

数据结构与算法

数据 是一个抽象的概念，将其进行分类后得到程序设计语言中的基本类型。如: int, float, char 等。数据元素之间不是独立的，存在特定的关系，这些关系便是结构。数据结构指数据对象中数据元素之间的关系。

Python 给我们提供了很多现成的数据结构类型，这些系统自己定义好的，不需要我们自己去定义的数据结构叫做Python 的内置数据结构，比如列表、元组、字典。而有些数据组织方式，Python 系统里面没有直接定义，需要我们自己去定义实现这些数据的组织方式，这些数据组织方式称之为 Python 的扩展数据结构，比如栈，队列等。

算法与数据结构的区别

数据结构只是静态的描述了数据元素之间的关系。

高效的程序需要在数据结构的基础上设计和选择算法。

程序 = 数据结构 + 算法

软件 = 程序 + 软件工程

软件公司 = 软件 + 商业模式

总结:算法是为了解决实际问题而设计的，数据结构是算法需要处理的问题载体

抽象数据类型(Abstract Data Type)

抽象数据类型(ADT)的涵义是指一个数学模型以及定义在此数学模型上的一组操作。即把数据类型和数据类型上的运算捆在一起，进行封装。引入抽象数据类型的目的是把数据类型的表示和数据类型上运算的实现与这些数据类型和运算在程序中的引用隔开，使它们相互独立。

最常用的数据运算有五种：

插入

删除

修改

查找

排序

算法

算法的概念

算法是计算机处理信息的本质，因为计算机程序本质上是一个算法来告诉计算机确切的步骤来执行一个指定的任务。一般地，当算法在处理信息时，会从输入设备或数据的存储地址读取数据，把结果写入输出设备或某个存储地址供以后再调用。

算法是独立存在的一种解决问题的方法和思想

对于算法而言，实现的语言并不重要，重要的是思想。

算法可以有不同的语言描述实现版本 (如C描述、C++描述、Python描述等)，我们现在是在用Python语言进行描述实现。

算法的五大特性

1.输入: 算法具有0个或多个输入

2.输出: 算法至少有1个或多个输出

3.有穷性:算法在有限的步骤之后会自动结束而不会无限循环，并且每一个步骤可以在可接受的时间内完成

4.确定性:算法中的每一步都有确定的含义，不会出现二义性

5.可行性:算法的每一步都是可行的，也就是说每一步都能够执行有限的次数完成

总结:


数据结构是一门研究算法的学科，自从有了编程语言也就有了数据结构。学好数据结构可以编写出更加漂亮更加有效率的代码。


算法是程序的灵魂，为什么有些网站能够在高并发，和海量吞吐情况下依然坚如磐石

二分法不能作用于集合和字典这种无序的结构，要使用列表这种有序的数据容器

数据结构与算法分析

算法效率衡量

```
 import timeit
import random
arr = [i for i in range(1000000)]
target = random.choice(arr)
print('target:\t',target)
```

```
 target: 675653
```



```
def traverse(array, target):
    """随机查找"""
    for number in array:
        if target == number:
            return number
    return False

def binary_search_normal(array, data) :
    """二分查找法 - 非递归实现"""
    start, end = 0, len(array) - 1
    while start <= end:
        mid_index = (start + end) // 2
        if array[mid_index] == data:
            # 如果查询到就返回
            return array.index(data)
        if data > array[mid_index]:
            start = mid_index + 1
        else:
            end = mid_index - 1
    return False
```



```
setup1 = """
from __main__ import traverse;
import random;
array = [i for i in range(1000000)];
target = random.choice(array)
"""

setup2="""
from __main__ import binary_search_normal;
import random;
array = [i for i in range(1000000)];
target = random.choice(array)
"""

timer1 = timeit.Timer(stmt="traverse(array, target)", setup=setup1)
timer2 = timeit.Timer(stmt="binary_search_normal(array, target)", setup=setup2)

print("traverse: %f" % timer1.timeit(number=100))
print("binary: %f" % timer2.timeit(number=100))
```

```
traverse: 0.986643
binary: 0.433293
```

执行时间反应算法效率

对于同一问题，我们给出了两种解法，在两种算法的实现中，我们对程序执行的时间进行了测算，发现两段程序执行的时间相差悬殊(1.750101秒相比于0.247522秒)，由此我们可以得出结论:实现算法程序的执行时间可以反应出算法的效率，即算法的优劣。

单靠时间值绝对可信吗？

假设我们将第二次尝试的算法程序运行在一台配置古老性能低下的计算机中，情况会如何？很可能运行的时间并不会比在我们的电脑中运行算法一的 1.750101 秒快多少。

单纯依靠运行的时间来比较算法的优劣并不一定是客观准确的！

程序的运行离不开计算机环境 (包括硬件和操作系统)，这些客观原因会影响程序运行的速度并反应在程序的执行时间上。那么如何才能客观的评判一个算法的优劣呢？

时间复杂度与“大O记法”

我们假定计算机执行算法每一个基本操作的时间是固定的一个时间单位，那么有多少个基本操作就代表会花费多少时间单位。显然对于不同的机器环境而言，确切的单位时间是不同的，但是对于算法进行多少个基本操作(即花费多少时间单位)在规模数量级上却是相同的，由此可以忽略机器环境的影响而客观的反应算法的时间效率。

对于算法的时间效率，我们可以用“大O记法”来表示。

大O记法：对于单调的整数函数f，如果存在一个整数函数g和实常数C>0，使得对于充分大的n总有f(n) <=c*g(n)，就说函数g是f的一个渐近函数 (忽略常数)，记为f(n)=O(g(n))。也就是说，在趋向无穷的极限意义下，函数f的增长速度受到函数g的约束，亦即函数f与函数g的特征相似。

