

数据结构与算法B

01 – 概论



北京大学

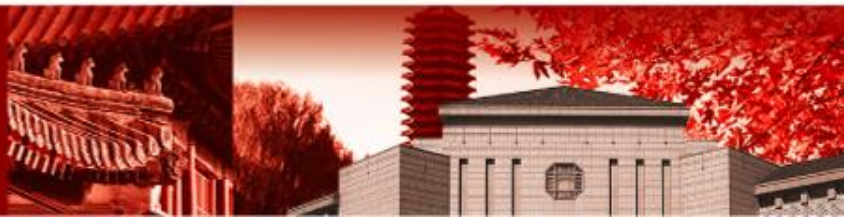


目录

- 计算与问题求解
- 图灵机
- 可以通过“计算”解决的问题
- 不可计算问题
- 计算复杂性



北京大学

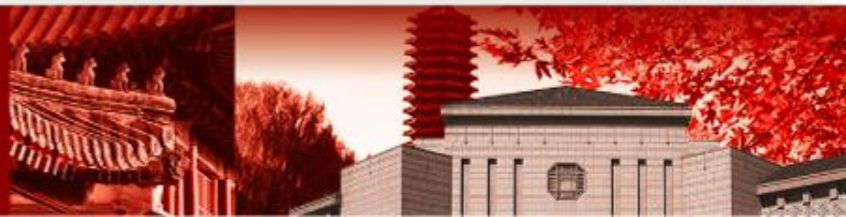


计算与问题求解

- 人们在生活、生产、学习、探索、创造过程中会遇到各种未知的事物
 - 云是什么？这种草（虫子）可以吃么？什么是无理数？什么是万物的起源？
 - 为什么会下雨？为什么食物放久了会发霉？为什么 $\sqrt{2}$ 是无理数？生命的意义是什么？
 - 怎么让粮食长得更多？怎么将楼房建到101层？
- 问题解决之道：从未知到已知
 - 感觉、经验
 - 占卜、求神
 - 逻辑、数学、实验
 - 工程、计算
 - 模型、模拟、仿真

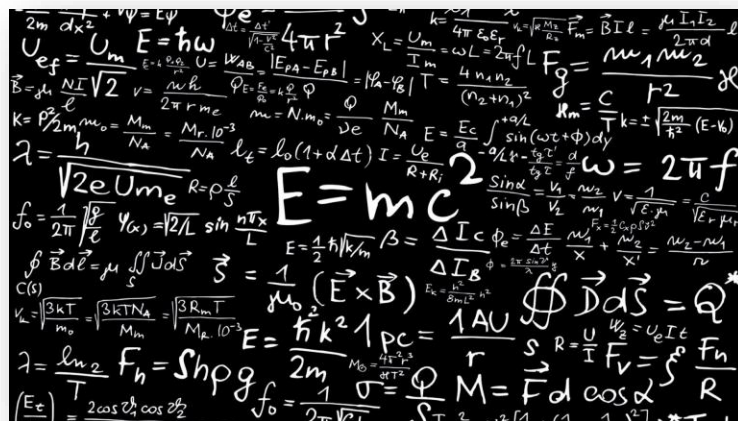


北京大学

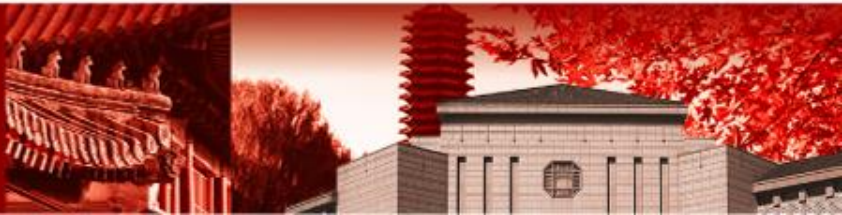


计算与问题求解

- 有些问题已经解决，很多问题尚未解决，有些问题似乎无法解决
 - **尚未解决和无法解决问题的共性**：表述含混、标准不一、涉及主观、结果不确定
- 数学：解决问题的终极工具
 - 在长期的发展过程中，人们把已经解决的问题逐渐表述为数学命题与模型；
 - 尚未解决的问题，人们试图通过数学建模，采用数学工具来解决；
 - 无法解决的问题，人们试图转换表述、明晰问题来部分解决。



北京大学

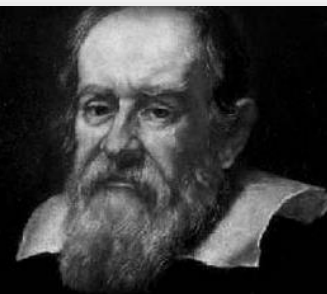


计算与问题求解

- 为什么是数学？
 - 数学具有清晰明确的符号表述体系；
 - 严密确定的推理系统；
 - 但正如科学不是万能的，数学也不是万能的
- 有些问题天然无法明确表述（主观、价值观、意识形态、哲学问题等）
 - 如何解决电车难题？如何选择少数优秀的人做项目？
 - 什么是艺术？如何衡量艺术品的价值？
- 有些可明确表述的问题仍然无法解决（留后待述）

**Mathematics is the alphabet in
which God has written the universe**

Galileo, Italian astronomer, mathematician and philosopher (1564 - 1642)

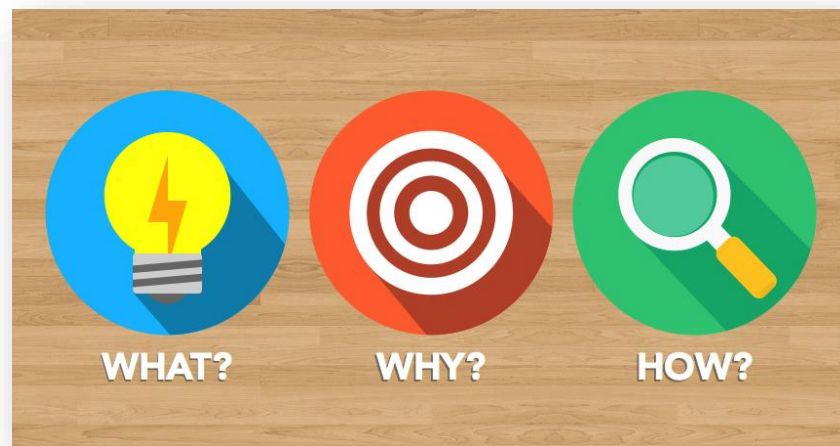


北京大学



计算与问题求解：问题的分类

- **What:** 是什么？
—面向判断与分类的问题；
- **Why:** 为什么？
—面向求因与证明的问题；
- **How:** 怎么做？
—面向过程与构建的问题。



北京大学



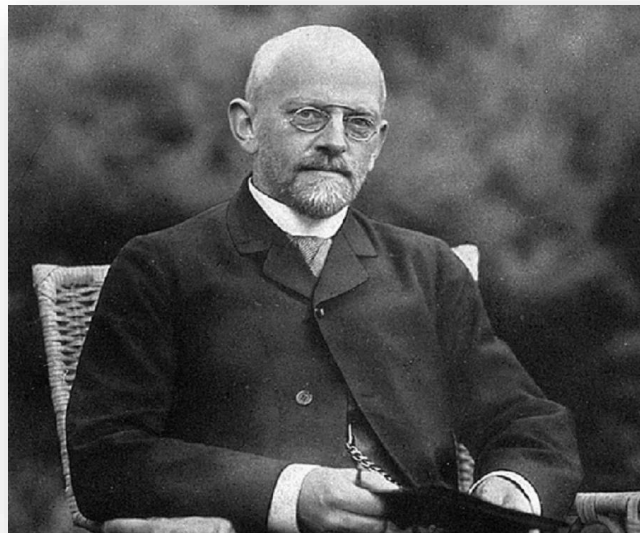
计算与问题求解

- 问题解决的“计算”之道

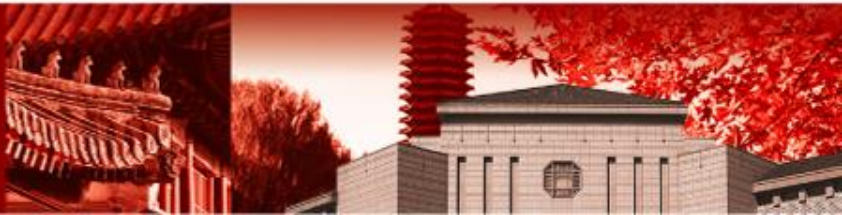
- 20世纪20年代，为了解决数学本身的可检验性问题，大数学家希尔伯特提出“能否找到一种基于有穷观点的能行方法 (effectively computable method)，来判定任何一个数学命题的真假”

- 抽象的“计算”概念提出

- 由有限数量的明确有限指令构成；
 - 指令执行在有限步骤后终止；
 - 指令每次执行都总能得到正确解；
 - 原则上可以由人单独采用纸笔完成，而不依靠其它辅助；
 - 每条指令可以机械地被精确执行，而不需要智慧和灵感。



北京大学

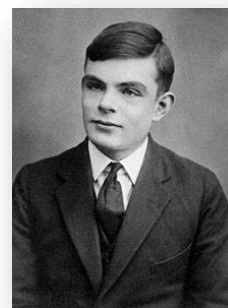
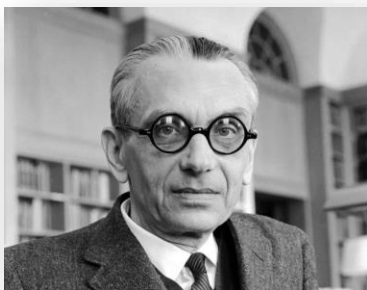


计算与问题求解

- 关于“计算”的数学模型

- 20世纪30年代，几位逻辑学家几乎同时各自独立提出了几个关于“计算”的数学模型

- 奥地利逻辑学家、数学家哥德尔(K.F. Godel, 1906-1978)和美国逻辑学家、数学家克莱尼(S.C. Kleene, 1909-1994)的递归函数模型
 - 美国逻辑学家、数学家丘奇(A. Church, 1903-1995)的Lambda演算模型
 - 波兰裔美国逻辑学家、数学家波斯特(E.L. Post, 1897-1954)的Post机模型
 - 英国逻辑学家、数学家图灵(A.M. Turing, 1912-1954)的图灵机模型



北京大学



计算与问题求解

- 后续研究证明，这几个“**基于有穷观点的能行方法**”的计算模型，全都是等价的
 - 在某个模型下“可计算”的问题，在另外的模型下也是“可计算”的
- 虽然希尔伯特的计划最终被证明无法实现
 - 即不存在“能行方法”来判定任何一个数学命题的真假
 - 总有数学命题，其真假是无法证明的
- 但“**能行可计算**”的概念，成为了计算理论的基础
 - 其中的一些数学模型（如图灵机）也成为现代计算机的理论基础

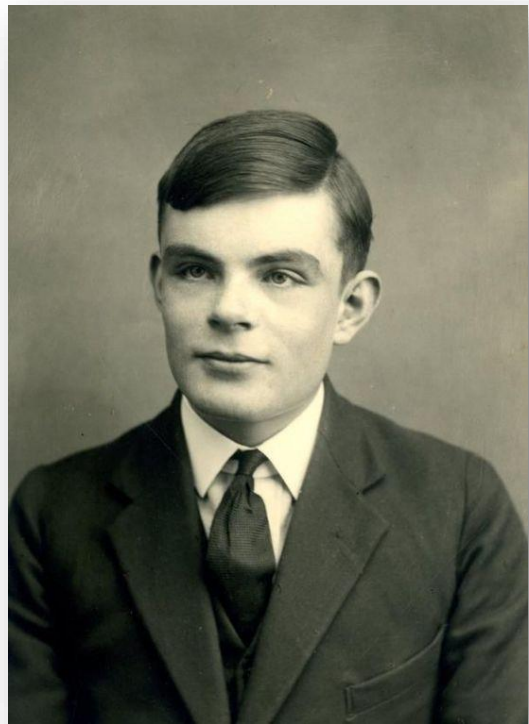


北京大学

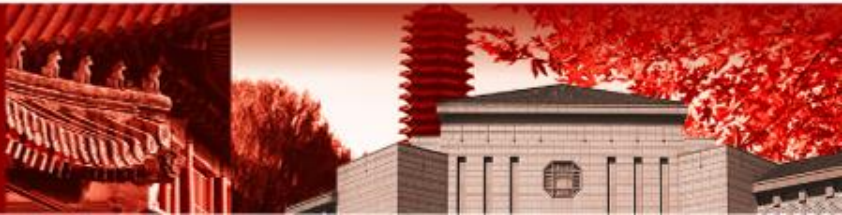


图灵机Turing Machine

- 1936年，Alan Turing提出的一种抽象计算模型
 - 基本思想是，用机器模拟人们用纸笔进行数学运算的过程，但比数值计算更为简单
- 基本概念
 - 在纸上写上或擦除某个符号；
 - 把注意力从纸的一个位置转向另一个位置
 - 在每个阶段，人要决定下一步的动作，依赖于：
 - (a) 此人当前所关注的纸上某个位置的符号和
 - (b) 此人当前思维的状态。

