



Data Structures

# 绪论 Introduction

2023年8月31日

学而不厌 诲人不倦

- ➡ **1.1** 问题的求解与程序设计
- ➡ **1.2** 数据结构的基本概念
- ➡ **1.3** 算法的基本概念
- ➡ **1.4** 算法分析

## 1.3 算法的基本概念

### 1-3-1 算法及算法的特性



算法的起源



算法的定义



算法的特性



好算法的特性

[illegible]

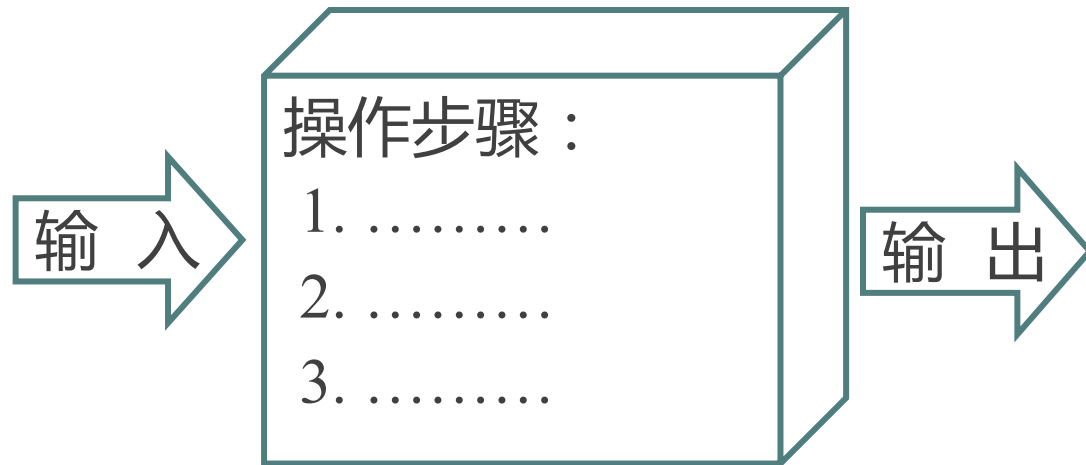
20

## 📌 图灵奖与算法

- 🏆 Hoare在26岁发明了闻名于世的快速排序算法；
- 🏆 Ronald、Shamir和Adleman发明了国际上最具影响力的公钥密码算法RSA；
- 🏆 Knuth编著的《程序设计的艺术》奠定了数据结构与算法领域的主要内容；
- 🏆 Floyd发明了求解多源点最短路径的 Floyd算法以及堆结构；
- 🏆 Karp在网络流和组合优化问题领域都发明了许多高效算法；
- 🏆 Hopcroft和他的学生Tarjan在数据结构和算法方面有众多创造性贡献；
- 🏆 姚期智 ( Chi-Chih Yao ) 发明了伪随机数的生成算法以及加密/解密算法；
- 🏆 Sutherland发明的图形图像算法改善了屏幕刷新的文件显示；
- 🏆 Dijkstra发明了单源点的最短路径算法Dijkstra算法；
- 🏆 Wilkinson在数值线性代数方面发现很多有意义的算法；
- 🏆 Blum发现了著名的算法设计技术——分支限界法.....

