



# 数据结构与算法（ Python ）-02/第3周

北京大学 陈斌

2021.03.23

# 线下课堂

- › 关于Canvas提交作业
- › 本周内容小结
- › 回答疑问
- › H1及不同实现
- › 介绍一个模块big\_O
- › 【K02】计时验证实验



# 关于Canvas交作业

- › **开始时间**：可以开始提交作业
- › **截止时间**：DDL，超过截止时间，得分-50%
- › **结束时间**：最后期限，超过后无法提交作业
- › **误区，注意！**  
只要是在结束时间之前，都是可以随时多次提交作业；  
但是，作业计分是以**最后一次提交**为准！

# 本周内容小结：W02-算法分析

## › 201 什么是算法分析

算法分析不是程序分析，更非代码分析，如何评判一个算法的性能？

## › 202 大O表示法

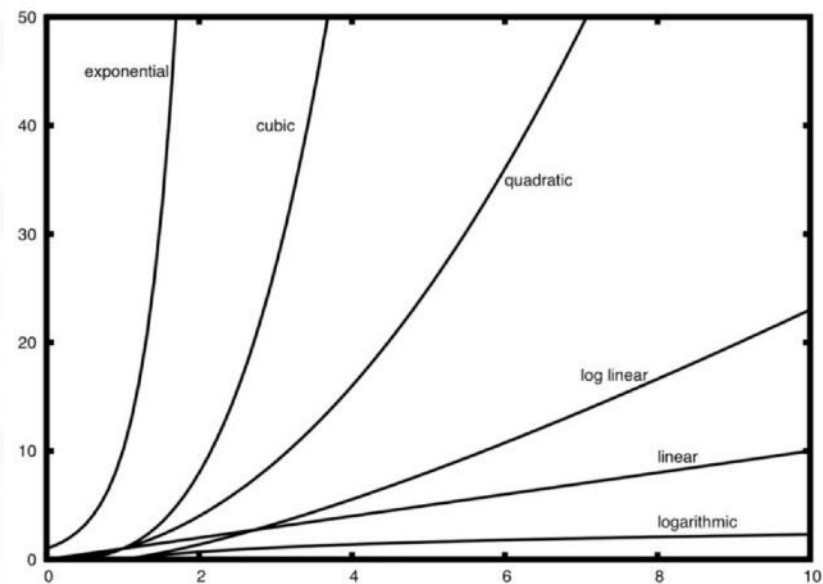
时间复杂度的一种表示，当问题规模线性增长时，所需处理时间的增长趋势

## › 203/4 “变位词” 判断问题

通过“变位词”的四个算法，体验复杂度概念

## › 205/6 Python数据类型的性能

Python内置的数据类型操作性能





# W02-201-什么是算法分析

## 算法分析的概念

- ❖ 比较程序的“好坏”，有更多因素  
代码风格、可读性等等
- ❖ 我们主要感兴趣的是算法本身特性
- ❖ 算法分析主要就是从计算资源消耗的角度来评判和比较算法  
更高效利用计算资源，或者更少占用计算资源的算法，就是好算法  
从这个角度，前述两段程序实际上是基本相同的，它们都采用了一样的算法来解决累计求和问题

## 运行时间检测的分析

- ❖ 但关于运行时间的实际检测，有点问题  
关于编程语言和运行环境
- ❖ 同一个算法，采用不同的编程语言编写，放在不同的机器上运行，得到的运行时间会不一样，**有时候会大不一样**：  
比如把非迭代算法放在老旧机器上跑，甚至可能慢过新机器上的迭代算法
- ❖ 我们需要更好的方法来衡量算法运行时间  
这个指标与**具体**的机器、程序、运行时段都**无关**

