数据结构与算法

算法分析

张晓平

武汉大学数学与统计学院

2017 年 9 月 29 日

目录

1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

目标

- 理解算法分析的重要性
- 能够使用大 O 符号描述算法执行时间
- 理解 Python 列表和字典的常见操作的大 O 执行时间
- 理解 Python 数据的实现是如何影响算法分析的。
- 了解如何对简单的 Python 程序做基准测试（benchmark）。

1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

什么是算法分析

问题 1　编写程序，计算前 n 个整数的和。

什么是算法分析

问题 1 编写程序，计算前 n 个整数的和。

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

print(sumOfN(10))
```

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

print(foo(10))
```

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

print(foo(10))
```

编码习惯不好，没有使用好的标识符来提升可读性，循环中使用了一个额外的赋值语句，这没有必要。

算法分析是基于每种算法使用的计算资源量来比较算法。我们说一个算法比另一个好，在于前者在使用资源方面更有效，或使用的资源更少。

从这个角度来看，上述两个函数很相似，都使用基本相同的算法来解决求和问题。

我们需要更多地考虑真正意义上的计算资源，有两种方式：

1. 空间复杂度，即算法解决问题所需的空间或者内存，通常由问题本身决定。有时一些算法会有特殊的空间需求，需要特别注意。
2. 时间复杂度，即算法执行所需的时间，可以通过基准分析（benchmark analysis）来测试程序的执行时间。

```
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
```

该函数嵌入了时间函数，返回一个包含了执行结果和消耗时间的元组。

执行上述函数 5 次，每次计算前 10,000 个整数的和：

```
>>>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
```

时间相当一致，平均需要 0.0019 秒。

若计算前 100,000 个整数的和:

```
>>>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
```

尽管时间更长（平均大约多 10 倍），但每次运行的时间仍非常一致。

若计算前 1000,000 个整数的和:

```
>>>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
```

平均运行时间也大约是前一次的 10 倍。

现改用求和公式

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

来设计函数 sumOfN3()。

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

print(sumOfN3(10))
```

对 sumOfN3() 做同样的基准测试，使用 5 个不同的 n (10 000, 100 000, 1 000 000, 10 000 000 和 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
```

对 sumOfN3() 做同样的基准测试，使用 5 个不同的 n (10 000, 100 000, 1 000 000, 10 000 000 和 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
```

输出结果表明：

1. 执行时间比之前任何例子都短
2. 执行时间与 n 无关。

这个基准测试告诉我们:

- 直观上看，迭代方案需要做更多的工作，因一些步骤被重复执行。
- 迭代方案的执行时间随 n 递增。
- 使用不同程序语言，或在不同计算机上执行，可能会导致不同的执行时间。

这个基准测试告诉我们:

- 直观上看，迭代方案需要做更多的工作，因一些步骤被重复执行。
- 迭代方案的执行时间随 n 递增。
- 使用不同程序语言，或在不同计算机上执行，可能会导致不同的执行时间。

基准测试计算的是程序执行的实际时间，它并不能真正地提供一个有用的度量，因为它取决于特定的机器、程序、时间、编译器和编程语言。

这个基准测试告诉我们:

- 直观上看，迭代方案需要做更多的工作，因一些步骤被重复执行。
- 迭代方案的执行时间随 n 递增。
- 使用不同程序语言，或在不同计算机上执行，可能会导致不同的执行时间。

基准测试计算的是程序执行的实际时间，它并不能真正地提供一个有用的度量，因为它取决于特定的机器、程序、时间、编译器和编程语言。

我们希望能找到一种独立于程序或计算机的度量方式，用于比较不同算法的效率。

1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

大 O 符号

当我们希望独立于任何特定程序或计算机来描述算法的时间复杂度时，重要的是量化算法所需操作或步骤的数量。选择适当的基本计算单位是个复杂的问题，并且将取决于如何实现算法。

