数据结构和算法（Python）

教材
教材

视频课

01什么是算法

算法就是计算机处理信息的本质，因为计算机程序的本质就是一个算法来高速计算机确切的步骤来执行一个指定的任务。
算法是独立存在的解决问题的方法和思 想，实现的语言不重要

算法的5特性：
输入：0或多个
输出：至少一个
有穷性：算法有限的步骤之后不会无限循环，并且在可以接受的时间内完成
确定性：算法中的每一步都有确定的含义
可行性：每一步都是可行的，也就是都能执行有限的次数完成

教材：数据结构与算法：python语言描述
进阶：算法导论（更深，重点学思想

```
import time
start_time = time.time()
for a in range(0,1001):
    for b in range(0,1001):
        for c in range(0,1001):
            if a**2+b**2 == c**2 and a+b+c ==1000:
                print("a, b, c: %d, %d, %d" % (a,b,c))
end_time = time.time()
print("Duration: %d" %(end_time-start_time))
print("finished")
```

```
a, b, c: 0, 500, 500
a, b, c: 200, 375, 425
a, b, c: 375, 200, 425
a, b, c: 500, 0, 500
Duration: 682
finished
```

改进： 因为a,b 确定下来c也就确定了

```python
import time
start_time = time.time()
for a in range(0,1001):
    for b in range(0,1001):
        c = 1000-a-b
        if a**2+b**2 == c**2:
            print("a, b, c: %d, %d, %d" % (a,b,c))
end_time = time.time()
print("Duration: %f" %(end_time-start_time))
print("finished")
```

```
a, b, c: 0, 500, 500
a, b, c: 200, 375, 425
a, b, c: 375, 200, 425
a, b, c: 500, 0, 500
Duration: 0.905913
finished
```

02 算法效率衡量

时间复杂度与大O表示法

单靠运行时间不可信不客观：和电脑新旧，运算能力有关

时间复杂度：描述算法时间上的效率

假定执行算法每一个基本操作的时间是固定的，**执行的基本运算数量**可以忽略机器环境的影响客观的反映算法的时间效率

"大O表示法"

> 时间复杂度：假设存在函数g，似的算法A处理规模为n的问题实力所用的时间为T(n)=O(g(n)),则称O(g(n))为算法A的监禁时间复杂度，简称为时间复杂度，记为T(n)

计量算法基本操作的规模函数中的常量因子可以忽略不计

03 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：没啥意义

算法完成工作最多需要多少基本操作，即**最坏时间复杂度**:一种保证，通常时间复杂度就是指最坏这个

算法完成工作平均需要多少基本操作，即平均时间复杂度：全面评价，但没有保证

##总结：

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 (比如if else)
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

最常见的时间复杂度

执行次数函数举例	阶	非正式术语
12	O(1)	常数阶
2n+3	O(n)	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
5log2n+20	O(logn)	对数阶
2n+3nlog2n+19	O(nlogn)	nlogn阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