# W02-202-大O表示法

## 算法时间度量指标

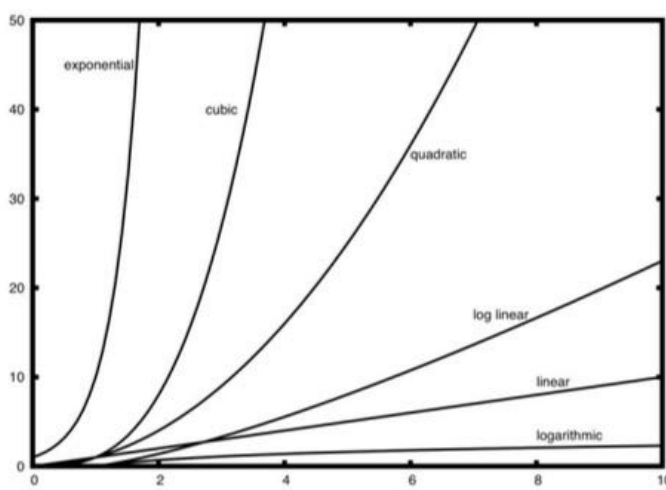
- ❖ 一个算法所实施的操作数量或步骤数可作为独立于具体程序/机器的度量指标
  - 哪种操作跟算法的具体实现无关?
  - 需要一种通用的基本操作来作为运行步骤的计量单位

## 问题规模影响算法执行时间

- ❖ 问题规模：影响算法执行时间的主要因素
- ❖ 在前n个整数累计求和的算法中，需要累计的**整数个数**合适作为**问题规模**的指标
  - 前100,000个整数求和对比前1,000个整数求和，算是同一问题的更大规模
- ❖ 算法分析的目标是要找出**问题规模**会怎么影响一个算法的**执行时间**

## 常见的大O数量级函数

- ❖ 通常当n较小时，难以确定其数量级
- ❖ 当n增长到较大时，容易看出其主要变化量级



$f(n)$	名称
1	常数
$\log(n)$	对数
$n$	线性
$n * \log(n)$	对数线性
$n^2$	平方
$n^3$	立方
$2^n$	指数

# W02-202-大O表示法

## 世界上最早的算法：欧几里德算法

- ❖ 辗转相除法处理大数时非常高效
- ❖ 它需要的步骤不会超过较小数位数的5倍  
加百利·拉梅(Gabriel Lamé)于1844年证明了这个结论  
并开创了计算复杂性理论。



› 请问这个算法的大O复杂度是多少？



# W02-203/4 “变位词” 判断问题

› 对于最好的 $O(n)$ 算法，其实是有额外的空间代价

## 解法4：计数比较-算法分析

❖ 值得注意的是，本算法依赖于两个长度为26的计数器列表，来保存字符计数，这相比前3个算法需要更多的存储空间

如果考虑由大字符集构成的词（如中文具有上万不同字符），还会需要更多存储空间。

❖ 牺牲存储空间来换取运行时间，或者相反，这种在**时间空间之间的取舍**和权衡，在选择问题解法的过程中经常会出现。



# W02-205/6 Python数据类型的性能

## › 用timeit来“验证”大O数量级 使用timeit模块对函数计时

- ❖ 创建一个Timer对象，指定需要**反复运行**的语句和只需要**运行一次**的“安装语句”
- ❖ 然后调用这个对象的timeit方法，其中可以指定反复运行多少次

```
from timeit import Timer
t1= Timer("test1()", "from __main__ import test1")
print "concat %f seconds\n" % t1.timeit(number= 1000)

t2= Timer("test2()", "from __main__ import test2")
print "append %f seconds\n" % t2.timeit(number= 1000)

t3= Timer("test3()", "from __main__ import test3")
print "comprehension %f seconds\n" % t3.timeit(number= 1000)

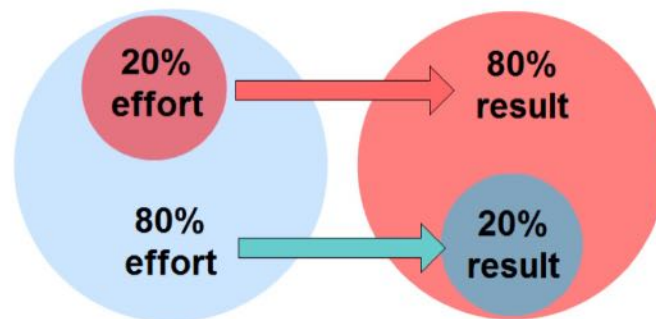
t4= Timer("test4()", "from __main__ import test4")
print "list range %f seconds\n" % t4.timeit(number= 1000)
```