# 算法的定义

✦ 算法：是对**特定问题**求解步骤的一种描述，是**指令**的有限**序列**



✦ 木须柿子的做法：

1. 柿子切块，鸡蛋加适量盐搅拌
2. 锅里放油
3. 把鸡蛋倒进去炒熟
4. 加入葱花
5. 把柿子放进去放少许盐和味精
6. 翻炒几下出锅装盘



算法不是问题的答案，而是解决问题的**操作步骤**

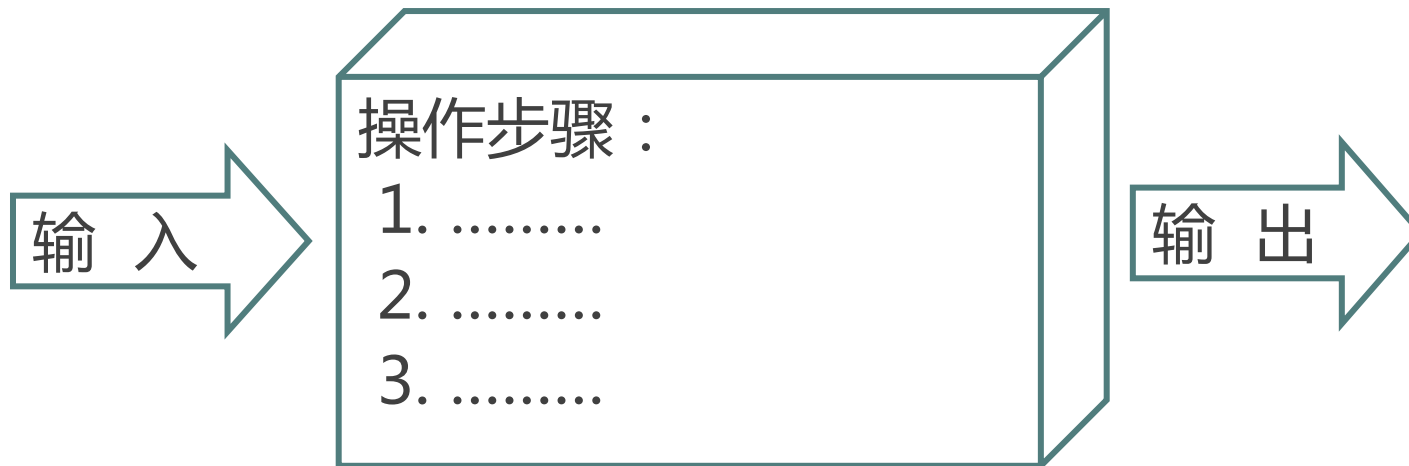
# 算法的特性

 算法的操作步骤应该满足什么要求？

(1) **有穷性**：总是在执行有穷步之后结束，且每一步都在有穷时间内完成

 每一条指令都不是无限循环！

 有穷不是数学意义上的概念！



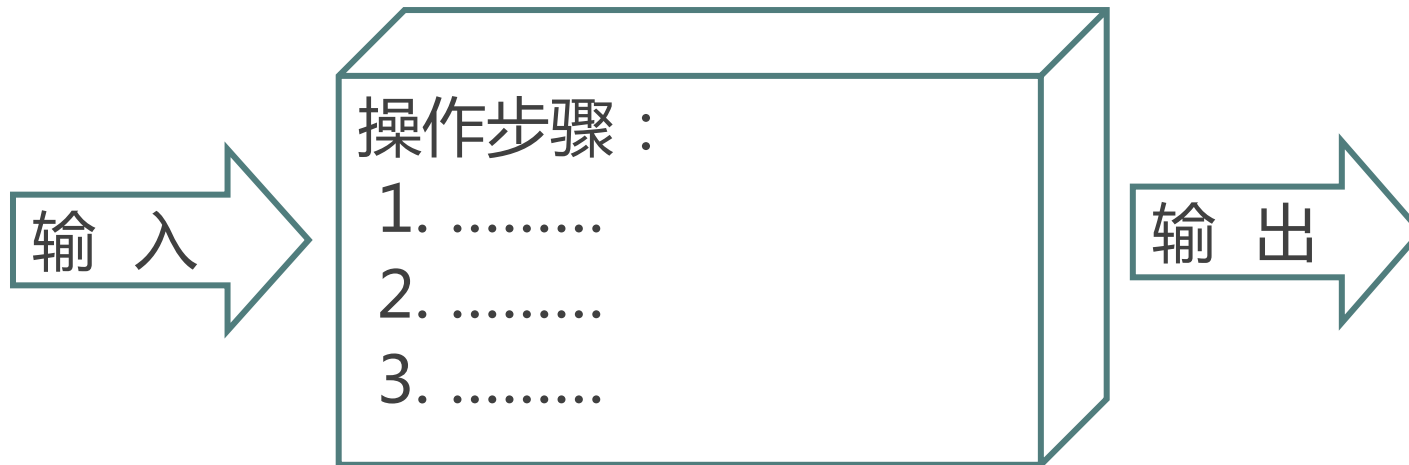
# 算法的特性

🕒 算法的操作步骤应该满足什么要求？

- (1) 有穷性：总是在执行有穷步之后结束，且每一步都在有穷时间内完成
- (2) 确定性：每一条指令必须有确切的含义，相同的输入得到相同的输出



上下文无关





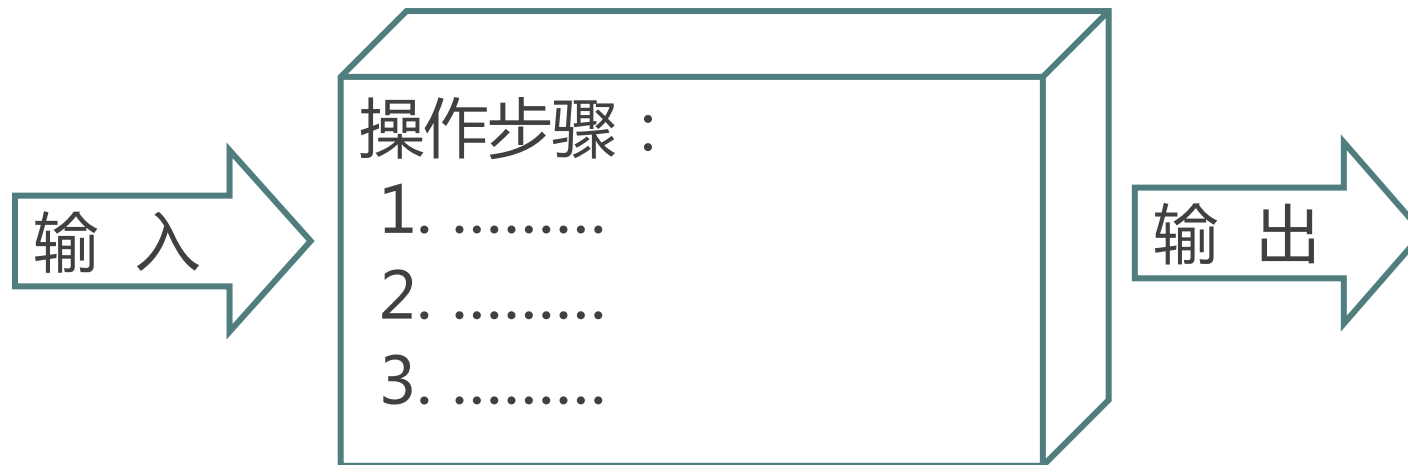
# 算法的特性

 算法的操作步骤应该满足什么要求？

- (1) 有穷性：总是在执行有穷步之后结束，且每一步都在有穷时间内完成
- (2) 确定性：每一条指令必须有确切的含义，相同的输入得到相同的输出
- (3) 可行性：操作步骤可以通过已经实现的基本操作执行有限次来实现



