第1章 算法

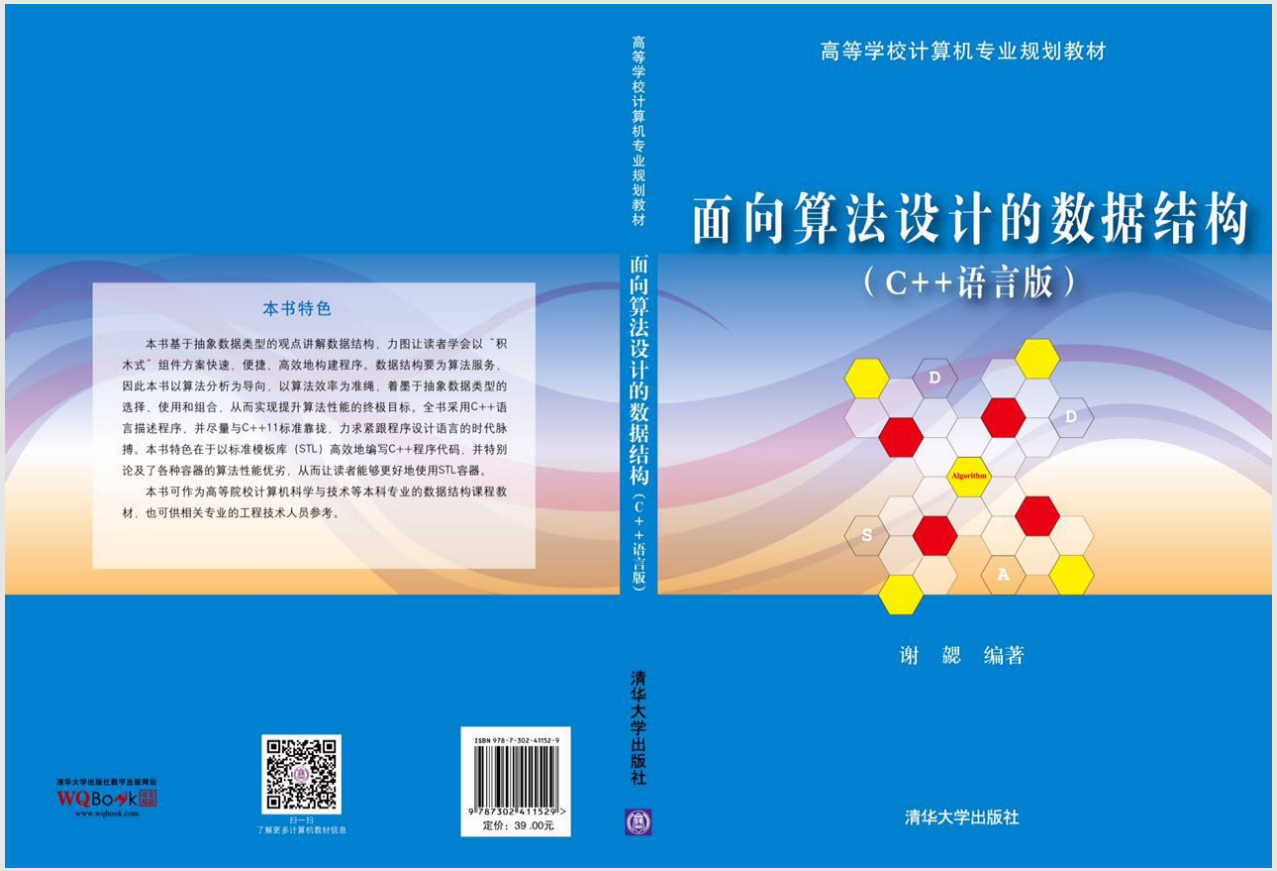
谢勰 (微博: @算海无涯-X)

西安邮电大学

August 1, 2016

教材

- 谢鑑, 《面向算法设计的数据结构（C++语言版）》. 清华大学出版社, 2015.



计算的目的是什么?

- 计算(computation)这个词早已突破了传统意义上的运算操作. 简单地说, 计算是解决人类以手工方式难以解决的问题. 从人类的效率特点而言, 可能这些问题是"无法解决"的.
- 计算并不意味着笨拙的苦力工作. 许多我们认为是只能依靠人类智慧才能解决的问题, 如今通过高质量的计算也可以达到目的, 棋类活动就是明证(深蓝和AlphaGo).
- 在计算的过程中, 一个至关重要的概念就是算法(algorithm). 从某种意义上说, 算法是在计算中实现人类智慧的关键枢纽.

何谓算法?

- 烹饪法(recipe)?
- 方法(method)?
- 过程(procedure)?
- 技术(technique)?
- Merriam-Webster词典定义:

  A procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end especially by a computer.

算法定义

- Knuth给出了更为专业的“定义”:

  Besides merely being a finite set of rules that gives a sequence of operations for solving a specific type of problem, an algorithm has five important features: 1) Finiteness. .... 2) Definiteness. .... 3) Input. .... 4) Output. .... 5) Effectiveness. ....

- 上述看似简单却相当精确的描述非常值得我们深入思考.
- 事实上, Knuth总结了算法的五个重要特性!

算法的特性

- 有穷性(Finiteness): 算法必须在有限步内结束.
- 确定性(Definiteness): 算法的所有步骤都必须精确定义, 不得模棱两可.
- 输入(Input): 算法通常得有输入数据, 不过输入可以为空.
- 输出(Output): 算法肯定会得到一个或多个输出数据. 输出和输入是有关系的, 而这种关系的确认通常会涉及算法是否“正确”地实现了设计者的意图.
- 能行性(Effectiveness): 算法的所有运算对于执行者而言都应是“基本”运算. “基本”这个词意味着原则上人可以用纸笔精确复原并完成算法的执行过程.

