

数据结构与算法 (Python)

散列的应用案例：Git

谢正茂 webg@PKU-Mail

计算机学院数据所

April 15, 2025

目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序



- 了解和实现顺序查找和二分法查找；
- 了解和实现选择排序、冒泡排序、归并排序、快速排序、插入排序和希尔排序；
- 了解用散列 Hashing 实现查找的技术；
- 了解抽象数据类型：映射 Map；
- 采用散列实现抽象数据类型 Map。

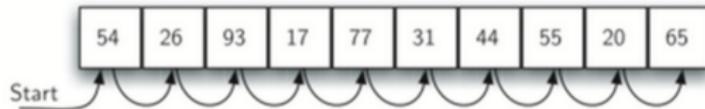
目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序



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，这种技术就称为“顺序查找”
- 要确定列表中是否存在需要查找的数据项，首先从列表的第一个数据项开始，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))
```

```
1 def sequentialSearch(alist, item):
2     for elem in alist:
3         if elem == item:
4             return True
5     return False
```

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
- 在顺序查找算法中，为了保证是一般情形，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
- 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，则比对数是 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

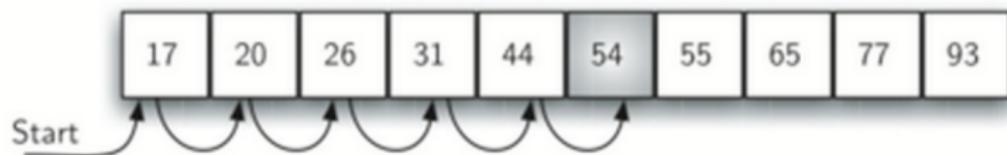
- 数据项在列表中，比对的一般情形如何？
 - 因为数据项在列表中各个位置出现的概率是相同的；
 - 所以平均状况下，比对的次数是 $n/2$ ；
- 所以，顺序查找的算法复杂度是 $O(n)$

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，比对过程与无序表基本相同
 - 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（两种风格代码）

```
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

```
1 #顺序查找有序表
2 def orderedSequentialSearch(alist, item):
3     for elem in alist:
4         if elem == item:
5             return True
6         elif elem > item:
7             return False
8     else:
9         return False
```

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。

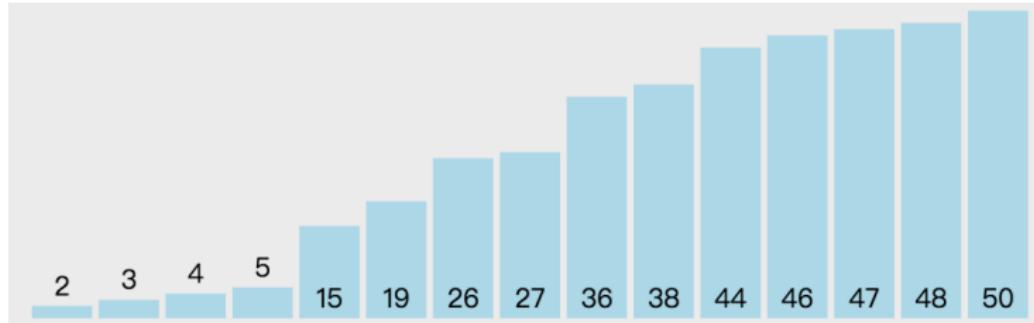
目录

- 本章目标
- 顺序查找
- **二分查找**
- 散列
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序



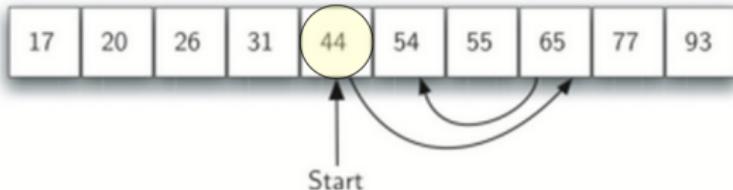
二分查找

- 那么对于有序表，有没有更好的查找算法？
- 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

```
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))
```

中间项比对

缩小比对范围

二分查找：分而治之 (Divide & Conquer)

- 二分查找算法实际上体现了解决问题的另一个典型策略：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
- 显然，递归算法就是一种典型的分而治之策略，二分法也可以用递归算法来实现
- ****slice** 操作的复杂度是 $O(N)$ 的，放在 $O(\log N)$ 的二分查找算法里面，整体复杂度也变成 $O(N)$ 的。实际使用中严重不可取！

The diagram illustrates the Divide & Conquer strategy for binary search. It features a blue arrow pointing right labeled "基本结束条件" (Basic Termination Condition) pointing to the base case of the recursive function. Another blue arrow pointing right labeled "缩小规模" (Reduce Scale) points to the recursive calls where the search space is halved. A blue speech bubble labeled "调用自身" (Call itself) points to the recursive call within the function body.

```
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)
```

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右表所示

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：

$$\frac{n}{2^i} = 1 \Rightarrow i = \log_2(n)$$

- 所以二分法查找的算法复杂度是 $O(\log n)$

主定理

- 可分解为子问题的问题复杂度
- 对于二分查找问题: $a=1, b=2, f(n)=O(1)$

主定理: 设 $a \geq 1, b > 1$ 为常数, $f(n)$ 为函数, $T(n)$ 为非负整数, 且

$$T(n) = aT(n/b) + f(n)$$

则有以下结果:

- 若 $f(n) = O(n^{\log_b a - \varepsilon}), \varepsilon > 0$, 那么 $T(n) = \Theta(n^{\log_b a})$
- 若 $f(n) = \Theta(n^{\log_b a})$, 那么 $T(n) = \Theta(n^{\log_b a} \log n)$
- 若 $f(n) = \Omega(n^{\log_b a + \varepsilon}), \varepsilon > 0$, 且对于某个常数 $c < 1$ 和充分大的 n 有 $a f(n/b) \leq c f(n)$, 那么 $T(n) = \Theta(f(n))$

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 $O(\log n)$
- 但要注意到本算法中除了比对，还有一个因素需要注意：
 - `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。
- “让发生的事情最高效”

目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序



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”
- 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？

身边的“数据结构”



- 快速“放/取”咖啡
- 顺序分配“取餐号”
- 层号为“取餐号”个位
- 空间冗余：10/12 层
- 散列表
- 顺序表/有序表

散列：基本概念

- 散列表（hash table，又称哈希表）是一种数据集，其中数据项的存储位置通过固定的映射函数获得，在查找定位时也可利用此函数获得 $O(1)$ 的访问效率。散列表中的每一个可用于保存数据项的存储位置，称为槽（slot），每个槽用不同的编号来区别。
- 例如：一个包含 11 个槽的散列表，槽的编号分别为 0~10
- 在插入数据项之前，每个槽的值都是 None，表示空槽
- 实现从数据项到存储槽的转换的，称为散列函数（hash function）
- 下面例子中，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

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

- 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
 - 如果数据项不是整数怎么办？比如身份证号的最后一位有可能是‘X’（非数项）
 - 字符和整数之间存在转换关系

散列：示例

- 本例中我们的散列函数是最简单的求余：
- $h(item) = item \% 11$

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

散列：示例

- 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
- 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，查找就变得无比简单
- 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进行计算，用得到的槽号看对应的槽中是否是要我们要找的
 - 实现了 $O(1)$ 时间复杂度的查找算法。
- 天底下没有免费的午餐，想想我们的代价是什么？

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，各自占据了不同的槽
- 假如再来一个数 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 (碰撞)”，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

44

完美散列函数：Perfect Hash Function

- 散列函数是一种离散函数
 - 定义域的大小 N ;
 - 值域的大小 M , 也等于散列表的容量;
 - 函数实际的输入个数 k , 也等于散列表中存储的不同元素个数。
- 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 ($k \leq M$), 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问题, 我们后面会有办法处理的。

完美散列函数：Perfect Hash Function

- 获得完美散列函数的一种方法是扩大散列表的容量，大到等于数据项的值空间大小；用恒等函数来做散列函数，这样不同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！对于仅需要保存班级 100 名同学的手机号来说，浪费了太多空间
 - 退而求其次的话，好的散列函数需要具备 3 个特性：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）

- 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，可以把散列值用作数据的“指纹”或者“摘要”
 - 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。
 - 离散函数 $y = \text{hash}(x)$ ，如果其 N 大于 M ，根据鸽笼原理，肯定存在冲突： $x_1 \neq x_2, \text{hash}(x_1) = \text{hash}(x_2)$
 - 但如果它的定义域和值域空间都非常“广阔”，实际取的 x 个数远远小于它们，那么冲突的可能性就“没有了”。
 - 尽管 $N > M$ ，只要 $k \ll M$ ，在计算机领域广泛当成“完美散列函数”来用。

- 作为一致性校验的数据“指纹”函数还需要具备更多的特性
 - 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；而从指纹计算原数据实际上是不可能的；
 - 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：
 - 弱抗冲突性：给定 Hashing 函数和一个数据项，很难找到另一个数据项，和原数据项具有相同的指纹
 - 强抗冲突性：给定 Hashing 函数，很难找到两个不同数据项，使它们具有相同的指纹
- 讨论 Python 内置的哈希函数：`hash()`
 - 是不是近似完美的？
 - 满足上面的特性吗？