机器可执行



# 算法的特性

**例 1** 设计算法求两个自然数的最大公约数。

**【想法】** 将这两个自然数分别进行质因数分解，然后找出所有公因子并将这些公因子相乘。例如， $48=2\times 2\times 2\times 2\times 3$ ， $36=2\times 2\times 3\times 3$ ，公因子有2、2、3，因此，48和36的最大公约数为 $2\times 2\times 3=12$ 。

**【算法】** 设两个自然数是 $m$ 和 $n$ ，算法如下：

步骤 1：找出  $m$  的所有质因子

步骤 2：找出  $n$  的所有质因子

步骤 3：从第 1 步和第 2 步得到的质因子中找出所有公因子

步骤 4：将找到的所有公因子相乘，结果即为  $m$  和  $n$  的最大公约数

不满足确定性、有穷性！



满足算法的特性吗？



如何找出所有质因子？如何找出所有公因子？



质因数分解尚未找到多项式时间算法

# 好算法的特性

 一个算法满足什么特性才能称之为好算法呢？

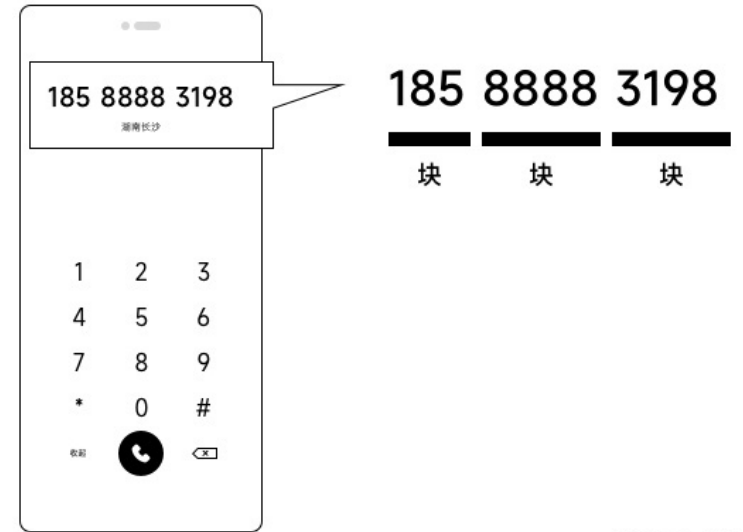
(1) **正确性**：算法能满足具体问题的需求，即对于**任何合法**的输入，算法都会得出正确的结果。

