计算思维通识教育

Computational Thinking

第4章 用计算思维求解问题

主讲人：曹轶臻

联系方式：caoyizhen@cuc.edu.cn

Programming is all about data structures and algorithms. Data structures are used to hold data while algorithms are **used to solve the problem** using that data.

CONTENTS

01

计算机求解复杂问题

问题求解的一般方法 1-1

算法是什么 1-2

什么是计算思维？

- 思维过程
- 简单问题 vs 复杂问题
- 得出最终解决方案：提供给人或者计算机来使用

计算思维能力≠计算机程序设计能力

- 前者让我们能够找到一种合适的方法来如何告诉计算机需要做什么以及如何做
- 后者通过程序设计语言告诉计算机去做什么以及如何做
- 前者重“规划”，后者重“执行”

1-1 问题求解的一般方法

计算思维的主要组成技术

问题分解

通过将复杂问题转换成为若干较为简单的子问题，从而降低该问题本身的认知难度，最终实现对其求解。

模式识别

通过对分解后形成的子问题进行分析，找到其主要特征，识别该子问题与其他问题的相似之处，以便进一步理解该子问题。

问题抽象

在对子问题进行分析后可以得到许多信息，我们需要从中提炼出那些有助于解决子问题的信息，同时剔除那些非重点的无用信息。

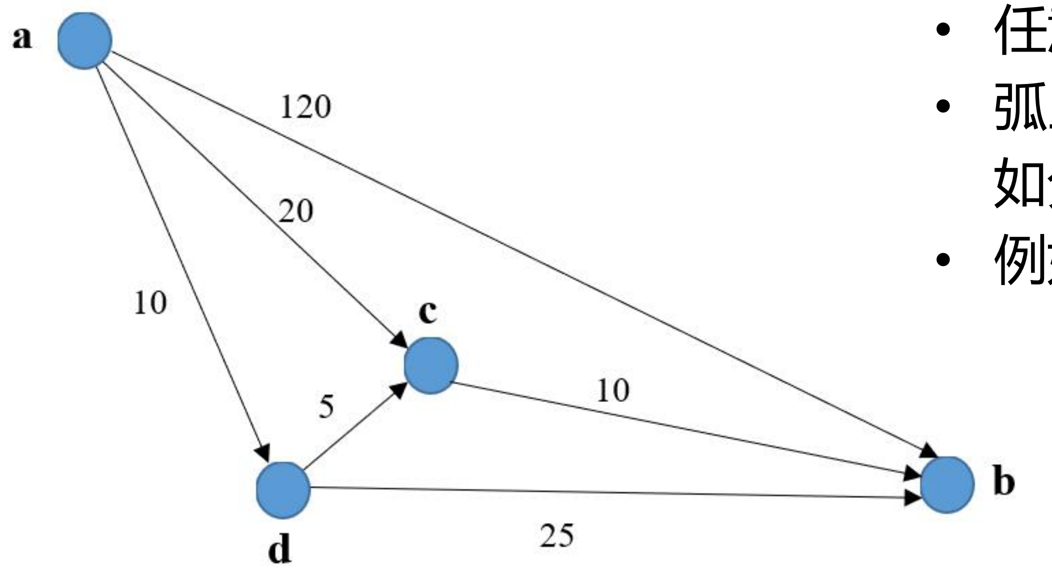
算法设计

描述原始问题的整体解决方案，这种方案通常是以包含简单操作的解决步骤组成。

1-1 问题求解的一般方法

复杂问题求解举例

假设我们需要计算从地点a到达地点b之间的[最短]路径



- 任意两点可能之间存在带有方向（箭头）的连边，称为弧
- 弧上的数字表示途经对应的实际道路所需要花费的时间（比如分钟），我们将其定义为该弧的路径长度
- 例如弧(a, b)的路径长度为120分钟（堵车？路窄？收费站？）