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
- 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^{48} ，地球上水分子数约为 10^{47}
 - 256 位二进制相当于 10^{77} ，已知宇宙所有基本粒子大约为 $10^{72\sim87}$
- 小明和小芳各从宇宙中挑一个原子，他们会不会挑到同一个原子？
- 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。
 - 王小云：提出了密码哈希函数的碰撞攻击理论，即模差分比特分析法，破解了包括 MD5、SHA-1 在内的 5 个国际通用哈希函数算法

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

```
1 import hashlib
2 print(hashlib.md5(b"hello world!").hexdigest())
3 print(hashlib.sha1(b"hello world!").hexdigest())
4 print(hashlib.sha256(b"hello world!").hexdigest()) # 比特币挖矿算法中出现
```

```
fc3ff98e8c6a0d3087d515c0473f8677
430ce34d020724ed75a196dfc2ad67c77772d169
7509e5bda0c762d2bac7f90d758b5b2263fa01ccbc542ab5e3df163be08e6ca9
```

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。

```
1 import hashlib
2 m = hashlib.md5()
3 m.update(b"hello world!")
4 m.update(b>this is part #2")
5 m.update(b>this is part #3")
6 print(m.hexdigest())
7 print(hashlib.md5((b"hello world!"
8                      b>this is part #2"
9                      b>this is part #3")).hexdigest()) # 字符串跨行语法
```

a12edc8332947a3e02e5668c6484b93a

a12edc8332947a3e02e5668c6484b93a

散列函数 MD5/SHA 系列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。

● 加密形式保存密码

- 仅保存密码的散列值，用户输入密码后，重新计算该值与原值比较；
- 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。
- $\text{SHA_512}(\text{*origin_password*} + \text{随机常数}) = \text{HASH_VALUE}$

```
[root@yongFeng etc]# tail /etc/shadow | grep `\$`  
stu2000093012:$6$P1FRB2NBdpz63b0$odgf2o5n4n0VFZw1Hp4JNfBq.k5fYdcUzpcPYApM4s3CDhWzHCZhMq50DGB7oBkI5tU2mm2KwPDjR584lt00hD:/18701:0:99999:7:::  
stu2000093013:$6$bx5CoGjVKRLGYuxr$ue0hnTxhSbSG6MeMoAW4JE5vnnSm$mpBz9pjk2LkLp1tKp0vvuk5xlpw11Br0VkmDb7wN0rNrQnWM:/18701:0:99999:7:::  
stu2000093018:$6$So5pQkzWkIjCz0$WRD6M11WQkJZerMSX4TGxvgW..54cbuUty4oz9A/J1Txfm10hQX29smVl384VNeQx0wwengSp9JLsz.H3AwUB0:18701:0:99999:7:::  
stu2000093022:$6$j-rGfnVtjnB1FYoov$jmzQOB.iy8DJWTTvcce8391kel0FPt7xbj9mfCbXq2IpoinWPFP6ewONE71F/LAkveFgSuuRXTiqbMS5U8Z.J:/18701:0:99999:7:::  
stu78035:$6$pom9V/vyVRruhy7$EVrreJ1xR3mg19GF5mDB..Ec1R/Fj21SzPqQ.BbF0jjem2GAly5umt9Poq2kAnuggFgFSRxFs5t9EvAxPyL11X1:18701:0:99999:7:::  
stu1900015953:$6$BlkykYT0TB1/WsVe$AG4u6e/..r/CASBQZBEcpgZXAHiycl5jx/WoVRKBPbsWP6nsM0LoPIY/0CLMfr1x1ak9vCSHxyhxw0L2dxhCab70:18705:0:99999:7:::  
tansc:$6$HnA5snoD3ZYJLswUS$uxpi1rCThf1h20RGoEymGH8L2/8MuSFdxHt/Gcrd0zE5Y5xPKZBpRQQUf1RpSvn6FgvwvR2W0fgFgb1f1:18706:0:99999:7:::  
lisui:$6$EVZYytMEfKwkg/$0Iypj0QhwQYeTfjoh4UVJfmR2IA3BGZekqlW07Db84kVm..34sYG1mKRV99YzgV4sQZPozKg1UmFGsx7sT0:18706:0:99999:7:::  
liys:$6$NZ3beKLDS5LMC0..B$c1RlxgLlookBu9X..aznr.ORTJglvlMsJhp4borBzV0ynKgbEeq1G3DftrzAOEbw0A2XgIkH3953qsp73A1:18708:0:99999:7:::  
stu1600094605:$6$bVlhQ6ochI0jwqgDW$63sUr61..yQBp7vz8m50zYpZkVc75c81n.thzJD3AwxNp0nQpoch5NBkrC/fjoL9PE1llNShesXzNTqb15y:18710:0:99999:7:::
```

散列函数设计：折叠法 Folding Method

- 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 $h(62767255)=1$
- 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，以便更好符合上述的 3 个特性。(少冲突/易计算/分散)

散列函数设计：平方取中法 Mid-Square Method

- 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 $44 \times 44 = 1936$ ，然后取中间的 93，对散列表大小 11 求余， $93 \% 11 = 5$

- 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

成熟的 Hash 算法之一：MD5

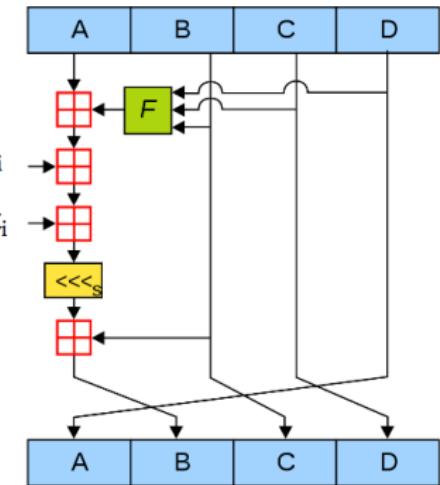
- 进行了 4 轮计算，繁琐但计算机算起来很快
- $\oplus, \wedge, \vee, \neg$ denote the XOR, AND, OR and NOT operations respectively.

$$F(B, C, D) = (B \wedge C) \vee (\neg B \wedge D)$$

$$G(B, C, D) = (B \wedge D) \vee (C \wedge \neg D)$$

$$H(B, C, D) = B \oplus C \oplus D$$

$$I(B, C, D) = C \oplus (B \vee \neg D)$$



散列函数设计：非数项

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

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

    return sum%tablesize
```

c a t
↓ ↓ ↓
99 + 97 + 116 = 312
312 % 11 —————→ 4

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
- 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

$$\begin{array}{ccccccc} & & \text{position} & & & & \\ & 1 & & 2 & & 3 & \\ c & \downarrow & a & \downarrow & t & \downarrow & \\ 99 * 1 + & 97 * 2 + & 116 * 3 & = & 641 & & \\ & & & & & 641 \% 11 & \longrightarrow 3 \end{array}$$

目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
 - 冲突的解决
 - 开地址法
 - 拉链法
 - 抽象数据类型“映射”
 - 散列的应用案例：Git
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序

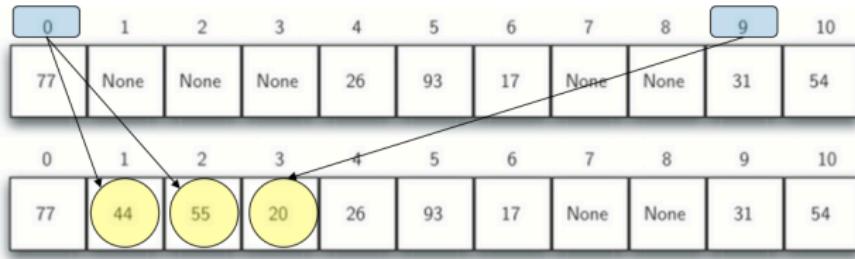


冲突解决方案（即使不完美，日子还得继续过）

- 如果两个数据项被散列映射到同一个槽，需要有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，所以解决散列冲突成为散列方法中很重要的一部分。
- 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，其中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。（如果到散列表的尾部还未找到，则要从首部接着扫描）
- 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**”

冲突解决方案：线性探测 Linear Probing

- 接前例，我们把 44、55、20 逐个插入到散列表中。
 - $h(44) == 0$, 发现 0# 槽已经被 77 占据了，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
- 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）。



冲突解决方案：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

- 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

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

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，也就是说 $\text{newhashvalue} = \text{rehash}(\text{oldhashvalue})$
 - 对于线性探测来说， $\text{rehash}(\text{pos}) = (\text{pos} + 1) \% \text{sizeoftable}$
 - “+3”的跳跃式探测则是： $\text{rehash}(\text{pos}) = (\text{pos} + 3) \% \text{sizeoftable}$
 - 跳跃式探测的再散列通式是： $\text{rehash}(\text{pos}) = (\text{pos} + \text{skip}) \% \text{sizeoftable}$
- 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

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： $h, h+1, h+4, h+9, h+16\dots$
- `skip` 的值也可以通过 `key` 计算得来： $skip=h2(key)$ ，这种方法被称为双散列函数探查法。

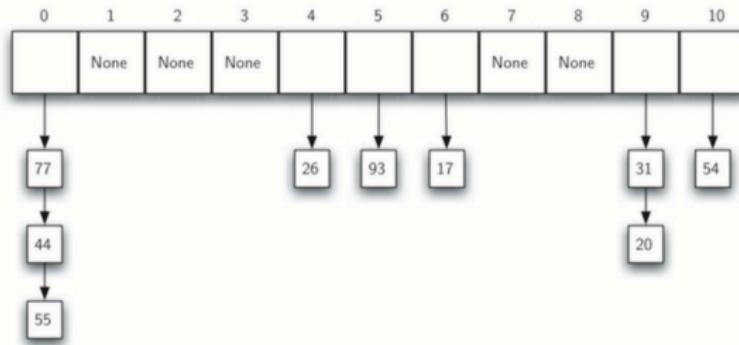
目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
 - 冲突的解决
 - 开地址法
 - **拉链法**
 - 抽象数据类型“映射”
 - 散列的应用案例：Git
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序



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。
- 与“开放地址”相比较，“数据项链”避免了不同槽之间的冲突可能，但是为存储指针增加了额外的空间开销。



- 本章目标
- 顺序查找
- 二分查找
- 散列
 - 冲突的解决
 - 抽象数据类型“映射”
 - 散列的应用案例：Git
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序



抽象数据类型“映射”:ADT Map

- Python 最有用的内置数据类型之一是“字典 Dictionary”
- 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通常称为“映射 map”
- 通过散列表实现 map，是一种常用的高效方案

抽象数据类型“映射”:ADT 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
- 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` 已经存在，则将 `val` 替换原来的旧关联值；
 - `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

- 仿照 Python dict，我们自己去实现一个 Map ADT
- 大多数程序设计语言已经提供了这些功能，为什么我们还要再去实现一遍？
 - 了解内部实现加深理解
 - Map 在计算机编程中的应用极为广泛
 - 常见的面试题：HashMap 和 TreeMap 的不同？
- 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
 - 在查找到一个 `key` 以后，在 `data` 列表对应相同位置的数据项即为关联数据
 - 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。

