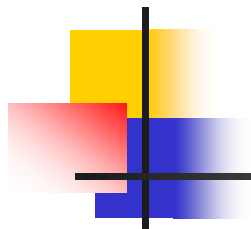




算法分析与设计

Analysis and Design of Algorithm

第2次课



算法复杂性

算法复杂性与算法效率

- 硬件速度增加，算法效率还那么重要吗？
 - 大小为 n 的实例的执行，需要 $10^{-4} \times 2^n$ s。则：

$$n = 10 \quad 10^{-4} \times 2^{10} \approx 0.1\text{s}$$

$$n = 20 \quad 10^{-4} \times 2^{20} \quad \text{扩大} 2^{10} \text{倍} \quad \text{一年至多解}$$

$$n = 30 \quad 10^{-4} \times 2^{30} \quad \text{扩大} 2^{10} \times 2^{10} \text{倍} \quad n=38$$

-
- **假设**计算机性能提高100倍，那么需要 $10^{-6} \times 2^n$ s。
一年时间可以解多大的实例？（求 n 的值）



算法复杂性与算法效率

- **假设**改进算法，在原来计算机上，需 $10^{-2} \times n^3$ s。

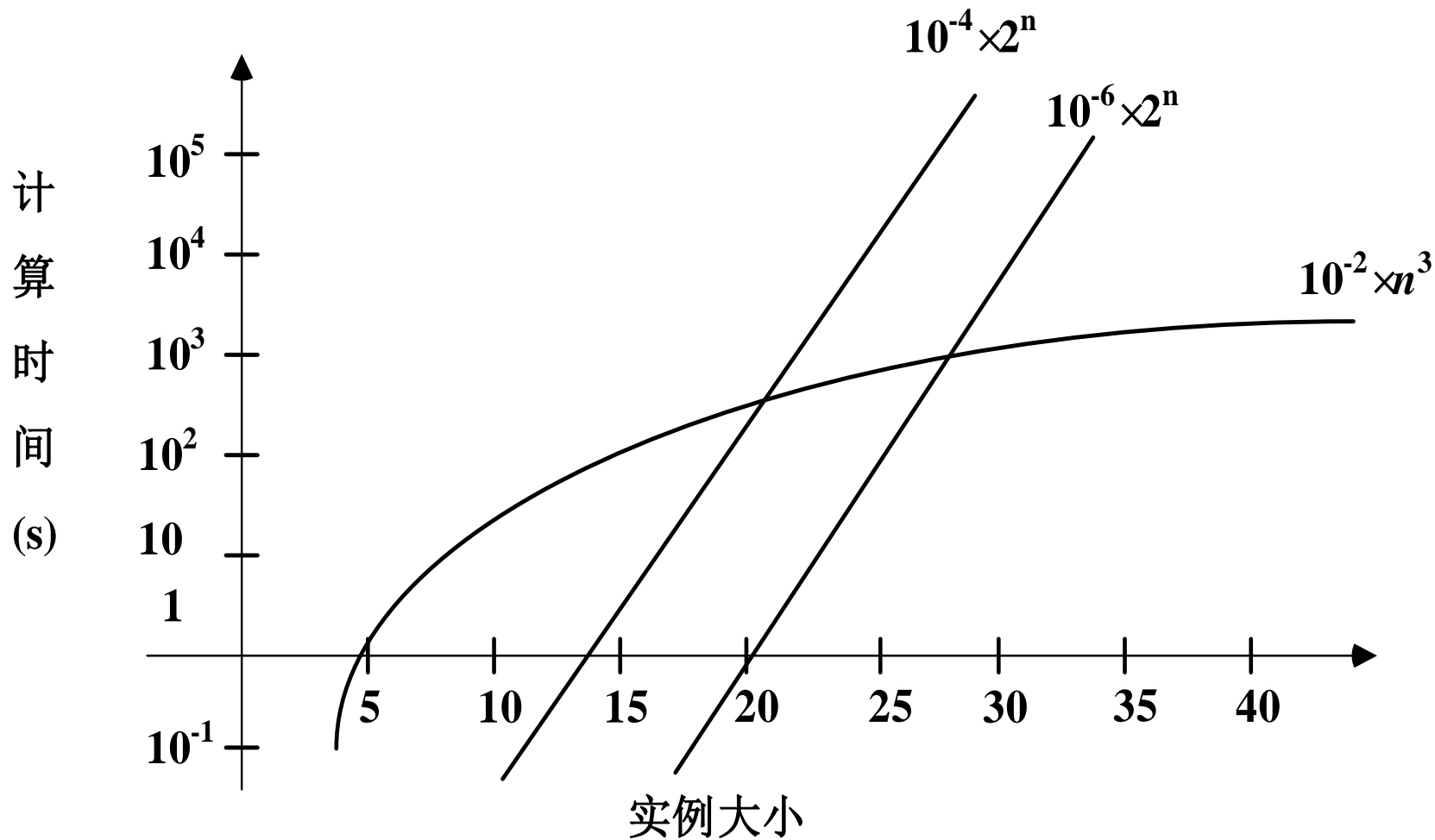
$$n = 10 \quad \approx 10s$$

$$n = 20 \quad 1 \sim 2 \text{ min}$$

$$n = 30 \quad \leq 4.5 \text{ min}$$

因此，用一天可以解决大小超过200的实例，一年处理 $n=1500$ 实例。

算法复杂性与算法效率



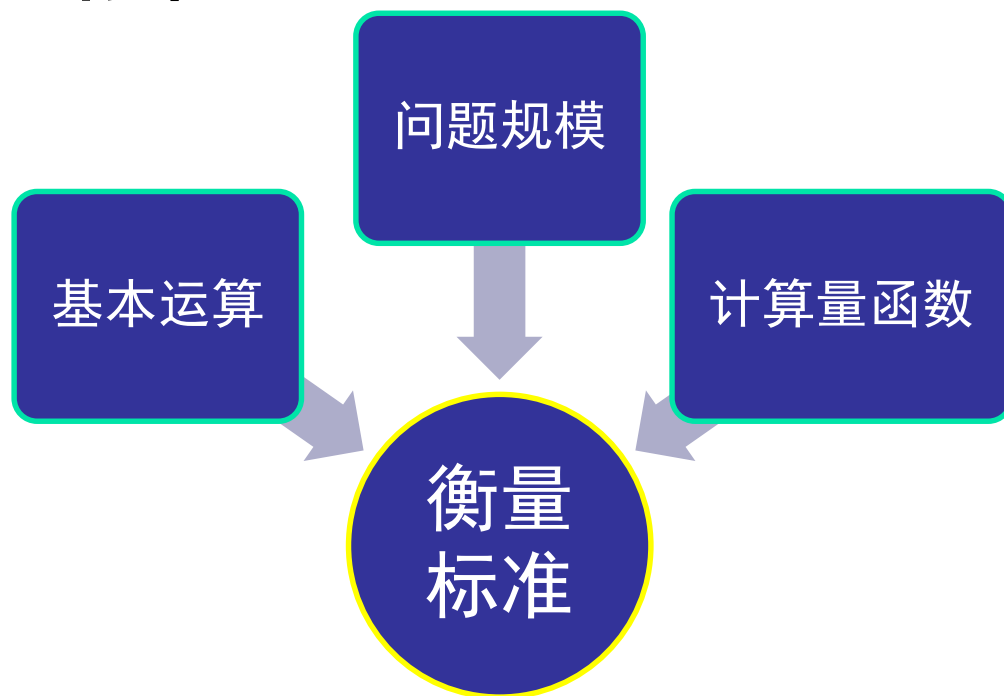
算法复杂性与算法效率

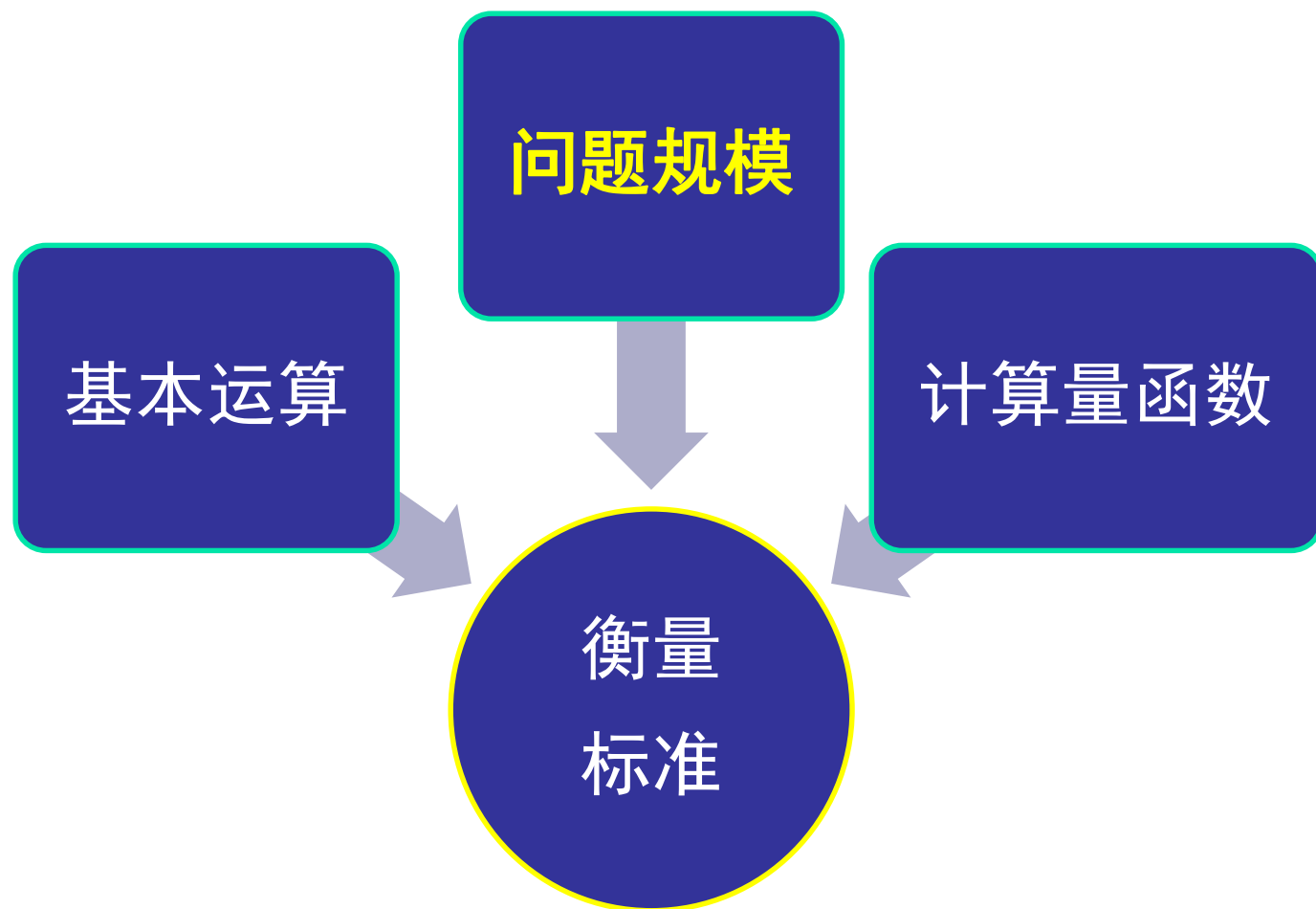
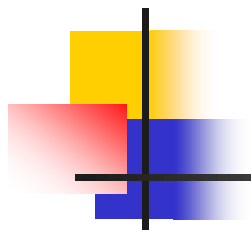
	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

太难!!!

