数据分析与算法设计
第2章 算法效率分析基础

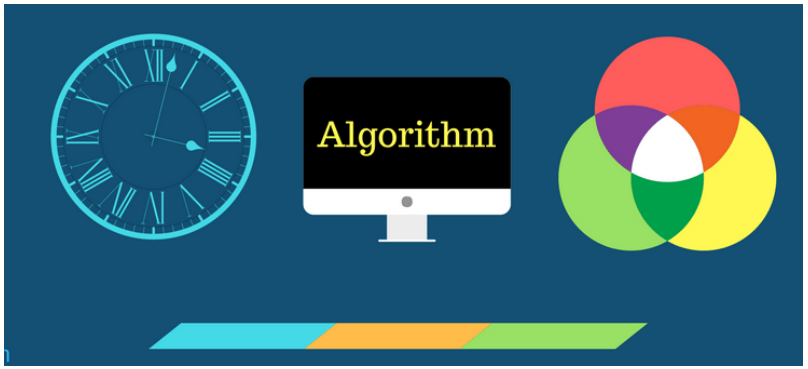
李旻
百人计划研究员
浙江大学 信息与电子工程学院
Email: min.li@zju.edu.cn

目录

- 算法效率分析框架
  - 时间效率
  - 空间效率
- 算法效率的渐进表征
  - 渐进符号
  - 基本效率类型
- 非递归算法的效率分析
- 递归算法的效率分析

效率分析框架

- 典型的算法效率
  - 时间效率：算法运行速度快慢的度量(时间复杂度)
  - 空间效率：算法运行存储空间大小的度量(空间复杂度)



我常常说，当你对所讲的内容能够进行度量并能够用数字来表达的时候，证明你对这些内容是有所了解的。如果你不能用数字来表达，表明你的认识是不完整的，也是无法令人满意的：无论它是什么内容，它也许正处于知识的初级阶段，但在你的思想中，几乎从没把它上升到一个科学的高度。

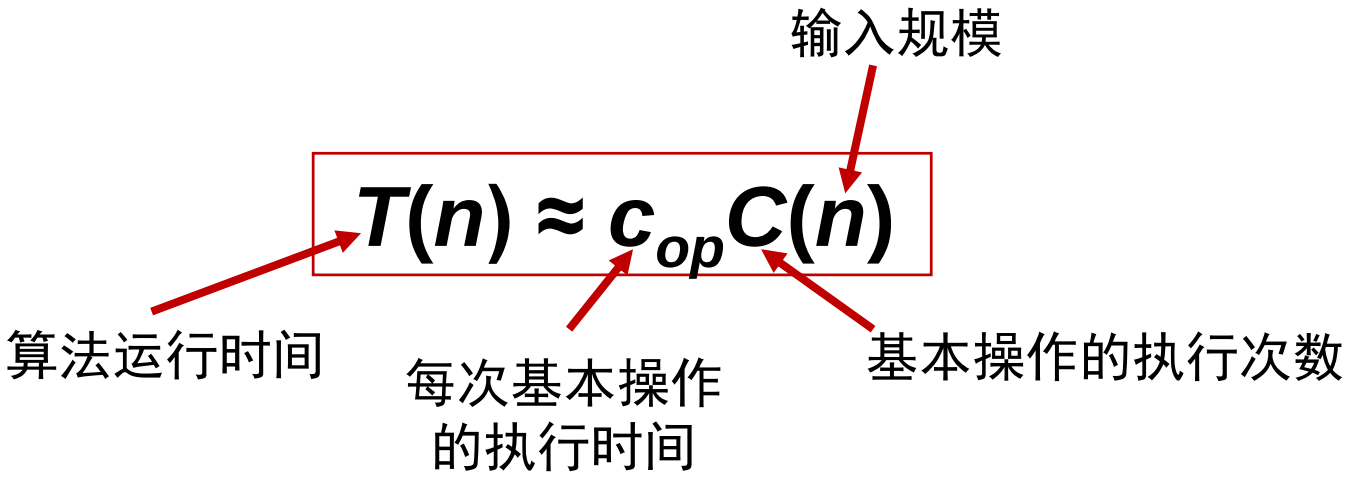
——开尔文爵士(1824—1907)

时间效率

- 时间效率是算法输入规模的函数
  - 常识：输入规模越大，算法运行时间越长
  - 选取算法关键参数作为输入规模的度量
    - 排序、查找等与列表相关的算法：输入规模为列表长度n
    - 两个矩阵相乘：输入规模为矩阵的维度或者元素的个数
    - 有向图相关的算法：输入规模为顶点及边的数目
- 输入规模会影响运行时间，但是仍没有给出具体的关于运行时间的定量刻画。而如果直接采用时间标准单位（如秒、毫秒等）来对算法进行计时，这种方法也存在明显的缺陷：它依赖于特定计算机的速度、程序实现的质量等因素，而且对于程序的实际时间进行精确计时也是困难的

时间效率

- 对于输入规模为n的算法，可采用算法的基本操作执行次数，来对其效率进行度量
  - 基本操作：算法中最重要的、对总运行时间的贡献最大的操作
    - 排序：比较
    - 矩阵乘法：乘法
    - 图问题：边或节点的访问



