2 算法分析

##2.1 目标##
_理解算法的重要性
_使用"Big-O"来评估运行时长
_理解Python列表和字典的"Big-O"运行时长
_理解Python数据实现方式对算法分析的影响
_学习如何对简单的Python程序进行评分

2.2 何为算法分析

计算机科学的学生会把他们的程序与别人的进行对比。你也许已经注意到了，计算机程序也常看起来很相似，尤其是那些简单的程序。那么问题来了，对于两个解决相同问题的程序，应该如何进行优选?

为了回答这个问题，我们需要知道程序和其底层算法有重要的区别。如第1章所述，算法是用来求解某个问题的通用的分步命令的集合：对于该类问题的任一实例，给定输入，便可以获得预定结果。程序则是以某种编程语言表达的算法。同一算法可能有多个程序，这取决于程序员和其所使用的编程语言。

为了进一步探索他们的区别，试考虑代码2.1中的函数。这个函数求解一个大家都很熟悉的问题，计算前n个证书的和。此算法设定一个初始值为0的累计和，然后遍历该n个整数，将其依次加到累加和上。

代码1.1:对前n个整数求和

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

    return theSum

print(sumOfN(10))
```

接下来考虑代码2.2中的函数。乍眼一看似乎很奇怪，但是仔细观察便会发现该函数与上一个函数基本上做的是同一件事。之所以看起来不那么清晰，原因便是较差的代码质量。此段代码没有使用具有区分性的变量名来提高可读性，而且还使用了并非必要的冗余赋值语句。

代码2.2:前n个整数求和的另一种实现

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

    return fred

print(foo(10))
```

回到之前提出的问题，一个程序是否比另一个程序好？答案取决于你的评判标准。如果你关注于可读性，那么函数**sumOfN**当然比**foo**好。事实上，读者可能在编程入门课中学习了不少这种例子，因为这些课程的目标之一就是提高可读性。同时，本课程也将对算法本身的描述进行研究。

算法分析主要根据各算法所消耗的计算资源来进行对比。考虑两个不同的算法，如果其中一个能够更高效地使用计算资源或者直接地使用更少的资源，那么它就应该比另一个更优。从这个角度来看，上面这两个函数是非常蕾丝的。他们使用了相同的算法来解决求和问题。

现在是时候考虑下计算资源这个重要的概念的真正含义了。实际上它可以从两个角度来看待。一种角度将其解释为算法解决对应问题所需的空间或者内存大小。一般来说，算法所需空间大小由问题实例本身决定。然而，优势某些算法会有特殊的空间需求，必须谨慎地解释这种变化。

除了空间要求，也可以依据运行时间大小来对算法进行分析和比较。这种测量有时被称为算法对“执行时间”或者“运行时间”。其中一种方法通过基准分析（benchmark analysis）来测定函数**sumOfN**的执行时间，即是说追踪程序计算出结果所需实际时间。在Python中，可以依据当前的系统时间来标定程序的开始时间和结束时间，从而实现基准分析。在**time**模块中有一个名为**time**的函数，它会返回从某个起点算起的，以秒为单位的当前系统时间。通过分别在程序运行开始时和结束时调用该函数并计算两次结果的差值，便可得到精准到秒的运行时间（大多数时候为小数）。

代码2.3：

```
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
```

代码2.3演示了原**sumOfN**函数在嵌入了时间函数后的情况。该函数返回一个包含计算结果和运行时间的元组。如果连续调用5次该函数，每一个都计算前1万个整数的和，结果如下：

```
>>>for i in range(5):
 print("Sum is %d required %10.7f seconds"%sumOfN(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
```

可以看到，运行时间是相当一致的，大约都是0.0019秒。计算前10万个整数的结果如下：

```
>>>for i in range(5):
 print("Sum is %d required %10.7f seconds"%sumOfN(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
>>>
```

同样地，运行时间还是高度一致，并且平均来说大概为之前的10倍。那么计算前100万的结果呢?

```
>>>for i in range(5):
 print("Sum is %d required %10.7f seconds"%sumOfN(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
>>>
```

在这种情况下，运行时间大概是前者的10倍。

考虑代码2.4，它演示了求和的另一种方法。该函数，**sumOfN3**，利用公式$\sum_{i=1}^{n} i = \frac{(n)(n+1)}{2}$来计算前n个整数之和而不用迭代。

对**sumOfN3**进行同样的基准测试，并且使用不同的n值(10,000、 100,000、 1,000,000、 10,000,000、 100,000,000)，所得结果如下：

```
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
```

结果有两个非常重要的地方需要注意。首先，运行时间比之前任意的例子都短。其次，不论n值为何，运行时间都高度一致。看起来不论n值为何，**sumOfN3**都几乎不受影响。

