

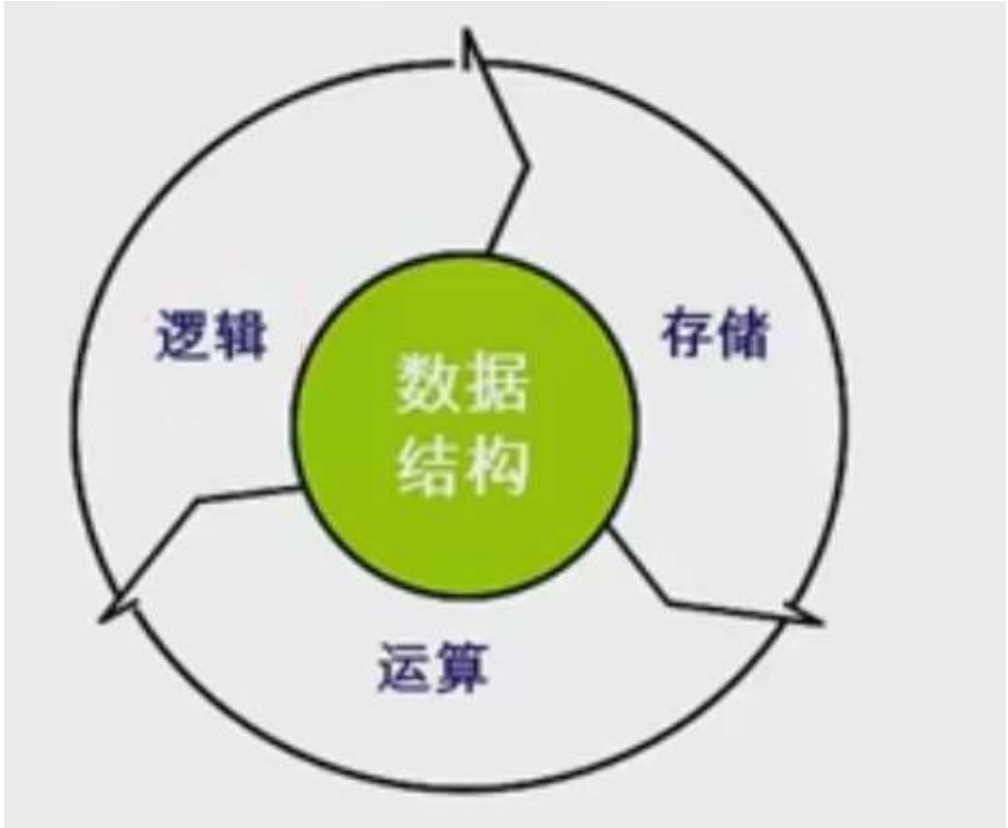
The background features a large, faint sphere with a network of thin, light blue lines connecting various points. Some of these points are highlighted with small, semi-transparent blue circles. The overall aesthetic is clean and technical, suggesting a focus on mathematics or computer science.

# 算法和算法分析

# 数据结构课程的内容体系

方面 步骤	数据表示	数据处理
抽象	逻辑结构	基本运算
实现	存储结构	算法
评价	不同数据结构的比较、选择和算法性能分析	
应用	经典数据结构的应用、计算机中常见操作	

Asymptotic Time Complexity (时间复杂度)  
Asymptotic Space Complexity (空间复杂度)



# 学习目标

The background of the slide features a faint, stylized network graph with nodes and edges, overlaid on a light blue globe. The graph consists of various colored nodes (blue, green, orange) connected by thin lines, forming a complex web. The globe is partially visible behind the graph and a central text box.

- 掌握算法的定义
- 了解算法的描述方法
- 掌握分析算法的时间复杂度
- 熟练分析算法的空间复杂度

# 见识算法

## 📌 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.....

# 见识算法



## 三大学报文章概览

超级计算系统互连网络带内管理的实现与评测·····

曹继军[1];肖立权[1];王克非[1];庞征斌[2];陈琳[1]

通用图形处理器线程调度优化方法研究综述·····

何炎祥[1, 2];张军[1, 2];沈凡凡[3];江南[1, 4];李清安[1, 2];刘子骏[1]

数据中心网络高效数据汇聚传输算法·····

陆菲菲[1, 2];郭得科[3];方兴[1];谢向辉[1];罗兴国[2]

面向三维多核片上系统的热感知硅后能耗优化方法·····

靳松[1];韩银和[2];王瑜[1]

