Algorithm analysis

算法分析

Compare programs, or compare algorithms?
对比程序，还是算法？

How to compare two different pragrams?
如何对比两个程序？

Looks different, but which program solves the same problem "better"?

看起来不同，但解决同一个问题的程序，哪个“更好”？

Difference Between Program and Algorithm：
程序和算法的区别：

An algorithm is a step-by-step description of problem solving

算法是对问题解决的分步描述

A program is an algorithm implemented in a certain programming language. The same algorithm can generate many programs through different programmers using different programming languages.

程序则是采用某种编程语言实现的算法，同一个算法通过不同的程序员采用不同的编程语言，能产生很多程序

cumulative sum problem
累计求和问题

Let's look at a program that completes the accumulation from 1 to n and then outputs the sum

我们来看一段程序，完成从1到n的累加，输出总和

set cumulative variable = 0　　设置累计变量=0

Loop from 1 to n, accumulating successively to the accumulating variable

从1到n循环，逐次累加到累计变量

return cumulative variable

返回累计变量

```
def sumOfN(n):
    theSum = 0
    for i in range(1, n + 1):
        theSum = theSum + i
    return theSum

print(sumOfN(10))
```

cumulative sum problem
累计求和问题

Looking at the second program, does it seem weird?

再看第二段程序 ，是否感觉怪怪的？

But in fact, the function of this program is the same as the previous paragraph 但实际上本程序功能与前面那段相同

This program fails because the variable names are unintelligible and contains useless junk code

这段程序失败之处在于：变量命名词不达意，以及包含了无用的垃圾代码

```
def foo(tom):
    fred = 0
    for bill in range(1, tom + 1):
        barney = bill
        fred = fred + barney
    return fred


print(foo(10))
```

The concept of algorithm analysis
算法分析的概念

Comparing the "good and bad" of a program, there are more factors
比较程序的“好坏”，有更多因素

Code style, readability, etc. 代码风格、可读性等等

We are mainly interested in the properties of the algorithm itself
我们主要感兴趣的是算法本身特性

Algorithm analysis is mainly to evaluate and compare algorithms from the perspective of computing resource consumption 算法分析主要就是从计算资源消耗的角度来评判和比较算法

Algorithms that use computing resources more efficiently, or use less computing resources, are good algorithms
更高效利用计算资源，或者更少占用计算资源的算法，就是好算法

From this perspective, the above two programs are basically the same, as they both use the same algorithm to solve the cumulative sum problem
从这个角度，前述两段程序实际上是基本相同的，它们都采用了一样的算法来解决累计求和问题

When it comes to code style and readability
说到代码风格和可读性

Why is Python's forced indentation good?
为什么Python的强制缩进是好的？

Sentences block function and visual effects are unified

语句块功能和视觉效果统一

A low-level bug in Apple　　苹果公司的一个低级Bug

Due to the negligence of indentation alignment in C language code writing
由于C语言源代码书写缩进对齐的疏忽

SSL connection verification is skipped

造成SSL连接验证被跳过

Revised iOS7.0.6 on 2014.2.22

2014.2.22修正iOS7.0.6

When it comes to code style and readability
说到代码风格和可读性

Code doesn't work as it seems
代码不像看起来那样运行

```
1  static OSStatus
2  SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedPa
3                                   uint8_t *signature, UInt16 signatureLen)
4  {
5      OSStatus        err;
6      ...
7
8      if ((err = SSLHashSHA1.update(&h
9          goto fail;
10     if ((err = SSLHashSHA1.update(&h
11         goto fail;
12         goto fail;
13     if ((err = SSLHashSHA1.final(&ha
14         goto fail;
15     err = sslRawVerify(ctx,
16                        ctx->peerPubK
17                        dataToSign,
18                        dataToSignLen,
19                        signature,
20                        signatureLen);
21     if(err) {
22         sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
23                     "returned %d\n", (int)err);
24         goto fail;
25     }
26
27 fail:
28     SSLFreeBuffer(&signedHashes);
29     SSLFreeBuffer(&hashCtx);
30     return err;
31 }
```

```
9          goto fail;
10     if ((err = SSLHashSH
11         goto fail;
12         goto fail;
13     if ((err = SSLHashSH
14         goto fail;
15     err = sslRawVerify(c
16                        c
```

Computing Resource Metrics
计算资源指标

What is computing resource then?

那么何为计算资源？

One is the storage space or memory required by the algorithm to solve the problem

一种是算法解决问题过程中需要的存储空间或内存

However, the storage space is affected by the change of the data scale of the problem itself.

但存储空间受到问题自身数据规模的变化影响

It is not easy to distinguish which storage space is required by the description of the problem itself and which is occupied by the algorithm

要区分哪些存储空间是问题本身描述所需，哪些是算法占用，不容易

Computing Resource Metrics
计算资源指标

What is computing resource then?
那么何为计算资源？

The other is the execution time of the algorithm

另一种是算法的**执行时间**

We can test the actual running of the program and get the real running time

我们可以对程序进行实际运行测试，获得真实的运行时间

runtime detection
运行时间检测

There is a time module in Python that can get the current time of the computer system

Python中有一个time模块 ，可以获取计 算机系统当前时间

The running time can be obtained by recording the system time of the algorithm starting time and ending time.

在算法开始前和结束后分别记录系统时间，即可得到运行时间

```
>>> help(time.time)
Help on built-in function time in module time:

time(...)
    time() -> floating point number

    Return the current time in seconds since the Epoch.
    Fractions of a second may be present if the system clock provides them.
```

```
>>> import time
>>> time.time()
1565878560.88039
>>>
```

runtime detection
运行时间检测

Runtime detection of cumulative summation programs
累计求和程序的运行时间检测

Use time to detect total running time　　用time检测总运行时间

Return cumulative sum, and running time (seconds)
返回累计和，以及运行时间(秒)

```
1   import time
2
3
4   def sumOfN2(n):
5       start = time.time()
6       theSum = 0
7       for i in range(1, n + 1):
8           theSum = theSum + i
9       end = time.time()
10      return theSum, end - start
11
12
13  for i in range(5):
14      print("Sum is %d required %10.7f seconds"
15            % sumOfN2(10000))
```

runtime detection
运行时间检测

Run 5 times in a row in the interactive window to see
在交互窗口连续运行5次看看

add cumulatively from 1 to 10,000　　1到10,000累加
each run takes about 0.0007 seconds　　每次运行约需0.0007秒

```
Sum is 50005000 required  0.0007980 seconds
Sum is 50005000 required  0.0007021 seconds
Sum is 50005000 required  0.0007031 seconds
Sum is 50005000 required  0.0007219 seconds
Sum is 50005000 required  0.0007060 seconds
```

runtime detection
运行时间检测

If it adds up to 100,000? 如果累加到100,000？

It looks like the runtime increases to 10 times of the 10,000

看起来运行时间增加到10,000的10倍

```
Sum is 5000050000 required  0.0078530 seconds
Sum is 5000050000 required  0.0078511 seconds
Sum is 5000050000 required  0.0087960 seconds
Sum is 5000050000 required  0.0082700 seconds
Sum is 5000050000 required  0.0077040 seconds
```

Add up further to 1,000,000? 进一步累加到1,000,000？

The runtime is 10 times longer than 100,000 again

运行时间又是100,000的10倍了

```
Sum is 500000500000 required  0.0817859 seconds
Sum is 500000500000 required  0.0781529 seconds
Sum is 500000500000 required  0.0803380 seconds
Sum is 500000500000 required  0.0783160 seconds
Sum is 500000500000 required  0.0776238 seconds
```

The second iteration-free accumulation algorithm
第二种无迭代的累计算法

Iteration-free algorithm using summation formula
利用求和公式的无迭代算法

```
17      def sumOfN3(n):
18          start = time.time()
19          theSum = (n * (n + 1)) / 2
20          end = time.time()
21          return theSum, end - start
```

Use the same method to check the running time
采用同样的方法检测运行时间

10,000; 100,000; 1,000,000
10,000,000; 100,000,000

```
Sum is 50005000 required  0.0000010 seconds
Sum is 5000050000 required  0.0000000 seconds
Sum is 500000500000 required  0.0000010 seconds
Sum is 50000005000000 required  0.0000000 seconds
Sum is 500000050000000 required  0.0000169 seconds
```

The second iteration-free accumulation algorithm
第二种无迭代的累计算法

Two points to pay attention to:

需要关注的两点:

The running time of this algorithm is much shorter than the previous one

这种算法的运行时间比前种都短很多

The running time has nothing to do with the size of the accumulated object n (the former algorithm is a multiple growth relationship)

运行时间与累计对象n的大小没有关系(前种算法是倍数增长关系)

The running time of the new algorithm is almost independent of the number that needs to be accumulated

新算法运行时间几乎与需要累计的数目无关

Analysis of runtime detection
运行时间检测的分析

Look at the first iterative algorithm
观察一下第一种迭代算法

contains a loop that may execute more sentences

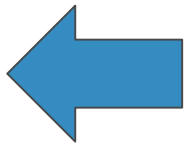
包含了一个循环，可能会执行更多语句

The number of times this loop runs is related to the accumulated value n. As n increases, the number of loops also increases.

