散列

What is hash
什么是散列

Hash: Hashing
散列：Hashing

Previously, we used the knowledge about the permutation relationship between the data items to improve the search algorithm

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

If the data items are ordered in terms of their size, the binary search can be used to reduce the algorithm complexity.

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。

Now we proceed to construct a new data structure that reduces the complexity of the search algorithm to O (1), a concept called a "Hashing"

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”

Hash: Hashing

散列：Hashing

To reduce the number of search to the constant level, we must have more prior knowledge of the position where the data item is located.

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。

If we can know in advance where the data item should appear in the data set, we can go directly to that location to see if the data item exists.

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可。

How can this be determined by the value of the data item?

由数据项的值来确定其存放位置，如何能做到这一点呢？

Hash: Basic concept

散列：基本概念

A hash table is a data set, in which the storage method of data items is particularly conducive to fast search and positioning in the future.

散列表(hashtable，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。

Each storage location in the hash table, called a (slot), which can be used to save the data items, and each slot has a unique name.

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。

Hash: Basic concept

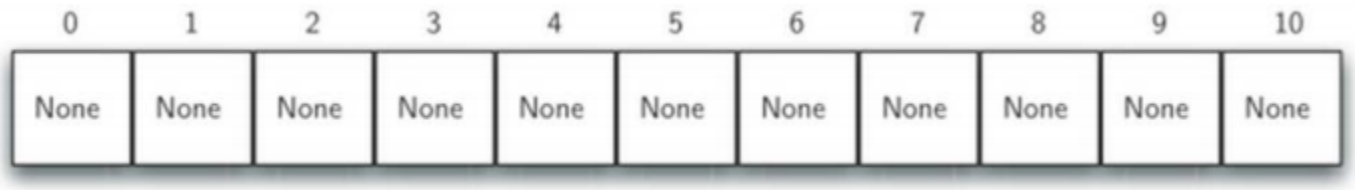
散列：基本概念

For example, a hash table of 11 slots with names of 0 to 10

例如：一个包含11个槽的散列表，槽的名称分别为0~10

Before inserting the data item, the value of each slot is *None*, indicating the empty slot

在插入数据项之前，每个槽的值都是None，表示空槽



Hash: Basic concept

散列：基本概念

Conversion from data item to storage slot name is called hash function

实现从数据项到存储槽名称的转换的，称为散列函数(hashfunction)

In the following example, the hash function accepts the data item as a parameter and returns an integer value of 0-10, representing the slot number (name) of the data item store

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)

Hash: example

散列：示例

To save the data items into the hash table, we designed the first hash function

为了将数据项保存到散列表中，我们设计第一个散列函数

Data item: 54,26,93,17,77,31

数据项：54，26，93，17，77，31

A common hash method is used by "finding the remainder", dividing the data item by the size of the hash table and getting the remainder as the slot number

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。

In fact, the remainder method appears with different way in all hash functions

实际上“求余数”方法会以不同形式出现在所有散列函数里

Because the slot number returned by the hash function must be within the hash size range, therefore, the remainder of the hash table size is generally calculated.

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

Hash: example
散列：示例

Our hash function is the simplest case:

本例中我们的散列函数是最简单的求余：

h(item)=item%11

After following the hash function h (item) and calculating the storage location for each data item, store the data items in the corresponding slot

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

Hash: example

散列：示例

The six data items in the example were inserted, occupying six of the 11 slots in the hash table.

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。

The proportion of the slots occupied by the data items is called the "load factor" of the hash table, where the load factor here is 6 / 11

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

After the data items are saved to the hash table, the search is very easy

数据项都保存到散列表后，查找就无比简单

To find out whether a data item exists in the table, we only need to use the same hash function to calculate the search item, and test whether there are data items in the slot corresponding to the returned slot number

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

The search algorithm of O(1) time complexity is implemented.

实现了O(1)时间复杂度的查找算法。

Hash: example

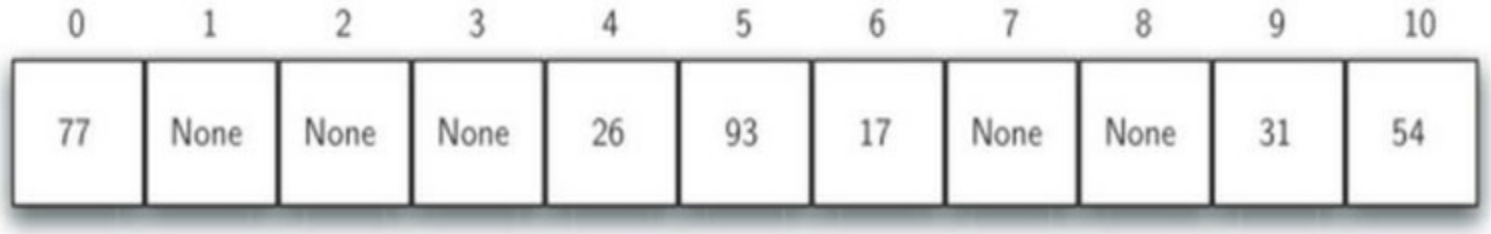
散列：示例

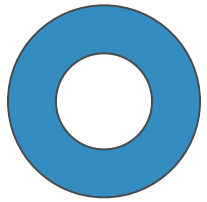
However, you may also see the problem with this solution, which is each occupying a different slot with quite coincident

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

If you gonna to save 44, h (44) =0, and it is assigned to the same 0 # slot as 77, this situation is called "collision", and we will discuss the solution to this problem later.

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。





Perfect hash function
完美散列函数

Perfect hash function
完美散列函数

Given a set of data items, a hash function can be called a "perfect hash function" if a hash function can map all data item to a different slot

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”

For a fixed set of data, you can always try to design a perfect hash function

对于固定的一组数据，总是能想办法设计出完美散列函数

However, if the data items change regularly, it is difficult to have a systematic method to design the corresponding perfect hash function

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

Of course, the collision is not a fatal error, and we will have ways to deal with it.

当然，冲突也不是致命性的错误，我们会有办法处理的。

Perfect hash function
完美散列函数

One way to obtain a perfect hash function is to expand the capacity of the hash table, so large that all possible data items are able to occupy different slots

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

But this approach is not practical for cases where possible data items are too large

但这种方法对于可能数据项范围过大的情况并不实用

- If we want to save the mobile phone number (11 digits), the perfect hash function requires a hash list with ten billion slots! It would waste too much storage space

  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

Second priority, good hash functions need to have characteristics

退而求其次，好的散列函数需要具备特性

- Minimum collision (almost perfect), low computational difficulty (less additional cost), sufficient dispersion of data items (space saving)

  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)

More uses for the perfect hash functions
完美散列函数的更多用途

In addition to arranging the storage location of data items in a hash table, hash techniques are also used in many areas of information processing

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

Because the perfect hash function can generate different hash values for any different data, this feature is widely used, if the hash value is taken as the "fingerprint" or "summary" of the data

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

The generation of fixed-length "fingerprints" of any length also requires uniqueness, which is not mathematically possible, but the cleverly designed "quasi-perfect" hash function can do this within a practical range.