基于OpenCL的Viola-Jones人脸检测算法性能优化研究·····

贾海鹏;张云泉;袁良;李士刚

软错误率变动对检查点机制的影响·····

贾文涛;张春元

求解Boltzmann模型方程高性能并行算法在航天跨流域空气动力学应

用研究

李志辉[1, 2];蒋新宇[1];吴俊林[1];徐金秀[2];白智勇[3]

☐ 求解#SMT问题的局部搜索算法

☐ 城市路网多路口路径动态实时选择方法

☐ 动作空间带平衡约束圆形Packing问题的拟物求解算法

☐ 基于图压缩的最大Steiner连通核查询处理

☐ 一种对时空信息的KNN查询处理方法

☐ 偏序时态XML索引TempPartialIndex

☐ 面向实体识别的聚类算法

一种不完备混合数据集集成聚类算法

史倩玉;梁吉业;赵兴旺

DOI: 10.7544/issn1000-1239.2016.20150592

2016 Vol. 53 (9): 1979-1989 [摘要] (160) HTML (1 KB) PDF

任务调度算法中新的自适应惯性权重计算方法

李学俊;徐佳;朱二周;张以文

DOI: 10.7544/issn1000-1239.2016.20151175

2016 Vol. 53 (9): 1990-1999 [摘要] (128) HTML (1 KB) PDF

术

基于CS的无线传感器网络动态分簇数据收集算法

张策;张霞;李鸥;王冲;张大龙

DOI: 10.7544/issn1000-1239.2016.20150459

2016 Vol. 53 (9): 2000-2008 [摘要] (88) HTML (1 KB) PDF

低占空比无线传感器网络同步MAC协议最优信标间隔分析

邢宇龙;陈永锐;易卫东;段成华

DOI: 10.7544/issn1000-1239.2016.20150463

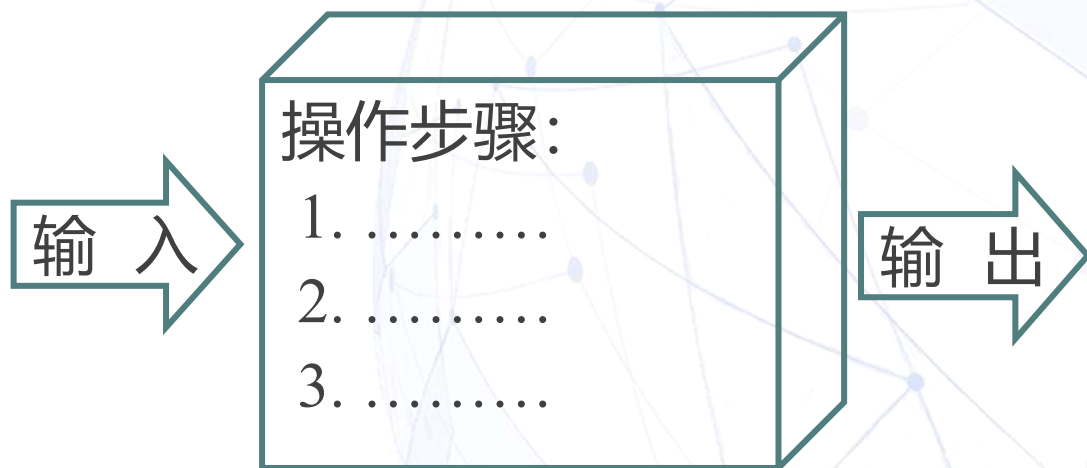
2016 Vol. 53 (9): 2009-2015 [摘要] (55) HTML (1 KB) PDF

可保障数据无中断传输的冗余树算法研究

算法是计算机科学的基石

# 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**



对于一个特定问题，可能没有算法，可能有多个性能不同的算法。



# 算法的特性

🕒 算法的操作步骤应该满足什么要求？

- (1) 输入：有0个或多个输入
- (1) 输出：有1个或多个输出
- (3) 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
- (4) 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
- (5) 可行性：描述算法的指令可以转换为某种程序设计语言对应的语句，并在计算机上可执行。



# 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



# 算法的特性

例 1 设计算法求两个自然数的最大公约数。

【经典算法】辗转相除法



如何描述这个算法

# 算法的描述方法



用自然语言描述算法



用流程图描述算法



用程序设计语言描述算法



用伪代码描述算法

# 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

# 自然语言描述算法

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——自然语言描述】** 设两个自然数为  $m$  和  $n$ ，欧几里德算法如下：