北京大学 陈斌 gischen@pku.edu.cn 2021

## › Python语言的实现中算法权衡

- ❖ 总的方案就是，让**最常用的操作性能最好**，牺牲不太常用的操作

80/20准则：80%的功能其使用率只有20%



# 关于timeit的几个参数：理解命名空间和作用域

```
1 from timeit import Timer
2
3 lst = list(range(1000000))
4
5
6 def test1():
7     global lst
8     lst.pop(0)
9
10
11 t1 = Timer("test1()", "from __main__ import test1")
12 print(f"{t1.timeit(number=1000):.3f} seconds")
```

名字test1在\_\_main\_\_中


需要执行多次的源代码

执行1次的“安装”语句

# <https://wiki.python.org/moin/TimeComplexity>

Operation	Average Case
Copy	$O(n)$
Append[1]	$O(1)$
Pop last	$O(1)$
Pop intermediate[2]	$O(n)$
Insert	$O(n)$
Get Item	$O(1)$
Set Item	$O(1)$
Delete Item	$O(n)$
Iteration	$O(n)$

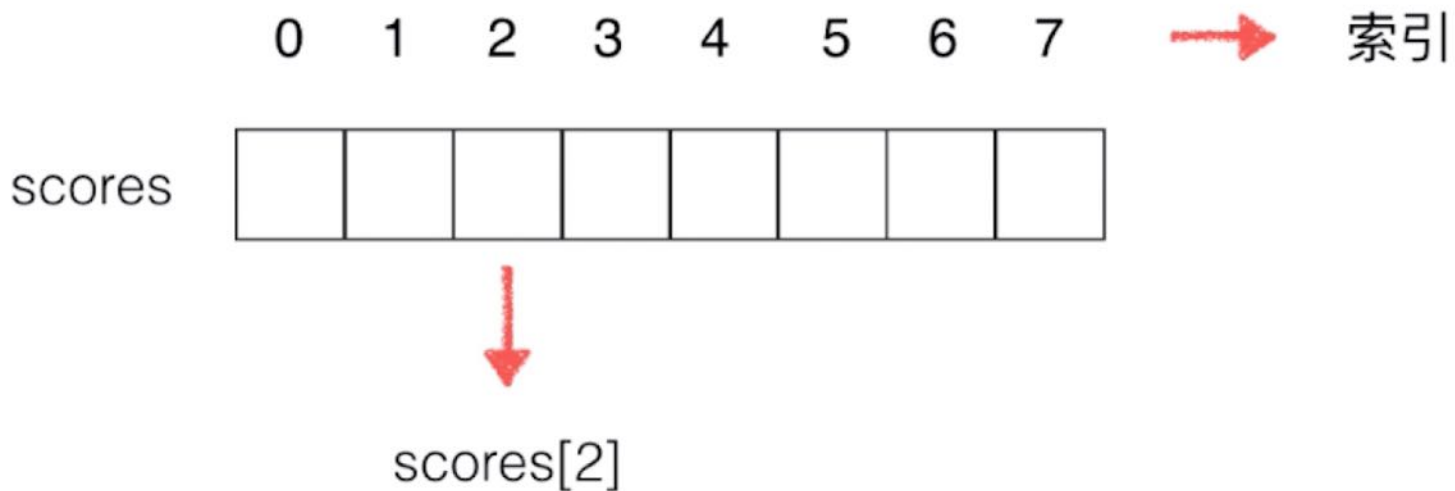
## list类型操作的平均时间复杂度

Get Slice	$O(k)$
Del Slice	$O(n)$
Set Slice	$O(k+n)$
Extend[1]	$O(k)$
 Sort	$O(n \log n)$
Multiply	$O(nk)$
x in s	$O(n)$
min(s), max(s)	$O(n)$
Get Length	$O(1)$