- 然而，路径（a, c, b）仅需费时30分钟

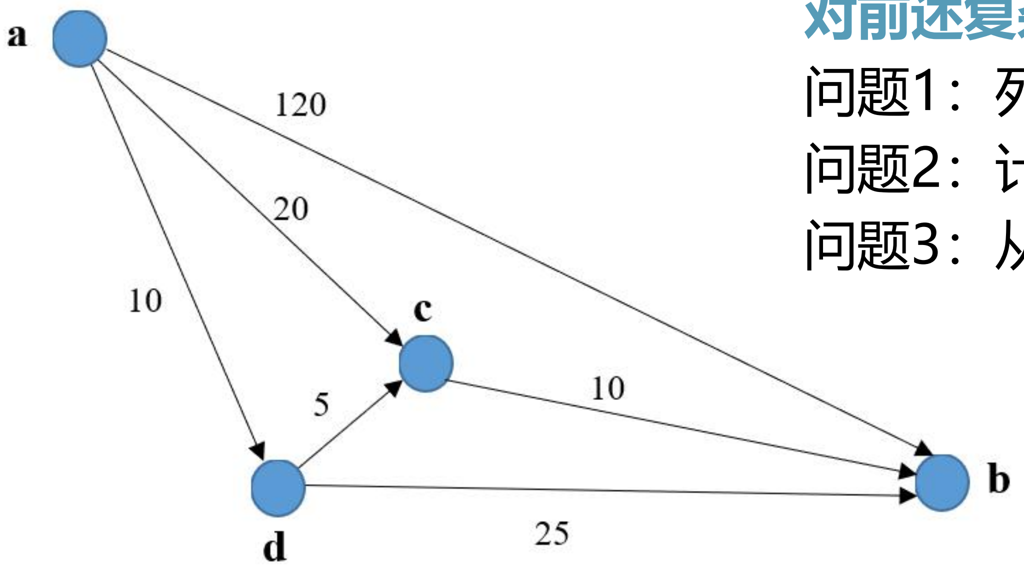
- 不禁思考：还有更快路线吗?