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））

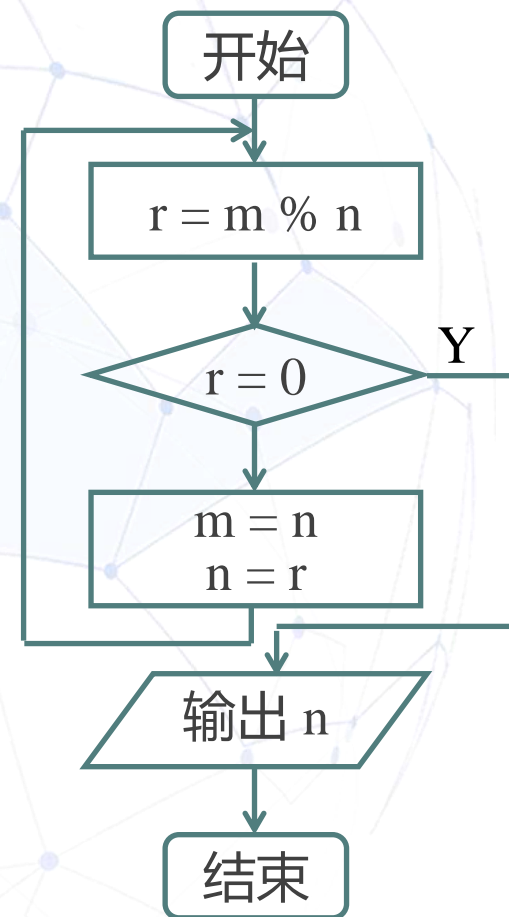
【算法——流程图描述】设两个自然数为  $m$  和  $n$ ，算法为

优点：流程直观

缺点：缺少严密性、灵活性

使用方法：描述简单算法

注意事项：注意抽象层次



# 程序语言描述算法

✦ 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——程序语言描述】** 设两个自然数为  $m$  和  $n$ ，算法为