时间效率

- 但是，很多算法的基本操作执行次数不仅取决于输入的规模，而且还取决于输入的细节
- 举例：顺序查找

//在一个指定的数组中顺序查找指定元素
//输入：A[0..n-1],*K*
//输出：指定查找元素在数组中的下标，没有返回-1

```
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

  - 最坏情况：n
  - 最好情况：1
  - 平均情况：(1+n)p/2+n(1-p)　（假定p为成功查找的概率）

时间效率

- 可进一步区分算法的最优、最差和平均效率
  - 最优效率：算法在输入对应最小C(n)情况下的效率
  - 最差效率：算法在输入对应最大C(n)情况下的效率
  - 平均效率：算法在考虑所有输入情况下的平均效率
    - 将规模为n的实例划分为几种类型，同种类型所需要的基本操作执行次数一样
    - 给定各类输入的概率分布，以此推导平均执行次数

- 平均效率分析最有意义，但也是最难的

举例与思考

- 对于以下每种算法，请指出（i）其输入规模的合理度量标准；（ii）它的基本操作；（iii）对于规模相同的输入来说，其基本操作的次数是否会有不同？

① 计算n个数的和
② 计算n！
③ 找出包含n个数字的列表的最大元素
④ 欧几里得法求公约数

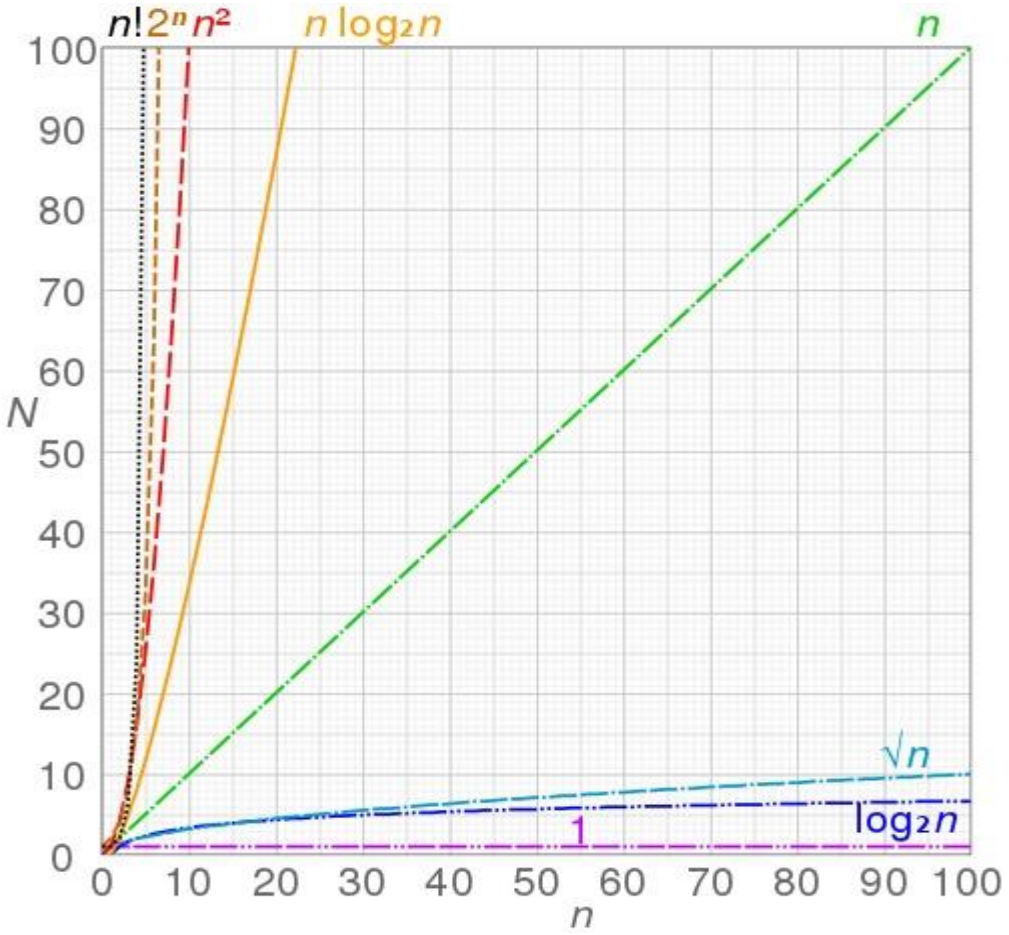
执行次数的增长次数

- 小规模输入在运行时间上的差别不足以将高效的算法和低效的算法区分开来
- 当输入规模n增长时，基本操作执行次数的增长次数更好地反映算法的优劣

典型C(n)函数

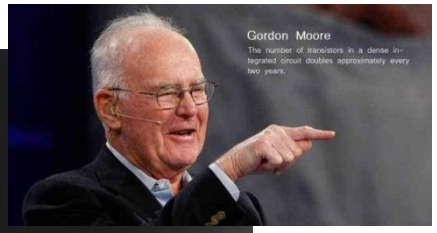
n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3\cdot10^1$	10^2	10^3	10^3	$3.6\cdot10^6$
10^2	6.6	10^2	$6.6\cdot10^2$	10^4	10^6	$1.3\cdot10^{30}$	$9.3\cdot10^{157}$
10^3	10	10^3	$1.0\cdot10^4$	10^6	10^9		
10^4	13	10^4	$1.3\cdot10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7\cdot10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0\cdot10^7$	10^{12}	10^{18}		

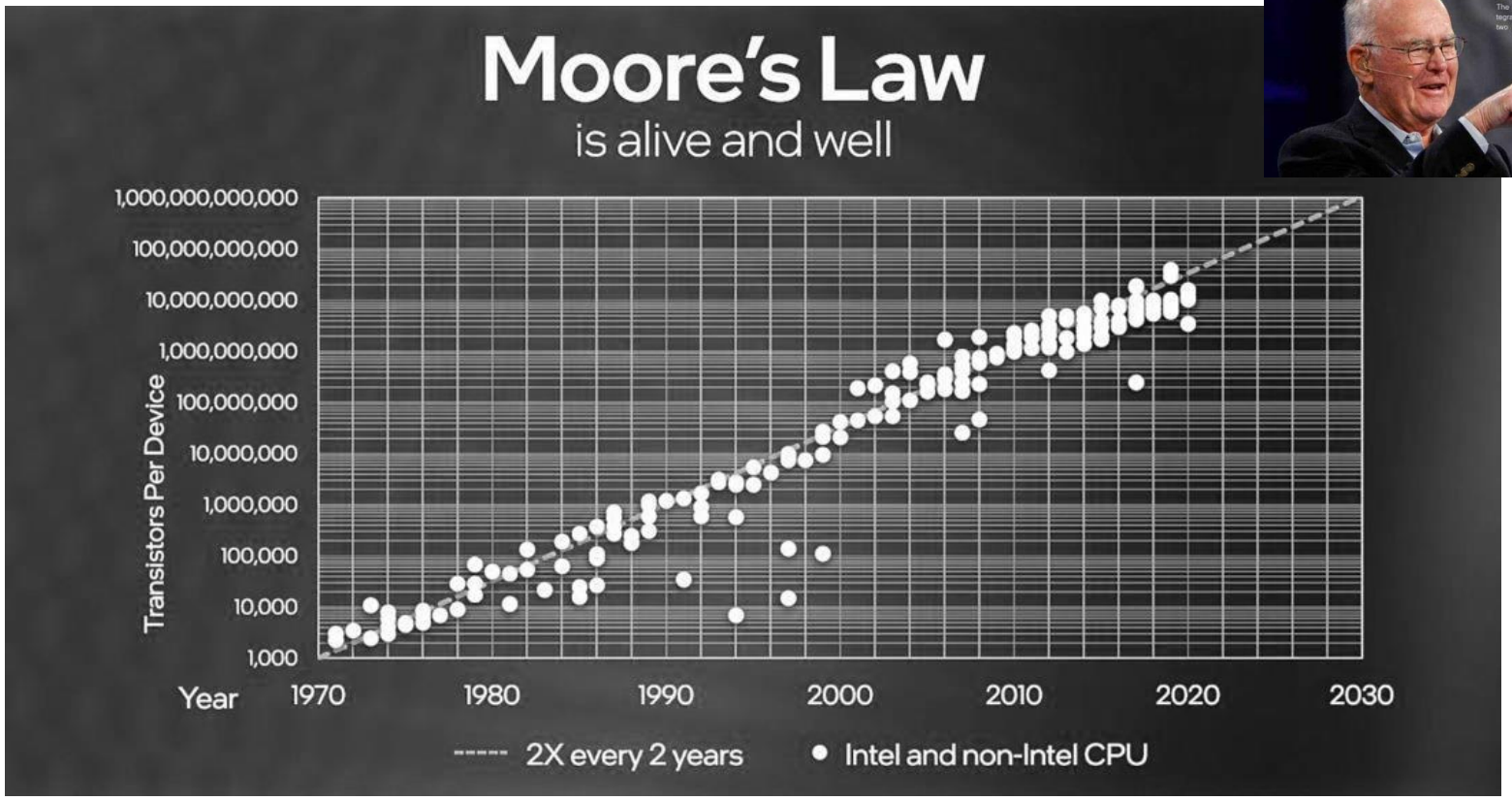
执行次数的增长次数



一个需要指数级操作次数的算法只能用来解决规模非常小的问题！ ☹

芯片发展：摩尔定律





计算芯片的持续发展可加速大规模算法的实现与执行！

目录

- 算法效率分析框架
  - 时间效率
  - 空间效率
- 算法效率的渐进表征
  - 渐进符号
  - 基本效率类型
- 非递归算法的效率分析
- 递归算法的效率分析

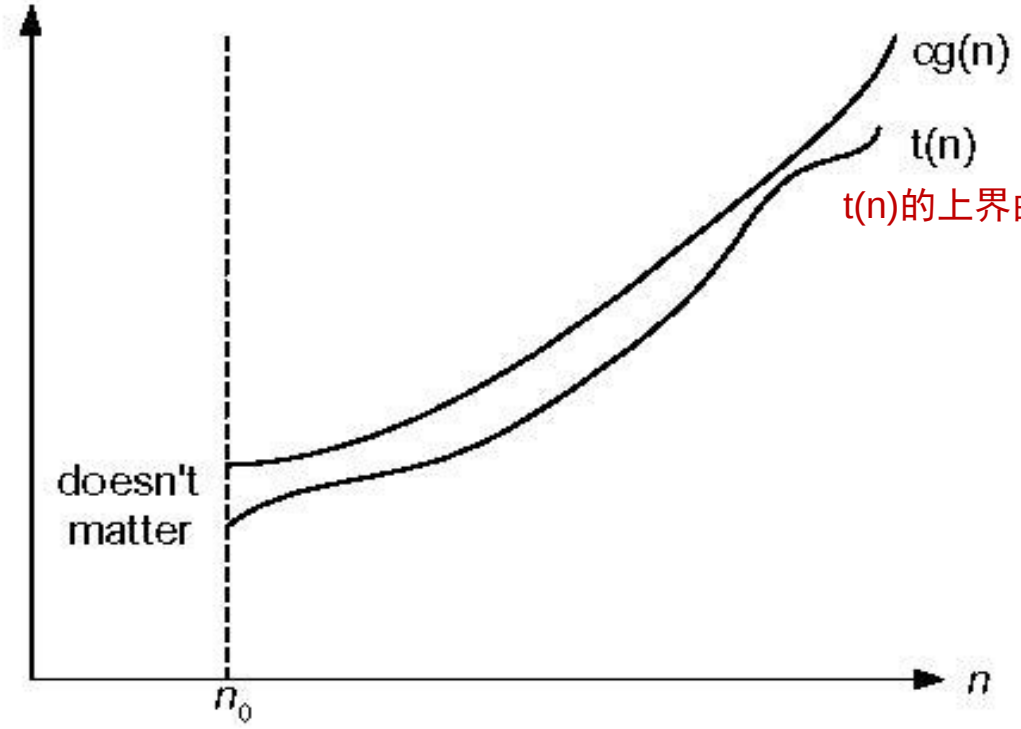
算法效率的渐进表征

- 渐进表征