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。

More uses for the perfect hash functions
完美散列函数的更多用途

The data "fingerprint" function used as a consistency check needs to have the following features

作为一致性校验的数据“指纹”函数需要具备如下的特性

①Compressibility: any length of data, the "fingerprint" length is fixed;

压缩性：任意长度的数据，得到的“指纹”长度是固定的；

②Easy to calculate: easy to calculate "fingerprint" from original data; (impossible to calculate original data from fingerprint);

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)；

③Anti-modification: small changes in the original data will cause big changes in the "fingerprint";

抗修改性：对原数据的微小变动，都会引起“指纹”的大改变；

④Collision resistance: Knowing original data and "fingerprint", it is very difficult to find data with the same fingerprint (forged)

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

Hash function:MD5 / SHA

散列函数MD5/SHA

The most well-known approximate perfect hash functions are the MD5 and SHA series functions

最著名的近似完美散列函数是MD5和SHA系列函数

The MD5 (Message Digest) transforms the data of any length to a *Summary* with a fixed length of 128 bits (16 bytes)

MD5（MessageDigest)将任何长度的数据变换为固定长为128位(16字节)的 “摘要”

The 128-bit binary is already a huge digital space: it is said to be the number of grains of sand on the earth

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

Hash function:MD5 / SHA

散列函数MD5/SHA

SHA (Secure Hash Algorithm) is another set of hash functions

SHA(SecureHashAlgorithm)是另一组散列函数

- SHA-0 / SHA-1 Output Hash Value of 160 bits (20 bytes),

  SHA-0/SHA-1输出散列值160位(20字节)，

  SHA-256 / SHA-224 outputs 256 and 224 bits, and SHA-512 / SHA-384 puts 512 and 384 bits, respectively

  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

The 160-bit binary is equivalent to 10 to the 48th power and the number of water molecules on Earth is estimated to be 10 to the 47th power

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

The 256-bit binary is equivalent to 10 to the 77th power, and all elementary particles in the universe are known to be about 72th to 87th power

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

Hash function:MD5 / SHA
散列函数MD5/SHA

Although three hash functions, MD5 / SHA-0 / SHA-1, were recently discovered
虽然近年发现MD5/SHA-0/SHA-1三种散列函数

whose ability to construct individual collisions (hash conflicts)
能够以极特殊的情况来构造个别碰撞(散列冲突)

But there is never a real threat in practicality.
但在实用中从未有实际的威胁。

Knowledge about orders of magnitude:
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)

The Python's hash function library, hashlib

Python的散列函数库hashlib

Python hash function library with MD5 and SHA series: hashlib

Python自带MD5和SHA系列的散列函数库：hashlib

Six hash functions like md5 / sha1 / sha224 / sha256 / sha 384 / sha512 are included

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'
```

The Python's hash function library, hashlib

Python的散列函数库hashlib

In addition to the hash calculation of a single string,

除了对单个字符串进行散列计算之外，

The *update* method can also be used to calculate the arbitrarily long data parts,

还可以用update方法来对任意长的数据分部分来计算，

In this way, no matter how big the data is, there will be no problem of using out of memory .

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'
>>>
```

The Perfect hash function is used for data consistency checking
完美散列函数用于数据一致性校验

Data file consistency judgment
数据文件一致性判断

Calculate its hash value for each file, and only compare the hash value to know whether the file content is the same;
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；

Used for network file download integrity verification;
用于网络文件下载完整性校验；

For file sharing systems: There is no need for the same files (especially movies) on the web plate to store multiple times.
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。

The Perfect hash function is used for data consistency checking
完美散列函数用于数据一致性校验

Save your password in an encrypted form
加密形式保存密码

Only save the hash value of the password. After the user enters the password, calculate the hash value and compare it;
仅保存密码的散列值，用户输入密码后，计算散列值并比对；

To tell whether the user has entered the correct password without saving the clear text of the password.
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。

The Perfect hash function is used for data consistency checking

完美散列函数用于数据一致性校验

File tamper-proof: the principle and the data file consistency judgment

防文件篡改：原理同数据文件一致性判断

Of course, there are more cryptography mechanisms to protect data files, tamper-proof, anti-denial, is the information technology foundation of e-commerce.

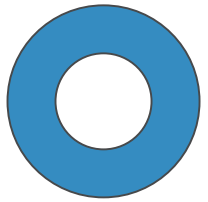
当然还有更多密码学机制来保护数据文件，防篡改，防抵赖，是电子商务的信息技术基础。

Lottery betting application

彩票投注应用

Before the lottery player bets, the institution publishes the hash value of the winning result, and then the lottery player makes a bet. After the lottery draw, the lottery player can verify whether the institution is cheating by comparing the published result with the hash value.

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。



Blockchain technology
区块链技术

The coolest application of hash functions: blockchain technology
散列函数的最酷应用：区块链技术

Blockchain is a distributed database
区块链是一种分布式数据库

- A node connected through a network
  通过网络连接的节点
- Each node holds the entire database
  每个节点都保存着整个数据库
- All data Data stored anywhere is synchronized
  所有数据任何地点存入的数据都会完成同步

The coolest application of hash functions: blockchain technology
散列函数的最酷应用：区块链技术

The most essential feature of blockchain is "decentralization"
区块链最本质特征是“去中心化”

There are no control center, coordination center nodes
不存在任何控制中心、协调中心节点

All the nodes are equal and cannot be controlled
所有节点都是平等的，无法被控制

How to prevent tampering and destruction without requiring mutual trust and authority?
如何做到不需要相互信任和权威，即可防止篡改和破坏？

Blockchain
区块链

Blockchain consists of blocks (block), divided into (head) and (body)

区块链由一个个区块(block)组成，区块分为头(head)和体(body)

The block head records some metadata and information linked to the previous block

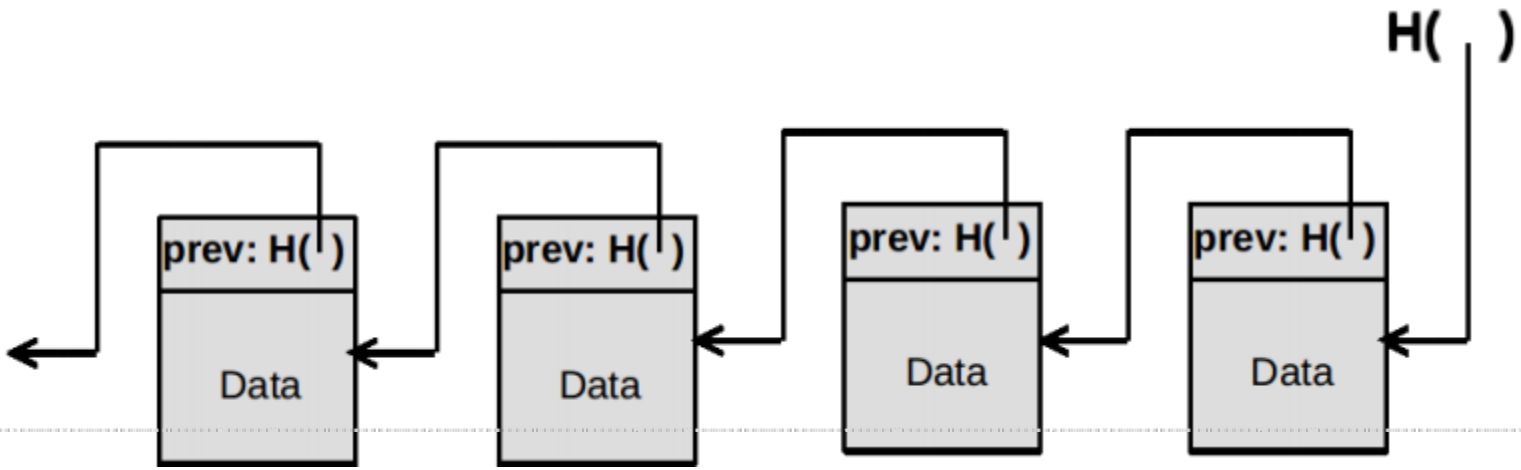
区块头记录了一些元数据和链接到前一个区块的信息

- Generation time, the hash value of the previous block (head + body)

  生成时间、前一个区块(head+body)的散列值

- The block records the actual data

  区块体记录了实际数据



Blockchain is not modifiable
区块链不可修改性

Because the hash value is resistant to modification, any change to a block data must cause a change in the hash value

由于散列值具有抗修改性，任何对某个区块数据的改动必然引起散列值的变化

In order not to cause this block to leave the chain, all subsequent blocks need to be modified, and owing to the "proof-of-work" mechanism, such large-scale modification is impossible unless you have 51% of the computing power of the whole network

为了不导致这个区块脱离链条，就需要修改所有后续的区块，由于有“工作量证明”的机制，这种大规模修改不可能实现的，除非掌握了全网51%以的计算力

Proof of Work (POW)
工作量证明(POW)

Since the blockchain is a large-scale distributed database, the synchronization is slow, so the addition speed of new blocks needs to be controlled

由于区块链是大规模的分布式数据库，同步较慢，新区块的添加速度需要得到控制

The speed currently adopted by the largest blockchain Bitcoin is to generate a block every 10 minutes averagely

目前最大规模区块链Bitcoin采用的速度是平均每10分钟生成一个区块

People pay a lot of calculations to figure out the effective hash value of a block

大家不惜付出海量的计算，去抢着算出一个区块的有效散列值

The first "miner" is qualified to hang the block in the blockchain

最先算出的那位“矿工”才有资格把区块挂到区块链中

Proof of Work (POW)
工作量证明

Aren't the hashes very easy to calculate? Why pay a lot of computing? Why get ahead?
散列不是非常容易计算吗？为什么要付出海量计算？为什么要抢先？

Why are the effective hash values so hard to calculate?

