数据结构与算法（Python）-02 算法分析

陈斌 北京大学地球与空间科学学院 gischen@pku.edu.cn

目录

- 本章目标
- 什么是算法分析
  - 大O表示法
  - 例子：“变位词”判断问题
- Python数据结构的性能
  - 列表List
  - 字典Dictionary

本章目标

- 了解算法分析的重要性
- 能够采用“大O”表示法来描述算法执行时间
- 了解Python列表和字典类型中通用操作执行时间的“大O”级别；
- 了解Python数据类型的具体实现对算法分析的影响；
- 了解如何对简单Python程序进行执行时间检测。

什么是算法分析

- **如何对比两个程序？**

  看起来不同，但解决同一个问题的程序，哪个“更好”？

- **程序和算法的区别**

  算法是对问题解决的分步描述

  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

- **我们来看一段程序，完成从1到n的累加，输出总和**

  设置累计变量=0

  从1到n循环，逐次累加到累计变量

  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))
```

什么是算法分析

- **再看第二段程序，是否感觉怪怪的？**

  但实际上本程序功能与前面那段相同

  这段程序失败之处在于：

  - 变量命名
  - 有无用的垃圾代码

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

    return fred

print(foo(10))
```

- **比较程序的“好坏”，有更多因素**

  代码风格、可读性等等

- **本课程中，我们主要感兴趣的是算法本身特性**

- **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**

  更高效利用计算资源，或者更少占用资源的算法，就是好算法

  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

说到代码风格和可读性

- **为什么Python的语句块强制缩进是好的？**
  语句块功能和视觉效果统一
- **苹果公司的一个低级Bug**
  造成SSL连接验证被跳过
  2014.2.22修正iOS7.0.6
- **不像看起来那样运行**
- **还有下面这样**

```
if ( ...... )
    return x;
if ( ...... )
    return y;
if ( ...... )
    foo();
    return z;
```

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedPa
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,              /* plaintext */
                       dataToSignLen,           /* plaintext length */
                       signature,
                       signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
        goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

计算资源指标

- **那么何为计算资源？**
- **一种是算法解决问题过程中需要的存储空间或内存**
  但存储空间受到问题自身数据规模的变化影响
  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
- **另一种是算法的执行时间**
  我们可以对程序进行实际运行测试，获得真实的运行时间
- **Python中有一个time模块，可以获取计算机系统当前时间**
  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

```
>>> import time
>>> time.time()
1426317539.975
```

```
>>> print time.time.__doc__
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.
```

1970/1/1 00:00:00

运行时间检测

- **累计求和程序的运行时间检测**
  增加了总运行时间
  函数返回一个元组tuple
  包括累计和，以及运行时间（秒）
- **在交互窗口连续运行5次看看**
  1到10,000累加
  每次运行约需0.0019秒

```
import time

def sumOfN2(n):
    start = time.time()

    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    end = time.time()

    return theSum,end-start
```

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN2(10000))
Sum is 50005000 required  0.0018950 seconds
Sum is 50005000 required  0.0018620 seconds
Sum is 50005000 required  0.0019171 seconds
Sum is 50005000 required  0.0019162 seconds
Sum is 50005000 required  0.0019360 seconds
```

运行时间检测

- **如果累加到100,000？**

  看起来运行时间增加到10,000的10倍

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN2(100000))
Sum is 5000050000 required  0.0199420 seconds
Sum is 5000050000 required  0.0180972 seconds
Sum is 5000050000 required  0.0194821 seconds
Sum is 5000050000 required  0.0178988 seconds
Sum is 5000050000 required  0.0188949 seconds
```

- **进一步累加到1,000,000？**

  运行时间又是100,000的10倍了

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN2(1000000))
Sum is 500000500000 required  0.1948988 seconds
Sum is 500000500000 required  0.1850290 seconds
Sum is 500000500000 required  0.1809771 seconds
Sum is 500000500000 required  0.1729250 seconds
Sum is 500000500000 required  0.1646299 seconds
```

第二种无迭代的累计算法

- 利用求和公式的无迭代算法

```
def sumOfN3(n):
    return (n*(n+1))/2

print(sumOfN3(10))
```

- 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 0.00000095 seconds
Sum is 5000050000 required 0.00000191 seconds
Sum is 500000500000 required 0.00000095 seconds
Sum is 50000005000000 required 0.00000095 seconds
Sum is 5000000050000000 required 0.00000119 seconds
```

- 需要关注的两点

  这种算法的运行时间比前种都短很多

  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）

- 新算法运行时间几乎与需要累计的数目无关

运行时间检测的分析

- **观察一下第一种迭代算法**
  包含了一个循环，可能会执行更多语句
  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
- **但关于运行时间的实际检测，有点问题**
  关于编程语言和运行环境
- **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**
  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
- **所以我们需要更好的方法来衡量算法的运行时间**
  这个指标与具体的机器、程序、运行时段，与编译器都无关