(2) **健壮性**：算法对**非法输入**的抵抗能力，即对于错误的输入，算法应能识别并做出处理，而不是产生错误动作或陷入瘫痪。

(3) **可理解性**：算法容易理解和实现。

(4) **抽象分级**：用合适的抽象分级来组织表达算法的思想，启发式规则 $7 \pm 2$ 。

(5) **高效性**：具有较短的执行时间并占用较少的辅助空间。



米勒原则：人类的短期记忆能力一般限于一次记忆 **5 ~ 9** 个对象

## 1.3 算法的基本概念

### 1-3-2 算法的描述方法

# 欧几里得算法

✦ 辗转相除求两个自然数的最大公约数（古希腊（公元前300年））

**【想法——基本思路】** 设两个自然数为  $m$  和  $n$ ，欧几里德算法的基本思想是将  $m$  和  $n$  辗转相除直到余数为 0

$m$	$n$	$r$
35	25	10
25	10	5
10	5	0

## 【算法——自然语言描述】

步骤1：将  $m$  除以  $n$  得到余数  $r$ ；

步骤2：若  $r$  等于0，则  $n$  为最大公约数，算法结束；否则执行步骤 3；

步骤3：将  $n$  的值放在  $m$  中，将  $r$  的值放在  $n$  中，重新执行步骤 1；

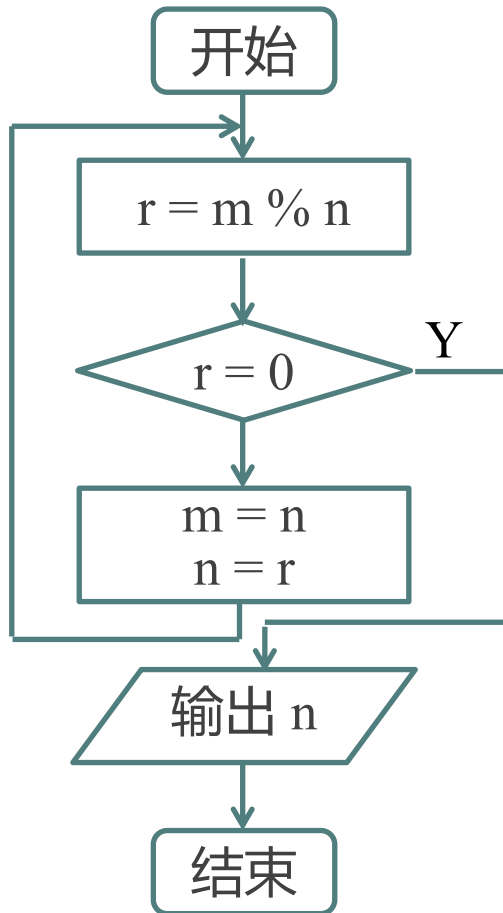
优点：容易理解；缺点：冗长、二义性

使用方法：粗线条描述**算法思想**；注意事项：避免写成自然段

# 流程图描述算法

✦ 辗转相除求两个自然数的最大公约数（古希腊（公元前300年））

## 【算法——流程图描述】



## 【算法——程序语言描述】

```
1  #include <stdio.h>
2  int ComFactor(int m, int n)
3  {
4      int r = m % n;
5      while (r != 0)
6      {
7          m = n;  n = r;
8          r = m % n;
9      }
10     return n;
11 }
12 int main( )
13 {
14     int x = ComFactor(35, 25);
15     printf("最大公约数是: %d\n", x);
16     return 0;
17 }
```

# 伪代码描述算法

优点：表达能力强，抽象性强，容易理解，容易实现

- ✦ 伪代码：介于自然语言和程序设计语言之间的方法，它采用某一程序设计语言的**基本语法**，操作指令**可以结合**自然语言来设计。
- ✦ 伪代码被称为“算法语言”或“第一语言”

## 【算法——伪代码描述】

输入：两个自然数  $m$  和  $n$   
输出： $m$  和  $n$  的最大公约数

1.  $r = m \% n$ ;
2. 循环直到  $r$  等于0
  - 2.1  $m = n$ ;
  - 2.2  $n = r$ ;
  - 2.3  $r = m \% n$ ;
3. 输出  $n$ ;

只描述子函数；省略主函数和头函数

## 【算法——程序语言描述】

```
1  #include <stdio.h>
2  int ComFactor(int m, int n)
3  {
4      int r = m % n;
5      while (r != 0)
6      {
7          m = n; n = r;
8          r = m % n;
9      }
10     return n;
11 }
12 int main( )
13 {
14     int x = ComFactor(35, 25);
15     printf("最大公约数是: %d\n", x);
16     return 0;
17 }
```

## 1.4 算法分析

### 1-4-1 算法的时间复杂度



度量算法效率的方法



算法的时间复杂度



# 算法分析

算法设计：面对一个问题，如何设计一个有效的算法

指导改进

检验评估

算法分析：对已设计的算法，如何评价或判断其优劣

 易读性

 可维护性

 健壮（容错）性


 可扩展性

 .....



效率（速度）  
算法的核心和灵魂

# 度量算法效率


 如何度量算法的效率呢？

 事后统计（定量分析）：将算法实现，测算其时间和空间开销

缺点：（1）编写程序实现算法将花费较多的时间和精力

（2）所得实验结果依赖于计算机的软硬件等环境因素

 事前分析（定性分析）：对算法所消耗资源的一种估算方法

 { 时间  
空间

 对估算方法有什么要求呢？

能够刻画效率；与语言环境无关；具有一般性.....