为什么有效散列值那么难算出？

Since it is difficult to calculate, the speed of new blocks is controlled to facilitate synchronization across the whole distributed network

因为很难算出，所以控制了新区块生成的速度，便于在整个分布式网络中进行同步

Each block sets a difficulty coefficient, divide it by the constant targetmax to get a target. The higher the difficulty coefficien is, the smaller the target is

每个区块设置了一个难度系数Difficulty，用常数targetmax除以它，得到一个target，难度系数越高，target越小

```
target = targetmax / difficulty
```

```
targetmax = 0x00000000FFFF0000000000000000000000000000000000000000000000000000
difficulty = 14484.162361
```

Why are the effective hash values so hard to calculate?
为什么有效散列值那么难算出？

The miner's job is to find a numerical Nonce and calculate the hash together with the entire block data. The hash value must be less than target to be a valid hash value

矿工的工作是，找到一个数值Nonce，把它跟整个区块数据一起计算散列，这个散列值必须小于target，才是有效的散列值

Since the hash value cannot be pushed back to the original value, this Nonce search is only violent and exhaustive, and the calculation of workload + luck is the only one way

由于散列值无法回推原值，这个Nonce的寻找只能靠暴力穷举，计算工作量+运气是唯一的方法。

```
target = targetmax / difficulty
```

```
targetmax = 0x00000000FFFF0000000000000000000000000000000000000000000000000000
difficulty = 14484.162361
```

A block of the Bitcoin: https ://blockexplorer .com /

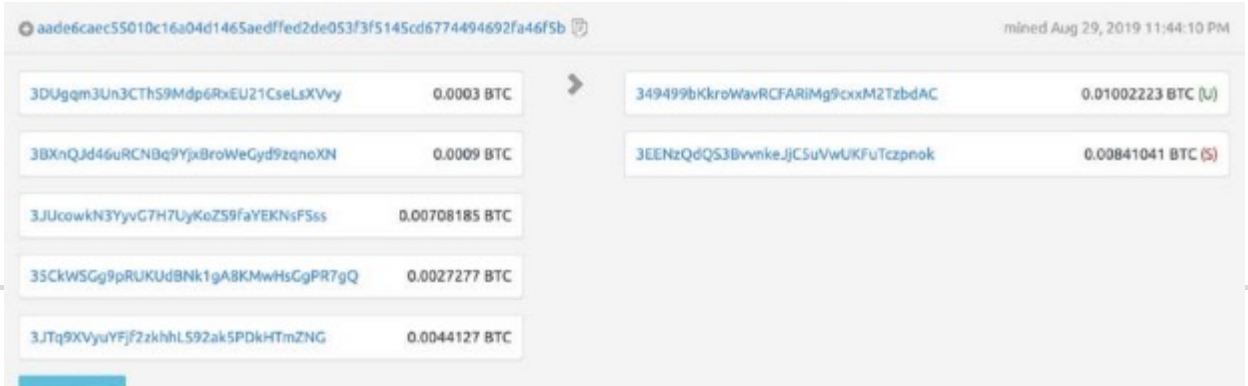
Bitcoin的一个区块https://blockexplorer.com/

Block #592304

BlockHash 00000000000000000001482359dd524f70e7ade31fba3d9a73fb4ca60ba9921f

Summary

Number Of Transactions	2375
Height	592304 (Mainchain)
Block Reward	12.5 BTC
Timestamp	Aug 29, 2019 11:44:10 PM
Mined by	
Merkle Root	279811f1a1d17ec1956cc6a...
Previous Block	592303
Difficulty	10183488432890
Bits	171ba3d1
Size (bytes)	946095
Version	545259520
Nonce	1154612744
Next Block	592305



Why are the miners rushing to generate blocks?
为什么矿工抢着生成区块？

Because of the interest!
因为有利益！

In the cryptocurrency Bitcoin, the data contained in the block is a "transaction record," or a "ledger book," which is critical to the monetary system
在加密货币Bitcoin中，区块内包含的数据是“交易记录”，也就是“账本”，这对于货币体系至关重要

The Bitcoin stipulates that each block contains a certain amount of Bitcoin as an "bookkeeping reward," thus encouraging more people to join the bookkeeping fashion
Bitcoin规定，每个区块中包含了一定数量的比特币作为“记账奖励”，这样就鼓励了更多人加入到抢先记账的行列

Why are the miners rushing to generate blocks?
为什么矿工抢着生成区块？

Due to the hardware existence of Moore's Law, the computing power will continue to increase, and the coefficient Difficulty will also continue to increase in order to maintain the speed of generating a block every 10 minutes

由于硬件摩尔定律的存在，计算力将持续递增，为了维持每10分钟生成一个区块的速度，难度系数Difficulty也将持续递增

- Mining Computing Power Upgrade: CPU (20MHas h / s) GPU (400MHash / s) FPGA (25GHash / s) AS IC (3.5THash / s) Large-scale cluster mining (3.5THash / s * X)

  挖矿计算力升级：CPU(20MHash/s)→GPU(400MHash/s)→FPGA(25GHash/s)→ASIC(3.5THash/s)→大规模集群挖矿(3.5THash/s*X)

In addition, in order to keep the monetary aggregate not increasing indefinitely, the bitcoin award is halved every four years

另外，为了保持货币总量不会无限增加，每4年奖励的比特币减半

- It started in 50 in 2008 and 12.5 in 2019

  2008年开始是50个，2019年为12.5个

Why are the miners rushing to generate blocks?
为什么矿工抢着生成区块？

Witness history! Just now, Bitcoin halved!

2024-04-20 09:09 Securities Times Sina Finance APP

The once-in-four-years software update, called the “halving,” is here.

Data shows that at 8:09 am Beijing time on April 20, 2024, Bitcoin successfully completed its fourth halving at block height 840,000. The mining reward of the Bitcoin network was halved from 6.25 BTC to 3.125 BTC. The last halving occurred on May 11, 2020.

After the Bitcoin halving was completed, the price of Bitcoin rose slightly and is now reported at US$63,914 per coin.