算法好坏的衡量尺度

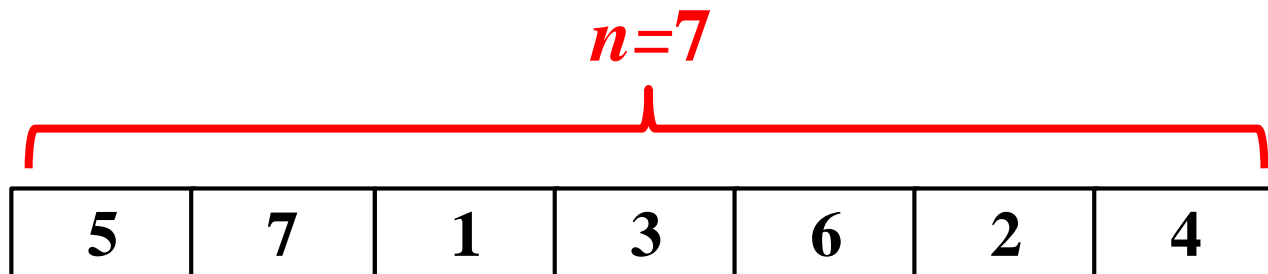
- 用所需的计算时间来衡量一个算法的好坏，不同的机器相互之间无法比较。
 - 能否有一个独立于具体计算机的客观衡量标准。
- 常见的衡量标准





问题的规模

- 问题的规模：一个或多个整数，作为输入数据量的测度。
 - 数表的长度(数据项的个数)，(问题：在一个长度为 n 的数组中寻找元素 x)；





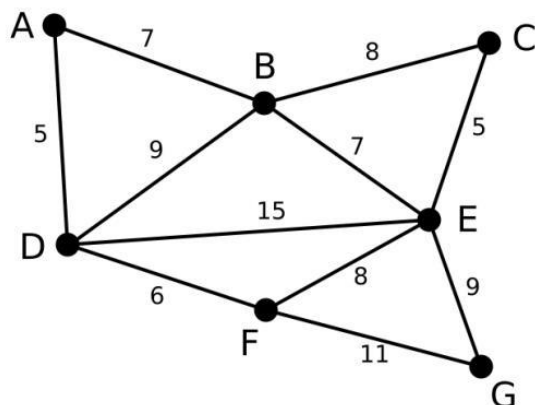
问题的规模

- 问题的规模：一个或多个整数，作为输入数据量的测度。
 - 数表的长度(数据项的个数)，(问题：在一个长度为 n 的数组中寻找元素 x)；
 - 矩阵的最大维数(阶数) (问题：求两个实矩阵相乘的结果)

$$n=3 \left\{ \begin{bmatrix} A & B \\ C & D \\ E & F \end{bmatrix} \times \begin{bmatrix} G & H \\ I & J \end{bmatrix} = \begin{bmatrix} AG + BI & AH + BJ \\ CG + DI & CH + DJ \\ EG + FI & EH + FJ \end{bmatrix} \right.$$

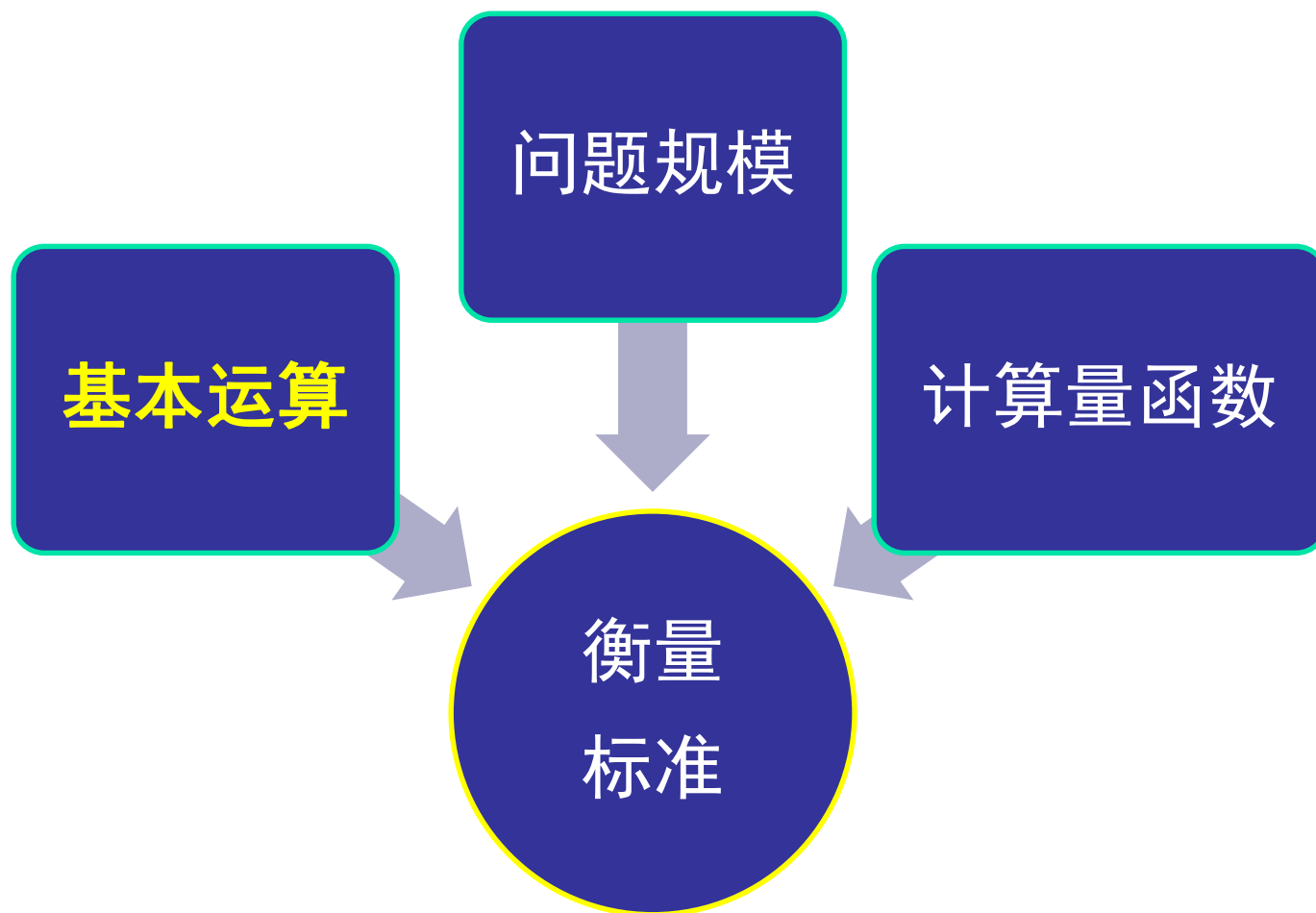
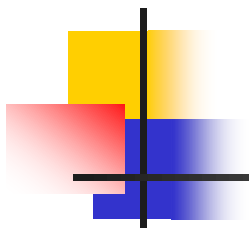
问题的规模

- 问题的规模：一个或多个整数，作为输入数据量的测度。
- 输入规模通常用 n 来表示，也可有两个以上的参数
 - 图中的顶点数和边数(图论中的问题)



顶点数：7

边数：11



基本运算

- 概念：
 - 也称为“元运算”
 - 指执行时间可以被一个常数限定，只与**环境**有关。
- 因此，分析时只需要关心执行的基本运算次数，而不是它们执行确切时间。
 - 例子：

a 次加法，一个**加** $\leq t_a$;
 m 次乘法，一个**乘** $\leq t_m$;
 s 次赋值，一个**赋值** $\leq t_s$;

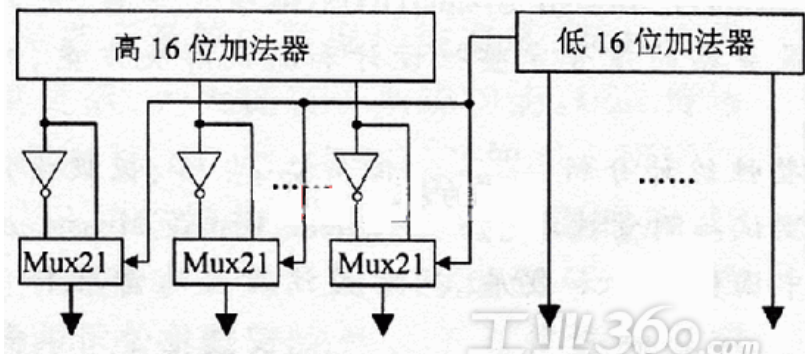
$$\left. \begin{array}{l} a \text{次加法, 一个加} \leq t_a; \\ m \text{次乘法, 一个乘} \leq t_m; \\ s \text{次赋值, 一个赋值} \leq t_s; \end{array} \right\} \begin{array}{l} t \leq at_a + mt_m + st_s \\ \leq \max(t_a, t_m, t_s) \times (a + m + s) \end{array}$$

只和**基本运算**相关

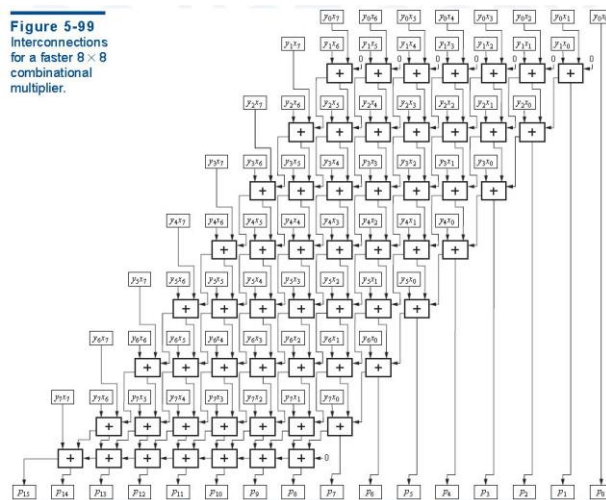
机器、语言编译

基本运算

- 一般可以认为加法和乘法都是一个单位开销的运算。
 - 理论上，这些运算都不是基本运算，因为操作数的长度影响了执行时间。



CPU加法器

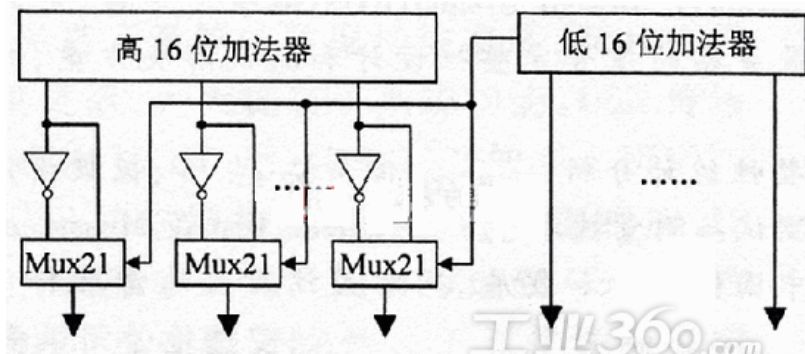


CPU乘法器

基本运算

- 一般可以认为加法和乘法都是一个单位开销的运算。

实际，只要实例中操作数长度相同，即可认为是基本运算。



CPU加法器

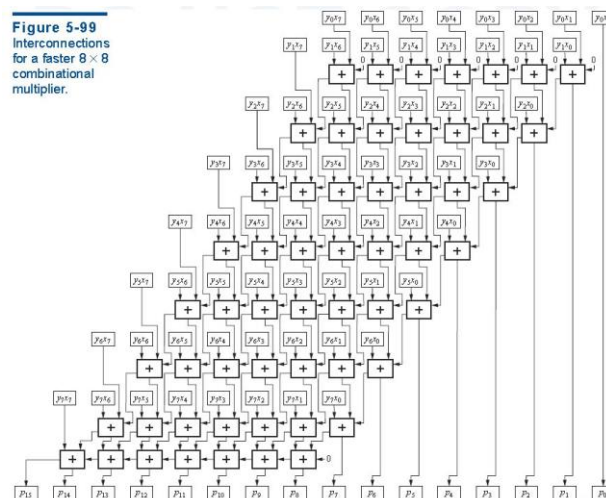


Figure 5-99
Interconnections
for a faster 8 x 8
combinational
multiplier.