这个循环运行次数跟累加值n有关系，n增加，循环次数也增加

```
def sumOfN(n):
    theSum = 0
    for i in range(1, n + 1):
        theSum = theSum + i
    return theSum

print(sumOfN(10))
```



Analysis of runtime detection
运行时间检测的分析

But about the actual detection of running time, there is a bit of a problem

但关于运行时间的实际检测，有点问题

About programming languages and running environments

关于编程语言和运行环境

The same algorithm, written in different programming languages and running on different machines, will get different running times, sometimes very different:

同一个算法，采用不同的编程语言编写，放在不同的机器上运行，得到的运行时间会不一样，有时候会大不一样：

For example, running a non-iterative algorithm on an old machine may even be slower than an iterative algorithm on a new machine

比如把非迭代算法放在老旧机器上跑，甚至可能慢过新机器上的迭代算法

Analysis of runtime detection
运行时间检测的分析

Thus, we need better ways to measure algorithm running time

我们需要更好的方法来衡量算法运行时间

This indicator has nothing to do with specific machines, programs, and operating periods.

这个指标与具体的机器、程序、运行时段都无关

Recall Big O Notation

回顾 大O表示法

Algorithmic Time Metrics
算法时间度量指标

The number of operations or steps performed by an algorithm can be used as an independent program/machine metric

一个算法所实施的操作数量或步骤数可作为独立于具体程序/机器的度量指标

Which operation is irrelevant to the specific implementation of the algorithm?

哪种操作跟算法的具体实现无关？

Requires a common base operation as a unit of measure for running steps

需要一种通用的基本操作来作为运行步骤的计量单位

An assignment statement is an appropriate choice

赋值语句是一个合适的选择

Algorithmic Time Metrics
算法时间度量指标

An assignment statement contains both (expression) evaluation and (variable)

一条赋值语句同时包含了(表达式)计算和(变量)

store two basic resources　　存储两个基本资源

Carefully observe the characteristics of programming language, in addition to the definition statement which has nothing to do with computing resources, there are mainly three kinds of control flow statement and assignment statement. The control flow only plays the role of organizing the statement, does not implement processing.

仔细观察程序设计语言特性，除了与计算资源无关的定义语句外，主要就是三种控制流语句和赋值语句，而控制流仅仅起了组织语句的作用，并不实施处理。

Assignment statement execution times
赋值语句执行次数

Analyze the execution times of the assignment statement of SumOfN

分析SumOfN的赋值语句执行次数

For "problem scale" n　　对于“问题规模”n

Number of assignment statements T(n)=1+n　　赋值语句数量T(n)=1+n

So, what is the problem scale?　　那么，什么是问题规模？

```
31  def sumOfN(n):
32      theSum = 0
33      for i in range(1, n + 1):
34          theSum = theSum + i
35      return theSum
```

Problem scale affects algorithm execution time
问题规模影响算法执行时间

Problem scale: the main factor that affects the execution time of an algorithm

问题规模：影响算法执行时间的主要因素

In the algorithm for the cumulative sum of the first n integers, the number of integers that need to be accumulated is suitable as an indicator of the problem scale

在前n个整数累计求和的算法中 ，需要累计的整数个数合适作为问题规模的指标

The sum of the first 100,000 integers is a larger scale of the same problem compared to the sum of the first 1,000 integers

前100,000个整数求和对比前1,000个整数求和，算是同一问题的更大规模

The goal of algorithm analysis is to find out how problem scale affects the execution time of an algorithm

算法分析的目标是要找出问题规模会怎么影响一个算法的执行时间

Order of Magnitude 数量级函数

The exact value of the basic operand function T(n) is not particularly important, what matters is the dominant part of T(n) that is the decisive factor

基本操作数量函数T(n)的精确值并不是特别重要，重要的是T(n)中起决定性因素的主导部分

From a dynamic perspective, when the problem scale increases, some parts of T(n) will overwhelm the contribution of other parts

用动态的眼光看，就是当问题规模增大的时候，T(n)中的一些部分会盖过其它部分的贡献

The order of magnitude function describes the dominant part of T(n) that increases the fastest as n increases

数量级函数描述了T(n)中随着n增加而增加速度最快的主导部分

Called "Big O" notation, denoted as O(f(n)), where f(n) represents the dominant part in T(n)

称作“大O”表示法，记作O(*f*(*n*))，其中*f*(*n*) 表示T(n)中的主导部分

A method for determining the run-time order of magnitude Big-O 确定运行时间数量级大O的方法

Example 1: T(n)=1+n

As n increases, the constant 1 becomes less and less important in the final result

当n增大时，常数1在最终结果中显得越来越无足轻重

So we can remove 1 and keep n as the main part, and the running time is on the order of O(n)

所以可以去掉1，保留n作为主要部分，运行时间数量级就是O(n)

$$O(n)$$

A method for determining the run-time order of magnitude Big-O 确定运行时间数量级大O的方法

Example 2: $T(n)=5n^2+27n+1005$

When n is very small, the constant 1005 plays a decisive role
当n很小时，常数1005其决定性作用

But when n gets bigger and bigger, the n^2 item becomes more and more important, and the other two items have less and less influence on the result
但当n越来越大，n^2项就越来越重要，其它两项对结果的影响则越来越小

Similarly, the coefficient 5 in the n^2 term has little effect on the growth rate of n^2
同样，n^2项中的系数5，对于n^2的增长速度来说也影响不大

So you can remove 27n+1005 and the part of the coefficient 5 in the order of magnitude, and determine it as $O(n^2)$
所以可以在数量级中去掉27n+1005，以及系数5的部分，确定为$O(n^2)$

$$O(n^2)$$

Other Factors Affecting Algorithm Running Time
影响算法运行时间的其它因素

Sometimes it's not just the problem scale that determines the runtime

有时决定运行时间的不仅是问题规模

Some specific data can also affect the algorithm running time

某些具体数据也会影响算法运行时间

- Divided into best, worst and average cases, average case reflect the mainstream performance of the algorithm

  分为最好、最差和平均情况，平均状况体现了算法的主流性能

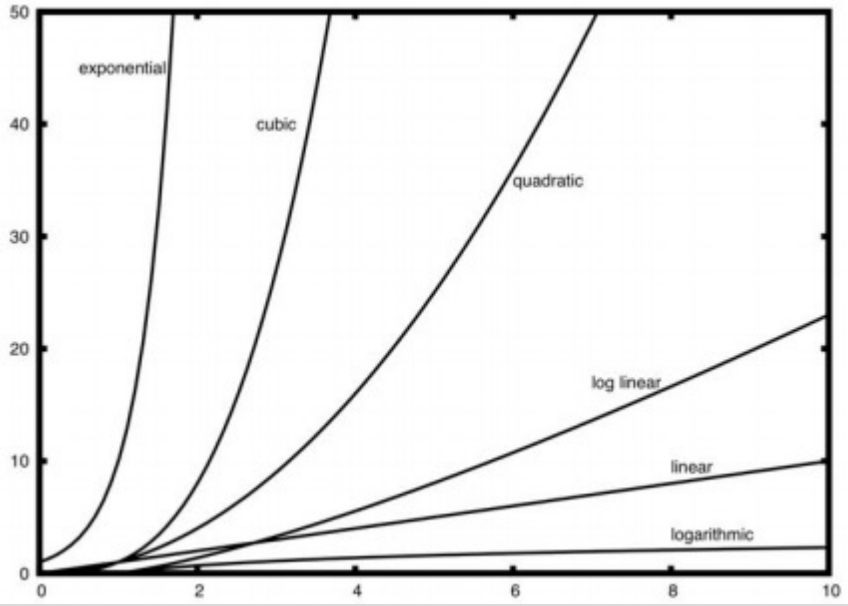
- The analysis of the algorithm should look at the mainstream, and not be confused by some specific operating conditions

  对算法的分析要看主流，而不被某几种特定的运行状况所迷惑

Common big-O order of magnitude functions
常见的大O数量级函数

Usually when n is small, it is difficult to determine its order of magnitude
通常当n较小时，难以确定其数量级

