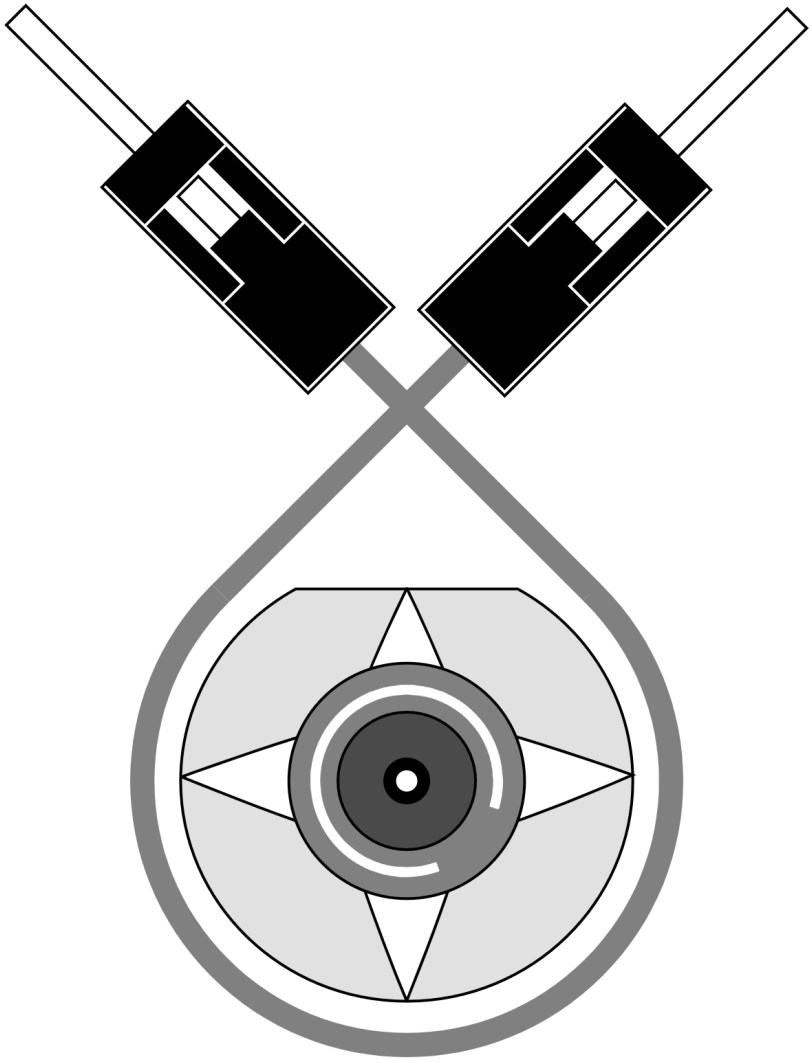
Algorithm Theory

Notes for SE3701: Algorithm Theory



posvirus

目录

Chapter 1 算法原理简介

1.1 算法的定义

在本节中，我们首先给出**算法**（**algorithm**）的定义。为定义算法，我们首先需要定义“**计算**”这一概念：

Definition 1.1

可由给定计算模型机械执行的计算步骤序列称为该模型下的一个**计算**。

进而我们可以给出算法的形式定义：

Definition 1.2

算法是满足下列特征的计算：

- **输入**：必须有 0 个或以上的输入量。
- **输出**：应有一个或以上输出量作为算法计算的结果。
- **确定性**：算法的描述无歧义，每一步都是严格定义和确定的动作。
- **有穷性/终止性**：通常要求算法有限步内必须停止。
- **能行性**：通常要求算法有限步内必须停止。

在给出算法的定义后，我们自然会对各种算法进行分析，其中我们最关注的便是算法的**性能**。但我们又会发现，分析一个算法的性能并不是那么显然的事情，更进一步说，我们现在还没有给出一个统一的量化指标对算法的性能进行量化。

但我们又发现，算法本质上是一种计算，而对某种计算的讨论必然会涉及这种计算是在怎样的计算模型下进行的。也就是说，**在不同的计算模型下，同一算法的性能也会有显著差异**。举一个简单的例子，使用 GPU 与 CPU 同时进行 MVM（矩阵向量乘法，Matrix-vector Multiplication）运算，显然单从时间性能上，GPU 要远优于 CPU，这是因为 GPU 采用并行式运算，而 CPU 仅能进行单线程运算。

而在本课程中，我们定义一种被称为 **RAM 机**（**Random Access Machine**）的简化计算模型，以后提到的所有算法均基于这一计算模型进行性能分析。

