Python 科学计算基础

第八章 程序运行时间的分析和测量

2025 年 9 月 5 日

目录

算法和时间性能的分析

- 算法
- 时间复杂度
- 时间复杂度的增长阶

程序运行时间的测量

提升运行效率的方法

- Numba
- 使用多个进程运行程序

实验 8：程序运行时间的分析和测量

程序运行时间

一个用高级程序设计语言 (例如 Python) 编写的程序的运行时间取决于多种因素，例如求解问题的算法、问题的规模、输入数据的特点、编译器的代码生成和优化、运行时系统的效率和 CPU 执行指令的速度等。

算法是求解问题的一系列计算步骤，用来将输入数据转换成输出结果。程序是使用某种程序设计语言对一个算法的实现。算法的时间性能是决定程序的运行时间的关键因素。求解一个问题的算法可以有多种，为了缩短程序的运行时间，应选择时间性能最好的算法。

算法

算法的主要性质列举如下。

- 算法的每个步骤应当是精确定义的，不允许出现歧义。
- 算法应在运行有穷步后结束。
- 如果一个算法对所有输入数据都能输出正确的结果并停止，则称它是正确的。

第三章的穷举法求解 [100,200] 的所有质数的设计方案描述了一个算法。

评价算法优劣的一个重要指标是时间性能，即在给定的问题规模下运行算法所消耗的时间。

时间复杂度

时间性能的分析对算法在计算机上的运行过程进行各种简化和抽象，把算法的运行时间定义为问题规模的函数，称为时间复杂度。当问题规模增长时，时间复杂度也会增长。分析的主要结果是时间复杂度的增长阶 (order of growth)，即时间复杂度的增长速度有多快。

当问题规模充分大时，增长阶决定了算法的时间性能。有些算法的运行时间受问题输入数据的影响很大，例如排序算法。此时需要对最坏、平均和最好三种情形进行分析。这里只对最坏情形进行分析，即估计时间复杂度的上界。

时间复杂度的记号

时间复杂度通常使用三种记号描述，目的是只需关注一个时间复杂度函数的表达式中增长阶最高的项并且忽略它的常数系数，该项的增长阶就是时间复杂度的增长阶。

- O 记号：设函数 $f(n)$ 和 $g(n)$ 是定义在非负整数集合上的正函数，如果存在两个正常数 c 和 n_0，使得当 $n \geq n_0$ 时 $f(n) \leq cg(n)$ 成立，则记为 $f(n) = O(g(n))$。当 n 增长到充分大以后，$g(n)$ 是 $f(n)$ 的一个上界。例如：$n^2 + 10n = O(n^3)$；$n^{100} + 100n^{99} = O(1.01^n)$。
- Ω 记号：$f(n) = \Omega(g(n))$ 当且仅当 $g(n) = O(f(n))$。
- Θ 记号：$f(n) = \Theta(g(n))$ 当且仅当 $f(n) = O(g(n))$ 并且 $g(n) = O(f(n))$。例如：$0.01n^3 + 9n^2 = \Theta(n^3)$。

时间复杂度的计算

算法由一些不同类别的的基本运算组成，包括算术运算、关系运算、逻辑运算、数组 (列表) 元素的访问和流程控制等。这些基本运算的运行时间都是常数。算法的时间复杂度等于每种基本运算的运行时间和其对应运行次数的乘积的总和，其中运行次数是问题规模的函数。

为了简化分析，一般只考虑运行次数最多的基本运算，因为当问题规模较大时它们的运行时间是时间复杂度中增长阶最高的项。

时间复杂度的计算

对于单层循环或嵌套的多层循环，运行次数最多的基本运算位于最内层循环。因此，循环的时间复杂度的增长阶由最内层循环的运行次数决定。

对于由递归结构构成的算法，根据递归公式可以得到时间复杂度满足的递归方程 $T(n) = aT(n/b) + f(n)$，其中 $T(n)$ 表示求解规模为非负整数 n 的原问题的时间复杂度。原问题被分解成 a 个规模为 n/b 的与原问题结构类似的子问题，其中 $a \geq 1$ 和 $b > 1$ 是常数，n/b 等于 $\lfloor n/b \rfloor$ 或 $\lceil n/b \rceil$。原问题的解可由这些子问题的解合并而成，合并过程的时间复杂度是一个函数 $f(n)$。