When n grows to be larger, it is easy to see its main changes
当n增长到较大时 ，容易看出其主要变化



$f(n)$	名称
1	常数
$Log(n)$	对数
n	线性
$n*Log(n)$	对数线性
n^2	平方
n^3	立方
2^n	指数

Determining Execution Time Order-of-magnitude Functions from Code Analysis 从代码分析确定执行时间数量级函数

The code assignment statement can be divided into 4 parts 代码赋值语句可以分为4个部分

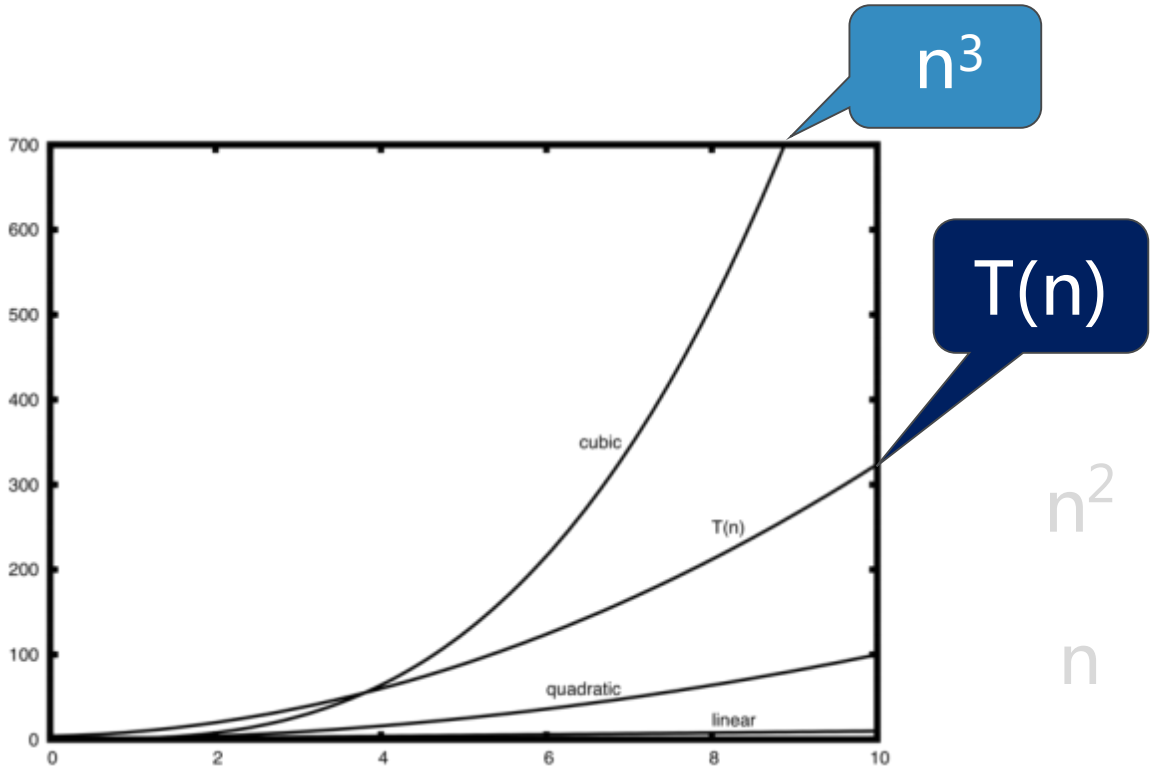
```
38  a = 5
39  b = 6
40  c = 10
41  for i in range(n):
42      for j in range(n):
43          x = i * i
44          y = j * j
45          z = i * j
46  for k in range(n):
47      w = a * k + 45
48      v = b * b
49  d = 33
```

$T(n) = 3+3n^2+2n+1$

$= 3n^2+2n+4$

Determining Execution Time Order-of-magnitude Functions from Code Analysis

从代码分析确定执行时间数量级函数



Keep only the highest-order term n^2 and remove all coefficients

仅保留最高阶项n^2 ，去掉所有系数

The order of magnitude is $O(n^2)$

数量级为$O(n^2)$

Anagram judgment

变位词判断问题

"Anagram" judgment problem
“变位词”判断问题

description
问题描述

The so-called "anagram" refers to the rearrangement of the constituent letters between two words

所谓“变位词”是指两个词之间存在组成字母的重新排列关系

Such as: heart and earth, python and typhon

如： heart和earth，python和typhon

For simplicity, it is assumed that the two words involved in judgment are solely of lower case letters and of equal length

为了简单起见，假设参与判断的两个词仅由小写字母构成，而且长度相等

Objective: Write a bool function with two words as parameters, return whether the two words are anagrams

解题目标：写一个bool函数，以两个词作为参数，返回这两个词是否变位词

It can show different orders of magnitude algorithms for the same problem

可以很好展示同一问题的不同数量级算法

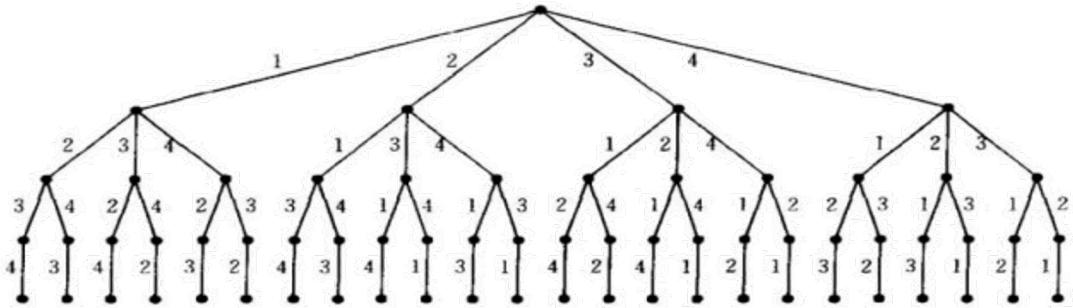
Solution 1: The way of brute force
解法1：暴力法

The idea of the way of brute force is: exhaust all possible combinations
暴力法解题思路为：穷尽所有可能组合

Arrange the characters that appear in s1 fully, and then see if s2 appears in the full arrangement list
将s1中出现的字符进行全排列 ，再查看s2是否出现在全排列列表中

The biggest difficulty here is to produce a full arrangement of all the characters in s1
这里最大困难是产生s1所有字符的全排列

According to the conclusion of combinatorial mathematics, if n characters are fully arranged, the number of all possible strings is n!
根据组合数学的结论，如果n个字符进行全排列 ，其所有可能的字符串个数为n！



Solution 1: The way of violence
解法1：暴力法

We know that n! grows even faster than 2^n

我们已知 n! 的增长速度甚至超过2^n

For example, the 20-character-long words, will result

例如，对于20个字符长的词来说，将产生

20! =2,432,902,008,176,640,000 candidates. If one candidate is disposed every microsecond, it takes nearly 80,000 years to complete all the matches.

20!=2,432,902,008,176,640,000个候选词.如果每微秒处理1个候选词的话，需要近8万年时间来做完所有的匹配。

Conclusion: The violence way may not be a good algorithm

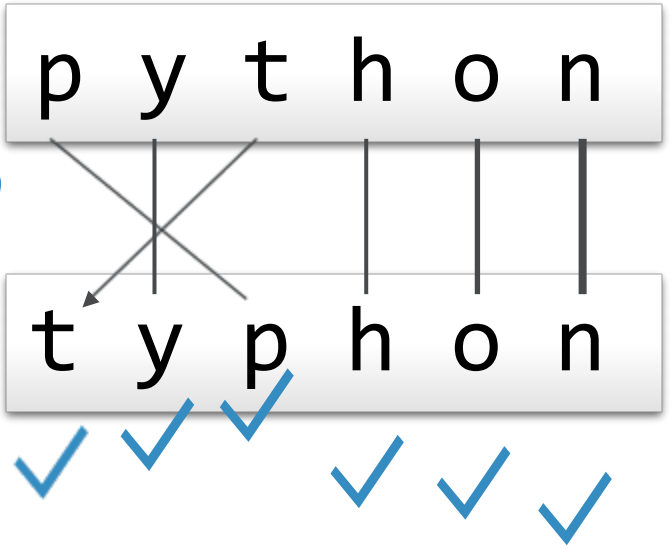
结论：暴力法恐怕不能算是个好算法

Solution 2: Check verbatim
解法2：逐字检查

Solution method idea
解法思路

Check the characters in word 1 individually to word 2

将词1中的字符逐个到词2中检查是否存在



tick mark if exists (to double checking for duplicates)

存在就“打勾”标记(重复检查)