例

插入排序是一个简单的排序方法, 以输入数据为2，5，3，1，4为例, 从算法的角度分析插入排序是否满足算法的五个特性.

解

输入是2，5，3，1，4. 插入排序每步依次将数据取出, 再按一定规则插入已排序的序列中, 容易验证该方法满足确定性. 可在纸上写出过程:

```
第1步：2                    // 取出2插入.
第2步：2，5                 // 取出5插入.
第3步：2，3，5              // 取出3插入.
第4步：1，2，3，5           // 取出1插入.
第5步：1，2，3，4，5        // 取出4插入.
```

从上述过程可以看出插入排序满足有穷性和能行性. 最后, 插入排序在本例中的输出是1，2，3，4，5.

怎样可以称为算法

- 可以证明插入排序满足算法的五个特性，因此它可称为一个算法.
- 满足算法的五个特性，即可称为算法. 我们可以寻找其他算法并证明它们是算法，例如冒泡排序和选择排序(留作练习).
- 插入排序的有穷性需要严格的证明(留作练习). 从更深入的角度看，有穷性的判定属于停机问题(halting problem).
- 顺便提及，插入排序来源于现实生活(扑克的取牌整理). 事实上，许多算法的设计都需要从实际中寻找灵感.

算法思维

- 算法体现着一种思维. 因为对于海量数据:
  - 人类在处理大量重复性和规律性的操作步骤时非常容易出错, 这点远不如机器.
  - 人的许多手工处理方式是无规则而且杂乱的, 这种随意的处理在数据量较大时根本无法实施.
  - 机器只能按照规则行事, 而算法就是它们的规则.
- 因此, 算法是处理困难问题的一种必然选择.
- 不夸张地说, 算法是机器的世界观.
- CMU前系主任J. M. Wing总结出“计算思维”(Computational thinking). 事实上, 各个行业的人都必须能够掌握计算思维这种古老而又弥新的模式.

算法应用

- 计算机学科:
  - 人工智能(Artificial Intelligence)
  - 数据库(Database)
  - 计算机网络(Computer Network)
  - 操作系统(Operating System)
  - 计算机图形学(Computer Graphics)
- 其他学科领域:
  - 信号处理(Signal Processing)
  - 模式识别(Pattern Recognition)
  - 集成电路设计(Integrated Circuit Design)
- 学科交叉:
  - 计算生物学(Computational Biology)

二分查找

- 问题: 在一个已经过整理的历史数据集中查找, 说得更准确些, 是在有序排列的数组data中查找某个值key.
- 实例: 在字典中查找某个词.
- 方案: 翻到字典的中间位置, 根据与当前位置单词的比较来决定下次应该选择哪一半作为新的待查空间, 这种方法一般称为二分查找(binary search).
- 程序: https://github.com/xiexiexx/DSAD/blob/master/Ch01/binarysearch.cpp.

二分查找程序是算法

- 二分查找程序满足有穷性. 因为该程序在循环体内的关键在于三路分支的判断:
  - key小于data[mid]: 转向前一半.
  - key大于data[mid]: 转向后一半.
  - key等于data[mid]: 跳出循环.

  这三类情况下都趋向于缩短当前的区间或者跳出循环, 可保证循环的终结.

  此外, 请注意多路分支语句处理判断的写法, 而且我们只用到了<运算符判断.
- 二分查找程序具备输入, 并规定了输入数据的形态, 以接口的形式对外宣布.

二分查找程序是算法 (续)

- 二分查找程序能输出数据，并且数据形式清楚明白地反映了问题的答案. 可以证明: 如果key可查到，found为true，且返回的位置pos对应的值为key; 反之found为false，并返回一个输入中不存在的pos位置(这里定为N).
- 二分查找程序已经用C++程序设计语言实现且编译无误，因此是确定的.
- 二分查找程序所使用的语句都在C++程序设计语言所能完成的范围之内，因而是能行的.

综上可知，二分查找程序是一个算法.

如何评估算法

- 算法是程序的灵魂. 而算法性能则是评估算法优劣的核心.
- 运行时间和占用空间是算法性能最关键的指标. 可以考察这些指标的实际值:
  - 可利用计时工具来考察运行时间. 我们在本书附录中提供了一个xTimer类, 计时精度可达毫秒级别.
  - 可利用操作系统中相关工具查看占用空间. 例如Windows操作系统下可运行perfmon.msc, 而Mac OS X中可使用Activity Monitor.
- 高效(Efficient)的算法意味其执行过程消耗的机器资源较少, 因此算法的执行过程应尽量少占用处理机时间和内存.

算法分析

- 真实的运行时间和占用空间这样的具体且“实际”的指标值却是不“实用”的性能指标.
  - 不同的计算机由于性能各异造成实测指标值不同.
  - 实测指标值随着输入数据不同而相差较大.
- 可在某些条件上测试而获得具体的指标值, 但这无法适用于所有情况.
- 我们需要更好的理论分析工具!
- 而这就是算法分析(Algorithm Analysis).
- 算法分析对数学要求很高, 不妨从Sedgewick和Flajolet所著*An Introduction to the Analysis of Algorithms*中一窥堂奥.