```
1 class HashTable:  
2     def __init__(self, size=11):  
3         self.size = size  
4         self.slots = [None] * self.size  
5         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，冲突解决则采用线性探测“加 1”再散列函数。

```
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

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

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

The diagram consists of three blue arrows pointing from text boxes to specific parts of the Python code. The top arrow points to the first if statement in the put method. The middle arrow points to the nested if statement within the else block. The bottom arrow points to the while loop in the rehash block.

- key不存在, 未冲突 → 第一个 if self.slots[hashvalue] == None: 行
- key已存在, 替换val → 第二个 if self.slots[hashvalue] == key: 行
- 散列冲突, 再散列, 直到找到空槽或者key → while self.slots[nextslot] != None and \ self.slots[nextslot] != key: 行

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

```
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])
print(H[17])
H[20]='duck'
print(H[20])
print(H[99])

def __getitem__(self, key):
    return self.get(key)

def __setitem__(self, key, data):
    self.put(key, data)

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

- 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
- 但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 和 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：

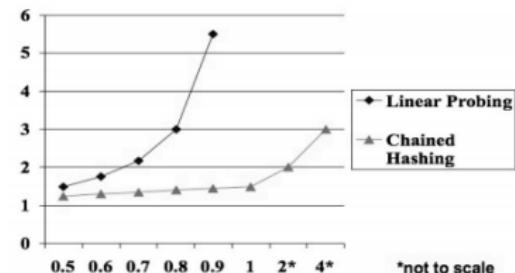
$$\frac{1}{2} \left(1 + \frac{1}{1 - \lambda} \right)^2$$

- 如果采用数据链来解决冲突 (λ 可以大于 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



$\lambda \approx 0.5$ 为安全值： $O(1)$

随堂思考

对于大小为13的散列表，27、130的散列值分别是多少？*

- 1 , 10
- 13 , 0
- 1 , 0
- 2 , 3

对于大小为11的散列表，如果插入如下数据项，并采用线性探测解决散列冲突，最后的布局是什么？*

113 , 117 , 97 , 100 , 114 , 108 , 116 , 105 , 99

- 100, __, __, 113, 114, 105, 116, 117, 97, 108, 99
- 99, 100, __, 113, 114, __, 116, 117, 105, 97, 108
- 100, 113, 117, 97, 14, 108, 116, 105, 99, __, __
- 117, 114, 108, 116, 105, 99, __, __, 97, 100, 113

目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
 - 冲突的解决
 - 抽象数据类型“映射”
 - 散列的应用案例：Git
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序

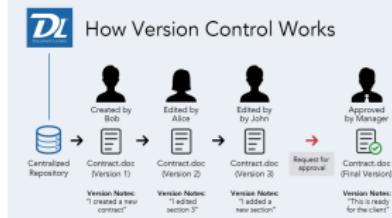


Git：一个分布式的版本管理和协作工具

- 散列的应用远比我们已知的要广
- 用理论知识去观察一个真实的软件系统
- Bitcoin vs. Git，很多共性
 - 都大量用了散列技术；都是“去中心化”的
 - 但 Bitcoin 离我们太“远”
- 版本管理需要处理时间和空间两个维度的问题，加上协作使问题更为复杂
 - <http://github.com/git/git>, 十万行源代码
- 从数据结构的角度去了解一个软件系统
 - "Bad programmers worry about the code. Good programmers worry about data structures and their relationships."
 - 希望用几张 slides 就能让同学们了解一个大型软件的工作原理

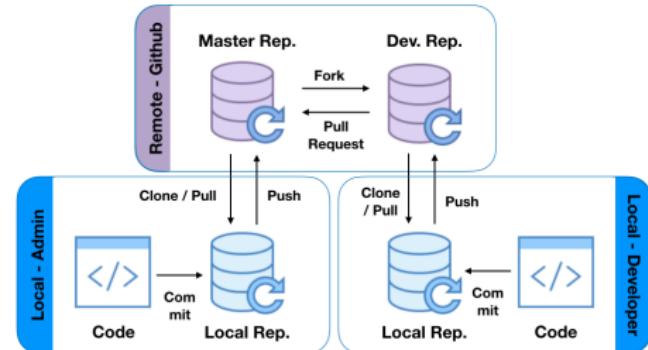


Linus Torvalds in 2018



一个 Git 仓库的组织结构

- 去中心化的结构
- 多个结点并不比单个结点有：
 - 更大的存储能力
 - 更强的计算能力
- 仓库之间就是“复制”关系
 - 强大的“分支-合并”能力
 - 高效；高并发能力
 - 离线工作能力
- 把 dsa2020 仓库在本地复制一份



```
$ git clone http://github.com/dsaClass/dsa2020
```

```
zhengmaoxie@m1book dsa2020 % ls -a
.
..
```

```
.git          README.md
.gitignore    course
```

```
openjudge
```

- 工作目录 Working directory
- 真正意义上的仓库 .git

```
zhengmaoxie@m1book dsa2020 % ls .git
COMMIT_EDITMSG  ORIG_HEAD      hooks           logs          refs
FETCH_HEAD      config        index           objects
HEAD           description   _info          packed-refs
```

Object database

Git 中的所有数据（包括文件、目录、提交和标签）都被存储为由哈希值标识的不可变压缩对象。

● 一个同学的解题文档

```
>>> print(md)

## 基本信息

#:48623036
题名:258307
提交人:斬歌2400012283
内存:4532kB
时间:32ns
语言:Python3
提交时间:2025-03-18 22:59:04

## 解释器路
```

对于一个字符串需要几次，可以分割成小问题。比如从 i 到 j 需要几次，用动态规划的状态转移。

对于 d_B , 有从1到1的次数为0

遍历由2到n长度的子串，再遍历所有可能的起始点和终止点
如果起始=终止，那么操作次数就等于当前位置向里缩进的状态

1

- 经过如下计算：

```
mdutf8=md.encode('utf-8')
data='blob {}\{}\'.format(
    len(mdutf8), md
).encode('utf-8')
hashlib.sha1(data).hexdigest()

>>> hashlib.sha1(data).hexdigest()
'0df36a21b6a6762aabcbc6934a47d233d37760ce0c'
```

- git 的 hash-object 子命令

```
[zhengmaoxie@m1book dsa2020 % git hash-object sbsmc  
8df26c21b4a6742aabb483e47d223d37769cc0c
```

- 用 '0df36' 去 git 中找这个文件

● Git 把文件放在哪儿了？

```
[zhengmaxiao@zhengmaxiao-macbook dsas2028 % ls .git/objects
```