If each character can be found, then the two words are anagrams

如果每个字符都能找到，则两个词是变位词

as long as one character is not found, it is not anagram

只要有1个字符找不到，就不是变位词

Solution 2: Check verbatim
解法2：逐字检查

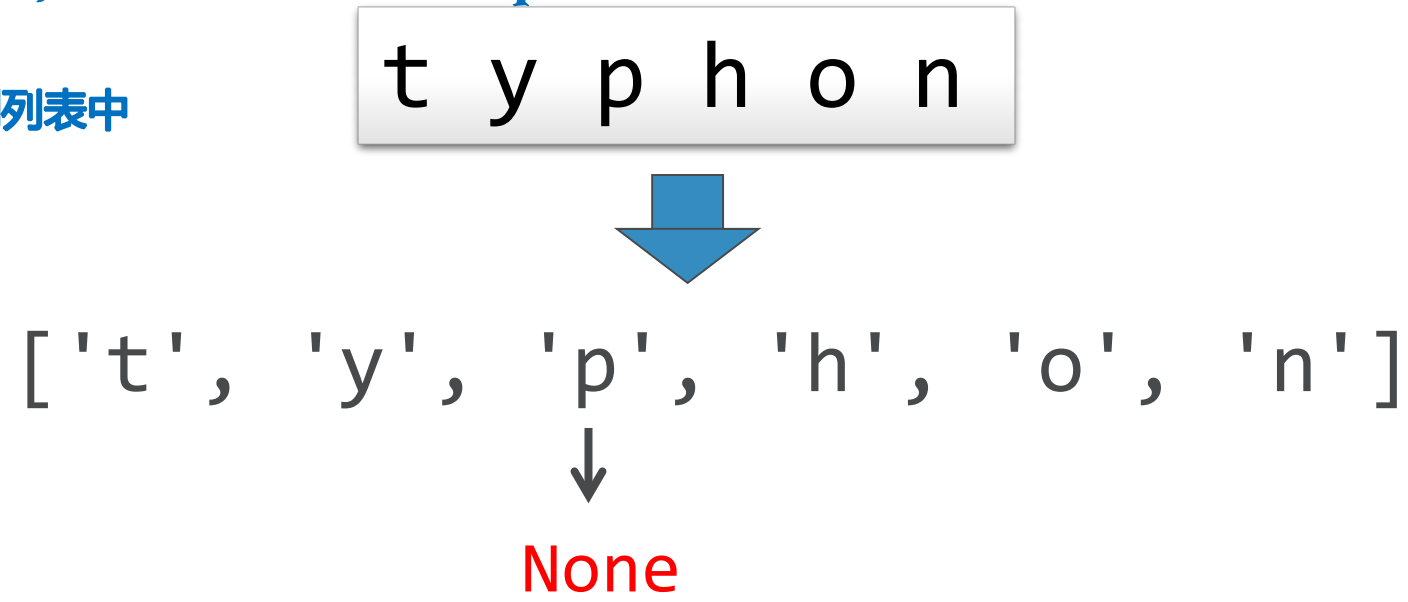
Program skills
程序技巧

Implement the tick mark: set the word 2 corresponding character to None

实现“打勾”标记：将词2对应字符设为None

Since the string is immutable, it needs to be copied to the list first

由于字符串是不可变类型，需要先复制到列表中



Solution 2: Check verbatim–program code

解法2：逐字检查–程序代码

```
1   def anagramSolution1(s1, s2):
2       alist = list(s2)
3       pos1 = 0
4       stillOK = True
5       while pos1 < len(s1) and stillOK:
6           pos2 = 0
7           found = False
8           while pos2 < len(alist) and not found:
9               if s1[pos1] == alist[pos2]:
10                  found = True
11              else:
12                  pos2 = pos2 + 1
13          if found:
14              alist[pos2] = None
15          else:
16              stillOK = False
17          pos1 = pos1 + 1
18      return stillOK
19
20
21  print(anagramSolution1('abcd', 'dcba'))
```

Copy s2 to the list
复制s2到列表

Each character of the loop s1
循环s1的 每个字符

One-by one contrast in s 2
在s2逐个 对比

Find, hook
找到，打勾

Not found, failed
未找到，失败

Solution 2: Check verbatim--algorithm analysis

解法2：逐字检查--算法分析

Problem scale: Number of characters included in the word(n)

问题规模：词中包含的字符个数n

The main part of the order is the double cycle

主要部分在于两重循环

The outer loop traverses s1 for each character, executing the inner loop for n times

外层循环遍历s1每个字符，将内层循环执行n次

The inner loop searches for characters in s2. The number of comparisons for each character is one of 1, 2...n, and they are different from each other.

而内层循环在s2中查找字符，每个字符的对比次数，分别是1、2…n中的一个，而且各不相同

So the total number of execution times is 1 + 2 + 3 + ……+n

所以总执行次数是1+2 +3 + …… +n

The order of magnitude is $O(n^2)$

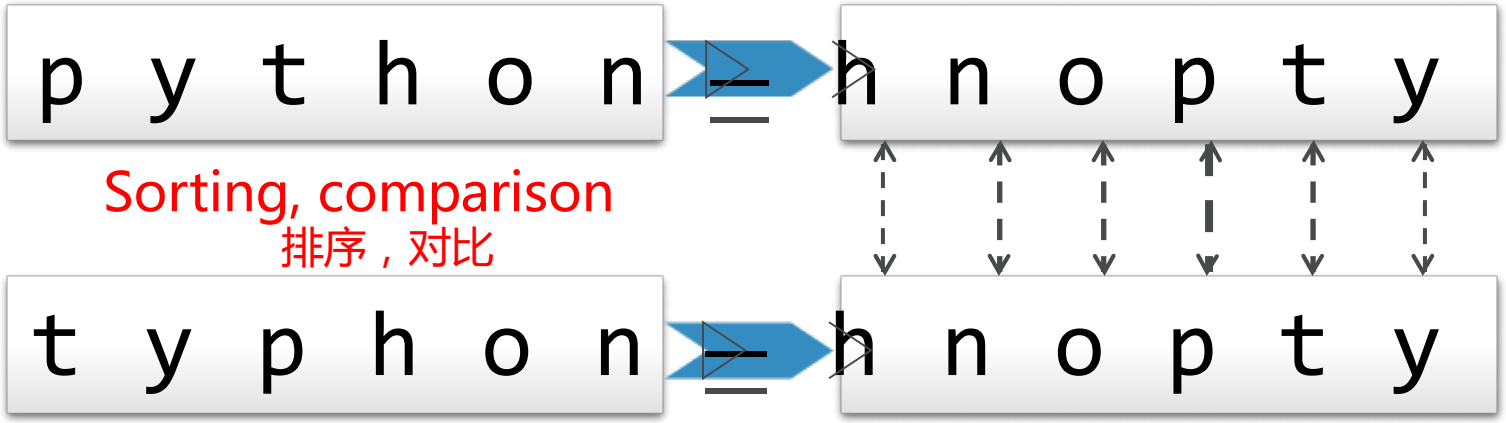
可知其数量级为$O(n^2)$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n \rightarrow 0(n2)$$

Solution 3: sort and comparison
解法3：排序比较

Problem solving ideas
解题思路

Order both strings in alphabetical order
将两个字符串都按照字母顺序排好序
Then compare whether it is the same character by character, if it is the same, it is an anagram
再逐个字符对比是否相同，如果相同则是变位词
If there is any difference, it is not an anagram
有任何不同就不是变位词



Solution 3: sort and comparison
解法3：排序比较

```
1   def anagramSolution2(s1, s2):
2       alist1 = list(s1)
3       alist2 = list(s2)
4
5       alist1.sort()
6       alist2.sort()
7       pos = 0
8       matches = True
9       while pos < len(s1) and matches:
10          if alist1[pos] == alist2[pos]:
11              pos = pos + 1
12          else:
13              matches = False
14      return matches
15
16
17  print(anagramSolution2('abcde', 'edcba'))
```

Turn to the list
转为列表

Sorting 分别排序

Contrast one by one 逐个对比

Solution method 3: sort comparison-algorithm analysis

解法3：排序比较-算法分析

At first glance, this algorithm has only one loop, which is executed at most n times, and the magnitude is O(n)

粗看上去，本算法只有一个循环，最多执行n次数，量级是O(n)

But the two sorts in front of the loop are not free

但循环前面的两个sort并不是无代价的

If you query the later chapter, you will find that the sorting algorithm uses a different solution, with a running time of almost O (n^2) Or O (n log n), larger than O(n)