算法效率指标

- 算法的时空效率指标:
  - 时间复杂度(Time complexity). 算法完全运行所需运算时间.
  - 空间复杂度(Space complexity). 算法完全运行所需存储空间.
- 时间复杂度和空间复杂度都可能随输入数据I的具体情况而有显著的改变.
- 输入数据的多少一般称为问题的规模量(size), 它是决定时间复杂度和空间复杂度的关键因素.
  - 一个长度为n的数组, 其问题规模量为n.
  - 一个具有V个顶点和E条边的图, 其问题规模量为$V+E$.
  - 一个长为l的二进制大数(Big Number), 其问题规模量为l. 注意, 这个大数的值至少在2^{l-1}级别.

输入数据的取值情况影响

- 输入数据I的具体取值情况也是较为重要的因素.

例

在数组中查找某个指定元素, 我们可以采用从头到尾逐个检查的方案.

- 如果恰好所找的元素就是数组首元素, 算法会马上执行完毕.
- 如果所找的元素位于数组尾部, 算法可能需要较长的时间.

- "好"的数据会使算法占用时空较少, "坏"的数据会让算法耗费相当多的机器资源.
- 如何区分数据取值情况对性能分析带来的影响?

三种情况下的算法分析

- 最坏情况(Worst-case)下的时间/空间复杂度. 问题规模量给定, 输入遍取所有可能情况下时间/空间复杂度的最大值.
- 最好情况(Best-case)下的时间/空间复杂度. 问题规模量给定, 输入遍取所有可能情况下时间/空间复杂度的最小值.
- 平均情况(Average-case)下的时间/空间复杂度. 问题规模量给定, 输入遍取所有可能情况下时间/空间复杂度的数学期望, 通常假设输入数据满足等概率分布.

三种情况下时间复杂度和空间复杂度仅为问题规模量的函数, 作为指标使用更为方便简洁. 通常我们考察最坏情况, 若问题规模量为n, 在不引起混淆的情况下, 我们一般将最坏情况下的时间复杂度和空间复杂度简记为$T(n)$和$S(n)$.

RAM模型

- 一般假设算法运行于理想状态下:
  - 独占CPU且独享内存. 我们主要考虑串行的算法执行, 那么只能在一个CPU上执行操作.
  - 假设内存是“无限”的, 不考虑内外存数据交换.
  - 算法在串行计算机上的执行过程通常可分解为一串指令序列, 统计各种指令的次数则可得到时间复杂度. 为简单起见不妨假设每条指令执行时间相同, 这样只需考虑指令的总数.
  - 假设算法可直接按地址访问存储空间, 且每次访问耗费常数时间.
- 以上假设其实就是利用随机存取机(Random-Access Machine, RAM)模型来分析算法.

时间复杂度与元操作

- 在具体的C++程序实现中，RAM不能直接用于分析算法，但可以作为借鉴.
- 可将复杂多样的程序执行过程分解为一系列“不可分割”的语句所组成的序列，或称为元操作(primitive operation)序列.
- 常见的元操作有赋值、算术运算、逻辑运算和跳转等. 于是，统计这些元操作即可达到分析时间复杂度的目标.
- 需要指出的是，算法分析时应将逗号表达式进行分解，尽量只使用分号作为间隔从而获得语句的独立性.
- 关于函数调用和递归函数调用的分析，将留待后文再作讨论.

例

分析if语句和while语句的结构，将其分解为元操作.

解

- if(expression)语句：先判断expression的真假，再利用其值进行选择性跳转，如果还有else分支则还要在if分支的结尾无条件跳转以避开else分支所包含的语句.
- while(expression)语句：在起始处先判断expression的真假，再利用其值进行选择性跳转，注意循环部分的末尾需加上一条无条件跳转语句以回到while语句起始处. 于是上述控制机制便可分解为跳转形式的元操作.
- expression分解为元操作很容易. 例如if (a > b && b > c)中的(a > b && b > c)表达式可分解为a > b和b > c的&&操作.

分析的层级

- 由于程序可由顺序、判断和循环等三种基本结构实现，那么经过前文中的分解与变换，便能很容易根据元操作分析算法的时间复杂度.
- 使用低级程序设计语言(如汇编语言)描述并分析算法可以得到更好的结果，但不利于理论分析.
- 高级程序设计语言层面上考察则有利于减轻分析工作的复杂程度，也基本上能较好地接近于机器指令序列分析.
- 事实上，对于渐近意义上的算法分析采用何种语言没有任何差异，因此我们将基于C++语言分析算法.

例

对于某个int型二维数组data进行赋值操作, 将其视为一个$n \times n$的矩阵, 对角线上元素赋值为x, 其他元素赋值为y. 比较图1和图2这两种程序实现的性能.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        if (i != j) // 对角线判断.
            data[i][j] = y;
        else
            data[i][j] = x;
```

图 1: 矩阵赋值方案A

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        data[i][j] = y;
// 在对角线上重新赋值.
for (int k = 0; k < N; k++)
    data[k][k] = x;
```

图 2: 矩阵赋值方案B

解

- 粗看: A方案对data进行赋值的次数较少, 而B方案重复对data对角线上的元素赋值.
- 但时间复杂度的分析才是评判优劣的标准. 可以考虑它们的对比分析:
  - A方案多做了n^2次对角线判断.
  - B方案所多做的是若干种元操作且总数为$an + b$, 其中a和b是两个确定的正整数(其值的计算留作练习).

  在n较大时差别立现!
- 从时间复杂度和程序清晰性的角度来看, B方案都是较好的.
- 其他讨论详见教材.