Investment Rese
Don’t miss the 14-day SOE reform + super bran + e-commerce + small
11-27 15:46

Market sentiment is wai two long-term value lea long-term layout opport
11-27 12:03

State-owned enterprise payment + e-commerce + elements + virtual
11-27 10:58

Live stock mark
Live
broadcast

见证历史！刚刚，比特币减半！

2024年04月20日 09:09 证券时报 新浪财经APP

四年一次的软件更新（称为“减半”）来了。

数据显示，北京时间2024年4月20日8:09，比特币于区块高度840000成功完成第四次减半，比特币网络的挖矿奖励由6.25BTC减半至3.125BTC，上一次减半发生在2020年5月11日。

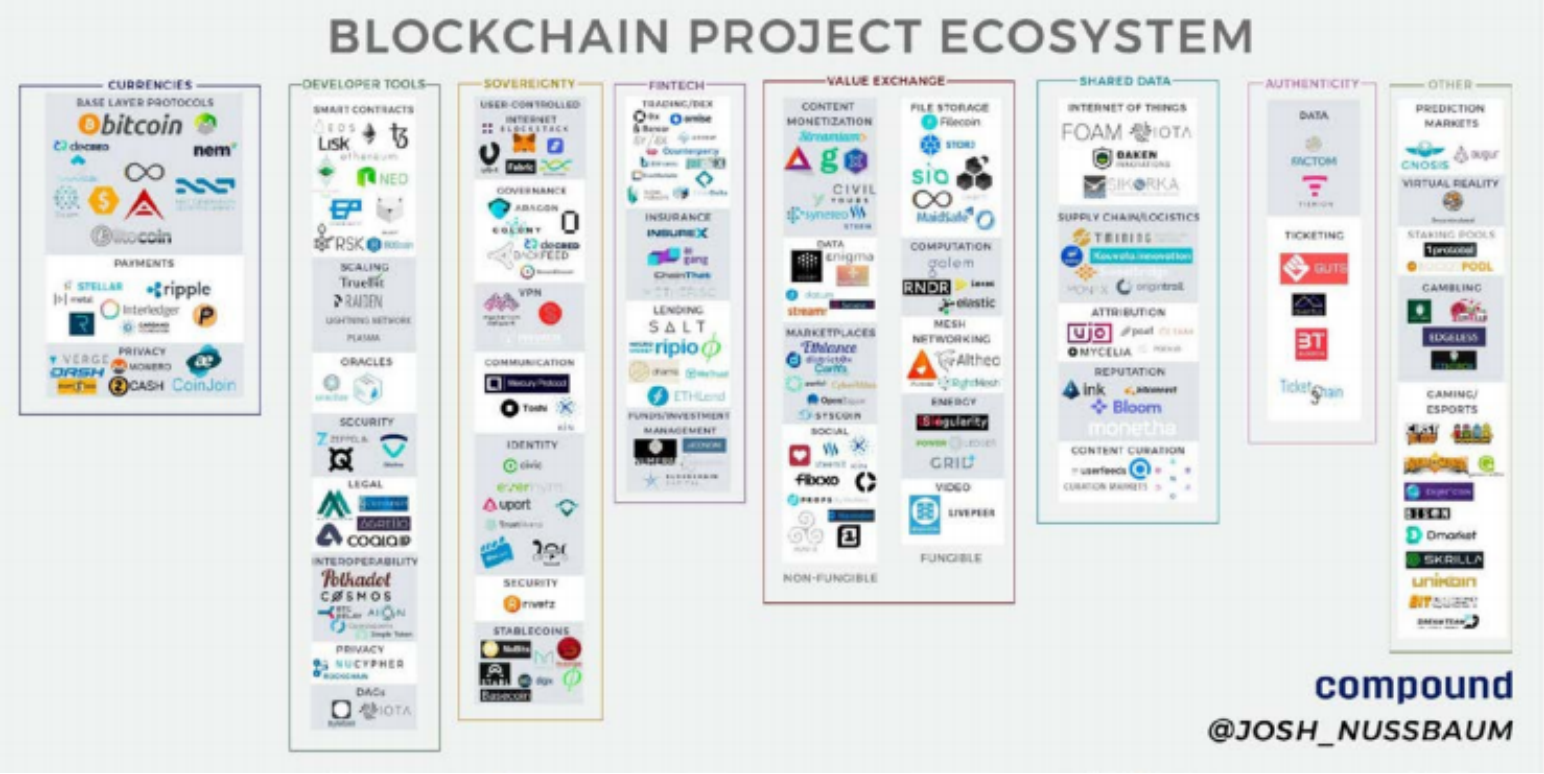
比特币减半完成后，比特币价格小幅上涨，现报63914美元/枚。

Cryptocurrency, Bitcoin
加密货币Bitcoin

The world of "miners"
“矿工”的世界

The rapidly expanding application of blockchain technology
区块链技术飞速扩张的应用

https://36kr.com/p/5114727.html



Hash function design
散列函数设计

Hash function design: Folding method
散列函数设计：折叠法

The basic step of the folding method to design the hash functions is that
折叠法设计散列函数的基本步骤是

Divide data items into segments,
将数据项按照位数分为若干段，

Add several pieces of numbers,
再将几段数字相加，

Finally, take the remainder of the size of the hash table
最后对散列表大小求余，得到散列值

For example, to the phone number 62767255
例如，对电话号码62767255

It can be divided into four segments (62,76,72,55)
可以两位两位分为4段(62、76、72、55)

Addition(62 + 76 + 72 + 55=265)
相加(62+76+72+55=265)

The hash table includes 11 slots, which is 265%11=1, so h (62767255) =1
散列表包括11个槽，那么就是265%11=1所以h(62767255)=1

Hash function design: Folding method

散列函数设计：折叠法

Sometimes the folding method also includes a step of interval number inversion

有时候折叠法还会包括一个隔数反转的步骤

For example, (62,76,72,55) inversion is reversed (62,67,72,55)

比如(62、76、72、55)隔数反转为(62、67、72、55)

Plus(62+67+72+55=256)

再累加(62+67+72+55=256)

For 11(256%11=3), so h'(62767255)=3

对11求余(256%11=3)，所以h'(62767255)=3

Although the interval inversion seems theoretically unnecessary, this step does provide a sutle-tuning means for the folding method to obtain the hash function, in order to better conform with the hash characteristics

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

Hash function design: the middle square method
散列函数设计：平方取中法

The square method is to square the data item first, then take the middle two digits of the squared number, and then calculate the remainder of the size of the hash table.

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

For example, make a hash for 44

例如，对44进行散列

First, 44 * 44=1936

首先44*44=1936

Then take the middle number 93

然后取中间的93

modulo hash table size 11, 93%11=5

对散列表大小11求余，93%11=5

Hash function design: the middle square method
散列函数设计：平方取中法

The following table is a comparison of the two hash functions
下表是两种散列函数的对比

- Both are the perfect hash functions
  两个都是完美散列函数
- The dispersion is very good
  分散度都很好
- The middle square method's calculation amount is slightly larger
  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

Hash function design: Non-numerical items

散列函数设计：非数项

We can also hash on non-numeric data items, viewing each character in the string as a ASCII code

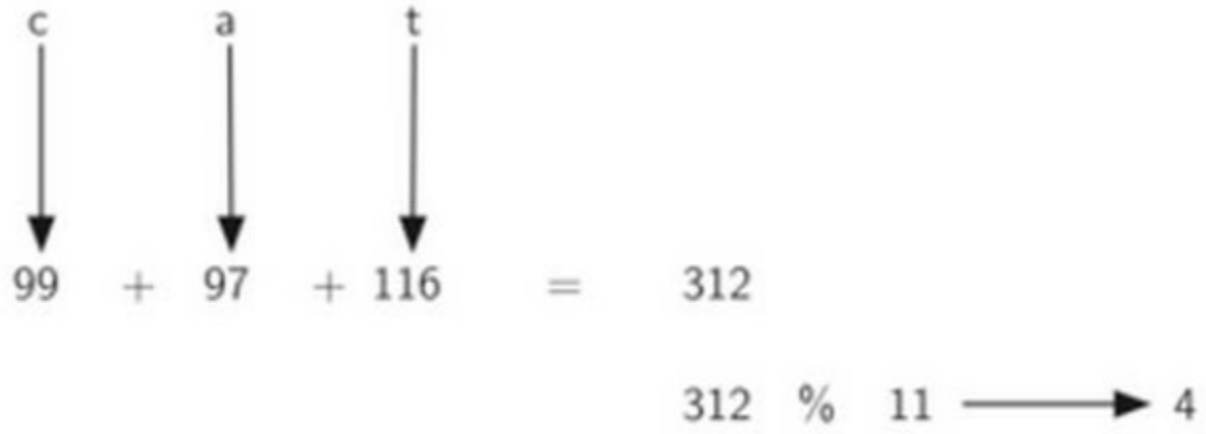
我们也可以对非数字的数据项进行散列，把字符串中的每个字符看作ASCII码即可

For example, cat, ord('c') ==99, ord('a') ==96, ord('t') ==116

如cat，ord('c')==99,ord('a')==96,ord('t')==116

Then accumulate these integers, and calculate the remainder of the size of the hash table