Definition 1.3

RAM 机是一种用于算法度量的一种机器无关的模型。RAM 机的处理器为**单线程处理器**，指令按顺序执行，不支持并行操作。RAM 机同时具有以下特点：

- 每个基本操作需要 1 个时间单位（**循环**和**子程序**不是基本操作）。
- 每次存储器访问需要 1 个时间单位（并假设**内存无限**）。

在定义了 RAM 机后，对于给定的算法，其所需的时间由 **RAM 机中的操作所需的时间单位**来度量；所需的空间则由 **RAM 机占用的存储单元的数量**来度量。

1.2 算法的描述

在算法分析中，为防止编程语言的多样性干扰我们对算法的理解，我们常常通过**伪代码**对一个算法进行描述。在 LaTeX 中可以使用 algorithm 与 algorithmic 宏包进行伪代码书写，如 algorithm 1所示。

Algorithm 1 Fourier-Mellin Based KCF

Input: Image I

preprocessed kernelized template T_κ

Output: scale σ, angle θ relation between I and T

1: fourier transform: $F = \mathcal{F}(I)$

2: high pass filter: $F_h = \mathcal{H}(F)$

$\mathcal{H}(x, y) = (1.0 - cos(\pi x)cos(\pi y))(2.0 - cos(\pi x)cos(\pi y))$

3: log-polar transform: $F_{lp} = \mathcal{L}(F_h)$

4: apply kernel function: $F_\kappa = \mathcal{K}(F_{lp})$

5: phase correlation: $(\Delta x, \Delta y) = \mathcal{C}(F_\kappa, T_\kappa)$

6: resolove scale and rotation:

$\theta = \alpha\Delta x, \sigma = log(\Delta y)$

where α is translation factor of pixel translation on fourier domain and polar angle on origin image

另外，在进行算法描述时，需遵循如下原则：

- ✿ 简洁清晰为上，抛弃无关细节，可以使用自然语言描述一个可能很复杂的过程。
- ✿ 伪代码只关心流程的正确性，不关心可否编译，变量等无须声明可直接使用。
- ✿ 伪代码不关心软件工程问题，常忽略数据抽象、模块性和错误处理。

事实上，我们也可对算法进行一个更为形式化的描述：

Definition 1.4

对一个待解决的问题 R，设问题 R的输入集合为 $Input$，输出集合为 $Output$，则算法 $\mathscr{F}$即建立了从 $Input$到 $Output$的映射：

$$Input \xrightarrow{\mathscr{F}} Output$$

1.3 算法的分析

在完成对一个算法的描述后，我们便可进一步对该算法进行分析，对算法的分析主要有 3 个部分：**算法的正确性分析**、**算法的复杂度分析**与**算法的实验分析**。

1.3.1 算法的正确性分析

要使一个算法可以被应用，首先要保证其**正确性**。算法的正确性被定义如下：

Definition 1.5

称一个算法具有**正确性**，或一个算法是正确的，当且仅当其对于每一个输入都最终停止并产生正确的输出结果。

反之，若一个算法不正确，则其应具有以下两条特征之一：

- ✿ 对所有或部分输入不停止。
- ✿ 对所有输入停止，但对某些输入产生错误的输出。

那么，是不是对于每一个算法，我们都要严格保证其正确性呢？答案是否定的。事实上，基于对实际问题求解精度与求解时间的考量，我们常常可以放宽对算法正确性的要求，只要求算法“**不那么错误**”，以此换取算法求解的高效率，**近似算法**与**随机算法**便是两个典型。

Definition 1.6

近似算法满足对所有输入都停止，且对所有输入产生近似正确的解。

Definition 1.7

随机算法满足对所有输入都停止，可能产生错误解，但**与输入不相关**，且错误解的产生**可控**。

一般来说，算法的正确性需要通过严格的数学证明才能给出，以下，我们给出算法正确性分析中一种称为**循环不变式判定法**的常用方法。

Theorem 1.1

循环不变式（***loop invariant***）是一组在循环体内、每次迭代均保持为真的性质（表达式），主要用来说明算法的正确性，循环不变式具有 3 条性质：

1. 在循环第一次迭代之前循环不变式为真。
2. 如果循环的某次迭代之前它为真，那么下次迭代之前它仍为真。
3. 循环中止时依然为真，且为我们提供一个有助于证明算法正确性的性质。

☞ **Remark**

循环不变式判定法与**数学归纳法**类似，但是循环不变式判定法需要对迭代中止时循环不变式的真值进行判定，而数学归纳法的迭代次数是无限的。

1.3.2 算法的复杂度分析

算法的复杂度分析是为了预测算法对不同规模输入[(1)]所需要的资源，提供衡量算法优劣的测度手段，为算法选择提供依据。更准确地说，算法的复杂度分析最终会得出算法所需资源随输入的变化函数。按算法所需的资源分，算法的复杂度可分为**时间复杂度**、**空间复杂度**、**I/O 复杂度**等。在此，我们只考查**时间复杂度**与**空间复杂度**。