CPU乘法器



以冒泡排序为例

```
void bubbleSort(T[] a) {  
    int len = a.length;  
    for (int i=0; i<n; i++) {  
        for (int j=0; j<n-i-1; j++) {  
            if (a[j]>a[j+1]) {  
                int temp = a[j];  
                a[j] = a[j+1];  
                a[j+1] = temp;  
            }  
        }  
    }  
}
```

赋值、比较、加法
都是基本运算



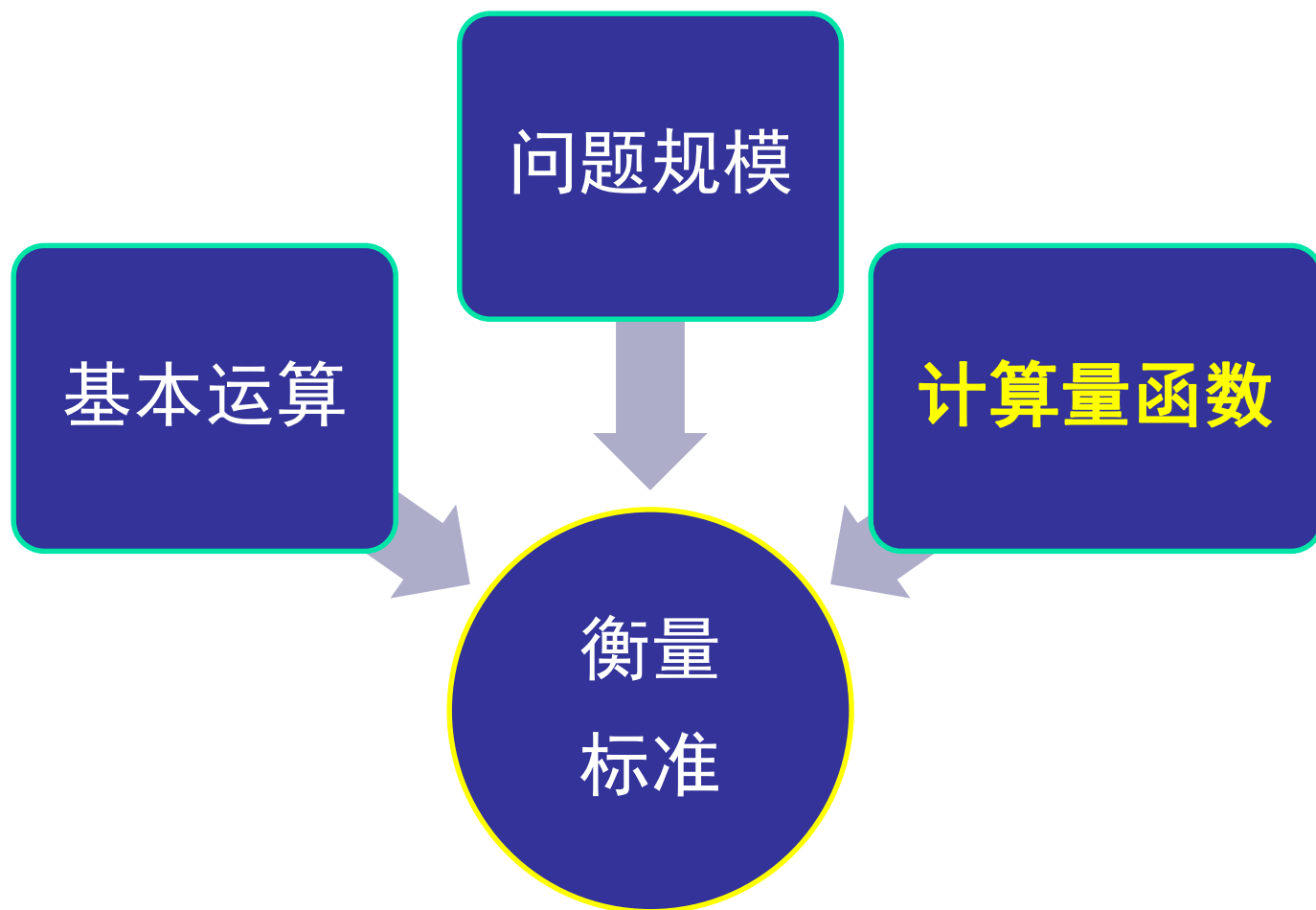
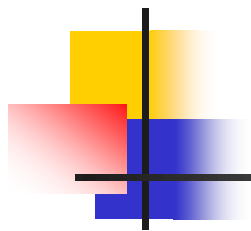
基本运算

- 例如

- 在一个表中寻找数据元素 x ： x 与表中的一个项进行比较；
- 两个实矩阵的乘法：实数的乘法（及加法） $C=AB$ ；
- 数组排序：表中的两个数据项进行比较。

- 通常情况下，讨论一个算法优劣时，我们只讨论基本运算的时间和执行次数。

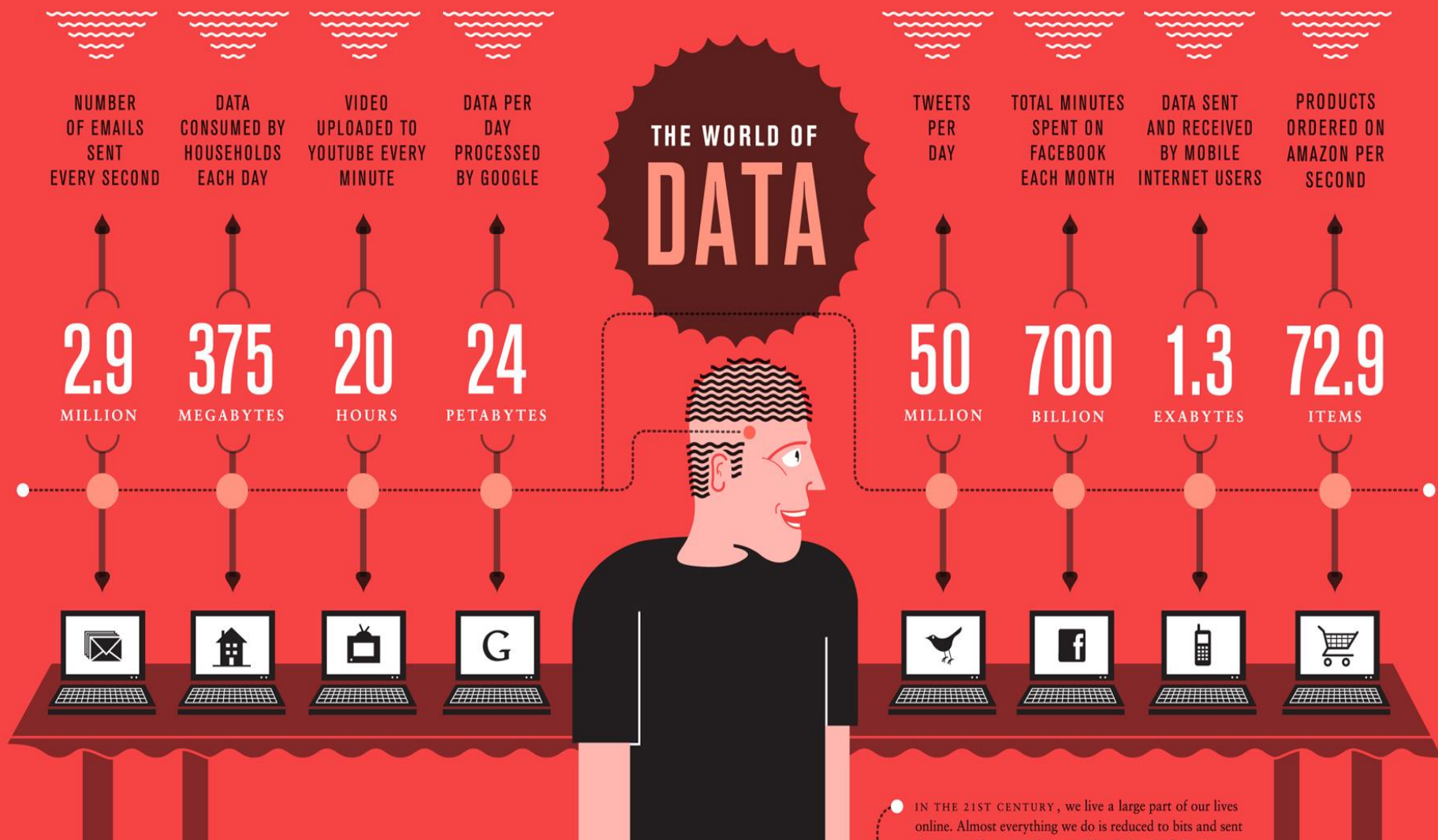
因为它是占支配地位的，其他运算可以忽略。





算法的计算量函数

- 利用计算量函数从理论上评估算法复杂性
- 算法复杂性是算法运行所需计算机资源的量。
 - 需要时间的→时间复杂性 (Time Complexity)
 - 需要空间的→空间复杂性 (Space Complexity)
- 反映算法的效率，并与运算计算机独立

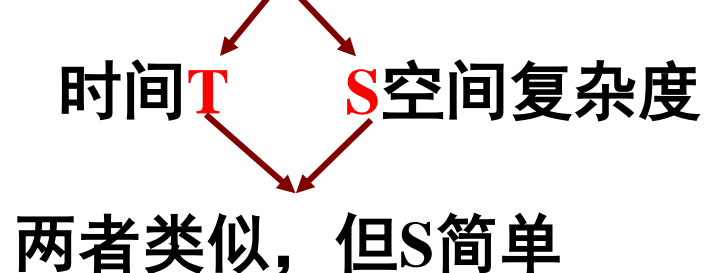


通常只考虑时间复杂性（本课程重点关注）
但是在大数据时代，空间复杂性也同样重要

$T(N, I, A)$ 的概念

- 计算量函数依赖于问题的规模(N), 输入(I)和算法(A)本身, 用 C 表示。

$C=F(N, I, A)$ 是一个三元函数。



例子: 利用插入排序对数组
{5, 7, 1, 3, 6, 2, 4}排序

$N \rightarrow 7$

$I \rightarrow$

5	7	1	3	6	2	4
---	---	---	---	---	---	---

$A \rightarrow$ `void insertSort(T[] a)`

$T(N, I, A)$ 的概念

- 计算量函数依赖于问题的规模(N), 输入(I)和算法(A)本身, 用 C 表示。

$C=F(N, I, A)$ 是一个三元函数。

能否将计算
量函数 $T(N,$
 $I, A)$ 简化



简化: 将 A 隐去

通常研究 $T(N, I)$ 在一台抽象计算机上运行所需时间

$T(N, I)$ 的概念

- 设抽象计算机的元运算有 k 种，记为 O_1, \dots, O_k ，每执行一次所需时间为 t_1, \dots, t_k 。
- 对算法A，用到元运算 O_i 的次数为 e_i 与 N, I 相关。

$$T(N, I) = \sum_{i=1}^k t_i e_i(N, I)$$

能否将计算
量函数 $T(N, I)$
简化



$T(N)$ 的概念

- 不可能规模 N 的每种合法输入 I 都去统计 $e_i(N, I)$, 对于 I 分别考虑:

最坏情况、最好情况、平均情况

$$T_{\max}(N) = \max_{I \in D_N} T(N, I) = \max_{I \in D_N} \sum_{i=1}^k t_i e_i(N, I) = \sum_{i=1}^k t_i e_i(N, I^*) = T(N, I^*)$$

$$T_{\min}(N) = \min_{I \in D_N} T(N, I) = \min_{I \in D_N} \sum_{i=1}^k t_i e_i(N, I) = \sum_{i=1}^k t_i e_i(N, \tilde{I}) = T(N, \tilde{I})$$

$$T_{\text{avg}}(N) = \sum_{I \in D_N} P(I) T(N, I) = \sum_{I \in D_N} P(I) \sum_{i=1}^k t_i e_i(N, I)$$

 合法输入集

$T(N)$ 的概念

- **进一步简化：**假设算法中用到的所有不同基本运算各执行一次需要的时间都是一个单位时间。
- 用输入规模的某个函数来表示算法的基本运算量，称为**算法的时间复杂性(度)**。

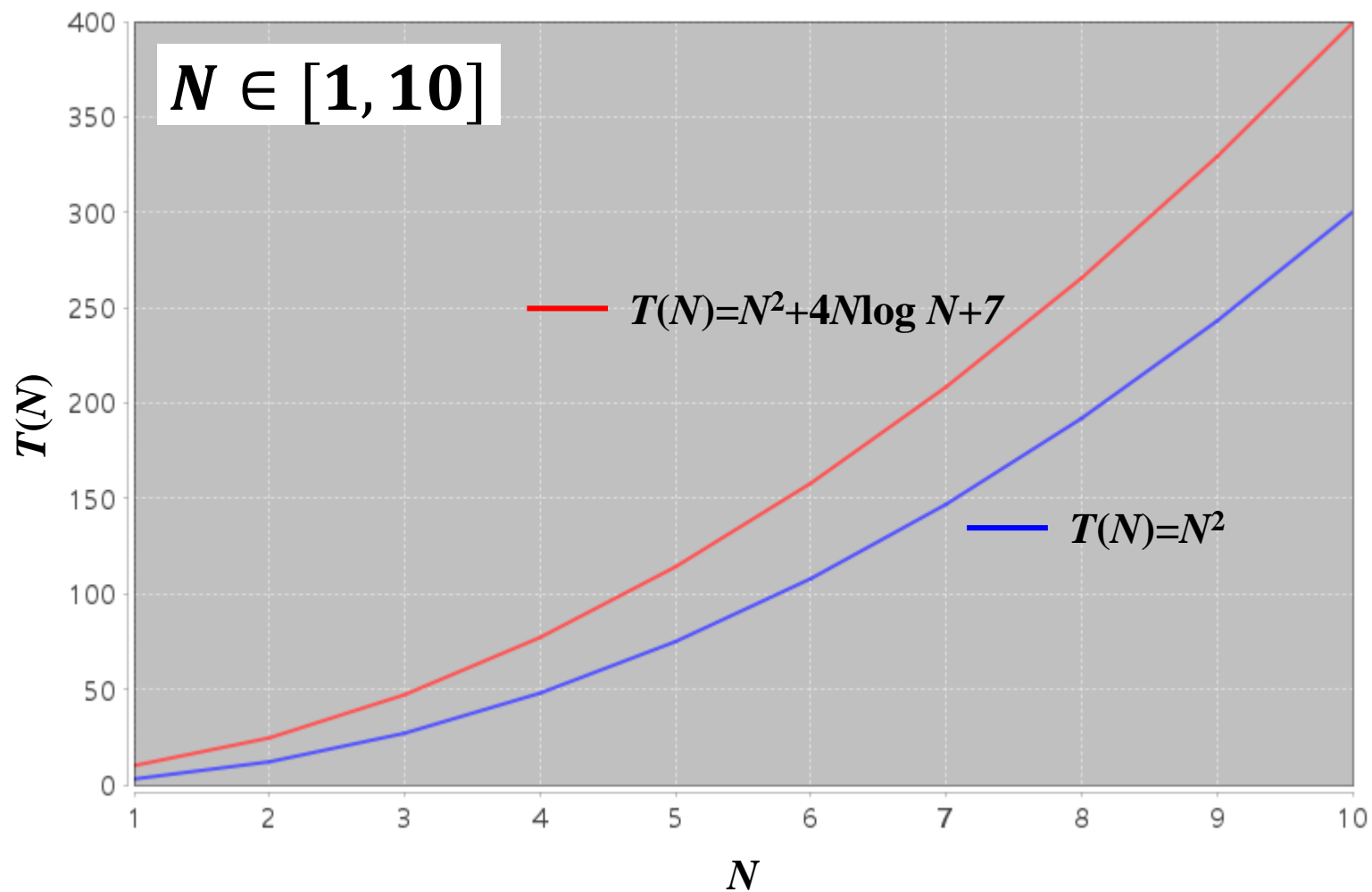
用 $T(N)$ 或 $T(N, M)$ 来表示，例如：

- $T(N)=5N+3$
- $T(N)=3N\log N+2N$
- $T(N)=4N^3+3N+2$
- $T(N)=2^N$
- $T(N, M)=2(N+M)$

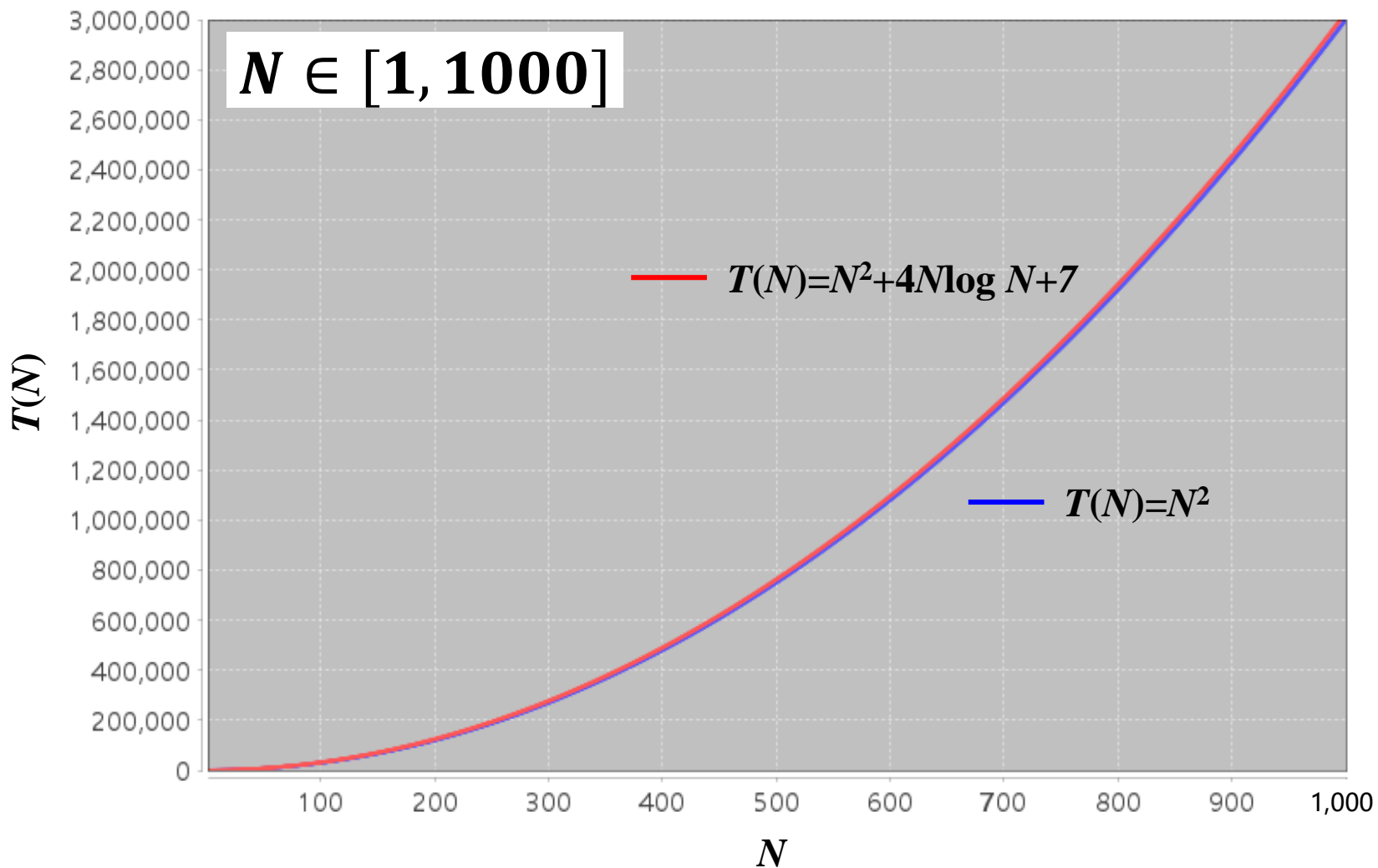
能否将计算
量函数 $T(N)$
简化



复杂性渐进性态



复杂性渐进性态





复杂性渐进性态

- 设 $T(N)$ 是前面定义的算法 A 复杂性函数。
 - N 递增到无限大, $T(N)$ 递增到无限大
 - 如存在 $\tilde{T}(N)$, 使 $N \rightarrow \infty$ 时, 有 $\frac{T(N) - \tilde{T}(N)}{T(N)} \rightarrow 0$
称 $\tilde{T}(N)$ 是 $T(N)$ 当 $N \rightarrow \infty$ 的渐进性态。
- 在数学上, $\tilde{T}(N)$ 是 $T(N)$ 当 $N \rightarrow \infty$ 的渐进表达式, 通常 $\tilde{T}(N)$ 是 $T(N)$ 中略去低阶项所留下的主项。
 - $\tilde{T}(N)$ 比 $T(N)$ 简单。



复杂性渐进性态

- 例如： $T(N) = 3N^2 + 4N \log N + 7$

$$\tilde{T}(N) = 3N^2$$

- 由 $\frac{T(N) - \tilde{T}(N)}{T(N)} = \frac{4N \log N + 7}{3N^2 + 4N \log N + 7} \rightarrow 0$

- 因为 $N \rightarrow \infty$, $T(N) \rightarrow \tilde{T}(N)$

- 所以有理由用 $\tilde{T}(N)$ 来替代 $T(N)$ 来度量A。



复杂性渐进性态

- 当比较两个算法的渐近复杂性的阶不同时，只要确定各自的阶，即可判定哪个算法效率高。

等价于

- 只要关心 $\tilde{T}(N)$ 的阶即可，不必考虑其中常数因子。
- 简化算法复杂性分析的方法和步骤，只要考察问题规模充分大时，算法复杂性在渐近意义下的阶。