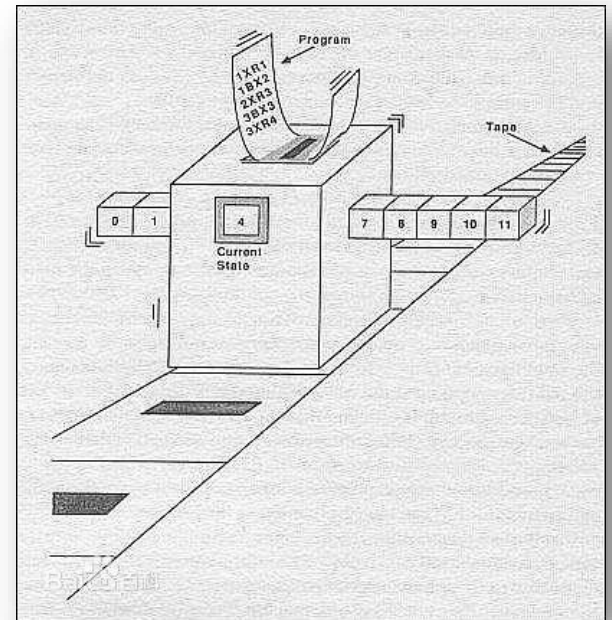


北京大学

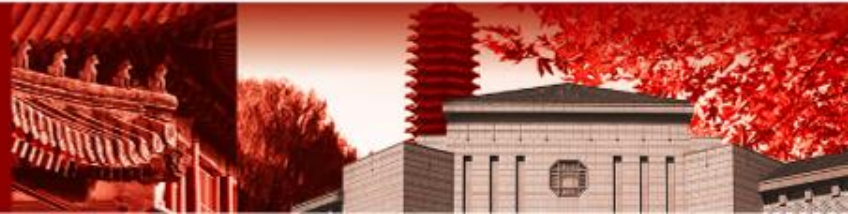


图灵机Turing Machine

- 图灵机由以下几部分构成
 - 一条无限长的纸带，分为一个个相邻的格子，每个格子可以记录一个符号
 - 一个读写头，可以在纸带上左右移动，能读出和擦写格子的字符
 - 一个状态寄存器，记录机器所在的状态，状态的数量是有限的
 - 一系列有限的控制规则
 - 每条规则指明了在当前状态下，根据读写头读入的字符来确定读写头擦写格子的字符，是否移动，是否改变状态

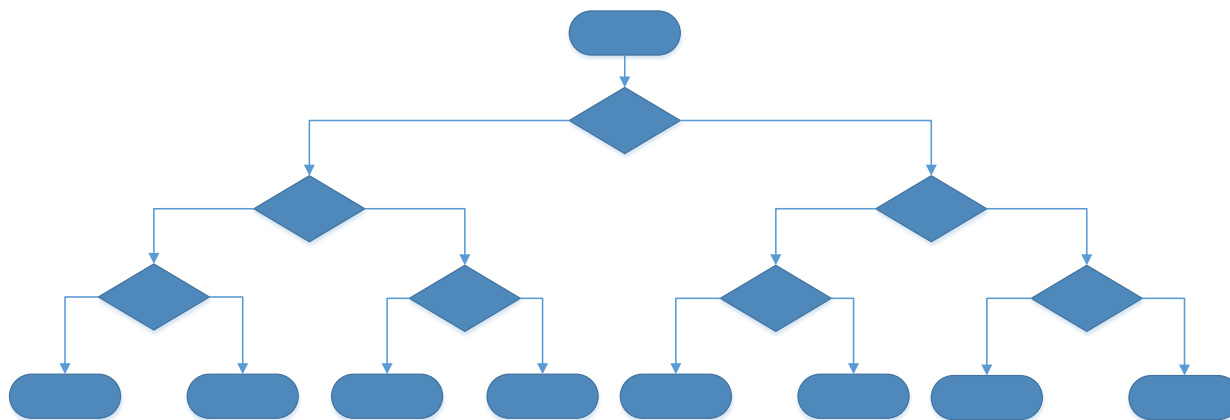


北京大学



可以通过“计算”解决的问题

- 如果用任何一个“有限能行方法”下的计算模型可以解决的问题，都算是“可计算”的
- **What:** 分类问题，可以通过树状的判定分支解决



北京大学

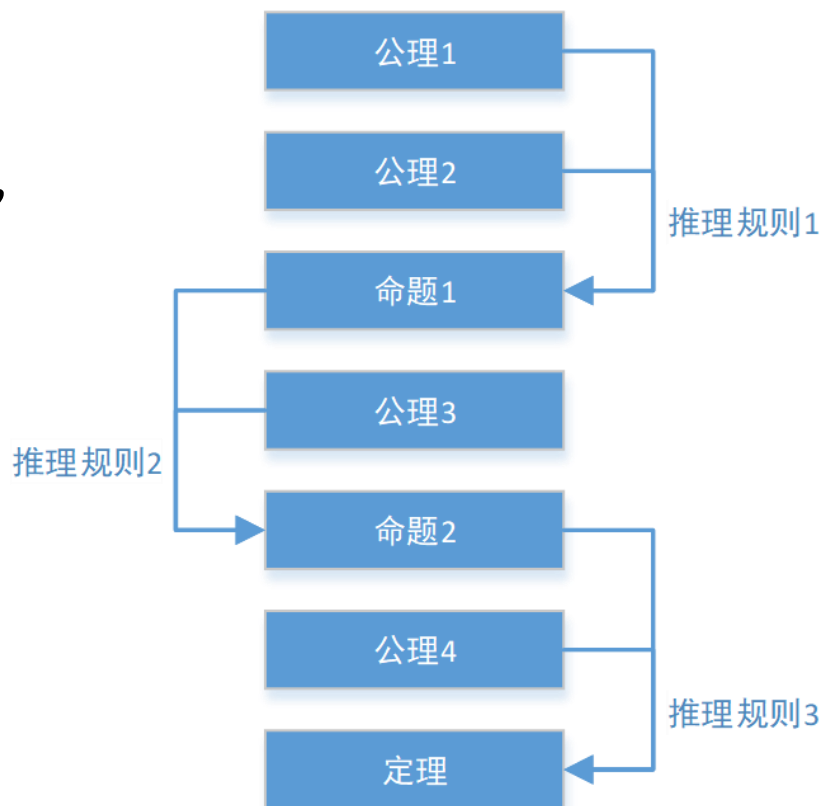


可以通过“计算”解决的问题

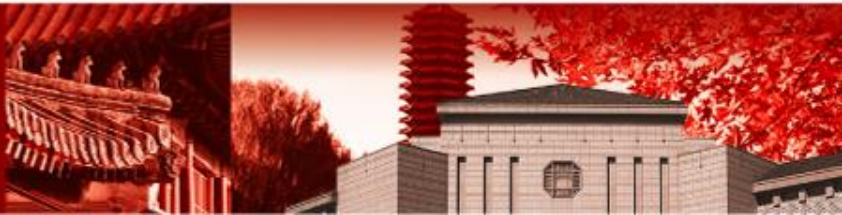
- Why: 证明问题，可以通过有限的公式序列来解决

—数学定理证明采用符号语言，从不证自明的公理出发，一步步推理得出最后待证明的定理

—我们在以往学习过的定理证明即为此类解决方法

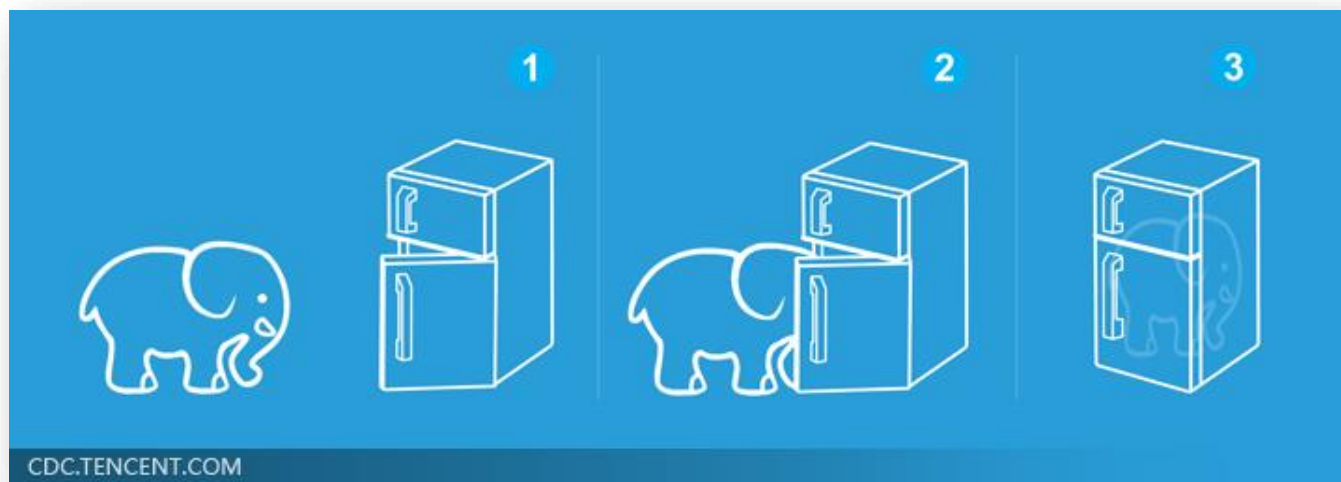


北京大学



可以通过“计算”解决的问题

- How: 过程问题，可以通过算法流程来解决
 - 解决问题的过程：算法和相应数据结构的研究，即为本课主要内容



北京大学



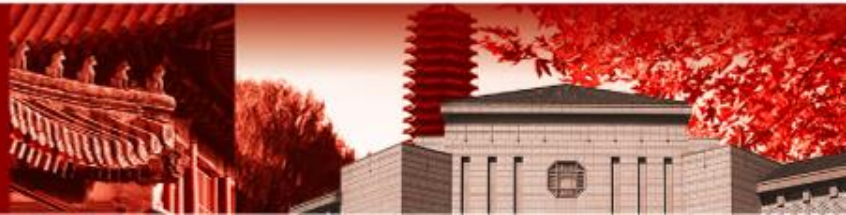
不可计算问题

- 有不少定义清晰，但无法解决的问题
 - 并不是目前尚未找到，而是在“基于有穷观点的能行方法”的条件下，已经被证明并不存在解决方案
- “停机问题”：判定任何一个程序在任何一个输入情况下是否能够停机
- 不可计算数：几乎所有的无理数，都无法通过算法来确定其任意一位是什么数字
 - 可计算数很少：如圆周率Pi，自然对数的底e

$$\pi = \frac{1}{2^6} \sum_{n=0}^{\infty} \frac{(-1)^n}{2^{10n}} \left(-\frac{2^5}{4n+1} - \frac{1}{4n+3} + \frac{2^8}{10n+1} - \frac{2^6}{10n+3} - \frac{2^2}{10n+5} - \frac{2^2}{10n+7} + \frac{1}{10n+9} \right)$$
$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$$