Definition 1.8

一个算法对特定输入的**时间复杂度**是该算法对该输入产生结果需要的原子操作的“步”数。

☞ **Remark**

此处隐含定义了时间复杂度是输入规模的函数，且假设每一步的执行需要**常数时间**。

Definition 1.9

一个算法对特定输入的**空间复杂度**是该算法对该输入产生正确结果所需要存储空间的大小。

☞ **Remark**

本课程所指的算法复杂度不是分析每种操作的精确个数，而是重点研究在**输入规模足够大的情况下，算法复杂度的增长趋势**，即**渐进复杂度**。

[(1)] **输入规模** N（$N \in \mathbb{N}_+$）是用于衡量算法输入大小的数学量，可通过一个函数由 $Input$ 映射得到 $F : Input \to N$，比如矩阵输入可用矩阵维数作为输入规模，排序问题可用元素数量作为输入规模等

针对某一资源对应的复杂度分析，还有以下概念值得了解：

Definition 1.10

设 $Input$ 是问题 R 的输入集合，$Complexity(X)$ 是求解 R 的算法 A 的复杂度函数，$|y|$ 是 R 的输入规模，则 A 的**最坏复杂度**定义为：

$$\max\{Complexity(|y|)|\forall y \in Input\}$$

最好复杂度定义为：

$$\min\{Complexity(|y|)|\forall y \in Input\}$$

Definition 1.11

设 $Input$ 是问题 R 的输入集合，$Complexity(X)$ 是求解 R 的算法 A 的复杂度函数，$|y|$ 是 R 的输入规模，y 作为 A 的输入的概率为 p_y，则 A 的**平均复杂度**定义为：

$$\sum_{y \in Input} Complexity(|y|) \times p_y$$

Definition 1.12

摊销复杂度一般不用于分析算法单次运行的复杂度，而是用于分析数据结构所执行的一系列操作的复杂度。

1.3.3 算法的实验分析

对于相当多的算法来说，形式化的证明与严格的理论分析相当困难且无必要。而**实验分析**通过将算法实际实现，并加载实际数据，可通过实验简便地观察算法的性能变化，算法的实验分析主要包含以下步骤：

- ✿ 确定实验目的。
- ✿ 决定度量效率的标准和度量单位（操作次数/绝对时间）。
- ✿ 准备输入样本（范围和大小）。
- ✿ 用程序实现待考察算法。
- ✿ 运行算法并加载输入样本，记录实验数据。
- ✿ 分析获得的实验数据。

Chapter 2 算法复杂度分析的数学基础

2.1 复杂度函数的阶

在 section 1.3.2中我们已经说明，本课程这所研究的算法复杂度为**渐进复杂度**，这种复杂度仅关心在输入规模足够大时的算法复杂度“**增长趋势**”，为方便对“增长趋势”进行更为精确、量化性的描述，在此我们记 $f(n)$为一个一般的复杂度函数，并基于此给出**复杂度函数的阶**的定义。

2.1.1 同阶函数集合：$\Theta(f(n))$

Definition 2.1

与 $f(n)$同阶的函数集合 $\Theta(f(n))$定义为：

$$\Theta(f(n)) = \{g(n) | \exists c_1, c_2 > 0,\ n_0,\ \forall n > n_0,\ c_1 f(n) \leqslant g(n) \leqslant c_2 f(n)\}$$

☞ **Remark**

注意复杂度函数 $f(n)$的一个天然属性：

$$f(n) \geqslant 0,\ \forall n \in |Input|$$

因此 $\Theta(f(n))$的定义也可改写为：

$$\Theta(f(n)) = \{g(n) | \exists c_1, c_2 > 0,\ n_0,\ \forall n > n_0,\ c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\}$$

同时，基于上述分析，我们不难得到以下结论：

Theorem 2.1

若存在 $f(n)$与 $g(n)$，使 $f(n) \in \Theta(g(n))$，则 $g(n) \in \Theta(f(n))$。

特别地，虽然我们定义的 $\Theta(f(n))$为一个集合，但我们在表示与 $f(n)$同阶的函数 $g(n)$时，常采用“$=$”替代“$\in$”，即：

$$g(n) = \Theta(f(n)) (\Leftrightarrow g(n) \in \Theta(f(n)))$$

另外，我们在实际复杂度分析时，对复杂度函数为**多项式**（**polynomial**）的算法很感兴趣，对这类复杂度函数，我们有以下定理：

Theorem 2.2

对于任意 $d \in \mathbb{N}_+$与任意常数 $a_d > 0$，成立：

$$p(n) = \sum_{i=0}^{d} a_i n^i = \Theta(n^d)$$

证明. 首先，注意到：

$$p(n) = \sum_{i=0}^{d} a_i n^i \leqslant \sum_{i=0}^{d} |a_i| n^i \leqslant (\sum_{i=0}^{d} |a_i|) n^d$$