大O表示法（Big-O）

- **一个算法所实施的操作数量或者步骤数可以作为独立于具体程序或机器的度量指标**

  哪种操作跟算法的具体实现无关？

  需要一种通用的基本操作来作为运行步骤的计量单位

- **赋值语句是一个合适的选择**

  一条赋值语句同时包含了（表达式）计算和（变量）存储两个基本资源

  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之外，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赋值语句的作用，并不实施处理。

- **分析SumOfN的赋值语句数量**

  对于“问题规模”n

  赋值语句数量T(n)=1+n

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

print(sumOfN(10))
```

大O表示法（Big-O）

- **问题规模影响算法执行时间**

  在前n个自然数累计求和的算法中，需要累计的自然数个数合适作为问题规模的指标

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

  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

- **数量级函数（Order of Magnitude function）**

  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

  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

  数量级函数描述了T(n)中随着n增加而增加速度最快的部分

  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

确定运行时间数量级大O的方法

- **例1：T(n)=1+n**

  当n增大时，常数1在最终结果中显得越来越无足轻重

  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)

- **例2：$T(n)=5n^2+27n+1005$**

  当n很小时，常数1005其决定性作用

  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

  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

  所以可以在数量级中去掉27n+1005，以及系数5的部分，确定为$O(n^2)$

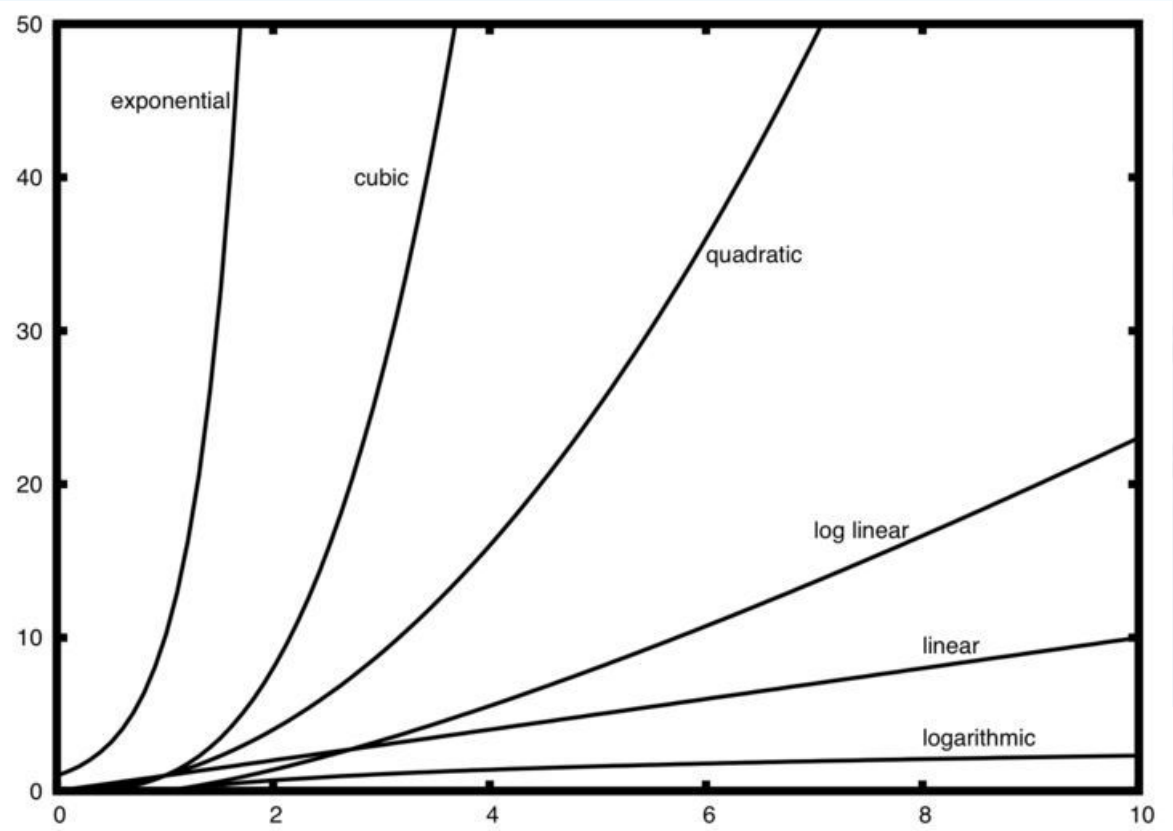
- **有时候，决定运行时间的不仅仅是问题规模，某些具体数据也会影响算法的运行时间**

  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

  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

常见的大O数量级函数

- 通常当n较小时，难以确定其数量级
- 当n增长到较大时，容易看出其主要变化量级



$f(n)$	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

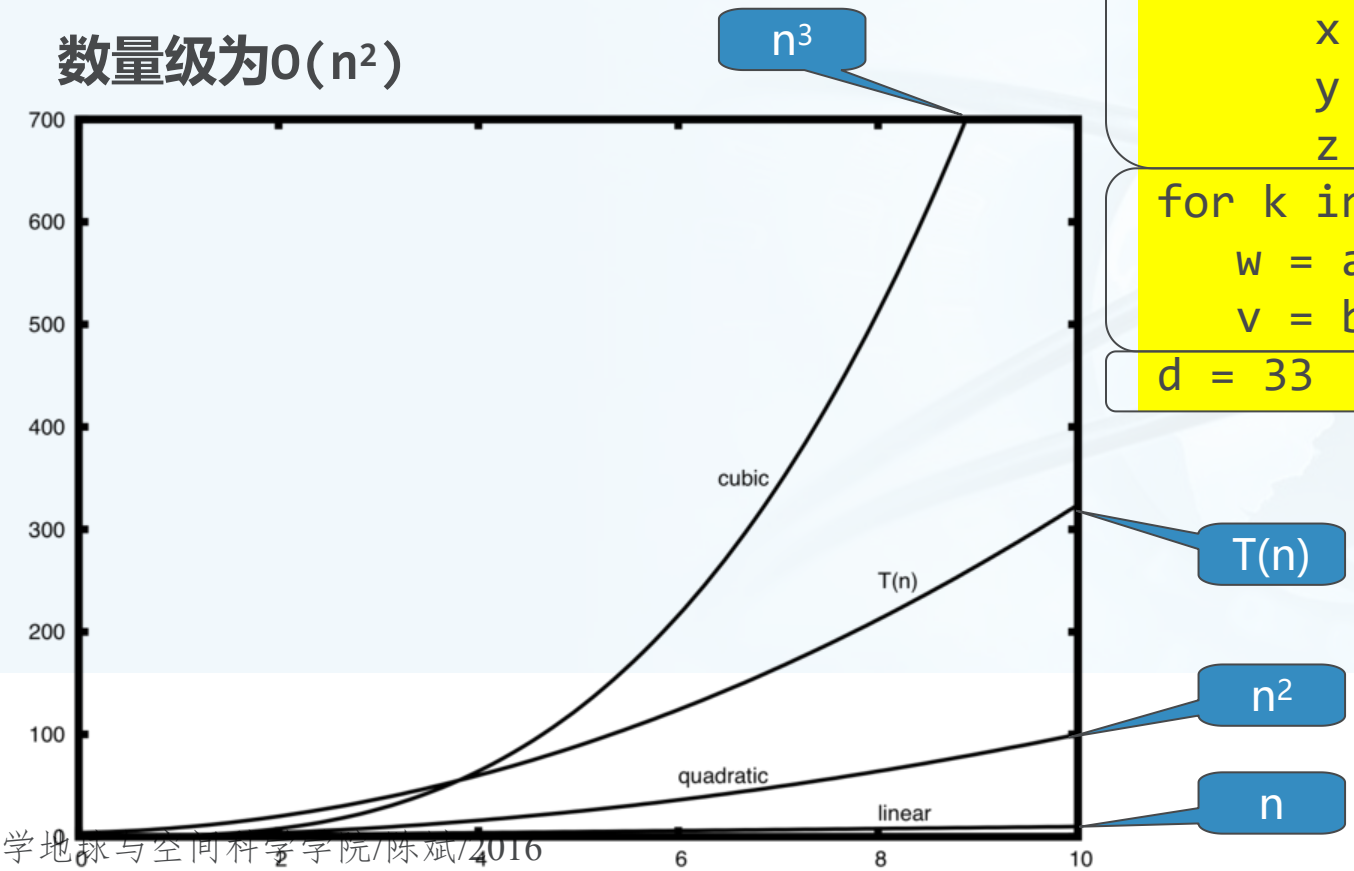
从代码分析确定执行时间数量级函数

- 代码赋值语句可以分为4个部分

  $T(n)=3+3n^2+2n+1=3n^2+2n+4$

- 仅保留最高阶项n^2，去掉所有系数
- 数量级为$O(n^2)$

