ❖ 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



冒泡排序：代码

```
def bubbleSort(alist):
```

n-1趟

```
    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
```

序错，交换

```
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 $1 \sim n-1$ 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

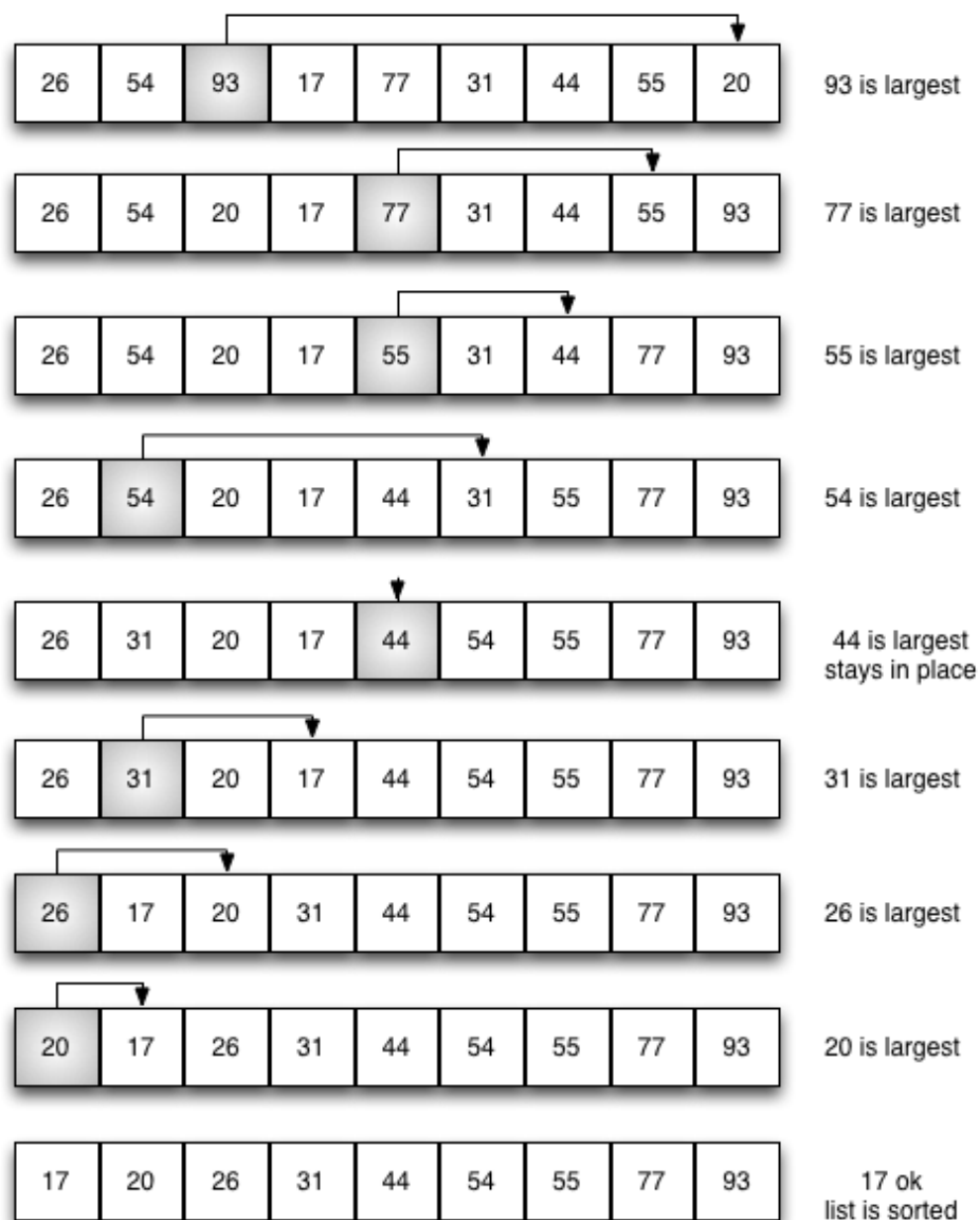
冒泡排序：性能改进

```
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)
```

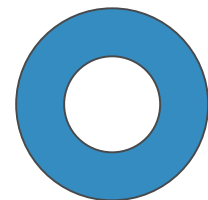
选择排序Selection Sort

- ❖ 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 $O(n)$



选择排序：代码

```
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



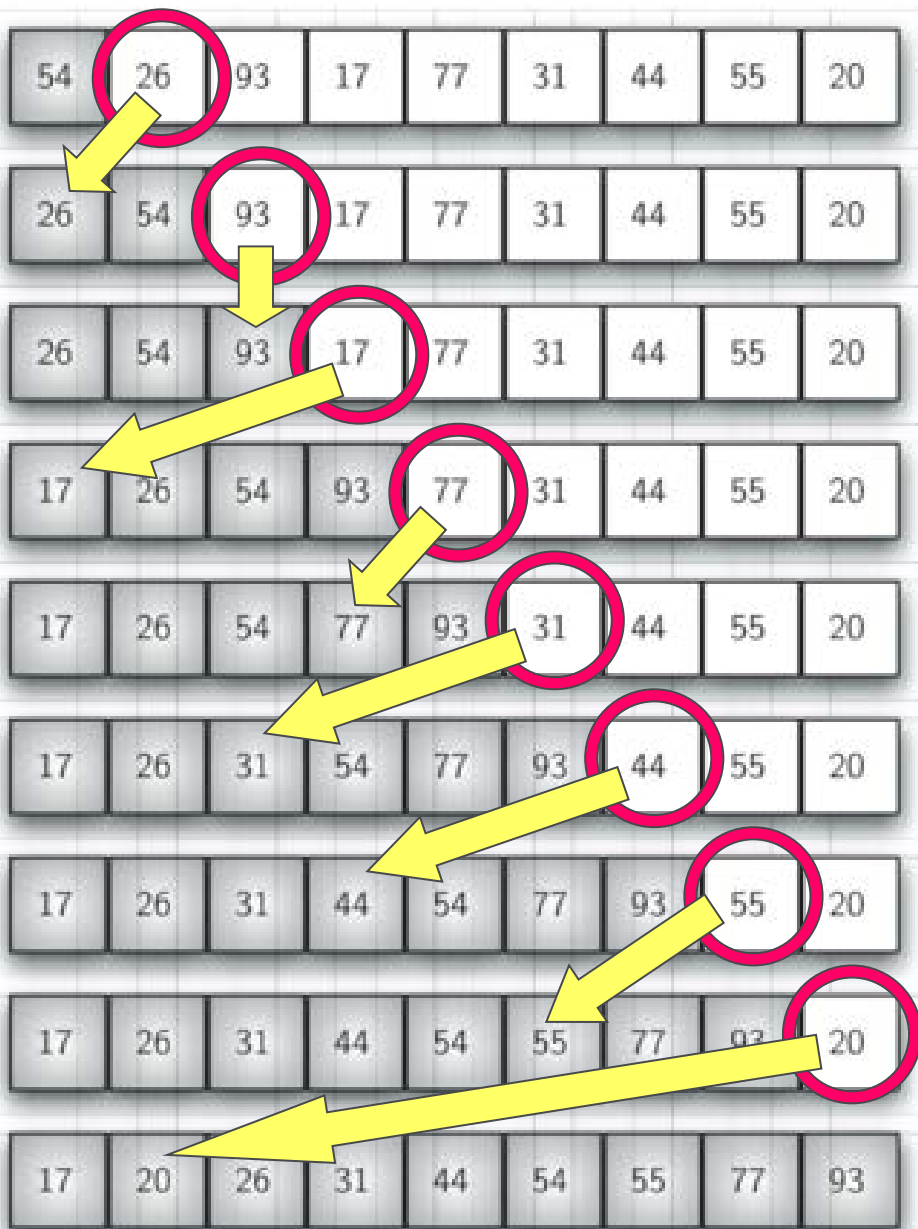
插入排序Insertion Sort

- ❖ 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 $O(n)$

Assume 54 is a sorted
list of 1 item



inserted 26

inserted 93

inserted 17

inserted 77

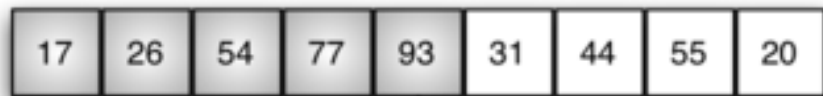
inserted 31

inserted 44

inserted 55

inserted 20

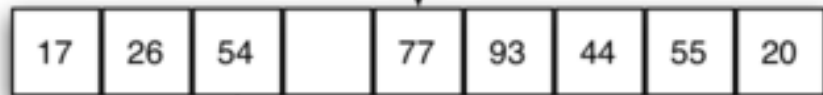
插入排序：思路



Need to insert 31
back into the sorted list



93>31 so shift it
to the right



77>31 so shift it
to the right



54>31 so shift it
to the right



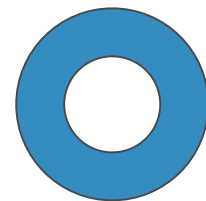
26<31 so insert 31
in this position

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] ← 新项/插入项  
        position = index  
  
        while position > 0 and alist[position-1] > currentvalue:  
            alist[position] = alist[position-1] ← 比对、移动  
            position = position - 1  
  
        alist[position] = currentvalue ← 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。