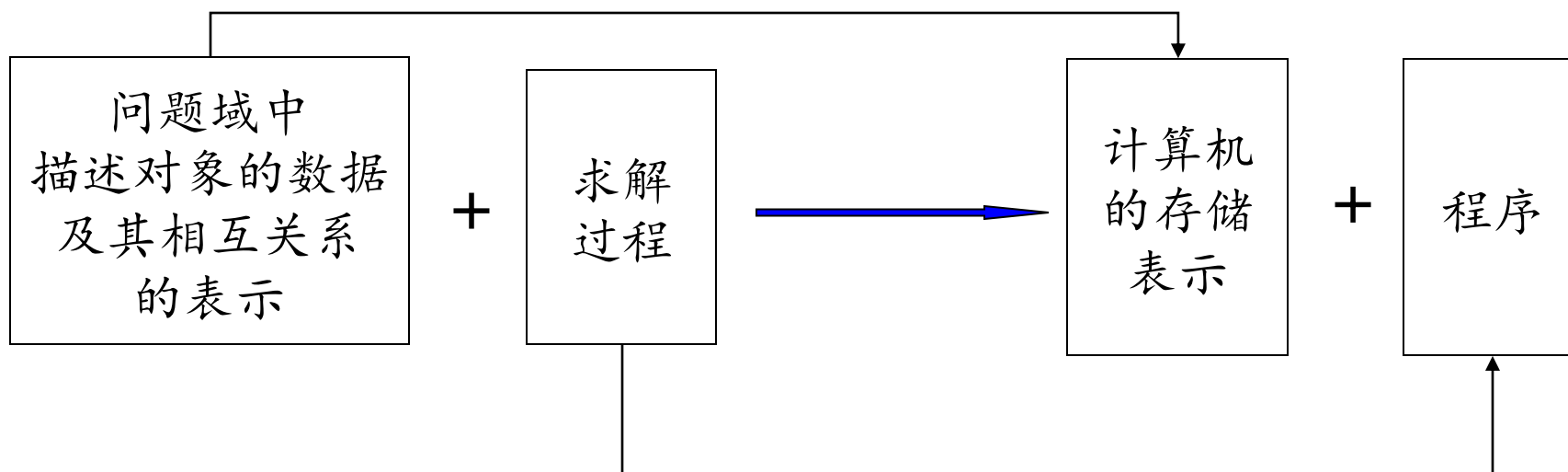


北京大学



计算机问题求解

建立求解问题的模型：把描述实际对象的数据及其相互关系映射到计算机的存储器上，编写程序模拟对象领域中的求解过程。

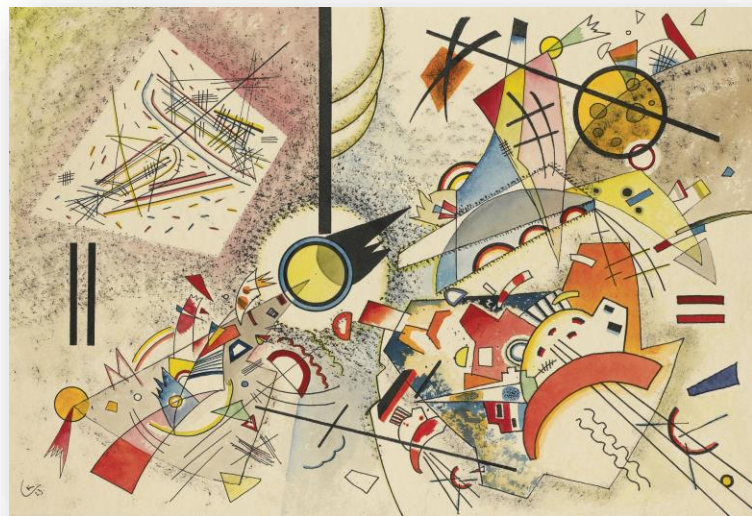


北京大学

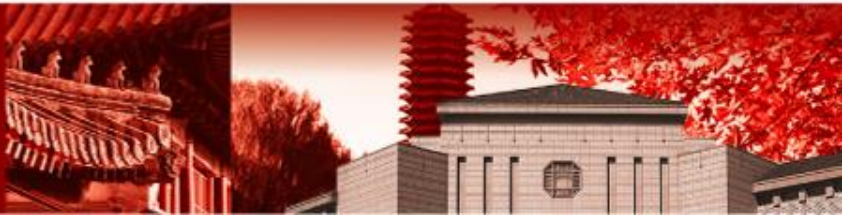


抽象：逻辑层次与物理层次

- 为了更好地处理机器相关性或独立性，引入了“抽象abstraction”的概念
- 从“**逻辑logical**”或者“**物理physical**”的不同层次上看待问题及解决方案



北京大学

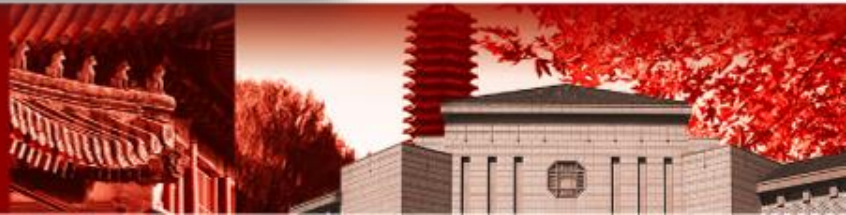


一个关于“抽象”的例子：汽车

- 从司机观点看来，汽车是可以带人去往目的地的代步工具
 - 司机上车、插钥匙、点火、换档、踩油门加速、刹车
- 从抽象的角度说，司机看到的是汽车的“**逻辑**”层次
 - 司机可以通过操作各个机构来达到运输的目的
- 这些操纵机构（方向盘、油门）就称为“**接口interface**”



北京大学



一个关于“抽象”的例子：汽车

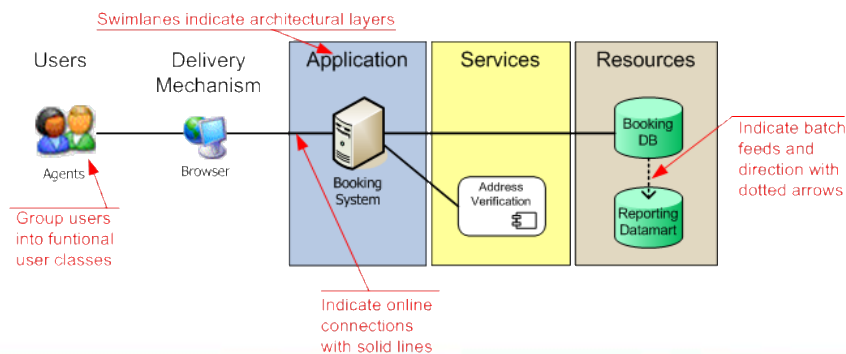
- 从汽车修理工的角度来看同一辆汽车，就会相当不同
- 他不仅要会驾驶汽车，还需要清楚每项功能是如何实现的
—如发动机工作原理，档位操作的机械结构，发动机舱内各处温度如何测量和控制等等
- 这些构成了汽车的“物理”层次
- 这些机构的工作原理就称为“实现implementation”



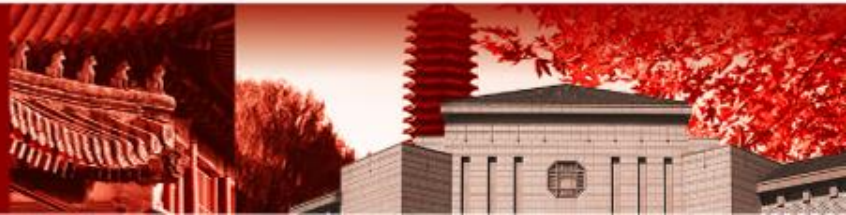
北京大学

在我们熟悉的计算机使用上也是如此

- 从一般大众用户观点看来，计算机可以用来编辑文档、收发邮件、上网聊天、处理照片等等
- 这些用户都不需要具备对计算机内部如何处理的知识
 - 利用这些功能是计算机的“逻辑”层次
- 而对于计算机科学家、程序员、技术支持、系统管理员来说，就必须要了解从硬件结构、操作系统原理到网络协议等各方面的低层次细节。

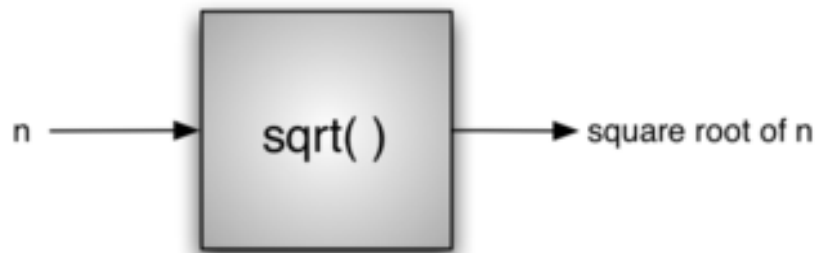


北京大学



编程开发也会涉及到抽象

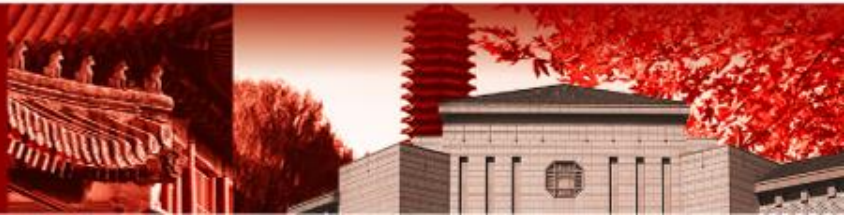
- “抽象”发生在各个不同层次上
- 即使对于程序员来说，使用编程语言进行编程，也会涉及到“抽象”
- 如计算一个数的平方根
 - 程序员可以调用编程语言的库函数`math.sqrt()`，直接得到结果，而无需关心其内部是如何实现
 - 这种功能上的“黑盒子”称作“过程抽象procedural abstraction”



```
>>> import math
>>> math.sqrt(16)
4.0
>>>
```



北京大学



什么是编程 (Programming)

- 编程是通过一种程序设计语言将**算法**变为计算机**可执行代码**的过程
 - 没有算法，编程无从谈起
- 图灵奖获得者Niklaus Wirth的著名公式：
算法+数据结构=程序
 - Pascal语言设计者
 - （另：尼克劳斯·维尔特于1995年提出了一条幽默定律：软件变慢的速度永远快过硬件变快的速度

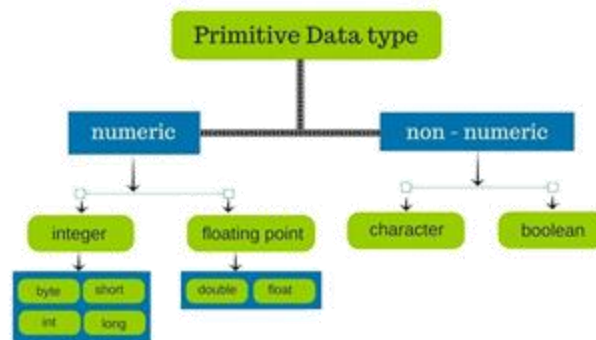
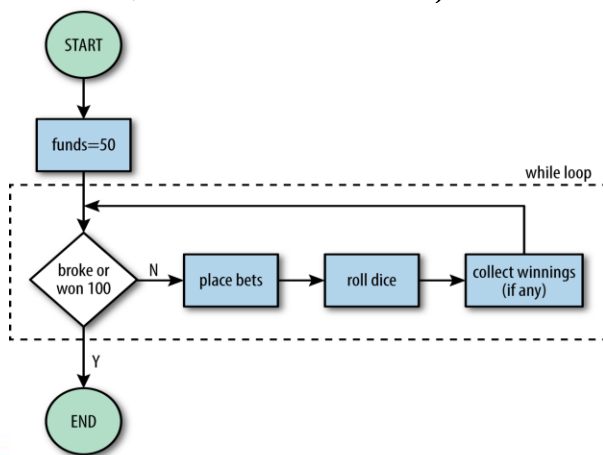


北京大学

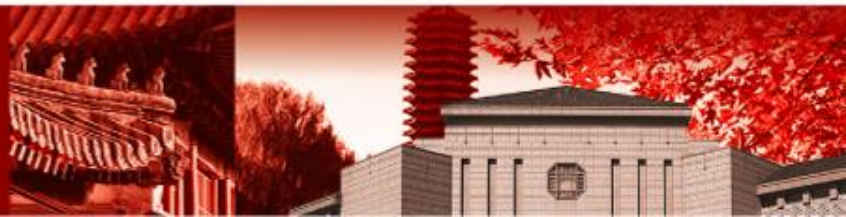


程序设计语言为编程提供的机制

- 程序设计语言需要提供实现“**过程**”和“**数据**”的机制
 - 具体表现为程序设计语言中的“控制结构”和“数据类型”
- 实现基本控制结构，程序设计语言均有**语句**相对应
 - 顺序处理、分支选择、循环迭代
- 提供最基本的**数据类型**来表示数据，如整数、字符等
 - 但对于复杂的问题而言，这些基本数据类型不利于算法的表达

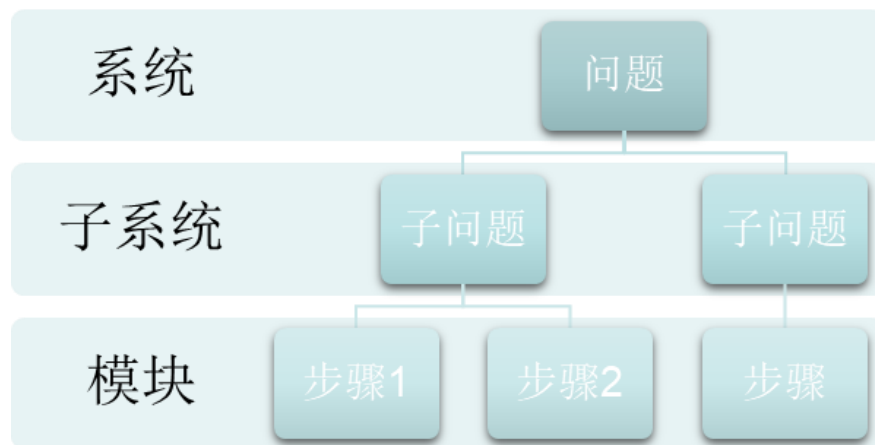


北京大学



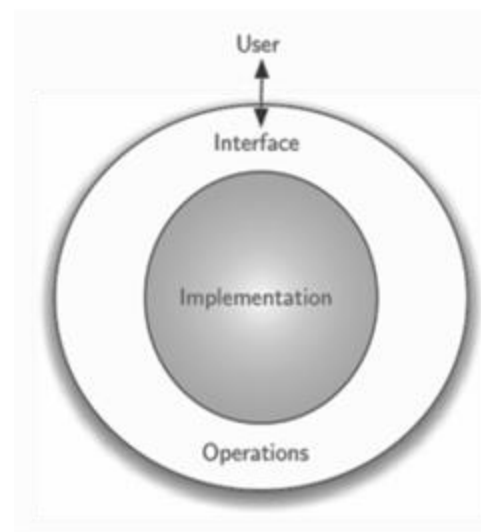
为什么要学习数据结构和抽象数据类型

- 还需要引入**控制复杂度**的方法，便于清晰高效地表达算法
- 为了控制问题和问题解决过程的复杂度，我们需要利用**抽象**来保持问题的“**整体感**”而不会陷入到过多的细节中去
- 这要求对现实问题进行建模的时候，对算法所要处理的数据，要保持与问题本身的一致性，不要有太多与问题无关的细节