其次，任取 c_1使 $a_d > c_1 > 0$，有：

$$p(n)-c_1n^d=(a_d-c_1)n^d+\sum_{i=0}^{d-1}a_in^i=(a_d-c_1)n^d(1+\sum_{i=0}^{d-1}\frac{a_i}{a_d-c_1}n^{i-d})$$

进而就有：

$$p(n)-c_1n^d\geqslant(a_d-c_1)n^d(1-\sum_{i=0}^{d-1}\left\lceil\left|\frac{a_i}{a_d-c_1}\right|\right\rceil n^{i-d})$$

我们可以通过解不等式组：

$$\left\lceil\left|\frac{a_i}{a_d-c_1}\right|\right\rceil n^{i-d}\geqslant\frac{1}{d},\ 1\leqslant i\leqslant d-1$$

进而取：

$$n_0=\left\lceil\max_{1\leqslant i\leqslant d-1}\{\sqrt[d-i]{d\left\lceil\left|\frac{a_i}{a_d-c_1}\right|\right\rceil}\}\right\rceil$$

可知对 $n > n_0$，有 $p(n)-c_1n^d\geqslant 0$，于是，对 $c_1 > 0, c_2=\sum_{i=0}^{d}|a_i| > 0$，当 $n > n_0$时，成立 $c_1n^d\leqslant p(n)\leqslant c_2n^d$，因此 $p(n)=\Theta(n^d)$，证毕。 ■

若 $f(n)=\Theta(n^k)$（$k\in\mathbb{N}$），则称 $f(n)$为**多项式界限**的。

2.1.2 低阶函数集合：$O(f(n))$

Definition 2.2

比 $f(n)$低阶的函数集合 $O(f(n))$定义为：

$$O(f(n))=\{g(n)|\exists c>0,\ n_0,\ \forall n>n_0,\ 0\leqslant g(n)\leqslant cf(n)\}$$

一般地，若 $g(n)=O(f(n))$，则我们称 $f(n)$为 $g(n)$的**上界**。同时，不难得出以下事实：

Theorem 2.3

对任意 $f(n)$，成立：

$$\Theta(f(n))\subseteq O(f(n))$$

证明. 由于对 $\forall g(n)\in\Theta(f(n))$，$\exists c_1, c_2 > 0,\ n_0$，使得 $n > n_0$时，$c_1f(n)\leqslant g(n)\leqslant c_2f(n)$。

于是对 $c=c_2 > 0$，n_0，当 $n > n_0$时，有 $0\leqslant g(n)\leqslant cf(n)$，因此 $g(n)\in O(f(n))$，进而由 $g(n)$的任意性，得 $\Theta(f(n))\subseteq O(f(n))$，证毕。 ■

2.1.3 高阶函数集合：$\Omega(f(n))$

Definition 2.3

比 $f(n)$高阶的函数集合 $\Omega(f(n))$定义为：

$$\Omega(f(n))=\{g(n)|\exists c>0,\ n_0,\ \forall n>n_0,\ 0\leqslant cf(n)\leqslant g(n)\}$$

一般地，若 $g(n)=\Omega(f(n))$，则我们称 $f(n)$为 $g(n)$的**下界**。同理，我们有 $\Omega(f(n))\subseteq\Theta(f(n))$成立。进而，我们有以下结论：

Theorem 2.4

对任意 $f(n)$与 $g(n)$，$f(n)=\Theta(g(n))$当且仅当（*iff.*）$f(n)=O(g(n))$且$f(n)=\Omega(g(n))$。

证明. 分为**充分性**（$\Rightarrow$）与**必要性**（$\Leftarrow$）进行证明。

$\Rightarrow$：已知 $f(n)=\Theta(g(n))$，因此 $\exists c_1, c_2>0,\ n_0$，当 $n>n_0$时，成立：

$$c_1 g(n) \leqslant f(n) \leqslant c_2 g(n) \tag{2.1}$$

观察 eq. (2.1) 的左半部分，可知 $\exists c_1>0,\ n_0$，当 $n>n_0$时，成立：

$$0 \leqslant c_1 g(n) \leqslant f(n)$$

因此 $f(n)=\Omega(g(n))$，同理观察 eq. (2.1) 的右半部分，可得 $f(n)=O(g(n))$，充分性得证。

$\Leftarrow$：已知 $f(n)=O(g(n))$且 $f(n)=\Omega(g(n))$，则：

$$\exists c_1>0,\ n_1,\ \forall n>n_1,\ 0 \leqslant c_1 g(n) \leqslant f(n)$$

$$\exists c_2>0,\ n_2,\ \forall n>n_2,\ 0 \leqslant f(n) \leqslant c_2 g(n)$$

因此有：

$$\exists c_1, c_2>0,\ n_0=\max\{n_1,n_2\},\ \forall n>n_0,\ c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