大 O 符号

对于求和算法，一个比较好的基本计算单位是对执行语句进行计数。

在 sumOfN() 中，总共有 $n+1$ 条赋值语句：1 次 `theSum = 0`，n 次`theSum = theSum + i`。

用函数 T 表示：$T(n)=1+n$，其中 n 称为‘问题的规模’，称‘T(n) 是解决规模为 n 的问题的时间复杂度’。

在求和函数中，使用 n 来表示问题的大小是有意义的。可以说，求 100 000 个整数和比求 1000 个整数的问题规模大，所需时间也更长。

我们的目标是表示出算法的时间复杂度是如何相对问题规模大小而改变的。

大 O 符号

计算机科学家更喜欢将这种分析技术进一步扩展。事实证明，确定 $T(n)$ 最主要的部分更为重要。

换句话说，当问题规模变大时，$T(n)$ 函数某些部分会超过其他部分。函数的数量级表示了随着 n 的值增加而增加最快的那些部分。

数量级通常称为大 O 符号，写为 $O(f(n))$。它表示对计算中的实际步数的近似。函数 $f(n)$ 提供了 $T(n)$ 最主要部分的表示方法。

大 O 符号

利用求和公式时，

$$T(n) = 1 + n.$$

当 n 变大时，常数 1 对于最终结果变得越来越不重要。

如果我们找的是 $T(n)$ 的近似值，我们可以删除 1，运行时间是 $O(n)$。

要注意，1 对于 $T(n)$ 肯定是重要的，但当 n 变大时，如果没有它，我们的近似也是准确的。

大 O 符号

假设某个算法确定的步数是

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

当 n 很小时，例如 1 或 2 ，常数 1005 似乎是函数的主要部分。

然而，随着 n 变大，n^2 这项变得越来越重要。事实上，当 n 真的很大时，其他两项在它们确定最终结果中所起的作用变得不重要。

当 n 变大时，为了近似 $T(n)$，我们可以忽略其他项，只关注 $5n^2$ 。系数 5 也变得不重要。