逻辑层次：数据抽象ADT

- 前面谈到的“过程抽象”启发我们进行“**数据抽象**”
 - 将复杂的内部实现隐藏起来，无需直接操作底层数据，从而控制复杂度
- 相对于基本数据类型的**抽象数据类型ADT: Abstract Data Type**
 - ADT是对数据进行处理的一种逻辑描述，并不涉及如何实现这些处理
 - 例如，图书馆的书籍分类系统中，抽象出树结构ADT，实现层次化管理，提高查找效率
 - 又例如，为了维护社交网络关系，抽象出图结构ADT
- ADT建立了一种对数据的“封装encapsulation”
 - 封装技术将可能的处理实现细节隐蔽起来，能有效控制算法的复杂度

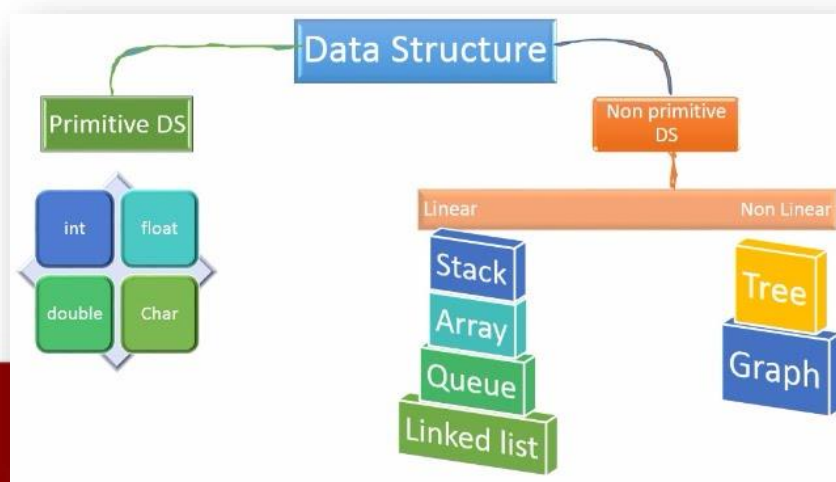


北京大学



物理层次：数据结构Data Structure

- 数据结构是对ADT的具体实现
 - 同一种ADT可以采用不同的数据结构来实现
- 数据结构采用程序设计语言的**控制结构**和**基本数据类型**来实现ADT所提供的**逻辑接口**
 - 属于ADT的“物理”层次
- 对数据实现“**逻辑**”层次和“**物理**”层次的分离，可以定义复杂的数据模型来解决问题，而不需要考虑此模型如何实现



接口的两端

- 由于对抽象数据类型可以有多种实现方案
- 独立于实现的数据模型
 - 让底层程序员专注于实现和优化数据处理，而无须改变使用接口
 - 让用户专注于问题的解决过程
- 如电动车与汽油车
 - 底层动力实现不同
 - 但开车的操作接口（方向盘、油门、刹车、档位）基本都是相同的



北京大学



《数据结构与算法》

- 数据结构的三要素：逻辑结构、存储结构、运算

- 逻辑结构：

线性结构 \subseteq 树形结构
 \subseteq 图结构

- 存储结构：

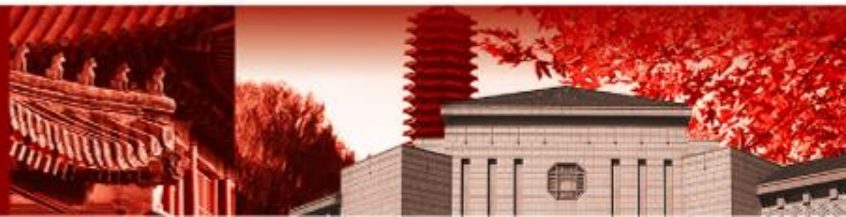
顺序方法、链接方法
散列方法、索引方法

- 运算：

增、删、改、查等基本运算
遍历、排序等复杂运算



北京大学



为什么要学习算法

- 首先，学习各种不同问题的解决方案
 - 有助于我们在面对未知问题的时候，能够根据类似问题的解决方案来更好解决
- 其次，各种算法通常有较大差异
 - 我们可以通过**算法分析技术**来评判算法本身的特性
 - 而不仅仅根据算法在特定机器和特定数据上运行的表现来评判它
 - 即使同一个算法，在不同的运行环境和输入数据的情况下，其表现的差异可能也会很大



北京大学



为什么要学习算法

- 在某些情况下，我们或许会碰到棘手的难题
 - 得能区分这种问题是根本不存在算法
 - 还是能找到算法，但需要耗费大量的资源
- 某些问题的解决可能需要一些折衷的处理方式
 - 我们需要学会在不同算法之间进行选择，以适合当前条件的要求

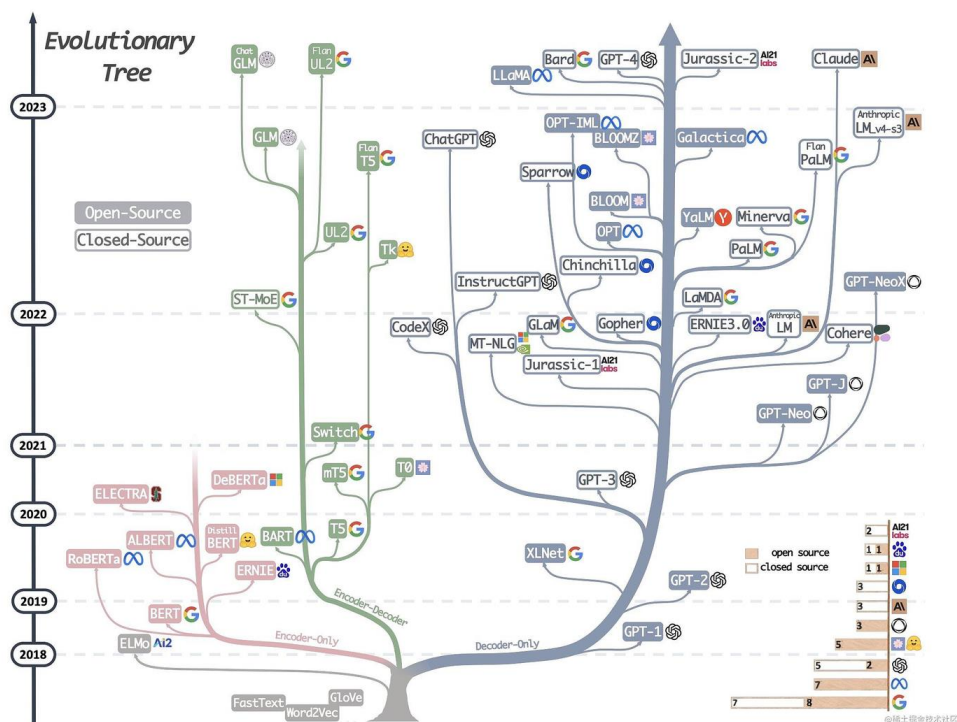


北京大学



为什么要学习算法：通用人工智能的影响

2017年谷歌发表的论文”Attention Is All you Need”拉开了大语言模型（LLM）发展的序幕，根据技术选型的不同逐渐分为三支。其中仅使用解码器(Decoder-Only)的大模型经过5年的蓬勃发展，最终于近年出现了ChatGPT、LLaMA、GPT-4、Claude、Copilot、DeepSeek等实用产品。



图片来源：Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond



北京大学



为什么要学习算法：通用人工智能的影响

ChatGPT: OpenAI开发的AI聊天机器人程序，于2022年11月推出ChatGPT, 2023年3月15日推出其升级版GPT-4. 2024年5月，OpenAI发布了GPT-4o模型，具备处理文本、图像和音频的能力。2024年9-12月，OpenAI推出了 o1 模型，旨在通过延长思考时间来解决更复杂的问题。



使用Python写一个猜数字大小的游戏

好的，下面是一个简单的Python猜数字游戏代码示例，规则是电脑随机生成一个1-100之间的整数，玩家通过输入猜测的数字来猜测这个数字，直到猜中为止。每次猜测后，程序会告诉玩家猜测的数字是太大还是太小。代码如下：

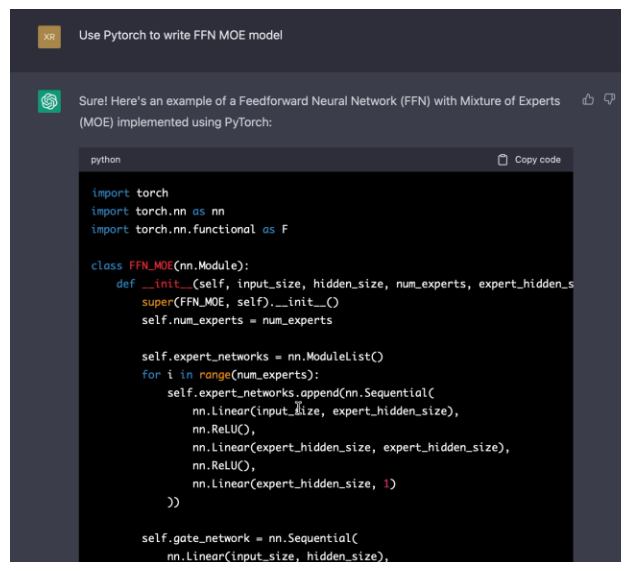
```
python
import random

number = random.randint(1, 100)
guess = int(input("请输入一个1-100之间的整数: "))

while guess != number:
    if guess > number:
        guess = int(input("猜测的数字太大了，请重新输入: "))
    else:
        guess = int(input("猜测的数字太小了，请重新输入: "))

print("恭喜你，猜中了！答案是", number)
```

这是一个基本的猜数字游戏，你可以根据需要对代码进行更改和扩展，比如加入猜测次数限制、错误检查、界面设计等等。



Use PyTorch to write FFN MOE model

Sure! Here's an example of a Feedforward Neural Network (FFN) with Mixture of Experts (MOE) implemented using PyTorch:

```
python
import torch
import torch.nn as nn
import torch.nn.functional as F

class FFN_MOE(nn.Module):
    def __init__(self, input_size, hidden_size, num_experts, expert_hidden_size):
        super(FFN_MOE, self).__init__()
        self.num_experts = num_experts

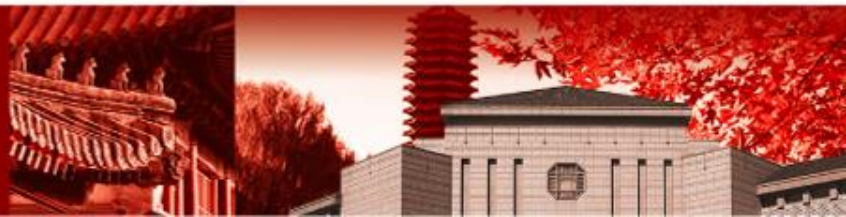
        self.expert_networks = nn.ModuleList()
        for i in range(num_experts):
            self.expert_networks.append(nn.Sequential(
                nn.Linear(input_size, expert_hidden_size),
                nn.ReLU(),
                nn.Linear(expert_hidden_size, expert_hidden_size),
                nn.ReLU(),
                nn.Linear(expert_hidden_size, 1)
            ))

        self.gate_network = nn.Sequential(
            nn.Linear(input_size, hidden_size),
```