再将这些整数累加，对散列表大小求余



code
代码

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

    return sum%tablesize
```

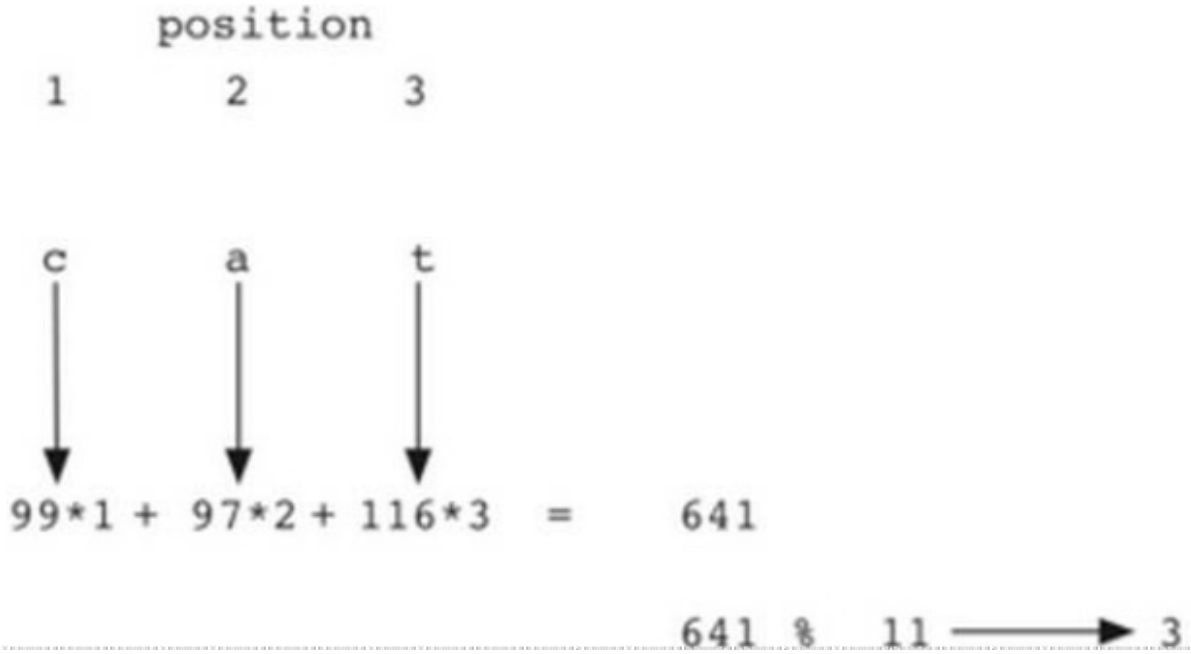
Hash function design
散列函数设计

Of course, such a hash function returns the same hash value for all anagram

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

To prevent this, you can take the location of the string as a weight factor, multiplied by the ord value

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



Hash function design
散列函数设计

We can also devise more hash function methods, but a basic starting point to insist on is that the hash function cannot be a computational burden for stored procedures and lookup processes

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

If the hash function design is too complex, to spend a lot of computational resources to calculate the slot numbers,

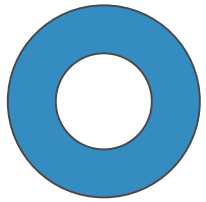
如果散列函数设计太过复杂，去花费大量的计算资源计算槽号

it will be better to simply do sequential or binary search might as will

可能还不如简单地进行顺序查找或者二分查找

Lost the meaning of the hash itself

失去了散列本身的意义



Solution to Collision

冲突解决方案

Solution to Collision

冲突解决方案

If two data items are mapped to the same slot, a systematic method is needed to save the second data item in the hash list, a process called "collision resolution"

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”

As mentioned earlier, if the hash function is perfect, there will be no hash conflict, but the perfect hash function is often unrealistic

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

Solving hash collisions becomes an important part of the hash approach.

解决散列冲突成为散列方法中很重要的一部分。

Solution to Collision
冲突解决方案

One way to solve the hash is that for conflicting data items to
解决散列的一种方法就是为冲突的数据项

Find another open empty slot to save it
再找一个开放的空槽来保存

The simplest thing is to scan back from the conflicting slot until you hit an empty slot
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

If not found at the end of the hash, scan from the first
如果到散列表尾部还未找到，则从首部接着扫描

This technique of finding empty slots is called the "open addressing"
这种寻找空槽的技术称为“开放定址openaddressing”

The slot-by-slot search backward method is the "linear probing" in the open address technology
向后逐个槽寻找的方法则是开放定址技术中 的 “线性探测 linearprobing”

Linear Probing
线性探测LinearProbing

We inserted 44,55,20 into the hash table one by one

我们把44、55、20逐个插入到散列表中

With h (44) =0, 0 # slot has been occupied by 77, backward to find first empty slot 1 #, save

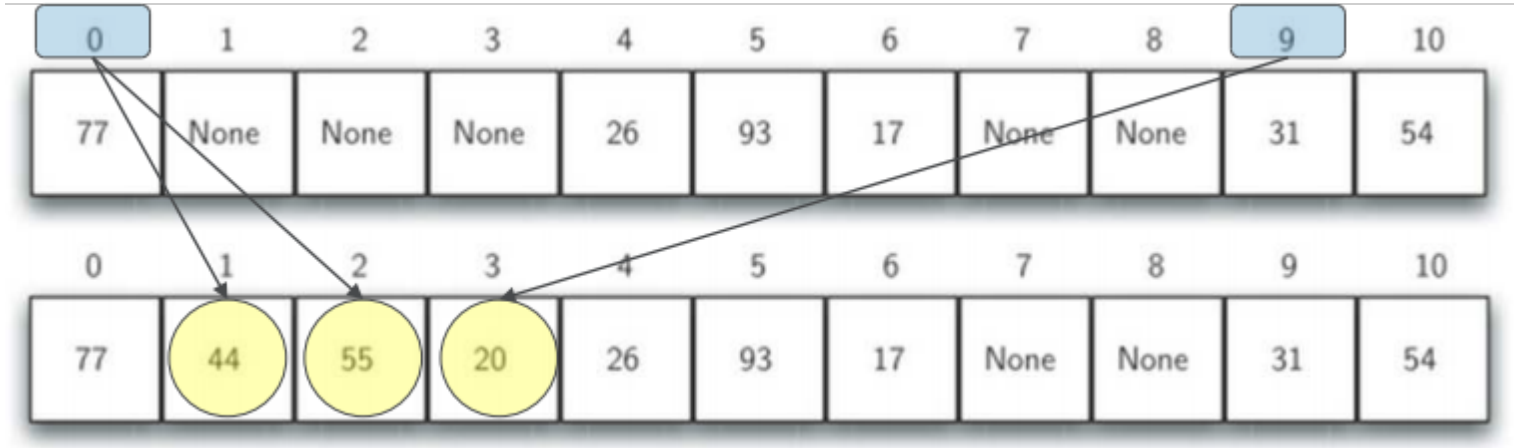
h(44)=0，但发现0#槽已被77占据，向后找到第一个空槽1#，保存

With h (55) =0, also 0 # slot has been occupied, backward to find the first empty slot 2 #, save

h(55)=0，同样0#槽已经被占据，向后找到第一个空槽2#，保存

For h (20) =9, find that 9 # slot has been occupied by 31, backward and then find 3 # slot from scratch

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



Linear Probing
线性探测LinearProbing

If a linear detection method is used to resolve the hash collisions,

采用线性探测方法来解决散列冲突的话，

The search of hash table follows the same rule

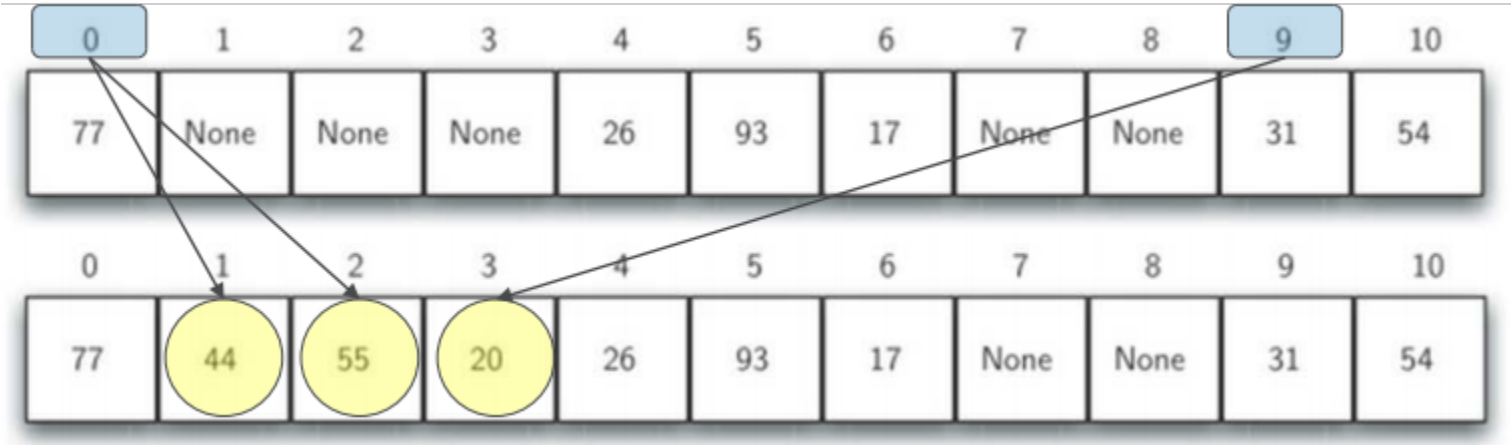
则散列表的查找也遵循同样的规则

If no search item is found in the hash, you must do the sequential search backward