时间复杂度: 假设存在函数g，使得算法A处理规模为n的问题示例所用时间为T(n)=O(g(n)，则称O(g(n))为算法A的渐近时间复杂度，简称时间复杂度，记为T(n)

如何理解“大O记法”

对于算法进行特别具体的细致分析虽然很好，但在实践中的实际价值有限。对于算法的时间性质和空间性质，最里要的是其数量级和趋势，这些是分析算法效率的主要部分。而计量算法基本操作数量的规模函数中那些常量因子可以忽略不计。例如，可以认为3n^2和100n^2属于同一个量级，如果两个算法处理同样规模实例的代价分别为这两个函数，就认为它们的效率“差不多”，都为n^2级。

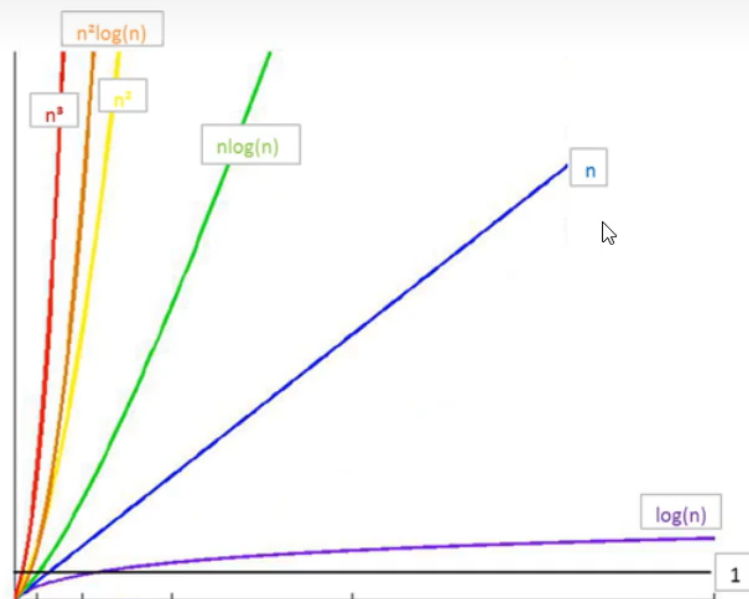
常见的时间复杂度

执行次数函数举例	阶	非正式术语
1 2	O(1)	常数阶
2n + 3	O(n)	线性阶
3n ² + 2n + 1	O(n ²)	平方阶
5log ₂ n + 20	O(logn)	对数阶
2n + 3nlog ₂ n + 19	O(nlogn)	nlogn阶
6n ³ + 2n ² + 3n + 4	O(n ³)	立方阶
2 ⁿ	O(2n)	指数阶

注意，经常将 log₂n (以2为底的对数) 简写成 logn

(常用 线性阶 平方阶 对数阶)

常见时间复杂度之间的关系



所消耗的时间从小到大

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

时间复杂度分为查询和插入的时间

最坏时间复杂度

分析算法时，存在几种可能的考虑:

- 算法完成工作最少需要多少基本操作，即最优时间复杂度。
- 算法完成工作最多需要多少基本操作，即最坏时间复杂度。
- 算法完成工作平均需要多少基本操作，即平均时间复杂度

对于最优时间复杂度，其价值不大，因为它没有提供什么有用信息，其反映的只是最乐观最理想的情况，没有参考价值。

对于最坏时间复杂度，提供了一种保证，表明算法在此种程度的基本操作中一定能完成工作。

对于平均时间复杂度，是对算法的一个全面评价，因此它完整全面的反映了这个算法的性质。但另一方面，这种衡量并没有保证，不是每个计算都能在这个基本操作内完成。而且，对于平均情况的计算，也会因为应用算法的实例分布可能并不均匀而难以计算。

因此，我们主要关注算法的最坏情况，亦即最坏时间复杂度。

时间复杂度的几条基本计算规则

1. 基本操作，即只有常数项，认为其时间复杂度为 $O(1)$
2. 顺序结构，时间复杂度按 **加法** 进行计算
3. 循环结构，时间复杂度按 **乘法** 进行计算
4. 分支结构，时间复杂度 **取最大值**
5. 判断一个算法的效率时，往往只需要关注操作数量的最高次项，其它次要项和常数项可以忽略
6. 在没有特殊说明时，我们所分析的算法的时间复杂度都是指 **最坏时间复杂度**

空间复杂度

类似于时间复杂度的讨论，一个算法的 **空间复杂度 $S(n)$** 定义为该算法所耗费的存储空间，它也是问题规模 n 的函数。

渐近空间复杂度也常常简称为 **空间复杂度**。

空间复杂度(Space Complexity)是对一个算法在运行过程中**临时占用存储空间大小的量度**。

算法的时间复杂度和空间复杂度合称为算法的复杂度。

内置类型性能分析

timeit 模块

timeit模块可以用来测试一小段Python代码的执行速度

```
1 | class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>)
```

Timer 是测量小段代码执行速度的类。

stmt 参数是要测试的代码语句 (statement)；

setup 参数是运行代码时需要的设置；

timer 参数是一个定时器函数，与平台有关。

```
1 | timeit.Timer.timeit(number=1000000)
```

Timer类中测试语句执行速度的对象方法。number参数是测试代码时的测试次数，默认为1000000次。

方法返回执行代码的平均耗时，一个float类型的秒数。