# 时间复杂度

每条语句**执行次数**之和 = 算法的**执行时间** = 每条语句执行时间之和



**基本语句**的执行次数

```
for (i = 1; i <= n; i++)  
    for (j = 1; j <= n; j++)  
        x++;
```



单位时间



执行次数 × 执行一次的时间






指令系统、编译的代码质量

- ✦ **基本语句**：执行次数与整个算法的执行次数成正比的操作指令
- ✦ **问题规模**：输入量的多少

# 时间复杂度

算法的运行时间 = 基本语句的执行次数

 如何计算算法中基本语句的执行次数呢？

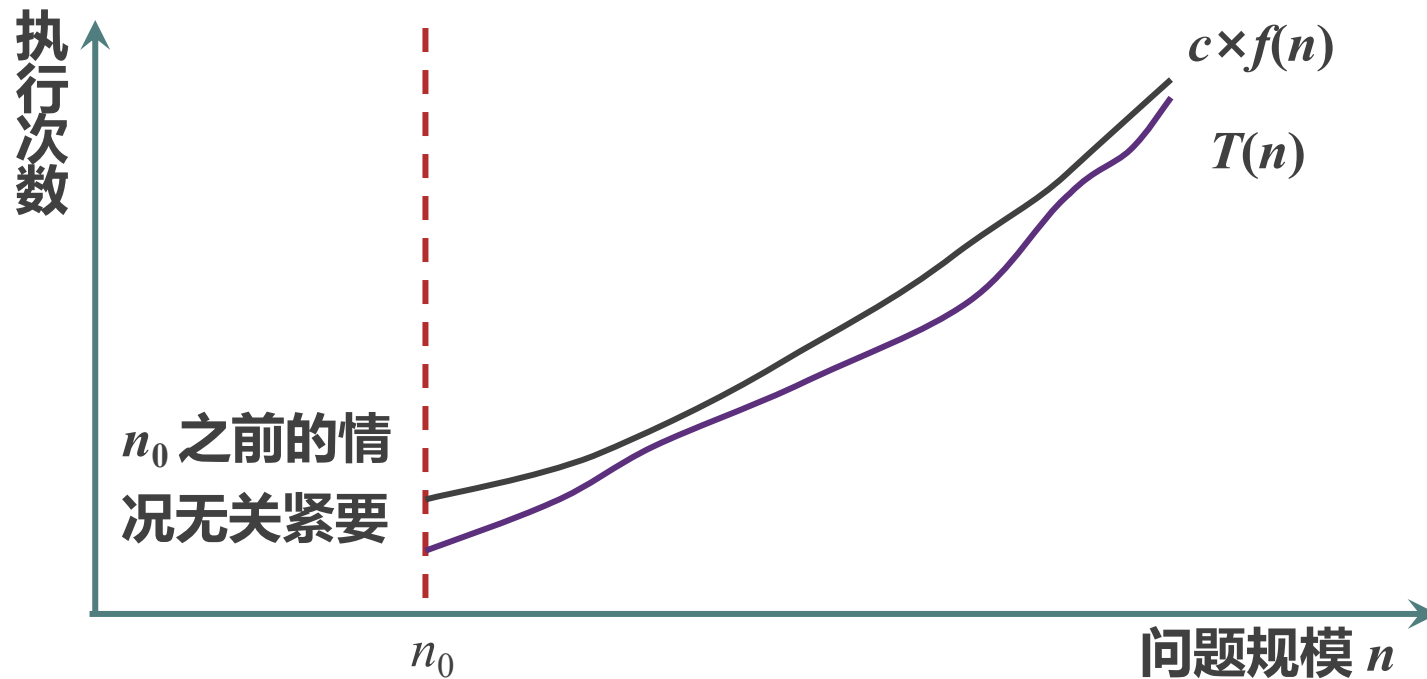
-  注意到，几乎所有算法，对于规模更大的输入需要运行更长的时间
-  运行算法所需要的时间  $T$  是问题规模  $n$  的函数，记作  $T(n)$
-  问题规模可以是多个变量  $\rightarrow$  多元函数

 如何表示算法的运行时间函数呢？

 **时间复杂度**：当问题规模充分大时，算法中基本语句的执行次数在渐近意义下的阶——关注的是**增长趋势**

# 大O 记号

**定义1-1** 若存在两个正的常数  $c$  和  $n_0$  , 对于任意  $n \geq n_0$  , 都有  $T(n) \leq c \times f(n)$  , 则称  $T(n) = O(f(n))$ 。



$T(n)$ 和 $f(n)$ 具有相同的增长趋势 ,  $T(n)$ 的增长至多趋同于函数 $f(n)$ 的增长

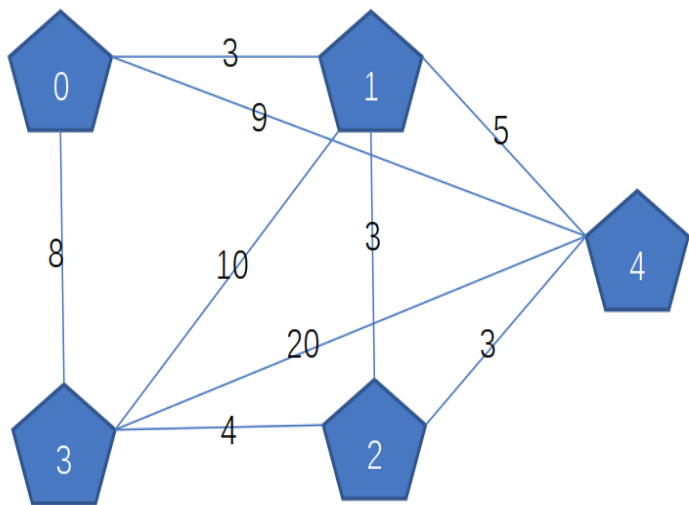
# 大O 记号

✦ 时间复杂度是在不同数量级的层面上比较算法，是一种估算技术

$$O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < \dots < O(2^n) < O(n!)$$