```
a=5
b=6
c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
d = 33
```



随堂作业2-1


```
def min1(lst):
def min2(lst):
```

- **写两个Python函数，功能为找到列表中最小数**
  别用list.sort()
- **要求函数1将每个数与其它所有数比较，运行时间数量级为O(n^2)**
- **要求函数2数量级为O(n)**
- **3种提交方法**
  电脑/微信：从课程网站找到作业提交表单直接输入代码提交

  微信：写在纸上（包括学号姓名）拍照，从微信表单上传附件提交

  纸质：直接交纸质版（包括学号姓名）
- **微信表单入口见微信公众号“chbpku”**
  菜单“教学课程”->“数据结构与算法”

“变位词”判断问题

- **“变位词”判断问题可以很好地展示不同数量级的算法**
- **问题描述**

  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

  如：heart和earth，python和typhon

  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
- **解题目标：写一个bool函数，以两个词作为参数，返回真假，表示这两个词是否变位词**

解法1：逐字检查

- 解法思路为将字符串1中的字符逐个到字符串2中检查是否存在，存在就“打勾”标记（防止重复检查），如果每个字符都能找到，则两个词是变位词，只要有1个字符找不到，就不是变位词
- 程序技巧

  实现“打勾”标记：将字符串2对应字符设为None

  由于字符串是不可变类型，需要先复制到列表中

解法1：逐字检查-程序代码


```
def anagramSolution1(s1,s2):
    alist = list(s2)
    pos1 = 0
    stillOK = True
    while pos1 < len(s1) and stillOK:
        pos2 = 0
        found = False
        while pos2 < len(alist) and not found:
            if s1[pos1] == alist[pos2]:
                found = True
            else:
                pos2 = pos2 + 1
        if found:
            alist[pos2] = None
        else:
            stillOK = False
        pos1 = pos1 + 1
    return stillOK

print(anagramSolution1('abcd','dcba'))
```

复制s2到列表

循环s1的每个字符

在s2列表逐个对比

找到，打勾

未找到，失败

解法1：逐字检查-算法分析

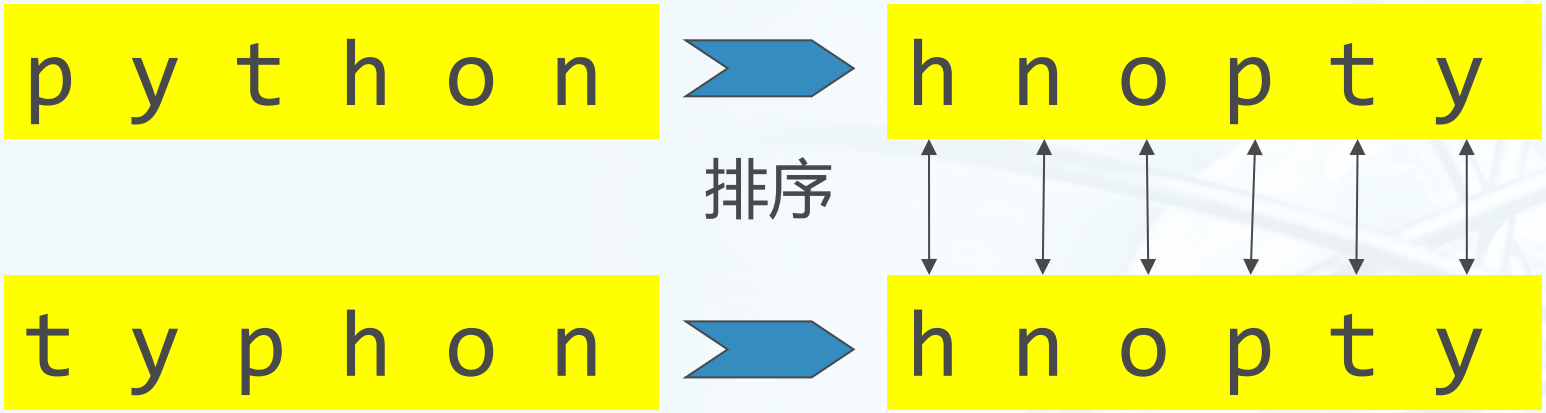
- 问题规模：词中包含的字符个数n
- 主要部分在于两重循环
- 外重循环要遍历字符串1每个字符，将内层循环执行n次
- 而内重循环在字符串2中查找字符，每查找一个字符的操作次数，分别是1、2、3……n中的一个，而且各不相同
- 所以总执行次数是1+2+3+……+n
- 可知其数量级为$O(n^2)$

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

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

解法2：排序比较

- 解题思路为：将两个字符串都按照字母顺序排好序，再逐个字符对比是否相同，如果相同则是变位词



解法2：排序比较


```
def anagramSolution2(s1,s2):
    alist1 = list(s1)
    alist2 = list(s2)

    alist1.sort()
    alist2.sort()
    pos = 0
    matches = True
    while pos < len(s1) and matches:
        if alist1[pos]==alist2[pos]:
            pos = pos + 1
        else:
            matches = False
    return matches