空间复杂度

- 算法的空间复杂度分析稍显简单，但需要注意它指的是为了实现算法而占用的空间，因此不包括输入和输出部分. 此外，程序代码本身所占的空间一般也不在空间复杂度的考察范围之内.
- 类比时间复杂度，空间复杂度也应该寻找“元类型”，即化简各种类型为不可分割的类型. 显然int，double，char这些类型都是“元类型”. 不过，空间复杂度不是简单的累加，而是考虑执行过程中所能达到的最大空间用量. 分析静态申请的存储空间较为简单，而分析动态申请或函数调用时所使用的空间特别需要考虑那些用完之后闲置的部分.

例

分析下列逆置数组算法Reverse的空间复杂度.

```
template <typename T>
void Reverse(T D[], T R[], int N)
{
    // 输入: 数组D; D的长度N.
    // 输出: 长度为N的数组R.
    // 算法完成任务: 将D中所有元素按逆序存于R中.

    for (int i = 0; i < N; i++)
        R[N - i - 1] = D[i];
}
```

解

该算法所需空间仅为局部变量i和Reverse函数的参数(两个指针和一个int型变量), 其空间复杂度相当少.

- 算法的空间复杂度仅会计入操作输入和输出额外所需的空间, 而并不考虑输入和输出本身, 例如Reverse函数仅仅使用了数组D和数组R的首元素地址.
- 极少数情况下传值参数可能会完全复制输入, 而函数体内也可能先暂存输出, 但这不意味着空间复杂度考虑了输入和输出, 只是由于算法使用输入/输出的副本而造成的.
- 事实上, "真正"的输入和输出都由算法使用者(调用者)提供.

时空权衡

- 一般而言，时间复杂度是比空间复杂度更为重要和更有价值的指标.
- 我们会很乐意为了争取时间而用较大的空间去交换，而现实中这样的优秀算法屡见不鲜.
- 当然，空间利用率也是展现算法设计技术高低的重要标准.
- 此外，在对空间有专门限制的场合(如嵌入式系统中)，应该更多地考虑空间需求. 而在目前智能手机乃至于可穿戴设备日益盛行的时代，空间复杂度的重要性也在逐步提升.
- 总而言之，我们需要根据具体情况来确定时间和空间的重要性. 事实上，对于时间和空间之间的权衡(trade-off)，永远是算法中有益而有趣的话题.

渐近记号

- 分析时间复杂度和空间复杂度的表达形式相当繁琐. 其实只需作出一个大致估计即可, 例如时间以秒计肯定强于以分钟计, 又如空间以KB计一定优于以TB计.

定义 (渐近记号)

假定$f(n)$和$g(n)$均为$\mathbb{N}$上的非负函数. 定义如下记号:

- O记号. 我们称$f(n) = O\big(g(n)\big)$, 若有

$$\exists c \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 \big(f(n) \leq c \times g(n)\big).$$

- Ω记号. 我们称$f(n) = \Omega\big(g(n)\big)$, 若有

$$\exists c \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 \big(f(n) \geq c \times g(n)\big).$$

渐近记号 (续)

定义 (渐近记号 (续))

- Θ记号. 我们称$f(n)=\Theta\big(g(n)\big)$, 若有

$$f(n)=O\big(g(n)\big) \quad \text{且} \quad f(n)=\Omega\big(g(n)\big).$$

- o记号. 我们称$f(n)=o\big(g(n)\big)$, 若有

$$\forall c\in\mathbb{R}^+\ \exists n_0\in\mathbb{N}\ \forall n\geq n_0\big(f(n)<c\times g(n)\big).$$

上述符号一般称为渐近记号(asymptotic notations). 需要注意, 渐近记号定义中的等号应看成“是”, 这意味着等号的左右不可颠倒.

渐近记号中的c和n_0不一定要取得非常好, 只要满足要求即可, 也就是说够用即可. 所以, 我们应尽量减少论证的复杂难度!

例

设$f(n)=32n^2+10n+5$, 求出它的渐近记号.

解

- 易知$n \geq 0$时, 恒有$f(n) \leq 32n^2+10n^2+5n^2=47n^2$(注意系数不变). 我们取$n_0=0, c=42$, 依定义可知$f(n)=O(n^2)$.
- 取$c=16$(最高项系数的一半), 我们想找到n_0, 使得$n \geq n_0$情况下满足$32n^2+10n+5 \geq 16n^2$, 易知$n \geq (5+\sqrt{105})/16$情况下可满足目标, 于是我们取$n_0=(5+\sqrt{105})/16$. 依定义可知$f(n)=\Omega(n^2)$.
- 易知, $f(n)=\Theta(n^2)$. 此外, $f(n)=o(n^3)$留作练习.

渐近记号简单明了, 足以作为函数的估计. 我们应尽量求出描述更为精确的Θ记号, 并且还要让O记号和Ω记号尽可能接近于Θ记号(严格地说是更为紧致). 在本例中, $f(n)=O(n^3)$和$f(n)=\Omega(n)$就不是好的估计.

渐近记号小议

简单说来，渐近记号的意义是：

- O记号描述了函数的上界(*upper bounds*)，o记号是更严格的上界.
- Ω记号描述了函数的下界(*lower bounds*).
- Θ记号描述了函数的紧致界(*tight bounds*)或者说量级(*order*).
- o记号给出了函数之间的严格序关系，一般以此衡量性能优劣.