我们说，$T(n)$ 具有的数量级为 $f(n) = n^2$ 或者 $O(n^2)$ 。

大 O 符号

有时算法的性能取决于数据的确切值，而不是问题规模的大小。对于这种类型的算法，我们需要根据最佳情况、最坏情况或平均情况来表征它们的性能。

- 最坏情况是指算法性能特别差的特定数据集。
- 最佳情况是指算法性能特别好的特定数据集。
- 大多数情况下，算法执行效率处在两个极端之间（平均情况）。

对于计算机科学家而言，需要了解这些区别，不被某一个特定的情况所误导。

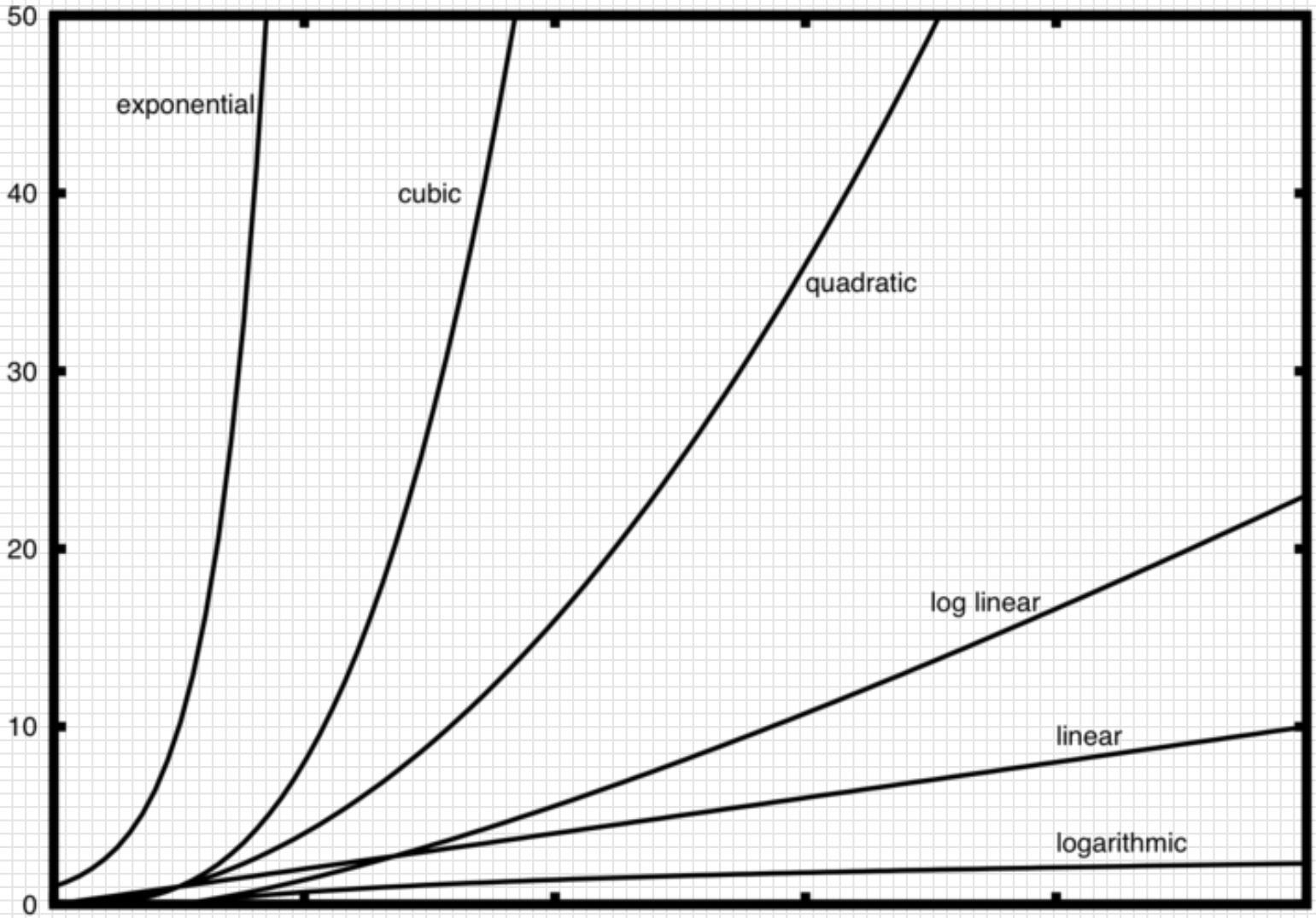
大 O 符号

当你学习算法时，一些常见的数量级函数将会反复出现。为了确定这些函数中哪个是最主要的部分，我们需要看到当 n 变大的时候它们如何相互比较。

$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponetial

大 O 符号

下图表示了各数量级的函数图。注意，当 n 很小时，函数彼此间不能很好的定义。很难判断哪个是主导的。随着 n 的增长，就有一个很明确的关系，很容易看出它们之间的大小关系。



大 O 符号

观察以下程序，虽然它没有做任何事，但是对我们获取实际的代码和性能分析是有益的。

```
a = 5                       # 1
b = 6                       # 1
c = 10                      # 1
for i in range(n):          # 3*n**2
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):          # 2*n
    w = a * k + 45
    v = b * b
d = 33                      # 1
```