优点：能由计算机执行

缺点：抽象性差，对语言要求高

使用方法：算法需要验证

注意事项：将算法写成**子函数**

```
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

- ✦ 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。
- ✦ 伪代码不是一种实际的编程语言，但在表达能力上类似于编程语言，同时极小化了描述算法的不必要的技术细节
- ✦ 伪代码被称为“算法语言”或“第一语言”

# 伪代码描述算法

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——伪代码描述】** 设两个自然数为  $m$  和  $n$ ，算法为

算法：ComFactor

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  $r = m \% n$ ;
  - 2.2  $m = n$ ;
  - 2.3  $n = r$ ;
3. 输出  $n$ ;

优点：表达能力强，抽象性强，容易理解，容易实现

算法0-0：求两个非负整数的最大公约数GCD( $m, n$ )

输入： $m, n \in$  非负整数集

输出： $m, n$  的最大公约数

1. **if**  $m < n$  **then** // 判断 $m$ 与 $n$ 的大小
2. |  $m \leftrightarrow n$  // 如 $m < n$ ，则交换 $m$ 与 $n$
3. **end**
4. **while**  $m \bmod n \neq 0$  **do** //  $m$ 不能整除 $n$ 执行循环
5. |  $r \leftarrow m \bmod n$  // 计算 $m$ 除以 $n$ 的余数 $r$
6. |  $m \leftarrow n$  // 用 $n$ 重新赋值 $m$ 值
7. |  $n \leftarrow r$  // 用 $r$ 重新赋值 $n$ 值
8. **end**
9. **return**  $n$  //  $m$ 整除 $n$ ， $n$ 即为最大公约数

形式不唯一

# 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

# 算法的性能标准



- (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) **可理解性(可读性)**: 算法容易理解和实现。
- (3) **高效性**: 具有较短的执行时间并占用较少的辅助空间。
- (4) **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。

# 算法分析

---




度量算法效率的方法



算法的时间复杂度



算法的空间复杂度


 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

 事前分析（定性分析）：对算法所消耗资源的一种估算方法

 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  
空间



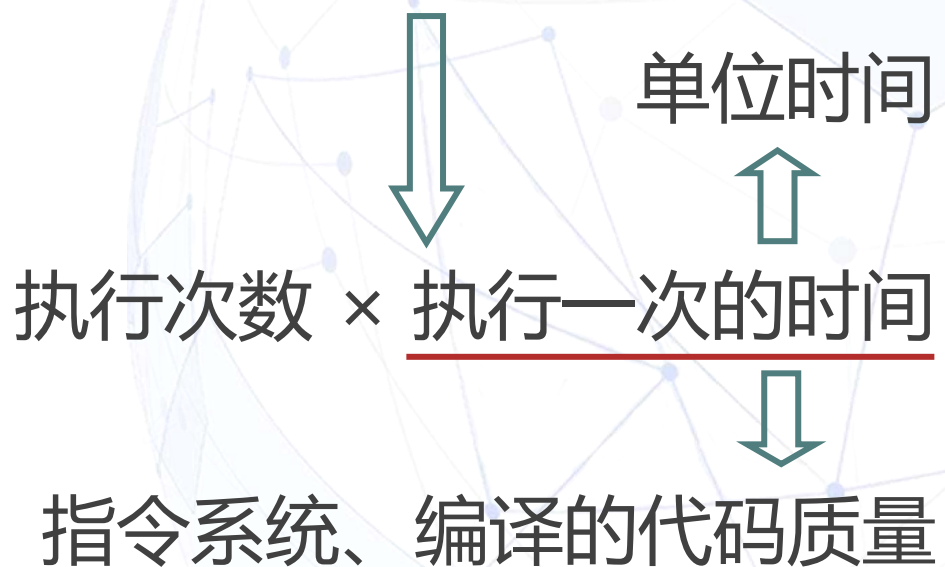
# 什么样的算法是好的？

**空间复杂度 $S(n)$**  —— 根据算法写成的程序在执行时**占用存储单元的长度**。这个长度往往与输入数据的规模有关。空间复杂度过高的算法可能导致使用的内存超限，造成程序非正常中断。

**时间复杂度 $T(n)$**  —— 根据算法写成的程序在执行时**耗费时间的长度**。这个长度往往也与输入数据的规模有关。时间复杂度过高的低效算法可能导致我们在有生之年都等不到运行结果。

# 时间复杂度

算法的**执行时间** = 每条语句执行时间之和