- 在算法分析中使用最多的是O记号，其原因是获得一个可用的O记号相对容易，而实际中常常难以得到显式的Θ记号.
- 更重要的是，O记号比较符合日常估计的物理意义. 例如“稍等几分钟”中的“分钟”决定了时间的量级上限，而O记号同样约束了函数的上界，从这个角度看算法分析中用O记号足矣.

例

设某算法时间复杂度为: $T_w(n) = 1000n$(最坏情况), $T_b(n) = 500$(最好情况), $T_a(n) = 300n$(平均情况), 讨论它们的渐近记号.

解

根据渐近记号的定义容易知道:

- $T_w(n) = O(n)$, $T_w(n) = \Omega(n)$, $T_w(n) = \Theta(n)$.
- $T_b(n) = O(1)$, $T_b(n) = \Omega(1)$, $T_b(n) = \Theta(1)$.
- $T_a(n) = O(n)$, $T_a(n) = \Omega(n)$, $T_a(n) = \Theta(n)$.

注意不要将渐近记号与算法分析中的最坏情况、最好情况和平均情况混淆. 渐近记号仅仅描述了函数的数学性质, 每种渐近记号都可以处理不同情况下的时间复杂度和空间复杂度, 也即渐近记号给出了渐近时间复杂度和渐近空间复杂度.

例

讨论$f(n)=5n^3-42n^2+63n-7$的渐近记号.

解

- 易知$n\geq 0$时, 恒有$f(n)\leq 5n^3+42n^3+63n^3+7n^3=117n^3$(注意系数取正). 我们取$n_0=0, c=117$, 依定义可知$f(n)=O(n^3)$.
- 取$c=5/2$(最高项系数的一半), 我们想找到n_0, 使得$n\geq n_0$情况下满足$5n^3-42n^2+63n-7\geq 5/2n^3$, 也即$5n^3/2\geq 42n^2-63n+7$, 不妨放缩为$5n^3/2\geq 42n^2+63n^2+7n^2\geq 42n^2-63n+7$, 这样一来$5n^3/2\geq 112n^2$也就是$n\geq 224/5$情况下即可满足目标, 于是我们取$n_0=224/5$. 依定义可知$f(n)=\Omega(n^3)$.
- 易知, $f(n)=\Theta(n)$.

练习: 使用Matlab等工具绘制并观察$f(n)$函数曲线形态.

例

考虑d阶多项式$f_d(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_0$, 若首项系数$a_d > 0$, 证明$f_d(n) = \Theta(n^d)$.

证明

对d使用归纳法:

(1) 当$d = 0$, 显然$a_0 = \Theta(1)$.

(2) 假设$d \leq k$时任意符合题设要求的多项式函数满足$f_d(n) = \Theta(n^d)$.

那么当$d = k + 1$时, 可考虑函数$R_{k+1}(n) = f_{k+1}(n) - a_{k+1} n^{k+1}$.

- $R_{k+1}(n) = 0$: 显证$f_{k+1}(n) = \Theta(n^{k+1})$.
- $R_{k+1}(n) \neq 0$: $R_{k+1}(n)$或$-R_{k+1}(n)$必有其一符合题设多项式条件, 于是存在整数$l \in [0, k]$使得$R_{k+1}(n) = \Theta(n^l)$或$-R_{k+1}(n) = \Theta(n^l)$. 而$a_{k+1} > 0$, 易知$f_{k+1}(n) = a_{k+1} n^{k+1} + R_{k+1}(n) = \Theta(n^{k+1})$.

综上可知题设多项式$f_d(n) = \Theta(n^d)$.

例

现有两台计算机H和L, H的运行速度是L的k倍. 我们为处理同一问题设计了两种不同的算法F和S, 在问题规模量为n情况下两种算法在L上所需时间分别为$2n^2$和n^3. 比较在计算机H上执行算法S与在计算机L上执行算法F的运行时间差异.

解

- 在计算机H上执行算法S的运行时间理论上应为n^3/k.
- 在计算机L上执行算法F的运行时间为$2n^2$.
- 容易知道在$n > 2k$时, 计算机L上执行算法F的运行时间较少.
- 一般而言, k不会是一个特别大的数字, 那么大多数情况下应该考虑在计算机L上执行算法F.
- 显然, 算法的好坏比机器的速度更重要,

渐近记号与性能评判

- 先在计算机L上比较算法，由于$2n^2 = o(n^3)$，因此算法F必然更快. 无论换多快的计算机H，都存在一个较大的n_0，使得问题规模量大于n_0时，在L上执行算法F的速度快于在H上执行算法S的速度，这意味着算法F在渐近意义上总是较快的.
- 如果某种算法在某台计算机上运行时间为$T(n)$，那么该算法在任意计算机上的运行时间都是$\Theta(T(n))$，于是Θ记号就得到了一台"抽象"计算机. 由于算法F和算法S在"抽象"计算机上的"运行时间"分别为$\Theta(n^2)$和$\Theta(n^3)$，而$\Theta(n^2) = o(\Theta(n^3))$，即可明确算法$F$在渐近意义上更快.
- 渐近记号不但给出了时间/空间复杂度的一个简单表述，更给出了理论意义上的衡量标准，特别是不依赖于具体机器即可有效地评判算法效率.

从极限考察渐近记号

为了简化讨论，我们仅考虑正函数. 严格说来，我们所讨论的任意“算法性能”函数$A(\cdot)$都必须形如$A\colon\ \mathbb{N}^+ \to \mathbb{R}^+$. 需要指出的是，在算法分析中尽管有些常见的函数可能出现取值为0的情况，但仍然不影响对它们的讨论，如$\log(\cdot)$.