print(anagramSolution2('abcde','edcba'))
print(anagramSolution1('abcd','dcba'))
```

分别都排序

逐个对比

解法2：排序比较-算法分析

- 粗看上去，本算法只有一个循环，最多执行n次，数量级应该是O(n)
- 但循环前面的两个sort并不是无代价的
- 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(n log n)，大过循环的O(n)
- 所以本算法中其决定性作用的步骤是排序步骤
- 本算法的运行时间数量级就等于排序过程的数量级O(n log n)

解法3：暴力法

- 暴力法解题思路为：穷尽所有可能组合
- 对于变位词问题来说，暴力法具体是，将字符串1中出现的字母进行全排列，再查看字符串2是否出现在全排列列表中
- 这里最大的困难是产生字符串1所有字母的全排列，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！
- 我们已知n！的增长速度甚至超过2^n

  例如，对于20个字符长的词来说，将产生

  20!=2,432,902,008,176,640,000 个候选词

  如果每秒钟处理一个候选词的话，需要77,146,816,596年（百亿）的时间来做完所有的匹配。
- 结论：暴力法恐怕不能算是个好算法

解法4：计数比较

- 最后一个算法解题思路为：对比两个字符串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
- 具体做法：为每个字符串设置一个26位的计数器，先检查每个字符串，在计数器中设定好每个字母出现的次数
- 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

解法4：计数比较-程序代码


```
def anagramSolution4(s1,s2):
    c1 = [0]*26
    c2 = [0]*26
    for i in range(len(s1)):
        pos = ord(s1[i])-ord('a')
        c1[pos] = c1[pos] + 1
    for i in range(len(s2)):
        pos = ord(s2[i])-ord('a')
        c2[pos] = c2[pos] + 1
    j = 0
    stillOK = True
    while j<26 and stillOK:
        if c1[j]==c2[j]:
            j = j + 1
        else:
            stillOK = False
    return stillOK

print(anagramSolution4('apple','pleap'))
```

分别都计数

计数器比较

解法4：计数比较-算法分析

- **计数比较算法中有3个循环迭代，但不象解法1那样存在嵌套循环**
  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

  第3个循环用于计数器比较，操作次数总是26次

- **所以总操作次数T(n)=2n+26，其数量级为O(n)，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。**

- **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**
  由于仅限于26个字母构成的词，本算法对空间额外的需求并不明显，但如果考虑由大字符集构成的词（如中文具有上万不同字符），情况就会有所改变。

- **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”

随堂作业2-2

- **判断下列代码段的大O级别**

  A. O(n) B. O(n^2) C. O(log n) D. O(n^3)

- **Q1：**

```
test = 0
for i in range(n):
    for j in range(n):
        test = test + i * j
```

- **Q2：**

```
test = 0
for i in range(n):
    test = test + 1
for j in range(n):
    test = test - 1
```

- **Q3：**

```
i = n
while i > 0:
    k = 2 + 2
    i = i // 2
```

Python数据结构的性能

- 前面我们了解了“大O表示法”以及对不同的算法的评估
- 下面来讨论下Python两种内置数据结构上各种操作的大O数量级

  列表List和字典Dictionary

  这是两种重要的Python数据结构，后面的课程会用来实现各种数据结构

  通过运行试验来确定其各种操作的运行时间数量级
- 对比List和Dictionary

类型	索引	添加	删除	更新	正查	反查	其它
List	连续整数i	append extend insert	pop remove*	a[i]=v	a[i] a[i:j]	index(v) count(v)	reverse sort
Dict	不可变类型值k	b[k]=v	pop	b[k]=v	b[k] copy	无	has_key update

List数据结构

- List数据结构各种操作（interface）的实现方法有很多，如何选择具体采用哪种实现方法？
- 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准则：80%的功能其使用率只有20%
- 我们来看一些常用的List操作
- 最常用的是：按索引取值和赋值（v= a[i]，a[i]= v）
  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)
- 另一个是列表增长，可以选择append()和__add__()
  lst.append(v)，执行时间是O(1)
  lst= lst+ [v]，执行时间是O(n+k)，其中k是被加的列表长度
  如何选择具体方法，决定了程序的性能

4种生成前n个整数列表的方法

- 第一种方法是用循环连接列表（+）方式生成
- 然后是用append()方法来添加元素生成
- 接着用列表推导式（List Comprehension）来做
- 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))
```

列表推导式

使用timeit模块对函数计时

- **对于每个函数具体的执行时间，timeit模块提供了一种在一致的运行环境下可以反复调用并计时的机制**
- **timeit计时的使用方法**

  首先创建一个Timer对象，需要两个参数，第一个是需要反复运行的语句，第二个是为了建立运行环境而只需要运行一次的“安装语句”

  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，计时完毕后返回以秒为单位的时间

```
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种生成前n个整数列表的方法计时

- **我们看到，4种方法运行时间差别很大**

  列表连接（concat）最慢，List range最快，速度相差近200倍。

  append也要比concat快得多

  另外，我们注意到列表推导式速度是append两倍的样子

- **当然，如果仔细分析，严格来说，上述时间除了具体列表操作的耗时，应该还包括函数调用的时间在内。**

  但函数调用花销的时间是常数

  可以通过调用空函数来确定

  并从上述时间减除掉函数调用时间