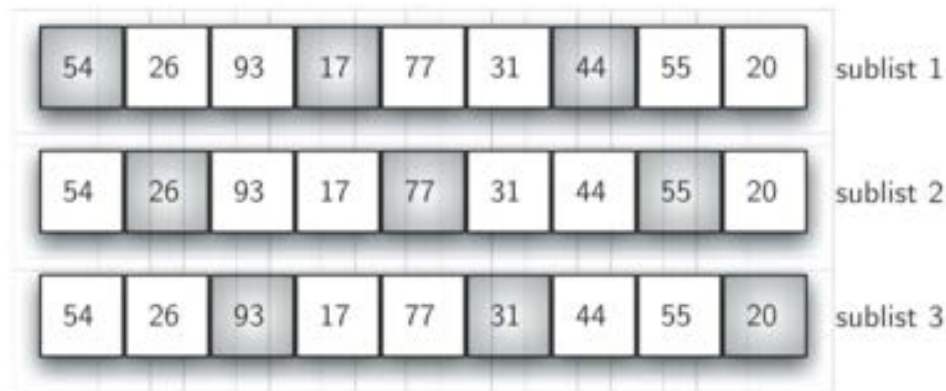
数据结构与算法 (Python版)

谢尔排序算法及分析

陈斌 北京大学 gischen@pku.edu.cn

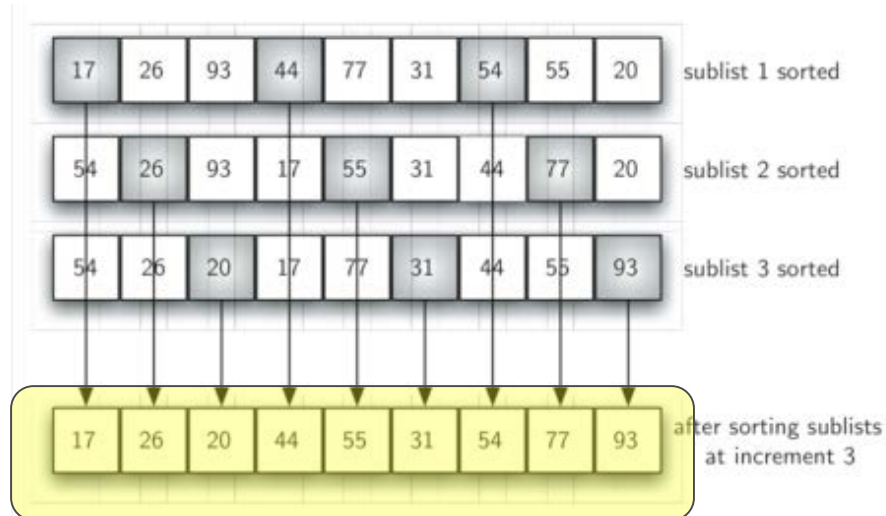
谢尔排序Shell Sort

- ❖ 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



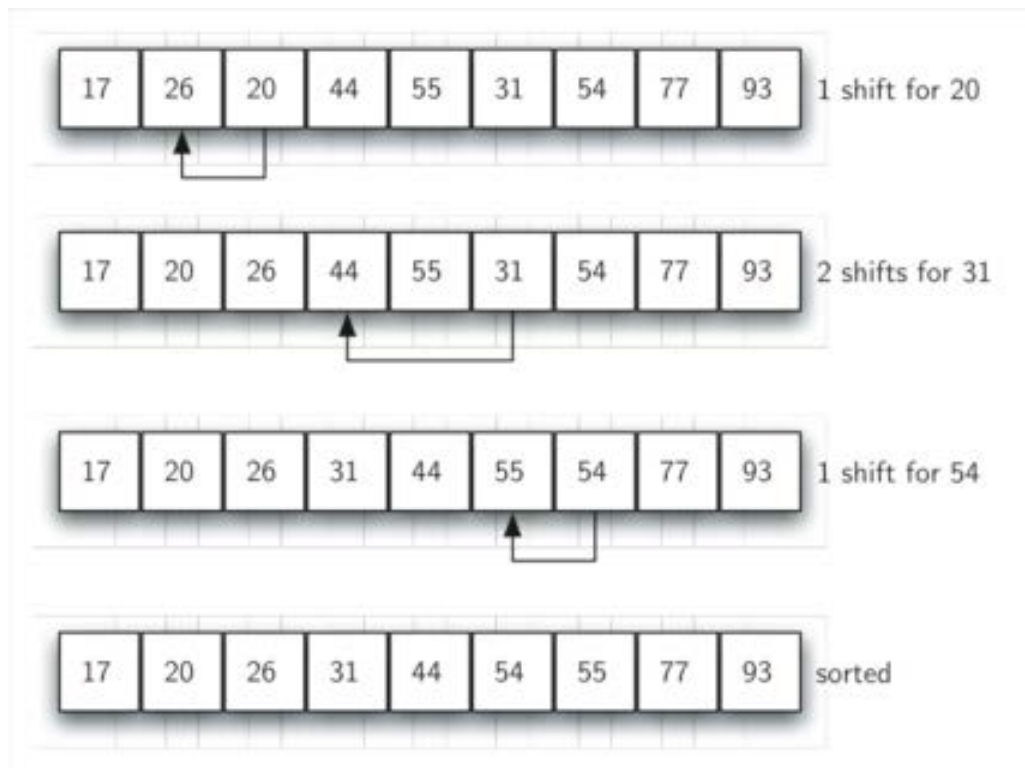
谢尔排序Shell Sort

- ❖ 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



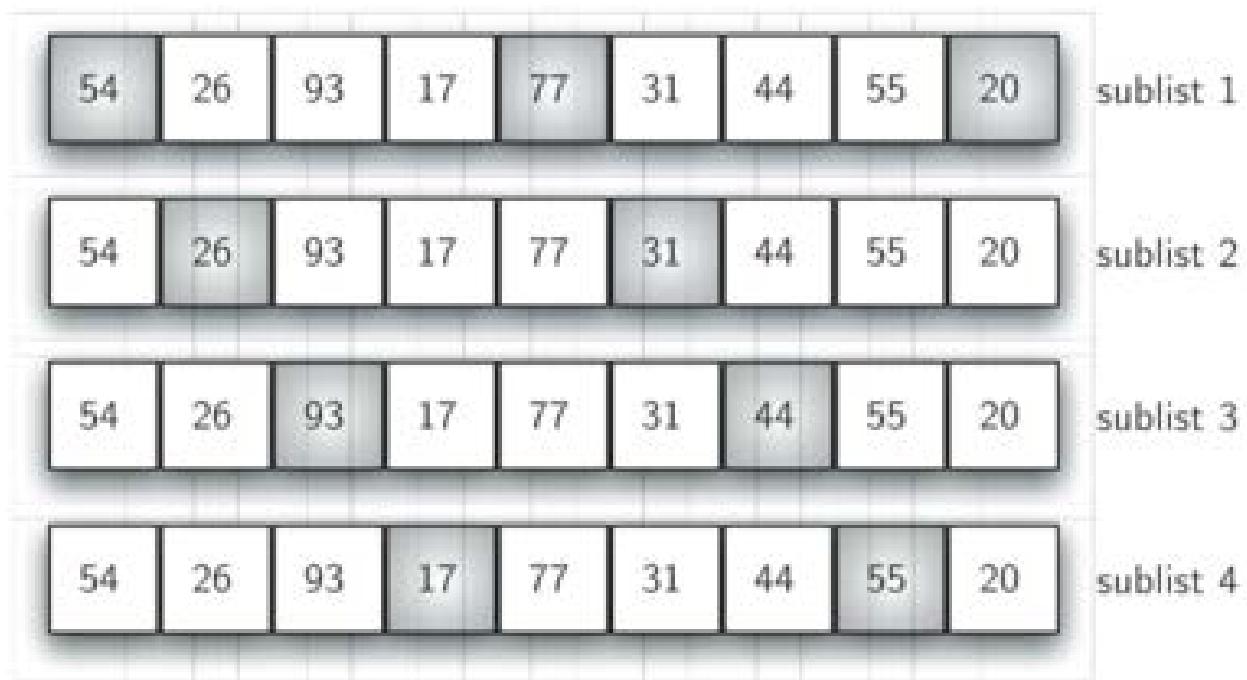
谢尔排序：思路

- ❖ 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

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

间隔设定

子列表排序

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

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

```
    sublistcount = sublistcount // 2
```