一段程序代码执行的时间性能一般与以下几种因素相关：

- **计算机的硬件性能**，如CPU、GPU的核心数和频率决定了机器的性能
- **编程语言和生成代码的质量**，如Python、C++等不同语言及编译器，所生成的可执行代码效率不同
- **问题和数据的规模**，如在10本书和100本书中寻找所需要的书籍，处理的方式和效率是不一样的
- **算法设计效率**，如针对相同规模大小为N的输入，算法需要消耗线性的时间还是二次幂的时间

# 时间复杂度

算法的**执行时间** = 每条语句执行时间之和 **受制因素太多**  
 $\approx$  每条语句**执行次数**之和

将所有语句的执行次数之和作为算法的**运行时间函数  $T(n)$** 。

**$n$** 指的是问题的规模，如被排序的数组中元素的个数，矩阵的阶数等。

当 **$n \rightarrow \infty$**  时，时间函数的数量级表示称为**渐进时间复杂度**，简称**时间复杂度**，记为 **$T(n) = O(f(n))$**

表示当 **$n$** 趋于无穷大时，算法运行时间由 **$f(n)$** 决定，运行时间的增长率与 **$f(n)$** 的增长率相同。

# 时间复杂度计算方法

- (1) 将所有语句的操作执行次数之和作为算法的运行时间函数 $T(n)$ 。
- (2) 忽略时间函数 $T(n)$ 的低次项部分和最高次项的系数，只取时间函数的最高次项，并辅以大O记号表示。
  - $n$ 指的是问题的规模，如被排序的数组中元素的个数，矩阵的阶数等。

程序段	语句执行次数
<code>m=0;</code>	1
<code>for (i=1;i&lt;=n;i++)</code>	$1+n+1+n$
<code>    m=m+1;</code>	$n$
时间函数	$f(n)=3n+3$
渐进时间复杂度	$T(n)=O(n)$

程序段。	语句执行次数。
<code>m=0;</code>	1
<code>for (i=1;i&lt;=n;i++)</code>	1,n+1,n
<code>for (j =i; j&lt;=n; j++)</code>	n, $n(n+1)/2+n$ , $n(n+1)/2$
<code>m=m+1;</code>	$n(n+1)/2$
时间函数。	$f(n)=1.5n^2+5.5n+3$
渐进时间复杂度	$T(n)=O(n^2)$

在上述算法中，将所有操作执行次数加起来，得到时间函数 $T(n)=1.5n^2+5.5n+3$ ， $n$ 是问题的规模，规模因子。

当 $n$ 足够大时，对 $T(n)$ 的值起决定性影响的是第一项 $1.5n^2$ ，第二、三项可以忽略不计，即可以认为 $T(n)$ 的值接近于 $1.5n^2$ ，进而认为该算法运行时间的增长率与 $n^2$ 的增长率是相同的，表示为 $T(n)=O(n^2)$ 。



# 时间复杂度计算的简化方法

◆将某个关键操作的执行次数的数量级作为时间复杂度。

关键操作通常是循环最深层的一句语句，是算法中执行原操作的次数数量级最高的一句语句，当执行次数最高的语句有多句时，可以选择任意一句为其关键操作，通常选取算法中完成主要任务的语句为关键操作。

程序段
<code>m=0;</code>
<code>for (i=1;i&lt;=n;i++)</code>
<code>  for (j =i; j&lt;=n; j++)</code>
<code>    m=m+1;</code>

渐进时间复杂度  $O(n^2)$

# 时间复杂度的渐进表示法

✦ 当问题规模充分大时，算法中某个关键操作的执行次数在渐近意义下的阶——关注的是增长趋势

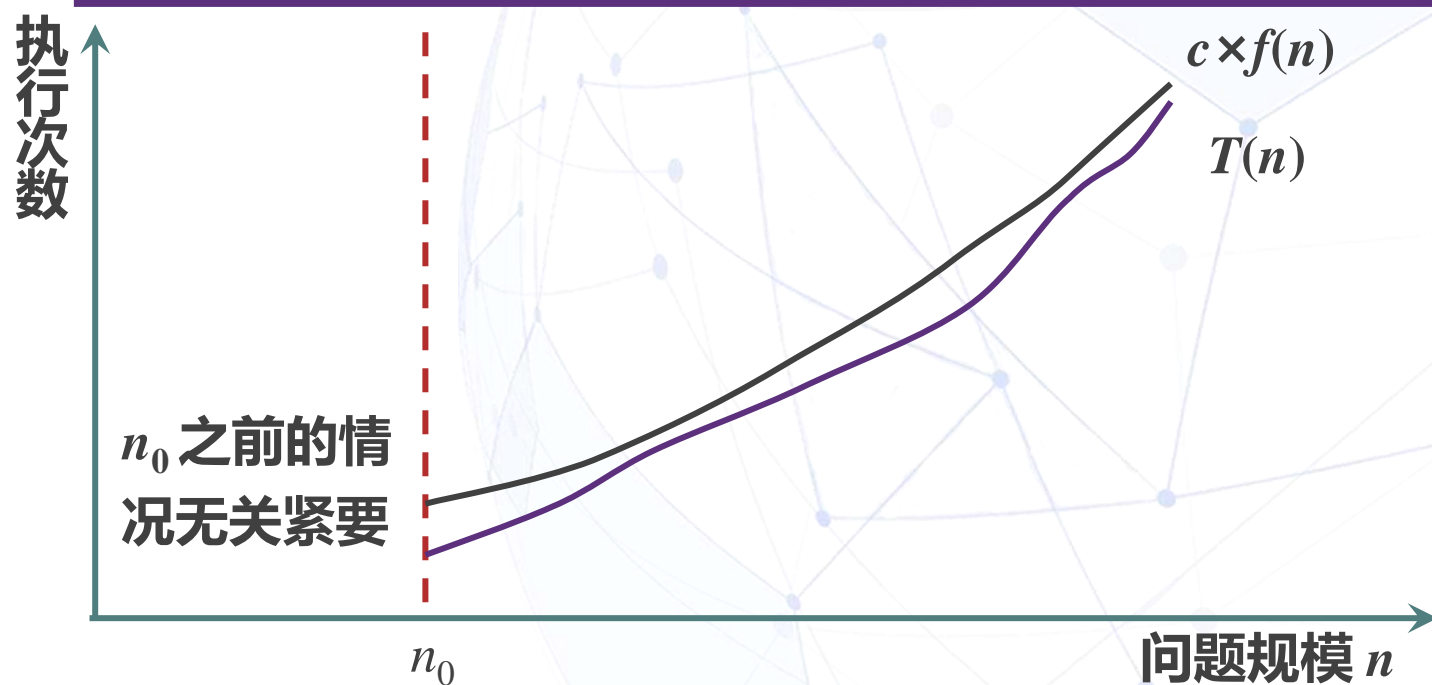
四种常见的渐近时间复杂度表示方法：