多项式时间，易解问题

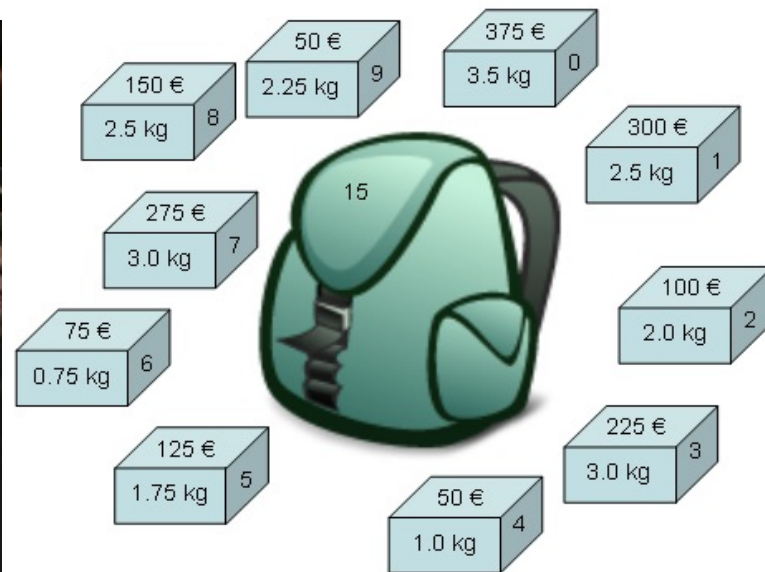
指数时间，难解问题



旅行商问题



舞伴问题



背包问题

# P = NP?

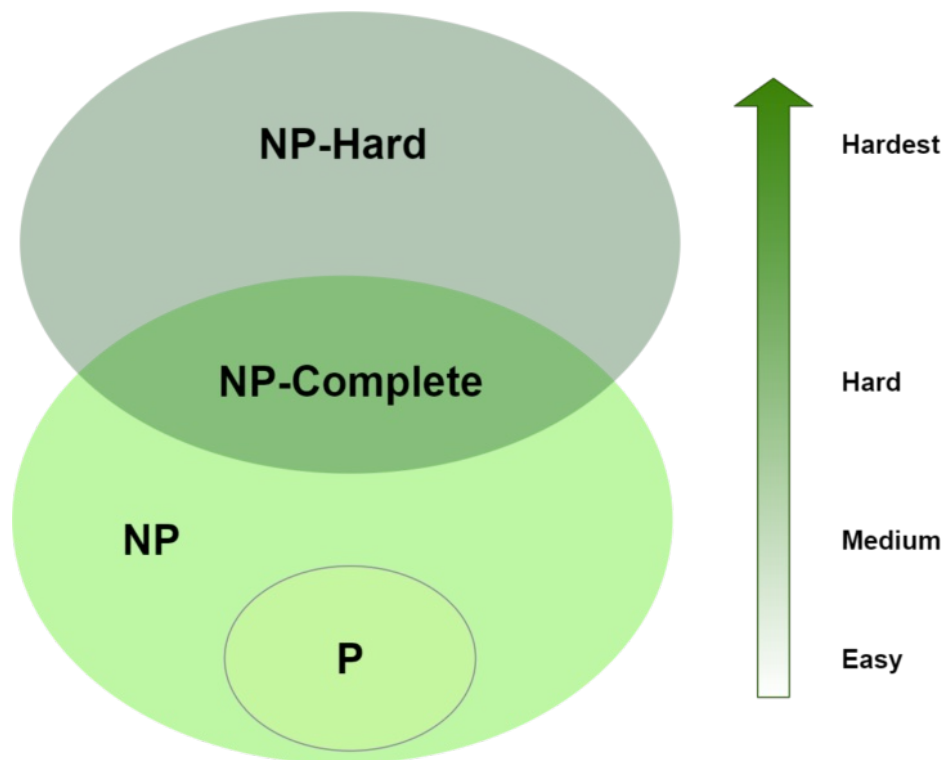
**P:** Polynomial (多项式), 凡能用 $O(n^k)$ 计算量解决的问题



**NP:** Non-deterministic (非确定性) Polynomial (多项式)

**NP-Hard:** NP难

**NP-Complete:** NP完全问题, NPC问题



$n$	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$	$3^n$	$n!$
10	$10^{-6}$	$10^{-5}$	$10^{-5}$	$10^{-4}$	$10^{-3}$	$10^{-3}$	0.059	0.45
20	$10^{-6}$	$10^{-5}$	$10^{-5}$	$10^{-4}$	$10^{-2}$	1(秒)	58(分)	1年
50	$10^{-5}$	$10^{-4}$	$10^{-4}$	0.0025	0.125	36年	$2 \times 10^{10}$ 年	$10^{57}$ 年
1000	$10^{-5}$	$10^{-3}$	$10^{-3}$	1	16小時	$10^{333}$ 年	極大	極大
$10^6$	$10^{-5}$	1	6	1月	$10^5$ 年	極大	極大	極大
$10^9$	$10^{-5}$	16小時	6天	3年	$3 \times 10^9$ 年	極大	極大	極大

表一：以計算機每秒做一百萬次時完成各層次計算量所約需的時間（若無單位，均以秒為單位） From：未來數學家的挑戰



## 1.4 算法分析

### 1-4-2 算法的空间复杂度



# 空间分析



算法在运行过程中需要哪些存储空间？

(1) 输入/输出数据占用的空间

⇒ 取决于问题，与算法无关

```
int CommonFactor(int m, int n)
{
}
}
```

```
void BubbleSort(int r[ ], int n)
{
}
}
```

```
void Equation(double a, double b, double c, double *p, double *q)
{
}
}
```