间隔缩小

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

```
        currentvalue = alist[i]  
        position = i
```

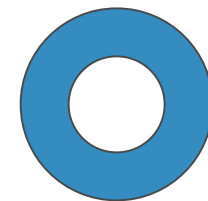
```
        while position>=gap and alist[position-gap]>currentvalue:  
            alist[position]=alist[position-gap]  
            position = position-gap
```

```
        alist[position]=currentvalue
```


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，这个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 $O(n^{3/2})$

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





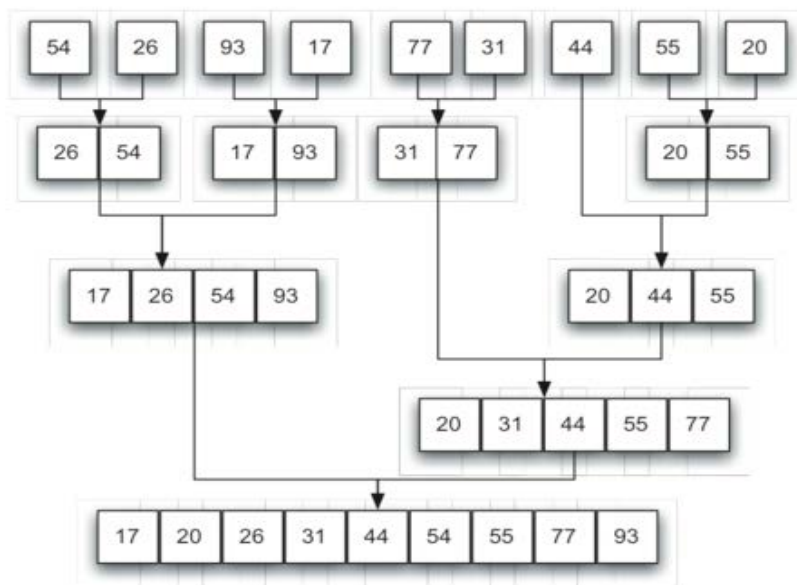
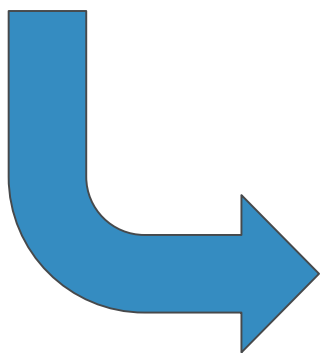
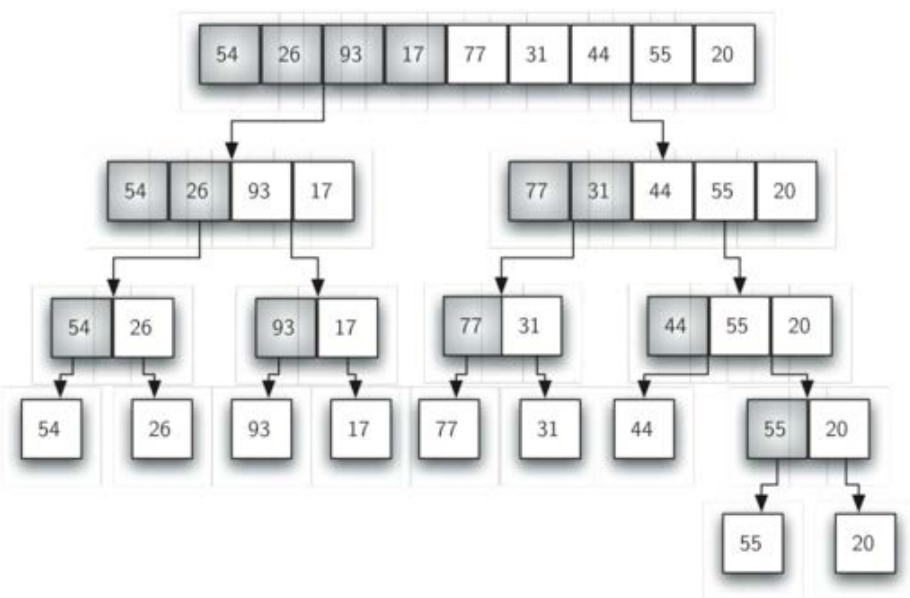
数据结构与算法 (Python版)

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



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

基本结束条件

```
mergeSort(lefthalf)
mergeSort(righthalf)
```

递归调用

```
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
```

拉链式交错把左右半部
从小到大归并到结果列表中

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

归并左半部剩余项

```
while j<len(righthalf):
    alist[k]=righthalf[j]
    j=j+1
    k=k+1
```

归并右半部剩余项

```
##    print("Merging ",alist)
```

另一个归并排序代码 (更Pythonic)