  - 算法执行次数随着输入规模的增长而增长的速度
  - 可用于区分和比较不同类型的算法效率

- 渐进符号

符号	含义
O(g(n))	增长次数小于等于cg(n)的函数集合（c是正常数）
Ω(g(n))	增长次数大于等于cg(n)的函数集合（c是正常数）
Θ(g(n))	增长次数介于c_1g(n)和c_2g(n)间的函数集合（常数$c_1 > c_2 > 0$）

渐进符号：O(g(n))

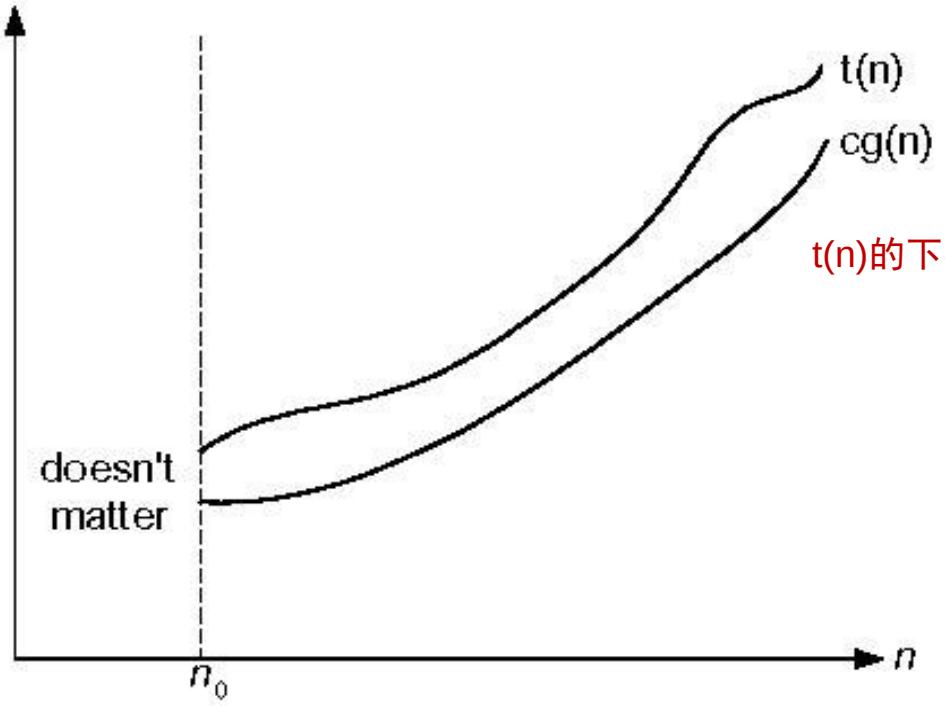


t(n)的上界由g(n)的常数倍所确定

$t(n) \in O(g(n))$

数学描述：对于足够大的$n>n_0$，存在常数$c>0$，使得$t(n) \leq cg(n)$

渐进符号：Ω(g(n))

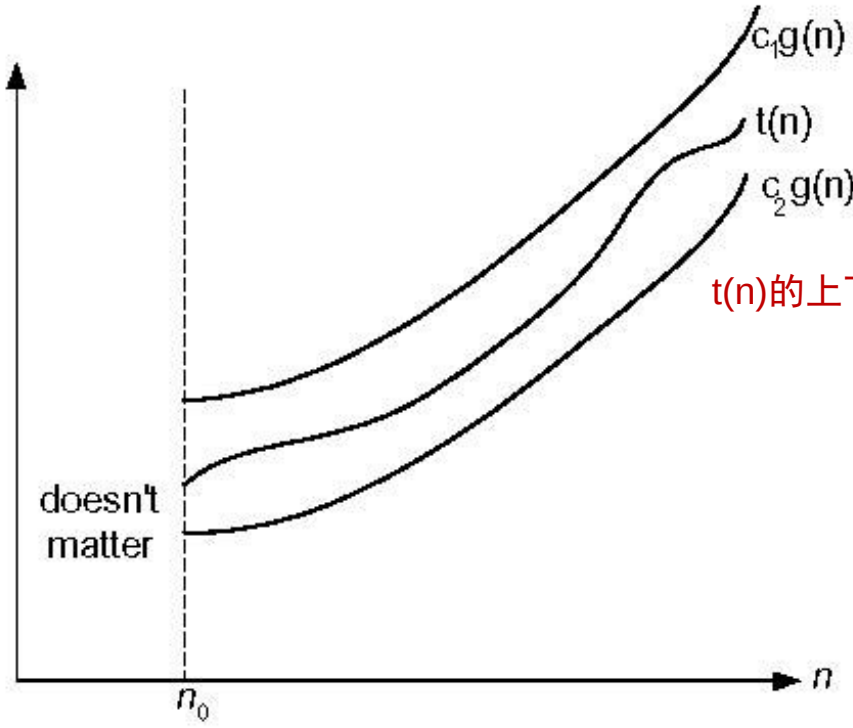


t(n)的下界由g(n)的常数倍所确定

$t(n) \in \Omega(g(n))$

数学描述：对于足够大的$n>n_0$，存在常数$c>0$，使得$t(n) \geq cg(n)$

渐进符号：Θ (g(n))



t(n)的上下界都由g(n)的常数倍所确定

t(n) ∈ Θ(g(n))

数学描述：对于足够大的n>n_0，存在常数$c_1 > c_2 > 0$，使得c_2g(n) ≤ t(n) ≤ c_1g(n)

举例与思考

- 以下的结论是否正确？

a. $n(n+1)/2 \in O(n^3)$

b. $n(n+1)/2 \in O(n^2)$

c. $n(n+1)/2 \in \Theta(n^3)$

d. $n(n+1)/2 \in \Omega(n)$

渐进符号的部分特性

① $t(n) \in O(g(n)) \Longrightarrow c*t(n) \in O(g(n))$

② $t(n) \in \Omega(g(n)) \Longrightarrow g(n) \in O(t(n))$

③ $t(n) \in O(f(n))$ & $f(n) \in O(g(n))$

$\Longrightarrow t(n) \in O(g(n))$

④ $t_1(n) \in O(g_1(n))$ & $t_2(n) \in O(g_2(n))$

$\Longrightarrow t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

⑤ $O(t(n))\cdot O(g(n)) = O(t(n)\cdot g(n))$

不同算法的增长次数比较

- 假设t(n)和g(n)是两个不同的增长次数函数，则它们比值的极限存在以下几种情况：

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0, & \Rightarrow t(n)\in O(g(n))\\ c>0, & \Rightarrow t(n)\in \Theta(g(n))\\ \infty, & \Rightarrow t(n)\in \Omega(g(n))\end{cases}$$

- 可利用强大的微积分技术来计算上述极限

洛必达法则

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