那么这个基准测试说明了什么？直观上，可以注意到迭代算法似乎做了更多工作，因为有很多步骤都是不断重复的，似乎这就是其耗时更久的原因。并且，迭代解法所需时间随着n的增加而增加。还有一个问题是，如果同一个函数在不同的电脑上运行或者使用不同的编程语言实现，很有可能结果也是不一样的。显然，计算机计算能力越弱，其运行**sumOfN3**的时间更久。

有必要找到一个根据运行时间来定量描述这些算法的模式。基准测试技术计算了运行所费的实际时间。它并不能提供真正的有效测量，因为它依赖于具体的机器，程序，日期，编译器以及编程语言。实际上需要的是不依赖于所使用的程序或者计算机的描述方式。这种测量可以用来单独对算法本身进行评价并且在不同的具体实现上对算法进行对比。

2.3 Big-O 表示法（高阶表示法）

若在独立于所使用的具体程序或者计算机的条件下，按照运行时间来对算法效率进行描述，确定该算法所需要的操作或者步骤数便显得尤为重要。如果每一步都被视为基本的计算单位，那么运行时间便可以表达为解决该问题所需要的步骤数。确定合适的基本计算单位可能是相当麻烦的，并且它还取决于该算法实现的方式。

对于上文所示的求和算法来说，比较好的基本计算单位也许是求和所需进行的赋值语句的总数。在**sumOfN**函数中，赋值语句总数是1（theSum = 0）加n（执行theSum = theSum + i的次数）。以函数来表示并记为T，其中T(n) = 1 + n。参数n一般指“问题的规模”，可以理解为“T(n)是解决规模为n的该问题实例所需的时间，即 1+n 步”。

在之前的那些求和函数中，用求和的项数来表示问题的规模是合理的。可以说，计算前100，000个整数之和的问题规模比计算前1，000个整数之和这一问题大。因此，看起来求解大规模问题所需时间比小规模问题要大得多。于是目标之一便是该算法的运行时间如何随着问题的规模而变化。

计算机科学家对这种分析技术做了进一步研究。很显然，求出算法的精准步数的重要性远远低于确定T(n)函数中起主导作用的那部分。换句话说，问题规模越大，T(n)函数中的一部分会倾向于掩盖另一部

分的作用。而这种起主导作用的部分，最终被用来进行对比。**量级函数**用来描述T(n)中随着n增长而增长得最快的那部分。该函数的量级也常被称为**高阶表示法**，并且记为O(f(n))。它给出了计算过程所需的实际步数的可靠近似。函数f(n)是原始T(n)函数中起主导作用那部分的简化表示。

在上面的例子中，T(n)=1+n。随着n增加，常数项1对最终结果的影响变得越来越小。可以将1舍去，然后将其作为T(n)的近似并称为O(n)。即使是1，对T(n)来说也是有一定意义的。然而当n足够大时，略去1的近似结果的精度几乎不受影响了。