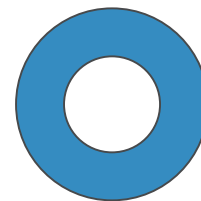
```
1  # merge sort
2  # 归并排序
3
4
5  def merge_sort(lst):
6      # 递归结束条件
7      if len(lst) <= 1:
8          return lst
9
10     # 分解问题, 并递归调用
11     middle = len(lst) // 2
12     left = merge_sort(lst[:middle]) # 左半部排好序
13     right = merge_sort(lst[middle:]) # 右半部排好序
14
15     # 合并左右半部, 完成排序
16     merged = []
17     while left and right:
18         if left[0] <= right[0]:
19             merged.append(left.pop(0))
20         else:
21             merged.append(right.pop(0))
22
23     merged.extend(right if right else left)
24     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 $O(n \log n)$

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

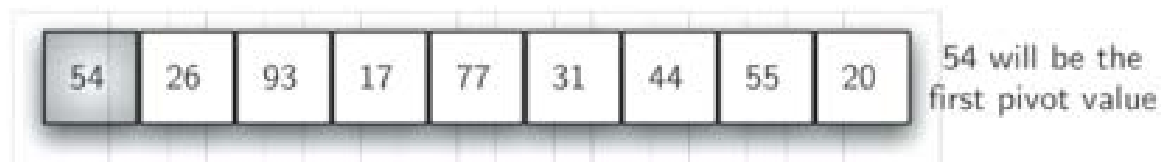
快速排序Quick Sort

❖ 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

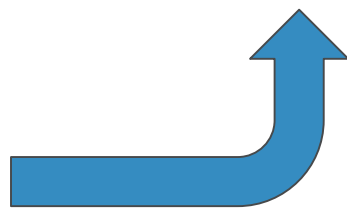
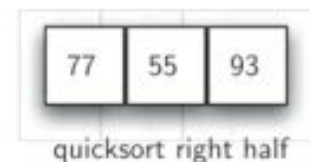
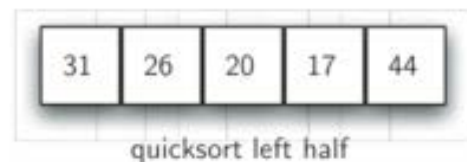
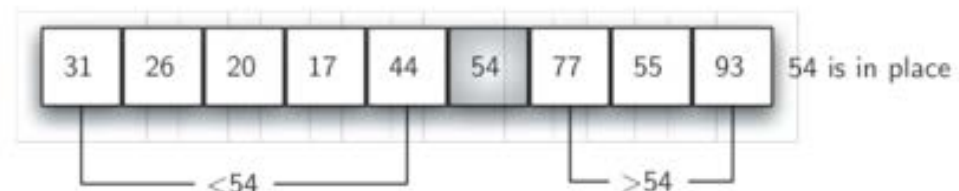
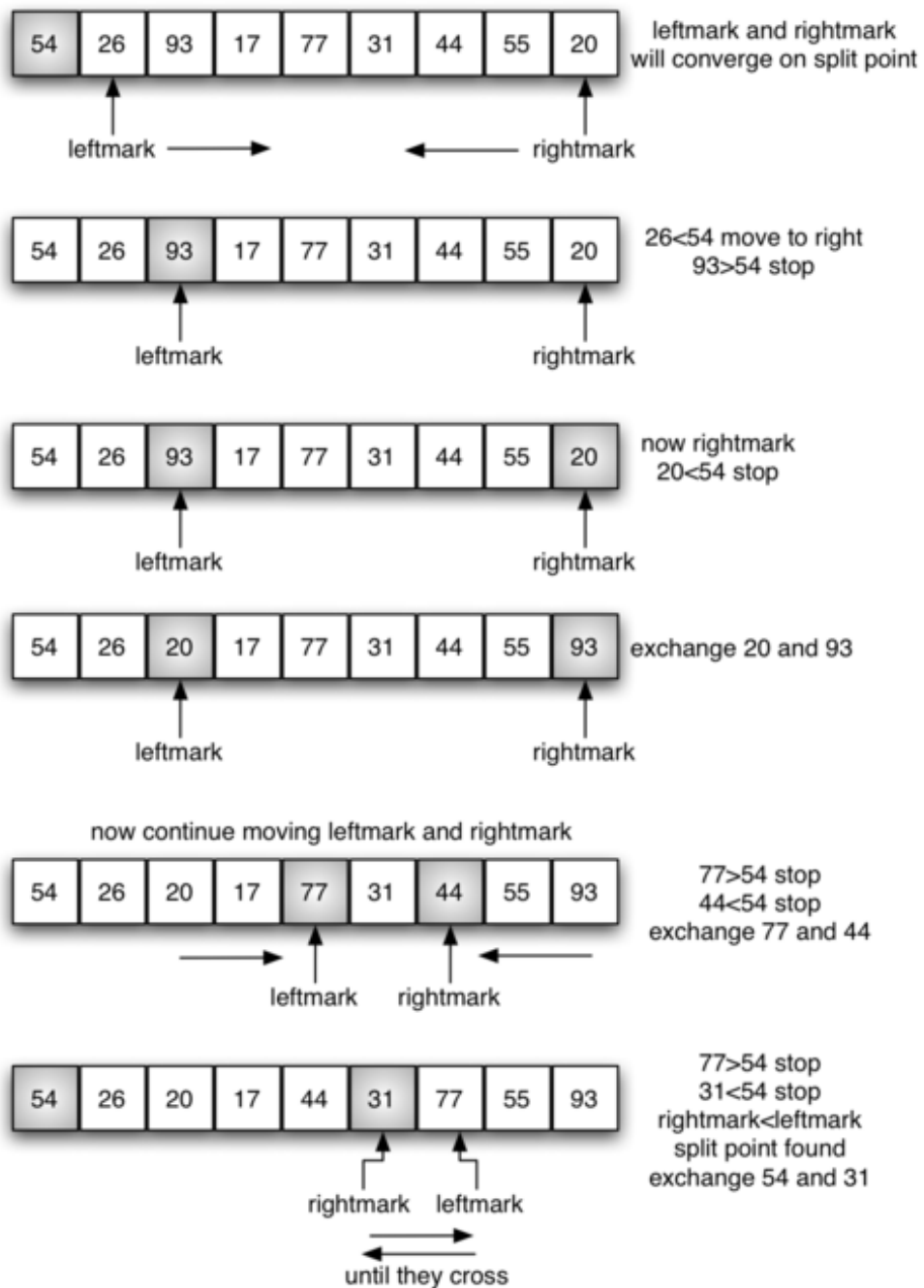
- 左标一直向右移动，碰到比中值大的就停止
- 右标一直向左移动，碰到比中值小的就停止
- 然后把左右标所指的数据项交换

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



快速排序：代码

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

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

基本结束条件

分裂

```
        splitpoint = partition(alist,first,last)
```

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

递归调用

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

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

选定“中值”

```
    leftmark = first+1
    rightmark = last
```

左右标初值

```
    done = False
    while not done:
```

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

向右移动左标

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

向左移动右标

```
    if rightmark < leftmark:
        done = True
```

两标相错就结束移动

```
    else:
```

```
        temp = alist[leftmark]
        alist[leftmark] = alist[rightmark]
        alist[rightmark] = temp
```

左右标的值交换

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

中值就位

```
    return rightmark
```

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 $O(n \log n)$ ；
- ❖ 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）

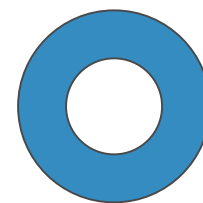
快速排序：算法分析

- ❖ 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？





数据结构与算法 (Python版)

什么是散列

陈斌 北京大学 gischen@pku.edu.cn

散列：Hashing

- ❖ 前面我们利用数据集中关于数据项之间排列关系的知识，来将查找算法进行了提升
- ❖ 如果数据项之间是按照**大小排好序**的话，就可以利用**二分查找**来降低算法复杂度。
- ❖ 现在我们进一步来构造一个新的数据结构，能使得查找算法的复杂度降到 $O(1)$ ，这种概念称为“散列Hashing”

散列：Hashing

- ❖ 能够使得查找的次数降低到**常数**级别，我们对数据项所处的位置就必须有更多的**先验知识**。
- ❖ 如果我们**事先**能知道要找的数据项**应该**出现在数据集中的什么**位置**，就可以直接到那个位置看看数据项是否存在即可。
- ❖ 由数据项的**值**来确定其存放位置，如何能做到这一点呢？

散列：基本概念

- ❖ 散列表 (hash table, 又称哈希表) 是一种数据集, 其中数据项的存储方式尤其有利于将来快速的查找定位。
- ❖ 散列表中的每一个存储位置, 称为槽 (slot), 可以用来保存数据项, 每个槽有一个唯一的名称。

散列：基本概念

- ❖ 例如：一个包含11个槽的散列表，槽的名称分别为0~10
- ❖ 在插入数据项之前，每个槽的值都是None，表示空槽

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

散列：基本概念

- ❖ 实现从数据项到存储槽名称的转换的，称为**散列函数** (hash function)
- ❖ 下面示例中，散列函数接受**数据项**作为参数，返回**整数值**0 ~ 10，表示数据项存储的**槽号** (名称)

散列：示例

- ❖ 为了将数据项保存到散列表中，我们设计第一个散列函数

数据项：54, 26, 93, 17, 77, 31

- ❖ 有一种常用的散列方法是“**求余数**”，将数据项除以散列表的大小，得到的余数作为槽号。

实际上“求余数”方法会以不同形式出现在所有散列函数里

因为散列函数返回的槽号必须在散列表大小范围之内，所以一般会对散列表大小求余

散列：示例

- ❖ 本例中我们的散列函数是最简单的求余：

$$h(item) = item \% 11$$

- ❖ 按照散列函数 $h(item)$ ，为每个数据项计算出存放的位置之后，就可以将数据项存入相应的槽中

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

散列：示例

- ❖ 例子中的6个数据项插入后，占据了散列表11个槽中的6个。

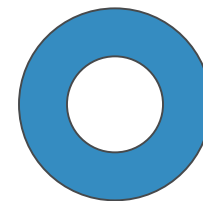
槽被数据项占据的比例称为散列表的“负载因子”，
这里负载因子为 $6/11$

- ❖ 数据项都保存到散列表后，查找就无比简单
- ❖ 要查找某个数据项是否存在于表中，我们只需要使用同一个散列函数，对查找项进行计算，测试下返回的槽号所对应的槽中是否有数据项即可
实现了 $O(1)$ 时间复杂度的查找算法。

散列：示例

- ❖ 不过，你可能也看出这个方案的问题所在，这组数据相当凑巧，各自占据了不同槽
- ❖ 假如还要保存44， $h(44)=0$ ，它跟77被分配到同一个0#槽中，这种情况称为“**冲突collision**”，我们后面会讨论到这个问题的解决方案。

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54





数据结构与算法 (Python版)

完美散列函数

陈斌 北京大学 gischen@pku.edu.cn

完美散列函数

- ❖ 给定一组数据项，如果一个散列函数能把每个数据项映射到不同的槽中，那么这个散列函数就可以称为“**完美散列函数**”
对于固定的一组数据，总是能想办法设计出完美散列函数
- ❖ 但如果数据项经常性的变动，很难有一个系统性的方法来设计对应的完美散列函数
当然，冲突也不是致命性的错误，我们会有办法处理的。

完美散列函数

- ❖ 获得完美散列函数的一种方法是扩大散列表的容量，大到**所有可能出现**的数据项都能够占据不同的槽
- ❖ 但这种方法对于可能数据项范围过大的情况并不实用
假如我们要保存手机号（11位数字），完美散列函数得要求散列表具有百亿个槽！会浪费太多存储空间
- ❖ 退而求其次，好的散列函数需要具备特性
冲突最少（近似完美）、计算难度低（额外开销小）、充分分散数据项（节约空间）

完美散列函数的更多用途

- ❖ 除了用于在散列表中安排数据项的存储位置，散列技术还用在信息处理的很多领域
- ❖ 由于完美散列函数能够对任何不同的数据生成不同的散列值，如果把散列值当作数据的“指纹”或者“摘要”，这种特性被广泛应用在数据的一致性校验上

由任意长度的数据生成长度固定的“指纹”，还要求具备唯一性，这在数学上是无法做到的，但设计巧妙的“准完美”散列函数却能在实用范围内做到这一点。

完美散列函数的更多用途

❖ 作为一致性校验的数据“指纹”函数需要具备如下的特性

压缩性：任意长度的数据，得到的“指纹”长度是固定的；

易计算性：从原数据计算“指纹”很容易；（从指纹计算原数据是不可能的）；

抗修改性：对原数据的微小变动，都会引起“指纹”的大改变；

抗冲突性：已知原数据和“指纹”，要找到相同指纹的数据（伪造）是非常困难的

散列函数MD5/SHA

- ❖ 最著名的近似完美散列函数是**MD5**和**SHA**系列函数
- ❖ MD5 (Message Digest) 将任何长度的数据变换为固定长为128位 (16字节) 的“摘要”
128位二进制已经是一个极为巨大的数字空间：
据说是地球沙粒的数量

散列函数MD5/SHA

❖ SHA (Secure Hash Algorithm) 是另一组散列函数

SHA-0/SHA-1输出散列值160位 (20字节),

SHA-256/SHA-224分别输出256位、224位,

SHA-512/SHA-384分别输出512位和384位

❖ 160位二进制相当于10的48次方, 地球上水分子数量估计是47次方

❖ 256位二进制相当于10的77方, 已知宇宙所有基本粒子大约是72 ~ 87次方

散列函数MD5/SHA

- ❖ 虽然近年发现MD5/SHA-0/SHA-1三种散列函数
- ❖ 能够以极特殊的情况来构造个别碰撞（散列冲突）
- ❖ 但在实用中从未有实际的威胁。
- ❖ 关于数量级的知识：
[http://zh.wikipedia.org/wiki/%E6%95%B0%E9%87%8F%E7%BA%A7_\(%E6%95%B0\)](http://zh.wikipedia.org/wiki/%E6%95%B0%E9%87%8F%E7%BA%A7_(%E6%95%B0))

Python的散列函数库hashlib

❖ Python自带MD5和SHA系列的散列函数库：hashlib

包括了md5 / sha1 / sha224 / sha256 / sha384 / sha512等6种散列函数