sha1	size	type
00 18	4d	68
01 19	33	4e
02 1a	35	4e
03 1b	56	6b
04 1c	37	53
05 1d	39	53
06 1e	3a	54
07 1f	2b	55
08 20	56	6f
09 21	3d	57
0a 22	3e	59
0b 23	3d	57
0c 24	3e	59
0d 25	68	89
0e 26	69	82
0f 27	68	83
10 28	66	86
11 29	53	86
12 2a	53	87
13 2b	54	88
14 2c	54	89
15 2d	54	8a
16 2e	54	8b
17 2f	54	8c
18 20	71	8b
19 21	72	8c
20 22	72	8e
21 23	72	8f
22 24	72	b3
23 25	72	b4
24 26	72	b5
25 27	72	b6
26 28	72	b7
27 29	72	b8
28 2a	72	b9
29 2b	72	b0
2a 2c	72	b1
2b 2d	72	b2
2c 2e	72	b3
2d 2f	72	b4
2e 20	72	b5
2f 21	72	b6
30 22	72	b7
31 23	72	b8
32 24	72	b9
33 25	72	b0
34 26	72	b1
35 27	72	b2
36 28	72	b3
37 29	72	b4
38 2a	72	b5
39 2b	72	b6
40 2c	72	b7
41 2d	72	b8
42 2e	72	b9
43 2f	72	b0
44 20	72	b1
45 21	72	b2
46 22	72	b3
47 23	72	b4
48 24	72	b5
49 25	72	b6
50 26	72	b7
51 27	72	b8
52 28	72	b9
53 29	72	b0
54 2a	72	b1
55 2b	72	b2
56 2c	72	b3
57 2d	72	b4
58 2e	72	b5
59 2f	72	b6
60 20	72	b7
61 21	72	b8
62 22	72	b9
63 23	72	b0
64 24	72	b1
65 25	72	b2
66 26	72	b3
67 27	72	b4
68 28	72	b5
69 29	72	b6
70 2a	72	b7
71 2b	72	b8
72 2c	72	b9
73 2d	72	b0
74 2e	72	b1
75 2f	72	b2
76 20	72	b3
77 21	72	b4
78 22	72	b5
79 23	72	b6
80 24	72	b7
81 25	72	b8
82 26	72	b9
83 27	72	b0
84 28	72	b1
85 29	72	b2
86 2a	72	b3
87 2b	72	b4
88 2c	72	b5
89 2d	72	b6
90 2e	72	b7
91 2f	72	b8
92 20	72	b9
93 21	72	b0
94 22	72	b1
95 23	72	b2
96 24	72	b3
97 25	72	b4
98 26	72	b5
99 27	72	b6
100 28	72	b7
101 29	72	b8
102 2a	72	b9
103 2b	72	b0
104 2c	72	b1
105 2d	72	b2
106 2e	72	b3
107 2f	72	b4
108 20	72	b5
109 21	72	b6
110 22	72	b7
111 23	72	b8
112 24	72	b9
113 25	72	b0
114 26	72	b1
115 27	72	b2
116 28	72	b3
117 29	72	b4
118 2a	72	b5
119 2b	72	b6
120 2c	72	b7
121 2d	72	b8
122 2e	72	b9
123 2f	72	b0
124 20	72	b1
125 21	72	b2
126 22	72	b3
127 23	72	b4
128 24	72	b5
129 25	72	b6
130 26	72	b7
131 27	72	b8
132 28	72	b9
133 29	72	b0
134 2a	72	b1
135 2b	72	b2
136 2c	72	b3
137 2d	72	b4
138 2e	72	b5
139 2f	72	b6
140 20	72	b7
141 21	72	b8
142 22	72	b9
143 23	72	b0
144 24	72	b1
145 25	72	b2
146 26	72	b3
147 27	72	b4
148 28	72	b5
149 29	72	b6
150 2a	72	b7
151 2b	72	b8
152 2c	72	b9
153 2d	72	b0
154 2e	72	b1
155 2f	72	b2
156 20	72	b3
157 21	72	b4
158 22	72	b5
159 23	72	b6
160 24	72	b7
161 25	72	b8
162 26	72	b9
163 27	72	b0
164 28	72	b1
165 29	72	b2
166 2a	72	b3
167 2b	72	b4
168 2c	72	b5
169 2d	72	b6
170 2e	72	b7
171 2f	72	b8
172 20	72	b9
173 21	72	b0
174 22	72	b1
175 23	72	b2
176 24	72	b3
177 25	72	b4
178 26	72	b5
179 27	72	b6
180 28	72	b7
181 29	72	b8
182 2a	72	b9
183 2b	72	b0
184 2c	72	b1
185 2d	72	b2
186 2e	72	b3
187 2f	72	b4
188 20	72	b5
189 21	72	b6
190 22	72	b7
191 23	72	b8
192 24	72	b9
193 25	72	b0
194 26	72	b1
195 27	72	b2
196 28	72	b3
197 29	72	b4
198 2a	72	b5
199 2b	72	b6
200 2c	72	b7
201 2d	72	b8
202 2e	72	b9
203 2f	72	b0
204 20	72	b1
205 21	72	b2
206 22	72	b3
207 23	72	b4
208 24	72	b5
209 25	72	b6
210 26	72	b7
211 27	72	b8
212 28	72	b9
213 29	72	b0
214 2a	72	b1
215 2b	72	b2
216 2c	72	b3
217 2d	72	b4
218 2e	72	b5
219 2f	72	b6
220 20	72	b7
221 21	72	b8
222 22	72	b9
223 23	72	b0
224 24	72	b1
225 25	72	b2
226 26	72	b3
227 27	72	b4
228 28	72	b5
229 29	72	b6
230 2a	72	b7
231 2b	72	b8
232 2c	72	b9
233 2d	72	b0
234 2e	72	b1
235 2f	72	b2
236 20	72	b3
237 21	72	b4
238 22	72	b5
239 23	72	b6
240 24	72	b7
241 25	72	b8
242 26	72	b9
243 27	72	b0
244 28	72	b1
245 29	72	b2
246 2a	72	b3
247 2b	72	b4
248 2c	72	b5
249 2d	72	b6
250 2e	72	b7
251 2f	72	b8
252 20	72	b9
253 21	72	b0
254 22	72	b1
255 23	72	b2
256 24	72	b3
257 25	72	b4
258 26	72	b5
259 27	72	b6
260 28	72	b7
261 29	72	b8
262 2a	72	b9
263 2b	72	b0
264 2c	72	b1
265 2d	72	b2
266 2e	72	b3
267 2f	72	b4
268 20	72	b5
269 21	72	b6
270 22	72	b7
271 23	72	b8
272 24	72	b9
273 25	72	b0
274 26	72	b1
275 27	72	b2
276 28	72	b3
277 29	72	b4
278 2a	72	b5
279 2b	72	b6
280 2c	72	b7
281 2d	72	b8
282 2e	72	b9
283 2f	72	b0
284 20	72	b1
285 21	72	b2
286 22	72	b3
287 23	72	b4
288 24	72	b5
289 25	72	b6
290 26	72	b7
291 27	72	b8
292 28	72	b9
293 29	72	b0
294 2a	72	b1
295 2b	72	b2
296 2c	72	b3
297 2d	72	b4
298 2e	72	b5
299 2f	72	b6
300 20	72	b7
301 21	72	b8
302 22	72	b9
303 23	72	b0
304 24	72	b1
305 25	72	b2
306 26	72	b3
307 27	72	b4
308 28	72	b5
309 29	72	b6
310 2a	72	b7
311 2b	72	b8
312 2c	72	b9
313 2d	72	b0
314 2e	72	b1
315 2f	72	b2
316 20	72	b3
317 21	72	b4
318 22	72	b5
319 23	72	b6
320 24	72	b7
321 25	72	b8
322 26	72	b9
323 27	72	b0
324 28	72	b1
325 29	72	b2
326 2a	72	b3
327 2b	72	b4
328 2c	72	b5
329 2d	72	b6
330 2e	72	b7
331 2f	72	b8
332 20	72	b9
333 21	72	b0
334 22	72	b1
335 23	72	b2
336 24	72	b3
337 25	72	b4
338 26	72	b5
339 27	72	b6
340 28	72	b7
341 29	72	b8
342 2a	72	b9
343 2b	72	b0
344 2c	72	b1
345 2d	72	b2
346 2e	72	b3
347 2f	72	b4
348 20	72	b5
349 21	72	b6
350 22	72	b7
351 23	72	b8
352 24	72	b9
353 25	72	b0
354 26	72	b1
355 27	72	b2
356 28	72	b3
357 29	72	b4
358 2a	72	b5
359 2b	72	b6
360 2c	72	b7
361 2d	72	b8
362 2e	72	b9
363 2f	72	b0
364 20	72	b1
365 21	72	b2
366 22	72	b3
367 23	72	b4
368 24	72	b5
369 25	72	b6
370 26	72	b7
371 27	72	b8
372 28	72	b9
373 29	72	b0
374 2a	72	b1
375 2b	72	b2
376 2c	72	b3
377 2d	72	b4
378 2e	72	b5
379 2f	72	b6
380 20	72	b7
381 21	72	b8
382 22	72	b9
383 23	72	b0
384 24	72	b1
385 25	72	b2
386 26	72	b3
387 27	72	b4
388 28	72	b5
389 29	72	b6
390 2a	72	b7
391 2b	72	b8
392 2c	72	b9
393 2d	72	b0
394 2e	72	b1
395 2f	72	b2
396 20	72	b3
397 21	72	b4
398 22	72	b5
399 23	72	b6
400 24	72	b7
401 25	72	b8
402 26	72	b9
403 27	72	b0
404 28	72	b1
405 29	72	b2
406 2a	72	b3
407 2b	72	b4
408 2c	72	b5
409 2d	72	b6
410 2e	72	b7
411 2f	72	b8
412 20	72	b9
413 21	72	b0
414 22	72	b1
415 23	72	b2
416 24	72	b3
417 25	72	b4
418 26	72	b5
419 27	72	b6
420 28	72	b7
421 29	72	b8
422 2a	72	b9
423 2b	72	b0
424 2c	72	b1
425 2d	72	b2
426 2e	72	b3
427 2f	72	b4
428 20	72	b5
429 21	72	b6

- 在磁盘上用 zlib 压缩

```
import zlib
fn='f36a21b6a6762aab6034a47d233d37760ce0c
with open(fn, 'rb') as file:
    data = file.read()
    print(zlib.decompress(data)
        .decode('utf-8'))
```

Object database

- 前面的存储格式称为“松散”的
 - 文件名为内容的散列值
 - 必须是完美散列函数
 - 同时还具备抗修改性、抗冲突性
- 在此基础上为了减少文件数量
 - 会把多个 objects 打包存储
- 为了进一步节省磁盘空间

```
dsa2020 % git log -p README.md
```