定理

设$f(\cdot)$和$g(\cdot)$均为“算法性能”函数，若自然数$n \to +\infty$时$f(n)/g(n)$存在极限，则有：

- $\lim\limits_{n\to+\infty} \frac{f(n)}{g(n)} > 0 \Longrightarrow f(n) = \Theta\big(g(n)\big)$.
- $\lim\limits_{n\to+\infty} \frac{f(n)}{g(n)} = 0 \Longrightarrow f(n) = o\big(g(n)\big)$.

例

任给常数$\epsilon > 0$, 证明$n \log n = o(n^{1+\epsilon})$.

证明

令$f(n) = n \log n$, $g(n) = n^{1+\epsilon}$, 易知

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = \lim_{n \to +\infty} \frac{\log n}{n^{\epsilon}} = \lim_{x \to +\infty} \frac{\log x}{x^{\epsilon}} = \lim_{x \to +\infty} \frac{1}{\epsilon \ln 2} \frac{1}{x^{\epsilon}} = 0,$$

因此, $n \log n = o(n^{1+\epsilon})$.

务必记住这个极限:

$$\lim_{n \to +\infty} \frac{\log n}{n^{\epsilon}} = 0. \qquad (\epsilon > 0)$$

常用的渐近时间复杂度

设问题的输入规模量为n, 常用的渐近时间复杂度如下:

- $O(1)$. 这称为常数时间(constant time).
- $O(\log n)$. 这称为对数时间(logarithmic time).
- $O(n)$. 这称为线性时间(linear time).
- $O(n \log n)$. 这称为线对时间(linearithmic time).
- $O(n^{1+\epsilon})$, 其中$0 < \epsilon < 1$. 这是个不大的时间复杂度.
- $O(n^2)$. 这称为平方时间(quadratic time).
- $O(n^d)$, 其中$d > 2$. 一般而言, 若算法运行时间$T(n)$是多项式函数, 则称$O(T(n))$为多项式时间(polynomial time).

常用的渐近时间复杂度 (续)

- $O(\lambda^n)$, 其中$\lambda > 1$. 这称为指数时间(exponential time), 最常见的指数时间是$O(2^n)$. 对固定的分支进行蛮力搜索一般会导致指数时间算法.
- $O(n!)$. 这称为阶乘时间(factorial time), 往往对应着组合算法.
- $O(n^n)$. 这是个很大的时间复杂度, 通常只在蛮力法中才会出现.

上述O记号换成Θ记号后在o记号下满足序关系, 例如$\Theta(n^2) = o(\Theta(n!))$. 换言之, 所对应的$\Theta$记号在渐近意义上按照从小到大次序排列.

线对时间算法很快! 如果拿$\Theta(n^{1.0001})$来对比, 对其认识就会比较深刻. $\Theta(n \log n)$比$\Theta(n^{1.0001})$还要快. 由此可凸显$O(n \log n)$时间之快, 它非常"接近"于$O(n)$时间, 因此又有拟线性时间之名, 可见绝非浪得虚名.

例

假定有台每秒能运行千亿条指令的计算机(其性能为100,000MIPS), 当需要执行的指令总数分别为$\log n$, n, $n\log n$, n^2, 2^n, $n!$时, 求出该计算机完成任务所需的运行时间.

解

我们为不同n值计算了相应的运行时间. 对于那些特别长的时间值, 我们以$\Uparrow$来表示.

可以看出, 不同的时间复杂度之间差异非常巨大, 而且随着问题规模量的增长这种差异越发明显. 此外需要注意, 现实中的量级可用常数转换, 例如60分钟则变为1小时, 但算法中不同量级之间却有着无法逾越的鸿沟, 例如$\Theta(n)$和$\Theta(n^2)$, 因为它们之间可用o记号给出序关系.

不同规模量下的算法运行时间

规模量n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	0.03纳秒	0.1纳秒	0.33纳秒	1纳秒	10纳秒	36.3微秒
20	0.04纳秒	0.2纳秒	0.86纳秒	4纳秒	10微秒	282天
30	0.05纳秒	0.3纳秒	1.47纳秒	9纳秒	10毫秒	8.4×10^{13}年
40	0.05纳秒	0.4纳秒	2.13纳秒	16纳秒	11秒	2.6×10^{29}年
50	0.06纳秒	0.5纳秒	2.82纳秒	25纳秒	3.1小时	9.6×10^{45}年
⋮	⋮	⋮	⋮	⋮	⋮	⋮
100	0.07纳秒	1纳秒	6.64纳秒	100纳秒	4×10^{11}年	⇑
1 000	0.10纳秒	10纳秒	99.7纳秒	10微秒	⇑	⇑
10 000	0.13纳秒	100纳秒	1.3微秒	1毫秒	⇑	⇑
100 000	0.16纳秒	1微秒	16.6微秒	100毫秒	⇑	⇑
1 000 000	0.20纳秒	10微秒	199.3微秒	10秒	⇑	⇑
10 000 000	0.23纳秒	100微秒	2.3毫秒	16.7分	⇑	⇑
100 000 000	0.27纳秒	1毫秒	26.6毫秒	27.8小时	⇑	⇑
1 000 000 000	0.30纳秒	10毫秒	0.3秒	116天	⇑	⇑

加和型无穷大量阶的比较

- 由于算法中经常使用循环语句，而此类语句的时间一般表达为加和的形式. 利用Stolz-Cesàro定理会化简问题的分析.
- Stolz-Cesàro定理: 设$f(\cdot)$和$g(\cdot)$均为“算法性能”函数，它们满足如下条件:
  - $g(n)$为递增函数.
  - 当自然数$n \to +\infty$时，$g(n) \to +\infty$.
  - 当自然数$n \to +\infty$时，$(f(n+1)-f(n))/(g(n+1)-g(n))$存在极限.