```
>>> import hashlib
>>> hashlib.md5("hello world!").hexdigest()
'fc3ff98e8c6a0d3087d515c0473f8677'
>>> hashlib.sha1("hello world!").hexdigest()
'430ce34d020724ed75a196dfc2ad67c77772d169'
```

Python的散列函数库hashlib

- ❖ 除了对单个字符串进行散列计算之外,
- ❖ 还可以用update方法来对任意长的数据分部分来计算,
- ❖ 这样不管多大的数据都不会有内存不足的问题。

```
>>> import hashlib
>>> m= hashlib.md5()
>>> m.update("hello world!")
>>> m.update("this is part #2")
>>> m.update("this is part #3")
>>> m.hexdigest()
'a12edc8332947a3e02e5668c6484b93a'
>>> |
```

完美散列函数用于数据一致性校验

- ❖ 数据文件一致性判断
- ❖ 为每个文件计算其散列值，仅对比其散列值即可得知是否文件内容相同；
- ❖ 用于网络文件下载完整性校验；
- ❖ 用于文件分享系统：网盘中相同的文件（尤其是电影）可以无需存储多次。

2).rm	939.7M	我的文件	✓ 极速秒传
-------	--------	------	--------

青版	1.41G	我的文件	✓ 极速秒传
----	-------	------	--------

mb	1.73G	我的文件	41%(44.48 KB/s)		×
----	-------	------	-----------------	--	---

完美散列函数用于数据一致性校验

- ❖ 加密形式保存密码
- ❖ 仅保存密码的散列值，用户输入密码后，计算散列值并比对；
- ❖ 无需保存密码的明文即可判断用户是否输入了正确的密码。

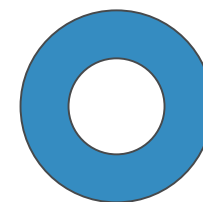
完美散列函数用于数据一致性校验

❖ 防文件篡改：原理同数据文件一致性判断

当然还有更多密码学机制来保护数据文件，
防篡改，防抵赖，是电子商务的信息技术基础。

❖ 彩票投注应用

彩民下注前，机构将中奖的结果散列值公布，
然后彩民投注，开奖后，彩民可以通过公布的结果和散列值对比，验证机构是否作弊。





数据结构与算法 (Python版)

散列函数设计

陈斌 北京大学 gischen@pku.edu.cn

散列函数设计：折叠法

❖ 折叠法设计散列函数的基本步骤是

将数据项按照位数分为若干段，

再将几段数字相加，

最后对散列表大小求余，得到散列值

❖ 例如，对电话号码62767255

可以两位两位分为4段（62、76、72、55）

相加（ $62+76+72+55=265$ ）

散列表包括11个槽，那么就是 $265\%11=1$

所以 $h(62767255)=1$

散列函数设计：折叠法

- ❖ 有时候折叠法还会包括一个**隔数反转**的步骤

比如 (62、76、72、55) 隔数反转为 (62、67、72、55)

再累加 ($62+67+72+55=256$)

对11求余 ($256\%11=3$)，所以 $h'(62767255)=3$

- ❖ 虽然隔数反转从理论上看来毫无必要，但这个步骤确实为折叠法得到散列函数提供了一种**微调**手段，以便更好符合散列特性

散列函数设计：平方取中法

- ❖ 平方取中法，首先将数据项做平方运算，然后取**平方数的中间**两位，再对散列表的大小求余
- ❖ 例如，对44进行散列
首先 $44 * 44 = 1936$
然后取中间的93
对散列表大小11求余， $93 \% 11 = 5$

散列函数设计：平方取中法

❖ 下表是两种散列函数的对比

两个都是完美散列函数

分散度都很好

平方取中法计算量稍大

Item	Remainder	Mid-Square
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

散列函数设计：非数项

- ❖ 我们也可以对非数字的数据项进行散列，把字符串中的每个字符看作ASCII码即可
如cat, $\text{ord}('c') == 99$, $\text{ord}('a') == 96$, $\text{ord}('t') == 116$
- ❖ 再将这些整数累加，对散列表大小求余

$$\begin{array}{ccccccc} c & & a & & t & & \\ \downarrow & & \downarrow & & \downarrow & & \\ 99 & + & 97 & + & 116 & = & 312 \\ & & & & & & 312 \% 11 \longrightarrow 4 \end{array}$$

代码

