

# 数据结构与算法 (Python)

## 算法分析

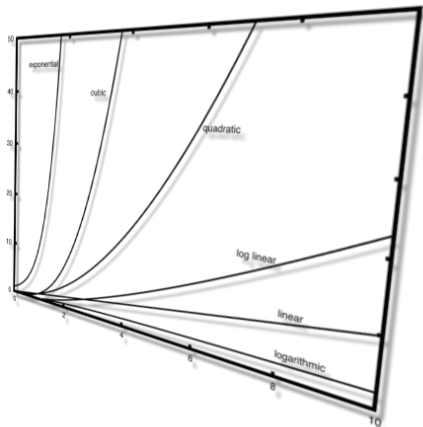
谢正茂 webg@PKU-Mail

北京大学计算机系

March 23, 2021

# 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

- 如何对比两个程序或者算法？
  - 看起来不同，但解决同一个问题的程序，哪个“更好”？
- 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，foo, bar, baz, \* 文档/注释 \*
    - 有无用的垃圾代码
- 比较程序的“好坏”，有很多因素
  - 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/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

```
def foo(tom):  
    fred = 0  
    for bill in range(1, tom + 1):  
        barney = bill  
        fred = fred + barney  
    return fred  
  
print(foo(10))
```

# 算法好坏的评价标准

- 在算法有穷、确定、可行（正确）的基础上
  - 首先要保证是个算法
- 评价主要看 3 方面的指标
  - 运行所花费的时间（时间复杂度）
  - 运行所占用的存储空间（空间复杂性）
  - 其他（如可读性、健壮性、易于维护性）

# 先说代码风格和可读性

- 为什么 Python 的语句块强制缩进是好的? vs. Java, C/C++

...

- 语句块功能、代码结构和视觉效果统一
- 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

```
1  if ( ..... )
2      return x;
3  if ( ..... )
4      return y;
5  if ( ..... )
6      foo();
7      return z;
```

```
1  static OSStatus
2  SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer s
3                                     uint8_t *signature, UInt16 signatureLen)
4  {
5      OSStatus      err;
6      ...
7
8      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
9          goto fail;
10     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
11         goto fail;
12     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
13         goto fail;
14     if ((err = SSLRawVerify(ctx,
15                             ctx->peerPubKey,
16                             dataToSign,          /* plaintext */
17                             dataToSignLen,        /* plaintext length */
18                             signature,
19                             signatureLen);
20         err != 0) {
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 }
```

# 计算资源指标

- 那么何为计算资源？
- 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
- 一种是算法解决问题过程中需要的存储空间或内存
  - 算法运行过程中所需要的存储空间
  - 除了存储输入、输出外，更需要用于存储中间结果
  - 存储一个数  $N$ ，通常需要  $\log N$  空间，当  $N$  较小（比如 1 个亿），也可以认为是个常数
- 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

```
>>> 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.
```

1970/1/1 00:00:00

```
>>> import time
>>> time.time()
1488738563.305619
```



# 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 0.0007 秒

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

```
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
```

# 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

```
10 for i in range(5):
11     print("Sum is %d required %10.7f seconds"
12           % sumOfN2(100000))
```

```
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
```

- 进一步累加到 1,000,000?
  - 运行时间又是 100,000 的 10 倍了

```
10 for i in range(5):
11     print("Sum is %d required %10.7f seconds"
12           % sumOfN2(1000000))
```

```
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
```

## 第二种无迭代的累计算法

- 利用求和公式的无迭代算法

$$S_n = \frac{n(a_1 + a_n)}{2}$$

- 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
- 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

```
14 def sumOfN3(n):  
15     start = time.time()  
16     theSum = (n * (n + 1 )) / 2  
17     end = time.time()  
18     return theSum, end - start
```

```
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 5000000050000000 required 0.0000169 seconds
```

# 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

# 解决问题所需的时间



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，编译器都无关

机器3

时间



# 大 O 表示法 (Big-O)

- 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
- 分析一个算法中，所有要执行的语句的数量
  - 抛弃那些不太重要的因素，只保留最主要的影响因素
- 分析 SumOfN 的语句数量
  - 对于“问题规模”  $n$
  - 语句数量
  - $T(n) = 1 + n + n = 2n + 1$

```
3 def sumOfN(n):
4     theSum = 0
5     for i in range(1, n+1):
6         theSum = theSum + i
7     return theSum
```

# 语句频度和时间复杂度

- 语句频度

- 语句可能重复的最大次数
- 语句频度是针对每条语句的

- 时间复杂度

- 设算法所有语句的语句频度的和是  $T(n)$
- 引入函数  $f(n)$ , 满足如下条件:

$f(n)$ 是当 $n$ 趋向无穷大时, 与 $T(n)$ 为同阶无穷大, 则:

- 则算法的时间复杂度 $T(n)=O(f(n))$
- $n$ 为算法的计算量或者问题规模 (size)
- $f(n)$ 是运算时间随 $n$ 增大时的增长率
- $O(f(n))$ 是算法时间特性的量度

# Big-O 的数学含义-函数渐近的界

设  $f$  和  $g$  是定义域为自然数集  $N$  上的函数

(1)  $f(n)=O(g(n))$

若存在正数  $c$  和  $n_0$  使得对一切  $n \geq n_0$  有  $0 \leq f(n) \leq cg(n)$

(2)  $f(n)=\Omega(g(n))$

若存在正数  $c$  和  $n_0$  使得对一切  $n \geq n_0$  有  $0 \leq cg(n) \leq f(n)$

(3)  $f(n)=o(g(n))$

对所有正数  $c > 0$  存在  $n_0$  使得对一切  $n \geq n_0$  有  $0 \leq f(n) < cg(n)$

(4)  $f(n)=\omega(g(n))$ .

对所有正数  $c > 0$  存在  $n_0$  使得对一切  $n \geq n_0$  有  $0 \leq cg(n) < f(n)$

(5)  $f(n)=\Theta(g(n)) \Leftrightarrow f(n)=O(g(n))$  且  $f(n)=\Omega(g(n))$

(6)  $O(1)$  表示常数函数



# 函数渐近的界的基本性质

**定理** 设  $f$  和  $g$  是定义域为自然数集  $N$  上的函数.

(1) 如果  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  存在, 并且等于某个常数  $c > 0$ , 那么

$$f(n) = \Theta(g(n)).$$

(2) 如果  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , 那么

$$f(n) = o(g(n)).$$

(3) 如果  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty$ , 那么

$$f(n) = \omega(g(n)).$$

# 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

# 语句频度和时间复杂度（举例）

例子:	程序	程序频度	时间复杂度	
1.	<code>x=x+1</code>	1	$O(1)$	常数阶
2.	<code>for i in range(n):</code>  <code>    x=x+1</code>	$n+1$  $n$	$T(n)=2n+1$  $O(n)$	线性阶
3.	<code>for i in range(n):</code>  <code>    for j in range (n):</code>  <code>        x=x+1</code>	$n+1$  $n(n+1)$  $n(n+1)$	$T(n)=2n(n+1)+(n+1)$  $O(n^2)$	平方阶

3种常见的错误：混淆频度和复杂度，保留常数，保留次要项

# 确定运行时间数量级大 $O$ 的方法

- 例 1:  $T(n)=n+1$

- 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  $O(n^2)$

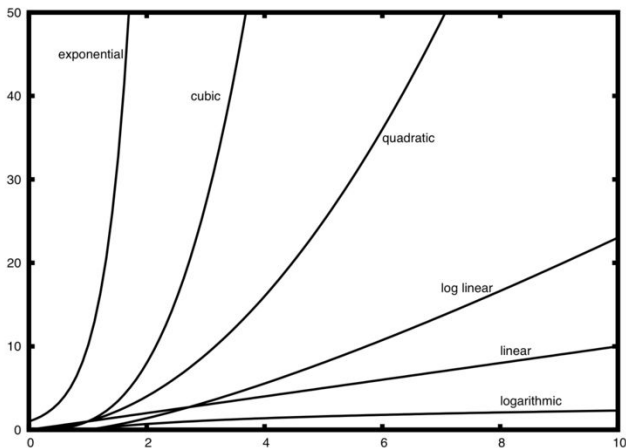
# 最好、平均、和最差时间复杂度

- 有时决定运行时间的不仅是问题规模，具体情况也会影响算法运行时间
  - 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

排序方式	时间复杂度			空间复杂度	稳定性	复杂性
	平均情况	最坏情况	最好情况			
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(n^{1.3})$			$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定	较复杂
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定	较复杂
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定	较复杂
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	稳定	较复杂

# 常见的大 O 数量级函数

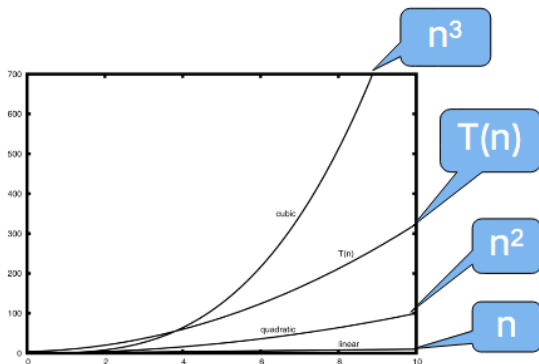
- 通常当  $n$  较小时, 难以确定其数量级
- 当  $n$  增长到较大时, 容易看出其主要变化量级



$f(n)$	名称
1	常数
$\log(n)$	对数
$n$	线性
$n \cdot \log(n)$	对数线性
$n^2$	平方
$n^3$	立方
$n^k$	多项式
$2^n$	指数
$n!$	阶乘