找到a→b的路径（简单问题） → 找到a→b的最短路径（复杂问题）

计算机与网络空间安全学院
School of Computer and Cyber Sciences

复杂问题求解举例

假设我们需要计算从地点a到达地点b之间的最短路径



对前述复杂问题进行分解：

问题1：列出点a到b之间的所有路径（路径上可能包括点c和/或d）

问题2：计算采用每条路径的总长度

问题3：从中选出最短的一条即可满足需要

一个复杂的问题转换成了三个相对更为明确的、且更容易解决的问题

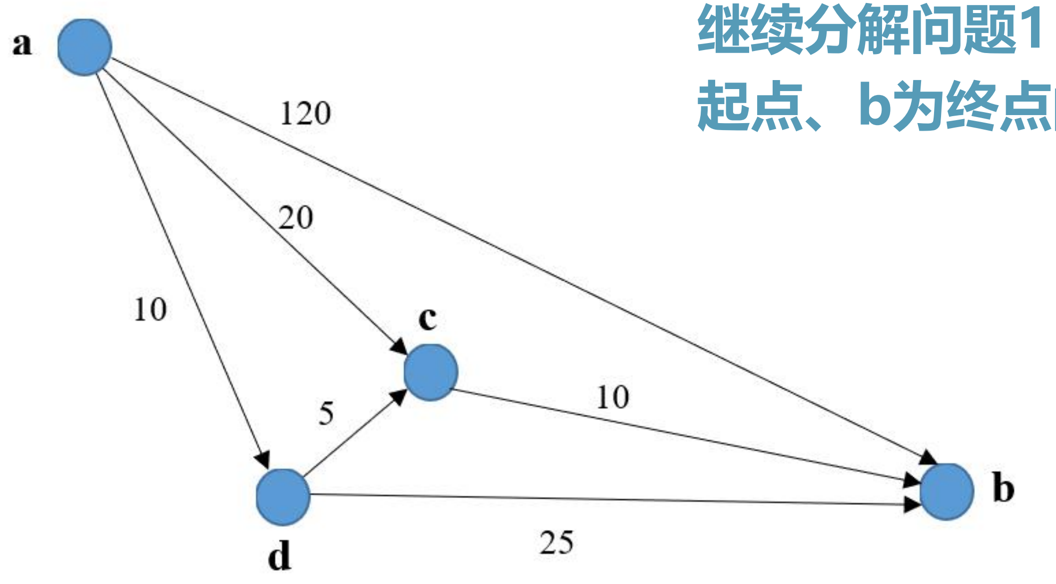
相比之下，问题1似乎仍然有些复杂

计算机与网络空间安全学院
School of Computer and Cyber Sciences

复杂问题求解举例

假设我们需要计算从地点a到达地点b之间的最短路径

继续分解问题1：按照a和b之间经过的中间点的数目来列举所有以a为起点、b为终点的路径



问题1.1：列出a和b之间有0个中间点的所有路径，这时只有（a, b）

问题1.2：列出a和b之间有1个中间点的所有路径，得到（a, c, b）和（a, d, b）

问题1.3：列出a和b之间有2个中间点的所有路径，得到（a, d, c, b）

一共产生了4条路径，分别是（a, b）、（a, c, b）、（a, d, b）和**(a, d, c, b)**，最后一个为最短路径

1-1 问题求解的一般方法

复杂问题求解举例

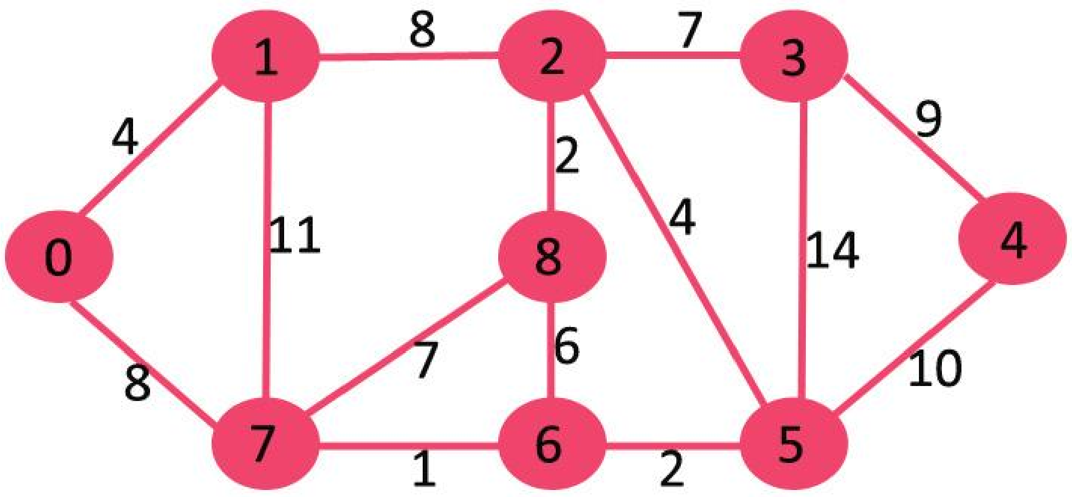
假设我们需要计算从地点a到达地点b之间的**最短路径**