```
>>>
concat 1.889487 seconds

append 0.091561 seconds

comprehension 0.038418 seconds

list range 0.009710 seconds
```

List基本操作的大O数量级

- 右表是List基本操作的数量级
- 我们注意到pop这个操作

  pop()从列表末尾移除元素，O(1)

  pop(i)从列表中部移除元素，O(n)

- 原因在于Python所选择的实现方法

  从中部移除元素的话，要把移除元素后面的元素全部向前挪位复制一遍

  这个看起来有点笨拙

  但后面章节我们会看到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)

  这也算是一种对常用操作和不常用操作的折衷方案

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)

list.pop的计时试验

- **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，相对同一个大小的list，分别调用pop()和pop(0)**
- **并对不同大小的list做计时试验，我们期望的结果是**
  pop()的时间不随list大小变化，pop(0)的时间随着list变大而变长
- **首先我们看对比**
  对于长度2百万的列表，执行1000次

  pop()时间是0.0007秒

  pop(0)时间是0.8秒

  相差1000倍

```
>>> x = list(range(2000000))
>>> popzero.timeit(number=1000)
0.7688910461739789
>>> x = list(range(2000000))
>>> popend.timeit(number=1000)
0.0007347123802041722
```

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

list.pop的计时试验

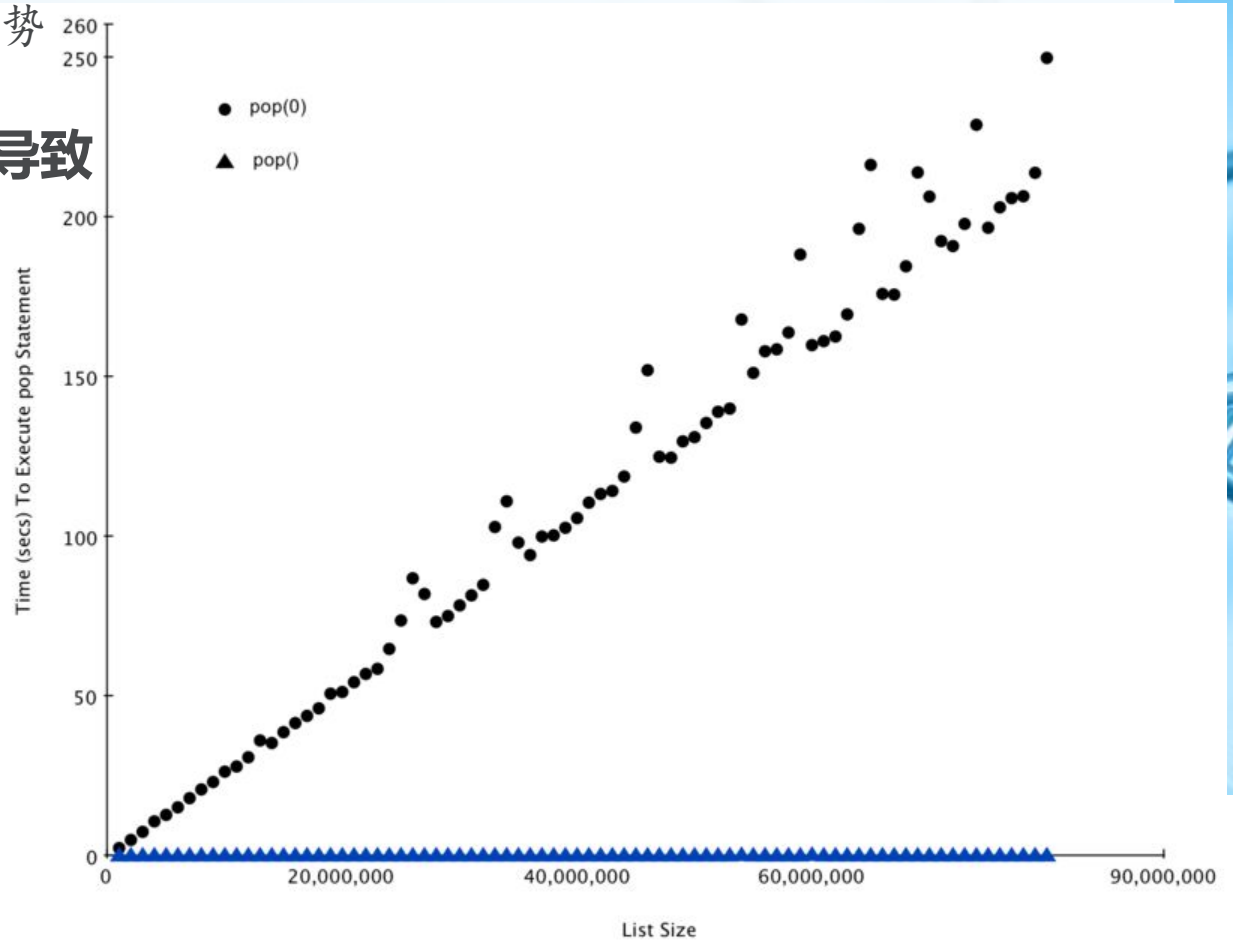
› 我们通过改变列表的大小来测试两个操作的增长趋势