应用示例

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}. \Longrightarrow \frac{1}{2}n(n-1)\in\Theta(n^2)$$

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=2\log_2 e\lim_{n\to\infty}\frac{1}{\sqrt{n}}=0. \Longrightarrow \log_2 n\in o(\sqrt{n})$$

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty. \Longrightarrow n!\in\Omega(2^n)$$

史特林公式

$$n!\approx\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for large values of } n$$

基本的效率类型

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

类　型	名　称	注　　释
1	常量	为数很少的效率最高的算法，很难举出几个合适的例子，因为典型情况下，当输入的规模变得无穷大时，算法的运行时间也会趋向于无穷大
$\log n$	对数	一般来说，算法的每一次循环都会消去问题规模的一个常数因子(参见 4.4 节)。注意，一个对数算法不可能关注它的输入的每一个部分(哪怕是输入的一个固定部分)：任何能做到这一点的算法最起码拥有线性运行时间
n	线性	扫描规模为 n 的列表(例如，顺序查找)的算法属于这个类型
$n \log n$	线性对数	许多分治算法(参见第 5 章)，包括合并排序和快速排序的平均效率，都属于这个类型
n^2	平方	一般来说，这是包含两重嵌套循环的算法的典型效率(参见下一节)。基本排序算法和 n 阶方阵的某些特定操作都是标准的例子
n^3	立方	一般来说，这是包含三重嵌套循环的算法的典型效率(参见下一节)。线性代数中的一些著名的算法属于这一类型
2^n	指数	求 n 个元素集合的所有子集的算法是这种类型的典型例子。“指数”这个术语常常被用在一个更广的层面上，不仅包括这种类型，还包括那些增长速度更快的类型
$n!$	阶乘	求 n 个元素集合的完全排列的算法是这种类型的典型例子

关于渐进时间效率

- 对于实际有限规模的输入，一个效率类型较差的算法有可能比效率类型较优的算法运行得更快
  - n^3 v.s. $10^6 n^2$

- 当比较两个算法的效率时，若两个算法是同阶的，必须进一步考察阶的常数因子才能辨别优劣

极限思想：变量与常量、无限与有限的对立统一关系，是唯物辩证法的对立统一规律在数学领域中的应用

目录

- 算法效率分析框架
  - 时间效率
  - 空间效率
- 算法效率的渐进表征
  - 渐进符号
  - 基本效率类型
- 非递归算法的效率分析
- 递归算法的效率分析

非递归算法的效率分析（举例）

ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

算法 *MaxElement*$(A[0..n-1])$