```
def hash(astring, tablesize):  
    sum = 0  
    for pos in range(len(astring)):  
        sum = sum + ord(astring[pos])  
  
    return sum%tablesize
```

散列函数设计

❖ 当然，这样的散列函数对所有的**变位词**都返回相同的散列值

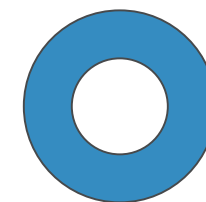
为了防止这一点，可以将字符串所在的位置作为权重因子，乘以ord值

position

1	2	3	
c	a	t	
↓	↓	↓	
99*1	+ 97*2	+ 116*3	= 641
641 % 11 → 3			

散列函数设计

- ❖ 我们还可以设计出更多的散列函数方法，但要坚持的一个基本出发点是，散列函数**不能**成为存储过程和查找过程的**计算负担**
- ❖ 如果散列函数设计太过复杂，去花费大量的计算资源计算槽号
可能还不如简单地进行顺序查找或者二分查找
- ❖ 失去了散列本身的意义





数据结构与算法 (Python版)

冲突解决方案

陈斌 北京大学 gischen@pku.edu.cn

冲突解决方案

- ❖ 如果两个数据项被散列映射到同一个槽，需要一个系统化的方法在散列表中保存第二个数据项，这个过程称为“**解决冲突**”
- ❖ 前面提到，如果说散列函数是完美的，那就不会有散列冲突，但完美散列函数常常是不现实的
- ❖ 解决散列冲突成为散列方法中很重要的一部分。

冲突解决方案

- ❖ 解决散列的一种方法就是为冲突的数据项**再找一个**开放的空槽来保存
最简单的就是从冲突的槽开始往后扫描，直到碰到一个空槽
如果到散列表尾部还未找到，则从首部接着扫描
- ❖ 这种**寻找空槽**的技术称为 “**开放定址** open addressing”
- ❖ 向后**逐个槽寻找**的方法则是开放定址技术中的 “**线性探测** linear probing”

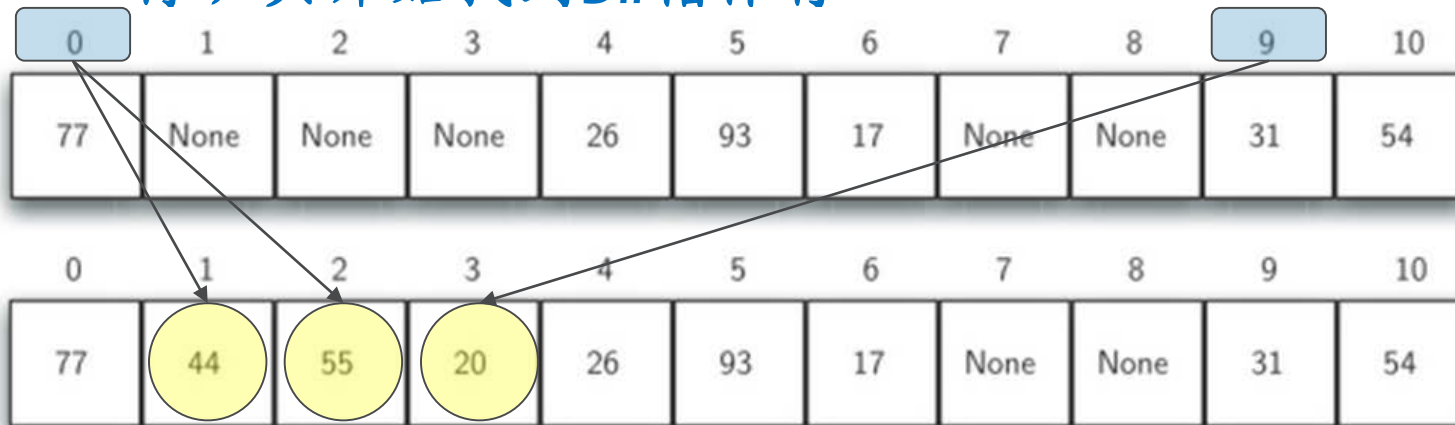
线性探测Linear Probing

❖ 我们把44、55、20逐个插入到散列表中

$h(44)=0$ ，但发现0#槽已被77占据，向后找到第一个空槽1#，保存

$h(55)=0$ ，同样0#槽已经被占据，向后找到第一个空槽2#，保存

$h(20)=9$ ，发现9#槽已经被31占据了，向后，再从头开始找到3#槽保存

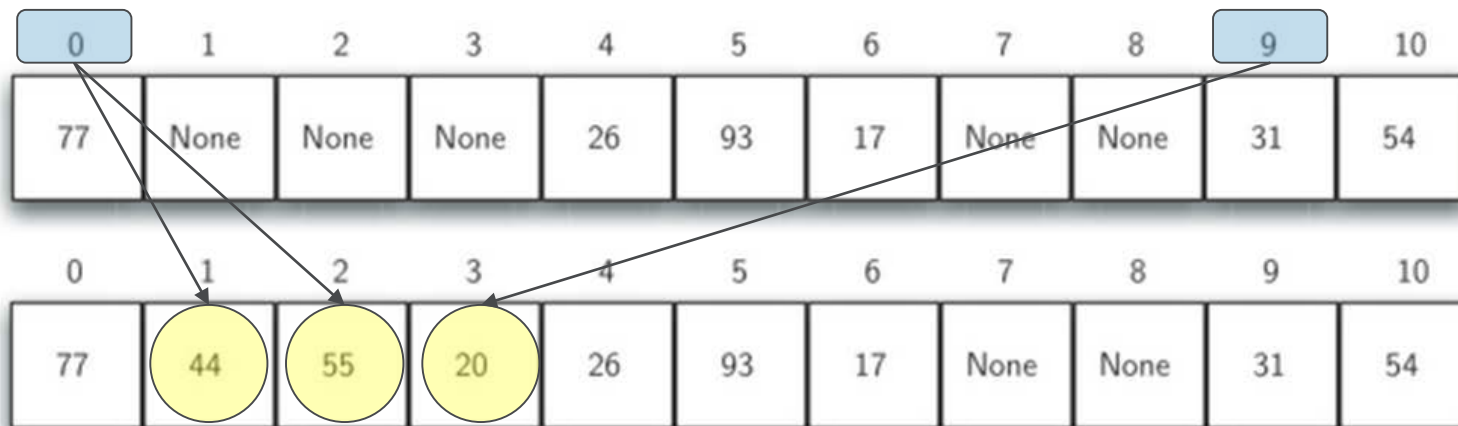


线性探测Linear Probing

❖ 采用线性探测方法来解决散列冲突的话，
则散列表的查找也遵循同样的规则

如果在散列位置没有找到查找项的话，就必须向
后做**顺序查找**

直到找到查找项，或者碰到空槽（查找失败）。



线性探测的改进

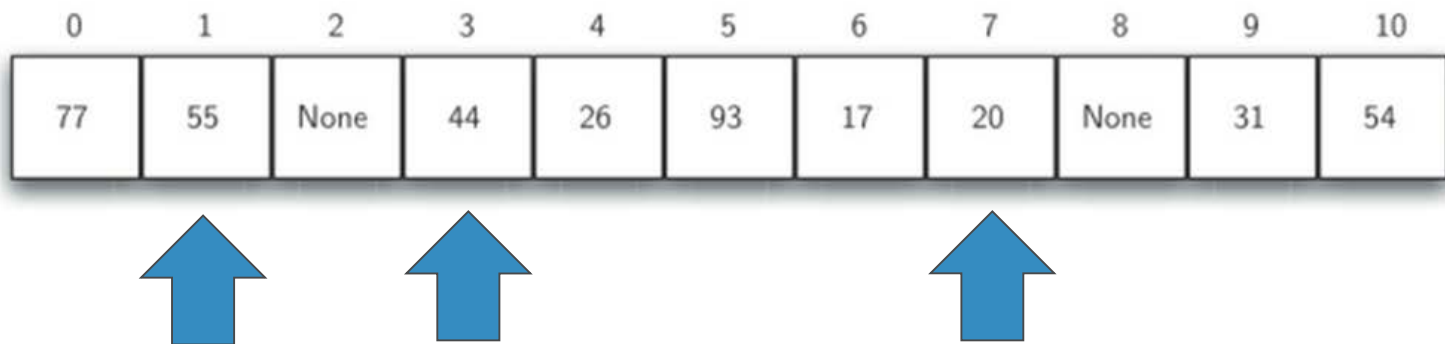
- ❖ 线性探测法的一个缺点是有**聚集** (clustering) 的趋势
- ❖ 即如果同一个槽冲突的数据项较多的话, 这些数据项就会在槽附近聚集起来
- ❖ 从而连锁式影响其它数据项的插入。

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

冲突解决方案：线性探测的改进

❖ 避免聚集的一种方法就是将线性探测扩展，从逐个探测改为跳跃式探测

下图是“+3”探测插入44、55、20



冲突解决方案：再散列rehashing

❖ 重新寻找空槽的过程可以用一个更为通用的“再散列rehashing”来概括

$\text{newhashvalue} = \text{rehash}(\text{oldhashvalue})$

对于线性探测来说, $\text{rehash}(\text{pos}) = (\text{pos} + 1) \% \text{sizeof table}$

“+3”的跳跃式探测则是: $\text{rehash}(\text{pos}) = (\text{pos} + 3) \% \text{sizeof table}$

跳跃式探测的再散列通式是: $\text{rehash}(\text{pos}) = (\text{pos} + \text{skip}) \% \text{sizeof table}$

冲突解决方案：再散列rehashing

- ❖ 跳跃式探测中，需要注意的是skip的取值，不能被散列表大小整除，否则会产生周期，造成很多空槽永远无法探测到
一个技巧是，把散列表的大小设为素数，如例子的11

0	1	2	3	4	5	6	7	8	9	10
77	44	20	55	26	93	17	None	None	31	54

冲突解决方案：再散列rehashing

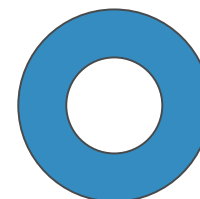
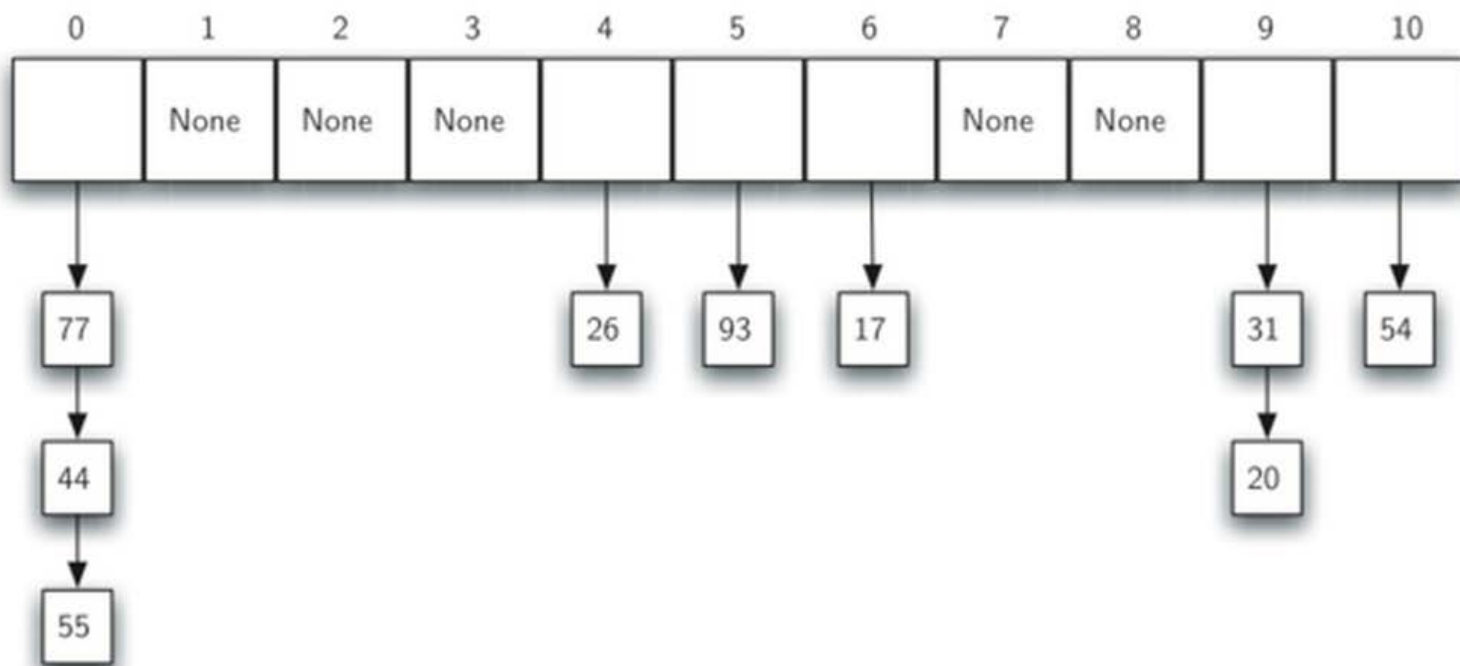
- ❖ 还可以将线性探测变为 “**二次探测** quadratic probing”
- ❖ 不再固定skip的值，而是逐步增加skip值，如1、3、5、7、9
- ❖ 这样槽号就会是原散列值以平方数增加：
 $h, h+1, h+4, h+9, h+16...$

0	1	2	3	4	5	6	7	8	9	10
77	44	20	55	26	93	17	None	None	31	54

冲突解决方案：数据项链Chaining

- ❖ 除了寻找空槽的开放定址技术之外，另一种解决散列冲突的方案是将容纳单个数据项的槽扩展为容纳**数据项集合**（或者对数据项链表的引用）
- ❖ 这样，散列表中的每个槽就可以容纳多个数据项，如果有散列冲突发生，只需要简单地将数据项添加到数据项集合中。
- ❖ 查找数据项时则需要查找同一个槽中的整个集合，当然，随着散列冲突的增加，对数据项的查找时间也会相应增加。

冲突解决方案：数据项链Chaining





数据结构与算法（Python版）

映射抽象数据类型及Python实现

陈斌 北京大学 gischen@pku.edu.cn

抽象数据类型 “映射” :ADT Map

- ❖ Python最有用的数据类型之一 “字典”
- ❖ 字典是一种可以保存key-data键值对的数据类型
其中关键码key可用于查询关联的数据值data
- ❖ 这种键值关联的方法称为 “**映射Map**”
- ❖ ADT Map的结构是键-值关联的无序集合
关键码具有唯一性
通过关键码可以唯一确定一个数据值

抽象数据类型“映射”:ADT Map

❖ ADT Map定义的操作如下:

`Map()`: 创建一个空映射, 返回空映射对象;

`put(key, val)`: 将`key-val`关联对加入映射中, 如果`key`已存在, 将`val`替换旧关联值;

`get(key)`: 给定`key`, 返回关联的数据值, 如不存在, 则返回`None`;

`del`: 通过`del map[key]`的语句形式删除`key-val`关联;

`len()`: 返回映射中`key-val`关联的数目;

`in`: 通过`key in map`的语句形式, 返回`key`是否存在于关联中, 布尔值

实现ADT Map

- ❖ 使用字典的优势在于，给定关键码key，能够很快得到关联的数据值data
- ❖ 为了达到**快速查找**的目标，需要一个支持高效查找的ADT实现
 - 可以采用列表数据结构加顺序查找或者二分查找
 - 当然，更为合适的是使用前述的散列表来实现，这样查找可以达到最快 $O(1)$ 的性能

实现ADT Map

- ❖ 下面，我们用一个HashTable类来实现ADT Map，该类包含了两个列表作为成员

其中一个slot列表用于保存key

另一个平行的data列表用于保存数据项

- ❖ 在slot列表查找到一个key的位置以后，在data列表对应相同位置的数据项即为关联数据

实现ADT Map：应用实例