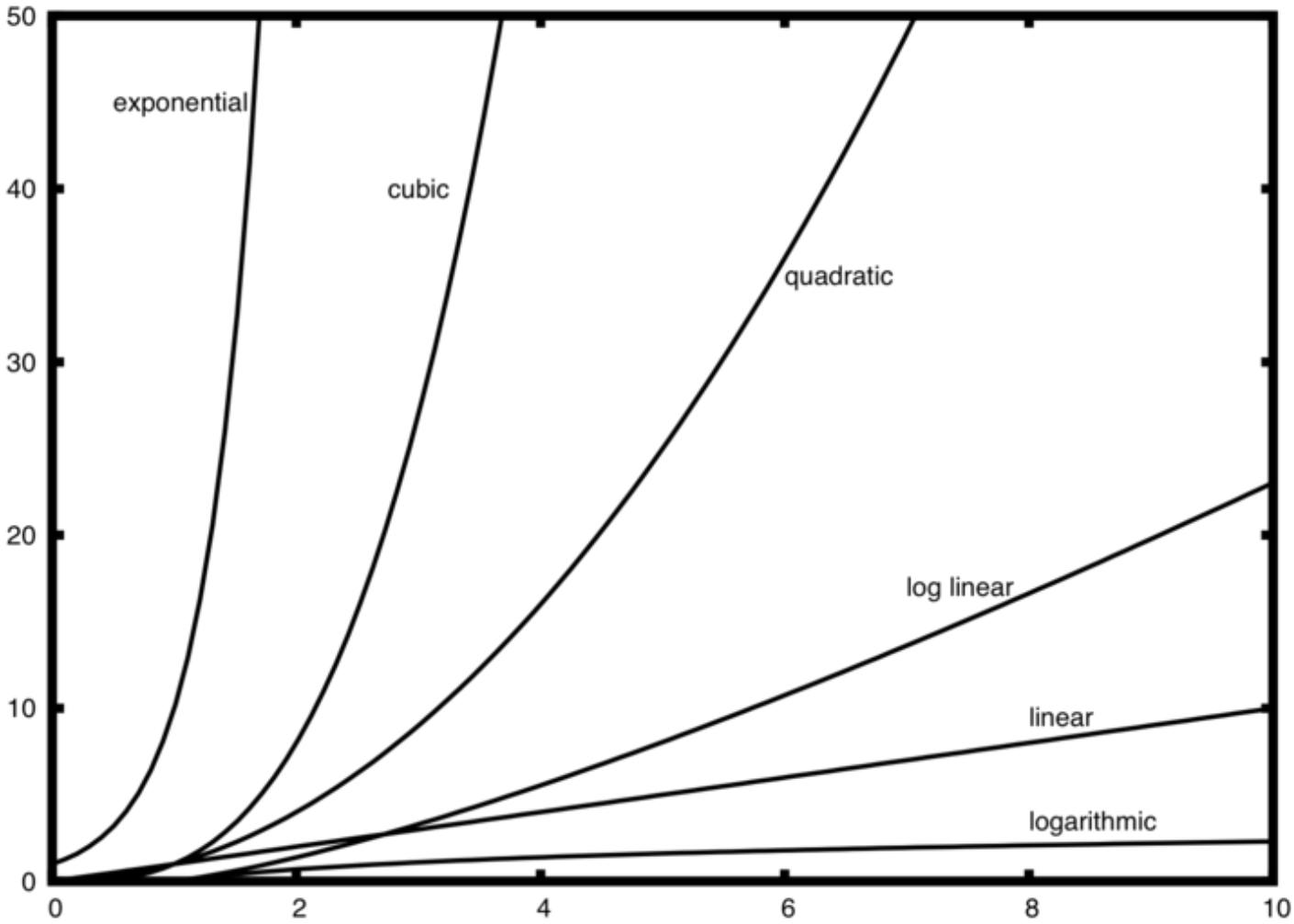
另外考虑一个例子，假设对某个算法，其准确的步数是$T(n) = 5n^2 + 27n + 1005$。当n很小时，比如说1或者2，常数项1005似乎是该函数起主要作用的部分。然而，随着n的增加，n^2项成为了最重要的那部分。实际上，当n足够大时，另两项对最终结果的影响就很微弱了。同样地，为了在n增加的过程中近似T(n)，可以略去其他项而重点关注$5n^2$。此外，系数5的作用也会在n变大的过程中降低。因此，T(n)的量级为$f(n) = n^2$，也可以简称为$O(n^2)$。

尽管在求和的例子中不会出现，但是有时确实存在一些情景，某个算法的性能取决于数据的具体值而不是其规模。对这些类别的算法的性能应该分别按照最佳情况，最差情况或者平均情况来进行描述。最差情况是指，某个特殊的数据集导致该算法运行效果极差，不过另一个不同的数据集在相同的算法上可能会有非常好的性能。然而在大多数情况下，算法都是在这两种极端状态中间的某个状态下，也就是所谓的平均状态。计算机科学家必须理解这些区别，以免被某个特例所误导。