时间复杂度的计算

递归方程可由 Master 定理求解。

- 若 $f(n) = O(n^{\log_b a - \epsilon})$ 对于常数 $\epsilon > 0$ 成立，则 $T(n) = \Theta(n^{\log_b a})$；
- 若 $f(n) = \Theta(n^{\log_b a})$ 成立，则 $T(n) = \Theta(n^{\log_b a} \log_2 n)$；
- 若 $f(n) = \Omega(n^{\log_b a + \epsilon})$ 对于常数 $\epsilon > 0$ 成立，并且当 n 充分大时 $af(n/b) \leq cf(n)$ 对于常数 $c < 1$ 成立，则 $T(n) = \Theta(f(n))$。

时间复杂度的增长阶

大多数算法的时间复杂度属于以下七类，按照增长阶从低到高的次序依次为：$O(1)$，$O(\log n)$，$O(n)$，$O(n\log n)$，$O(n^2)$，$O(n^3)$ 和 $O(a^n)(a>1)$。

一般认为增长阶为 $O(n^k)(k$ 是一个常数$)$ 的算法是可行的。增长阶为 $O(a^n)(a>1)$ 的算法只适用于规模较小的问题。
以下列举属于这七类的一些常用算法。

$O(1)$

时间复杂度属于 $O(1)$ 的算法与问题的规模无关，包括算术运算、逻辑运算、关系运算、读写简单类型的变量 (int、float 和 bool 等)、读写数组中某个索引值的元素等。

$O(\log n)$

程序 8.1 实现了二分查找算法。该算法采用迭代方法，将指定元素 k 和列表 s(已按从小到大的次序排序) 的中间位置的元素进行比较。

- 如果相等，则已找到并返回。
- 如果小于，则只需在左半边的子列表中继续查找。
- 如果大于，只需在右半边的子列表中继续查找。

循环的终止条件是下界大于上界，如果此条件满足则表示未找到。每进行一次迭代，查找范围缩小一半。对于长度为 n 的列表 s，循环次数不超过 $\lceil \log n \rceil$，因此二分查找算法的时间复杂度是 $O(\log n)$。

$O(n)$

程序 8.3 实现了线性查找算法。第 2 行至第 3 行的循环在列表 s 中查找指定元素 k 是否出现，若出现则返回其索引值，否则在第 4 行返回-1 表示未找到。循环在最坏情形下的运行次数是列表 s 的长度 n，因此线性查找算法的时间复杂度是 $O(n)$。

程序 3.11 实现了计算一个列表中的所有元素的最大值和最小值的算法，它的时间复杂度也是 $O(n)$。

$O(n \log n)$

程序 8.4 实现了从小到大排序的归并排序算法。算法的主函数是 merge_sort，它把待排序的列表等分成左右两个子列表，分别对它们递归调用 merge_sort 进行排序，然后调用辅助函数 merge_ordered_lists 把两个已经排好序的子列表归并在一起成为一个排好序的列表。

归并排序的运行时间包括三部分，即递归调用左子列表、递归调用右子列表和归并排好序的两个子列表。对于长度为 n 的列表，时间复杂度 $T(n)$ 满足方程 $T(n) = 2T(n/2) + \Theta(n)$，根据 Master 定理可知归并排序算法的时间复杂度是 $O(n \log n)$。

$O(n^2)$

程序 8.6 实现了从小到大排序的插入排序算法。插入排序的基本思想是将待排序列表中的每个元素依次插入到合适的位置。

对于长度为 n 的列表 s，内层循环在最坏情况下 (待排序列表是从大到小的顺序) 的运行次数为 $1+2+\ldots+(n-1)=n(n-1)/2$。因此插入排序算法的时间复杂度是 $O(n^2)$。