练习：按照渐近阶从低到高的顺序排列以下表达式：

$$n!, 4n^2, \log n, 3^n, 20n, 2, n^{2/3}$$

复杂性渐进性态

- 当比较两个算法的渐近复杂性的阶不同时，只要确定各自的阶，即可判定哪个算法效率高。

等价于

- 只要关心 $\tilde{T}(N)$ 的阶即可，不必考虑其中常数因子。
- 简化算法复杂性分析的方法和步骤，只要考察问题规模充分大时，算法复杂性在渐近意义下的阶。

$$\tilde{T}(N) = \underbrace{3} N^2$$

是否可以去掉系数？

能否将计算
量函数 $\tilde{T}(N)$
简化





渐近分析的记号

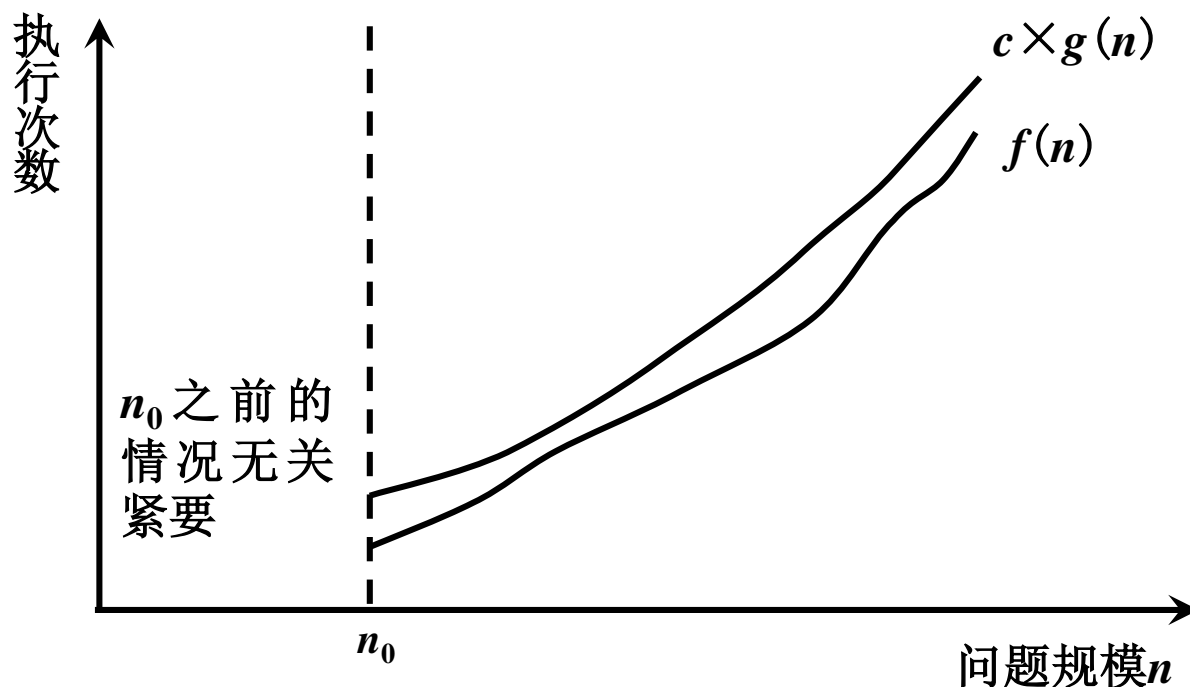
- 渐近上界记号 O
- 渐近下界记号 Ω
- 紧渐近界记号 Θ
- 非紧上界记号 o
- 非紧下界记号 ω

下面的讨论中，对所有 n ， $f(n) \geq 0$ ， $g(n) \geq 0$

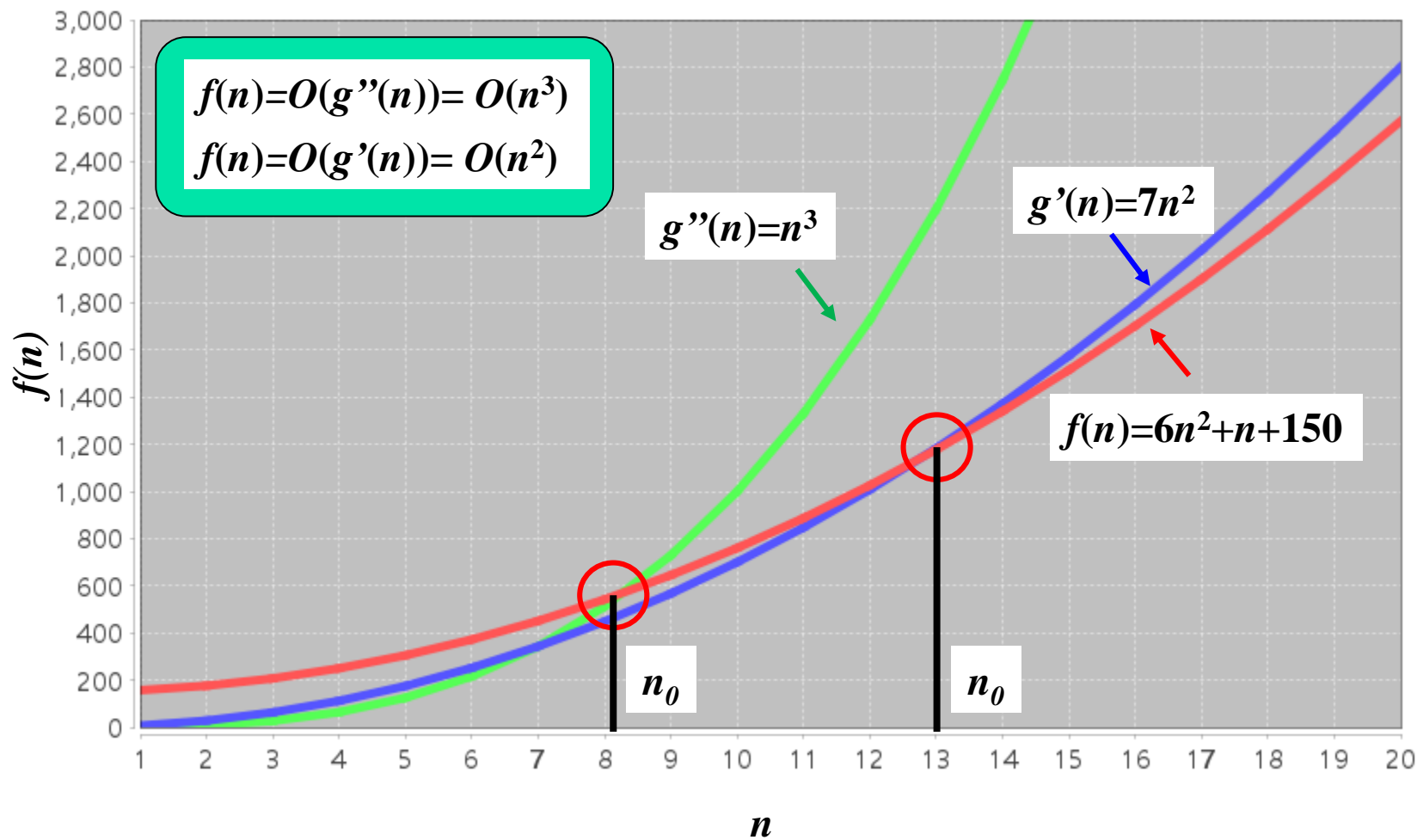
渐近上界记号O

■ 渐近上界记号O

- 若存在两个正的常数 c 和 n_0 ，使得对所有 $n \geq n_0$ ，都有： $f(n) \leq c \times g(n)$ ，则称 $f(n) = O(g(n))$