```
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)
```

```
>>>
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
```

list.pop的计时试验

- **通过将试验数据画成图表，可以看出增长趋势**
  pop()是平坦的常数
  pop(0)是线性增长的趋势
- **其中散落的点是误差导致**
  系统中其它进程调度
  资源占用等



Dictionary数据结构

- 字典与列表不同，它可以根据关键码（key）来找到数据项，而列表是根据位置（index）来找到数据项
- 后面的课程会介绍字典的几种不同实现方法
- 最常用的取值get item和赋值set item操作，其性能为O(1)
- 另一个重要操作是判断字典中是否存在某个关键码（key），这个性能也是O(1)
- 某些罕见的情况下性能会劣化
  set/get/contains -> O(n)
  后面课程讲到实现的时候会分析

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)

List和Dictionary的in操作对比

- **设计一个性能试验来验证List中检索一个值，以及Dictionary中检索一个值的计时对比**

  生成一个包含连续值的List和一个包含连续关键码Key的Dictionary

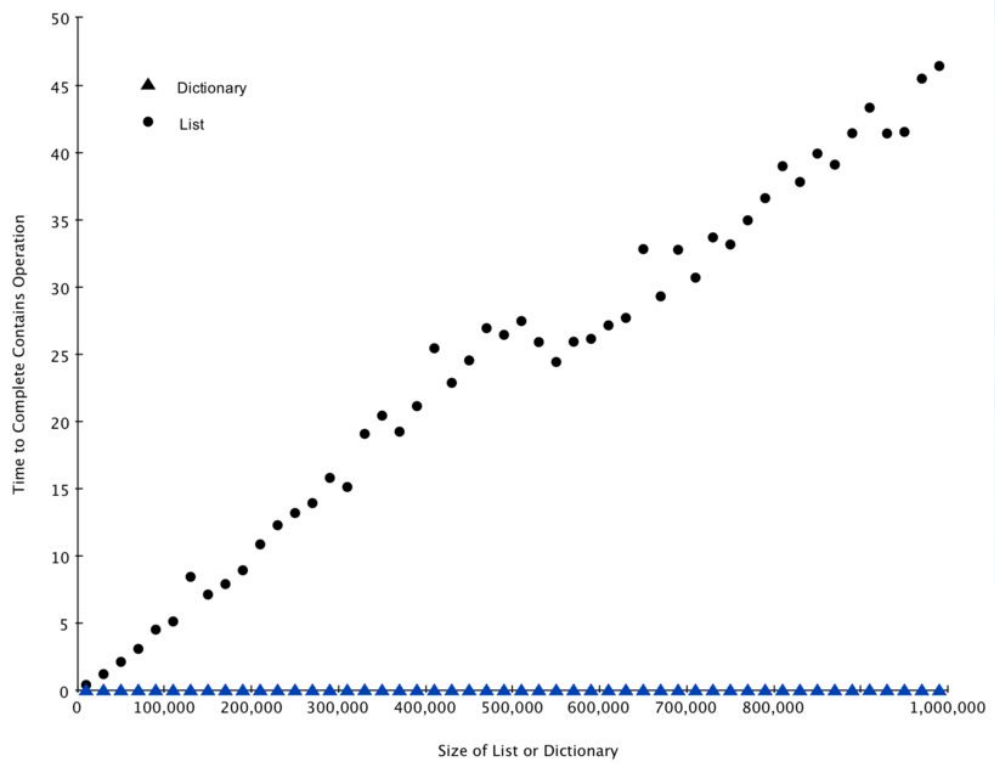
  用随机数来检验，采用同一个操作符 in 来判断。

```
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))
```

List和Dictionary的in操作对比

- 可见字典的执行时间与字典规模大小无关，是常数
- 而列表的执行时间则随着列表的规模加大而线性上升Python
- Python官方的算法复杂度网站：
  https://wiki.python.org/moin/TimeComplexity



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

随堂作业2-3

- Q4:下面哪个List操作不是O(1)?
  - a) list.pop(0)
  - b) list.pop()
  - c) list.append()
  - d) list[10]
  - e) 以上都是 O(1)

- Q5：下面哪个字典操作是O(1)？
  - a) 'x' in mydict
  - b) del mydict['x']
  - c) mydict['x'] == 10
  - d) mydict['x']= mydict['x']+ 1
  - e) 以上都是 O(1)

- 微信公众号chbpku和课程网站有在线提交入口（随堂作业2-3）

Python小知识：特殊方法

- **Python有很多内置的语句、函数和操作符，可以用在不同的类型中**

  print a，可以在输出终端显示不同类型变量的字符串形式

  del a，可以销毁对象a；del lst[2]，可以删除列表第3个元素

  len(a)，返回列表、元组或字典类型变量所包含元素的个数

  int(a),float(a)，转换整数、浮点数、字符串为整数或浮点数

  a+b，可以返回整数和、浮点数和、字符串的连接、列表的连接

  a*b，可以返回整数乘积、浮点数乘积、字符串重复、列表的重复

- **这些语句、函数和操作符，同样也可以用于用户自定义的类**

  这些内置的语句函数和操作符，应用到用户自定义的类，会产生什么后果呢？

  比如我们定义了一个类Color，那print一个Color对象会出现什么？两个Color对象的加法，结果怎么样？

Python小知识：特殊方法

- Python提供了一些特殊方法，来让自定义的类也能响应上述的内置语句、函数和操作符