如果查询下后面的章节，会发现排序算法采用不同的解决方案，其运行时间数量级差不多是$O(n^2)$或者O(nlogn)，大过循环的O(n)

So the time-leading step of this algorithm is the sorting step

所以本算法时间主导的步骤是排序步骤

The order of magnitude of the running time of this algorithm is equal to the order of magnitude of the sorting process O(nlogn)

本算法的运行时间数量级就等于排序过程的数量级O(nlogn)

Solution 4: Counting comparison
解法4：计数比较

Comparing the number of times each letter appears in the two words, if the 26 letters appear the same times, the two strings must be anagrams

解题思路：对比两个词中每个字母出现的次数 ，如果26个字母出现的次数都相同的 话 ，这两个字符串就一定是变位词

Specifical Way: Set up a 26-digit counter for each word, check each word first, and set how many times each letter appears in the counter

具体做法：为每个词设置一个26位的计数器，先检查每个词，在计数器中设定好每个字母出现的次数

After the counting stage is completed, enter the comparison stage, to see if the counters of the two strings are the same, if the same, to output the conclusion of the anagrams

计数完成后，进入比较阶段，看两个字符串的计数器是否相同，如果相同则输出是变位词的结论

Solution 4: Counting Comparison-program code

解法4：计数比较-程序代码

```
1   def anagramSolution4(s1, s2):
2       c1 = [0] * 26
3       c2 = [0] * 26
4       for i in range(len(s1)):
5           pos = ord(s1[i]) - ord('a')
6           c1[pos] = c1[pos] + 1
7       for i in range(len(s2)):
8           pos = ord(s2[i]) - ord('a')
9           c2[pos] = c2[pos] + 1
10      j = 0
11      stillOK = True
12      while j < 26 and stillOK:
13          if c1[j] == c2[j]:
14              j = j + 1
15          else:
16              stillOK = False
17      return stillOK
18
19
20  print(anagramSolution4('apple', 'pleap'))
```

Count
分别都计数

Counter comparison
计数器比较

Solution 4: Counting Comparison-algorithm analysis
解法4：计数比较-算法分析

There are 3 loop iterations in counting comparison, but not like
计数比较算法中有3个循环迭代，但不像

there is a nested loop in solution 3
解法3那样存在嵌套循环

The first two loops are used to count a string with operations equal to the string length n
前两个循环用于对字符串进行计数，操作次数等于字符串长度n

The third cycle was used for counter comparisons, always 26 times
第3个循环用于计数器比较，操作次数总是26次

So the total number of operations is T (n) =2n + 26, and the order of magnitude is O (n)
所以总操作次数T(n)=2n+26 ，其数量级为O(n)

This is a linear order of magnitude algorithm which has the best performance among four anagram-judgment algorithms
这是一个线性数量级的算法，是4个变位词判断算法中性能最优的

Solution 4: Counting Comparison-algorithm analysis
解法4：计数比较-算法分析

It is worth noting that this algorithm relies on two counter lists of length 26 to store character counts, which requires more storage space than the first 3 algorithms

值得注意的是，本算法依赖于两个长度为26的计数器列表，来保存字符计数，这相 比前3个算法需要更多的存储空间

If you consider words composed of large character sets (such as Chinese has tens of thousands of different characters), and more storage space is also needed.

如果考虑由大字符集构成的词(如中文具有上万不同字符)，还会需要更多存储空间。

To sacrifice storage space for running time, or, on the contrary, this trade-offs between time and space, often occurs in the process of choosing a problem solution.

牺牲存储空间来换取运行时间，或者相反，这种在时间空间之间的取舍和权衡，在选择问题解法的过程中经常会出现。

“不可随处小便”，“小处不可随便”

Performance for the Python Data Type

Python数据类型的性能

Performance of the Python data types
Python数据类型的性能

Earlier we learned about "big O notation" and the evaluation of different algorithms
前面我们了解了“大O表示法”以及对不同的算法的评估

Let's discuss the big-O order of magnitude of various operations on Python's two built-in data types
下面来讨论下Python两种内置数据类型上各种操作的大O数量级

- List and the dictionary
  列表list和字典dict
- These are two important Python data types, and later courses will be used to implement various data structures
  这是两种重要的Python数据类型，后面的课程会用来实现各种数据结构
- Run the tests to estimate the orders of magnitude of their various operational running times
  通过运行试验来估计其各种操作运行时间数量级

Contrast list and dict operations
对比list和dict的操作

type 类型	list	dict
index of matrix 索引	自然数i	不可变类型值key
add 添加	append、 extend、 insert	b[k]=v
delete 删除	pop、 remove*	pop
update 更新	a[i]=v	b[k]=v
Is to check 正查	a[i]、 a[i:j]	b[k]、 copy
Anti-check 反查	index(v)、 count(v)	无
else 其它	reverse、 sort	has_key、 update

List Datatypes
List列表数据类型

There are many ways to implement various list operations (interface). How to choose a concrete implementation method?

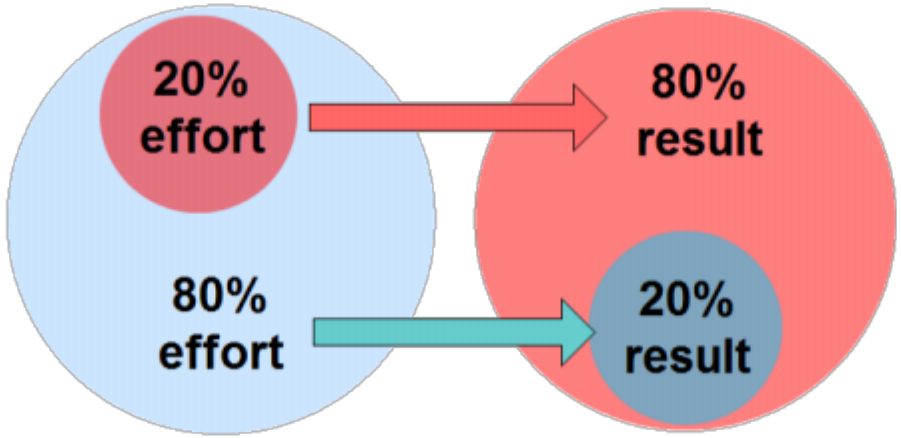
list类型各种操作 （interface） 的实现方 法有很多 ，如何选择具体哪种实现方法？

The overall solution is to make the most common operations perform the best and sacrifice the less common ones

总的方案就是 ，让最常用的操作性能最好 ，牺牲不太常用的操作

80 / 20 Rule: 80% of features use only 20%

80/20准则：80%的功能其使用率只有20%



List data type common operation performance

List列表数据类型常用操作性能

The most commonly used is: Get and assign by index(v= a [i], a [i] = v)

最常用的是：按索引取值和赋值（v= a[i]，a[i]= v)

Due to the random access properties of the list, both operation execution times are independent of the list size, both being O(1)

由于列表的随机访问特性，这两个操作执行时间与列表大小无关，均为O(1)

The other is the list growth, you can choose append () and __ add __() "+"

另一个是列表增长 ， 可以选择append() 和__add__() “ +”

lst.append (v), the execution time is O (1)

lst.append(v)，执行时间是O(1)

lst = lst + [v], the execution time is O (n + k), where k is the list length being added

lst= lst+ [v]，执行时间是O(n+k)，其中k是被加的列表长度

Choosing which method to manipulate the list determines the performance of the program

选择哪个方法来操作列表，决定了程序的性能

4 Methods to generate the list of the first n integers

4种生成前n个整数列表的方法

The first is the loop connection list (+)

首先是循环连接列表（+）方式生成

Secondly we can use the append method

然后是用append 方法添加元素生成

Then we can do it with **a list derivation**

接着用**列表推导式**来做

Finally we can use **range function call converted to the list**

最后是range函数调用转成列表

```
def test1():
    l = []
    for i in range(1000):
        l = l + [i]

def test2():
    l = []
    for i in range(1000):
        l.append(i)

def test3():
    l = [i for i in range(1000)]

def test4():
    l = list(range(1000))
```

Time the function using the timeit module
使用timeit模块对函数计时

To create a Timer object that specifies sentences that need to run repeatedly and Install sentences that only need to run once
创建一个Timer对象 ，指定需要反复运行的语句和只需要运行一次的“安装语句”

Then call the timeit method of this object, which can specify how many times to run repeatedly
然后调用这个对象的timeit方法 ，其中可以指定反复运行多少次

```
from timeit import Timer
t1= Timer("test1()", "from __main__ import test1")
print "concat %f seconds\n" % t1.timeit(number= 1000)

t2= Timer("test2()", "from __main__ import test2")
print "append %f seconds\n" % t2.timeit(number= 1000)

t3= Timer("test3()", "from __main__ import test3")
print "comprehension %f seconds\n" % t3.timeit(number= 1000)

t4= Timer("test4()", "from __main__ import test4")
print "list range %f seconds\n" % t4.timeit(number= 1000)
```