前图展示的是较为简单的路况，实际生活中起点a和终点b之间会存在更多个中间点，经由这些中间点可以产生更多的备选路径。

如果采用前面的方法，需要**列举出所有可能产生的路径**，即便是按照中间点的数目的升序来逐一列出也并非易事：

- 步骤将非常繁琐，还有可能遗漏部分备选路径，且整个过程的实际运行效率不高。
- 需要寻找其他更优的方法！

- 对于这样的“复杂问题”，我们可以尝试换一种思路来解决产生最短路径的问题，即利用著名的Dijkstra算法（迪杰斯特拉算法）。

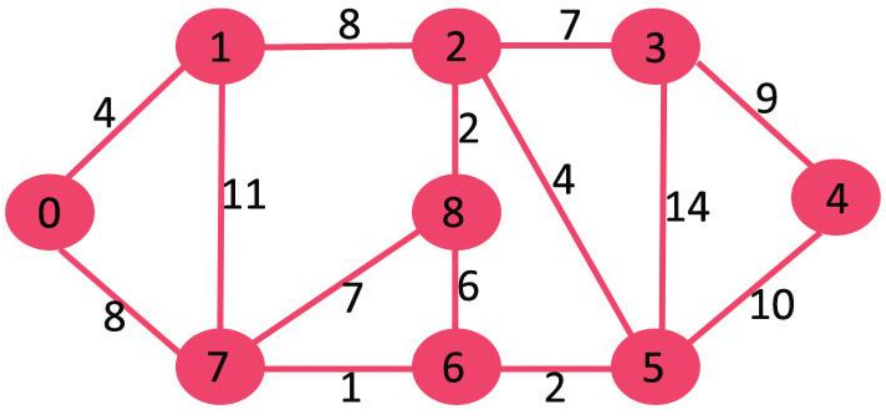


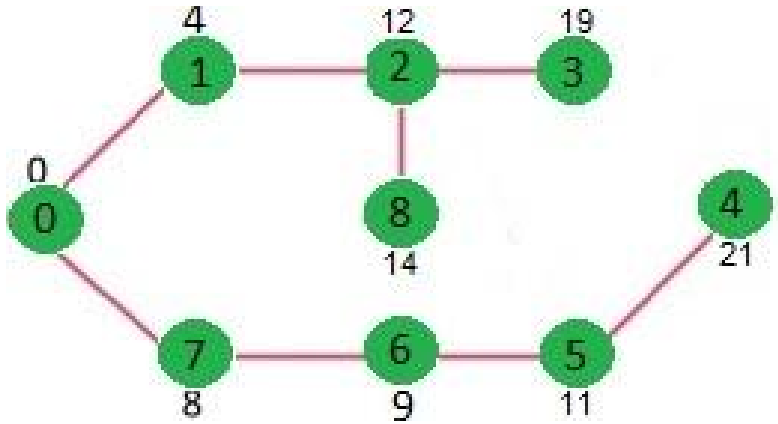
- Dijkstra算法用于求解上图中任意两点之间的最短路径，在这里我们将“最短路径”定义为“途经时间最少的最短路径”

Dijkstra算法简介

思想：由短到长逐步产生从起始点到其余各点的最短路径。（以当前已找到最短路径的节点为基础，逐步更新到其他节点的最短距离，直至找到起始节点到其他节点的最短路径）

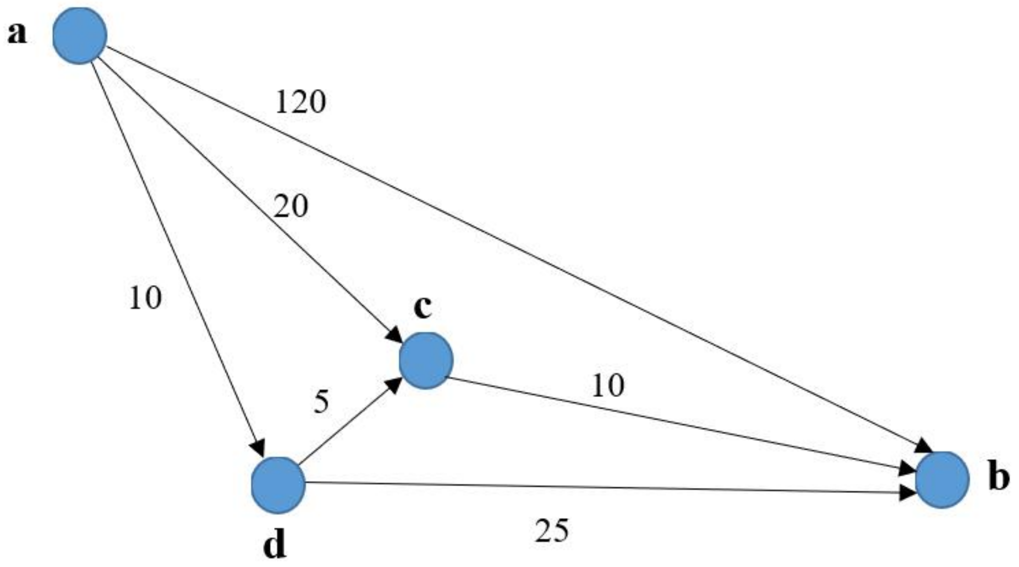
从0到1的距离 = 4
从0到2的最小距离=12，路径：0->1->2
从0到3的最小距离=19，路径：0->1->2->3
从0到4的最小距离=21，路径：0->7->6->5->4
从0到5的最小距离=11，路径：0->7->6->5
从0到6的最小距离= 9，路径：0->7->6
从0到7的最小距离= 8，路径：0->7
从0到8的最小距离=14，路径：0->1->2->8