```
H=HashTable()
H[54]="cat"
H[26]="dog"
H[93]="lion"
H[17]="tiger"
H[77]="bird"
H[31]="cow"
H[44]="goat"
H[55]="pig"
H[20]="chicken"
print(H.slots)
print(H.data)
```

```
print(H[20])      >>> [77, 44, 55, 20, 26, 93, 17, None, None, 31, 54]
                  ['bird', 'goat', 'pig', 'chicken', 'dog', 'lion', 'tiger', None,
print(H[17])      None, 'cow', 'cat']
H[20]='duck'      chicken
print(H[20])      tiger
print(H[99])      duck
                  None
```

实现ADT Map

- ❖ 保存key的列表就作为**散列表**来处理，这样可以迅速查找到指定的key
- ❖ 注意散列表的大小，虽然可以是任意数，但考虑到要让冲突解决算法能有效工作，应该选择为素数。

```
class HashTable:  
    def __init__(self):  
        self.size = 11  
        self.slots = [None] * self.size  
        self.data = [None] * self.size
```

实现ADT Map: put方法代码

- ❖ hashfunction方法采用了简单**求余**方法来实现散列函数，而冲突解决则采用**线性探测**“加1”再散列函数。

key不存在，未冲突

key已存在，替换val

散列冲突，再散列，
直到找到空槽或者key