# 空间分析



算法在运行过程中需要哪些存储空间？

(1) 输入/输出数据占用的空间

⇒ 取决于问题，与算法无关

(2) 算法本身占用的空间

⇒ 与算法相关，大小固定

```
int CommonFactor(int m, int n)
{
    int r = m % n;
    while (r != 0)
    {
        m = n; n = r;
        r = m % n;
    }
    return n;
}
```

# 空间复杂度

 算法在运行过程中需要哪些存储空间？

- |                  |   |
|------------------|---|
| (1) 输入/输出数据占用的空间 |  取决于问题，与算法无关 |
| (2) 算法本身占用的空间    |  与算法相关，大小固定  |
| (3) 执行算法需要的辅助空间  |  与算法相关，体现效率  |

 **空间复杂度**：算法在执行过程中需要的辅助空间数量

除算法本身和输入输出数据所占用的空间外，算法临时开辟的存储空间

 空间复杂度也是问题规模的函数，通常记作： $S(n) = O(f(n))$

# 空间复杂度

```
void BubbleSort(int r[ ], int n)
{
    int j, temp, bound, exchange = n;  $O(1)$ 
    while (exchange != 0)
    {
        bound = exchange; exchange = 0;
        for (j = 1; j < bound; j++)
            if (r[j] > r[j+1]) {
                temp = r[j];
                r[j] = r[j+1];
                r[j+1] = temp;
                exchange=j;
            }
    }
}
```

```
void Merge(int r[ ], int s, int m, int t)
{
    int r1[n];  $O(n)$ 
    int i = s, j = m + 1, k = s;
    while (i <= m && j <= t)
    {
        if (r[i] <= r[j]) r1[k++] = r[i++];
        else r1[k++] = r[j++];
    }
    while (i <= m) r1[k++] = r[i++];
    while (j <= t) r1[k++] = r[j++];
    for (i = s; i < t; i++)
        r[i] = r1[i];
}
```



就地 ( 原地 ) 算法：空间复杂度为 $O(1)$ ，辅助空间是常数

## 1.4 算法分析

### 1-4-3 算法分析举例



非递归算法的时间复杂度分析



递归算法的时间复杂度分析



最好、最坏、平均情况

# 增长率

定理1-1：若 $T(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$ 是一个 $m$ 次多项式，  
则 $T(n) = O(n^m)$



关注**增长率**——忽略所有低次幂和最高次幂的系数

# 非递归算法

例 1  $++x;$

$O(1)$  常数阶

例 2  $\text{for } (i = 1; i \leq n; ++i)$   
 $++x;$

$O(n)$  线性阶

例 3  $\text{for } (i = 1; i \leq n; ++i)$   
 $\text{for } (j = 1; j \leq n; ++j)$   
 $++x;$

$O(n^2)$  平方阶

## 非递归算法

例 4   for (i = 1; i <= n; ++i)  
          for (j = 1; j <= n; ++j)  
          {  
            c[i][j] = 0;  
            for (k = 1; k <= n; ++k)  
              c[i][j] += a[i][k] \* b[k][j];  
          }


$O(n^3)$  立方阶



## 非递归算法


例 5 for (i = 1; i <= n; ++i)  
for (j = 1; j <= i-1; ++j)  
++x;

$$\sum_{i=1}^n \sum_{j=1}^{i-1} 1 = \frac{n(n-1)}{2} \quad O(n^2)$$

 分析的策略是从内部（或最深层部分）向外展开

例 6 for (i = 1; i <= n; i = 2 \* i)  
++x;

$O(\log_2 n)$  对数阶

 分析的策略是设其执行次数为  $T(n)$ ，则有  $2^{T(n)} \leq n$ ，即  $T(n) \leq \log_2 n$

$$\log_2 n = \frac{\log n}{\log 2} \quad \Rightarrow \quad T(n) = O(\log_2 n) = O(\log n)$$

# 递归算法

📎 分析的策略是根据递归过程建立递推关系式并求解（求和表达式）

例 7 分析递推式  $T(n) = \begin{cases} 7 & n = 1 \\ 2T(n/2) + 5n^2 & n > 1 \end{cases}$  的时间复杂度

$$\begin{aligned} \text{假定 } n = 2^k \quad T(n) &= 2T(n/2) + 5n^2 \\ &= 2(2T(n/4) + 5(n/2)^2) + 5n^2 \\ &= 2(2(2T(n/8) + 5(n/4)^2) + 5(n/2)^2) + 5n^2 \\ &= 2^k T(1) + 2^{k-1} 5 \left(\frac{n}{2^{k-1}}\right)^2 + \cdots + 2 \times 5 \left(\frac{n}{2}\right)^2 + 5n^2 \end{aligned}$$

$$T(n) = 7n + 5 \sum_{i=0}^{k-1} \left(\frac{n}{2^i}\right)^2 = 7n + 5n^2 \left(2 - \frac{1}{2^{k-1}}\right) = 7n + 5n^2 \left(2 - \frac{2}{n}\right) = 10n^2 - 3n \leq 10n^2 = O(n^2)$$

# 递归算法



递归算法一般存在如下通用分治递推式：

$$T(n) = \begin{cases} c & n = 1 \\ aT(n/b) + cn^k & n > 1 \end{cases} \quad \text{其中 } a, b, c, k \text{ 都是常数}$$