此即为 $f(n)=\Theta(g(n))$，必要性得证，进而原命题得证。 ■

Example 2.1

证明：$\log n! = \Theta(n\log n)$

Solution

首先，注意：

$$\log n! = \sum_{k=1}^{n} \log k \leqslant n\log n$$

因此 $\log n! = O(n\log n)$，另一方面：

$$\log n! = \sum_{k=1}^{n} \log k \geqslant \sum_{k=\lceil n/2\rceil}^{n} \log k \geqslant (n-\left\lceil\frac{n}{2}\right\rceil+1)\log\frac{n}{2},\ n\geqslant 4$$

进而有：

$$\log n! \geqslant \frac{n}{2}\log\frac{n}{2} \geqslant \frac{1}{4}n\log n,\ n\geqslant 4$$

因此 $\log n! = \Omega(n\log n)$，由Theorem 2.4知 $\log n! = \Theta(n\log n)$，证毕。

2.1.4 严格低阶函数集合：$o(f(n))$

Definition 2.4

比 $f(n)$严格低阶的函数集合 $o(f(n))$定义为：

$$o(f(n)) = \{g(n) | \forall c > 0,\ \exists n_0,\ \forall n > n_0,\ 0 \leqslant g(n) \leqslant cf(n)\}$$

☞ **Remark**

Definition 2.4中的 n_0可以与任意正常数 c有关。

一般地，若 $g(n) = o(f(n))$，则我们称 $f(n)$为 $g(n)$的严格上界。

Example 2.2

证明：$f(n) = o(g(n)) \Rightarrow \lim\limits_{n\to+\infty} \dfrac{f(n)}{g(n)} = 0$

Solution

由于 $f(n) = o(g(n))$，因此：

$$\forall \varepsilon > 0,\ \exists n_0,\ \forall n > n_0,\ 0 \leqslant f(n) \leqslant \varepsilon g(n)$$

此亦即：

$$\forall \varepsilon > 0,\ \exists n_0,\ \forall n > n_0,\ 0 \leqslant \frac{f(n)}{g(n)} \leqslant \varepsilon$$

进而就有 $\lim\limits_{n\to+\infty} \dfrac{f(n)}{g(n)} = 0$，证毕。

2.1.5 严格高阶函数集合：$\omega(f(n))$

Definition 2.5

比 $f(n)$严格高阶的函数集合 $\omega(f(n))$定义为：

$$\omega(f(n)) = \{g(n) | \forall c > 0,\ \exists n_0,\ \forall n > n_0,\ 0 \leqslant cf(n) \leqslant g(n)\}$$

一般地，若 $g(n) = \omega(f(n))$，则我们称 $f(n)$为 $g(n)$的严格下界。

2.2 渐进符号的统一性质

在 section 2.1中，我们给出了函数的阶的定义以及一些相关性质，作为渐进复杂度分析的基础。一般地，我们也称 $\Theta, O, o, \Omega, \omega$这类符号为**渐进符号**，称 $\Theta(f(n)), O(f(n)), o(f(n)), \Omega(f(n)), \omega(f(n))$这类函数集合为**渐进表达式**，$f(n)$则称为渐进函数，比如 $g(n) = O(n^2)$，我们就称 $g(n)$对应的一个渐进表达式为 $O(n^2)$，$g(n)$对应的一个渐进函数为 n^2。

以下，我们以 Γ抽象地表示各类渐进符号，讨论渐进符号所具有的统一性质。

Property 2.1

若 $f_1(n) = \Gamma(g_1(n))$且 $f_2(n) = \Gamma(g_2(n))$，则：

$$f_1(n) + f_2(n) = \Gamma(g_1(n) + g_2(n)),\ \Gamma = \Theta, O, \Omega$$

证明. 首先，以 $\Gamma=\Theta$为例进行证明，基于 $f_1(n)=\Theta(g_1(n))$与 $f_2(n)=\Theta(g_2(n))$，有：

$$\exists c_{11},c_{12}>0,\ n_1,\ \forall n>n_1,\ c_{11}g_1(n)\leqslant f_1(n)\leqslant c_{12}g_1(n)$$

$$\exists c_{21},c_{22}>0,\ n_2,\ \forall n>n_2,\ c_{21}g_1(n)\leqslant f_2(n)\leqslant c_{22}g_1(n)$$

于是对 $n>\max\{n_1,n_2\}$，有：

$$c_{11}g_1(n)+c_{21}g_2(n)\leqslant f_1(n)+f_2(n)\leqslant c_{12}g_1(n)+c_{22}g_2(n)$$