- ChatGPT的编码能力能够胜任简单重复的编程任务
- ChatGPT的上下文学习能力使其也能完成简单业务流程的编写

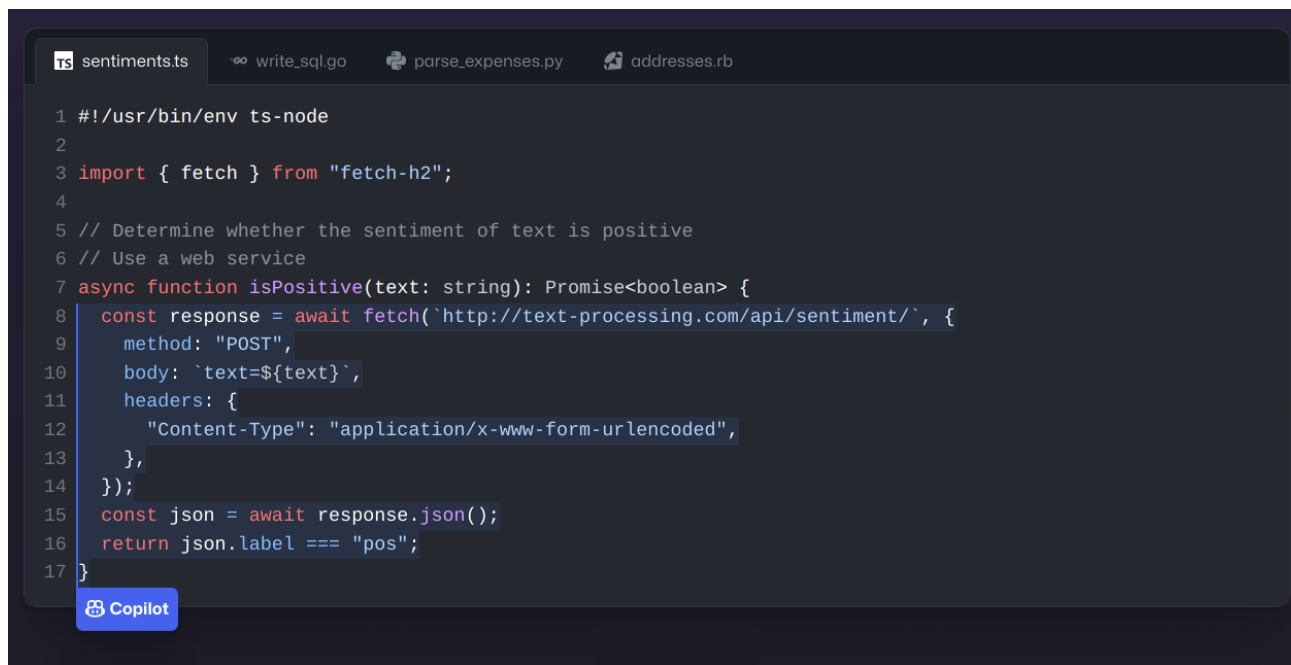


北京大学



为什么要学习算法：通用人工智能的影响

GitHub Copilot: GitHub和OpenAI合作开发的一个人工智能工具。用户在使用IDE时，只需用注释编写代码意图和函数头，Copilot即可自动补全代码。2021年6月29日对外公开。



```
1 #!/usr/bin/env ts-node
2
3 import { fetch } from "fetch-h2";
4
5 // Determine whether the sentiment of text is positive
6 // Use a web service
7 async function isPositive(text: string): Promise<boolean> {
8   const response = await fetch('http://text-processing.com/api/sentiment/', {
9     method: "POST",
10    body: `text=${text}`,
11    headers: {
12      "Content-Type": "application/x-www-form-urlencoded",
13    },
14  });
15  const json = await response.json();
16  return json.label === "pos";
17 }
```

Copilot



北京大学

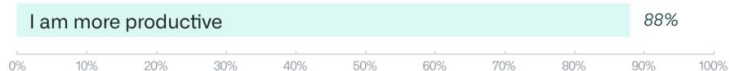


为什么要学习算法：通用人工智能的影响

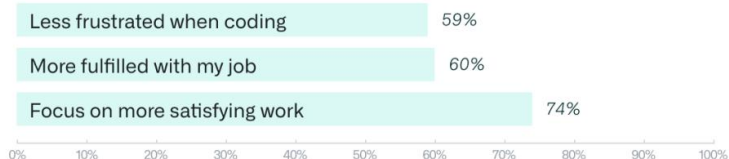
- **GitHub Copilot:** 可以解释复杂的编码概念，检测安全漏洞，编写单元测试，帮助每个开发人员快速进行创新

When using GitHub Copilot...

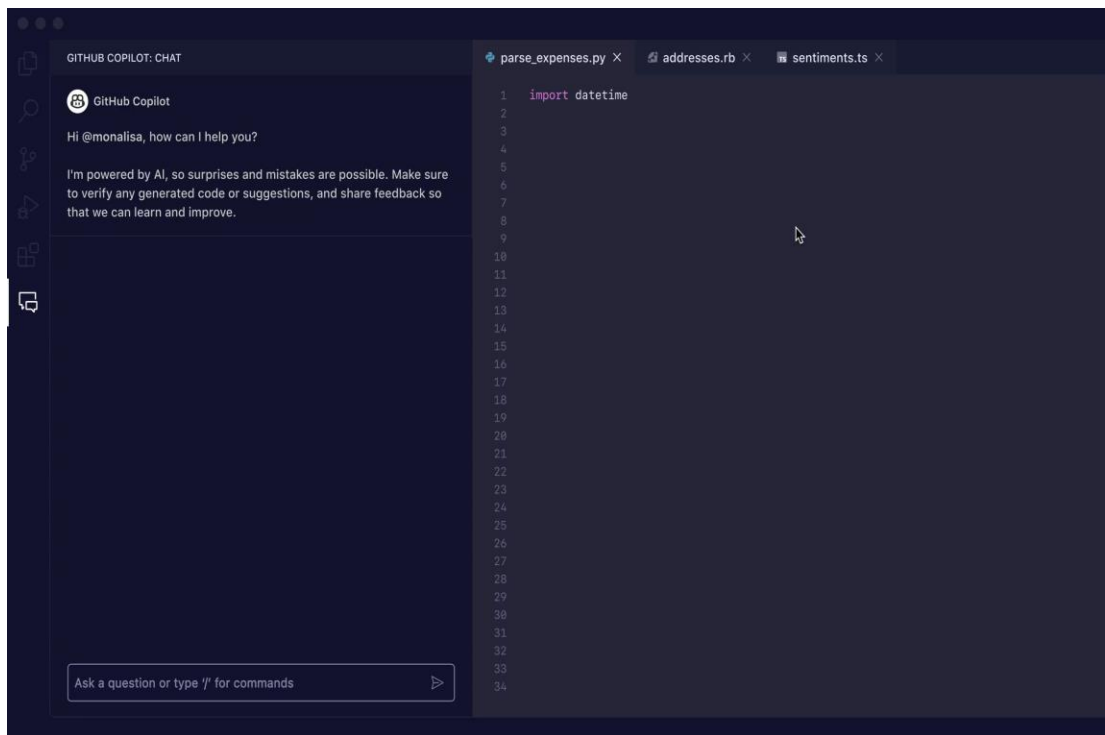
Perceived Productivity



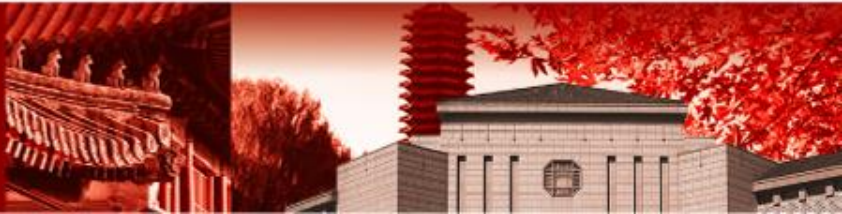
Satisfaction and Well-being*



Efficiency and Flow*



北京大学



为什么要学习算法：通用人工智能的影响

DeepSeek: 深度求索公司推出的通用人工智能模型，聚焦于搜索引擎技术领域，在处理复杂查询、理解语义等方面表现出色，为用户提供更优质的搜索体验；同时也具有数学推理与代码生成能力

- 能够胜任简单重复的编程任务
- 具备一定的推理能力，能够解答基本的算法题目

```
python 复制

import random

def guess_number_game():
    # 生成一个1到100之间的随机数字
    target_number = random.randint(1, 100)
    attempts = 0

    print("欢迎来到猜数字游戏！")
    print("我已经想好了一个1到100之间的数字，快来猜猜看吧！")

    while True:
        try:
            # 获取用户输入
            guess = int(input("请输入你猜的数字 (1-100): "))
            attempts += 1

            # 判断猜测结果
            if guess < target_number:
                print("❗ 你猜的数字太小了，再大一点！")
            elif guess > target_number:
                print("❗ 你猜的数字太大了，再小一点！")
            else:
                print(f"🎉 恭喜！你猜对了！你用了 {attempts} 次猜中了数字 {target_number}！")
                break # 猜对后退出循环
        except ValueError:
            print("⚠️ 请输入有效的数字！")

    print("游戏结束，感谢游玩！")

# 运行游戏
guess_number_game()
```

```
python 复制

def count_ways(M, N):
    # 创建一个二维数组来存储中间结果
    dp = [[0] * (N + 1) for _ in range(M + 1)]

    # 初始化基本情况
    for i in range(M + 1):
        dp[i][1] = 1 # 只有一个盘子，只有一种分法
    for j in range(N + 1):
        dp[0][j] = 1 # 没有苹果，只有一种分法

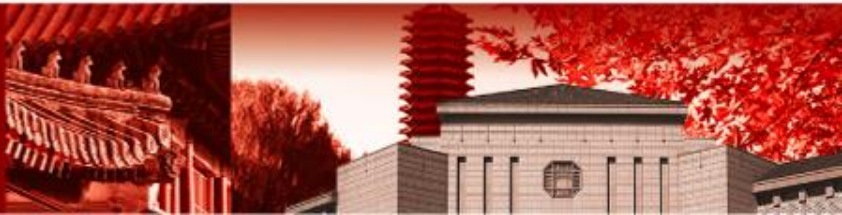
    # 填充动态规划表
    for i in range(1, M + 1):
        for j in range(2, N + 1):
            if i < j:
                dp[i][j] = dp[i][i] # 如果苹果比盘子少，等同于将 i 个苹果放入 i 个盘子
            else:
                dp[i][j] = dp[i][j - 1] + dp[i - j][j] # 递归关系

    return dp[M][N]

# 读取输入
t = int(input())
for _ in range(t):
    M, N = map(int, input().split())
    print(count_ways(M, N))
```



北京大学



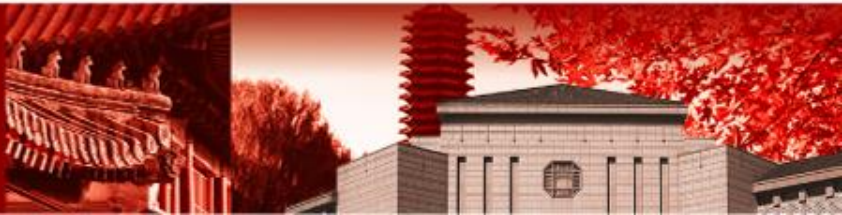
为什么要学习算法：通用人工智能的影响

• DeepSeek带来的深刻影响和思考：

- 拿下了美国和中国区App Store免费榜的双料第一，成为首个超越OpenAI ChatGPT的AI助手类应用。
- 外媒用“震动硅谷”形容DeepSeek带给美国的行业震撼，其主打的低成本开源模型概念更冲击华尔街，英伟达市值一周缩水超5000亿美元。
- 这个不足140人的精悍团队，在人工智能基础研究领域取得的突破，让众多国际科技巨头都为之侧目。
- 这支平均年龄仅28岁的研发团队，几乎全部成员均在中国本土完成高等教育，彻底颠覆了科技行业对人才来源的固有认知。
- 团队成员逾八成来自北大、清华等国内顶尖学府，核心研发骨干中尚有在读博士生的身影。这支年轻劲旅的出现呈现出的是全球创新版图正在发生深刻重构。

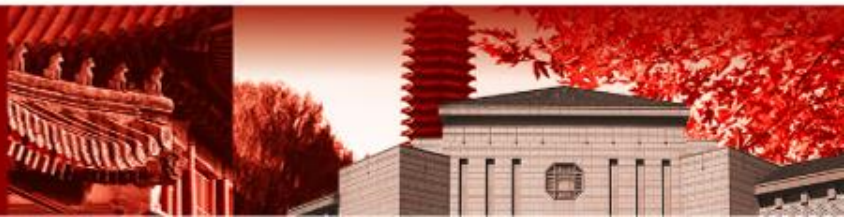


北京大学



为什么要学习算法：通用人工智能的影响

- 1、ChatGPT、DeepSeek、GitHub Copilot、OpenAI Sora大家用过吗？对于学习、工作和生活带来哪些影响？
- 2、软件开发和算法设计有什么影响？

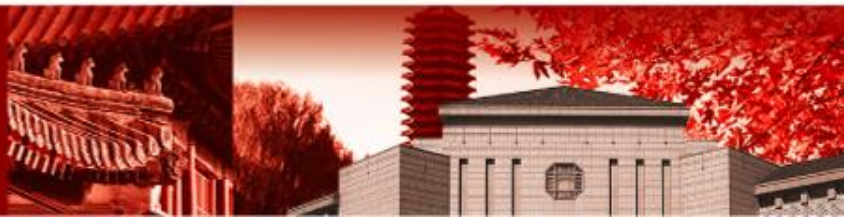


为什么要学习算法：通用人工智能的影响

- 通用人工智能的推理、生成、检索能力提升了人们的效率
 - 对于已有的知识，简单重复的基本任务，AI 能够帮助人们解决
- 但对于人机物高度融合的复杂问题，AI 仍无法给出解决方案
 - 理解算法的正确性，为新的问题设计新的算法，或为复杂系统进行分析与设计，通用人工智能无法胜任
 - 算法设计仍然需要人来进行
- 我们应当更加注意培养创新性思维方式，分析问题和解决问题的能力，以及针对复杂系统的分析和设计思想养，而不再是仅仅写好代码！