```
//求给定数组中最大元素的值
//输入：实数数组 A[0..n-1]
//输出：A 中最大元素的值
maxval ← A[0]
for i ← 1 to n-1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

实例1：求最大元素

- 确定基本操作：是赋值运算还是比较运算？

- 求基本操作的执行次数
  记$C(n)$为比较运算的执行次数，则

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

非递归算法的效率分析

分析非递归算法效率的通用方案

1. 决定用哪个（哪些）参数作为输入规模的度量。
2. 找出算法的基本操作（作为一个规律，它总是位于算法的最内层循环中）。
3. 检查基本操作的执行次数是否只依赖输入规模。如果它还依赖一些其他的特性，则最差效率、平均效率以及最优效率（如有必要）需要分别研究。
4. 建立一个算法基本操作执行次数的求和表达式[1]。
5. 利用求和运算的标准公式和法则来建立一个操作次数的闭合公式，或者至少确定它的增长次数。

- 使用变量n来描述输入规模
- 定义算法的基本操作
- 在输入规模为n的情况下，区分最坏、平均和最好情况
- 对基本操作执行的次数求和
- 使用相关公式和规则对求和进行化简

实例2：元素的唯一性问题

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

算法 UniqueElements($A[0..n-1]$)

```
//验证给定数组中的元素是否全部唯一
//输入：数组 A[0..n − 1]
//输出：如果 A 中的元素全部唯一，返回 true
//          否则，返回 false
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

实例2：元素的唯一性问题

- 基本操作：比较

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

实例3：正整数的二进制表示位宽

ALGORITHM *Binary*(n)

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

$count \leftarrow 1$

while $n > 1$ **do**

$count \leftarrow count + 1$

$n \leftarrow \lfloor n/2 \rfloor$

return *count*

算法 *Binary*(n)

//输入：十进制正整数 n

//输出：n 在二进制表示中的二进制数字个数

$count \leftarrow 1$

while $n > 1$ **do**

$count \leftarrow count + 1$

$n \leftarrow \lfloor n / 2 \rfloor$

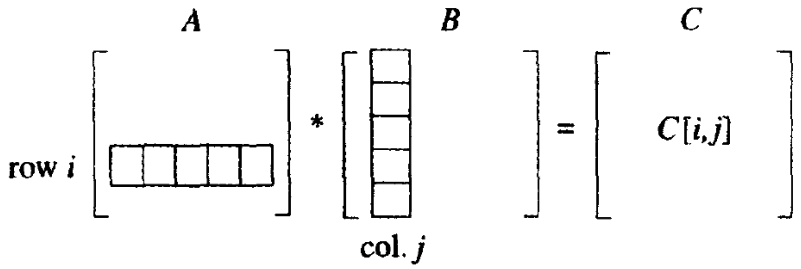
return *count*

the comparison $n > 1$ will be executed is actually $\lfloor \log_2 n \rfloor + 1$

实例4：矩阵的乘法

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $C[i, j] \leftarrow 0.0$
    for $k \leftarrow 0$ **to** $n-1$ **do**
      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C



算法 *MatrixMultiplication*(*A*[0..*n*-1,0..*n*-1],*B*[0..*n*-1,0..*n*-1])

//用基于定义的算法计算两个 *n* 阶矩阵的乘积
//输入：两个 *n* 阶矩阵 *A* 和 *B*
//输出：矩阵 *C* = *AB*
for *i* ← 0 **to** *n*-1 **do**
  for *j* ← 0 **to** *n* - 1 **do**
    C[*i*,*j*] ← 0.0
    for *k* ← 0 **to** *n* -1 **do**
      C[*i*,*j*] ← *C*[*i*,*j*] + *A*[*i*,*k*]*B*[*k*,*j*]
return *C*

实例4：矩阵的乘法

- 输入规模的度量：n
- 基本操作：乘法
- 执行次数表达式

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = n^3$$

- 运行时间：$T(n) \approx C_m M(n) \in \Theta(n^3)$
  - C_m为执行一次乘法在某计算机上所需要的时间

- 若同时考虑乘加法？

打破线性方程求解速度极限，华人学者新算法获计算机理论顶会SODA2021最佳论文奖