4 ways to generate a list of first n integers timing
4种生成前n个整数列表的方法计时

We see that the 4 methods have very different runtimes
我们看到 ，4种方法运行时间差别很大

List connection (concat) is the slowest, and List range is the fastest, with a speed difference of nearly 200 times.
列表连接(concat) 最慢， List range最快， 速度相差近200倍。

append is also much faster than concat
append也要比concat快得多

Also, We noticed that the list comprehension is twice as fast as append
另外，我们注意到列表推导式速度是append两倍的样子

```
>>>
concat 1.889487 seconds

append 0.091561 seconds

comprehension 0.038418 seconds

list range 0.009710 seconds
```

Big-O order of magnitude for basic List operations

List基本操作的大O数量级

Operation	Big-O Efficiency
index []	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O(n+k)
reverse	O(n)
concatenate	O(k)
sort	O(n log n)
multiply	O(nk)

Timing test for list.pop
list.pop的计时试验

We note that pop operation
我们注意到pop这个操作

The pop() removes the element from the end of the list, O (1)

pop()从列表末尾移除元素，O(1)

pop(i) remove element from middle of list, O(n)

pop(i)从列表中部移除元素，O(n)

The reason lies in the implementation method chosen by the Python
原因在于Python所选择的实现方法

To remove an element from the middle, we should remove the element behind that one first

从中部移除元素的话，要把移除元素后面的元素

Then copy all and move them forward, which seems awkward, but this implementation ensures that the list is index and assigned quickly to O (1)

全部向前挪位复制一遍，这个看起来有点笨拙，但这种实现方法能够保证列表按索引取值和赋值的操作很快，达到O(1)

This is also a compromise for both common and uncommon operations

这也算是一种对常用和不常用操作的折衷方案

Timing test for list.pop
list.pop的计时试验

In order to verify the big-O order of magnitude in the table, we compare the pop operation in both cases for actual timing contrast

为了验证表中的大O数量级 ，我们把两种情况下的pop操作来实际计时对比

For a list of the same size, call pop() and pop(0) respectively

相对同一个大小的list，分别调用pop()和pop(0)

Time list of different sizes, the expected result is

对不同大小的list做计时，期望的结果是

The time of pop() does not change with the list size, and the time of pop(0) becomes longer with the list size

pop()的时间不随list大小变化，pop(0)的时间随着list变大而变长

```
import timeit
popzero = timeit.Timer("x.pop(0)", "from __main__ import x")
popend = timeit.Timer("x.pop()", "from __main__ import x")
```

Timing test for list.pop
list.pop的计时试验

First, let's look at the comparison
首先我们看对比

For lists of length 2 million, performed 1000 times
对于长度2百万的列表，执行1000次

The pop() time is 0.0007 seconds
pop()时间是0.0007秒

The pop(0) time is 0.8 seconds
pop(0)时间是0.8秒

The difference is about 1,000 times
相差1000倍

```
>>> x = list(range(2000000))
>>> popzero.timeit(number=1000)
0.7688910461739789
>>> x = list(range(2000000))
>>> popend.timeit(number=1000)
0.00073471238020417 22
```

Timing test for list.pop
list.pop的计时试验

We tested the increasing trend of the two operations by changing the list size

我们通过改变列表的大小来测试两个操作的增长趋势

```
import timeit
popzero = timeit.Timer("x.pop(0)", "from __main__ import x")
popend = timeit.Timer("x.pop()", "from __main__ import x")
print "pop(0)   pop()"
for i in range(1000000,100000001,1000000):
    x = list(range(i))
    pt = popend.timeit(number=1000)
    x = list(range(i))
    pz = popzero.timeit(number=1000)
    print "%15.5f, %15.5f" %(pz,pt)
```

Timing test for list.pop
list.pop的计时试验

We tested the increasing trend of the two operations by changing the list size

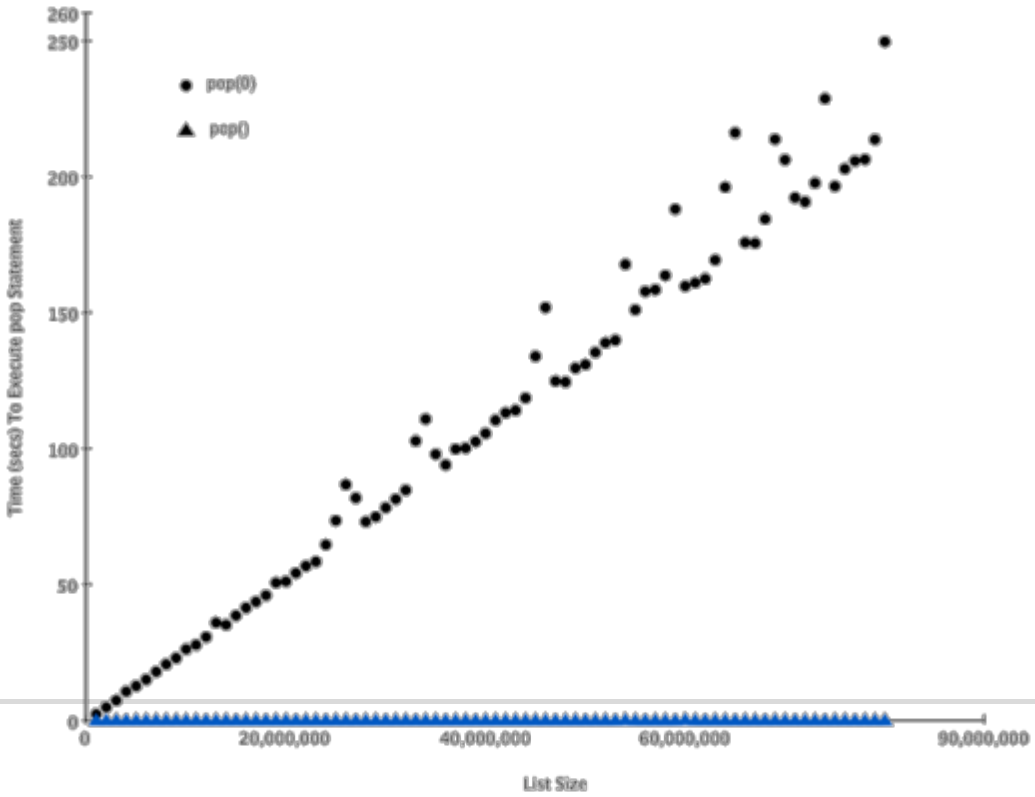
我们通过改变列表的大小来测试两个操作的增长趋势

```
>>>
pop(0)    pop()
          0.23149,          0.00078
          0.68661,          0.00020
          1.43575,          0.00045
          2.00506,          0.00027
          2.71711,          0.00032
          3.32652,          0.00030
          4.03600,          0.00032
          4.56179,          0.00026
          5.17211,          0.00034
          5.75793,          0.00025
          6.28499,          0.00028
          6.63129,          0.00033
          7.15702,          0.00027
```

Timing test for list.pop
list.pop的计时试验

The test data in a chart shows the growing trend
将试验数据画成图表 ，可以看出增长趋势

- The pop () is a flat constant
  pop()是平坦的常数
- The pop (0) is a trend of linear growth
  pop(0)是线性增长的趋势



The dict data type
dict数据类型

Unlike list, dictionary find data items according to the key code (key), and the list is based on location (index)

字典与列表不同 ，根据关键码（key）找到数据项 ，而列表是根据位置 （index)

The most commonly used values get and assigned set with a performance of O(1)

最常用的取值get和赋值set，其性能为O(1)

Another important operation, contains (in), is to determine whether a key code (key) exists in the dictionary, which is also O (1)

另一个重要操作contains(in)是判断字典中是否存在某个关键码(key)，这个性能也是O(1)

operation	Big-O Efficiency
copy	O(n)
get item	O(1)
set item	O(1)
delete item	O(1)
contains (in)	O(1)
iteration	O(n)

Comparing the in operations of the list and the dict

list和dict的in操作对比

A performance trial was designed to validate a value in list and a timing contrast for a value in dict

设计一个性能试验来验证list中检索一个值， 以及dict中检索一个值的计时对比

Both a list containing continuous values and a dict containing a continuous key code key were generated, testing the time-consuming of the operator'in' with random numbers.