# 回答疑问

- 提问：为什么为使列表按下标赋值和下标取值的时间复杂度为 $O(1)$ ，一定要在删除元素后把后面的都往前移一位？
- 1，如何组织一组数据，按下标取值赋值，能达到 $O(1)$ 的时间复杂度？
- 2，在发生增加、删除时，如何维持这组数据？



# 回答疑问

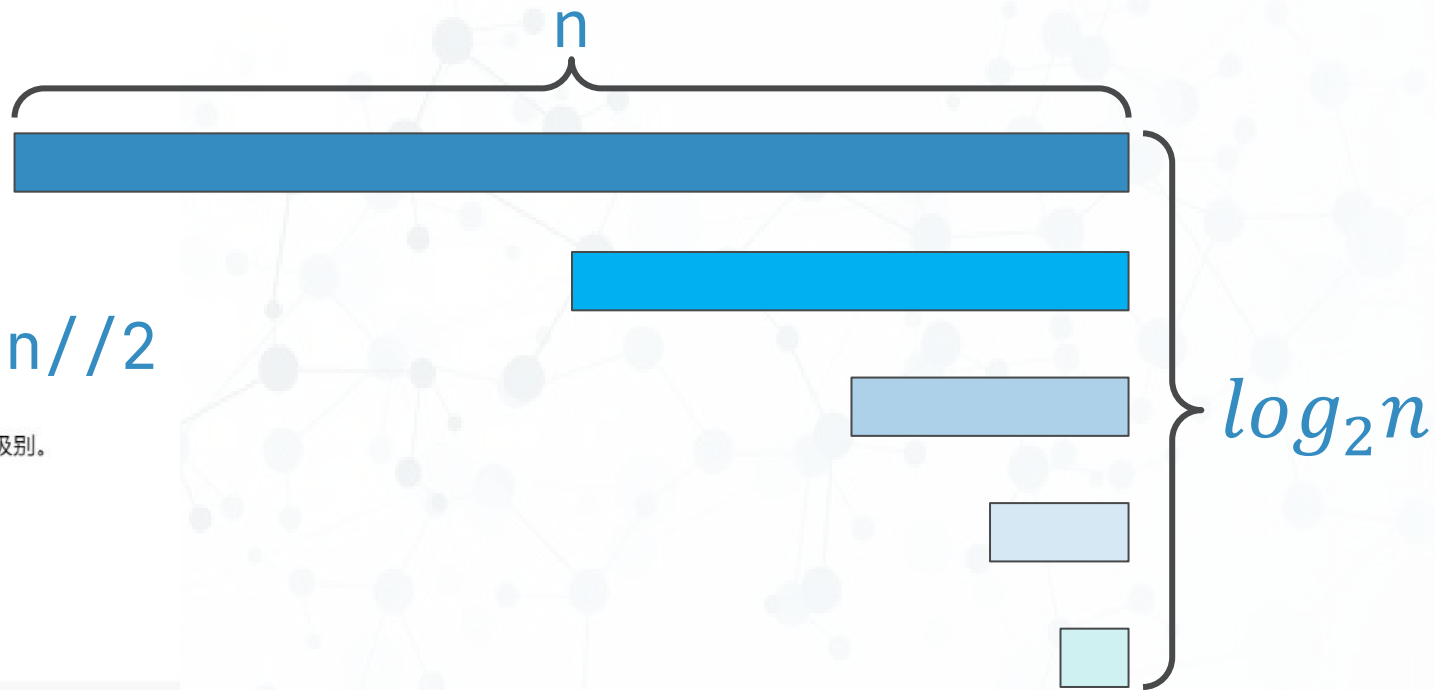
- › index的具体算法？为什么在字符串里index比挨个迭代还慢
- › 一道题目：请问这道题的大O级别是怎么判断出的呢？

5 单选 (2分) 以下是一个快速幂算法：

```
def pow(x, n):  
    if n==0:  
        return 1  
    elif n==1:  
        return x  
    elif n%2==0:  
        return pow(x*x, n//2)  
    else:  
        return pow(x*x, n//2)*x
```