在学习算法的过程中，有很多常见的量级函数会一次又一次地出现。如下表2.1所示。为了确定这些函数中哪些会在T(n)中起主要作用，必须它们随n的增大而增大的速度。

f(n)	Name
1	常数
logn	对数
n	线性
nlogn	线性对数
n^2	二次函数
n ^3	三次函数
2^n	指数函数

图1给出了上表中各函数的图像。注意到当n很小的时候，它们彼此之间很难以区分，也就没办法确定起主要作用的那部分。然而，当n增大时，便会有明显的区别了。



最后再举一个例子，假设有如下所示的一段Python代码。尽管它没啥实际意义，但确实可以用来演示对实际代码进行分析。

代码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
```

赋值语句总数是由四部分构成的。第一部分是常数3，即该代码段的开始部分的前三个赋值语句。第二部分是n^2，因为嵌套迭代的存在，有3个语句执行了n^2次。第三部分是2n，有两个语句迭代了n次。最后，第四项是常数1，即最后一个赋值语句。最终得到：

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

通过观察指数，可以明显地发现n^2是起主导作用的那部分，因此该代码片段量级为$O(n^2)$。注意，当n变大时，起主导作用的那项的系数是可以省略的。

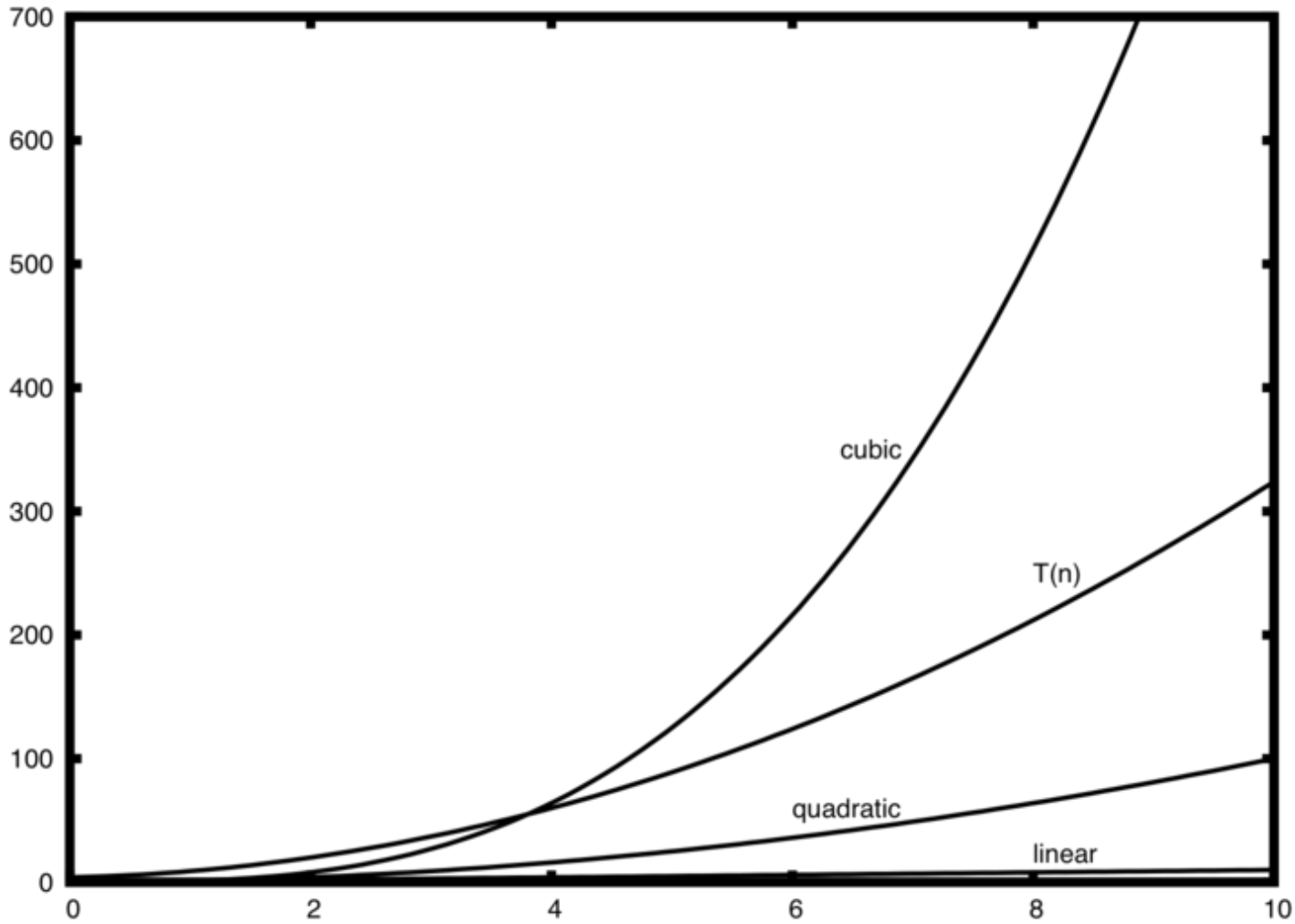


图2给出了几个常用的高阶函数与上文讨论的$T(n)$函数的对比。可以看到，$T(n)$函数在起始的时候比立方函数大。然而，当n变大时，立方函数迅速地超越了$T(n)$。不难看出，当n继续增大时，$T(n)$函数会越来越接近于二次函数。

2.4 实例：变位词检测

经典的字符串变位词检测是一个演示不同数量级算法的好例子。如果一个字符串是另一个字符串的重新组合，那么它们便互为变位词。比如说，'heart'和'earth'，'python'和'typhon'都是变位词。为了简化问题，假定该问题下每一对字符串的长度相同，并且都是由26个小写祖母组成。目标是写出一个布尔函数，输入两个字符串并返回它们是否为变位词。

2.4.1 解法1: 核定

第一种解法是检查在第一个字符串出现的字母是否也在第二个字符串中出现了。如果每一个字母都“核对”成功，那么这两个字符串便互为变位词。每核对完一个字母都将其用Python特殊值None来替换掉。然而，由于Python中的字符串是不可变的，所以首先要将另一个字符串转为列表。第一个字符串中的每一个字母都将与列表中的字母进行对比，如果找到了就用None替换掉。可执行代码1演示了该函数：

```
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'))
```

可执行代码1: 逐一核定

接下来对该算法进行分析。注意s1的n个字母都会导致一次对s2中的至多n个字母的遍历。而列表中的n个元素，每一个都将被访问一次以检查是否匹配s1中的字母。因此，执行总数便是1到n之和。可以写作：

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

随着n增大，n^2将超过n的作用，并且系数$\frac{1}{2}$可以被省略。因此该解的量级为$O(n^2)$。

2.4.2 解法2:排序比较

考虑到s1和s2虽然不同，但它们如果是变位词的话必定有相同的字母。因此，从a到z按字母顺序排序，若两个词互为变位词，则排序后它们将完全一致。可执行代码2演示了该解法。Python中可以使用列表内置的**sort**方法。