生成包含连续值的list和包含连续关键码key的dict，用随机数来检验操作符in的耗时。

```
import timeit
import random

for i in range(10000,1000001,20000):
    t = timeit.Timer("random.randrange(%d) in x"%i,
                     "from __main__ import random,x")
    x = list(range(i))
    lst_time = t.timeit(number=1000)
    x = {j:None for j in range(i)}
    d_time = t.timeit(number=1000)
    print("%d,%10.3f,%10.3f" % (i, lst_time, d_time))
```

Comparing the in operations of the list and the dict
list和dict的in操作对比

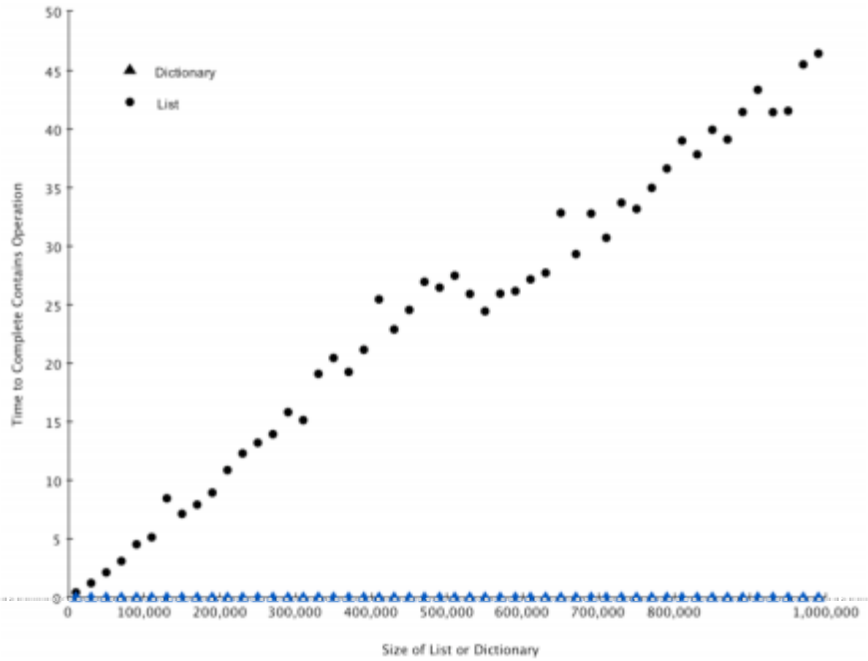
The execution time of the visible dictionary is independent of the scale and is a constant

可见字典的执行时间与规模无关 ，是常数

The execution time of the list increases linearly as the size of the list increases

而列表的执行时间则随着列表的规模加大 而线性上升

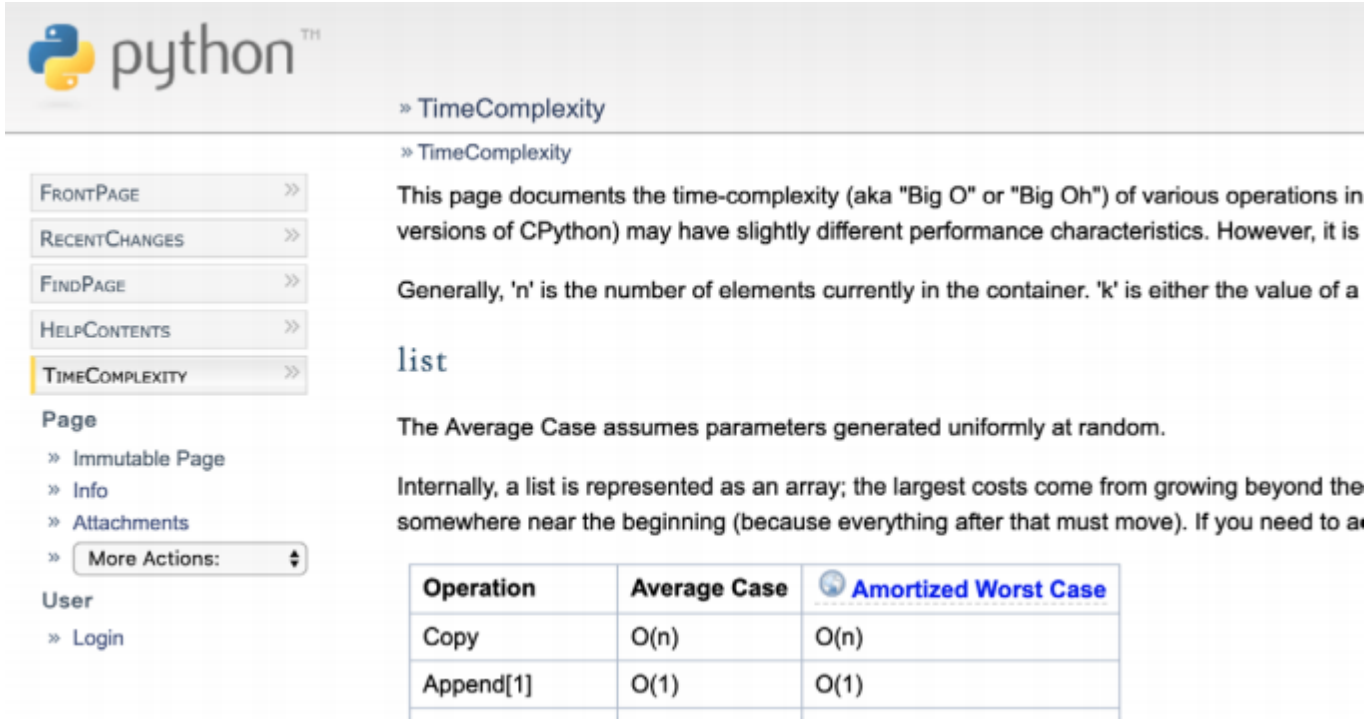
10000,	0.062,	0.001
30000,	0.189,	0.001
50000,	0.329,	0.001
70000,	0.440,	0.001
90000,	0.582,	0.001
110000,	0.710,	0.001
130000,	0.840,	0.001
150000,	0.951,	0.001
170000,	1.077,	0.001
190000,	1.227,	0.001
210000,	1.377,	0.001
230000,	1.499,	0.001
250000,	1.618,	0.001
270000,	1.738,	0.001
290000,	1.892,	0.001
310000,	2.055,	0.001



More Python datatype operation complexity
更多Python数据类型操作复杂度

Python's official algorithm complexity website:
Python官方的算法复杂度网站：

https://wiki.python.org/moin/TimeComplexity



Discussion

Discuss

1）The time complexity of an algorithm is $O(n^2)$, indicating that (　).

某算法的时间复杂度是$O(n^2)$，表明该算法的（　）。

A.The scale of the problem is n^2　问题规模是n^2

B.The execution time is equal to n^2　执行时间等于n^2

C.The execution time is proportional to n^2　执行时间与n^2成正比

D.The scale of the problem is proportional to n^2　问题规模与n^2成正比

时间复杂度为$O(n^2)$，说明算法的时间复杂度$T(n)$满足$T(n)\leq cn^2$(c为比例常数)，即$T(n)=O(n^2)$，时间复杂度$T(n)$是问题规模n的函数，其问题规模仍然是n而不是n^2。

2）In the following statement, the wrong is (　).

下面的说法中，错误的是（　）。

Ⅰ.The meaning of the algorithm working in-place is that it does not need any additional auxiliary space
　　算法原地工作的含义是指不需要任何额外的辅助空间

Ⅱ.Under the same scale, the algorithm with complexity $O(n)$ is always superior to the algorithm with complexity $O(2^n)$ in time
　　在相同的规模下，复杂度为$O(n)$的算法在时间上总是优于复杂度为$O(2^n)$的算法

Ⅲ. The so-called time complexity refers to an upper bound of the estimated algorithm execution time in the worst case
　　所谓时间复杂度，是指最坏情况下估算算法执行时间的一个上界

Ⅳ. For the same algorithm, the higher the level of implementation language, the lower the execution efficiency
　　同一个算法，实现语言的级别越高，执行效率越低

A.Ⅰ　　B.Ⅰ,Ⅱ　　C.Ⅰ,Ⅳ　　D.Ⅲ

Ⅰ，算法原地工作是指算法所需的辅助空间是常量。Ⅱ，本项考查算法效率的理解，时间复杂度是指渐近时间复杂度，不要想当然地去给n赋予一个特殊值，时间复杂度为$O(n)$的算法必然优于时间复杂度为$O(2^n)$的算法。Ⅲ，时间复杂度总是考虑最坏情况下的时间复杂度，以保证算法的运行时间不会比它更长。Ⅳ，正确。

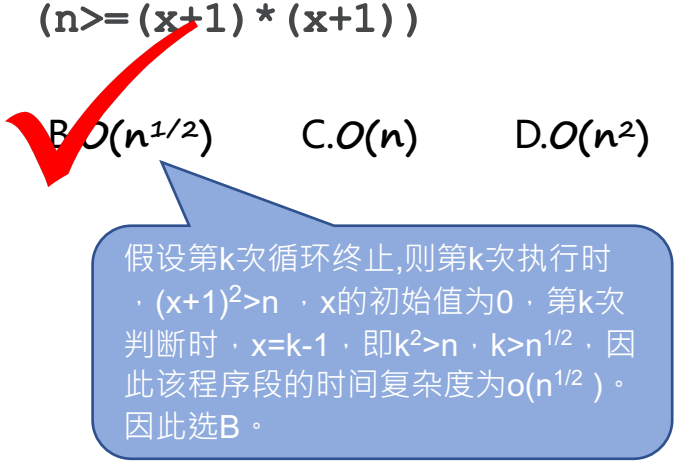
Discuss

3）Let **n** be a non negative integer describing the scale of the problem, and the time complexity of the following program segments is (　).