```
def hashfunction(self, key):  
    return key% self.size
```

```
def rehash(self, oldhash):  
    return (oldhash+ 1)% self.size
```

```
def put(self, key, data):  
    hashvalue = self.hashfunction(key)
```

```
    if self.slots[hashvalue] == None:  
        self.slots[hashvalue] = key  
        self.data[hashvalue] = data
```

```
    else:
```

```
        if self.slots[hashvalue] == key:  
            self.data[hashvalue] = data #replace
```

```
        else:
```

```
            nextslot = self.rehash(hashvalue)
```

```
            while self.slots[nextslot] != None and \  
                  self.slots[nextslot] != key:
```

```
                nextslot = self.rehash(nextslot)
```

```
    if self.slots[nextslot] == None:  
        self.slots[nextslot]=key  
        self.data[nextslot]=data
```

```
    else:
```

```
        self.data[nextslot] = data #replace
```

实现ADT Map: get方法

标记散列值为
查找起点

```
def get(self, key):  
    startslot = self.hashfunction(key)
```

找key,
直到空槽或回到起点

```
    data = None  
    stop = False  
    found = False  
    position = startslot  
    while self.slots[position] != None and \  
           not found and not stop:
```

未找到key,
再散列继续找

```
        if self.slots[position] == key:  
            found = True  
            data = self.data[position]  
        else:  
            position = self.rehash(position)  
            if position == startslot:  
                stop = True
```

```
    return data
```

回到起点, 停

实现ADT Map：附加代码

❖ 通过特殊方法实现[]访问

```
def __getitem__(self, key):  
    return self.get(key)  
  
def __setitem__(self, key, data):  
    self.put(key, data)
```

散列算法分析

❖ 散列在最好的情况下，可以提供 $O(1)$ 常数级时间复杂度的查找性能

由于散列冲突的存在，查找比较次数就没有这么简单

❖ 评估散列冲突的最重要信息就是负载因子 λ ，一般来说：

如果 λ 较小，散列冲突的几率就小，数据项通常会保存在其所属的散列槽中

如果 λ 较大，意味着散列表填充较满，冲突会越来越多，冲突解决也越复杂，也就需要更多的比较来找到空槽；如果采用数据链的话，意味着每条链上的数据项增多

散列算法分析

❖ 如果采用线性探测的开放定址法来解决冲突 (λ 在0~1之间)

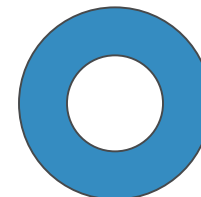
成功的查找, 平均需要比对次数为: $\frac{1}{2} \left(1 + \frac{1}{1-\lambda} \right)$

不成功的查找, 平均比对次数为: $\frac{1}{2} \left(1 + \left(\frac{1}{1-\lambda} \right)^2 \right)$

❖ 如果采用数据链来解决冲突 (λ 可大于1)

成功的查找, 平均需要比对次数为: $1 + \lambda/2$

不成功的查找, 平均比对次数为: λ





数据结构与算法 (Python版)

排序与查找小结

陈斌 北京大学 gischen@pku.edu.cn

本章小结

- ❖ 在无序表或者有序表上的顺序查找，其时间复杂度为 $O(n)$
- ❖ 在有序表上进行二分查找，其最差复杂度为 $O(\log n)$
- ❖ 散列表可以实现常数级时间的查找
- ❖ 完美散列函数作为数据一致性校验，应用很广
- ❖ 区块链技术是一种去中心化的分布式数据库，通过“工作量证明”机制来维持运行

本章小结

- ❖ 冒泡、选择和插入排序是 $O(n^2)$ 的算法
- ❖ 谢尔排序在插入排序的基础上进行了改进，采用对递增子表排序的方法，其时间复杂度可以在 $O(n)$ 和 $O(n^2)$ 之间
- ❖ 归并排序的时间复杂度是 $O(n \log n)$ ，但归并的过程需要额外存储空间
- ❖ 快速排序最好的时间复杂度是 $O(n \log n)$ ，也不需要额外的存储空间，但如果分裂点偏离列表中心的话，最坏情况下会退化到 $O(n^2)$

算法的选择：一封同学来信

❖ 冒泡排序算法中提到一个对其改进的算法

利用“如果一趟比对未产生任何交换，则已排好序”的“短路”特性，消除无效比对，理应更快

```
1  def BubbleSort(alist):
2      passnum = len(alist) - 1
3      while passnum > 0:
4          for i in range(passnum):
5              if alist[i] > alist[i + 1]:
6                  alist[i],alist[i + 1] = alist[i + 1],alist[i]
7          passnum -= 1
8      return alist
```

```
10 def ShortBubbleSort(alist):
11     → exchanges = True
12     passnum = len(alist) - 1
13     while passnum > 0 and exchanges: ←
14     →     exchanges = False
15         for i in range(passnum):
16             if alist[i] > alist[i + 1]:
17     →                 exchanges = True
18                 alist[i],alist[i + 1] = alist[i + 1],alist[i]
19         passnum -= 1
20     return alist
```

算法的选择：一封同学来信

❖ 但有细心的同学自己动手做了试验，结果不太乐观：

我试了一下排序算法

发现shortBubbleSort比BubbleSort慢很多啊

（每组数据运行10次取平均）

算法就是照着幻灯片上写的，为什么呢？

算法的选择：一封同学来信

❖ 第一次数据：

`alist = [i for i in range(10) for j in range(10)]`（再用shuffle打乱）

`{'Sort': 0.0, 'shuffle': 0.0, 'SelectionSort': 0.0, 'InsertionSort': 0.0, 'BubbleSort': 0.0,
'ShortBubbleSort': 0.0016000032424926757, 'ShellSort': 0.0}`

❖ 第二次数据：

`alist = [i for i in range(100) for j in range(100)]`（再用shuffle打乱）

`{'Sort': 0.0, 'shuffle': 0.0, 'SelectionSort': 2.8600001335144043, 'InsertionSort':
3.8279998302459717, 'BubbleSort': 7.0, 'ShortBubbleSort': 7.921999931335449,
0.04700016975402832}`

❖ 第三次数据：

`alist = [i for i in range(1000) for j in range(100)]`（再用shuffle打乱）

`{'Sort': 0.031000137329101562, 'shuffle': 0.031000137329101562, 'SelectionSort':
297.72199988365173, 'InsertionSort': 397.25999999046326, 'BubbleSort': 765.3139998912811,
'ShortBubbleSort': 847.1009998321533, 'ShellSort': 0.6410000324249268}`

（运行了一天）

算法的选择：一封同学来信

❖ 为什么理论和实际相差甚远？

到底还能不能愉快地选择一个好算法了？

❖ ShortBubbleSort的“短路”优势高度依赖于数据的初始布局

如果数据布局的随机度很高，造成每趟比对都会发生交换的话，ShortBubbleSort就完全没有优势，还要额外付出一个exchanges变量和相应赋值语句的代价，反倒比原始的冒泡排序要慢。

算法的选择：一封同学来信

❖ 同学的测试代码没有问题，问题在于排序对象，其数据是经过`random.shuffle`来乱序的

首先，`alist`是用`range`嵌套生成，生成的数据从小到大排列，非常整齐；

而这个`shuffle`会尽量把数据打乱到最混乱的程度，造成数据布局随机度很高；

这样，`ShortBubbleSort`的短路特性就完全失效，还要付出`exchanges`变量的判断、赋值代价，实测比原始冒泡排序算法要慢不少。

算法的选择：一封同学来信

- ❖ 所以排序算法有时候并不存在绝对的优劣，尤其是时间复杂度相同的算法们
- ❖ 要在特定的应用场合取得最高排序性能的话，还需要对**数据本身**进行分析，针对数据的特性来选择相应排序算法

算法的选择：一封同学来信

❖ 另外，除了时间复杂度，有时候空间复杂度也是需要考虑的关键因素

归并排序时间复杂度 $O(n \log n)$ ，但需要额外一倍的存储空间

快速排序时间复杂度最好的情况是 $O(n \log n)$ ，而且不需要额外存储空间，但“中值”的选择又成为性能的关键，选择不好的话，极端情况性能甚至低于冒泡排序

❖ 算法选择不是一个绝对的优劣判断，需要综合考虑各方面的因素

包括运行环境要求、处理数据对象的特性