$T(n) = O(f(n))$	$T(n) = \Omega(f(n))$	$T(n) = \Theta(f(n))$	$T(n) = o(f(n))$
------------------	-----------------------	-----------------------	------------------

与大O表示法的“小于等于”不同，小o表示法代表“严格小于”，即T(n)的数量级小于f(n)的数量级。

# 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

以大O表示的时间复杂度，是对算法执行时间的一种保守估计，是算法性能的上界，即对于规模为 $n(n \geq n_0)$ 的任意输入，算法的运行时间都不会超过 $O(f(n))$ ，即时间性能不会差于 $O(f(n))$ 。

假设  $T(n) = 2n^2 + n$

存在正常数 $c=3$ 和 $n_0=1$ ，当 $n \geq n_0$ 时， $T(n) = 2n^2 + n \leq 2n^2 + n^2 \leq cn^2$ ，总是成立，因此 $T(n) = O(n^2)$ 。

# 时间复杂度的渐进表示法

$T(n) = O(f(n))$  表示存在常数  $C > 0, n_0 > 0$  使得当  $n \geq n_0$  时有  $T(n) \leq C \cdot f(n)$

$T(n) = \Omega(f(n))$  表示存在常数  $C > 0, n_0 > 0$  使得当  $n \geq n_0$  时有  $T(n) \geq C \cdot f(n)$

$T(n) = \Theta(f(n))$  表示存在常数  $C_1 > 0, C_2 > 0, n_0 > 0$ , 使得当  $n \geq n_0$  时有  $C_1 \cdot f(n) \leq T(n) \leq C_2 \cdot f(n)$ , 即  $T(n) = O(f(n)) = \Omega(f(n))$

# 大O 记号

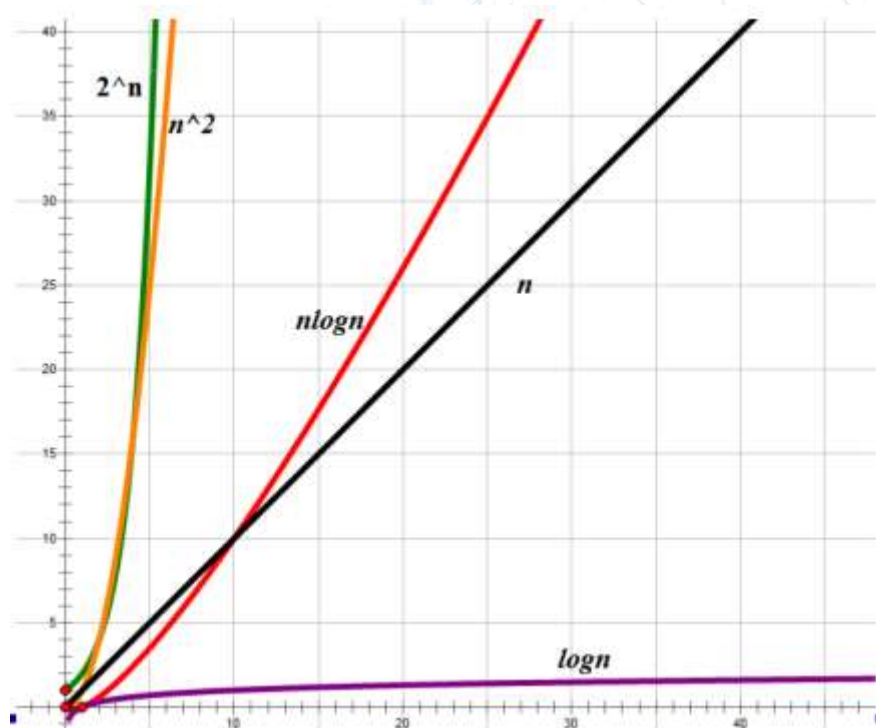


常见的时间复杂度:

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

多项式时间, 易解问题

指数时间, 难解问题



输入规模  $n$

函数	1	2	4	8	16	32
1	1	1	1	1	1	1
$\log n$	0	1	2	3	4	5
$n$	1	2	4	8	16	32
$n \log n$	0	2	8	24	64	160
$n^2$	1	4	16	64	256	1024
$n^3$	1	8	64	512	4096	32768
$2^n$	2	4	16	256	65536	4294967296
$n!$	1	2	24	40320	2092278988800	$26313 \times 10^{33}$



1. 算法的时间复杂度都要通过算法中基本语句的执行次数来确定。

☐ A 正确

☒ B 错误

提交

2. 算法的时间复杂度是对算法所消耗资源的一种精确计算。

☐ A 正确

☒ B 错误

提交

6. 某算法的时间复杂度是 $O(n^2)$ ，表明该算法（ ）。

- ☐ A 问题规模是 $n^2$
- ☐ B 执行时间等于 $n^2$
- ☒ C 执行时间与 $n^2$ 成正比
- ☐ D 问题规模与 $n^2$ 成正比

提交

# 时间复杂度计算原则



## 渐近表示法的计算

**求和定理：** 假设两个已知程序片段的时间复杂度分别为 $T_1(n)=O(f(n))$ 和 $T_2(n)=O(g(n))$ ，那么顺序组合两个程序片段得到的程序的时间复杂度为：

$$T_1(n)+T_2(n)= O(\text{Max}(f(n), g(n)))$$

**用途：** 适用于顺序语句/程序片段

### 变量计数