设n是描述问题规模的非负整数，下列程序段的时间复杂度是(　)。

```
x=0;
while  (n>=(x+1)*(x+1))
x=x+1;
```

A.$O(\log n)$　B.$O(n^{1/2})$　C.$O(n)$　D.$O(n^2)$



Discuss

1）When considering computing problems, it is necessary to clearly distinguish the three concepts of problem, problem example and algorithm, and understand the relationship between them. Can you distinguish the above three concepts? Please cite some calculation problems and examples.

在考虑计算问题时，需要清晰地区分问题、问题实例和算法三个概念，并理解它们之间的关系，你能区分以上三个概念吗？请举出一些计算问题及实例。

●问题：一个问题W是需要解决（需要用计算求解）的一个具体需求。例如判断任一个正整数N是否为素数，求任一个方形矩阵的行列式的值等。

∗Problem: a problem W is a specific requirement that needs to be solved (it needs to be solved by calculation). For example, determining whether any positive integer n is a prime number and finding the value of the determinant of any square matrix.

●问题实例：问题W的一个实例w是该问题的一个具体例子，通常可以通过一组具体的参数设定。例如判断1013是否为素数，或者求一个具体方形矩阵的行列式的值。显然，一个问题反映了其所有实例的共性。

∗Problem example: an example w of problem W is a specific example of the problem, which can usually be set through a set of specific parameters. For example, it is determined whether 1013 is a prime number or the value of the determinant of a specific square matrix is obtained. Obviously, a problem reflects the commonality of all its examples.

Discuss

make some noise

●算法：解决问题W的一个算法A，是对一种计算过程的严格描述。对W的任何一个实例w，实施算法A描述的计算过程，就能得到w的解。例如，一个判断素数的算法应该能给出1013是否为素数的判断，也能判断其他正整数是否为素数；一个求矩阵的行列式值的算法必须能应用于任何矩阵，得到其行列式的值。

∗Algorithm: algorithm **A** for solving problem **W** is a strict description of a calculation process. For any instance **w** of **W**, the solution of **w** can be obtained by implementing the calculation process described in algorithm **A**. For example, an algorithm for judging a prime number should be able to judge whether 1013 is a prime number and whether other positive integers are prime numbers; An algorithm for finding the value of the determinant of a matrix must be applied to any matrix to obtain the value of its determinant.

下面是一些计算问题及其实例：

1. 求两个（两维）矩阵的乘积；求任意两个具体矩阵的乘积都是其实例。
2. 将一个整系数多项式分解为一组不可约整系数多项式因子的乘积；计算具体多项式的因式是这个问题的实例。
3. 将一个图像旋转90度，其实例是要求旋转的各种具体图像。
4. 辨认出数字相机取景框里的人脸，每次拍照的取景框图像都是其实例。

Discuss

2）When analyzing a problem, we often consider two aspects. One is the scale of the problem instance, which is the basic scale used when analyzing an algorithm. The cost of the algorithm will be described based on it. Different scales lead to different conclusions about the cost of the algorithm. On the other hand, when the same algorithm is used to solve the problem, the calculation cost may be different even for instances of the same size. Try to discuss prime number judgment and matrix determinant calculation from these two aspects.

在分析问题时，我们常常从两个方面考虑，一个是问题实例的规模，这是分析一个算法时采用的基本尺度，算法的代价将基于它描述。尺度不同，有关算法代价的结论就会不同。另一个是，在使用同一个算法解决问题时，即使对于同样规模的实例，计算的代价也可能不同。试着从这两个方面讨论一下素数判断和矩阵行列式计算。

对判断素数的问题，有两个尺度可能作为实例的规模度量。一种可能是用被检查整数的数值，另一种可能是取被检查整数按某种进制（例如十进制或二进制）表示的数字串长度。这里出现了一个大问题：整数的数值与其进制表示的数字串长度之间有着指数关系，长m的数字串可以表示数值直至B^m-1的整数，其中B为进制的基数（例如10或者2）。

For the problem of judging prime numbers, there are two scales that can be used as the scale measure of an example. One possibility is to use the value of the checked integer, and the other possibility is to take the length of the string of digits represented by the checked integer in a certain base (such as decimal or binary). There is a big problem here: there is an exponential relationship between the value of an integer and the length of the digit string represented by its base. A digit string with a length of m can represent an integer with a value up to B^m-1, where B is the base of the base (for example, 10 or 2).

Discuss

对矩阵行列式问题也有两种可能，可以取矩阵的维数作为尺度，或者取矩阵中的元素个数。两者之间也有一个平方的差异。虽然上述几种尺度选择好像都有道理，但是，选择不同的尺度描述算法的代价，就会得到很不一样的结果。

There are also two possibilities for the matrix determinant problem. The dimension of the matrix can be taken as the scale, or the number of elements in the matrix can be taken. There is also a square difference between the two. Although the above-mentioned scale selection seems reasonable, the cost of choosing different scales to describe the algorithm will result in very different results.

对于同一算法的计算代价问题，以素数判断算法为例，同样由100个十进制数字表示的数，如果被判断的是偶数，算法启动后只需做一次检查，立刻就可以给出否定的结果；如果是奇数，就需要花更多时间。在这方面,处理不同问题的算法,其性质也可能不同。

For the calculation cost of the same algorithm, take the prime number judgment algorithm as an example. If the number represented by 100 decimal numbers is judged to be an even number, the algorithm only needs to check once after startup, and a negative result can be given immediately; If it is an odd number, it will take more time. In this regard, the nature of the algorithms that deal with different problems may be different.

Discuss

3）After the above discussion, we will naturally think that since there are many possible considerations for measuring the cost of a certain algorithm, what should be the measurement standard in practice?

经过上面的讨论，我们很自然的会思考，既然度量某一算法的代价存在多种可能的考虑，那么实际中应该以什么为衡量标准呢？

第一种考虑的价值不大，因为它没提供什么有用信息。在实际中也是如此，对完成工作所需时间的最乐观的估计基本上没有价值。

The first consideration is of little value because it provides little useful information. In practice, too, the most optimistic estimate of the time required to complete the work is basically worthless.

第二种考虑提供了一种保证，说明在这段时间之内，本算法一定能完成工作。这种代价称为最坏情况的时间代价。

The second consideration provides a guarantee that the algorithm can complete the work within this period of time. This cost is called the time cost of the worst case.

Discuss

第三种考虑是希望对算法做一个全面评价（处理规模为n的实例的平均时间代价），因此它完整全面地反映了这个算法的性质。但另一方面，这个时间代价并没有保证，不是每个计算都能在这一时间内完成。此外，“平均”还依赖于所选的实例，以及这些实例出现的概率。通常总需要做一些假定，例如假定规模同为n的各种实例是均匀分布的。然而在应用实践中，实际问题实例有自己的分布情况，未必与理论分布一致，而实际中的真实分布通常很难确定。这些都给平均代价概念的应用带来困难。

The third consideration is to make a comprehensive evaluation of algorithm (the average time cost of processing instances with scale n), so it fully reflects the nature of this algorithm. On the other hand, this time cost is not guaranteed, and not every calculation can be completed within this time. In addition, average also depends on the selected instances and the probability of their occurrence. It is usually necessary to make some assumptions. For example, it is assumed that various instances with the same size of N are uniformly distributed. However, in the application practice, the actual problem examples have their own distribution, which may not be consistent with the theoretical distribution, and the actual distribution is usually difficult to determine. All these bring difficulties to the application of the concept of average cost.