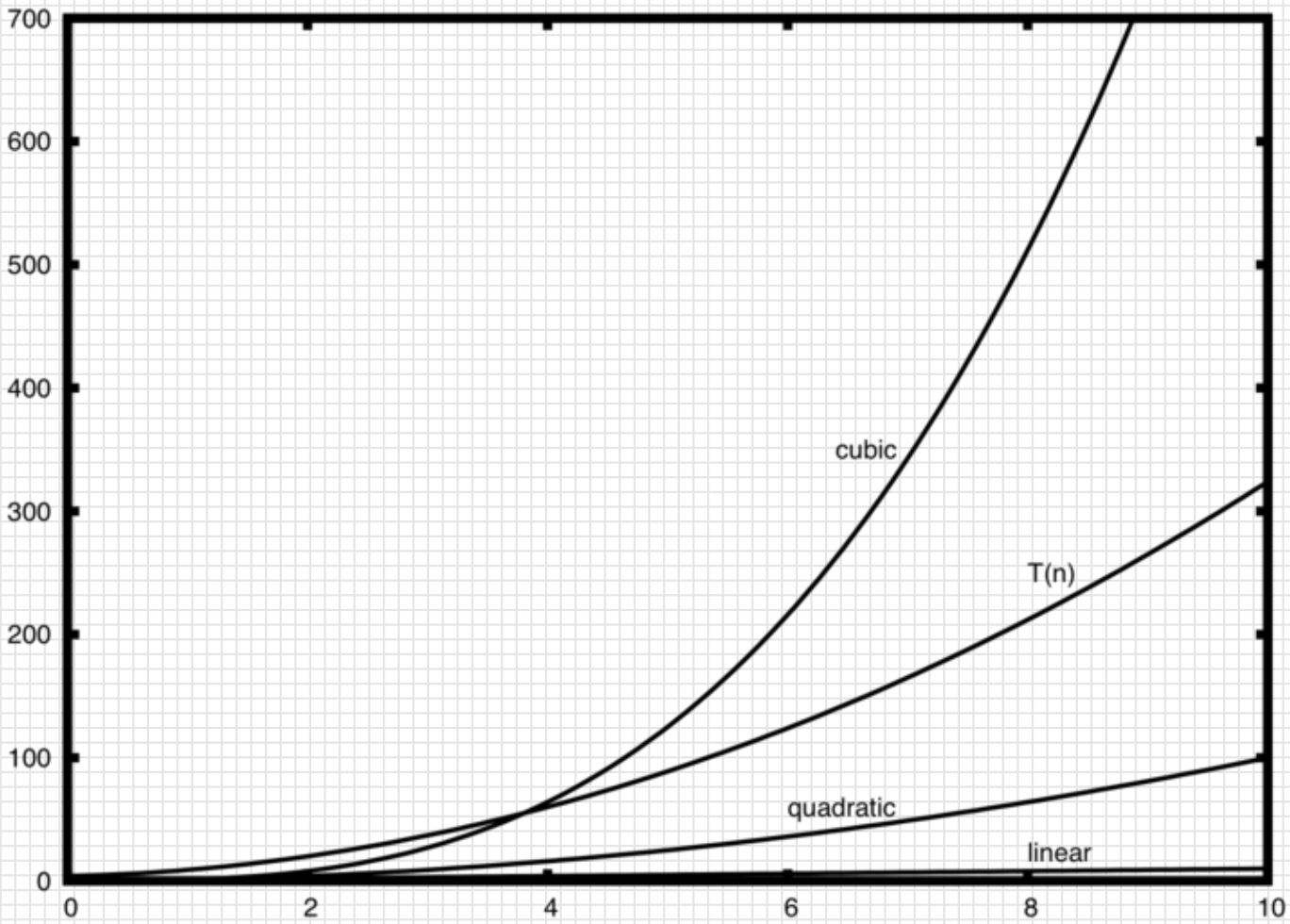
大 O 符号

观察以下程序，虽然它没有做任何事，但是对我们获取实际的代码和性能分析是有益的。

```
a = 5                       # 1
b = 6                       # 1
c = 10                      # 1
for i in range(n):          # 3*n**2
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):          # 2*n
    w = a * k + 45
    v = b * b
d = 33                      # 1
```

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

大 O 符号



一开始，$T(n)$ 大于三次函数，后来随着 n 的增长，三次函数超过了 $T(n)$。$T(n)$ 随着二次函数继续增长。

1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

一个乱序字符串检查的例子

举一个“字符串乱序检查”的例子，以展示不同量级的算法。

定义 (乱序字符串)　乱序字符串是指一个字符串只是另一个字符串的重新排列。

一个乱序字符串检查的例子

举一个“字符串乱序检查”的例子，以展示不同量级的算法。

定义 (乱序字符串)　乱序字符串是指一个字符串只是另一个字符串的重新排列。

例　'heart' 和'earth' 就是乱序字符串。'python' 和'typhon' 也是。

一个乱序字符串检查的例子

举一个“字符串乱序检查”的例子，以展示不同量级的算法。

定义 (乱序字符串)　乱序字符串是指一个字符串只是另一个字符串的重新排列。

例　'heart' 和'earth' 就是乱序字符串。'python' 和'typhon' 也是。

为简单起见，假定所讨论的两个字符串长度相等，且都由 26 个小写字母组成。

编写一个布尔函数，将两个字符串做参数并返回它们是不是乱序。

1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

方案一：检查

第一种方法是检查第一个字符串是不是出现在第二个字符串中。如果可以检验到每一个字符，那这两个字符串一定是乱序。

方案一：检查

第一种方法是检查第一个字符串是不是出现在第二个字符串中。如果可以检验到每一个字符，那这两个字符串一定是乱序。

可以通过用 None 替换字符来了解一个字符是否完成检查。

方案一：检查

第一种方法是检查第一个字符串是不是出现在第二个字符串中。如果可以检验到每一个字符，那这两个字符串一定是乱序。

可以通过用 None 替换字符来了解一个字符是否完成检查。

但是，由于 Python 字符串是不可变的，所以第一步是将第二个字符串转换为列表。

方案一：检查

第一种方法是检查第一个字符串是不是出现在第二个字符串中。如果可以检验到每一个字符，那这两个字符串一定是乱序。

可以通过用 None 替换字符来了解一个字符是否完成检查。

但是，由于 Python 字符串是不可变的，所以第一步是将第二个字符串转换为列表。

检查第一个字符串中的每个字符是否存在于第二个列表中，如果存在，替换成 None。

方案一：检查

```
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'))
```

方案一：检查

注意到 s_1 的每个字符都会在 s_2 中进行 n 个字符的比较，比较次数为 n^2。所以这个算法复杂度为 $O(n^2)$。

1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

方案二：排序和比较

注意到：即使 s_1, s_2 不同，它们都是由完全相同的字符组成的。所以，可以按照字母顺序排列每个字符串，如果排列后的两个字符串相同，则它们就是乱序字符串。

方案二：排序和比较

注意到：即使 s_1, s_2 不同，它们都是由完全相同的字符组成的。所以，可以按照字母顺序排列每个字符串，如果排列后的两个字符串相同，则它们就是乱序字符串。

```
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'))
```