```
x = 0; y = 0;
```

$T_1(n) = O(1)$

```
for ( int k = 0; k < n; k ++ )  
    x ++;
```

$T_2(n) = O(n)$

```
for ( int i = 0; i < n; i ++ )  
    for ( int j = 0; j < n; j ++ )  
        y ++;
```

$T_3(n) = O(n^2)$

$$\begin{aligned} T(n) &= T_1(n)+T_2(n)+T_3(n) = O(\max(1, n, n^2)) \\ &= O(n^2) \end{aligned}$$



## 渐近表示法的计算

**求积定理：** 假设两个已知程序片段的时间复杂度分别为 $T1(n)=O(f(n))$ 和 $T2(n)=O(g(n))$ ，那么交叉乘法组合两个程序片段得到的程序时间复杂度为：

$$T1(n) \cdot T2(n) = O(f(n) \cdot g(n))$$

**用途：** 适用于嵌套/多层嵌套循环语句

### 变量计数

```
x = 0; y = 0;
```

$T_1(n) = O(1)$

```
for ( int k = 0; k < n; k ++ ) {
```

$T_2(n) = O(n)$

```
    x ++;
```

```
    for ( int i = 0; i < n; i ++ )
```

$T_3(n) = O(n^2)$

```
        for ( int j = 0; j < n; j ++ )
```

```
            y ++;
```

```
}
```

$$\begin{aligned} T(n) &= T_1(n) + (T_2(n) * T_3(n)) = T_1(n) + O(n * n^2) \\ &= O(\max(1, n^3)) = O(n^3) \end{aligned}$$



5. 以下程序段的时间复杂度是（ ）。