取 $c_1=\max\{c_{11},c_{21}\},c_2=\max\{c_{12},c_{22}\}$，可知：

$$c_1(g_1(n)+g_2(n))\leqslant f_1(n)+f_2(n)\leqslant c_2(g_1(n)+g_2(n)),\ \forall n>\max\{n_1,n_2\}$$

这就说明：$f_1(n)+f_2(n)=\Theta(g_1(n)+g_2(n))$，证毕。 ■

☞ **Remark**

对 $\Gamma=O$，进一步有 $f_1(n)+f_2(n)=O(\max\{g_1(n),g_2(n)\})$；对 $\Gamma=\Omega$，进一步有 $f_1(n)+f_2(n)=\Omega(\min\{g_1(n),g_2(n)\})$。

Property 2.2

传递性：

$$f(n)=\Gamma(g(n))\wedge g(n)=\Gamma(h(n))\Rightarrow f(n)=\Gamma(h(n)),\ \Gamma=\Theta,O,o,\Omega,\omega$$

Property 2.3

自反性：

$$f(n)=\Gamma(f(n)),\ \Gamma=\Theta,O,\Omega$$

Property 2.4

转置对称性：

$$f(n)=O(g(n))\Leftrightarrow g(n)=\Omega(f(n))$$

$$f(n)=o(g(n))\Leftrightarrow g(n)=\omega(f(n))$$

证明. 略。 ■

☞ **Remark**

对两个复杂度函数 $f(n)$或 $g(n)$，其关系不一定居于：

$$g(n)=\Theta(f(n)),\ g(n)=O(f(n)),\ g(n)=o(f(n)),\ g(n)=\Omega(f(n)),\ g(n)=\omega(f(n))$$

这五者之一，比如 $f(n)=n$与 $g(n)=n^{1+\sin n}$。

2.3 复杂度分析的一些数学技巧

2.3.1 Floor 函数与 Ceiling 函数

Floor 函数“$\lfloor\rfloor$”与 **Ceiling** 函数“$\lceil\rceil$”是进行复杂度分析时常用的两种函数，以下给出这两个函数所具有的一些性质。

Property 2.5

对任意 $n \in \mathbb{Z}$，有：

$$\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n$$

证明. 分类讨论即可：

若 $n = 2k$，$k \in \mathbb{Z}$，则 $\left\lceil \frac{n}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor = k$，有 $\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n$；

若 $n = 2k+1$，$k \in \mathbb{Z}$，则 $\left\lceil \frac{n}{2} \right\rceil = k+1$，$\left\lfloor \frac{n}{2} \right\rfloor = k-1$，有 $\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n$，证毕。 ■

Property 2.6

对任意 $a, b, n \in \mathbb{Z}$，$a, b \neq 0$，有：

$$\left\lceil \frac{\lceil n/a \rceil}{b} \right\rceil = \left\lceil \frac{n}{ab} \right\rceil, \quad \left\lfloor \frac{\lfloor n/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{n}{ab} \right\rfloor$$

证明. 以对 $\left\lceil \frac{\lceil n/a \rceil}{b} \right\rceil = \left\lceil \frac{n}{ab} \right\rceil$ 的证明为例，分类讨论：

若 $n = kab$，$k \in \mathbb{Z}$，则显然成立；

若 $n = kab + \alpha$，$k \in \mathbb{Z}$ 且 $\alpha \in (0, ab)$，则：

$$\text{LHS} = \left\lceil \frac{\lceil kb + \alpha/a \rceil}{b} \right\rceil = k + \left\lceil \frac{\lceil \alpha/a \rceil}{b} \right\rceil \xrightarrow{\alpha/a \in (0,b)} k = \text{RHS}$$

■

2.3.2 积分法求解复杂度：一个例子

“积分法”的理论依据：

Theorem 2.5

✿ 若 $f(n)$ 单调递增，则：

$$\int_{m-1}^{n} f(x)\mathrm{d}x \leqslant \sum_{k=m}^{n} f(k) \leqslant \int_{m}^{n+1} f(x)\mathrm{d}x$$

✿ 若 $f(n)$ 单调递减，则：

$$\int_{m}^{n+1} f(x)\mathrm{d}x \leqslant \sum_{k=m}^{n} f(k) \leqslant \int_{m-1}^{n} f(x)\mathrm{d}x$$

积分法的复杂度求解示例：

Example 2.3

证明：$H(n) = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1)$

Solution

记 $f(x)=\frac{1}{x}$，则：

$$H(n)=\sum_{k=1}^{n} f(k) \geqslant \int_{1}^{n+1} \frac{\mathrm{d}x}{x} > \ln n$$