# 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
  - $T(n) = 3 + (n + 4n^2) + 3n + 1 = 4n^2 + 4n + 4$
- 仅保留最高阶项  $n^2$ ，去掉所有系数
- 数量级为  $O(n^2)$



```
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
```

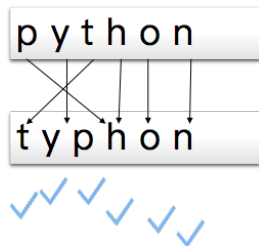
# “变位词” 判断问题

- “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

```
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'))
```

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

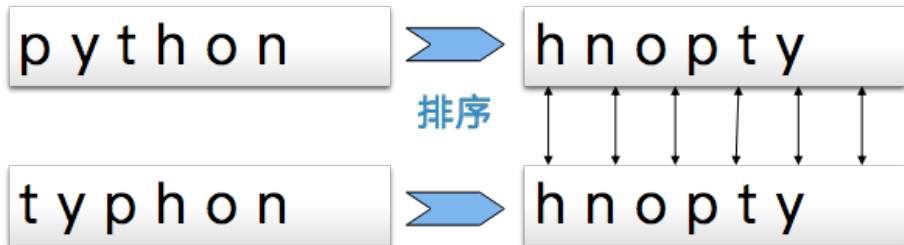
# 解法 1: 逐字检查-算法分析

- 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
- **True:** 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
- **False:** 提前返回, 执行次数小于 **True**。
- 所以总执行次数是  $1+2+3+\dots+n$
- 可知其数量级为  $O(n^2)$

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

## 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

```
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'))
```

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。
- 执行次数：  $n! * n$
- 结论：暴力法恐怕不能算是个好算法
- 密码学领域：运算的逆，只有“暴力法”求解。

## 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

```
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'))
```

分别都计数

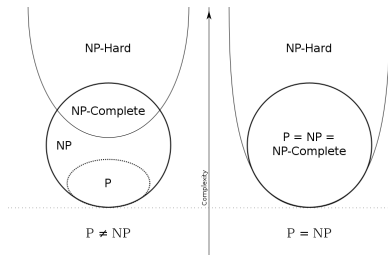
计数器比较

## 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。

# NP, NP-Complete 和 NP-hard

- 所有问题都可以表述为判定问题
  - 哈密顿路径/回路问题
  - 邮差问题（判定）：是否存在小于  $D$  的哈密顿回路
- P 问题：问题的求解，可以在多项式时间内解决
- NP 问题：给一个解，可以在多项式时间内判定（验证），“N”  $\neq$  “not”
- NP-hard 问题：任意的 NP 问题，都可以在多项式时间内归结到该问题
- NP-Complete 问题：同时是 NP 和 NP-hard 的问题
- NP vs. P：计算机理论的一个主要未解难题。



# 当我们遇到一个新问题的时候

- 首先想想是不是 NP
  - 给一个解，是否有  $p$  算法能判定
- 再想想是不是 P
  - 找一个  $P$  的算法，别管是多少次项
- 其次想想是不是 NP-hard
  - 是否能把一个 NPC 问题，归结到这个问题

## ● 常见的 NPC 问题

- SAT
- 背包问题
- 哈密顿路径、邮差问题
- 子图匹配问题 (subgraph isomorphism)
- 子集求固定和问题 (subset sum)
- 完全子图问题 (clique problem)
- 最小点覆盖问题 (minimal vertex cover)
- 独立集合问题 (independent set)
- 控制集问题 (Dominating set)
- 图着色问题 (Graph coloring)

# 多项式函数与指数函数

时间复杂度函数	问题规模					
	10	20	30	40	50	60
$n$	$10^{-5}$	$2*10^{-5}$	$3*10^{-5}$	$4*10^{-5}$	$5*10^{-5}$	$6*10^{-5}$
$n^2$	$10^{-4}$	$4*10^{-4}$	$9*10^{-4}$	$16*10^{-4}$	$25*10^{-4}$	$36*10^{-4}$
$n^3$	$10^{-3}$	$8*10^{-3}$	$27*10^{-3}$	$64*10^{-3}$	$125*10^{-3}$	$216*10^{-3}$
$n^5$	$10^{-1}$	3.2	24.3	1.7 分	5.2 分	13.0 分
$2^n$	.001 秒	1.0 秒	17.9 分	12.7 天	35.7 年	366 世纪
$3^n$	.059 秒	58 分	6.5 年	3855 世纪	$2*10^8$ 世纪	$1.3*10^{13}$ 世纪