- 增量保存近似文件
- 大文件的多个相邻版本
- 记录 Δ 来节省空间

```
zhengmaoxie@m1book dsa2020 % ls .git/objects/pack  
pack-d1b4afd83990604c30fb75448910ff4d3ea27c6e.idx  
pack-d1b4afd83990604c30fb75448910ff4d3ea27c6e.pack  
zhengmaoxie@m1book dsa2020 % git log -p README.md  
commit 8c323f73b25ee98f0563bb14efdb515b61224e2  
Author: m1book <zhengmaoxie@gmail.com>  
Date: Tue Apr 1 13:37:37 2025 +0800
```

随堂小测 2 地址

```
diff --git a/README.md b/README.md  
index b17309c..b182c39 100644  
--- a/README.md  
+++ b/README.md  
@@ -70,4 +70,5 @@ contributor下次开新的branch的时候，并不需要每次都 fork 主仓库  
- 2/25/2025 第三周 (3/4/2025) 上课时间随堂小测，考察 Python 基础技能。  
- 小测地址：http://xzmdsa.openjudge.cn/pythonbasic/  
- 3/25/2025 第七周 (4/1/2025) 上课时间随堂小测，考察线性表、递归与动规两章  
+ - 小测地址：http://xzmdsa.openjudge.cn/2025test2/
```

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e  
2431d67693845a04d72e26dd3bf7bd58fbcb1 commit 223 155 12  
69bcdaff5328728ab1c0812ce09ff7fa7d26969d7 commit 214 152 167  
80d02664cb23ed55b226516648c0e7d5d8a3deh99 commit 214 145 319  
43168a1b87613d1281e5560855a83eb8fde3d687 commit 213 146 464  
092917823486a082e94d727c20a99024e14e1fc2 commit 214 146 610  
782478739c72b05e2edff522fdfe85d5ba65df9f commit 165 118 756  
d368d0ac0678be6cc5e505be58126d3526706e54 tag 130 122 874  
fe879577cb8ccfcf25441725141e310d7d239f tree 136 136 996  
d8329fc1cc938780ff0d9f94e0d364e0ea74f579 tree 36 46 1132  
dee2e1b793907545e508a2e22dd5ba6c5bc4586 tree 136 136 1178  
d9827c8b2c2a97e391a85d481f1c79127a01d tree 6 17 1314 1 \\  
dee2e1b793907545e50a2e2dd5ba6c5bc4506  
3c49cd789d888d889c1073707c358541b0e614 tree 8 19 1331 1 \\  
dee2e1b793907545e50a2e2dd5ba6c5bc4506  
0155b6e4229851634a0ff03eb265b69f5a2d5f61341 tree 71 76 1350  
83baae618046e55c73a72b1a725275c760663a blob 18 19 1426  
fa49b677972391ad58037050f2a75f74e3671e92 blob 9 18 1445  
b042a60eef7dff760008df33ce372b945b6e884e blob 22854 5799 1463  
033b4468fa6b2a9547a70d8d1bbe8bf3f9ed0d5 blob 9 20 7262 1 \\  
b042a60eef7dff760008df33ce372b945b6e884e  
1f7a7a472ab73d9643fd15f6da379c4acb3e3a blob 10 19 7282  
non delta: 15 objects  
chain length = 1: 3 objects  
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Git 中的 commit

- commit 日志：历史版本信息

```
dsa2020 % git log
```

```
commit ea20c053ffa6506d9381a7741af36c15dfde59b0
Merge: 09657ed 241c4df
Author: mibook <zhengmaoxie@gmail.com>
Date:   Tue Mar 25 20:54:39 2025 +0800

    Merge branch 'cf7219'

P0930:复杂的整数划分问题，后继解答

commit 09657ed73ae482ba7149917e4f8153e1f1cf9dcf
Merge: e1abccca b217d20
Author: Patrickxzxm <zhengmaoxie@gmail.com>
Date:   Tue Mar 25 20:42:34 2025 +0800

    Merge pull request #560 from CHAI-bt/hw3

Hw3 250305:小木棍，后继解答
```

- parent 从后往前指针
- 最新的一个 commit

```
dsa2020 % git cat-file -p ea20c05
```

```
tree ea0ea7cf616b638dc510463443309017d9fef767
parent 09657ed73ae482ba7149917e4f8153e1f1cf9dcf
parent 241c4df5e46727cb15aa8144137045be18c7ebd7
author mibook <zhengmaoxie@gmail.com> 1742907279 +0800
committer mibook <zhengmaoxie@gmail.com> 1742907279 +0800

    Merge branch 'cf7219'

P0930:复杂的整数划分问题，后继解答
```

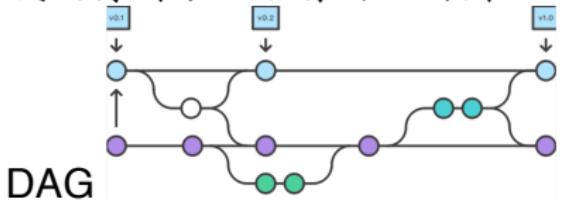
- 它的一个 parent

```
dsa2020 % git cat-file -p 241c4
```

```
zhengmaoxie@mibook dsa2020 % git cat-file -p 241c4
tree b24dd93d647bd7dffe447e8c9577b321bf6e8cc6
parent 4e545844480b9d849cb9ef970e713962688372ea7
author Cyfef <598061731@qq.com> 1742887292 +0000
committer Cyfef <598061731@qq.com> 1742887292 +0000

7219
```

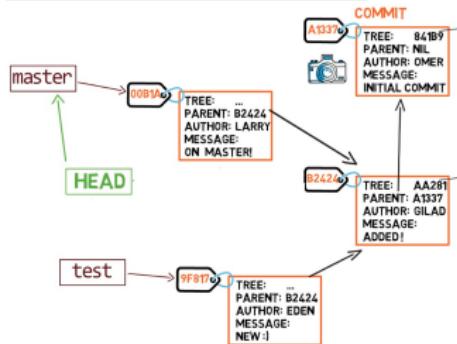
- merge 有两个 parent, 普通 commit 有一个 parent, 初始 commit 没有 parent
- 把每个 commit 看成一个顶点, parent 关系看成一条有向边, 我们得到了一张有向无圈图



DAG: Commit Graph

- 由哈希算法带来的不可篡改性

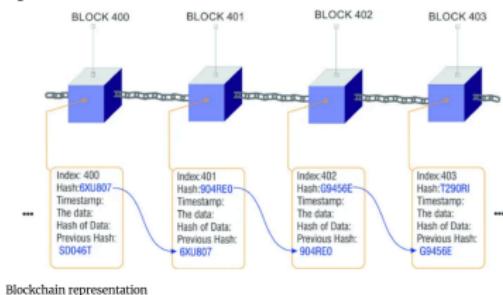
```
dsa2020 % git log --graph
```



- 固定 commit，它串起的所有结点都不可篡改
- 有向无圈图

- Bitcoin: Blockchain

Fig. 3



- 反时序单链表

Commit 中如何组织文件

● commit 中的 tree

```
dsa2020 % git cat-file -p ea0ea
```

```
100644 blob 286da5756e7e49320be0f3df57374c69dba24ba7 .gitignore  
100644 blob 25a5121d5e64fcfa305545ae00034538ca8f43ec README.md  
040000 tree d8a07476d28f1d34c7dc79b7cdf4bd66e3536f7 course  
040000 tree 0dfe284eca70d4cb9a924f1a5298d653a47bf3ba openjudge
```

```
dsa2020 % git cat-file -p 0dfe284
```

```
040000 tree 0a2d33c012a3da864e05da437664e251f1fbda8 8471  
040000 tree bdcf4fa7504643ea95c58b6450b0970d1f0482f 8581  
040000 tree 44580820837b5ab5bf8f86aa14f2d11lddbabf9 8758  
040000 tree 852279c587b12359463c9898ff7f33ed1bba2655 8780  
040000 tree f7c76784995475e431784477b6fad3f3fd187d7 90  
040000 tree 05e6b5b3ca27850ca3ee304d29db90c739adb442 9199  
040000 tree a330df400e319749d7439690f364d48cd3522ac 943  
100644 blob 861f31b1d3964a031bc599610674d3ba590337383 README.md
```

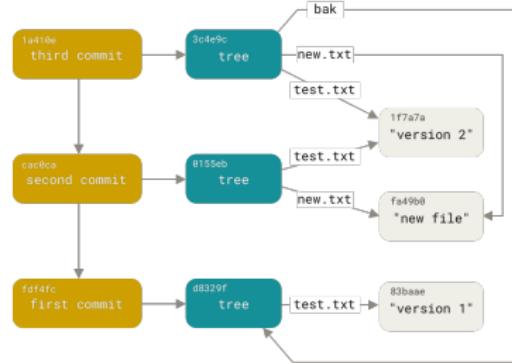
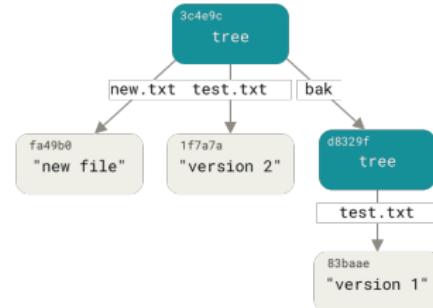
- tree 表示目录
- blob 表示文件—叶子结点
- 名字与内容分离

```
[zhengmaoxie@m1book foo % tree
```

```
.  
└── bak  
    └── test.txt  
── new.txt  
── test.txt
```