同时由：

$$H(n)=1+\sum_{k=2}^{n} f(k) \leqslant 1+\int_{1}^{n} \frac{\mathrm{d}x}{x} = 1+\ln n$$

因此：$H(n)=\ln n+O(1)$，证毕。

2.4 递归算法复杂度分析：以分治算法为例

递归算法（**recursion algorithm**）是一类特殊的算法，这类算法在执行时会对自身进行调用，因而其复杂度函数会根据算法具体进行递归调用的过程而满足一个特定的**递推关系/递推方程**（recurrence relation），对递归算法复杂度的分析通常从求解递推关系入手。

此处，由于还未涉及具体的算法知识，因此我们以递归算法的一种——**分治算法**为例，对其复杂度进行分析。一般地，分治算法复杂度函数 $T(n)$所满足的递推关系通常具有如 eq. (2.2) 所示的统一形式：

$$\begin{cases} T(n)=aT(n/b)+f(n) & n \geqslant 2 \\ T(n)=c & n=1 \end{cases} \tag{2.2}$$

其中，$T(n)=aT(n/b)+f(n)$代表分治算法可将一个输入规模为 n的问题分为 a个输入规模为 n/b的子问题，在分治的过程中，额外引入的复杂度为 $f(n)$，而 $T(1)$为**初始条件**（initial condition），代表输入规模为 1时对应的复杂度，$T(1)$应该是一个常数 c。

☞ **Remark**

eq. (2.2) 只是一个形式化的记法，事实上 n/b可能并不为整数，如需完全精确地表达可能需要增加繁琐的“$\lceil\rceil$”与“$\lfloor\rfloor$”函数，我们这里将 n/b解释为 $\lfloor n/b\rfloor$或 $\lceil n/b\rceil$，也就是说 $T(n/b), T(\lfloor n/b\rfloor), T(\lceil n/b\rceil)$这三种写法不会影响递推方程的渐进性质，在 section 2.4.3中将解释这一结论。

对分治算法递推关系的求解主要有 3 种方法：**迭代法**、**代入法**与**主定理法**，以下将分进行别说明。

2.4.1 迭代法

迭代法即是通过对递推关系的逐次迭代，最终直接推出 $T(n)$的渐进表达式，下面举几个例子：

Example 2.4

求解递推方程：$T(n)=2T(n/2)+cn$

☞ **Remark**

初始条件 $T(1)=c$对 $T(n)$在充分大时的增长趋势一般无影响，因此无需给出。

Solution

直接对该递归方程进行迭代：

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 8T(n/8) + 3cn \\ &= \cdots \\ &= 2^k T(n/2^k) + kcn \end{aligned}$$

令 $n/2^k = 1$得 $k = \log_2 n$，于是就有：

$$T(n) = 2^{\log n} T(1) + cn \log n = \Theta(n \log n)$$

☞ **Remark**

在书写对数时，由于存在关系：

$$\log_a n = \log_a b \cdot \log_b n$$

因此我们可以直接将对数简化地记作 $\log n$，而不关心其底数。

Example 2.5

求解递推方程：$T(n) = 3T(\lfloor n/4 \rfloor) + n$

☞ **Remark**

这里的递推方程中含有 Floor 函数 “$\lfloor\rfloor$”，说明这个递推方程大概率是精确关系式，我们在求解时也要从较为严谨的数学角度出发。

Solution

直接对该递归方程进行迭代：

$$\begin{aligned} T(n) &= 3T(\lfloor n/4 \rfloor) + n \\ &= 9T(\lfloor n/16 \rfloor) + 3\left\lfloor \frac{n}{4} \right\rfloor + n \\ &= 27T(\lfloor n/64 \rfloor) + 9\left\lfloor \frac{n}{16} \right\rfloor + 3\left\lfloor \frac{n}{4} \right\rfloor + n \\ &= \cdots \\ &= 3^k T(\lfloor n/4^k \rfloor) + \sum_{i=0}^{k-1} 3^i \left\lfloor \frac{n}{4^i} \right\rfloor \end{aligned}$$

不妨记 $4^k \leqslant n < 4^{k+1}$，于是 $\log_4 n - 1 < k \leqslant \log_4 n$，进而 $k = \lceil \log_4 n \rceil - 1$，知：

$$\begin{aligned} T(n) &= 3^{\lceil \log_4 n \rceil - 1} T(\lfloor n/4^k \rfloor) + \sum_{i=0}^{\lceil \log_4 n \rceil - 2} 3^i \left\lfloor \frac{n}{4^i} \right\rfloor \\ &\sim nT(1) + \sum_{i=0}^{+\infty} \left(\frac{3}{4}\right)^i n = \Theta(n) \end{aligned}$$