```
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'))
```

可执行代码2:排序比较

乍看之下，可能回认为该算法为$O(n)$，因为在排序后仅有一个迭代。然而，Python中的**sort**方法本身也会消耗一定资源。在后面的章节将会说明，排序一般来说不是$O(n^2)$就是$O(nlogn)$，因此在这里排序操作比迭代消耗更多的资源。但是最终，算法的量级跟排序过程是一样的。

2.4.3 解法3:暴力法

暴力法一般来说就是尝试所有的可能。对于变位词检测问题，考虑s1中字母付出所能构成的所有字符串，将他们组合为一个列表，检测s2是否在其中。然这种方法有个难点。当从s1中创建字符串时，第一个字母有n种可能，第二个字母有n-1个可能...以此类推。所有可能的字符串总数为$n*(n-1)*(n-2)*...*3*2*1$，即$n!$。尽管有些字符串是重复的，程序也不能提前对此进行判断，因此它还是将生成$n!$个不同的字符串。

$n!$比n^2增长得快。实际上，如果s1中有20个字母，那么将会有$20! = 2,432,902,008,176,640,000$个字符串组合。如果每秒处理1个，也将需要77,146,816,596年来遍历整个列表。这显然不是个好解

法。

2.4.4 解法4:计数对比

本书给出的变位词检测问题的最后一种解法是计数对比法。如果两个词为变位词，那么它们的a字母、b字母、c字母等等个数都将是完全一致的。为了确定两个字符串是否是变位词，首先要统计每个字母出现的次数。因为一共有26个可能的字母，所以可以创建一个包含26个计数结果的列表，与每一个字母一一对应。某个字母每出现1次，便在其对应的计数位置上加1。最后，如果两张计数表完全一致，这两个字符串就应是变位词。可执行代码3演示了该解法。

```
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'))
```

该解法也有很多迭代过程。然而，跟第一种算法不同的是，它并没有嵌套迭代。用于对字母计数的前两个迭代都是n次的。第三个迭代用于对比两个计数表，始终是26步的。因此总计为$T(n) = 2n + 26$步。该算法的数量级是线性的。

在结束这个例子前，再谈谈空间需求的问题。尽管最后一种解法以线性时间复杂度永兴，但是它是通过使用了额外的空间来保存字母计数表。换句话说，这个算法牺牲了空间来换取时间。

这是经常出现的一种情况。在很多情境下，都需要在时间和空间中进行取舍。在本例中，额外使用的空间并不大。然而，如果基本字母有上百万的话，那就需要注意了。作为计算机科学家，在对算法进行选择时，需要由你在给定问题下最佳效率地使用计算机资源。

2.5 Python 数据结构的性能

现在读者应对高阶复杂度表示法和不同的复杂度函数有了一定了解，本节介绍Python列表和字典的高阶复杂度。基于计时实验，演示特定的操作在各数据结构上的运行的代价和好处。读者有必要清楚地了解这些Python数据结构的效率，因为它们将是实现自定义数据结构的基本构件。本节不会解释它们的性能为何会是这样的。在后续章节，读者将会遇到一些利用了列表和词典的实现，并且发现其性能是取决于实现的方式。

2.6 列表

Python的设计者在实现列表数据结构时有很多选择。每一个选择都会对列表的操作性能造成影响。他们研究了列表数据结构最经常被使用的方式，并依此来优化了列表的实现方式，所以列表的常用操作运行效率都很高。当然，他们也尽力提高了那些不怎么常用的操作的运行速度，但是如果必须进行取舍时，不常用的操作速度往往被牺牲了以换取更高的常用操作运行速度。

对列表上的某个位置进行索引和赋值是两个很常见的操作。无论列表规模有多大，这两种操作的时间消耗都是一样的。当某个操作的运行速度与列表规模无关时，其时间复杂度就是O(1)。

另一个常见的操作是对列表进行扩充。可以使用**append**方法或者**concatenation**方法来实现。append方法为O(1)，而concatenation为O(k)，其中k为被连接的列表的规模。选择正确的工具可以使程序运行更加有效率。

下面试着用4种方法生成从0到n的列表。首先，利用for循环以及concatenation生成，然后试着将concatenation换位append；接下来，使用列表推导式生成列表；最后，使用封装了列表构造器的**range**函数。代码3演示了以上操作。

代码3