1-1 问题求解的一般方法

Dijkstra算法的应用



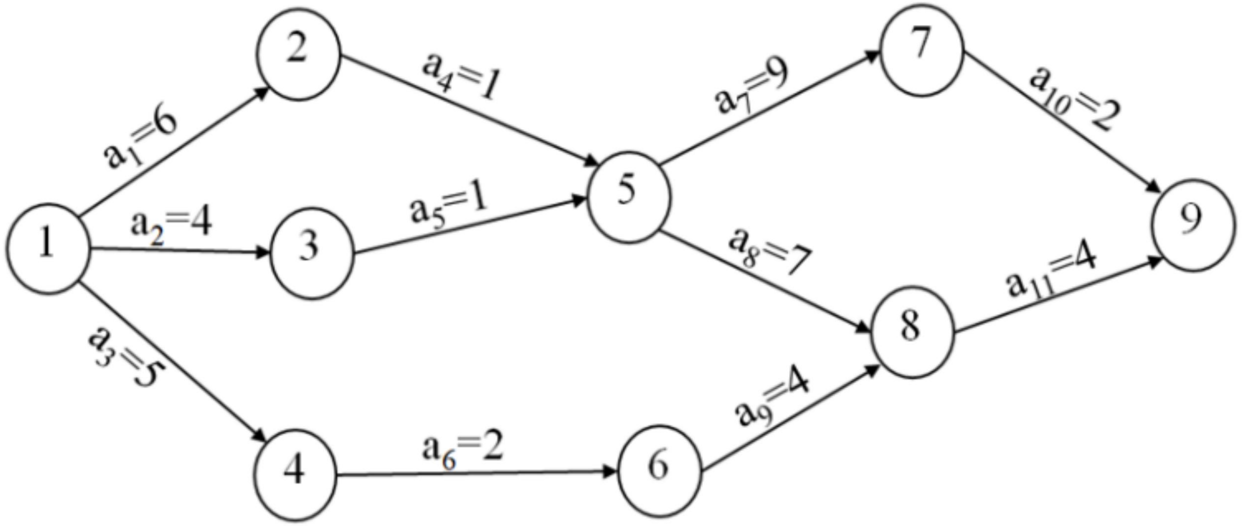
Dijkstra算法的运行过程中其实产生的不仅仅是图中点a到b之间的最短路径，同时还可以识别出点a到其余各点的最短路径

不仅如此，当我们调整起始点甚至可以产生图中任意两点之间的最短路径

该算法还可以用于解决物流公司的配送业务路径规划问题，从而降低配送成本和配送商品价格

Dijkstra算法的应用

将配送员可以途经的实际地点当作图形中的点（图中的①、②、…⑨），用弧连接这些相互之间可以直接到达（即不经过第三点）的点，并在弧上标记配送员从一个地点到达另外一个地点所需时间



配送路径规划问题就转换成为从图中找到一条路径，这条路径的特点是从配送起始点开始、直到到达配送目标点为止，且整个路径长度最短

1-1 问题求解的一般方法

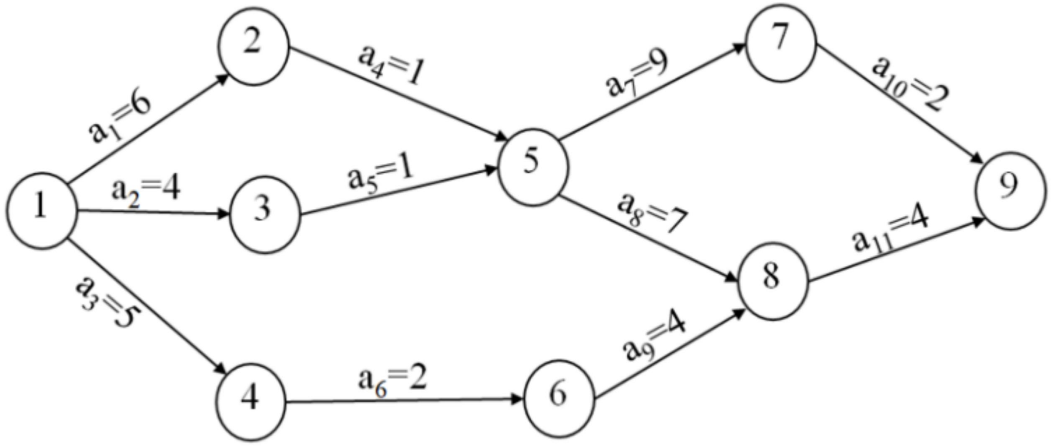
我们把求取两个地点间的最短路径的复杂问题抽象为图问题，并利用Dijkstra算法即可得到答案。