渐近上界记号O





渐近上界记号O

- 练习：求下列函数的渐近上界

- $3n^2 + 10n$

- $n^2/10 + 2^n$

- $21 + 1/n$

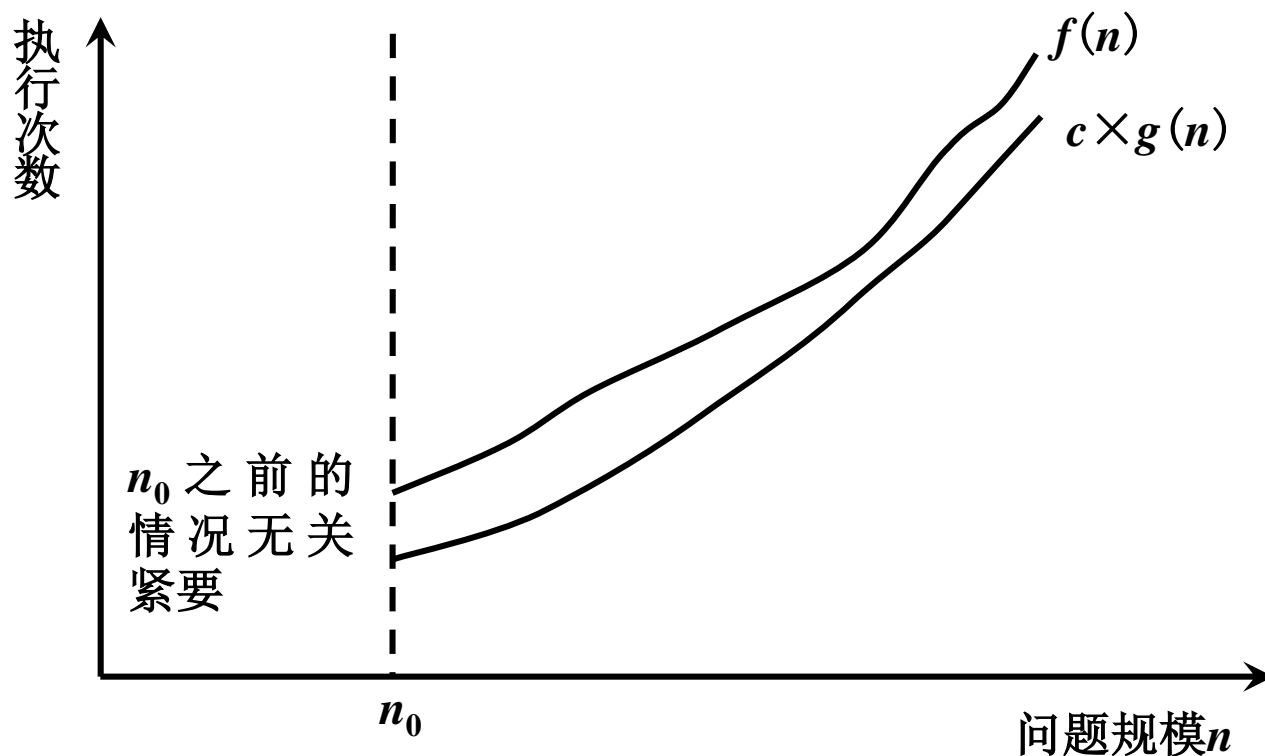
- $\log n^3$

- $10\log 3^n$

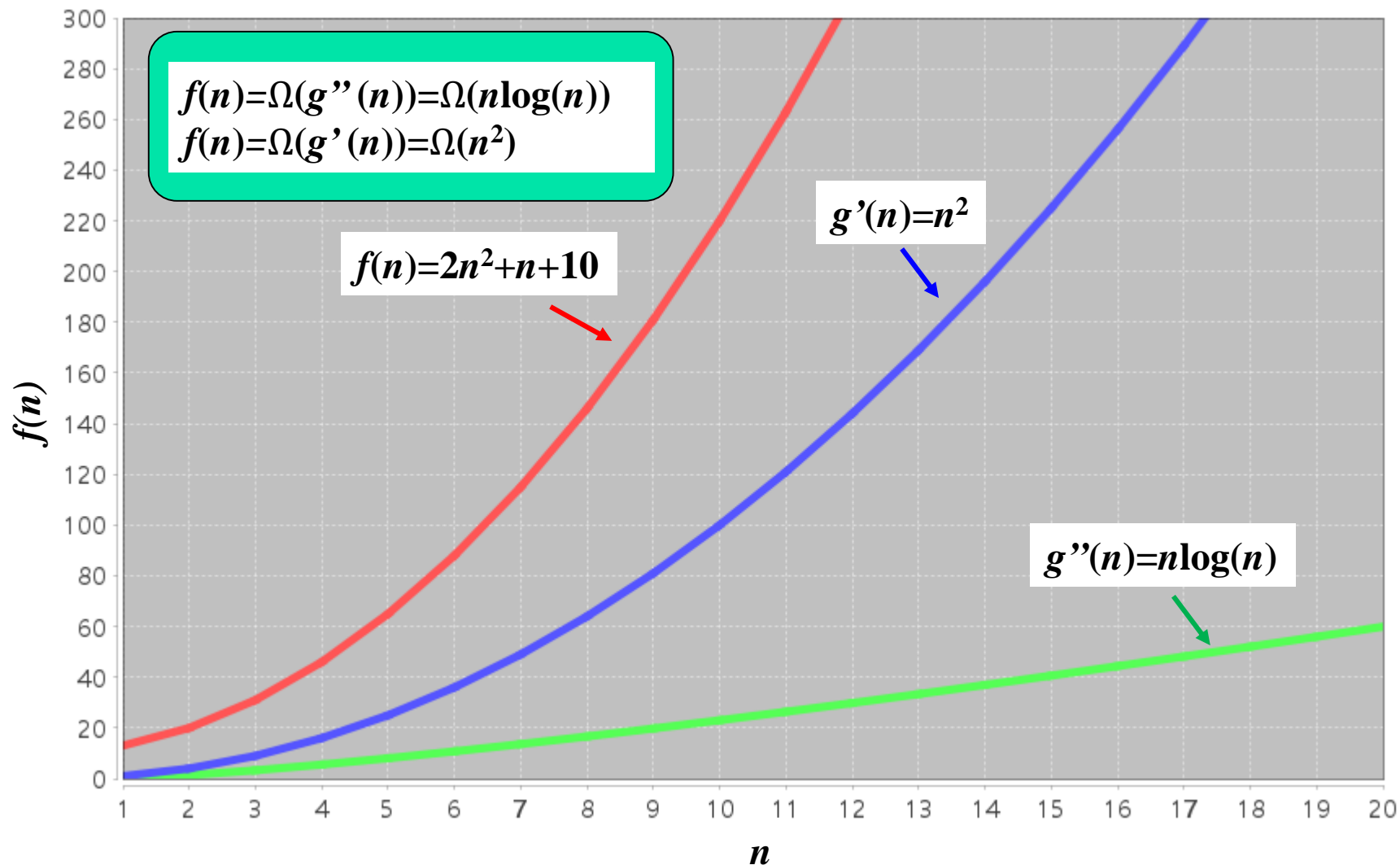
渐近下界记号 Ω

■ 渐近下界记号 Ω

- 若存在两个正的常数 c 和 n_0 ，使得对所有 $n \geq n_0$ ，都有： $f(n) \geq c \times g(n)$ ，则称 $f(n) = \Omega(g(n))$



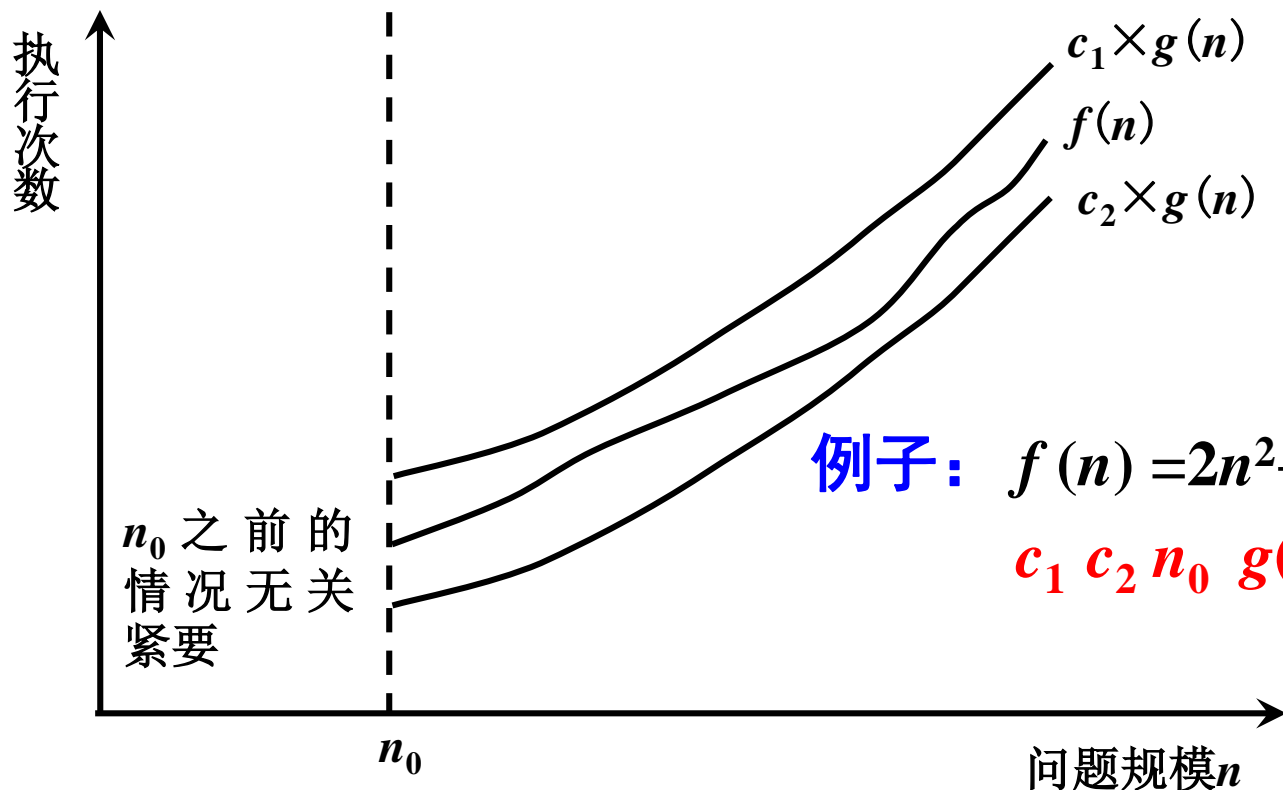
渐近分析的记号



紧渐近界记号 Θ

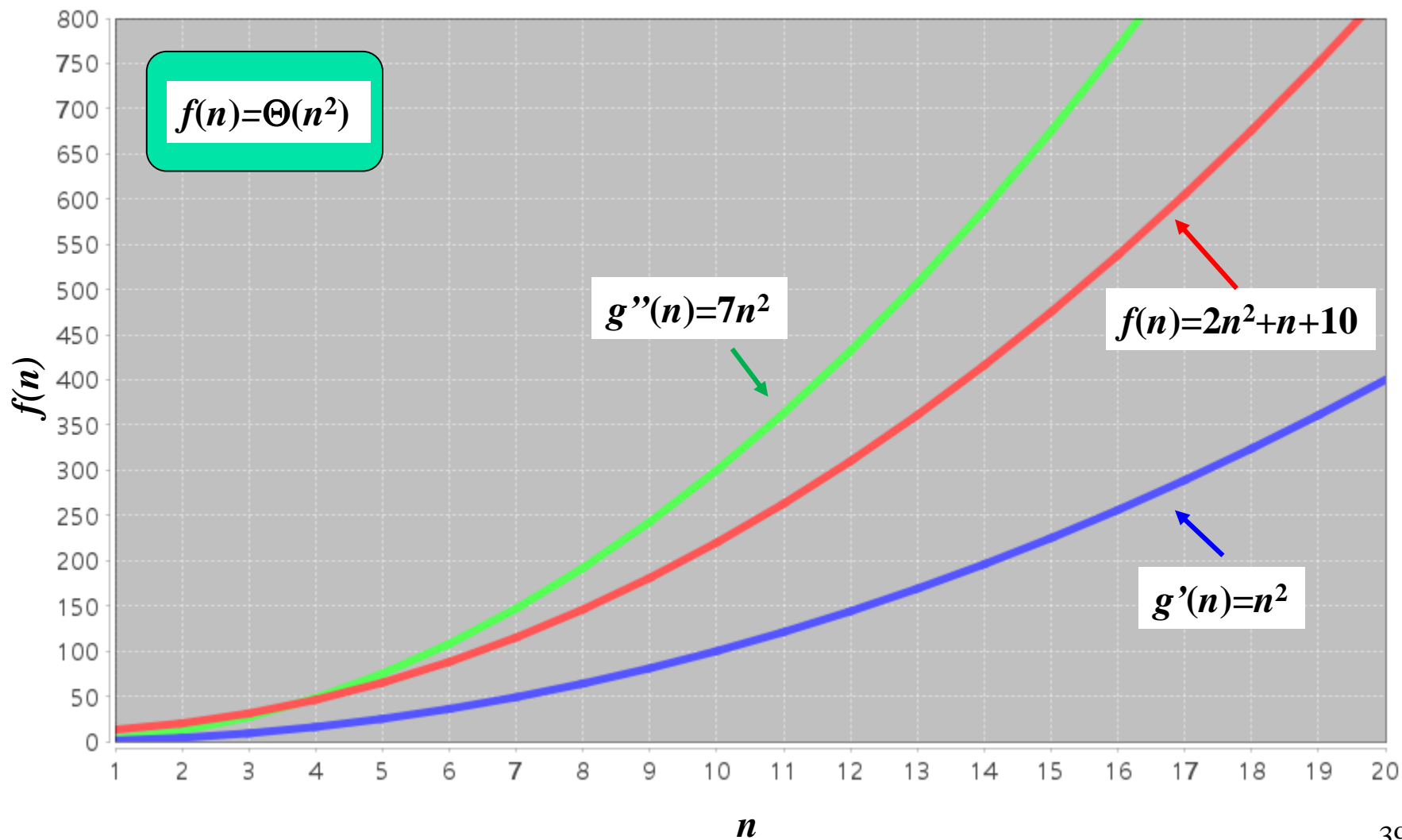
■ 紧渐近界记号 Θ

- 若存在三个正的常数 c_1 、 c_2 和 n_0 ，使得对所有 $n \geq n_0$ ，都有： $c_1 \times g(n) \geq f(n) \geq c_2 \times g(n)$ ，则称 $f(n) = \Theta(g(n))$



例子： $f(n) = 2n^2 + n + 10$
 $c_1 \ c_2 \ n_0 \ g(n) = ?$

紧渐近界记号 Θ



非紧上/下界记号

■ 非紧上界记号 o

- $o(g(n)) = \{ f(n) \mid \text{对于任何正常数 } c > 0, \text{ 存在正数 } n_0 > 0 \text{ 使得对所有 } n \geq n_0 \text{ 有: } 0 \leq f(n) < cg(n) \}$
- 等价于 $f(n) / g(n) \rightarrow 0$, as $n \rightarrow \infty$ 。

■ 非紧下界记号 ω

- $\omega(g(n)) = \{ f(n) \mid \text{对于任何正常数 } c > 0, \text{ 存在正数 } n_0 > 0 \text{ 使得对所有 } n \geq n_0 \text{ 有: } 0 \leq cg(n) < f(n) \}$
- 等价于 $f(n) / g(n) \rightarrow \infty$, as $n \rightarrow \infty$ 。
- $f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$