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2 \\ \vdots \qquad\qquad \vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n = b_m \end{cases}$$

这个时候，我们就只能求助矩阵乘法了。

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

那么，问题来了，采用高斯消元法，求解的复杂度就是**O(n3)**。

也就是说，如果n从2增加到4，求解复杂度就会增加2^3即8倍。

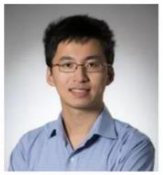
n越来越大，计算的步骤就会**以3次方的速度快速增加**……

想想机器学习、工程项目里极其复杂的矩阵乘法，是不是有点头皮发麻的感觉。

好消息是，现在，这个困扰工程师们已久的**基础问题**，有了突破性进展。

计算机理论顶会SODA 2021的最佳论文，用“猜”答案的方式，一口气把算法复杂度减小到了O(n2.3316)！

论文作者，是来自佐治亚理工学院的两位数学家：彭泱和Santosh Vempala。



Best Paper Award at ACM-SIAM Symposium on Discrete Algorithms (SODA) 2021

Professor Richard Peng and Professor Santosh Vempala received the best paper award at ACM-SIAM Symposium on Discrete Algorithms (SODA) 2021 for their breakthrough work on Solving Sparse Linear Systems Faster than Matrix Multiplication.

这项研究具体是怎么一回事？快来一起研究研究。

IOI金牌获得者，靠“猜”推动研究

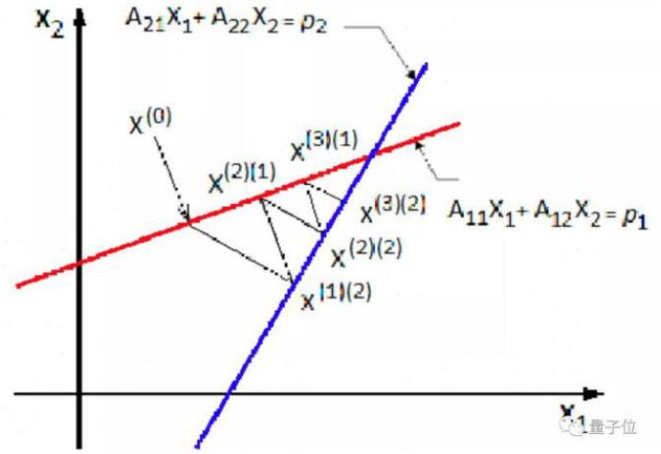
IOI金牌获得者、来自佐治亚理工学院的助理教授彭泱，和他的合作者Santosh Vempala共同提出了一种全新的思路。

打破线性方程求解速度极限，华人学者新算法获计算机理论顶会SODA2021最佳论文奖

这种方法就是：猜测每个未知数的值，把它们代入方程后，查看结果与实际值相差有多大。

然后，修正未知数的值，再猜一次。

这种方法，在计算机方向上被称为**迭代法**。



彭泱的这种迭代算法，在方程的数量变得极多、且每个方程涉及的未知数较少时，显示出了巨大的优势。

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

此前，没有人能够证明，“迭代法”对于稀疏矩阵而言，是否会比“矩阵乘法”更快。

当然，这种算法并不只靠“猜”。

彭泱设计的算法中，他们还会在给出多组随机数的同时，将这些随机数组并行计算。

这就像是数百个人同时在山林中搜索宝藏，肯定比一个人反复搜索要更快。

但这种算法的设计，还面临两个难点：

如何保证设计出来的数，**足够随机**、不偏向问题的任何一部分？

人工智能与科学计算融合发展

可解释、可通用的下一代人工智能方法重大研究计划2023年度项目指南

可解释、可通用的下一代人工智能方法重大研究计划面向人工智能发展国家重大战略需求，以人工智能的基础科学问题为核心，发展人工智能新方法体系，促进我国人工智能基础研究和人才培养，支撑我国在新一轮国际科技竞争中的主导地位。

1. 深度学习的表示理论和泛化理论。

研究卷积神经网络（以及其它带对称性的网络）、图神经网络、transformer网络、循环神经网络、低精度神经网络、动态神经网络、生成扩散模型等模型的泛化误差分析理论、鲁棒性和稳定性理论，并在实际数据集上进行检验；研究无监督表示学习、预训练-微调范式等方法的理论基础，发展新的泛化分析方法，指导深度学习模型和算法设计。

2. 深度学习的训练方法。

研究深度学习的损失景观，包括但不限于：临界点的分布及其嵌入结构、极小点的连通性等，深度学习中的非凸优化问题、优化算法的正则化理论和收敛行为，神经网络的过参数化和训练过程对于超参的依赖性问题、基于极大值原理的训练方法、训练时间复杂度等问题，循环神经网络记忆灾难问题、编码-解码方法与Mori-Zwanzig方法的关联特性，发展收敛速度更快、时间复杂度更低的训练算法及工具，建立卷积网络、Transformer网络、扩散模型、混合专家模型等特定模型的优化理论及高效训练方法，深度学习优化过程对泛化性能的影响等。

3. 微分方程与机器学习。

研究求解微分方程正反问题及解算子逼近的概率机器学习方法；基于生成式扩散概率模型的物理场生成、模拟与补全框架；基于微分方程设计新的机器学习模型，设计和分析网络结构、加速模型的推理、分析神经网络的训练过程。

面向具有实际应用价值的反问题，研究机器学习求解微分方程的鲁棒算法；研究传统微分方程算法和机器学习方法的有效结合方法；研究高维微分方程的正则性理论与算法；研究微分方程解算子的逼近方法（如通过机器学习方法获得动理学方程、弹性力学方程、流体力学方程、Maxwell方程以及其它常用微分方程的解算子）；融合机器学习方法处理科学计算的基础问题（求解线性方程组、特征值问题等）。