问它对于n的大O级别。

- ☐ A.  $O(n)$
- ☐ B.  $O(1)$
- ☒ C.  $O(\log n)$
- ☐ D.  $O(n \log n)$



# 作业详解：【H1】在Pi中查找生日

请在提供的“pi50.4.bin”文件所包含的5000万位圆周率中查找日期，编写程序实现如下功能：

- 1，输入一个8位数字的日期，打印这个日期所在的位置，如果找不到，则输出“NOT FOUND”；
- 2，输入“2021”，查找从20210101到20211231的365天，每个日期是否存在于5000万位圆周率中；输出3个数值：(存在的日期个数；不存在的日期个数；查找所花的总时间（秒）)

请提交PDF，包括如下内容：

- 1，源代码，注释中说明你的思路；
- 2，运行结果的截图。



# H1中搜索2021年365天是否存在Pi前5000万位中

› 首先，读入数据文件

```
5 f = open("pi50.4.bin", "rb")
6 dbuff = f.read()
```

› 然后，构造5000万位Pi字符串（关于encode/str/bytes）

```
8 s = ("".join("%02x" % d for d in dbuff)).encode()
```

› 直观的算法：把20210101 ~ 20211231这365个字符串逐个判断

```
9 d1 = datetime.date(2021, 1, 1)
10 found = 0
11 for n in range(365):
12     day = ((d1 + datetime.timedelta(days=n)).strftime("%Y%m%d")).encode()
13     if day in s:
14         found += 1
```

## 第二个算法：扫描Pi串，找到前缀2021，判断后4位

› 构造所有日期的字符串**集合**：

```
10 d1 = datetime.date(2021, 1, 1)
11 days = {
12     ((d1 + datetime.timedelta(days=n)).strftime("%m%d")).encode()
13     for n in range(365)
14 }
```

› **扫描前缀**

› **匹配日期**

```
16 y = b"2021"
17 found = 0
18 idx = s.find(y) # 扫描查找前缀2021
19 while idx >= 0:
20     p4 = s[idx + 4 : idx + 8] # 后4位日期
21     if p4 in days: # 判断日期是否在日期集合中
22         found += 1
23         days.remove(p4) # 存在则加1，并从集合中移除
24     idx = s.find(y, idx + 1) # 继续扫描
```

## 第三方模块big\_O

```
pip install big-O
```

- › 估算一个函数的大O数量级
- › `big_o(`
- › `func`, # 需要估计大O数量级的算法函数, 1个输入参数
- › `data_generator`, # 能产生输入参数的函数, 以N为参数
- › `min_n`, `max_n`, `n_measures`, # 最小N, 最大N, 取多少个N
- › `n_repeats`, # 重复执行多少次func, 来计算执行时间
- › `n_timings`) # 重复测量多少次, 保留最好测量结果
- › 返回值: (`best_class`, `fitted`) 最佳拟合的复杂度, 以及其它拟合信息



## 第三方模块big\_O

```
pip install big-O
```

- › 一些通用的data\_generator
- › `big_o.datagen.n_(N)` 就是参数N
- › `big_o.datagen.integers(N, min, max)` 返回N个随机整数list
- › `big_o.datagen.range_n(N)` 返回参数N的range(N)列表list
- › 参考：<https://pypi.org/project/big-O/>

# big\_O用法示例

```
1 from big_o import big_o, datagen
2
3 # 估算排序函数sorted的大O数量级
4 best, others = big_o(
5     sorted,
6     lambda n: datagen.integers(n, 10000, 50000),
7     min_n=1000,
8     max_n=100000,
9     n_measures=100,
10 )
11 print(best)
```

## 【K02】验证list添加数据项的大O数量级

- › 请编写程序，验证向一个列表添加数据项的两种方法，复杂度对比：  
`list.insert(0, item)`：添加数据项成为列表的第一个元素  
`list.append(item)`：添加数据项成为列表的最后一个元素
- › 数据列表的规模为1万起，步长1万，10个规模点，每个规模点操作1000次；
  1. 用`timeit`模块获得操作时间，并进行对比。
  2. 用`big_o`模块来拟合两种方法的复杂度。
- › 提交代码和输出结果的截屏。



# 下课！