方案二：排序和比较

你可能认为该算法的时间复杂度是 $O(n)$，因为只用了一个简单的循环来比较排序后的字符串。

方案二：排序和比较

你可能认为该算法的时间复杂度是 $O(n)$，因为只用了一个简单的循环来比较排序后的字符串。

但是，调用 Python 的排序函数是有开销的。我们在后面将会讲到，排序的时间复杂度通常是 $O(n^2)$ 或 $O(n \log n)$。

方案二：排序和比较

你可能认为该算法的时间复杂度是 $O(n)$，因为只用了一个简单的循环来比较排序后的字符串。

但是，调用 Python 的排序函数是有开销的。我们在后面将会讲到，排序的时间复杂度通常是 $O(n^2)$ 或 $O(n \log n)$。

所以排序操作比循环花费更多，从而该算法跟排序过程有同样的时间复杂度。

1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

方案三：穷举法

解决这类问题的一种强有力的方法是穷举所有可能性。

方案三：穷举法

解决这类问题的一种强有力的方法是穷举所有可能性。

对于乱序检测，我们可以生成 s_1 的所有乱序字符串列表，然后查看是不是有 s_2。这种方法有一点困难。

方案三：穷举法

解决这类问题的一种强有力的方法是穷举所有可能性。

对于乱序检测，我们可以生成 s_1 的所有乱序字符串列表，然后查看是不是有 s_2。这种方法有一点困难。

当 s_1 生成所有可能的字符串时，第一个位置有 n 种可能，第二个位置有 $n-1$ 种，第三个位置有 $n-2$ 种，等等。总数为 $n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 = n!$。虽然一些字符串可能是重复的，程序也不可能提前知道这样，所以他仍然会生成 $n!$ 个字符串。

方案三：穷举法

解决这类问题的一种强有力的方法是穷举所有可能性。

对于乱序检测，我们可以生成 s_1 的所有乱序字符串列表，然后查看是不是有 s_2。这种方法有一点困难。

当 s_1 生成所有可能的字符串时，第一个位置有 n 种可能，第二个位置有 $n-1$ 种，第三个位置有 $n-2$ 种，等等。总数为 $n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 = n!$。虽然一些字符串可能是重复的，程序也不可能提前知道这样，所以他仍然会生成 $n!$ 个字符串。

事实上，$n!$ 比 n^2 增长要快很多。如果 s_1 有 20 个字符长，则会产生 $20! = 2\ 432\ 902\ 008\ 176\ 640\ 000$ 个字符串。

如果我们每秒处理一种可能字符串，那么需要 77 146 816 596 年才能过完整个列表。所以这不是很好的解决方案。

1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

方案四：计数和比较

最后一种解决方法是：利用两个乱序字符串具有相同数目的 a, b, c 等字符的事实。

1. 首先计算的是每个字母出现的次数。由于有 26 个可能的字符，我们就用一个长度为 26 的列表，每个可能的字符占一个位置。
2. 每次看到一个特定的字符，就增加该位置的计数器。
3. 最后如果两个列表的计数器一样，则字符串为乱序字符串。