注意，经常将log2n（以2为底的对数）简写成logn

必须记住：

$$O(1)<O(logn)<O(n)<O(nlogn)<O(n^2)<O(n^2logn)<O(n^3)<O(2^n)<O(n!)<O(n^n)$$

Python 内置类型性能分析 -- timeit

> 列表生成式
>
> Python内置的一种极其强大的生成列表 list 的表达式。返回结果必须是列表。

> 基本语法：

`[ 变量表达式 for 变量 in 表达式 ]`

```python
#对比四种list列表构造方式的效率

from timeit import Timer

def test1(): #append()
    l = []
    for i in range(10000):
        l.append(i)

def test2(): #+=
```

```
    l = []
    for i in range(10000):
        l += [i]

def test2_1(): #+
    l = []
    for i in range(10000):
        l = l + [i]

def test3(): #列表生成式 [i for i in range]
    l = [i for i in range(10000)]

def test4(): #可迭代对象直接生成列表 list(range)
    l = list(range(10000))

def test5(): #extend()
    l = []
    for i in range(10000):
      l.extend([i])




timer1= Timer("test1()","from __main__ import test1")
print("append():",timer1.timeit(number=1000),"seconds")

timer2= Timer("test2()","from __main__ import test2" )
print("+=:",timer2.timeit(number=1000),"seconds")

timer2_1= Timer("test2_1()","from __main__ import test2_1" )
print("+:",timer2_1.timeit(number=1000),"seconds")

timer3= Timer("test3()","from __main__ import test3" )
print("[i for i in range]:",timer3.timeit(number=1000),"seconds")

timer4= Timer("test4()","from __main__ import test4" )
print("list(range):",timer4.timeit(number=1000),"seconds")

timer5= Timer("test5()","from __main__ import test5" )
print("extend():",timer5.timeit(number=1000),"seconds")
```

```
append(): 0.64611264999985 seconds
+=: 0.6851804809998612 seconds
+: 112.89560661899986 seconds
[i for i in range]: 0.2947488119998525 seconds
list(range): 0.1802340100002766 seconds
extend(): 0.9059880499999053 seconds
```

可以看出：光用+很费时间，可以选择使用+= or extend（）替换

insert 和 append 比较

> list.insert(pos, elmnt)

```
def test1(): #append()
    l = []
    for i in range(10000):
        l.append(i) #尾部添加


def test6(): #extend()
    l = []
    for i in range(10000):
      l.insert(0, i) #头部添加

timer1= Timer("test1()","from __main__ import test1")
print("append():",timer1.timeit(number=1000),"seconds")

timer6= Timer("test6()","from __main__ import test6" )
print("insert():",timer6.timeit(number=1000),"seconds")
```

```
append(): 0.6008801650004898 seconds
insert(): 19.97815154900036 seconds
```

从结果可以看出，append从尾端添加元素效率远远高于insert从顶端添加元素

原因： 由于列表list的数据存储[方式决定的

list内置操作的时间复杂度

操作	平均时间复杂度
list[index]	O(1)
list.append	O(1)
list.insert	O(n)
list.pop(index), default last element	O(1)
list.remove	O(n)

Index:O(1)
append: O(1)
contain(in): O(n)查找

dict内置操作的时间复杂度

Operation	Big-O Efficiency
copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$

Table 2.3: Big-O Efficiency of Python Dictionary Operations

数据结构

算法关注的是问题的解决步骤，没有关注处理的是什么样的数据

> 我们如何用Python中的类型来保存一个班的学生信息？ 如果想要快速的通过学生姓名获取其信息呢?

实际上当我们在思考这个问题的时候，我们已经用到了数据结构。列表和字典都可以存储一个班的学生信息，但是想要在列表中获取一名同学的信息时，就要遍历这个列表，其时间复杂度为O(n)，而使用字典存储时，可将学生姓名作为字典的键，学生信息作为值，进而查询时不需要遍历便可快速获取到学生信息，其时间复杂度为O(1)

概念

数据结构是计算机存储、组织数据的方式。数据结构是指相互之间存在一种或多种特定关系的数据元素的集合。

简而言之：一组数据如何保存

基本数据类型： float、int, char...

所以list， dict...已经是一种高级的数据结构了

算法与数据结构的区别

程序 = 数据结构 + 算法

算法：为解决实际问题设计的

数据结构： 算法处理问题的载体

抽象数据类型(ADT：Abstract Data Type)

定义：一个数学模型以及定义在此数学模型上的一组操作

例子：

先规定好数据如何保存（数据结构），定义这些数据结构支持的操作（具体怎么实现不去管）

```
class students(object):

    def adds
    def pop
    def sort
    def ...
```

```
a = [1, 2]
a *=10
print(a)


```

```
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

最常用的数据运算有五种：

```
插入
删除
修改
查找
排序
```