```
>>> a= Color(255, 0, 0)
>>> b= Color(0, 255, 0)
>>> a
red
>>> b
green
>>> a+b
yellow
>>> c= Color(128, 0, 0)
>>> c
Color(128,0,0)
>>> print a, b, c
red green Color(128,0,0)
>>> str(c)
'Color(128,0,0)'
>>>
```

```
class Color(object):
    def __init__(self, r, g, b):
        self.value= (r, g, b)
        if (r, g, b)== (255,0,0):
            self.name= "red"
        elif (r, g, b)== (0, 255, 0):
            self.name= "green"
        elif (r, g, b)== (255, 255, 0):
            self.name= "yellow"
        else:
            self.name= "N/A"
    def __add__(self, b):
        return Color(self.value[0]+ b.value[0], \
                     self.value[1]+ b.value[1], \
                     self.value[2]+ b.value[2])
    def __str__(self):
        if self.name!= "N/A":
            return self.name
        else:
            return "Color(%d,%d,%d)" % self.value
    __repr__= __str__
```

Python小知识：模块

- **Python在核心语言成分之外，还有很多实现不同功能的函数库，称为module模块，Python自带的模块很多**

字符串处理

扩展的数据类型

数值计算与数学函数

文件与目录访问

数据持久化

数据压缩与存档

多种格式文件读写（csv、ini等）

加密处理

操作系统底层服务

进程间通讯和网络

XML数据处理啊

Internet数据处理（email编解码等）

Internet 通 讯 协 议 处 理（HTTP,SMTP,POP,各种Server）

多媒体数据处理（音频、图像）

国际化语言

可执行程序处理

图形用户界面Tk

开发工具

调试工具

软件打包与发布

Python运行环境服务

Python代码解析和运行

与具体操作系统相关的模块

Python小知识：模块

- 要用到这些模块的功能，就需要使用import语句导入模块（类似C语言里面的include）

```
import shelve

d = shelve.open(filename) # open -- file may get suffix added by low-level
                          # library
d[key] = data   # store data at key (overwrites old data if
                # using an existing key)
data = d[key]   # retrieve a COPY of data at key (raise KeyError if no
                # such key)
del d[key]      # delete data stored at key (raises KeyError
                # if no such key)
flag = d.has_key(key)   # true if the key exists
klist = d.keys() # a list of all existing keys (slow!)

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']      # extracts the copy
temp.append(5)      # mutates the copy
d['xx'] = temp      # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()       # close it
```

Python小知识：模块

- **import语句除了导入模块，还为程序引入了新的名字空间**

```
import time
time.time()
```

- **由于要频繁引用导入模块的功能，有时调用模块的名字空间比较繁琐**

```
import timeit
t= timeit.Timer.timeit(....)
```

- **可以用from ... import ... 来直接将模块的某些部分导入当前命名空间**

```
from timeit import Timer
t= Timer(....)
```

不足之处在于如果导入到当前命名空间的符号太多的话，容易引起混淆冲突

Python小知识：缺省参数

- **python的参数列表可以设置缺省参数以及缺省值**

```
def func(a, b= 0, c= "abc")
func(2)
func(2, 4)
func(2, 4, "xyz")
```

func(2, c= "xyz"): 中间的参数采用缺省值，需要指明后面参数的名称

Python小知识：IDLE的使用

- IDLE分为交互终端Shell，和编辑器Editor
- 用编辑器来编辑python源文件，再用Run->Run Module来运行
- 运行的结果会显示在交互终端Shell里面，同时可以在Shell里面继续输入后续命令来验证运行结果等

杂事

- **关于各种提交表单**

  提交作业，有截止日期的限制，别超过截止日期

  有的浏览器不兼容提交表单，建议采用Firefox浏览器

  表单提交成功的标志是：表单消失，显示成功的字样

- **关于机房上机**

  照顾到对Python编程十分熟练的同学，凡是在上机前一天（周一23:59前）提交了上机作业的同学，周二下午可以不必去机房上机

  鼓励多到机房上机，能够与同学和助教多交流