方案四：计数和比较

```
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'))
```

方案四：计数和比较

该方案仍有多次迭代，但与方案一不同，它不嵌套。

前两个迭代的时间复杂度为 $O(n)$，第三个迭代用于比较两个计数列表，时间复杂度为 26，共计 $T(n) = 2n + 26$，即 $O(n)$。

这样，我们就找到了一个线性量级的算法。

方案四：计数和比较

最后，我们来讨论下空间复杂度。虽然最后一个方案的时间复杂度是线性的，但它需要额外的存储来保存两个字符计数列表。换句话说，该算法牺牲空间以换取时间。

方案四：计数和比较

最后，我们来讨论下空间复杂度。虽然最后一个方案的时间复杂度是线性的，但它需要额外的存储来保存两个字符计数列表。换句话说，该算法牺牲空间以换取时间。

很多情况下，你需要在空间和时间之间做出权衡。上述程序中开辟的额外空间不大，但是如果有数百万个字符，就需要关注。

方案四：计数和比较

最后，我们来讨论下空间复杂度。虽然最后一个方案的时间复杂度是线性的，但它需要额外的存储来保存两个字符计数列表。换句话说，该算法牺牲空间以换取时间。

很多情况下，你需要在空间和时间之间做出权衡。上述程序中开辟的额外空间不大，但是如果有数百万个字符，就需要关注。

作为一名计算机科学家，当给定一个特定的算法，将由你决定如何使用计算资源。

1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

Python 数据结构的性能

你已经对大 O 记号和不同函数之间的差异有了了解。本节的目标是告诉你 Python 列表和字典操作的大 O 性能。

给出一些基于时间的实验来说明每种数据结构的时间复杂度和使用这些数据结构的好处。

重要的是了解这些数据结构的效率，因为它们是本课程实现其他数据结构所用到的基础模块。

1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

列表

Python 的设计者在实现列表时有很多选择，每一种选择都可能影响列表操作的性能。

对于常见的操作，对其实现进行了优化，以保证它们的处理非常快速。

当然，他们还试图使较不常见的操作快速，当需要做出折衷时，较不常见的操作的性能通常牺牲以支持更常见的操作。

列表

两个常见的操作是索引和分配到索引位置。无论列表有多大，这两个操作都需要相同的时间。当这样的操作和列表的大小无关时，它们是 $O(1)$。

列表

另一种常见的操作是扩展列表，有两种方法可以创建更长的列表：

- append 方法：时间复杂度为 $O(1)$
- 拼接运算符：时间复杂度为 $O(k)$，其中 k 是要拼接的列表的大小。

这对你来说很重要，因为它可以帮助你通过选择合适的工具来提高程序的效率。

列表

观察四种不同的方式，以生成一个 0 的列表。

```
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

为获取每个函数的执行时间，使用 Python 的 timeit 模块。该模块允许 Python 开发人员在一致的环境中运行函数，并使用尽可能相似的操作系统的时序机制来进行跨平台时序测量。

列表

要使用 timeit，你需要创建一个 Timer 对象，其参数是两条 Python 语句:

1. 第一个参数是一个你想要执行时间的 Python 语句;
2. 第二个参数是一个将运行一次以设置测试的语句。

然后 timeit 模块将计算执行语句所需的时间。默认情况下，timeit 将尝试运行语句一百万次。当它完成时，它返回时间作为表示总秒数的浮点值。

由于它执行语句一百万次，可以读取结果作为执行测试一次的微秒数。你还可以传递 timeit 一个参数名字为 number，允许你指定执行测试语句的次数。以下显示了运行我们的每个测试功能 1000 次需要多长时间。

列表

```
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")
```

列表

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

列表

在上面的例子中，我们对 `test1()`，`test2()` 等的函数调用计时，setup 语句可能看起来很奇怪，所以我们详细说明下。你可能非常熟悉 from ,import 语句，但这通常用在 python 程序文件的开头。在这种情况下，`from __main__ import test1` 从 `__main__` 命名空间导入到 timeit 设置的命名空间中。timeit 这么做是因为它想在一个干净的环境中做测试，而不会因为可能有你创建的任何杂变量，以一种不可预见的方式干扰你函数的性能。

列表

从上面的试验清楚的看出，

1. append 操作比拼接快得多；
2. 列表生成器的速度是 append 的两倍；
3. list 函数是列表生成器的两倍。

列表

操作	大 O 效率
`index[]`	$O(1)$
`index assignment`	$O(1)$
`append`	$O(1)$
`pop()`	$O(1)$
`pop(i)`	$O(n)$
`insert(i,item)`	$O(n)$
`del`	$O(n)$
`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`	$O(nk)$

列表

作为一种演示性能差异的方法，我们用 timeit 来做一个实验。我们的目标是验证从列表从末尾 pop 元素和从开始 pop 元素的性能。同样，我们也想测量不同列表大小对这个时间的影响。我们期望看到的是，从列表末尾处弹出所需时间将保持不变，即使列表不断增长。而从列表开始处弹出元素时间将随列表增长而增加。