```
for (i = 0; i < m; i++) {
    for (j = 0; j < n/2; j++)
        cout << data[i][j];
    cout << endl;
}
```

- ☐ A  $O(n^2)$
- ☐ B  $O(m^2)$
- ☒ C  $O(m \times n)$
- ☐ D  $O(m \times n/2)$

提交



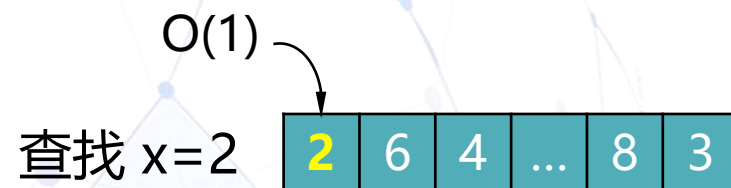
基本语句的执行次数是否只和问题规模有关？

如果算法的时间代价与输入数据有关，需要分析最好情况、最坏情况、平均情况

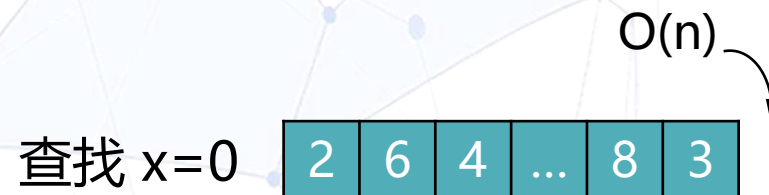
# 时间复杂度的最好、最坏和平均情况

- 例：假设现有一函数F，其功能是在一个无序的数组A中查找变量 x 出现的位置。如果找到则停止，并返回x在数组A中的下标；若没有找到，则返回-1。

**最好情况复杂度：**在最理想的情况下，算法所能达到的最高效率。要查找的变量 x 正好是数组的第一个元素，最好情况时间复杂度 $O(1)$



**最坏情况复杂度：**这是算法可能遇到的最糟糕的情况的效率。在这种情况下，算法耗时最长。如果数组中没有要查找的变量 x，需要把整个数组都遍历一遍，这时对应的是最坏情况时间复杂度 $O(n)$



**平均情况复杂度：**这是算法在所有可能输入的平均效率。在计算平均情况复杂度时，通常假设所有输入出现的概率符合特定的分布（最简单的可以假设所有输入为等可能）。对于函数F，其平均时间复杂度为 $O(n)$

$$\sum_{i=1}^n i \cdot \frac{1}{n+1} = \frac{n(n+1)}{2(n+1)} = O(n)$$

时间复杂度还跟数据集的状态有关

# 时间复杂度的最好、最坏和平均情况

在分析一般算法的效率时，我们经常关注下面两种复杂度

- 最坏情况复杂度  $T_{worst}(n)$
- 平均复杂度  $T_{avg}(n)$

$$T_{avg}(n) \leq T_{worst}(n)$$

# 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  $\Rightarrow$  取决于问题，与算法无关
- (2) 算法本身占用的空间  $\Rightarrow$  与算法相关，大小固定
- (3) 执行算法需要的辅助空间  $\Rightarrow$  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： $S(n) = O(f(n))$

# 空间复杂度分析示例

将一维数组a中的n个数逆序存放到原数组中去

```
for (j = 0; j < n/2; j++){  
    temp = a[j];  
    a[j] = a[n-1-j];  
    a[n-1-j] = temp;  
}
```

$O(1)$

```
for (j = 0; j < n; j++){  
    b[j] = a[n-1-j];  
  
for (j = 0; j < n; j++){  
    a[j] = b[j];  
}
```

$O(n)$

📌 空间复杂度为 $O(1)$ 意味着辅助空间是**常数**





## 算法优化 - 空间复杂度

**例：**假设需要输出由小到大，从1 到n的所有的数字。可用递归调用完成。

```
void PrintN ( int N )
{ if ( N ){
    PrintN( N - 1 );
    printf("%d\n", N );
}
return;
}
```

```
void PrintN ( int N )
{ int i;
  for ( i=1; i<=N; i++ ){
    printf("%d\n", i );
  }
  return;
}
```

**递归算法：**直到 $n=0$ 开始返回上一层，并从1开始输出，一直到最后打印 $n$ 。该函数通常只能执行数万次，就会因递归层数过多，系统栈空间不足而报错，也称**递归爆栈**。因为递归时，每次进入更深一层，都需要将当前空间的状态进行存储，消耗一定的内存空间。

**循环算法：**只涉及1个变量的维护，循环调用多少次，内存消耗也不变。不会内存溢出。

# 时间性能和空间性能的平衡

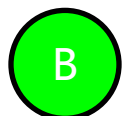
- 算法的时间效率和空间效率不能兼顾
- 空间换取时间的例子
  - 被查找的表上增加索引结构，提高查找时间性能
  - 根据月份获得该月天数，用列表存储每月天数

3. 算法的空间复杂度指的是输入/输出数据占用的空间以及执行算法的辅助空间。



A

正确



B

错误

提交

4. 就地算法或原地算法指的是算法在执行过程中不需要辅助空间。

☐ A 正确

☒ B 错误

提交

7. 下面说法错误的是（ ）。

- ☒ A 算法原地工作的含义是指不需要任何额外的辅助空间
- ☒ B 在相同的规模 $n$ 下，复杂度 $O(n)$ 的算法在时间上总是优于复杂度 $O(n^2)$ 的算法
- ☐ C 所谓时间复杂度是指最坏情况下，估算算法执行时间的一个上界
- ☐ D 同一个算法，实现语言的级别越高，执行效率就越低

提交

# 课堂小结

---

算法的性能评价方法

算法的时间和空间复杂度分析





# 课后作业



## 今日作业

- 请编写在长度为 $n$ 的整型数组 $a$ 中找出最大值的算法，并分析算法的时间复杂度。