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

Until the search item is found, or meet the empty slot (search failed).

直到找到查找项，或者碰到空槽(查找失败)。



Improvements in the linear probing
线性探测的改进

A disadvantage of the linear probing method is the tendency of aggregation (clustering)

线性探测法的一个缺点是有聚集(clustering)的趋势

That is, if there are more collision data items in the same slot,

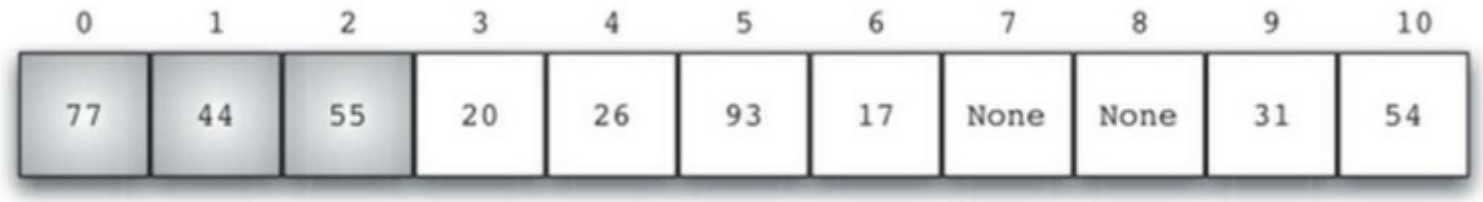
即如果同一个槽冲突的数据项较多的话，

These data items will be clustered near the slot

这些数据项就会在槽附近聚集起来

Thereby affecting the insertion of other data items in a cascading way

从而连锁式影响其它数据项的插入。



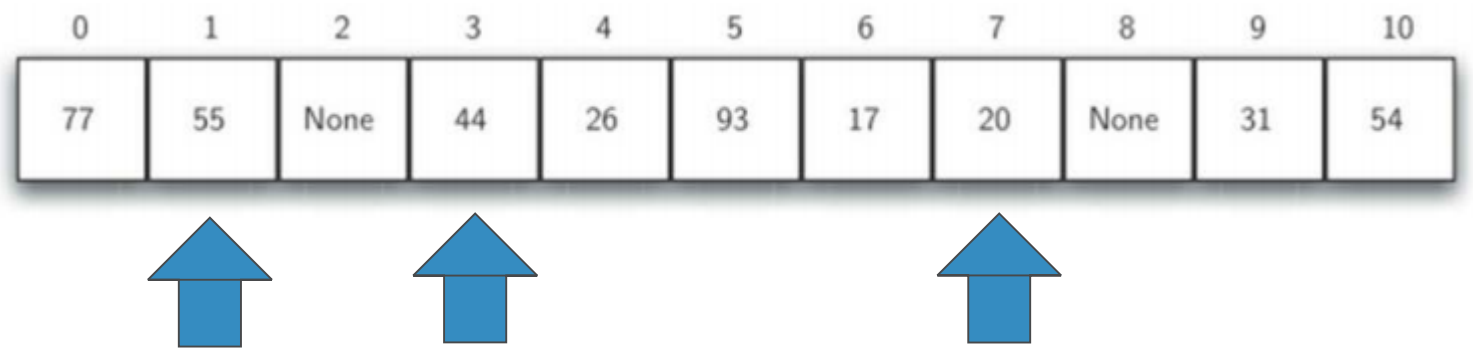
Collision solution: the improvement of linear probing
冲突解决方案：线性探测的改进

One way to avoid aggregation is to extend the linear probing, from one by one to jumping probing

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

The following figure shows the "+ 3" probing insertion 44,55,20

下图是“+3”探测插入44、55、20



Collision solution: rehashing again

冲突解决方案：再散列

The process of refinding empty grooves can be done with a more versatile way “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

newhashvalue =rehash (oldhashvalue), for linear probesrehash (pos)= (pos+1)%siz eoftable

newhashvalue=rehash(oldhashvalue)，对于线性探测来说，rehash(pos)=(pos+1)%sizeoftable

The "+ 3" jump probe is: rehash (pos) = (pos + 3)%sizeoftable

“+3”的跳跃式探测则是：rehash(pos)=(pos+3)%sizeoftable

The rehash formula for the jump detection is: reha sh (pos) = (pos + skip)%sizeoftable

跳跃式探测的再散列通式是：rehash(pos)=(pos+skip)%sizeoftable

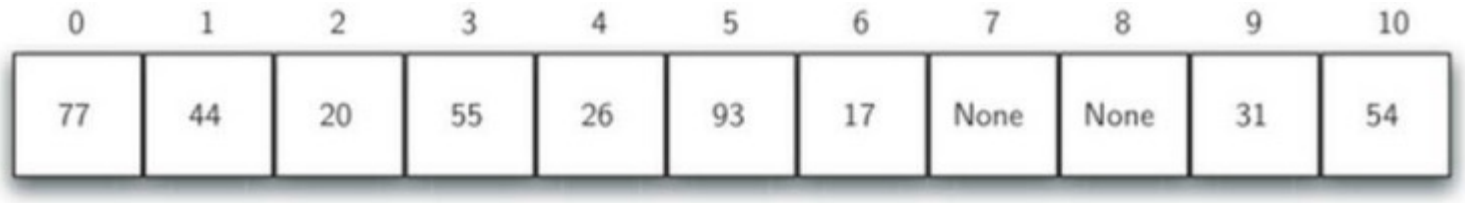
Collision solution: rehashing again
冲突解决方案：再散列

In jumping probing, it is important to note that the value of skip cannot be divisible by the hash table, otherwise it will produce cycles, resulting in many empty grooves can never be detected

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

One trick is to set the size of the hash table to the prime, as 11 in the example

一个技巧是，把散列表的大小设为素数，如例子的11

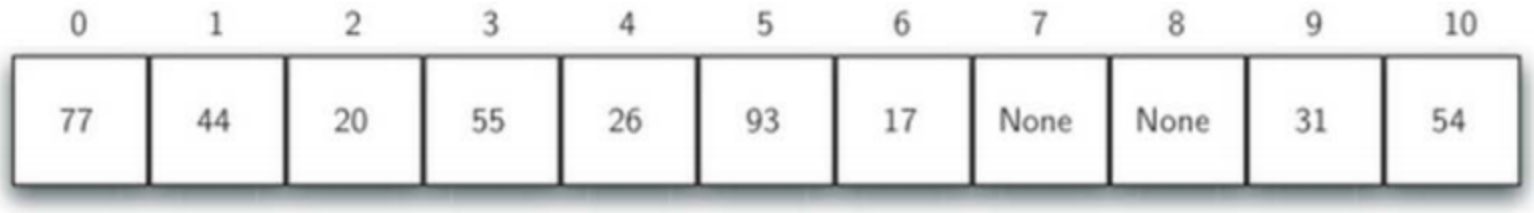


Collision solution: rehashing again
冲突解决方案：再散列

The linear probe can also be changed into a "quadratic probing"
还可以将线性探测变为“二次探测”

Instead of fixing the value of the skip, it will gradually increase the skip's values, such as 1,3,5,7, and 9
不再固定skip的值，而是逐步增加skip值，如1、3、5、7、9

In this way, the slot number will be the original hash value that increased by the square number: h, h + 1, h + 4, h + 9, h + 16...
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...



Conllision solution: Data Chaining

冲突解决方案：数据项链Chaining

In addition to the open addressing technology to finding empty slots, another solution to solve hash collisions is to expand a slot for individual data items to a collection of data items (or a reference to a linked list of data)

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)

This way, each slot in the hash table can accommodate multiple data items, and if a hash collision occurs, simply add the data items to the data item collection.

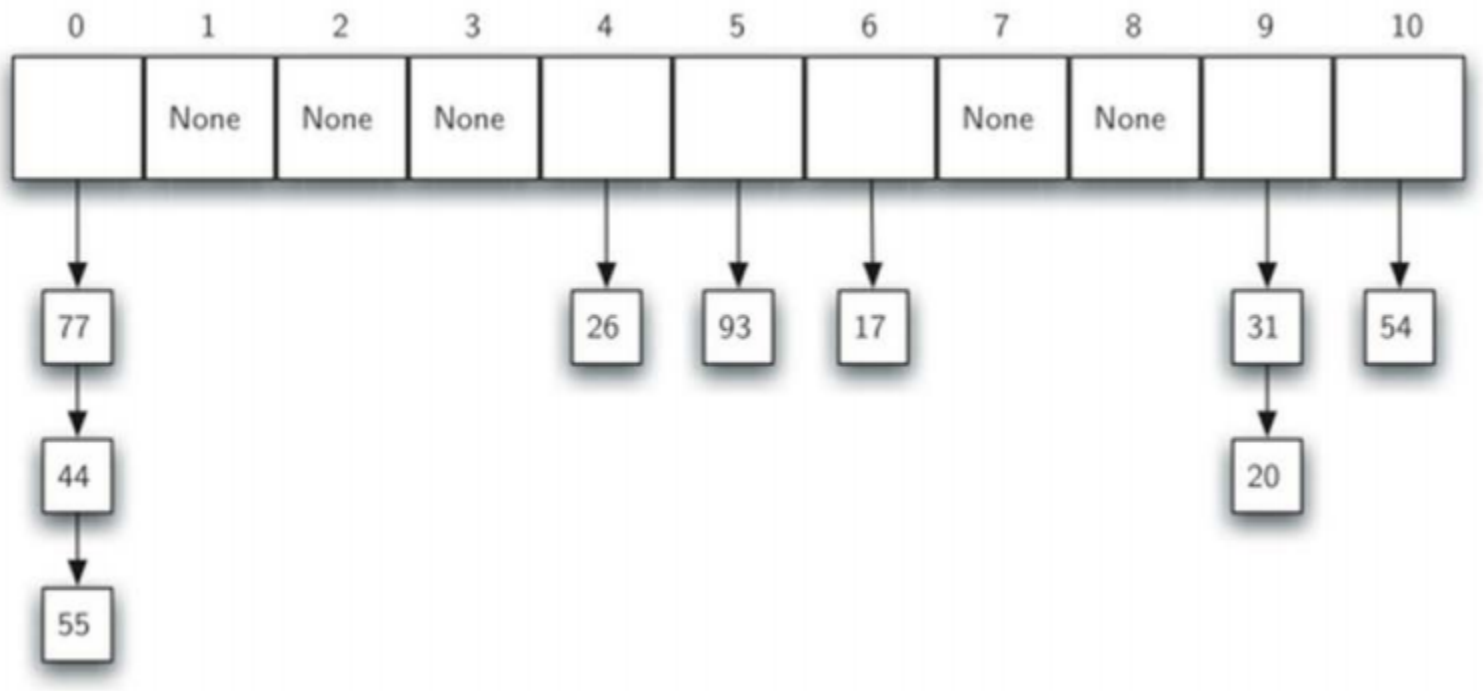
这样，散列表中的每个槽就可以容纳多个数据项，如果有散列冲突发生，只需要简单地将数据项添加到数据项集合中。