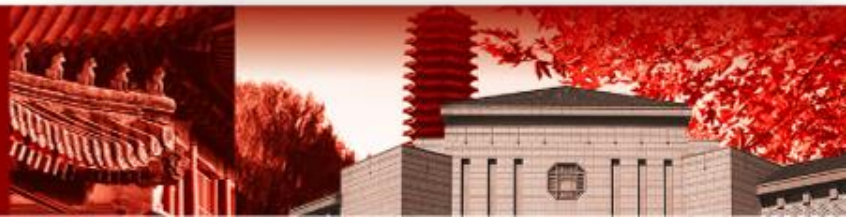
北京大学



什么是算法分析

- 如何对比两个程序？
 - 看起来不同，但解决同一个问题的程序，哪个“更好”？
- 程序和算法的区别
 - 算法是对问题解决的分步描述
 - 程序则是采用某种编程语言实现的算法，同一个算法通过不同的程序员采用不同的编程语言，能产生很多程序
- 我们来看一段程序，完成从1到n的累加，输出总和
 - 设置累计变量=0
 - 从1到n循环，逐次累加到累计变量
 - 返回累计变量

```
def sumOfN(n):  
    theSum = 0  
    for i in range(1, n + 1):  
        theSum = theSum + i  
    return theSum  
  
print(sumOfN(10))
```



什么是算法分析

- 再看第二段程序，是否感觉怪怪的？
 - 但实际上本程序功能与前面那段相同
 - 这段程序失败之处在于：
 - 变量命名
 - 有无用的垃圾代码
- 比较程序的“好坏”，有更多因素
 - 代码风格、可读性等等
- 本课程中，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

```
def foo(tom):  
    fred = 0  
    for bill in range(1, tom + 1):  
        barney = bill  
        fred = fred + barney  
    return fred  
  
print(foo(10))
```



北京大学



计算资源指标

- 那么何为计算资源？
- 一种是算法解决问题过程中需要的**存储空间或内存**
 - 但存储空间受到问题自身数据规模的变化影响
 - 不容易区分哪些存储空间是描述问题本身所需，哪些是算法占用
- 另一种是算法的**执行时间**
 - 我们可以对程序进行实际运行测试，获得真实的运行时间
- Python中有一个**time**模块，可以获取计算机系统当前时间
 - 在算法开始前和结束后分别记录系统时间，即可得到运行时间

```
>>> help(time.time)
Help on built-in function time in module time:
```

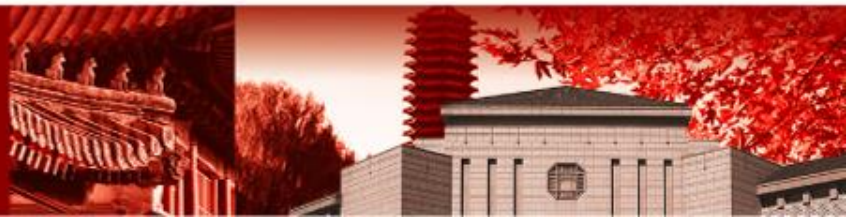
```
time(...)
    time() -> floating point number
```

```
Return the current time in seconds since the Epoch.
Fractions of a second may be present if the system clock provides them.
```

```
>>> import time
>>> time.time()
1551764033.4839659
>>>
```



北京大学



运行时间检测

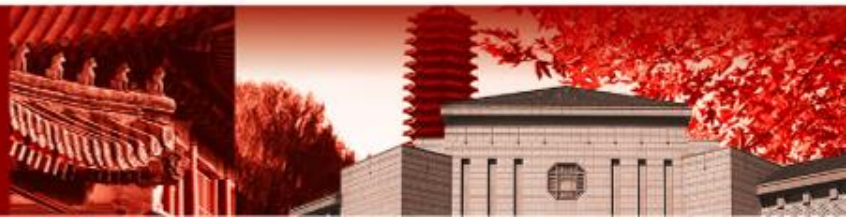
- 累计求和程序的运行时间检测
 - 程序增加了总运行时间
 - 函数返回一个元组tuple
 - 包括累计和，以及运行时间（秒）
- 在交互窗口连续运行5次看看
 - 1到10,000累加
 - 每次运行约需0.0007秒

```
1 import time
2
3
4 def sumOfN2(n):
5     start = time.time()
6     theSum = 0
7     for i in range(1, n + 1):
8         theSum = theSum + i
9     end = time.time()
10    return theSum, end - start
11
12
13 for i in range(5):
14     print("Sum is %d required %10.7f seconds"
15           % sumOfN2(10000))
```

```
Sum is 50005000 required 0.0007980 seconds
Sum is 50005000 required 0.0007021 seconds
Sum is 50005000 required 0.0007031 seconds
Sum is 50005000 required 0.0007219 seconds
Sum is 50005000 required 0.0007060 seconds
```



北京大学



运行时间检测

- 如果累加到100,000?
 - 看起来运行时间增加到10,000的10倍

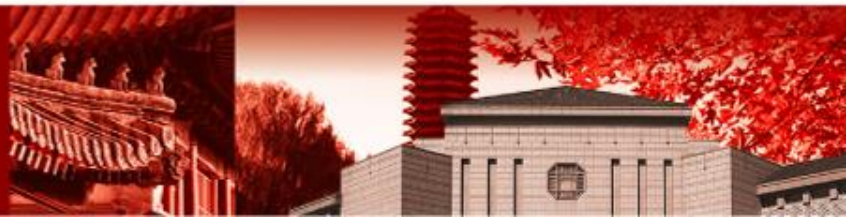
```
Sum is 5000050000 required 0.0078530 seconds
Sum is 5000050000 required 0.0078511 seconds
Sum is 5000050000 required 0.0087960 seconds
Sum is 5000050000 required 0.0082700 seconds
Sum is 5000050000 required 0.0077040 seconds
```

- 进一步累加到1,000,000?
 - 运行时间又是100,000的10倍了

```
Sum is 50000050000 required 0.0817859 seconds
Sum is 50000050000 required 0.0781529 seconds
Sum is 50000050000 required 0.0803380 seconds
Sum is 50000050000 required 0.0783160 seconds
Sum is 50000050000 required 0.0776238 seconds
```



北京大学



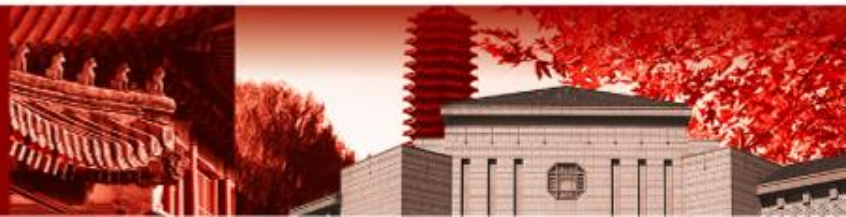
第二种无迭代的累计算法

- 利用求和公式的无迭代算法
- 采用同样的方法检测运行时间
 - 10,000; 100,000; 1,000,000
 - 10,000,000; 100,000,000

```
17 def sumOfN3(n):  
18     start = time.time()  
19     theSum = (n * (n + 1)) / 2  
20     end = time.time()  
21     return theSum, end - start
```

```
Sum is 50005000 required 0.0000010 seconds  
Sum is 5000050000 required 0.0000000 seconds  
Sum is 500000500000 required 0.0000010 seconds  
Sum is 50000005000000 required 0.0000000 seconds  
Sum is 5000000050000000 required 0.0000169 seconds
```

- 需要关注的两点
 - 这种算法的运行时间比前种都短很多
 - 运行时间与累计对象n的大小没有关系（前种算法是倍数增长关系）
- 新算法运行时间几乎与需要累计的数目无关



运行时间检测的分析

- 观察一下第一种迭代算法
 - 包含了一个循环，可能会执行更多语句
 - 这个循环运行次数跟累加值 n 有关系， n 增加，循环次数也增加
- 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
- 所以我们需要更好的方法来衡量算法的运行时间
 - 这个指标与具体的机器、程序、运行时段，与编译器都无关

```
1 import time
2
3
4 def sumOfN2(n):
5     start = time.time()
6     theSum = 0
7     for i in range(1, n + 1):
8         theSum = theSum + i
9     end = time.time()
10    return theSum, end - start
11
12
13 for i in range(5):
14     print("Sum is %d required %10.7f seconds"
15           % sumOfN2(10000))
```



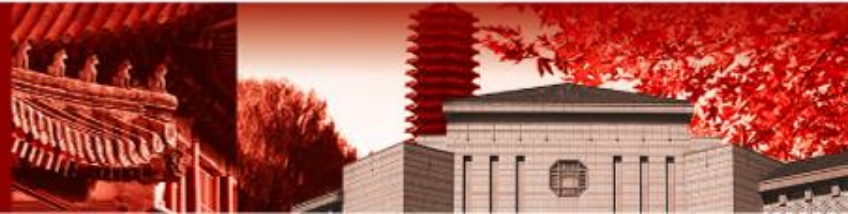
北京大学



大O表示法 (Big-O)

- 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
- **赋值语句**是一个合适的选择
 - 赋值语句同时包含了 (表达式) 计算和 (变量) 存储两个基本资源
 - 仔细观察程序设计语言特性, 除了与计算资源无关的定义语句之外, 主要就是三种控制流语句和赋值语句, 而控制流仅仅起了组织赋值语句的作用, 并不实施处理。
- 分析SumOfN的赋值语句数量
 - 对于“问题规模” n
 - 赋值语句数量 $T(n)=1+n$

```
31 def sumOfN(n):  
32     1 theSum = 0  
33     n for i in range(1, n + 1):  
34         theSum = theSum + i  
35     1 return theSum
```



大O表示法 (Big-O)

- 问题规模影响算法执行时间

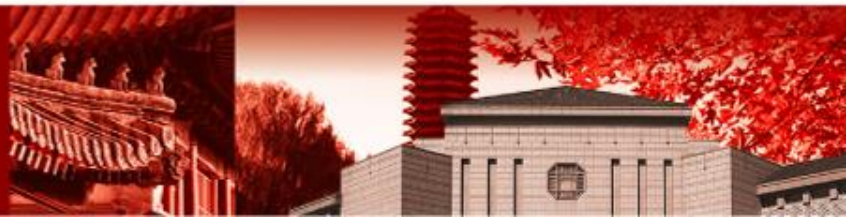
- 在前 n 个自然数累计求和的算法中，需要累计的自然数个数合适作为问题规模的指标
- 前100,000个自然数求和对比前1,000个自然数求和，算是同一问题的更大规模
- 算法分析的目标是要找出问题规模会怎么影响一个算法的执行时间

- 数量级函数 (Order of Magnitude function)

- 基本操作数量函数 $T(n)$ 的精确值并不是特别重要，重要的是 $T(n)$ 中起决定性因素的主导部分
- 用动态的眼光看，就是当问题规模增大的时候， $T(n)$ 中的一些部分会盖过其它部分的贡献
- 数量级函数描述了 $T(n)$ 中随着 n 增加而增加速度最快的部分
- 称作“大O”表示法，记作 $O(f(n))$ ，其中 $f(n)$ 表示 $T(n)$ 中的主导部分



北京大学

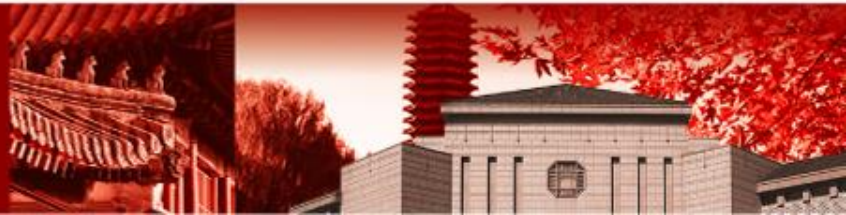


确定运行时间数量级大O的方法

- 例1: $T(n)=1+n$
 - 当 n 增大时, 常数1在最终结果中显得越来越无足轻重
 - 所以可以去掉1, 保留 n 作为主要部分, 运行时间数量级就是 $O(n)$
- 例2: $T(n)=5n^2+27n+1005$
 - 当 n 很小时, 常数1005其决定性作用
 - 但当 n 越来越大, n^2 项就越来越重要, 其它两项对结果的影响则越来越小
 - 同样, n^2 项中的系数5, 对于 n^2 的增长速度来说也影响不大
 - 所以可以在数量级中去掉 $27n+1005$, 以及系数5的部分, 确定为 $O(n^2)$
- 有时决定运行时间的不仅是问题规模, 某些具体数据也会影响算法运行时间
 - 分为最好、最差和平均情况, 平均状况体现了算法的主流性能
 - 对算法的分析要看主流, 而不被某几种特定的运行状况所迷惑