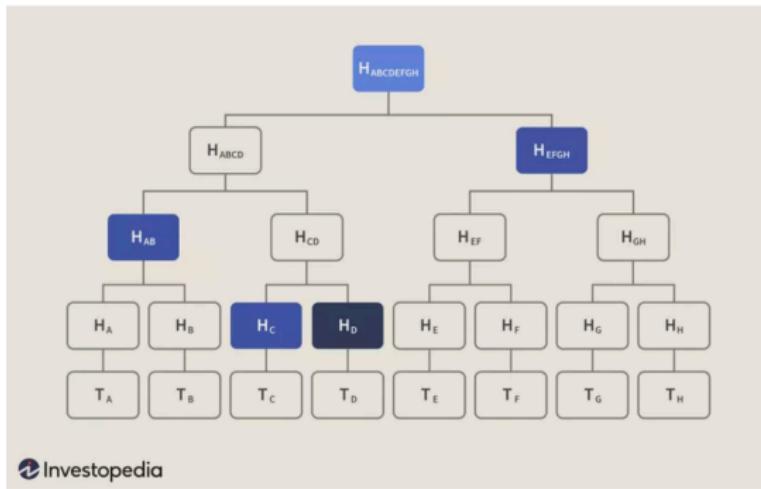
```
2 directories, 3 files
```

● 相同的 tree 和 blob 复用



Commit 中如何组织文件

- 固定 tree 结点，它串起的内容都不可篡改
- 修改一个 blob，会改动到根路径上的所有 tree 结点
- 从 commit 中的 tree 属性开始，可以串起该版本对应的所有文件
- blockchain 中的 merkle tree，下图中
 - H_{AB}, H_C, H_{EFGH} , and the root $H_{ABCDEFGH}$ 能够证明 T_D 的真伪



目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序



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
- 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										
Initial array state: 54, 26, 93, 17, 77, 31, 44, 55, 20, 20										Exchange
After first pass step 1: 26, 54, 93, 17, 77, 31, 44, 55, 20, 20										No Exchange
After first pass step 2: 26, 54, 93, 17, 77, 31, 44, 55, 20, 20										Exchange
After first pass step 3: 26, 54, 17, 93, 77, 31, 44, 55, 20, 20										Exchange
After first pass step 4: 26, 54, 17, 77, 93, 31, 44, 55, 20, 20										Exchange
After first pass step 5: 26, 54, 17, 77, 31, 93, 44, 55, 20, 20										Exchange
After first pass step 6: 26, 54, 17, 77, 31, 44, 93, 55, 20, 20										Exchange
After first pass step 7: 26, 54, 17, 77, 31, 44, 55, 93, 20, 20										Exchange
Final state after first pass: 26, 54, 17, 77, 31, 44, 55, 20, 20, 93										93 in place after first pass

冒泡排序：代码

n-1趟 →

```
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
```

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]

- **passnum**: 本趟比较元素的最大下标

冒泡排序：算法分析

- 前面的实现中初始数据项的排列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： $n(n-1)/2$
 - 比对的时间复杂度是 $O(n^2)$
- 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
- 排序保持稳定：两个数据项的排序码相等，排序后他们的顺序不变（保序）。
 - 冒泡排序是保持稳定的
- 什么时候出现最坏情况？逆序。

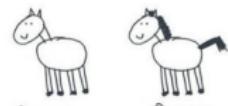
冒泡排序：算法分析

- 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。
- 就好像画画一样，算法也是先有轮廓，再逐步改进、完善细节。

怎样画马



① 基两个圆圈



② 基上腿

③ 基上脸



③ 基上脸

④ 基上毛发



④ 基上毛发

⑤
再添加其他细节
就大功告成了！

冒泡排序：性能改进

- 冒泡排序算法的一个特点：监测一趟比对，看是否发生过交换，就可以提前确定排序是否完成
- 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

```
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)
```

目录

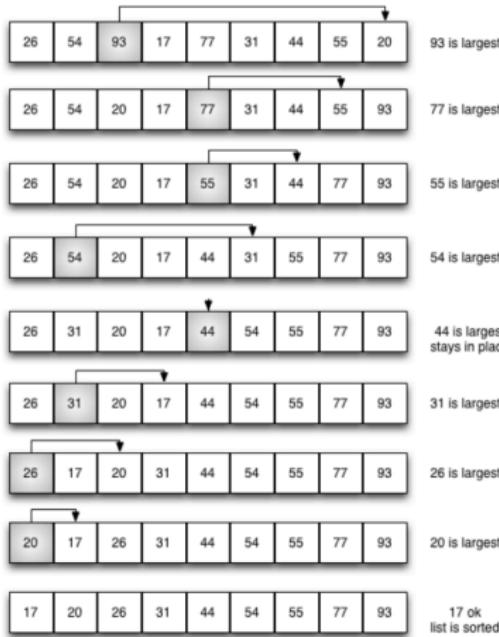
- 本章目标
- 顺序查找
- 二分查找
- 散列
- 冒泡排序、**选择排序**、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序



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 $O(n)$
- 选择排序保持稳定吗？

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
```

- 注意元素交换对排序稳定性的影响
- 能不能改进算法，保持选择排序的稳定性？
- 黄金口诀：稳定性看移动数据是否跨等值项

目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
- 冒泡排序、选择排序、**插入排序**
- 谢尔排序、归并排序、快速排序
- 分配排序



插入排序 Insertion Sort

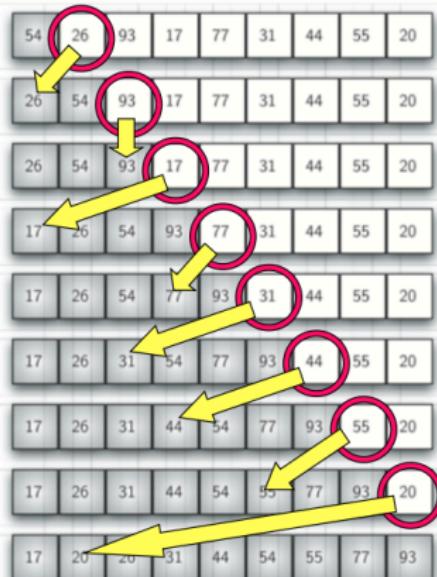
- 插入排序时间复杂度仍然是 $O(n^2)$ ，但算法思路与前两个不同
- 插入排序维持一个已排好序的子列表，其位置始终在列表的前部，然后逐步扩大这个子列表直到全表
 - 第 1 趟，子列表仅包含第 1 个数据项，将第 2 个数据项作为“新项”插入到子列表的合适位置中，这样已排序的子列表就包含了 2 个数据项；
 - 第 2 趟再继续将第 3 个数据项 从后往前 跟前 2 个数据项比对，并移动比自身大的数据项，空出位置来，以便加入到子列表中
 - 经过 $n-1$ 趟比对和插入，子列表扩展到全表，排序完成



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 $O(n)$ ；且无需移动数据
- 最坏情况，逆序。

插入排序：思路



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



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)):
```

新项/插入项 → currentvalue = alist[index]
position = index

比对、移动 → while position>0 and alist[position-1]>currentvalue:
alist[position]=alist[position-1]
position = position-1

插入新项 → alist[position]=currentvalue

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。

- 思考：对已排好部分用二分查找插入位置，会不会更高效？
- 思考：插入排序保持稳定吗？

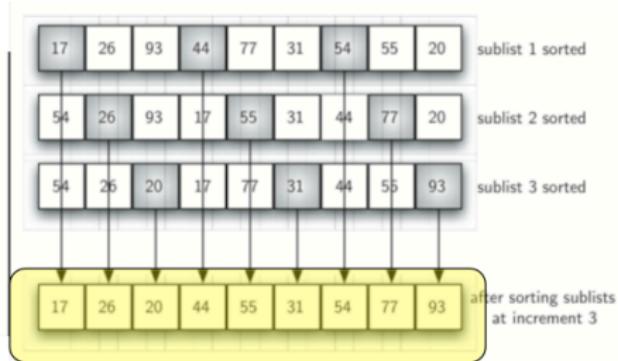
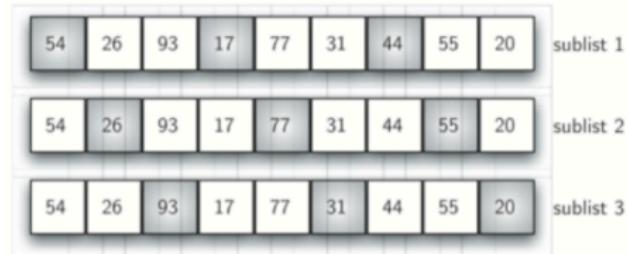
目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
- 冒泡排序、选择排序、插入排序
- **谢尔排序、归并排序、快速排序**
- 分配排序