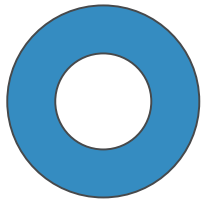
When you find data items, you need to find the entire collection in the same slot, and of course, as the hash collisions increases, the search time for data items increases accordingly.

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。

Collision solution: Data Chaining

冲突解决方案：数据项链





the abstract data types“Map ” and the Python implementation

映射抽象数据类型及Python实现

Abstract data type “Map”: ADT Map
抽象数据类型“映射”:ADT Map

One of the most useful data types of Python is a "dictionary"

Python最有用的数据类型之一“字典”

A dictionary is a data type that can hold key-data value pairs

字典是一种可以保存key-data键值对的数据类型

Where the key can be used to query the associated data value

其中关键码key可用于查询关联的数据值

This method of key-value association is called "Map"

这种键值关联的方法称为“映射Map”

The structure of the ADT Map is an unordered set of key-value associations

ADTMap的结构是键-值关联的无序集合

Key codes are unique

关键码具有唯一性

A data value can be uniquely determined by the key

通过关键码可以唯一确定一个数据值

Abstract data type, Map: ADT Map
抽象数据类型“映射”:ADTMap

The ADT Map operation is defined as follows:

ADTMap定义的操作如下：

①Map (): Create an empty mapping that returns the empty mapping object;

Map()：创建一个空映射，返回空映射对象；

②The put (key, val): Add the key-val association pair to the mapping, and replace the old association value with val if the key already exists;

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；

③The get(key): Given the key, returns the associated data value and, if not present, returns the None;

get(key)：给定key，返回关联的数据值，如不存在，则返回None；

④The del: Delete a key-val association as a statement of delmap [key];

del：通过delmap[key]的语句形式删除key-val关联；

⑤The len (): returns the number of key-val associations in the mapping;

len()：返回映射中key-val关联的数目；

⑥Keyinmap: Returns whether key exists in the association as a statement of keyinmap　　keyinmap：通过keyinmap的语句形式，返回key是否存在于关联中

Implement the ADT Map
实现ADTMap

The advantage of using the dictionary is that, given the key, the associated data value data is quickly obtained

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

To achieve the goal of fast finding, an ADT implementation supporting efficient finding is needed

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

- You can use the list data structure plus sequential search or binary search

  可以采用列表数据结构加顺序查找或者二分查找

- Of course, it is more appropriate to use the hash table mentioned before, so that the search can achieve the fastest O(1) performance

  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

Implement the ADT Map
实现ADTMap

Below, we implement ADT Map with a HashTable class that contains two lists as members

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

- One of the slot lists is used to save the key

  其中一个slot列表用于保存key

- Another parallel data list is used to save the data items

  另一个平行的data列表用于保存数据项

After the slot list finds the location of a key, the data item in the same position in the data list is the associated data

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

Implementing ADT Map: Application instance

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