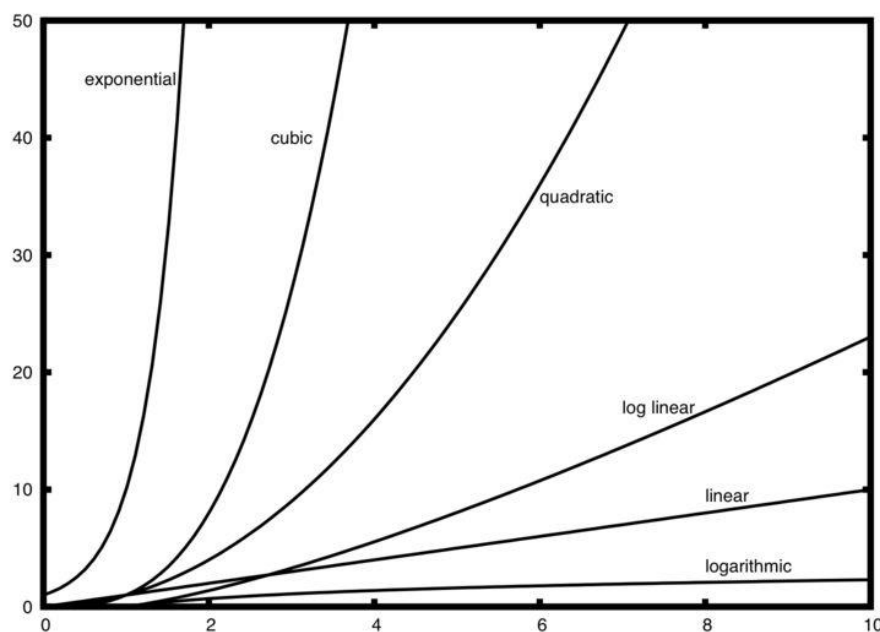


北京大学



常见的大O数量级函数

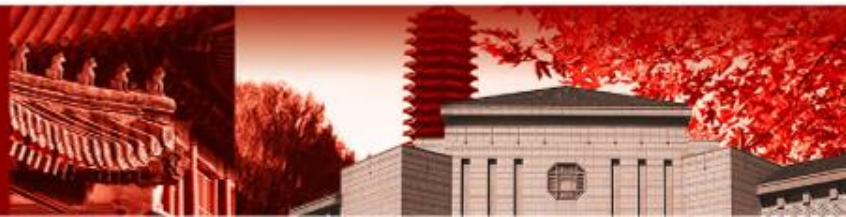
- 通常当n较小时，难以确定其数量级
- 当n增长到较大时，容易看出其主要变化量级



$f(n)$	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

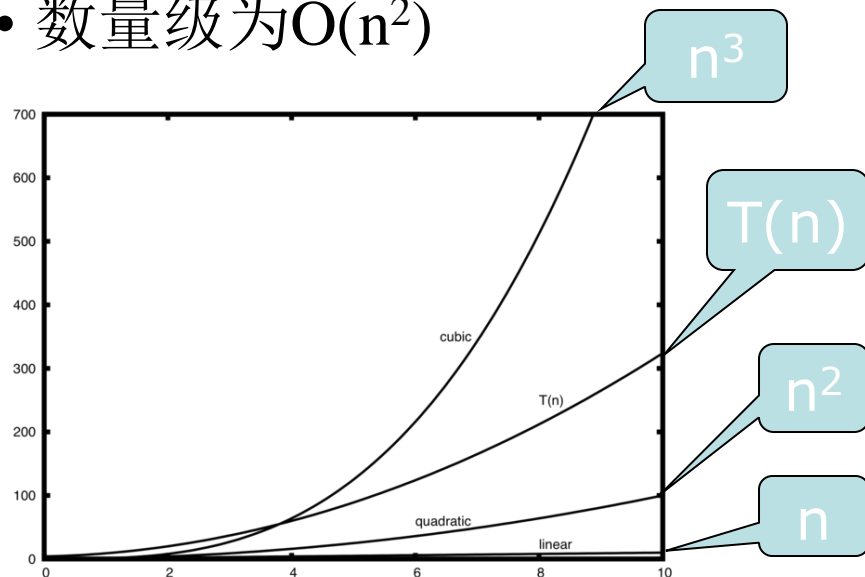


北京大学



从代码分析确定执行时间数量级函数

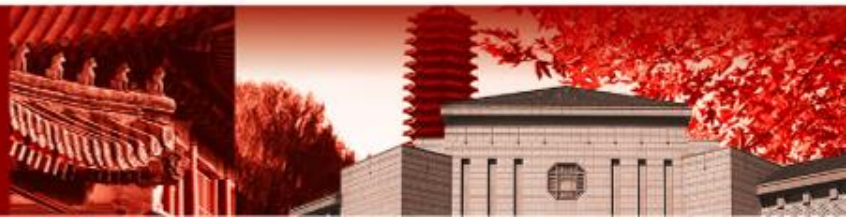
- 代码赋值语句可以分为4个部分
 - $T(n)=3+3n^2+2n+1=3n^2+2n+4$
- 仅保留最高阶项 n^2 ，去掉所有系数
- 数量级为 $O(n^2)$



```
38     a = 5
39     b = 6
40     c = 10
41     for i in range(n):
42         for j in range(n):
43             x = i * i
44             y = j * j
45             z = i * j
46     for k in range(n):
47         w = a * k + 45
48         v = b * b
49     d = 33
```



北京大学



其它两个算法复杂度表示法（大 Ω 、大 Θ ）

- 大 O 表示法

- 表示了所有上限中最小的那个上限

- 大 Ω 表示法

- 表示了所有下限中最大的那个下限

$$f(n) = \Omega(g(n)) \text{ 当且仅当 } g(n) = o(f(n))$$

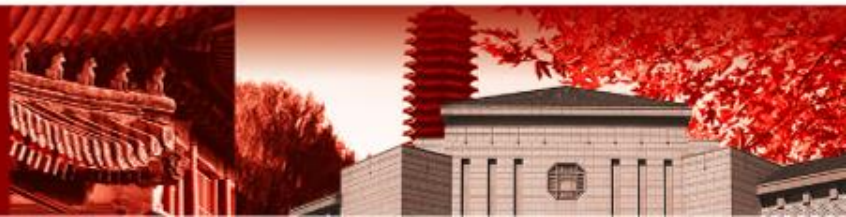
- 大 Θ 表示法

- 如果上下限相同，那么就可以用大 Θ 表示

$$f(n) = \Theta(g(n)) \text{ 当且仅当 } f(n) = O(g(n)) \text{ 且 } f(n) = \Omega(g(n))$$



北京大学



例：“变位词”判断问题

- 问题描述

- 所谓“变位词”是指两个词之间存在组成字母的重新排列关系
 - 如：heart和earth，python和typhon
 - 为了简单起见，假设参与判断的两个词仅由小写字母构成，而且长度相等
- 解题目标：写一个bool函数，以两个词作为参数，返回真假，表示这两个词是否变位词

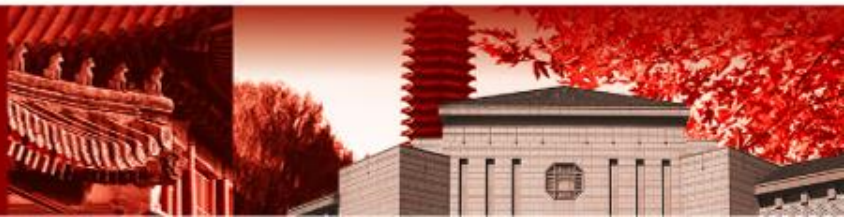
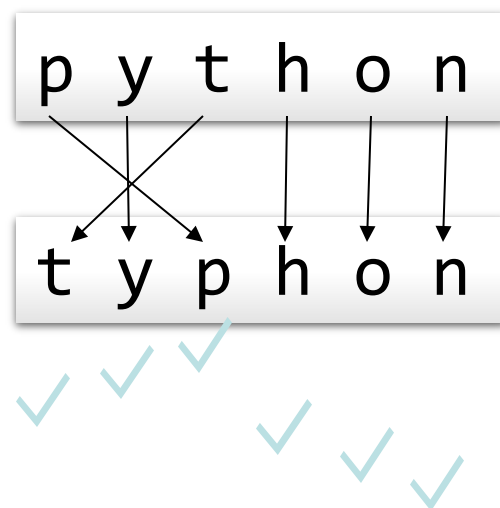


北京大学



解法1：逐字检查

- 解法思路为将字符串1中的字符逐个到字符串2中检查是否存在
 - 存在就“打勾”标记（防止重复检查）
 - 如果每个字符都能找到，则两个词是变位词
 - 只要有1个字符找不到，就不是变位词
- 程序技巧
 - 实现“打勾”标记：将字符串2对应字符设为None
 - 由于字符串是不可变类型，需要先复制到列表中



解法1：逐字检查-程序代码

```
1 def anagramSolution1(s1, s2):
2     alist = list(s2)
3     pos1 = 0
4     stillOK = True
5     while pos1 < len(s1) and stillOK:
6         pos2 = 0
7         found = False
8         while pos2 < len(alist) and not found:
9             if s1[pos1] == alist[pos2]:
10                 found = True
11             else:
12                 pos2 = pos2 + 1
13         if found:
14             alist[pos2] = None
15         else:
16             stillOK = False
17         pos1 = pos1 + 1
18     return stillOK
19
20
21 print(anagramSolution1('abcd', 'dcba'))
```

复制s2到列表

循环s1的每个字符

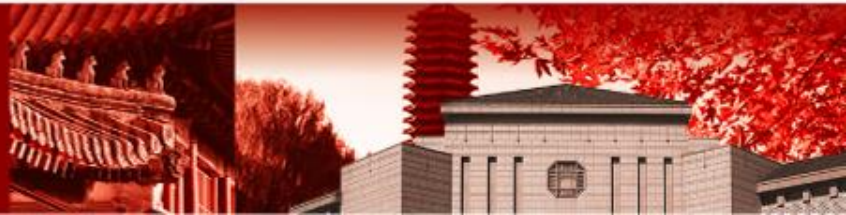
在s2列表逐个对比

找到，打勾

未找到，失败



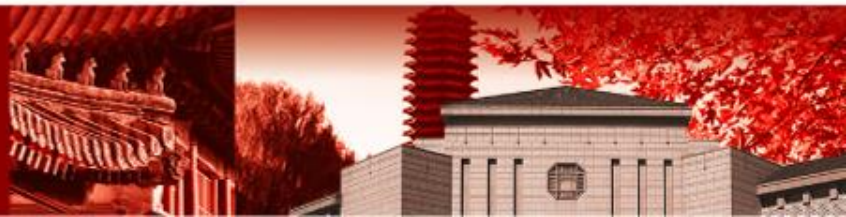
北京大学



解法1：逐字检查-算法分析

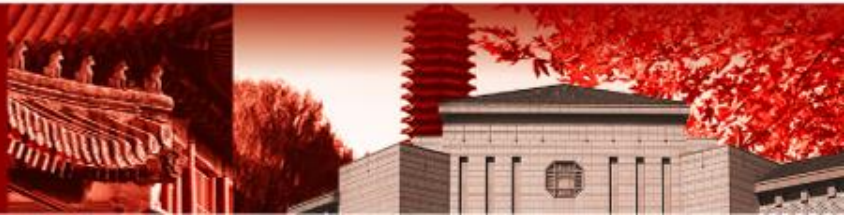
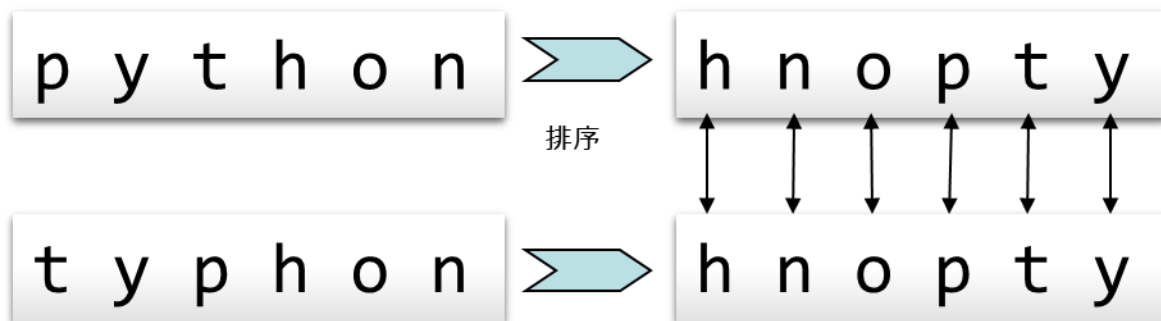
- 问题规模：词中包含的字符个数 n
- 主要部分在于两重循环
 - 外重循环要遍历字符串1每个字符，将内层循环执行 n 次
 - 而内重循环在字符串2中查找字符，每查找一个字符的操作次数，分别是1、2、3…… n 中的一个，而且各不相同
- 所以总执行次数是 $1+2+3+\cdots+n$
- 可知其数量级为 $O(n^2)$

$$\begin{aligned}\sum_{i=1}^n i &= \frac{n(n+1)}{2} \\ &= \frac{1}{2}n^2 + \frac{1}{2}n\end{aligned}$$