例：渐近意义下的 O

- 如果存在正的常数 C 和自然数 N_0 ，使得当 $N \geq N_0$ 时，有 $f(N) \leq C \times g(N)$ ，则称函数 $f(N)$ 当 N 充分大时上有界，且 $g(N)$ 是它的一个上界，记为 $f(N) = O(g(N))$ 。
- 也即 $f(N)$ 的阶不高于 $g(N)$ 的阶。
 - $\forall N \geq 1, 3N \leq 4N \Rightarrow 3N = O(N)$ ；
 - $\forall N \geq 1, N + 1024 \leq 1025N \Rightarrow N + 1024 = O(N)$ ；
 - $\forall N \geq 10,$
 $2N^2 + 11N - 10 \leq 3N^2 \Rightarrow 2N^2 + 11N - 10 = O(N^2)$ ；
 - $\forall N \geq 1, N^2 \leq N^3 \Rightarrow N^2 = O(N^3)$ ；
 - $N^3 \neq O(N^2)$ ，无 $N \geq N_0$ 使得 $N^3 \leq CN^2 \quad N \leq C$ ；

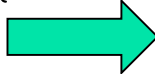
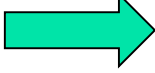
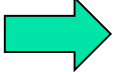
反例



O 的运算性质

- $O(f) + O(g) = O(\max(f, g))$
- $O(f) + O(g) = O(f + g)$
- $O(f) \cdot O(g) = O(f \cdot g)$
- 如果 $g(N) = O(f(N)) \Rightarrow O(f) + O(g) = O(f)$
- $O(cf(N)) = O(f(N))$ 其中 c 是一个正的常数
- $f = O(f)$

冒泡排序复杂度 O

```
void bubbleSort(T[] a) {  
    int len = a.length;  只做1次  
    for (int i=0; i<n; i++) {  n次循环  
        for (int j=0; j<n-i-1; j++) {  最多n-1次内循环  
            if (a[j]>a[j+1]) {  
                int temp = a[j];  
                a[j] = a[j+1];  
                a[j+1] = temp;  
            }  
        }  
    }  
}
```

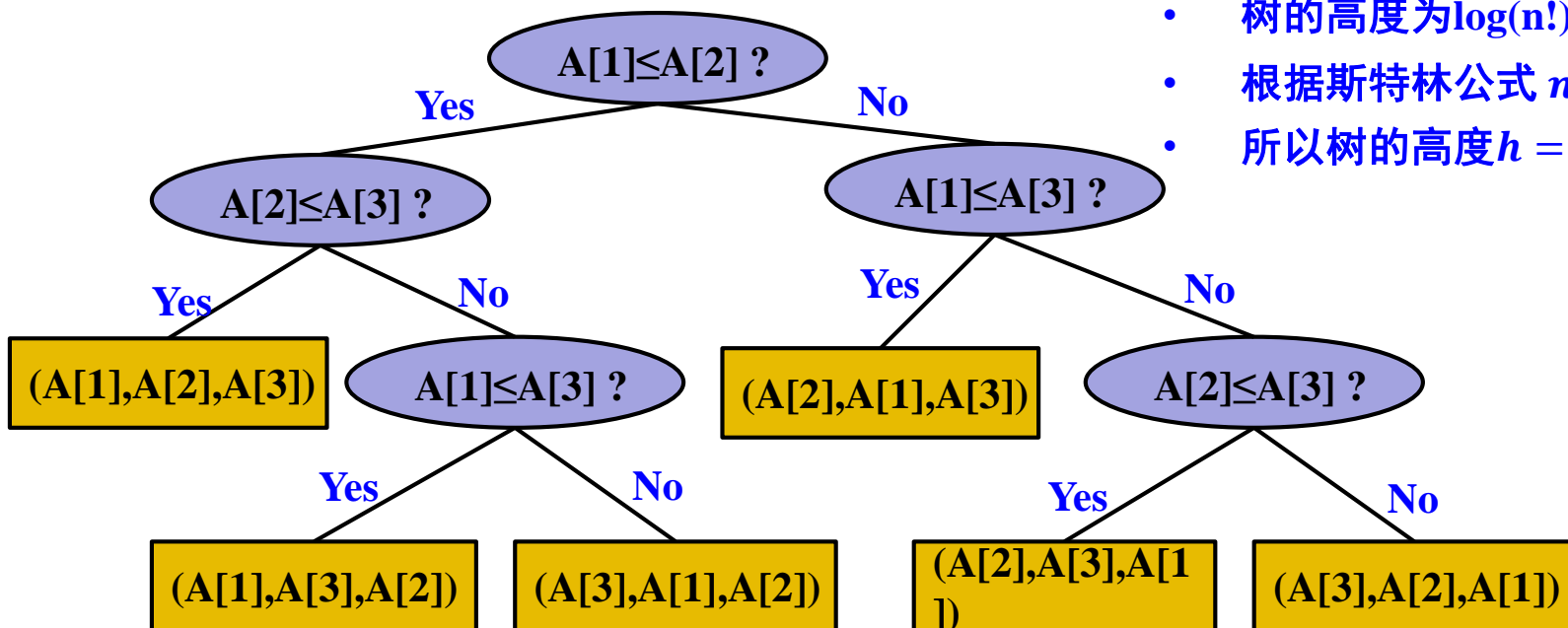
$O(n^2)$

基于比较的排序算法的 Ω

对于任意大的 n ，任何比较排序算法在最坏情况下至少需要 $c \cdot n \log n$ 次比较操作。所以，时间复杂度的下界为 $\Omega(n \log n)$ 。

可以利用决策树证明

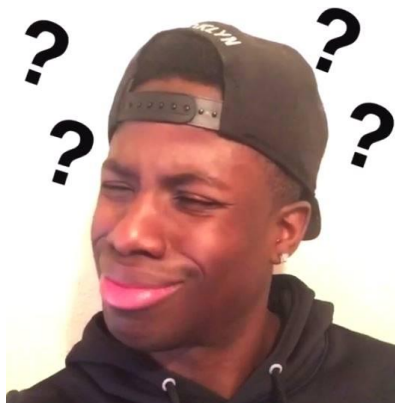
- 树有 $n!$ 个叶子
- 树的高度为 $\log(n!)$
- 根据斯特林公式 $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$
- 所以树的高度 $h = \Omega(n \log n)$



问题的计算复杂度分析

■ 问题

- 哪个排序算法效率最高? 快速排序的平均时间复杂度最好
- 是否可以找到更好的排序算法? 比较排序时间复杂度下界为 $\Omega(n \log n)$
- 排序问题计算难度如何?
- 问题计算复杂度的估计方法 伪代码中基本运算执行的次数



哪个排序算法效率最高?
如何分析排序问题计算难度?

NP完全性理论



重要的问题类型

- **查找/检索问题**
 - 在给定的集合中寻找一个给定的值
- **排序问题**
 - 按升序（降序）重新排列给定列表中的数据项
- **图问题**
 - 图的遍历、最短路径以及有向图的拓扑排序
- **组合问题**
 - 寻找一个组合对象（排列或子集）满足一些重要特性
- **几何问题**
 - 处理类似于点、线、多面体这样的几何对象

易解问题与难解问题

- 通常将存在**多项式时间**算法的问题看作是**易解问题**（Easy Problem）；
 - 排序问题、查找问题
 - 而将需要**指数时间**算法解决的问题看作是**难解问题**（Hard Problem）。
 - 旅行商问题、图着色问题
- 问题规模

难解问题的例子：旅行商问题

■ 明星演唱会

- **地点：**北京、上海、广州、深圳、南京、杭州、武汉、长沙、成都、重庆
- **线路：**从北京出发，跑遍各大城市，回到北京

票价	北京	上海	广州
北京	0	500	600
上海	100	0	800
广州	1000	200	0
.....

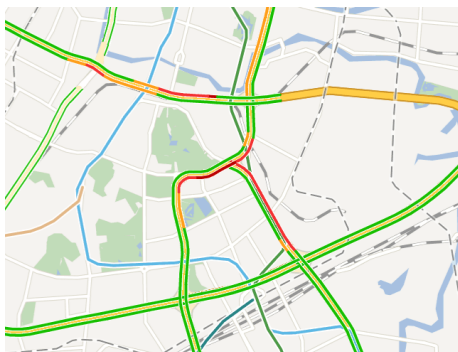
- **目标：**考虑机票价格，确定票价最少的线路

难解问题的例子：旅行商问题

■ 问题定义

- 城市集合： $C = \{c_1, c_2, \dots, c_n\}$
- 城市距离： $d(c_i, c_j)$
- 距离不对称： $d(c_i, c_j) \neq d(c_j, c_i)$

■ 目标：求遍历所有城市（不重复）的最短路径



道路拥堵情况下的
送快递问题



考虑城市单行线
的送快递问题



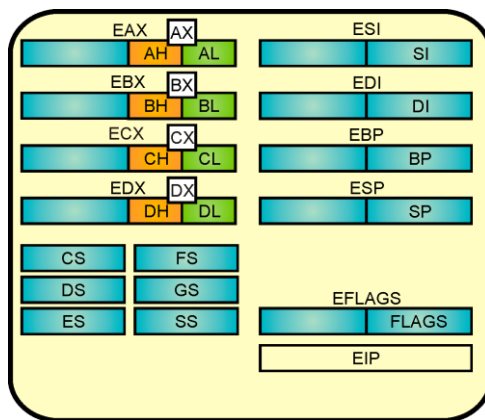
全国巡回演唱会的
路线安排问题

难解问题的例子：图着色问题

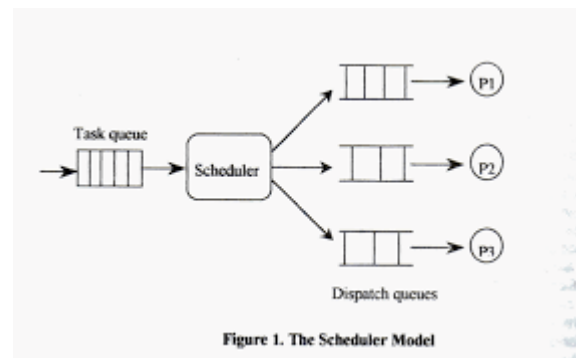
- 给定无向连通图 $G=(V,E)$ ，求图 G 的最小色数 k ，使得用 k 种颜色对 G 中顶点着色，可使任意两个顶点着色不同。
 - k 个颜色的集合为{颜色1，颜色2，...，颜色 k }。