```
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)
4.821356058120727 5

x = list(range(2000000))
popend.timeit(number=1000)
0.0003161430358886719
```

这段代码展示了两种 pop 方式的比较。从第一个示例看出，从末尾弹出需要 0.0003 毫秒。从开始弹出要花费 4.82 毫秒。对于一个 200 万的元素列表，相差 16000 倍。

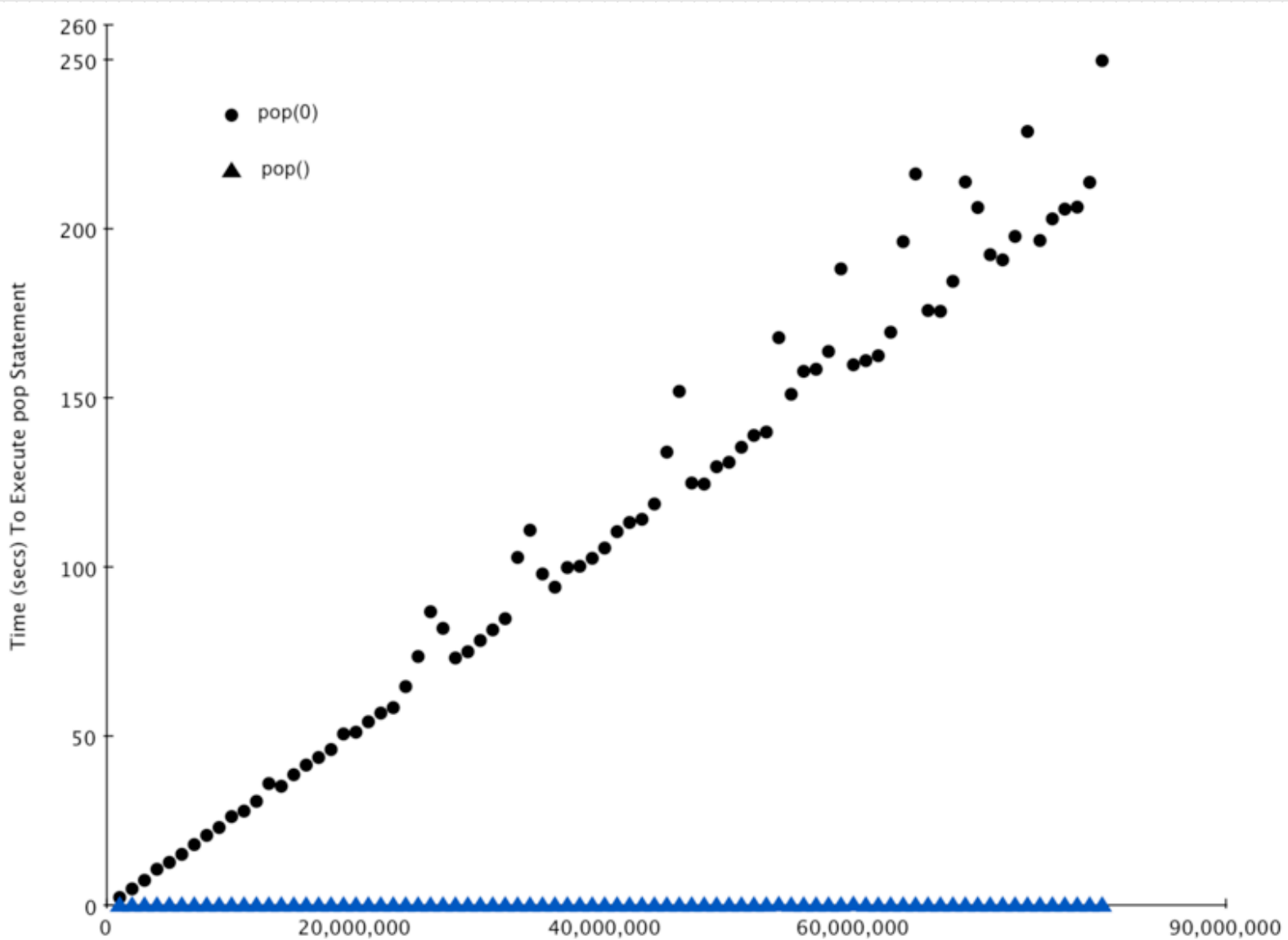
列表

虽然我们第一个测试显示 pop(0) 比 pop() 慢，但它没有证明 pop(0) 是 $O(n)$，pop()是 $O(1)$。要验证它，我们需要看下一系列列表大小的调用效果。