解法2：排序比较

- 解题思路为：将两个字符串都按照字母顺序排好序，再逐个字符对比是否相同，如果相同则是变位词



解法2：排序比较

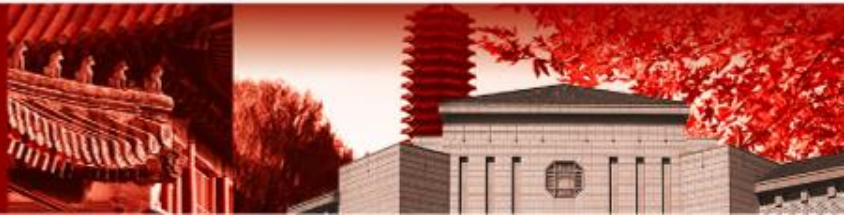
```
1  def anagramSolution2(s1, s2):
2      alist1 = list(s1)
3      alist2 = list(s2)
4
5      alist1.sort()
6      alist2.sort()
7      pos = 0
8      matches = True
9      while pos < len(s1) and matches:
10         if alist1[pos] == alist2[pos]:
11             pos = pos + 1
12         else:
13             matches = False
14     return matches
15
16
17 print(anagramSolution2('abcde', 'edcba'))
```

分别都排序

逐个对比



北京大学



解法2：排序比较-算法分析

- 粗看上去，本算法只有一个循环，最多执行 n 次，数量级应该是 $O(n)$
- 但循环前面的两个sort并不是无代价的
 - 如果查询下后面的章节，会发现排序算法采用不同的解决方案，其运行时间数量级差不多是 $O(n^2)$ 或者 $O(n \log n)$ ，大过循环的 $O(n)$
- 所以本算法中其决定性作用的步骤是排序步骤
- 本算法的运行时间数量级就等于排序过程的数量级 $O(n \log n)$

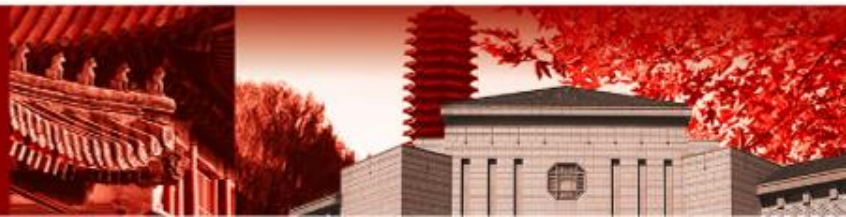


北京大学



解法3：暴力法

- 暴力法解题思路为：穷尽所有可能组合
- 对于变位词问题来说，暴力法具体是，将字符串1中出现的字母进行全排列，再查看字符串2是否出现在全排列列表中
- 这里最大的困难是产生字符串1所有字母的全排列，根据组合数学的结论，如果 n 个字符进行全排列，其所有可能的字符串个数为 $n!$ ，而 $n!$ 的增长速度甚至超过 2^n
 - 例如，对于20个字符长的词来说，将产生 $20! = 2,432,902,008,176,640,000$ 个候选词
 - 如果每秒钟处理一个候选词的话，需要77,146,816,596年（百亿）的时间来做完所有的匹配。
- 结论：暴力法恐怕不能算是个好算法



解法4：计数比较

- 最后一个算法解题思路为：对比两个字符串中每个字母出现的次数，如果26个字母出现的次数都相同的话，这两个字符串就一定是变位词
- 具体做法：为每个字符串设置一个26位的计数器，先检查每个字符串，在计数器中设定好每个字母出现的次数
- 计数完成后，进入比较阶段，看两个字符串的计数器是否相同，如果相同则输出是变位词的结论



北京大学



解法4：计数比较-程序代码

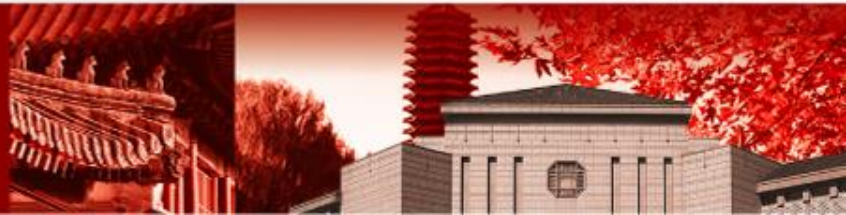
```
1 def anagramSolution4(s1, s2):
2     c1 = [0] * 26
3     c2 = [0] * 26
4     for i in range(len(s1)):
5         pos = ord(s1[i]) - ord('a')
6         c1[pos] = c1[pos] + 1
7     for i in range(len(s2)):
8         pos = ord(s2[i]) - ord('a')
9         c2[pos] = c2[pos] + 1
10    j = 0
11    stillOK = True
12    while j < 26 and stillOK:
13        if c1[j] == c2[j]:
14            j = j + 1
15        else:
16            stillOK = False
17    return stillOK
18
19
20 print(anagramSolution4('apple', 'pleap'))
```

分别都计数

计数器比较

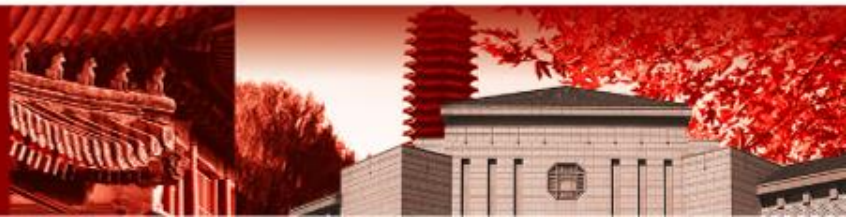


北京大学



解法4：计数比较-算法分析

- 计数比较算法中有3个循环，但不存在解法1中的嵌套循环
 - 前两个循环用于对字符串进行计数，操作次数等于字符串长度 n
 - 第3个循环用于计数器比较，操作次数总是26次
- 所以总操作次数 $T(n)=2n+26$ ，其数量级为 $O(n)$ ，这是一个线性数量级的算法，是4个变位词判断算法中性能最优的。
- 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
 - 由于仅限于26个字母构成的词，本算法对空间的需求并不明显，但如果考虑由大字符集构成的词（如中文具有上万不同字符），情况就会有所改变。
- 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。



随堂练习

- 判断下列代码段的大O级别

–A. $O(n)$ B. $O(n^2)$ C. $O(\log n)$ D. $O(n^3)$

- Q1:

```
test = 0
for i in range(n):
    for j in range(n):
        test = test + i * j
```

- Q2:

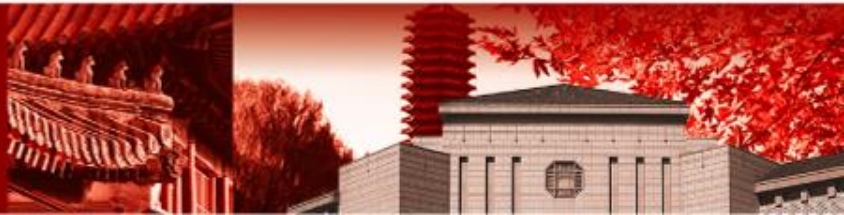
```
test = 0
for i in range(n):
    test = test + 1
for j in range(n):
    test = test - 1
```

- Q3:

```
i = n
while i > 0:
    k = 2 + 2
    i = i // 2
```



北京大学



随堂练习

- 判断下列代码段的大O级别

–A. $O(n)$ B. $O(n^2)$ C. $O(\log n)$ D. $O(n^3)$

• Q1:

B

```
test = 0
for i in range(n):
    for j in range(n):
        test = test + i * j
```

• Q2:

A

```
test = 0
for i in range(n):
    test = test + 1
for j in range(n):
    test = test - 1
```

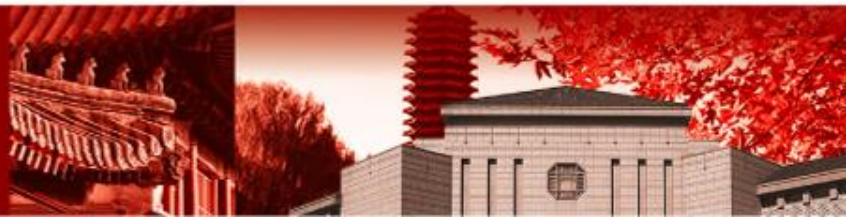
• Q3:

C

```
i = n
while i > 0:
    k = 2 + 2
    i = i // 2
```



北京大学

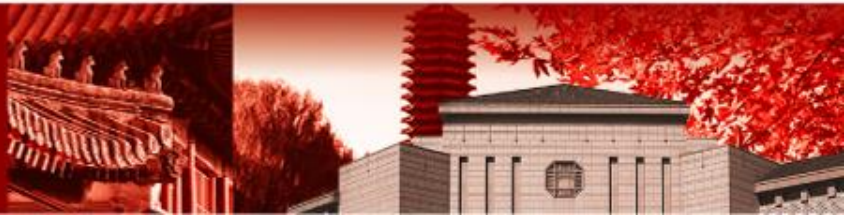


算法的空间复杂度

- 算法的空间复杂度是对算法在运行过程中占用额外存储空间大小的量度
- 空间复杂度算的是**变量的个数**，计算规则基本和时间复杂度类似，也使用大O渐进表示法
- 空间复杂度主要关注算法运行时所需的额外存储空间，包括以下两部分：
 - **固定空间**：是指算法运行时必须占用的存储空间，这部分空间的大小与输入数据的规模无关
 - 例如，程序中定义的常量、变量等。
 - **可变空间**：是指算法运行时根据输入规模动态分配的存储空间，这部分空间的大小会随着输入数据的规模而变化。
 - 例如，动态数组、链表、树等数据结构以及函数调用栈占用的存储空间。



北京大学



算法的空间复杂度

- 如果算法运行过程中占用的存储空间大小与输入规模无关，则称其空间复杂度为 $O(1)$ 。

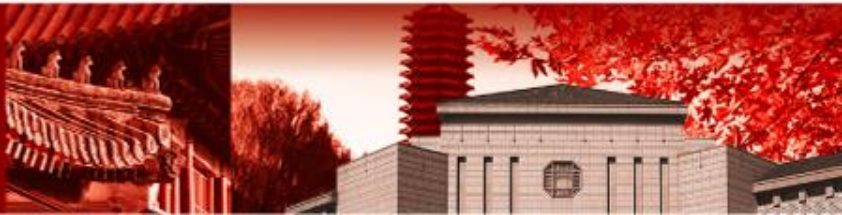
```
def add(a, b):  
    return a + b
```

- 如果算法运行过程中占用的存储空间与输入规模成正比，则称其空间复杂度为 $O(n)$ 。

```
def sum_array(arr):  
    n = len(arr) # 输入数组的长度  
    partial_sums = [0] * n # 创建一个与输入数组等长的数组存储部分和  
  
    for i in range(n):  
        partial_sums[i] = arr[i] + (partial_sums[i - 1] if i > 0  
else 0)  
  
    return partial_sums[-1] # 返回最终的总和
```



北京大学



算法的空间复杂度

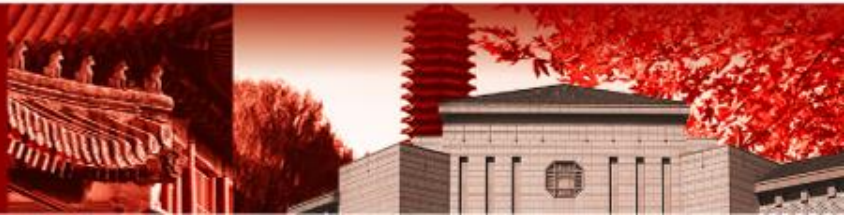
- 如果算法运行过程中占用的存储空间与输入规模的平方成正比，则称其空间复杂度为 $O(n^2)$ 。

```
def create_matrix(n):  
    matrix = [[0] * n for _ in range(n)] # 创建一个  $n \times n$  的二维矩阵  
  
    for i in range(n):  
        for j in range(n):  
            matrix[i][j] = i + j # 填充矩阵元素  
  
    return matrix
```

- 如果算法运行过程中占用的存储空间与输入规模的对数成正比，则称其空间复杂度为 $O(\log n)$ 。
- 如果算法运行过程中占用的存储空间与输入规模的指数成正比，则称其空间复杂度为 $O(2^n)$ 。



北京大学



算法的空间复杂度

- 区分时间复杂度与空间复杂度

- 时间复杂度为 $O(\log n)$

- 空间复杂度为 $O(1)$

```
def power_of_two(n):  
    result = 1  
    while n > 0:  
        if n % 2 == 1:  
            result *= 2  
        n //= 2  
    return result
```



北京大学