抽象要求我们关注待解决问题的重点概念和关键细节

例如在分析过程中，我们重点处理的对象包括：

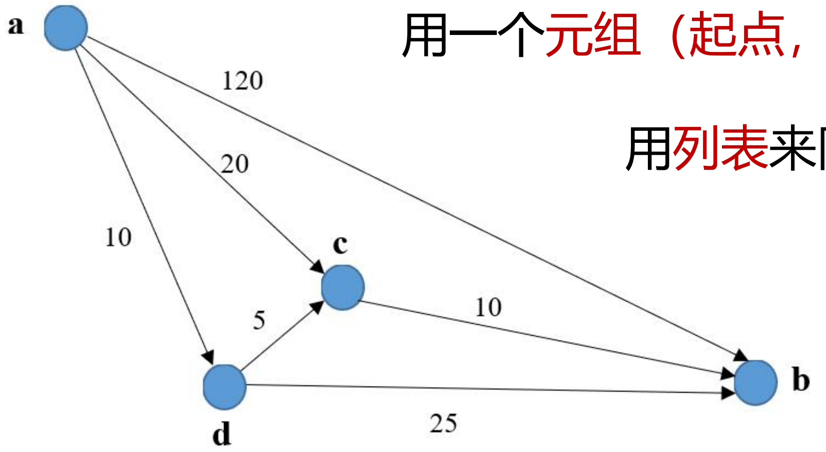
- 每对地点之间的路径，
- 途经该路径需要花费的时长

而不需要每个地点的具体地理位置。



1-1 问题求解的一般方法

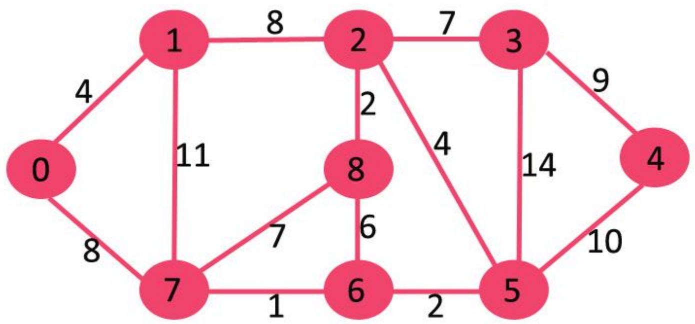
用数据结构表示重点信息



用一个元组（起点，终点，路径长度）表示路径，例如（'a', 'b', 120)

用列表来同时存放多条路径，arc_lst = [('a', 'b', 120), ('a', 'c', 20), ('a', 'd', 10)]

用字典表示一条路径所有途经的中间地点，
quickPathSet = {'ab':[25, ['a', 'd', 'c', 'b']]}



用二维数组表示图

```
g = Graph(9) #创建九个顶点的图对象
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
           [4, 0, 8, 0, 0, 0, 0, 11, 0],
           [0, 8, 0, 7, 0, 4, 0, 0, 2],
           [0, 0, 7, 0, 9, 14, 0, 0, 0],
           [0, 0, 0, 9, 0, 10, 0, 0, 0],
           [0, 0, 4, 14, 10, 0, 2, 0, 0],
           [0, 0, 0, 0, 0, 2, 0, 1, 6],
           [8, 11, 0, 0, 0, 0, 1, 0, 7],
           [0, 0, 2, 0, 0, 0, 6, 7, 0]
           ]

g.dijkstra(0) #调用dijkstra算法
```