$O(n^3)$

程序 3.17 实现了穷举法求解 3-sum 问题。三重循环的最内层循环的运行次数是 $\sum_{i=0}^{n-3}\sum_{j=i+1}^{n-2}(n-j-1)=n(n-1)(n-2)/6$，因此该算法的时间复杂度是 $O(n^3)$。

$O(a^n)(a > 1)$

程序 3.18 实现了穷举法求解 subset-sum 问题。程序包含两个内循环：第 6 行至第 7 行的 for 循环和第 8 行的 sum(subset)。它们的运行次数都不超过 $n(2^n - 1)$，因此该算法的时间复杂度是 $O(n2^n) = O(2.0001^n)$。

测量一个程序的运行时间

timeit 模块的 timeit 函数测量一个程序重复运行多次所需时间。

程序 8.8 定义的两个字符串 s1 和 s2 表示两个程序。程序 s1 向一个空列表中添加 10 万个元素。程序 s2 生成一个指定长度的由随机数构成的列表，然后对其排序。第 18 行测量了程序 s1 运行 10 次的平均运行时间。第 20 行至第 22 行的循环使用不同的长度值运行程序 s2，测量其运行 10 次的平均运行时间。

测量每一行语句的运行时间

性能分析工具 line_profiler 可测量程序中加了@profile 装饰器的函数中每一行语句的运行时间。在 PowerShell Prompt 窗口中运行命令"conda install line_profiler" 或 "pip install line_profiler" 安装 line_profiler。

程序 8.9 计算 Julia 集并绘图显示。对于复平面上的每个点 z，Julia 集由迭代过程 $z_0 = z, z_{n+1} = z_n^2 + c$ 定义。迭代的终止条件是 $|z| \geq 2$ 或者 $n \geq N$，其中 N 是预先设定的最大迭代次数。将以原点为中心的正方形区域内定义的等距网格上的每个点的迭代次数 n 映射到某一灰度或颜色，即可生成一个灰度或彩色图片。

测量每一行语句的运行时间

设 Miniconda(或 Anaconda) 的安装路径是"C:\Programs\MiniConda"，在操作系统的命令行窗口进入文件"julia_set.py" 所在目录，然后运行命令 "C:\Programs\MiniConda\Scripts\kernprof -l -v julia_set.py" 可显示程序中 calc_z_python 和 calc_Julia 这两个函数的每行语句的运行时间。

calc_Julia 函数的第 30 行的函数调用消耗了整个函数大部分的运行时间，而 calc_z_python 函数中的内循环消耗了整个函数大部分的运行时间，是整个程序的性能瓶颈。

Python 语言程序的运行效率

C 语言是一种静态类型和编译执行的语言，其语法要求程序中的所有变量必须具有类型声明。编译器在编译程序时进行类型的分析和检查，并可以依据类型信息进行代码优化。

Python 是一种动态类型和解释执行的语言，其语法要求程序中的变量不能有类型声明，类型的分析和检查由 Python 解释器在运行程序时完成。这种动态特性虽然降低了编写程序的工作量，但也同时降低了程序的运行效率。

这里介绍两种提升 Python 语言程序的运行效率的方法：Numba 和使用多个进程运行程序。

Numba

Numba 是一种 Python 及时 (just-in-time) 编译器，主要针对使用 NumPy 数组或循环结构的 Python 函数。

安装 Numba 的命令是"conda install numba"。其使用方法是在函数的定义前面加上装饰器 “@numba.jit(nopython=True)”。添加了装饰器的函数在首次被调用时，Numba 为其生成适合本机 CPU 的经过了优化的机器代码。在这之后，调用该函数时只运行机器代码而尽量避免解释执行，因此达到了编译执行的效果。装饰器中的参数设置 nopython=True 表示完全避免解释执行。有时这一要求无法满足，可以省略该设置，此时 Numba 将部分无法编译的部分代码交由 Python 解释器执行。

Numba

程序 8.10 的第 3 行生成了一个由随机数组成的数组。函数 my_sum 使用 for 循环计算该数组中所有元素的和。NumPy 库的 sum 函数实现的功能与 my_sum 相同。第 11 行使用 np.allclose 函数判断这两个函数的计算结果在允许的误差范围内是否相等。若不相等，np.allclose 函数返回 False，assert 语句会报错。