```
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))
```

可以使用Python的**timeit**模块来追踪每个函数的运行时间。通过在稳定一致的环境中运行函数并且使用尽可能相同的计时机制，实现跨平台的运行时间测量。

为了使用timeit，首先要创建一个接收两个参数的**Timer**对象。第一参数是待测的Python语句，第二个参数是一段用来初始化该测试所需要的语句。接下来**timeit**模块便会执行待测语句一定次数，并且对每次运行进行计时。缺省为执行100万次。执行完毕，将总运行时间按秒计以浮点数的形式表示出来。但是，由于它默认运行待测语句100万次，所以可以将结果视为运行一次该语句所需要的微秒数。也可以向**timeit**传入**number**参数指定待测语句执行次数。下面的代码演示了每个待测函数运行1000次所需时间。

```
t1 = Timer("test1()", "from __main__ import test1")
print("concat ",t1.timeit(number=1000), "milliseconds")
t2 = Timer("test2()", "from __main__ import test2")
print("append ",t2.timeit(number=1000), "milliseconds")
t3 = Timer("test3()", "from __main__ import test3")
print("comprehension ",t3.timeit(number=1000), "milliseconds")
t4 = Timer("test4()", "from __main__ import test4")
print("list range ",t4.timeit(number=1000), "milliseconds")

concat  6.54352807999 milliseconds
append  0.306292057037 milliseconds
comprehension  0.147661924362 milliseconds
list range  0.0655000209808 milliseconds
```

可以看到，在待测语句之前先是调用了函数test1(), test2()等。后面的初始化语句可能看起来有点奇怪，下面将详细地说明。读者大概已经对from, import语句比较熟悉了，它们一般来说是被用在Python程序文件的开始。在此处，语句**from __main__ import test1**从命名空间__main__中将函数**test1**导入到**timeit**为测时实验所准备的命名空间里。**timeit**模块之所以这样做，是为了将测试实验运行在一个去除一些无用、干扰变量的环境中，以避免它们对结果造成不可预测的影响。