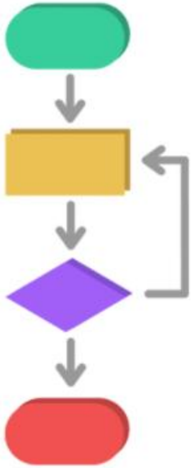
用数据结构表示重点信息

任何程序设计中都离不开对数据结构的使用，这些数据结构可以是简单的整数或字符串，也可以是复杂的多维数组、多层嵌套列表或者队列、树、图等。

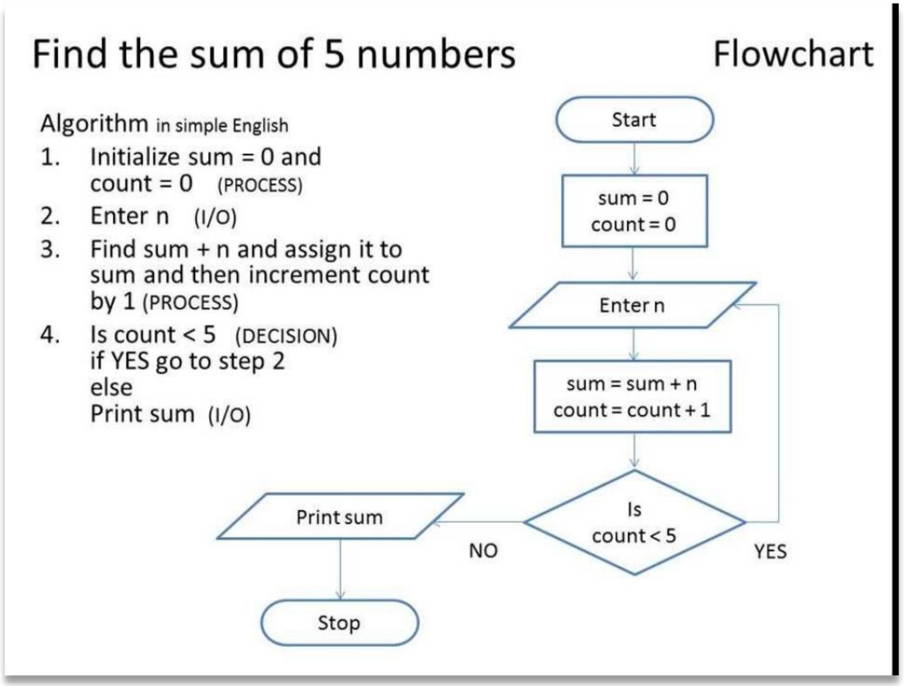
数据结构是计算机程序设计中一种常见的抽象工具，帮助我们聚焦问题的重点，降低解决问题的难度，并促成问题得以在计算机上被解决。

一旦我们把问题中关注的对象用合适的数据结构表示出来，结合分析得到的解决方案（算法），就能够顺利形成计算机可以理解并执行的解题步骤（程序）了

1-2 什么是算法

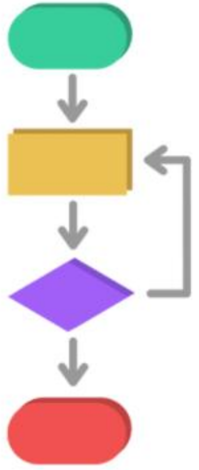


利用计算思维的最终目的是产生对复杂问题的解决方案，这样的方案是解决特定问题的步骤的集合，必须以某种容易理解的方式给出，例如我们可以使用自然语言/流程图/伪代码列出其具体步骤，这种描述就是算法（Algorithm）。



实际上，
我们已经使用Python写过一些算法了

1-2 什么是算法



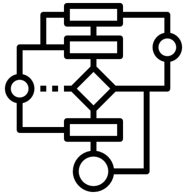
学习数据结构和算法使我们能够编写高效和优化的计算机程序。

好算法应具有的品质

- 应准确定义输入和输出
- 算法中的每一步都应该清晰明确
- 算法不应包含计算机代码。相反，算法应该以可以在不同编程语言中使用的方式编写
- 在解决问题的许多不同方法中，算法应该是最有效的

算法示例：找到三个数中最大的数

```
第 1 步：开始
第 2 步：声明变量 a、b 和 c
第 3 步：读取变量 a、b 和 c
第 4 步：如果 a > b
           如果 a > c
              显示 a 是最大的数字
           否则
              显示c是最大的数
        否则
           如果 b > c
              显示 b 是最大的数
           否则
              显示 c 是最大的数
第 5 步：停止
```