函数 my_cumsum 使用 for 循环计算该数组的累计和。NumPy 库的 cumsum 函数实现的功能与 my_cumsum 相同。第 20 行使用 np.allclose 函数判断这两个函数的计算结果在允许的误差范围内是否相等。

Numba

在 IPython 窗口中运行的“%timeit”命令可测量单行语句在反复运行时的平均运行时间。

程序 8.11 使用该命令测量了以上 4 个函数的平均运行时间，结果表明 NumPy 库的函数的运行效率比 Python 循环高 100 倍以上。
程序 8.12 为 my_sum 和 my_cumsum 加上装饰器。程序 8.13 再次测量了 my_sum 和 my_cumsum 的平均运行时间，结果表明使用 Numba 编译的 Python 函数的运行效率提升了 200 倍以上，接近甚至超过了 NumPy 库的函数。

使用多个进程运行程序

一个程序要在计算机上运行，需要由操作系统为其分配多种资源，如内存空间、CPU 时间等。 进程是正在运行的程序的一个实例。

当前的 CPU 通常包含多个核心，可以同时运行多个进程。如果一个计算任务可分解为一些相互独立的子任务，则可为每个子任务创建一个进程，使这些子任务在多核 CPU 上同时运行，最后综合所有子任务的运行结果得到原任务的运行结果。与只使用单个进程的运行方式相比，使用多个进程的运行方式可以充分利用多核 CPU 的计算能力从而提高运行效率。

使用多个进程运行程序

包 (package) 是多个功能相关且位于同一个根目录下不同子目录的模块的综合体。导入包中的一个模块时需提供该模块的完整路径名，路径名中用点表示路径分隔符。

标准库的 multiprocessing 包提供了创建和运行进程的功能。标准库的 concurrent.futures 模块提供了 ProcessPoolExecutor 类，该类可创建一个由指定数量的进程构成的进程池，其 submit 方法用于同时分配一个进程给一个子任务运行并返回一个 Future 类的对象。Future 类提供了异步执行任务的能力。Future 类的 result 方法返回子任务的运行结果。

使用多个进程运行程序

以计算圆周率 $\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots$ 为例，程序 8.14 比较了两种运行方式在 Intel(R) Core(TM) i7-10700 CPU 上的运行时间。公式的右边有无穷多项，只计算前 N 项的和。函数 task 对于序列 [i+1, N, n] 中的每个数计算其平方的倒数并求和，n=1 时的序列就是公式中的序列 [1, N]，n>1 时的序列是序列 [1, N] 的一个子序列。序列 [1, N] 等于各子序列的并集，因此原任务可分解为多个子任务。

测量结果显示：使用单个进程运行需要 8.95 秒，而使用多个进程运行需要 3.00 秒。

实验 8：程序运行时间的分析和测量

本实验的目的是掌握程序运行时间的分析方法和测量方法。

在 Blackboard 系统提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序和运行结果。

1. 3-sum 问题

1. 实现一个时间复杂度为 $O(n^2 \log n)$ 的算法，解决 3.5 节的 3-sum 问题。

提示：假定存在一个子集 $\{a, b, c\}$ 满足 $x = a + b + c$，则 $x - a = b + c$。原问题可以转换为另一个问题：对于 S 中的任意两个元素 b 和 c，在 $S - \{b, c\}$ 中查找一个等于 $x - b - c$ 的元素。为了提高查找的效率，可以先将所有元素按照从小到大的顺序排序，然后使用二分查找。

2. 测量程序的运行时间

生成长度为 100,200,...,900,1000 的由随机数构成的列表，使用"%timeit" 命令分别测量插入排序 (程序 8.6)、归并排序 (程序 8.4) 和快速排序 (程序 4.14) 这三种排序算法的运行时间。在测量之前需要删除程序中的 print 语句。

3. 测量语句的运行时间

测量归并排序 (程序 8.4) 的两个函数中的每条语句的运行时间。在测量之前需要删除程序中的 print 语句。