地图着色

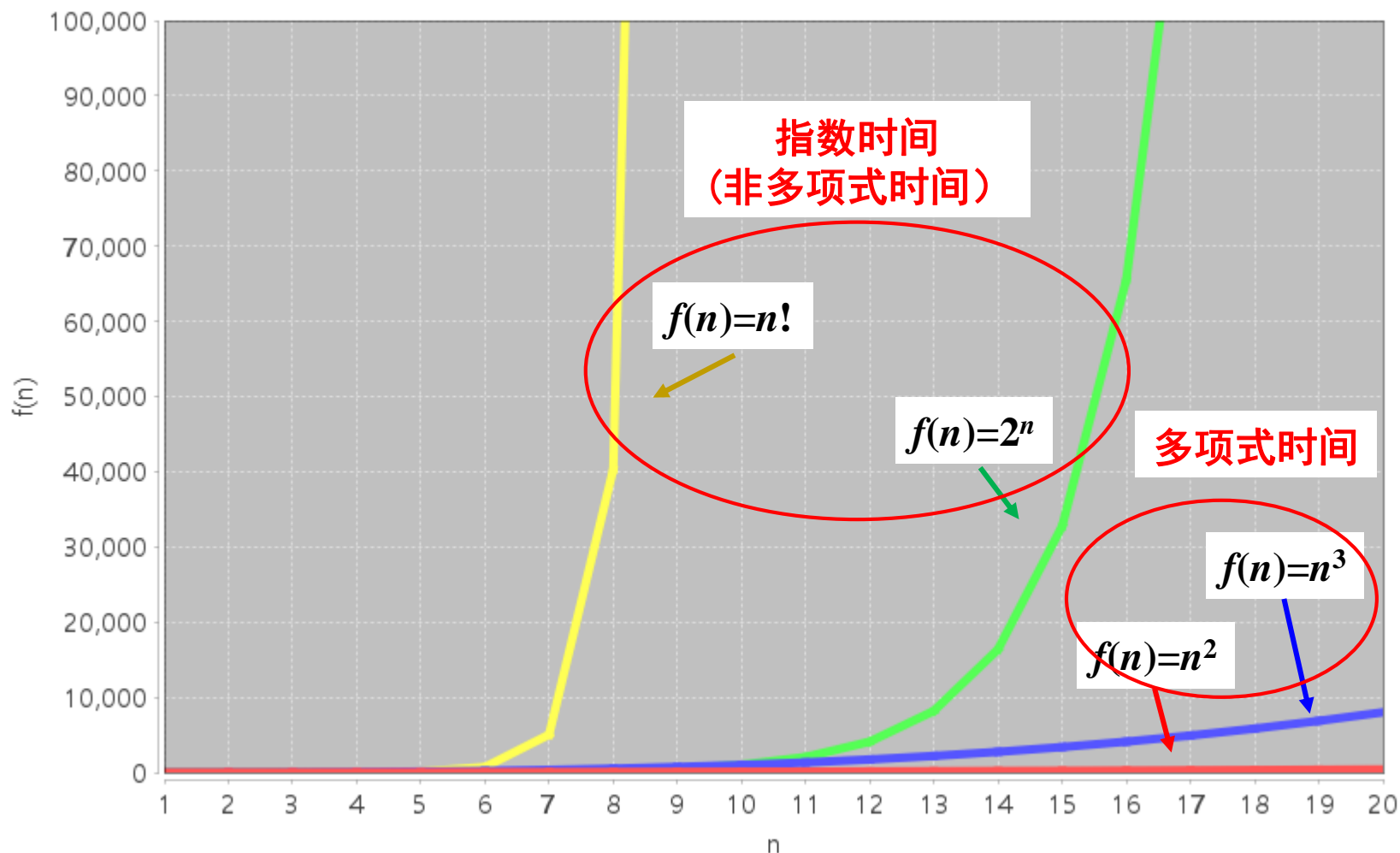


程序编译器的
寄存器分配算法



任务调度

易解问题与难解问题



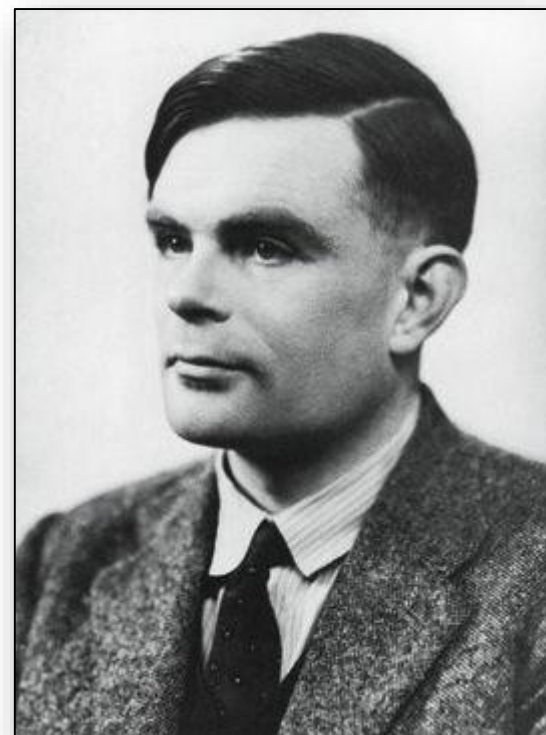


易解问题与难解问题

- 为什么把多项式时间复杂性作为易解问题和难解问题的分界线？
 - 多项式函数与指数函数的增长率有本质的差别
 - 计算机性能的提高对多项式时间算法和指数时间算法的影响不同

不可解问题：图灵停机问题

- 希尔伯特的可判定性问题
 - 是否存在一种通用的机械过程，能够判定任何数学命题的真假
- 图灵的方法
 - 设计一个图灵机覆盖所有的“机械过程”，如果存在一个问题，图灵机判定不了，那么就说明，不存在这种“通用的”过程，这样就证明了原问题。



是否存在这样的图灵机呢？答案是否！

不可解问题：图灵停机问题

```
bool God_algo(char* program, char* input)//程序是program, 输入是input
{
    if( <program> halts on <input> ) //如果 (if) 程序 (program) 能够通过
                                     //输入 (input) 判断停机 (halt)
        return true; //这个程序判断出来了
    return false; //这个程序不能判断出来
}
```

```
bool Satan_algo(char* program) //继续判断是否有这样一个程序program
{
    if( God_algo(program, program) ) //使用之前那个万能算法检查成功了
    {
        while(1); //永远循环下去!
        return false; //这个地方永远执行不到, false是与下方true匹配
    }
    else
        return true; //万能程序没有检查出来
}
```

执行Satan_algo(Satan_algo)会发生什么呢，是否停机？



P类问题和NP类问题

- 判定问题
- 确定性算法与P类问题
- 非确定性算法与NP类问题



P类和NP类问题的主要差别

- P类问题可以用多项式时间的**确定性算法**来进行判定或求解；
- NP类问题可用多项式时间的**非确定性算法**来进行判定或求解。

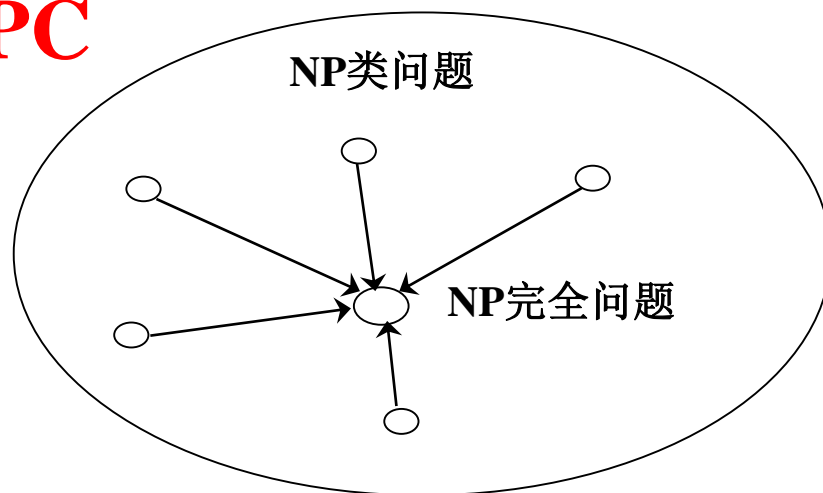
判定问题 (Decision Problem)

- 仅仅要求回答 “yes”或 “no”的问题
- 判定问题的重要特性——证比求易

$$P \subseteq NP$$

NP完全问题

- 令 Π 是一个判定问题，如果问题 Π 属于 NP 类问题，并且对 NP 类问题中的每一个问题 Π' ，都有可以在多项式时间内将 Π' 规约到 Π ，则称问题 Π 是一个 **NP 完全问题 (NP Complete Problem)**，有时把 NP 完全问题记为 **NPC**

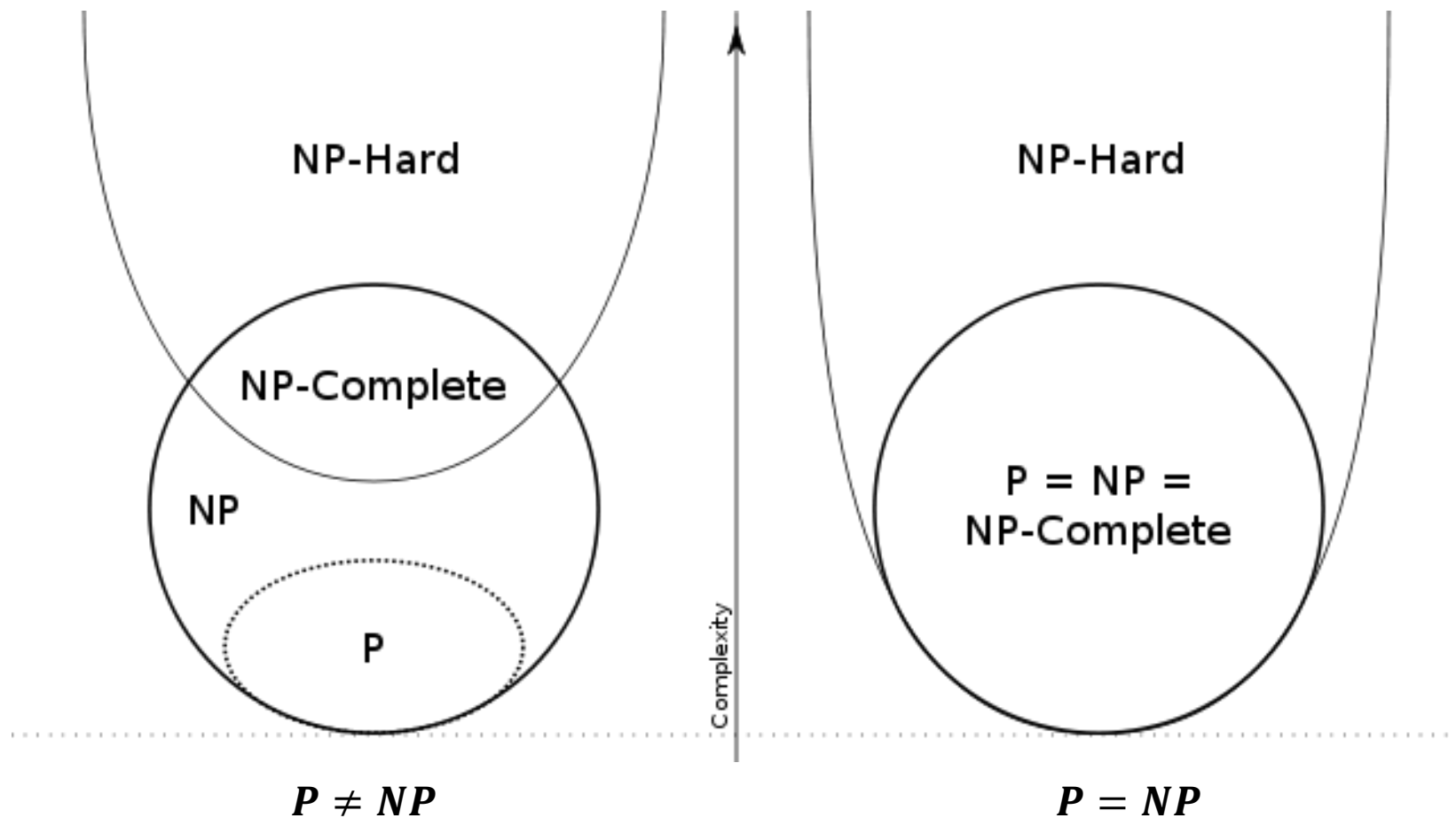




一些基本的NP完全问题

- SAT问题(Boolean Satisfiability Problem)
- **最大团问题(Maximum Clique Problem)**
- **图着色问题(Graph Coloring Problem)**
- 哈密顿回路问题(Hamiltonian Cycle Problem)
- **TSP问题(Traveling Salesman Problem)**
- **顶点覆盖问题(Vertex Cover Problem)**
- 最长路径问题(Longest Path Problem)
- 子集和问题(Sum of Subset Problem)

三者关系





小结

- 算法效率和算法复杂性的关系
- 算法复杂性的概念
- 算法复杂性的渐进符号
- 多项式时间和非多项式时间
- P 问题、NP问题、NP完全问题
- 经典的NP完全问题