通俗地讲，算法只不过是解决问题的步骤。它们本质上是一种解决方案。

解决阶乘问题的算法可能如下所示

例如问题：计算 n 的阶乘（factorial ）

```
Initialize fact = 1
For every value v in range 1 to n:
    Multiply the fact by v
fact contains the factorial of n
```

上面的算法是用伪代码和自然语言（此例为英文）写的

```python
def factorial(n):
    fact = 1
    for i in range(1, n + 1):
        fact = fact * i
    return fact
```

如果它是用编程语言编写的，我们会将其称为代码。（此例为Python语言）

1-2 什么是算法

使用数据结构和算法使代码可扩展

尝试解决一个简单的问题：

找到1到10^{11}之间的自然数总和

很容易快速给出算法和代码

```
Initialize sum = 0
for every natural number n in range 1 to 10^11(inclusive):
    add n to sum
sum is your answer
```

```python
def find_sum(n):
    sum = 0
    for i in range(1, n + 1):
        sum += i
    return sum

print(find_sum(int(1e11)))
```

算法和代码都没有写错

然而，代码run了很久，也没有得到结果...

使用数据结构和算法使代码可扩展

计算机程序最宝贵的两种资源，其一是时间。

运行代码的时间 = 指令数 * 执行每条指令的时间

```
def find_sum(n):
    sum = 0
    for i in range(1, n + 1):
        sum += i
    return sum

print(find_sum(int(1e11)))
```

分析一下，上面find_sum函数里执行的指令总数记为x，

则 $x = 1 + 10^{11} + 1 = 10^{11} + 2$

我们假设一台计算机可以在一秒钟内执行 $y = 10^7$ 条指令（它可能因机器配置而异）。

运行y条指令的时间 = 1 秒

运行1条指令的时间 = 1 / y 秒

运行x条指令的时间 = x * (1 / y) 秒

因此可得

这段代码运行的时间 = $x / y = (10^{11} + 2) / 10^7$

大约需要167分钟，
这是不能接受的！

1-2 什么是算法

使用数据结构和算法使代码可扩展

计算机程序最宝贵的两种资源，其一是时间。
运行代码的时间 = 指令数 * 执行每条指令的时间

前 N 个自然数的和可由以下公式给出：

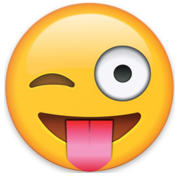
```
Sum = N * (N + 1) / 2
```

- 指令的数量取决于你使用的代码，执行每段代码所花费的时间取决于你的机器和编译器。

将前面的代码转换为：

```
def find_sum(n):
    return n * (n + 1) / 2
```

- 此代码仅在一条指令中执行，并且无论n的值是什么都可以马上完成任务。
- 即使让n大于宇宙中的原子总数，它也会立即计算得到结果。



使用数据结构和算法使代码可扩展

尝试解决一个简单的问题：

找到1到10^{11}之间的自然数总和

第一个算法是不可扩展的，这是因为它的时间花销需要随着问题规模的增长而线性增长。

第二个解决方案具有很强的可扩展性，不需要花费更多时间来解决更大的问题。这被称为恒定时间算法。

计算机程序最宝贵的两种资源，

其一是时间

其二是内存空间。

- 内存很宝贵，内存并不总是充足可用。
- 在处理需要您存储或生成大量数据的代码/系统时，尽可能节省内存使用对您的算法至关重要。例如：
  - 在存储关于人的数据时，可以仅存储出生日期而不再存储年龄来节省内存。因为始终可以使用出生日期和当前日期即时计算年龄。
  - 使用邻接表存储大规模的稀疏图，而不是邻接矩阵。

1-2 什么是算法

从计算思维的角度考虑可扩展性，就是规模加上处理能力，这意味着解决更大规模问题时，算法/解决方案的性能质量表现如何。

例如，给上课提供一个解决方案，考虑上课的规模

- 30人上课，需要小教室+黑板+粉笔就可以解决问题
- 100人上课，需要大教室、投影、数字笔
- 200人上课，需要大教室、多个显示器/投影、麦克风、扬声器
- 1000人上课，需要一个剧场，多个屏幕、扩声系统
- 10000人上课，需要专业线上会议系统/教学系统、高带宽网络、服务器集群的支持

本次课程结束，请继续加油！

Computational Thinking

计算机与网络空间安全学院
School of Computer and Cyber Sciences