目录

- 算法效率分析框架
  - 时间效率
  - 空间效率
- 算法效率的渐进表征
  - 渐进符号
  - 基本效率类型
- 非递归算法的效率分析
- 递归算法的效率分析

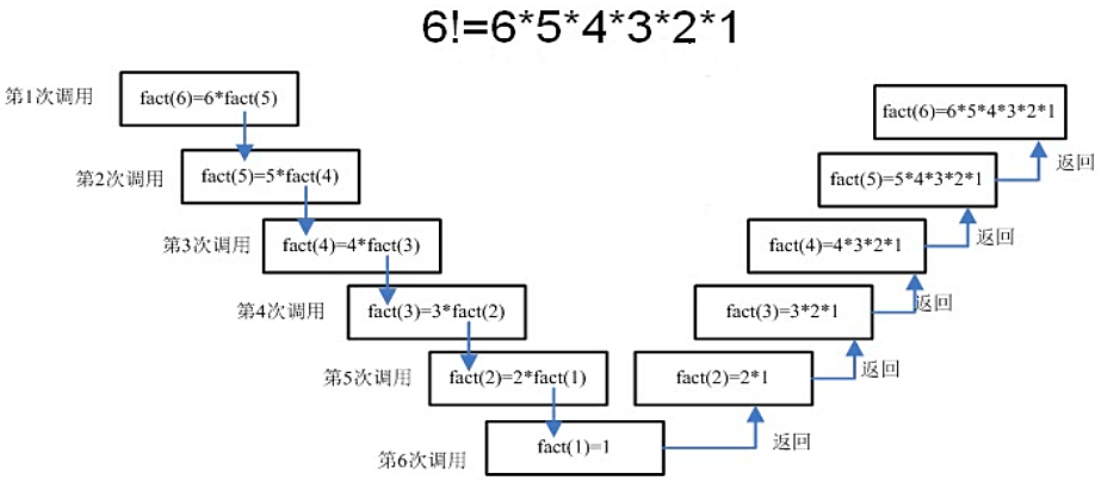
递归算法的效率分析

- 递归算法
  - 函数调用自身的情况称为递归
- 递归的条件
  - 子问题与原问题属于同样的问题，且更为简单
  - 不能无限制调用，必须有出口条件

递归算法的效率分析：实例1

- 从简单的阶乘递归算法出发
  - 定义： $n! = 1 * 2 * \ldots *(n-1) * n$ for $n \geq 1$ and $0! = 1$
  - 递归公式：

    $F(n) = F(n-1) * n$ for $n \geq 1$, and $F(0) = 1$



系列问题：输入规模？基本操作？执行次数的递推式？初始条件？

基本操作和执行次数的递推

- 基本操作：乘法

  $F(n) = F(n\text{-}1) * n \quad \text{for} \quad n>0$

- 执行次数递推式

$$C(n) = \underbrace{C(n-1)}_{\text{计算}F(n-1)} + \underbrace{1}_{\text{计算}F(n-1)*n} \quad \text{for } n > 0$$

  - 初始条件：$C(0)=0$

- 时间效率

$$\begin{aligned} C(n) &= (C(n-2)+1)+1 = C(n-2)+2 \\ &= (C(n-3)+1)+2 = C(n-3)+3 \\ &\cdots \\ &= C(n-n)+n = n \in O(n) \end{aligned}$$

递归算法的效率分析

分析递归算法时间效率的通用方案

(1) 决定用哪个(哪些)参数作为输入规模的度量标准。

(2) 找出算法的基本操作。

(3) 检查一下，对于相同规模的不同输入，基本操作的执行次数是否可能不同。如果有这种可能，则必须对最差效率、平均效率以及最优效率做单独研究。

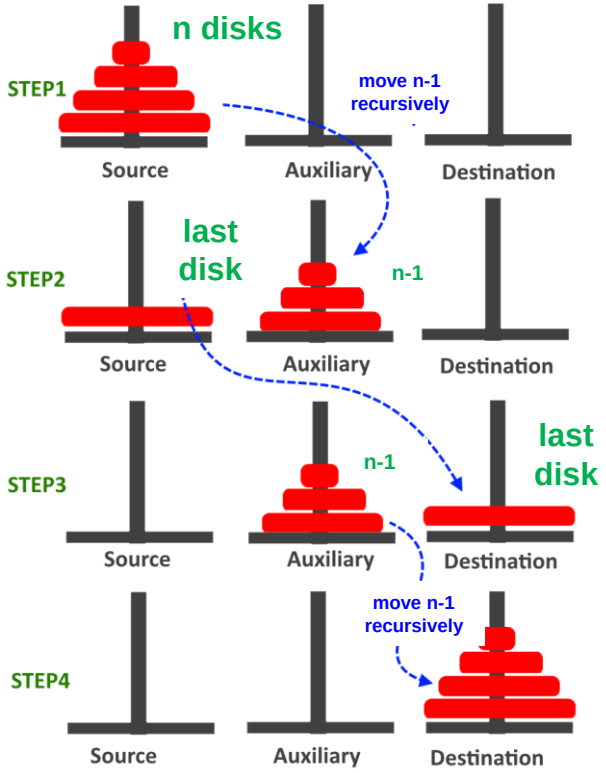
(4) 对于算法基本操作的执行次数，建立一个递推关系以及相应的初始条件。

(5) 解这个递推式，或者至少确定它的解的增长次数。

实例2：汉诺塔

- 问题
  - 将n个不同大小的盘子从第一个柱子上，通过第二个柱子移动到第三个柱子上，始终要保持下面的盘子比上面的盘子大

- 递归算法



实例2：汉诺塔

- 输入规模：盘子个数*n*
- 基本操作：盘子的移动
- 执行次数递推式

$$C(n) = C(n-1) + 1 + C(n-1)$$
$$= 2C(n-1) + 1, \quad \text{for } n > 1$$

  - 初始条件：$C(1)=1$

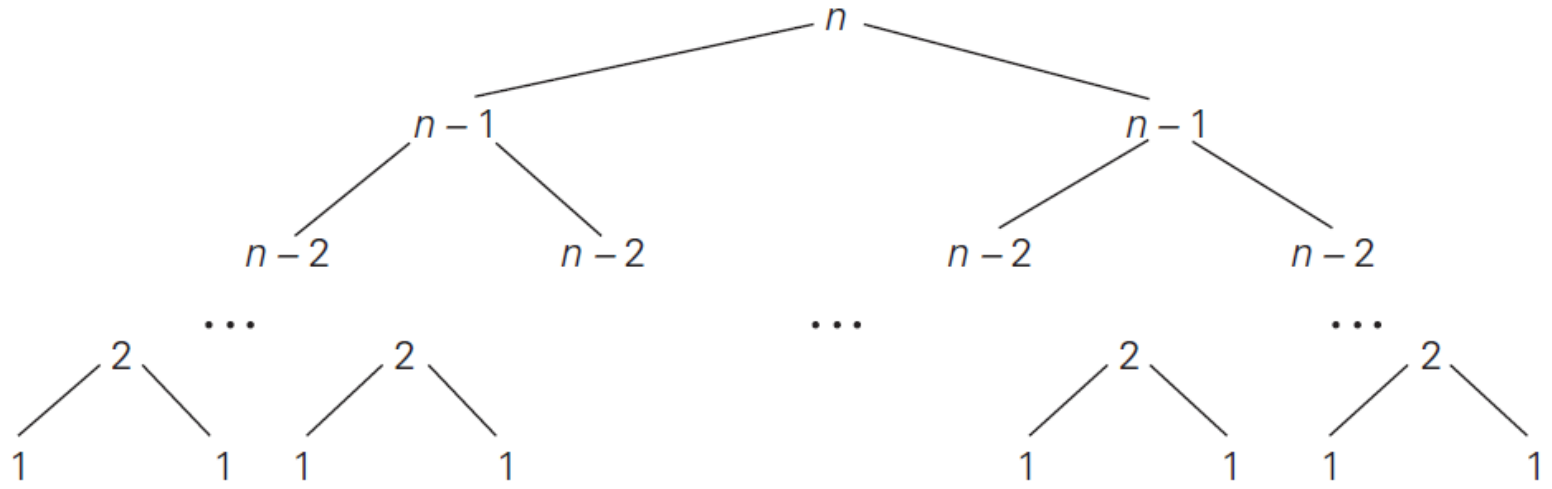
- 时间效率 $C(n) = 2C(n-1) + 1$

$$= 2(2C(n-2)+1)+1 = 2^2C(n-2) + 2 + 1$$
$$= 2^2(2C(n-3)+1) + 2 + 1 = 2^3C(n-3) + 2^2 + 2 + 1$$
$$\ldots$$
$$= 2^{n-1}C(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n\text{-}1 \in O(2^n)$$

指数级复杂度！

实例2：汉诺塔

- 汉诺塔递归算法的递归调用树表示
  - 基本操作：调用函数本身



调用的全部次数：$C(n)=\sum_{l=0}^{n-1}2^l=2^n-1\in O(2^n)$

不是算法不好，而是问题本身决定了它在计算上的难度。

实例3:正整数的二进制表示位宽递归求解

ALGORITHM *BinRec*(n)

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

if $n = 1$ **return** 1

else return $BinRec(\lfloor n/2 \rfloor) + 1$

基本运算是加法，执行次数用以下公式计算

- C(n) = C(n/2)+1
- C(1) = 0

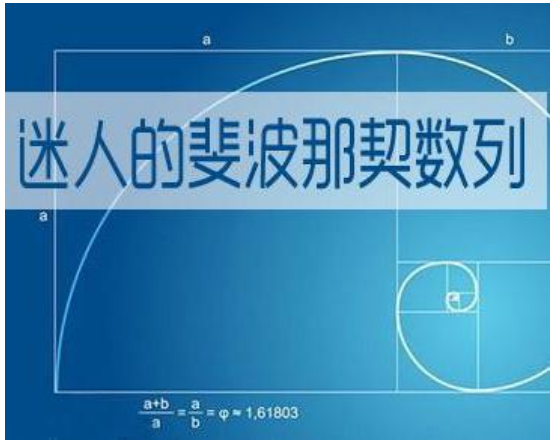
因此

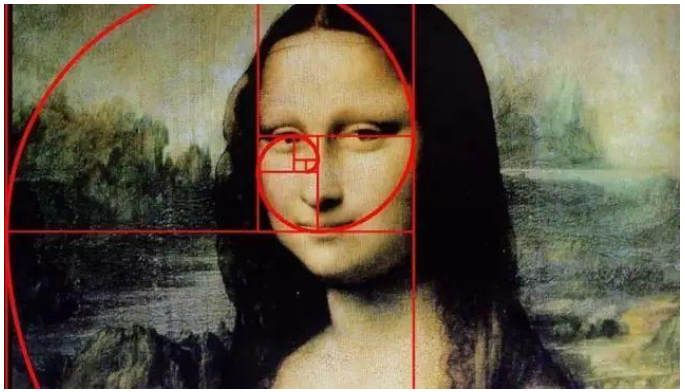
C(n) = C(n/2)+1 = C(n/2^2)+2

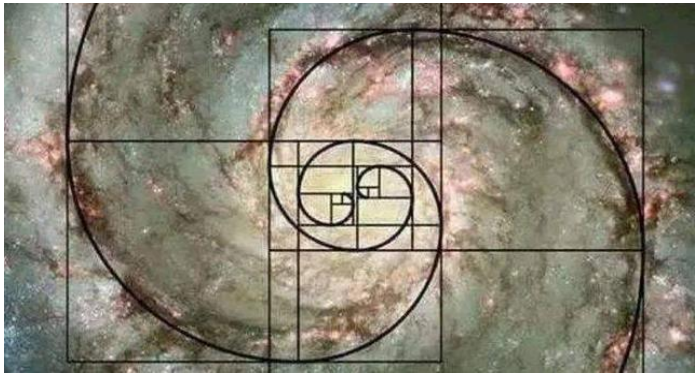
对n=2^k求解得 c(n) = c(n/2^k)+k = $\log_2 n \in \Theta(\log n)$

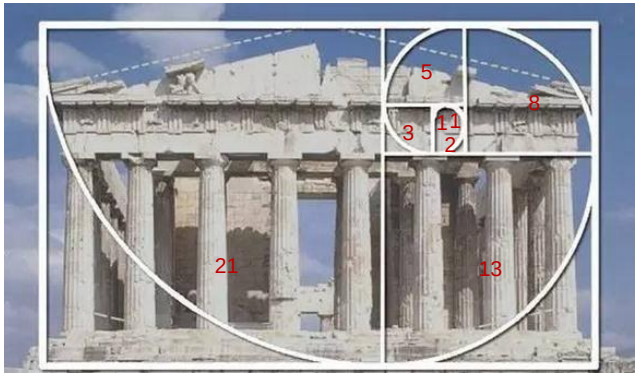