  那么，

$$\lim_{n\to+\infty}\frac{f(n)}{g(n)}=\lim_{n\to+\infty}\frac{f(n+1)-f(n)}{g(n+1)-g(n)}.$$

例

证明: $\log(n!) = \Theta(n \log n)$.

证明

令$f(n) = \log(n!)$, $g(n) = n \log n$, 在$n > 1$的情况下考察

$$\frac{f(n)}{g(n)} = \frac{\log(n!)}{n \log n},$$

将$f(n)$改写为$\sum_{i=1}^{n} \log i$, 利用Stolz-Cesàro定理可知:

$$\begin{aligned}\lim_{n \to +\infty} \frac{f(n)}{g(n)} &= \lim_{n \to +\infty} \frac{\sum_{i=1}^{n+1} \log i - \sum_{i=1}^{n} \log i}{(n+1)\log(n+1) - n \log n} \\ &= \lim_{n \to +\infty} \left(1 + \frac{\log\left((1+\frac{1}{n})^n\right)}{\log(n+1)}\right)^{-1} = 1\end{aligned}$$

由Θ记号的极限性质定义可知$\log(n!) = \Theta(n \log n)$.

非常重要的两个性质

在分析归并排序等算法时，我们经常需要考虑$\log(n!)$，所以务必记住这个极限：

$$\lim_{n\to+\infty}\frac{\log(n!)}{n\log n}=1.$$

在算法中常用该极限的推论：

$$\log(n!)=\Theta(n\log n).$$

再次回顾e的定义：

$$\lim_{n\to+\infty}\left(1+\frac{1}{n}\right)^n=e.$$

例

设d为正整数, 证明

$$\sum_{k=1}^{n} k^d = \Theta(n^{d+1}).$$

证明

令$f(n) = \sum_{k=1}^{n} k^d$, $g(n) = n^{d+1}$, 考察$f(n)/g(n) = (\sum_{k=1}^{n} k^d)/n^{d+1}$, 该式满足Stolz-Cesàro定理的条件, 可知:

$$\begin{aligned}\lim_{n\to+\infty} \frac{f(n)}{g(n)} &= \lim_{n\to+\infty} \frac{\sum_{k=1}^{n+1} k^d - \sum_{k=1}^{n} k^d}{(n+1)^{d+1} - n^{d+1}} \\ &= \lim_{n\to+\infty} \frac{(n+1)^d}{(n+1)^{d+1} - n^{d+1}} = \frac{1}{d+1},\end{aligned}$$

根据Θ记号的极限性质定义, 本例可得证.

乘积型无穷大量阶的比较

- 递归是算法中另一种大量使用的技术, 不过有时它的效率可能相当低, 如处理难解问题(intractable problem)的算法其性能最坏可能达到$O(n!)$或$O(n^n)$.
- 要了解算法性能是"较差"还是"很差", 必须采用o记号得到更深入的性能对比.
- 由于$O(n!)$和$O(n^n)$都是乘积型的无穷大量, 对此类无穷大量若采用求比值的方法, 可以得到较为简便的分析.

比值判别

定理

设$f(\cdot)$和$g(\cdot)$均为“算法性能”函数. 考虑商函数

$$q(n) = \frac{f(n)}{g(n)},$$

若自然数$n \to +\infty$时比值$q(n+1)/q(n)$存在极限且满足

$$\lim_{n \to +\infty} \frac{q(n+1)}{q(n)} < 1,$$

则有$f(n) = o\left(g(n)\right)$.

证明

此处略去, 有兴趣的同学可参阅教材.

例

设$\lambda > 1$为某一给定的正整数，证明$\lambda^n = o(n!)$.

证明

令$f(n) = \lambda^n$, $g(n) = n!$, 考察

$$q(n) = \frac{f(n)}{g(n)} = \frac{\lambda^n}{n!},$$

求比值$\frac{q(n+1)}{q(n)} = \frac{\lambda}{n+1}$, 易知

$$\lim_{n \to +\infty} \frac{q(n+1)}{q(n)} = 0 < 1,$$

可知$\lambda^n = o(n!)$.

例

证明$2^n n! = o(n^n)$.

证明

令$f(n) = 2^n n!$, $g(n) = n^n$, 考察

$$q(n) = \frac{f(n)}{g(n)} = \frac{2^n n!}{n^n},$$

求比值

$$\frac{q(n+1)}{q(n)} = \frac{2}{\left(1+\frac{1}{n}\right)^n},$$

易知$\lim\limits_{n\to+\infty} \frac{q(n+1)}{q(n)} = \frac{2}{e} < 1$, 本例得证. 此外, $n! = o(n^n)$是显然的.

n^n拥有相当高的阶, 因此$\Theta(n^n)$时间算法相当差. 此外, $n^n = o(3^n n!)$, 其中的差异可由*Stirling*近似来解释, 此处不再详述.

对数型无穷大量阶的比较

对于比较复杂的函数，我们可以同时取对数来简化分析.