从上面的实验可以看出，**append**操作比**concatenation**操作快得多（0.3毫秒：6.54毫秒）。此外还有两种方式，即调用**range**中的列表构造器以及使用列表推导式。有趣的是，列表推导式是for循环加append函数的2倍速度。

还有最后一点是，以上实验结果都包括了调用test函数所耗费的时间。但是可以认定，对于这四次实验，每个实验在调用test函数所耗费的时间是相同的，因此结果的对比是具有意义的。因此说

concatenation函数耗时6.54毫秒是不准确的，而应该说concatenation测试函数耗时6.54毫秒。作为练习，可以对空函数进行测时，然后将其从之前的测试结果减掉。

现在读者应该已经明白了如果对性能进行准确的测量。表2给出了所有的基本列表操作的时间复杂度。仔细考察表2，可以看到比较奇怪的是，**pop**有两个时间复杂度。若**pop**在列表的末尾被调用，其复杂度为O(1)，若是在开头或者列表中间的任何位置，其复杂度又是O(n)。原因在于Python对列表实现方式的选择。按Python的实现方式，若某项从列表的开头移除，那么其它的所有项都必须向前挪动一位。这可能看起来傻傻的，但是这样一来，索引操作的复杂度便成了O(1)。这是一项不错的权衡。

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)

为了演示这种在性能上的差距，再次使用timeit模块进行一次实验。目标是验证在已知规模的列表上分别对末尾和开头进行**pop**操作。同时也测定了不同规模列表所需的时间。预测的结果是，在末尾进行**pop**操作所需时间与列表规模无关，而在开头进行的则随着列表规模增大而增大。

代码4按此进行了测定。从第一个例子可以看出，从末尾**pop*8耗时0.0003毫秒，而从开头则耗时4.82毫秒。对于200万规模的列表来说，相差可达16,000倍。

对于代码4，还有更多地方值得注意。首先是语句**from __main__ import x**。虽然并没有在测试中定义函数，但在测试仍将使用对象x。这种方法可以保证仅对**pop**语句本身进行测定，并且尽可能地保证准确性。因为要循环1000次，因此必须指出列表的规模在循环中每次都将-1。但考虑到初始列表归为为200万，所以对整体的影响仅0.05%。

代码4

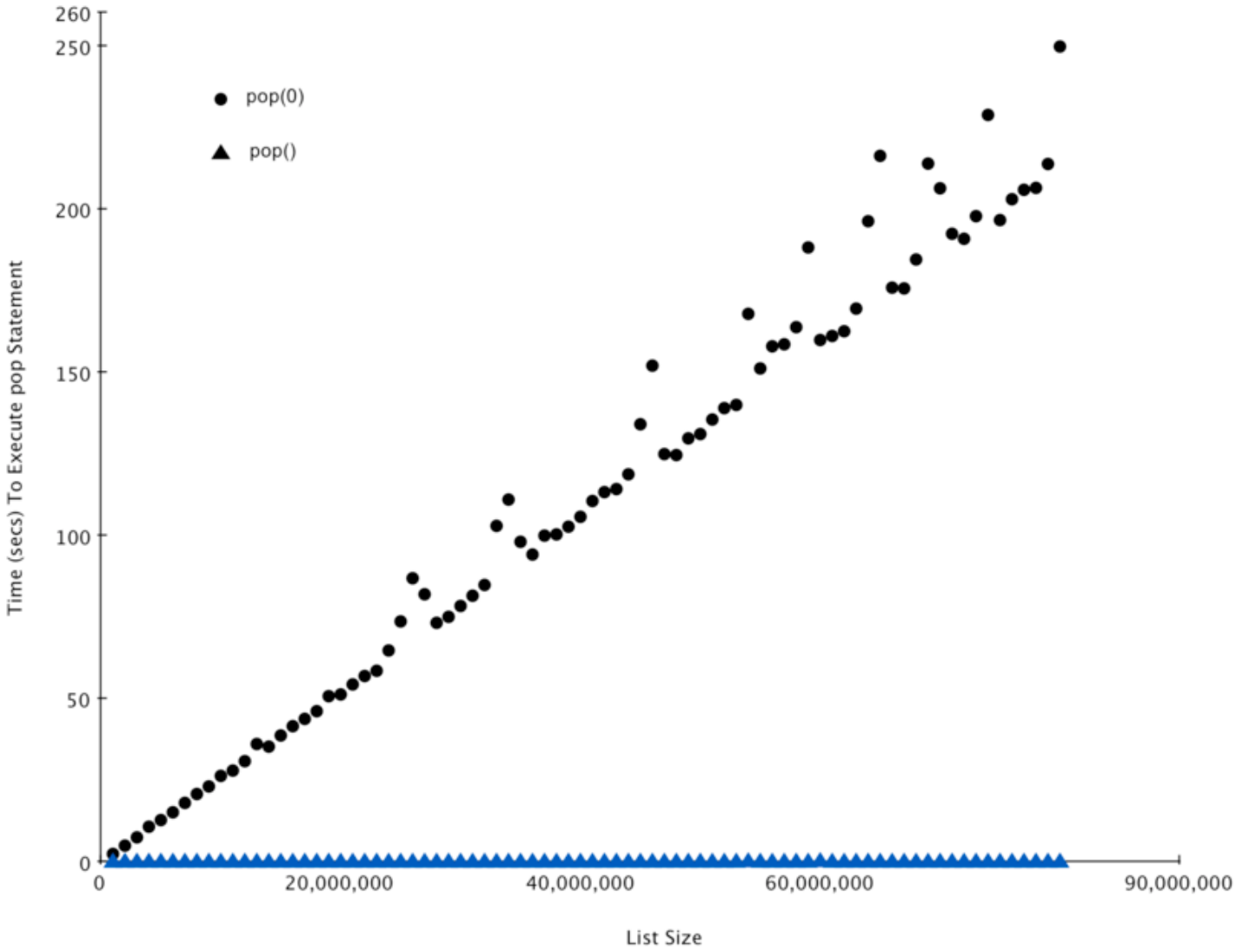
```
popzero = timeit.Timer("x.pop(0)",
                       "from __main__ import x")
popend = timeit.Timer("x.pop()",
                      "from __main__ import x")