```
>>>
[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
```

Implement the ADT Map
实现ADTMap

The list that saves the key is processed as a hash table, so that the key can be quickly found

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

Note the size of the hash table, although it can be any number, it should be a prime number, considering that the collision solution for the algorithm can work effectively.

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
```

Implement the ADT Map: put method code

实现ADTMap：put方法代码

The hashfunction method uses a simple reminder method to realize the hash function, while the collision solution uses the linear probe "plus 1" rehash function.

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。

The key does not exist, and it does not conflict
key不存在，未冲突

The key already exists, replace the val
key已存在，替换val

Hash collides, and then rehash until the empty slot or key is found
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)

      nextslot = self.rehash(nextslot)

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

Implementing the ADT Map: get method

实现ADTMap：get方法

Tag hash value as the start of the search
标记散列值为查找起点

search for key, until the empty slot or back to the start point
找key，直到空槽或回到起点

No key is found, rehash and continue the search
未找到key，再散列继续找

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

    data = None
    stop = False
    found = False
    position = startslot
    while self.slots[position] != None and  \
                         not found and not stop:
        if self.slots[position] == key:
            found = True
            data = self.data[position]
        else:
            position=self.rehash(position)
            if position == startslot:
                stop = True
    return data
```

Back to the starting point, stop
回到起点，停

Implement ADTMap: Additional code
实现ADTMap：附加代码

Implement access by special methods
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)
```

Hash-based algorithm analysis
散列算法分析

At best, hash provides searching performance with O (1) as constant time complexity

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

Due to hash collisions, searching comparisons are not that simple

由于散列冲突的存在，查找比较次数就没有这么简单

The most important information to assessing hash collision is the load factor λ, in general:

评估散列冲突的最重要信息就是负载因子λ，一般来说：

If it is small, the odds of hash collision are small, and data items are usually saved in the hash slot to which they belong

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

If larger, the hash table filling is full, more collides and more complex collision solution, and more comparisons are needed to find empty slots; if the data chain is used, it means more data items on each chain

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

Hash-based algorithm analysis
散列算法分析

If using a open addressing method of linear probe to resolve collision (between 0 and 1)

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)

Successful search, the average number of comparisons is:

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

Unsuccessful search, the average number of comparisons is:

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

If the data chain is used to resolve collisions (λ is greater than 1)

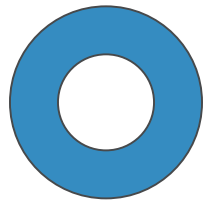
如果采用数据链来解决冲突(λ可大于1)

For a successful search, the average number of comparing times is: 1 + λ/ 2

成功的查找，平均需要比对次数为：1+λ/2

Unsuccessful search, the average number of comparing times is: λ

不成功的查找，平均比对次数为：λ



Summary of Sort and Search

排序与查找小结

chapter summary
本章小结

Sequential search on an unordered list or an ordered list has a time complexity of O(n)

在无序表或者有序表上的顺序查找，其时间复杂度为O(n)

The worst time complexity of binary search on an ordered list is $O(\log_2 n)$

在有序表上进行二分查找，其最差复杂度为 $O(\log_2 n)$

The hash table can achieve a constant level of time complexity

散列表可以实现常数级时间的查找

Perfect hash function is widely used as a data consistency verification

完美散列函数作为数据一致性校验，应用很广

Blockchain technology is a decentralized, distributed database that works through a "proof of work" mechanism

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

chapter summary
本章小结

Bubble, selection, and insertion sorting are the algorithm with time complexity of $O(n^2)$

冒泡、选择和插入排序是$O(n^2)$的算法

Shell sort is improved on the basis of insertion sort, using the method of sorting increasing sub-tables, and its time complexity can be between $O(n)$ and $O(n^2)$

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

The time complexity of merg sorting is O(nlogn), but the merging process requires additional storage space

归并排序的时间复杂度是O(nlogn)，但归并的过程需要额外存储空间

The best time complexity for quick sorting is O (nlogn) and does not require additional storage space, but the worst situation will be $O(n^2)$ if the split point deviates from the center of the list

快速排序最好的时间复杂度是O(nlogn)，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，最坏情况下会退到$O(n^2)$

Algorithm choice: a letter from a classmate
算法的选择：一封同学来信

An improved version is mentioned in the bubble sort algorithm
冒泡排序算法中提到一个对其改进的算法

Using the "short circuit" feature of "if no exchange, in order", the invalid comparison will be eliminated, and the algorithm itself should be faster

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
```

Algorithm choice: a letter from a classmate
算法的选择：一封同学来信

However, some careful students did the experiment by themselves, and the results were not very optimistic:
但有细心的同学自己动手做了试验，结果不太乐观：

I tried the sorting algorithm
我试了一下排序算法
shortBubbleSort is much slower than BubbleSort (each set of data is run 10 times averagely)
发现shortBubbleSort比BubbleSort慢很多啊(每组数据平均运行10次)
The algorithm is written on the slide. Why?
算法就是照着幻灯片上写的，为什么呢？

Algorithm choice: a letter from a classmate

算法的选择：一封同学来信

First-time data:

第一次数据：

alist=[iforiinrange(10)forjinrange(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}

Second time data:

第二次数据：

alist=[iforiinrange(100)forjinrange(100)](再用shuffle打乱)

{'Sort':0.0,'shuffle':0.0,'SelectionSort':2.8600001335144043,'InsertionSort':3.827999830245 9717,'BubbleSort':7.0,'ShortBubbleSort':7.921999931335449,'ShellSort':0.0470001697540283 2}

Third time of data:

第三次数据：

alist=[iforiinrange(1000)forjinrange(100)](再用shuffle打乱)

{'Sort':0.031000137329101562,'shuffle':0.031000137329101562,'SelectionSort':297.72199988365 173,'InsertionSort':397.25999999046326,'BubbleSort':765.3139998912811,'ShortBubbleSort':847.1 009998321533,'ShellSort':0.6410000324249268}

(Running for one day) (运行了一天)

Algorithm choice: a letter from a classmate
算法的选择：一封同学来信

Why are theory far from the practice?

为什么理论和实际相差甚远？

Can you still happily choose a good algorithm?

到底还能不能愉快地选择一个好算法了？

The "short circuit" advantage of the ShortBubbleSort is highly dependent on the initial layout of the data

ShortBubbleSort的“短路”优势高度依赖于数据的初始布局

If the data layout is so stochastic that every comparison is exchanged, the ShortBubble Sort has no advantage at all; it costs an additional exchanges variable and the corresponding assignment sentence, which is slower than the original bubble sort.

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。

Algorithm choice: a letter from a classmate

算法的选择：一封同学来信

There is no problem with the test code, the problem is the object during the sorting. Its data is out of order through random.shuffle

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

First of all, *alist* is nested with *range*, the generated data is arranged from small to large, which is very neat;

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；

And this shuffle will try to disrupt the data to the most chaotic degree, resulting in a high degree of random data layout;

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；

In this way, the short-circuit characteristic of ShortBubbleSort will be completely invalid; besides, the cost of exchanges variables will be paid. The actual measurement is much slower than the original bubble sorting algorithm.

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。

Algorithm choice: a letter from a classmate
算法的选择：一封同学来信

So sometimes there are no absolute pros or cons to sorting algorithms, especially those with the same time complexity
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

In order to achieve the highest sorting performance in a specific applicable situation, we also need to analyze the data itself, and select the corresponding sorting algorithm according to the characteristics of the data
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

Algorithm choice: a letter from a classmate

算法的选择：一封同学来信

In addition to time complexity, sometimes spatial complexity is also a key factor to consider

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

- Merge sort’s time complexity is O(nlogn), but requires an additional double storage space

  归并排序时间复杂度O(nlogn)，但需要额外一倍的存储空间

  The best time complexity of quick sort is O (nlogn), and there is no additional storage space, but the "median" selection is the key to performance. And in the extreme case of poor selection, the performance is even lower than bubble sorting

  快速排序时间复杂度最好的情况是O(nlogn)，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

Algorithmic choice is not an absolute judgment of advantages and disadvantages, we need to consider all aspects of factors

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

- Includes running environment requirements, and attributes of data objects to be processed

  包括运行环境要求、处理数据对象的特性