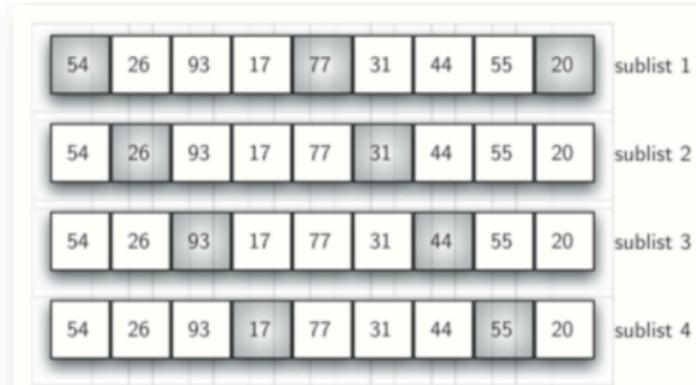
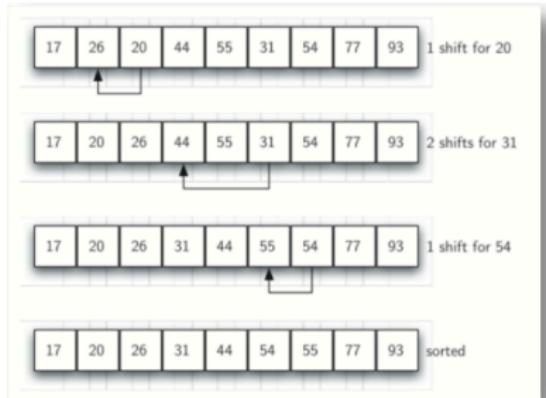
谢尔排序 Shell Sort

- 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
 - 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（左图）
- 子列表的间隔（取子列表的步长，同时也是子列表的个数）一般从 $n/2$ 开始，每趟倍减： $n/4, n/8 \dots \dots$ 直到 1



谢尔排序：代码

```
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
```

谢尔排序：算法分析

- 粗看上去，谢尔排序以插入排序为基础，进行了多次插入排序，效率应该更差才是
- 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做的“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 $O(n^{3/2})$
- 谢尔排序是稳定的吗？

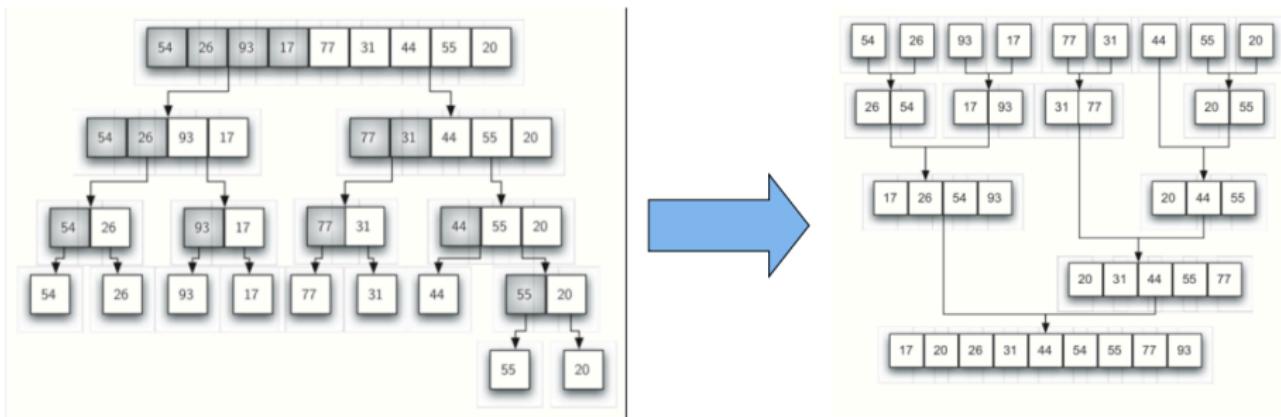
目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
- 冒泡排序、选择排序、插入排序
- 谢尔排序、**归并排序**、快速排序
- 分配排序



归并排序 Merge Sort

- 下面我们来看看分而治之的策略在排序中如何应用
- 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。



归并排序：代码

```
1 def mergeSort(lst):
2     if len(lst) <= 1:                      #基本结束条件
3         return
4     left, right = lst[:len(lst)//2], lst[len(lst)//2:]
5     mergeSort(left)                         #递归调用
6     mergeSort(right)
7     lst.clear()
8     i = j = 0
9     while i < len(left) and j < len(right):    #拉链式归并剩余最小项
10        if left[i] <= right[j] :
11            lst.append(left[i])
12            i += 1
13        else :
14            lst.append(right[j])
15            j += 1
16    else:                                     #归并所有剩余项
17        lst.extend(left[i:] if i<len(left) else right[j:])
18    return
19
20 if __name__ == "__main__":
21     l = [1, 0, 0, 8, 6]
22     mergeSort(l)
23     print(l)
```

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，分裂的层数为 $O(\log n)$
 - 归并的过程，相对于每一层的分裂，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，每层分裂都进行一次 $O(n)$ 的数据项归并，所以总的时间复杂度是 $O(n \log n)$
- 最后我们还注意到有两个切片操作，他们的复杂度是 $O(n)$ ，对上述的分析没有影响。
- 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
- 我们可以再次套用“主定理”，对归并排序进行分析。
- 归并排序是稳定的吗？

随堂思考

- 给定排序列表 [21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]，在归并排序的第 3 次递归调用时，排序的是哪个子表？
 - ① [16, 49, 39, 27, 43, 34, 46, 40]
 - ② [21, 1]
 - ③ [21, 1, 26, 45]
 - ④ [21]
- 排序数据同上，归并排序中，哪两个子表是最先归并的？
 - ① [21, 1] and [26, 45]
 - ② [1, 2, 9, 21, 26, 28, 29, 45] and [16, 27, 34, 39, 40, 43, 46, 49]
 - ③ [21] and [1]
 - ④ [9] and [16]

目录

- 本章目标
- 顺序查找
- 二分查找
- 散列
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、**快速排序**
- 分配排序



快速排序 Quick Sort

- 另一个采用分而治之策略的排序算法是快速排序，其优势是只需要很少的额外空间，这一点比归并排序强
 - 但是！快速排序不是总能“等分”数据表，这样在性能上就会打折扣
- 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
 - 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”应处的位置

- #### ● 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

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

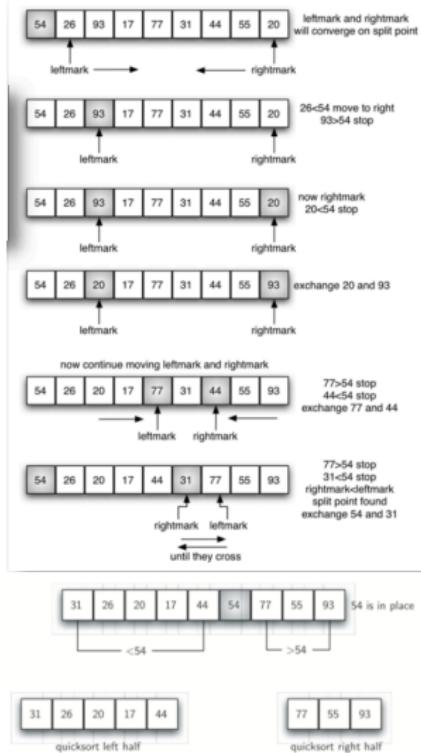
- 继续前述的移动过程，直到左标移到右标的右侧、停止移动

- 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 \leq 中值，右半部 \geq 中值

- 三种特殊情况：都比中值大；都小；等于中值。



快速排序：代码

```
18 def quickSort(alist):
19     _quickSort(alist, 0, len(alist))
20
21 def _quickSort(alist, first, last): #递归形式函数
22     if last - first > 1: # more than 1 elems基本结束条件
23         pivot = split(alist, first, last) #分裂为左右两半
24         _quickSort(alist, first, pivot)    #递归调用
25         _quickSort(alist, pivot+1, last)
26
27 if __name__ == "__main__":
28     quickSort(l:=[21,22,9]); print(l)
29     quickSort(l:=[54,26,93,17,77,31,44,55,21]); print(l)
```

```
[zhengmaoxie@m1book dsa2020 % python code/qSort.py
[9, 21, 22]
[17, 21, 26, 31, 44, 54, 55, 77, 93]]
```

快速排序：代码

```
1 def split(alist, first, last):
2     """
3         选第一个元素为中值，进行分裂
4     """
5     left, right = first+1, last-1      #左右标初值
6     while left <= right:                #左右相错之前一直执行
7         """ 访问元素之前先考虑下标是否越界 """
8         while left<=right and alist[left] <= alist[first]:
9             left += 1          #右移左标
10        while right>=left and alist[right] >= alist[first]:
11            right -= 1        #左移右标
12        if left < right:    #交换左右标元素，并继续移动
13            alist[left], alist[right] = alist[right], alist[left]
14        else:                  #中值点就位
15            alist[first], alist[right] = alist[right], alist[first]
16    return right                      #返回中值点位置
```

快速排序：算法分析

- 快速排序基本的过程也分为两部分：分裂和递归
 - 分裂过程将每项都与中值进行比对，时间复杂度是 $O(n)$ ；
 - 只使用了左右两个指针变量，分裂算法是原地工作的
 - 如果每次总能把数据表分裂成相等的两部分，则分裂层数是 $O(\log n)$ ，用递归实现的话，递归的深度也是 $O(\log n)$
 - 综合起来，快速排序时间复杂度就是 $O(n \log n)$ ；
 - 空间代价是 $O(\log n)$ ，非递归实现需要用栈保存每层的分裂结果，栈的深度也是 $O(\log n)$
- 但是如果不幸的话，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）
 - 比如：输入数据表已经排好序，或者完全逆序的情况