实例4：斐波那契数列（黄金分割数列）

0，1，1，2，3，5，8，13，21，34,...









实例4：斐波那契数列

- 数列的递推式和初始条件

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n > 1$$
$$F(0) = 0, \qquad F(1) = 1.$$

- 算法1：递归求解第n个斐波那契数

算法 $F(n)$
//根据定义，递归计算第 n 个斐波那契数
//输入：一个非负整数 n
//输出：第 n 个斐波那契数
if $n \leqslant 1$ **return** n
else return $F(n-1) + F(n-2)$

实例4：斐波那契数列

- 算法1（递归算法）
  - 输入规模：n
  - 基本操作：加法
  - 执行次数的递推式：

$$C(n)=C(n-1)+C(n-2)+1,\quad \text{for } n>1$$

    - 初始条件：$C(0)=0,\quad C(1)=0$

  - 时间效率

$$[C(n)+1]-[C(n-1)+1]-[C(n-2)+1]=0$$

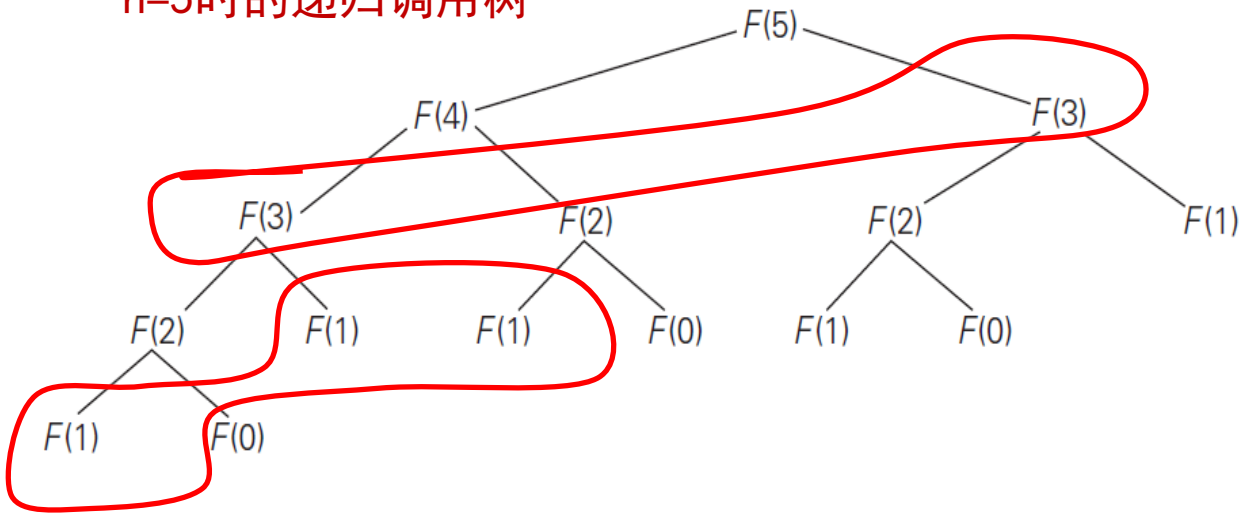
齐次递推式求解（教材附录B） $\Longrightarrow$ $C(n)=\frac{1}{\sqrt{5}}(\phi^{n+1}-\hat{\phi}^{n+1})-1\in\Theta(\phi^n)$

$\phi=(1+\sqrt{5})/2\approx 1.61803,\quad \hat{\phi}=-\phi^{-1}\approx -0.61803$

实例4：斐波那契数列

- 算法1（递归算法）
  - 指数级复杂度
  - 根源：相同的函数值一遍又一遍地重复计算

n=5时的递归调用树



我们应该谨慎使用递归算法，因为它们的简洁可能会掩盖其低效率的事实。

实例4：斐波那契数列

- 算法2（迭代算法）

算法 Fib(n)
//根据定义，迭代计算第 n 个斐波那契数
//输入：一个非负整数 n
//输出：第 n 个斐波那契数
$F[0] \leftarrow 0$; $F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
  $F[i] \leftarrow F[i-1] + F[i-2]$
return $F(n)$

  - 基本操作：加法
  - 执行次数：$C(n) = n - 1 \in \Theta(n)$

实例4：斐波那契数列

- 算法3（直接求解法）

$$F(n)-F(n-1)-F(n-2)=0$$

齐次递推式求解（教材附录B） $\Longrightarrow$ $F(n)=\frac{1}{\sqrt{5}}(\phi^n-\hat{\phi}^n)$

$\phi=(1+\sqrt{5})/2\approx 1.61803,\quad \hat{\phi}=-\phi^{-1}\approx -0.61803$

- 基本操作：乘法
- 执行次数：$C(n)=2(n-1)\in\Theta(n)$

实例4：斐波那契数列

- 算法4（基于矩阵乘方求解）

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix}$$

$$= \ldots$$

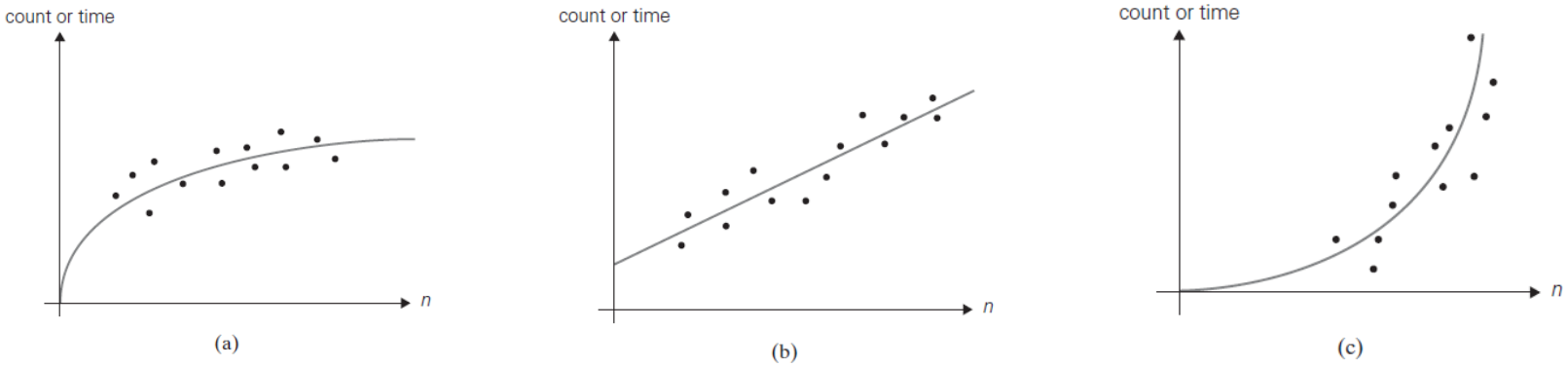
$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

时间效率提升为$\Theta(\log n)$

为什么？

算法效率：数学分析 vs 经验分析

- 数学分析
  - 优：不依赖于特定的输入和计算机
  - 劣：适用性有限，平均效率刻画困难
- 经验分析：
  - 优：基于实验统计来做分析和预测，适用于任何算法
  - 劣：依赖实验中的特定样本实例和计算机



基于实验统计执行时间或次数的散点图分析

小结

- 算法效率
  - 时间效率：算法的运行速度；空间效率：算法需要的额外空间
- 时间效率
  - 关键因素：输入规模、基本操作的执行次数
  - 对有些算法，相同规模的输入，运行时间也会有相当大的不同，需要进一步分析最优效率、最差效率和平均效率
  - 效率的渐进表征：O(g(n))、Ω(g(n))、Θ(g(n))
  - 基本效率类型：常数、对数、线性、线性对数、平方、立方和指数
- 具体算法分析
  - 非递归算法：建立算法的基本操作执行次数的求和表达式，然后确定“和函数”的增长次数
  - 递归算法：建立算法的基本操作执行次数的递推关系式，然后确定它的增长次数

课后作业

章 X	节 X. Y	课后作业题 Z	思考题 Z
2	2.1	10	4
	2.2	5	11, 12
	2.3	4	10, 12, 13
	2.4	4	5, 13, 14
	2.5	8	2, 3, 11
	2.6	1	

注：只需上交“课后作业题”；以“学号姓名_chX.pdf ”规范命名，提交到“学在浙大”指定文件夹。DDL：2024年3月12日