原问题规模  $\rightarrow$   $n$   
求解  $a$  个子问题  $\rightarrow$   $aT(n/b)$   
子问题规模  $\rightarrow$   $n/b$   
合并解的时间  $\rightarrow$   $cn^k$

$$T(n) = \begin{cases} O(n^{\log_b a}) & a > b^k \\ O(n^k \log_b n) & a = b^k \\ O(n^k) & a < b^k \end{cases}$$

例 8 设某算法运行时间的递推式描述如下，分析该算法的时间复杂度

$$T(n) = \begin{cases} 1 & n = 2 \\ 2T(n/2) + n & n > 2 \end{cases}$$

解： $a = 2$ ， $b = 2$ ， $c = 1$ ， $k = 1$

即满足  $a = b^k$

因此  $T(n) = O(n \log_2 n)$

$$T(n) = \begin{cases} O(n^{\log_b a}) & a > b^k \\ O(n^k \log_b n) & a = b^k \\ O(n^k) & a < b^k \end{cases}$$

## 各种情况

 基本语句的执行次数是否只和问题规模有关？

**例 9** 在一维整型数组  $A[n]$  中顺序查找与给定值  $k$  相等的元素

```
int Find (int A[ ], int n, int k)
{
    for (i = 0; i < n; i++)
        if ( $A[i] == k$ ) break;
    return i;
}
```

 **最好情况**：1 次， $O(1)$

 **最坏情况**： $n$  次， $O(n)$

 **平均情况**： $n/2$  次， $O(n)$

如果算法的时间代价与输入数据有关，则需要分析**最好情况**、**最坏情况**、**平均情况**

# 本章小结

📖 数据结构课程和学科相关背景

📖 掌握相关基本概念

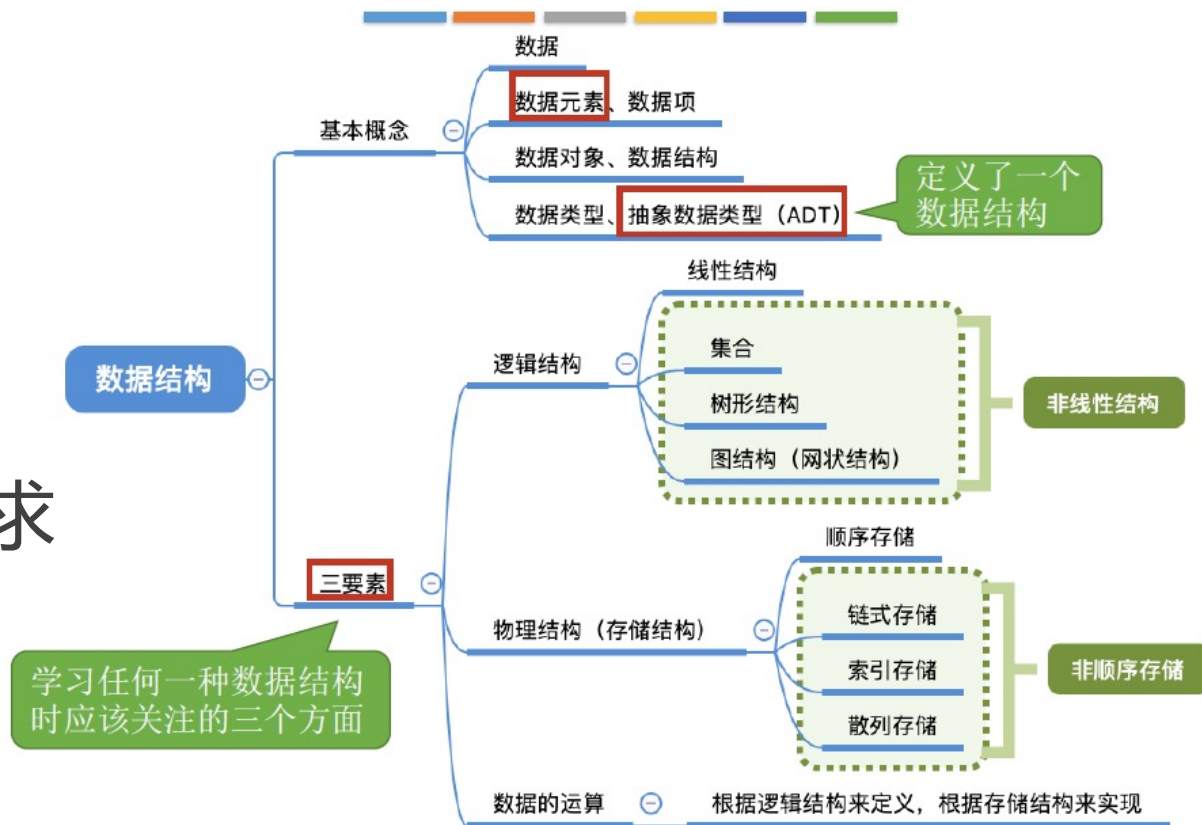
数据结构(逻辑结构、物理结构)

数据类型、抽象数据类型

📖 理解算法的重要特性和设计要求

📖 掌握算法时间复杂度的计算

## 知识回顾





*Thank You !*

*Q & A*