```
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))
```

列表

下图展示了我们实验的结果，你可以看到，随着列表变长，pop(0) 时间也增加，而 pop() 时间保持非常平坦。这正是我们期望看到的 O(n) 和 O(1)



1. 目标
2. 什么是算法分析
3. 大 O 符号
4. 一个乱序字符串检查的例子
   - 方案一：检查
   - 方案二：排序和比较
   - 方案三：穷举法
   - 方案四：计数和比较
5. Python 数据结构的性能
   - 列表
   - 字典

字典

python 中第二个主要的数据结构是字典。

与和列表不同，你可以通过键而不是位置来访问字典中的项目。在本书的后面，你会看到有很多方法来实现字典。

字典的 get 和 set 操作都是 O(1)。

另一个重要的操作是 contains，检查一个键是否在字典中也是 O(1)。

字典

操作	大 O 效率
`copy`	$O(n)$
`get item`	$O(1)$
`set item`	$O(1)$
`delete item`	$O(1)$
`contains(in)`	$O(1)$
`iteration`	$O(n)$

字典

我们现在来比较列表和字典之间的 contains 操作的性能。

在此过程中，我们将确认列表的 contains 操作符是 O(n)，字典的 contains 操作符是 O(1)。

我们将在实验中列出一系列数字。然后随机选择数字，并检查数字是否在列表中。

如果我们的性能表是正确的，列表越大，确定列表中是否包含任意一个数字应该花费的时间越长。

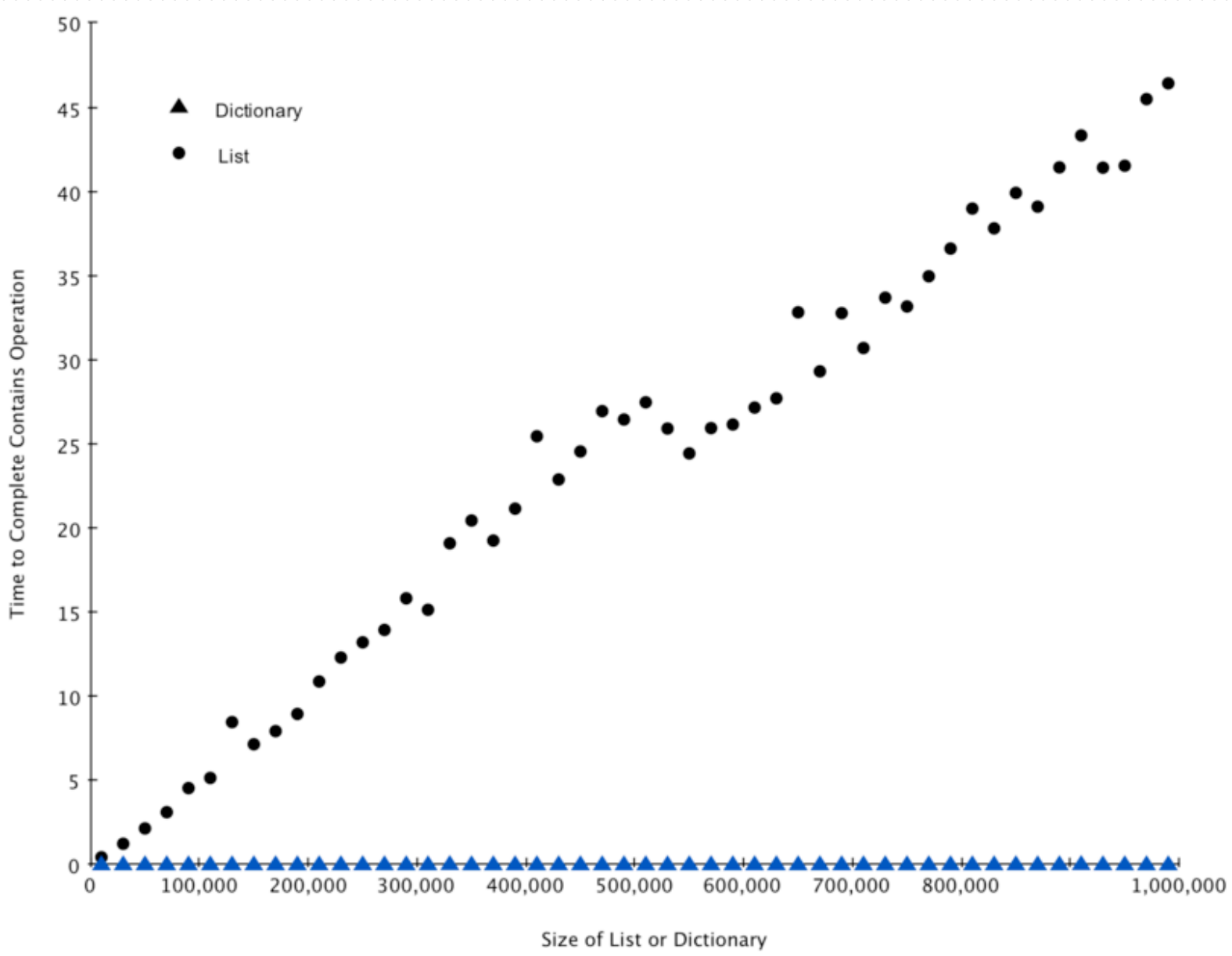
```
import timeit
import random

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))
```

字典


Dictionary
List
Time to Complete Contains Operation
Size of List or Dictionary
0
5
10
15
20
25
30
35
40
45
50
0
100,000
200,000
300,000
400,000
500,000
600,000
700,000
800,000
1,000,000

字典

上图展示了前一段的结果。你可以看到字典一直更快。对于最小的列表大小为 10 000 个元素，字典是列表的 89.4 倍。

对于最大的列表大小为 990 000 个元素。字典是列表的 11,603 倍!

你还可以看到列表上的 contains 运算符所花费的时间与列表的大小成线性增长。

这验证了列表上的 contains 运算符是 O(n) 的断言。

还可以看出，字典中的 contains 运算符的时间是恒定的，即使字典大小不断增长。

事实上，对于字典大小为 10 000 个元素，contains 操作占用 0.004 毫秒，对于字典大小为 990 000 个元素，它也占用 0.004 毫秒。