x = list(range(2000000))
popzero.timeit(number=1000)
4.8213560581207275

x = list(range(2000000))
popend.timeit(number=1000)
0.00031614303588867 19
```

第一个测试表明pop(0)确实比比pop()慢，然而并不能证明pop(0)复杂度就是O(n)而pop()是O(1)。为了对此进行证明，必须研究在不同规模下的列表下这两者的性能。代码5对此进行了演示。

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

图3给出了实验结果。可以发现随着列表规模的增大，pop(0)耗时越来越大，而pop()耗时则是几乎不变的。这与预料的一致。

实验中的一些误差可能来源于在进行测试时所运行的其它程序，它们会降低程序的运行速度。这也是为何要运行1000次的最重要的原因，即以统计学的方式获得足够多的信息来保证测量的可靠性。



2.7 字典##

第二个主要的Python数据结构是字典。读者大概还记得，字典与列表的区别之一是，列表可以通过键来访问其中的项而不是位置。本书的后续章节将介绍字典的很多种实现方式。目前最重要的就是明白字典中的get和set操作时间复杂度都是O(1)。字典的另一个重要的操作是contain，即检查某键是否存在于字典中，其时间复杂度也为O(1)。所有字典操作的时间复杂度如表3所示。值得注意的是，这里列出的都是平均时间复杂度。在某些情况下，contain，get，set操作都可能退化为O(n)。这些在后面的章节将会仔细介绍。

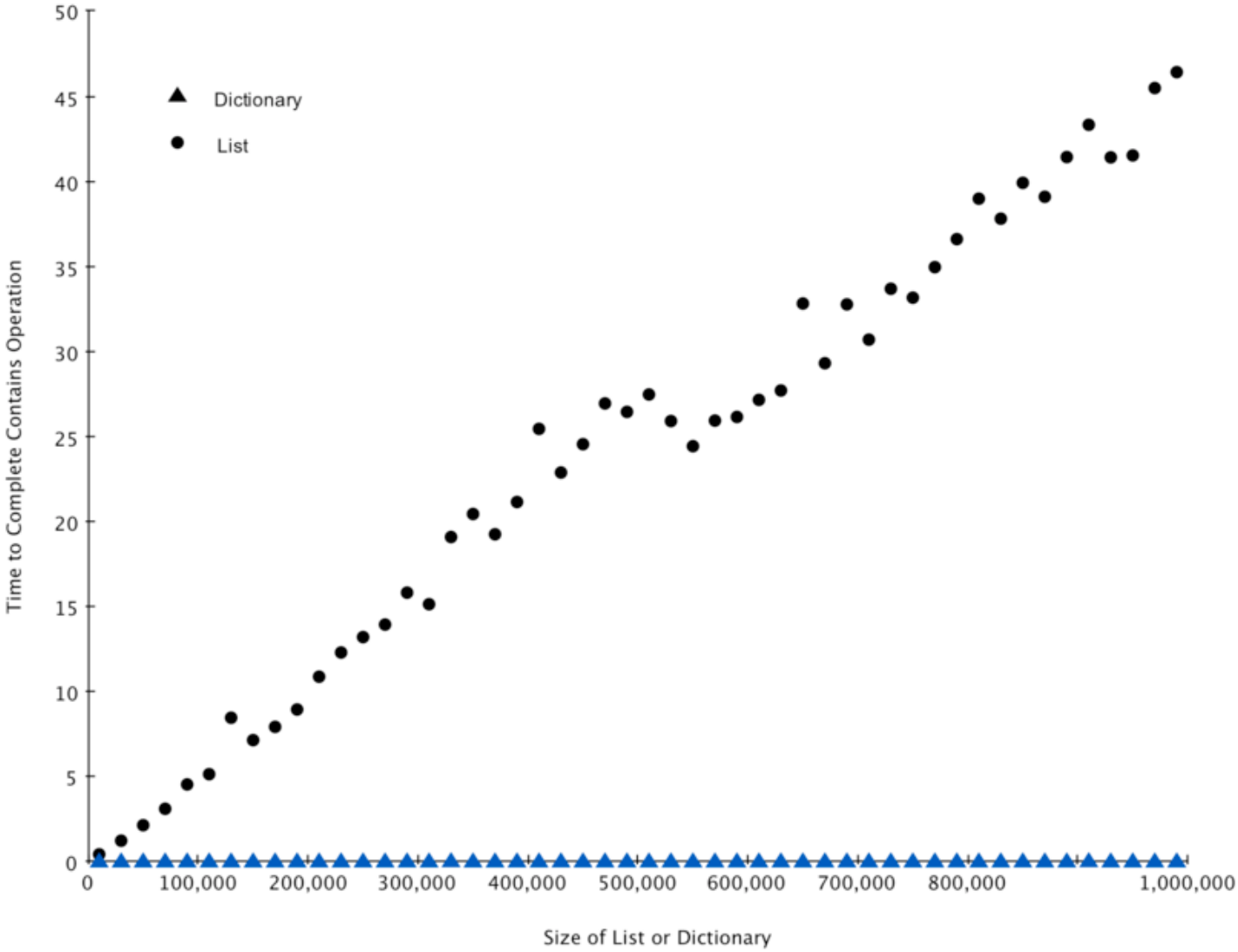
operation	Big-O Efficiency
copy	O(n)
get item	O(1)
set item	O(1)
delete item	O(1)

operation	Big-O Efficiency
contains (in)	O(1)
iteration	O(n)

在本章的最后一个实验，将对列表和字典的contain操作性能进行对比，确认列表为O(n)而字典为O(1)。如代码6所示。

```
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))
```

图4给出了表6的结果。可以看到，字典与预期取得了一致，运行速度更快。对于最小的含有10,000个元素的字典和列表来说，前者比后者快了89.4倍。同样可以发现，contain操作符的时间复杂度在列表中是与列表规模成线性正相关的，这也证实了列表的contain操作符的时间复杂度是O(n)。然而，字典的contain操作符消耗的时间并不随字典的规模变化而变化。实际上，10,000规模和990,000规模的字典，它们的时间复杂度是一致的。



../_images/listvdict.png

因为Python本身是不断发展中的，也就是说不断地改变。Python数据结构的性能信息可以在Python官网上看到。在本文写作之时，Time Complexity Wiki词条上对时间复杂度有着不错的解释。

2.8 总结

- 算法复杂度分析与算法实现方式无关
- 时间复杂度标识法通过识别算法中起主要作用的那部分来对算法进行评价