定理

设$p(\cdot), q(\cdot), r(\cdot)$均为“算法性能”函数. 它们满足:

- 当自然数$n \to +\infty$时, $p(n) \to +\infty$且$q(n) \to +\infty$.
- $\log p(n) = \Theta\left(r(n)\right)$.
- $\log q(n) = o\left(r(n)\right)$.

则$q(n) = o\left(p(n)\right)$.

证明

此处略去, 有兴趣的同学可参阅教材.

例

使用对数方法重新证明$\lambda^n = o(n!)$.

证明

令$p(n) = n!$, $q(n) = \lambda^n$, $r(n) = n \log n$.

我们已证明

$$\log\left(p(n)\right) = \log(n!) = \Theta(n \log n),$$

而且易证

$$\log\left(q(n)\right) = n \log \lambda = o(n \log n).$$

由定理3可知$\lambda^n = o(n!)$.

可以看出$r(n)$这个函数非常关键, 它起到了参照物的作用.

例

整数排序算法是较为复杂的算法, 关于其研究已经相当深入. 现有两种整数排序算法, 其时间复杂度分别为$O\left(n\frac{\log n}{\log\log n}\right)$ 和$O(n\sqrt{\log n})$, 设$p(n)=n\frac{\log n}{\log\log n}$, $q(n)=n\sqrt{\log n}$, 讨论$p(n)$与$q(n)$的阶关系.

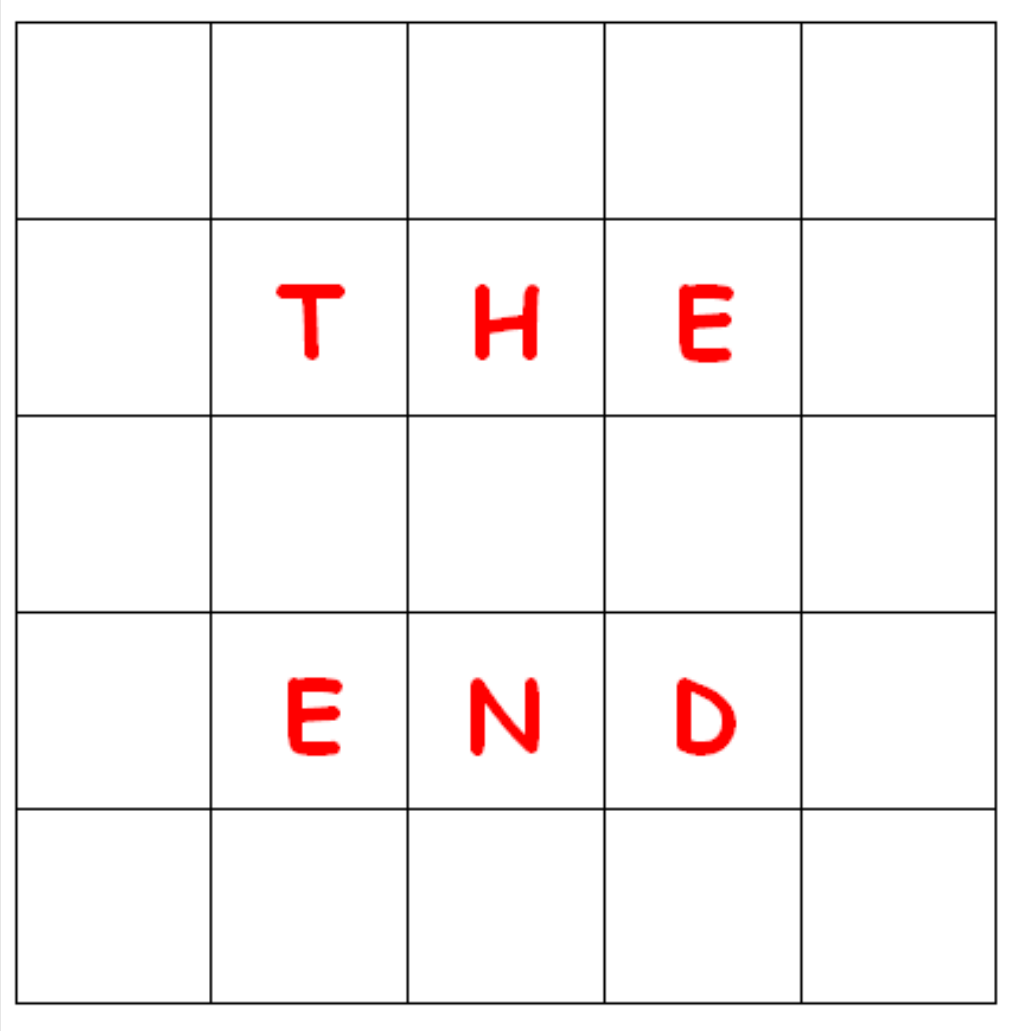
解

直接采用定理3难以获得结果, 注意到需要考察的公式可以化简为

$$\frac{q(n)}{p(n)}=\frac{\log\log n}{\sqrt{\log n}},$$

我们只需分析上式的极限即可. 令$P(n)=\sqrt{\log n}$, $Q(n)=\log\log n$, 取对数后易知$Q(n)=o\left(P(n)\right)$, 进而可知$q(n)=o\left(p(n)\right)$.
这意味着$O\left(n\frac{\log n}{\log\log n}\right)$的算法在最坏情况下可能优于$O(n\sqrt{\log n})$的算法.

本章完结



For more Information, please visit:

PROJECT: DSAD.

WEBSITE: https://github.com/xiexiexx/dsad.

EMAIL: DSAD2015@163.com.

WEIBO: @算海无涯-X.