- 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
 - 改进有限，增加了代码的复杂度

- 给定排序列表 [14, 17, 13, 15, 19, 10, 3, 16, 9, 12]，快速排序在第 2 次分裂后，列表内容是（分裂算法有不同的实现，但是中值的位置是一样的）：
 - a) [9, 3, 10, 13, 12]
 - b) [9, 3, 10, 13, 12, 14]
 - c) [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]
 - d) [9, 3, 10, 13, 12, 14, 19, 16, 15, 17]
- 给定排序列表 [1, 20, 11, 5, 2, 9, 16, 14, 13, 19]，快速排序如果采用“三点取样”法，得到的第一个“中值”是：
 - a) 1 b) 9 c) 16 d) 19
- 下面哪些算法，即使在最坏情况下，复杂度还保证是 $O(n \log n)$
 - a) 谢尔排序 b) 快速排序 c) 归并排序 d) 插入排序

目录

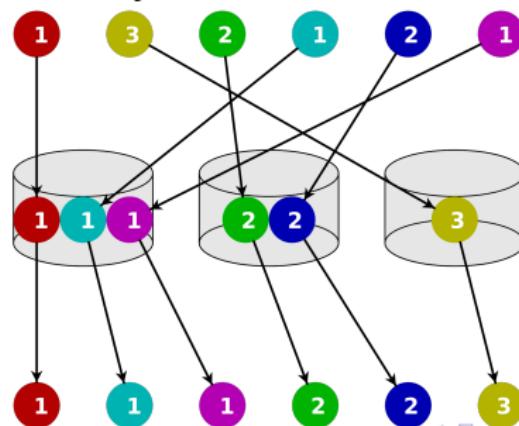
- 本章目标
- 顺序查找
- 二分查找
- 散列
- 冒泡排序、选择排序、插入排序
- 谢尔排序、归并排序、快速排序
- 分配排序



- 有一类排序问题，排序“码”可以分解成 d 个部分。通过对各个部分分别排序，最终实现对整个排序码的排序。
 - 世界杯小组赛排名：先看积分，再看净胜球，最后看进球数
 - 扑克牌排序：假设花色地位高于面值，则先看花色、再看面值
 - 邮编排序：100871, 422000 ..., 先看高位，再看低位
 - 整数排序，把高位用零补齐：123, 089, 333, 004, 056, 239
- 需要事先知道记录序列的一些具体情况

桶式排序

- 桶式排序是分配排序的一种简化版本 ($d=1$)，也可以看成分配排序 P 步中的一步。
- 分配排序和桶式排序的关系可以是“一般与特殊”，也可以是“整体与部分”
- 事先知道序列中排序的 **key** 取值只有少数的情况
- 将具有相同 **key** 值的记录都分配到同一个桶中，然后依次按照编号从桶中取出记录，组成一个有序序列
- 不需要比较两个元素的 **key**



桶式排序-计数法实现

待排数组:

7 3 8 9 6 1 8' 1' 2

每个桶count:

0	1	2	3	4	5	6	7	8	9
0	2	1	1	0	0	1	1	2	1

前若干桶的
累计count:

0	2	3	4	4	4	5	6	8	9
0	0	2	3	4	4	4	5	6	8

收集:

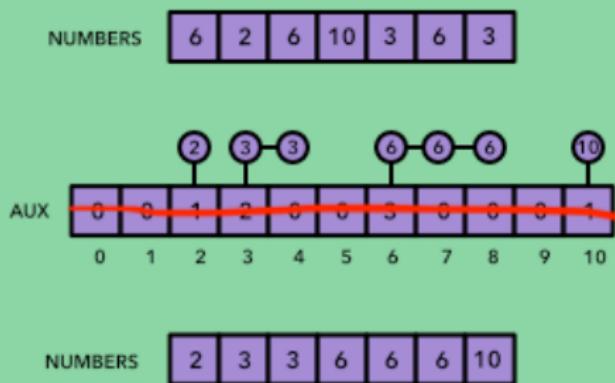
0	1	2	3	4	5	6	7	8	8'	9
1	1'	2	3	6	7	8	8'			9

- 获得累加计数 C 之后，从后往前扫描待排数组，对于元素 i，从 C 中查到自己的名次 C[i]，把 i 放入输出数组的第 C[i]-1 位（下标为 C[i]-1），同时 C[i] 减一。

桶排序-计数法实现算法分析

- 数组长度为 n , 所有记录在区间 $[0, m)$ 上
- 时间代价:
 - 统计计数: $O(n+m)$, 先计数再累计;
 - 总的时间代价为 $O(m+n)$
 - 适用于 m 相对于 n 很小的情况
- 空间代价
 - m 个计数器, 长度为 n 的临时数组, $O(m+n)$
- 排序保持稳定: 在排序码相等的情况下, 保持输入的序 (保序)
- 应用举例: 高考成绩排名

桶式排序-容器法实现



BUCKET SORT

<https://scriptverse.academy>

- 桶式排序只适合 m 很小的情况，否则空间开销太大。
- 基数排序：当 m 很大时，可以将一个记录的值即排序码拆分为多个部分来进行比较
- 假设长度为 n 的序列：
 - $R = \{r_0, r_1, \dots, r_{n-1}\}$
- 记录的排序码 K 包含 d 个子排序码：
 - $K = (k_{d-1}, k_{d-2}, \dots, k_1, k_0)$
- R 对排序码有序，即对于任意两个记录 $R_i, R_j (0 \leq i < j \leq n - 1)$ ，都满足：
 - $(k_{i,d-1}, k_{i,d-2}, \dots, k_{i,1}, k_{i,0}) \leq (k_{j,d-1}, k_{j,d-2}, \dots, k_{j,1}, k_{j,0})$
- 高位先排，递归分治
- 低位先排，要求稳定排序！

- MSD, Most Significant Digit first
- 先处理高位 k_{d-1} 将序列分到若干桶中
- 然后再对每个桶处理次高位 k_{d-2} ，分成更小的桶
- 依次重复，直到对 k_0 排序后，分成最小的桶，每个桶内含有相同排序码 (k_{d-1}, \dots, k_1, k_0)
- 最后将所有的桶中的数据依次连接在一起，成为一个有序序列
- 这是一个分、分、...、分、收的过程

- LSD, Least Significant Digit first
- 从最低位 k_0 开始排序
- 对于排好的序列再用次低位 k_1 排序；
- 依次重复，直至对最高位 k_{d-1} 排好序后，整个序列成为有序的
- 分、收；分、收；…；分、收的过程
- 反复调用“桶式排序”即可；
- 比较简单，计算机常用

基数排序的实现

- 主要讨论 LSD
- 原始输入数组 R 的长度为 n , 基数为 m , 排序码个数为 d
- 算法复杂度:
 - 时间复杂度: 等于桶式排序的 d 倍 $O(d*(n+m))$
 - 空间复杂度: 等于桶式排序 $O(n+m)$
 - 当 $d,m \ll n$ 时, 可以认为都是 $O(n)$
 - 单纯从复杂度上来说, 我们获得了最优的排序算法
 - 与 hash 搜索类似, 都是用空间换时间
 - 使用的场景受限: 排序码的取值范围是离散有限的;
 - 通过设计, 需要把 d 和 m 控制在一个较小的数值范围内。

链表之上的排序

- 在顺序表的基础上实现了各种排序算法，如何对链表进行排序？
 - 把链表复制到顺序表中，排序之后再复制回来
 - 直接对链表进行排序，哪些算法适用
 - 插入排序 vs. 希尔排序 (course/node.py)
 - 其他的排序算法呢？选择排序...

```
10 def insertSort(header:Node):  
11     if header is None:  
12         return None  
13     p, q = header, header.next  
14     p.next = None  
15     while q is not None:  
16         n, n.next, q = q, None, q.next  
17         prev, i = None, p  
18         while i is not None and i.data <= n.data:  
19             prev, i = i, i.next  
20         if prev is not None:  
21             prev.next, n.next = n, i  
22         else:  
23             n.next, p = p, n  
24     return p
```

1 假设线性表中每个元素有两个数据项 key1 和 key2，现对线性表按以下规则进行排序：先根据数据项 key1 的值进行非递减排序；在 key1 值相同的情况下，再根据数据项 key2 的值进行非递减排序。满足这种要求的排序方法是（ ）。

- A. 先按 key1 值进行冒泡排序，再按 key2 值进行直接选择排序
- B. 先按 key2 值进行冒泡排序，再按 key1 值进行直接选择排序
- C. 先按 key1 值进行直接选择排序，再按 key2 值进行冒泡排序
- D. 先按 key2 值进行直接选择排序，再按 key1 值进行冒泡排序

2 上一题得到的排序算法是稳定的吗？

答题：<https://ks.wjx.top/vj/ey1aoKB.aspx>

- 内排序：指在排序期间数据对象全部存放在内存的排序。
 - 内排序是排序的基础。
 - 内排序效率用比较次数来衡量。
 - 本章介绍的排序算法都可以用来进行内排序
- 外排序：指在排序期间全部对象太多，不能同时存放在内存中，必须根据排序过程的要求，不断在内，外存间移动的排序。
 - 在数据量大的情况下，只能分块排序，但块与块间不能保证有序。
 - 块分得足够小，可以放进内存中完成单个块的排序。
 - 再通过多路归并的办法对排好序的小块进行合并排序
 - 重点考虑外存的特点：顺序的读写；效率与读/写次数有关。

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
- 冒泡排序、选择排序和插入排序是 $O(n^2)$ 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 $O(n \log n)$ ，但如果分裂点偏离列表中心的话，最坏情况下会退化到 $O(n^2)$ ；它需要用 $O(\log N)$ 的栈空间保存每次分裂的结果。
- 分配排序在有限的应用场景下，实现了 $O(n)$ 的算法复杂度