表中默认单位为秒

# 多项式函数与指数函数

时间复杂度函数	1小时可解的问题实例的最大规模		
	计算机	快100倍的计算机	快1000倍的计算机
$n$	$N_1$	$100 N_1$	$1000 N_1$
$n^2$	$N_2$	$10 N_2$	$31.6 N_2$
$n^3$	$N_3$	$4.64 N_3$	$10 N_3$
$n^5$	$N_4$	$2.5 N_4$	$3.98 N_4$
$2^n$	$N_5$	$N_5 + 6.64$	$N_5 + 9.97$
$3^n$	$N_6$	$N_6 + 4.19$	$N_6 + 6.29$

# 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 + [v1, v2, ..., vk]`, 执行时间是  $O(n+k)$ ,  $k$  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))
```

# 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

- 我们注意到 **pop** 这个操作
  - **pop()** 从列表末尾移除元素,  $O(1)$
  - **pop(i)** 从列表中移除元素,  $O(n)$
- 原因在于 **Python** 所选择的实现方法 (\* 食堂排队)
  - 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

```
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

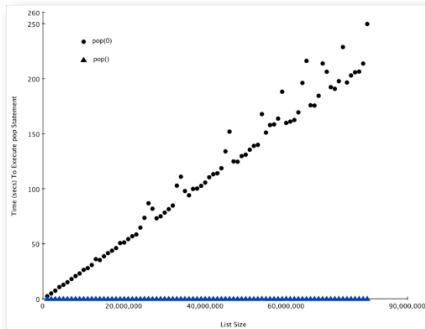
- 我们通过改变列表的大小来测试两个操作的增长趋势

```
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对比，从不同的容器 List 与 Dictionary 中检索一个值花的时间。
  - 生成一个包含 N 个连续值的 List 和一个包含 N 个连续关键码 Key 的 Dictionary，N 代表问题的规模。
  - 从容器中检索不同的 r 值往往花不一样的时间。有最好的情况，也有最坏的情况，我们随机产生一些 r 来检验，估计出一个“平均”的时间。
  - 我们安排 N 进行等差增长，取得每一个 N 下容器 in 操作的“平均”时间，进行比较。

```
import timeit
import random

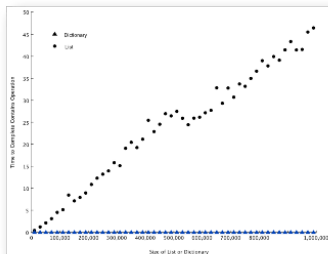
for i in range(10000,1000001,20000):
    t = timeit.Timer("random.randrange(%d) in x"%i,
                     "from __main__ import random,x")
    x = list(range(i))
    lst_time = t.timeit(number=1000)
    x = {j:None for j in range(i)}
    d_time = t.timeit(number=1000)
    print("%d,%10.3f,%10.3f"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

# 使用 `timeit` 来作算法计量，需要考虑到的问题

- 同一台计算机，CPU 负载波动影响算法  $F$  的运行时间。
- 问题  $F$  在规模为  $N$  的情况下重复 `number` 次：  
 $F(N) * \text{number}$
- 为什么要重复“`* number`”
  - 减少计算机负载波动带来的测量误差。
  - $F(N)$  可能有内部的 `cache` 机制：第一遍运行花时间，以后的运行可以是瞬间完成。  
所以需要在每次运行时加入一定的随机性，破坏掉 `cache`。见前面的“`in`”操作对比。
- 如果  $N$  超出了计算机内存的规模，运行效率会产生新的问题
  - 使用硬盘缓存、内存频繁扇入扇出，造成颠簸。
  - 扩大内存，也会显著提高计算机性能。

# 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

# Python 小知识：特殊方法

- `foo(obj)` 或 `foo obj`  $\implies$  `obj.__foo__()` 需要特殊函数做“内应”
- 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 提供了一些特殊机制，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

- continue ...

- 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：使用模块

- 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` 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：使用模块

- 并不是所有的模块都自带了，有时候 `import` 会报错，找不到所需的模块

```
>>> import matplotlib.pyplot as plt
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'matplotlib'
>>> from sys import path
>>> path
['', '/usr/local/Cellar/python3/3.6.4_2/Frameworks/Python.framework/Versions/3.6/lib/python36.zip', '/usr/local/Cellar/python3/3.6.4_2/Frameworks/Python.framework/Versions/3.6/lib/python3.6', '/usr/local/Cellar/python3/3.6.4_2/Frameworks/Python.framework/Versions/3.6/lib/python3.6/lib-dynload']
>>> from stack import Stack
>>>
Mini2-1:code zhengmaoxie$ ls
__pycache__      hw               hw.py            intRef.py        stack.pyc
aa.py            hw.c             in2post.py       postEval.py      varRef.py
except.py        hw.dSYM          indent.py         stack.py          while.py
Mini2-1:code zhengmaoxie$
```

- 可以通过 `pip` 安装没有找到的模块
  - `$python -m pip install -U matplotlib`



# 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