这里只能使用同阶符号 $\sim$进行分析，不是严格相等。

2.4.2 代入法

代入法指的是首先对递推方程的解的形式进行猜测，再通过**待定系数法**与**数学归纳法**证明解的正确性，代入法技巧性较高，对数学直觉的要求也较高，此处不作太多讨论，仅举几例进行说明。

Example 2.6

已知递推方程：$T(n)=2T(n/2)+n$，求解 $T(n)$的上界。

Solution

猜测 $T(n)=O(n\log n)$，则以下证明对适当选取的 $c>0$，当 n适当大时，有：

$$0\leqslant T(n)\leqslant cn\log n \tag{2.3}$$

使用数学归纳法证明，假设 eq. (2.3) 对 $\forall k,\ 0<k<n$成立，则 $k=n$时有：

$$T(n)=2T(n/2)+n\leqslant cn\log\frac{n}{2}+n=cn\log n+(1-c\log 2)n$$

可知对 $c>\log^{-1}2>0$，有 $T(n)\leqslant cn\log n$，进而可知 $T(n)=O(n\log n)$。

☞ Remark

一般地，我们所说的“**求解递推方程**”指的是求解 $T(n)=\Theta(f(n))$，而对于其余的渐进表达式，如 $O(f(n))/\Omega(f(n))$，我们一般会直接说明求解 $T(n)$的上界/下界。

Example 2.7 （添加低阶项）

求解下述递推方程：

$$T(n)=\begin{cases}d & n=1\\ T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+n & n>1\end{cases}$$

Solution （方法 1）

观察递推方程的形式，不难猜测 $T(n)=\Theta(n\log n)$。以下，我们希望使用数学归纳法对该猜测进行证明。我们不妨先按Example 2.6给出的求解方法，假设 $\exists c_1,c_2>0$，对 $\forall k,\ 0<k<n$，成立：

$$c_1k\log k\leqslant T(k)\leqslant c_2k\log k \tag{2.4}$$

则对 $k=n$，有：

$$\begin{aligned}T(n)\ &\geqslant c_1\left\lceil\frac{n}{2}\right\rceil\log\left\lceil\frac{n}{2}\right\rceil+c_1\left\lfloor\frac{n}{2}\right\rfloor\log\left\lfloor\frac{n}{2}\right\rfloor+n\\ &\geqslant c_1\frac{n}{2}\log\frac{n}{2}+c_1(\frac{n}{2}-1)\log(\frac{n}{2}-1)+n\end{aligned}$$

整理后可得：

$$\begin{aligned} T(n) &\geqslant n(1+\frac{c_1}{2}\log\frac{n}{2}+\frac{c_1}{2}\log(\frac{n}{2}-1))-c_1\log(\frac{n}{2}-1) \\ &= c_1 n\log n+\left[n(1-\frac{c_1}{2}\log 2+\frac{c_1}{2}\log(\frac{1}{2}-\frac{1}{n}))-c_1\log(\frac{n}{2}-1)\right] \\ &\geqslant c_1 n\log n+\left[n(1-\frac{c_1}{2}\log 2+\frac{c_1}{2}\log\frac{1}{6})-c_1\log(\frac{n}{2}-1)\right],\ n\geqslant 3 \\ &= c_1 n\log n+\left[n(1-\frac{c_1}{2}\log 12)-c_1\log(\frac{n}{2}-1)\right] \end{aligned}$$

因此，当 $c_1<\dfrac{2}{\log 12}$ 时，必存在 n_0，当 $n>n_0$ 时，$T(n)\geqslant c_1 n\log n$，此时我们仅需将归纳假设改为对 $\forall k,\ n_0<k<n$ 时进行归纳即可，同理可以证明 eq. (2.4) 的右半边，进而 $T(n)=\Theta(n\log n)$。

Solution （方法 2）

以上方法的技巧性较强，事实上，我们可以通过**添加低阶项**的方式来简化证明过程，所谓添加低阶项，就是在归纳假设时不再仅将 $cn\log n$ 作为 $T(n)$ 的上/下界，而使用 $cn\log n+dn+e$ 之类的更一般的形式作为 $T(n)$ 的上/下界。以下，我们以对 $T(n)$ 上界的确定为例进行说明：

假设 $\exists c,d>0$，对 $\forall k,\ 0<k<n$，成立：

$$T(k)\leqslant ck\log k+dk$$

则对 $k=n$，有：

$$\begin{aligned} T(n) &\leqslant c\left\lceil\frac{n}{2}\right\rceil\log\left\lceil\frac{n}{2}\right\rceil+c\left\lfloor\frac{n}{2}\right\rfloor\log\left\lfloor\frac{n}{2}\right\rfloor+d\left\lceil\frac{n}{2}\right\rceil+d\left\lfloor\frac{n}{2}\right\rfloor+n \\ &= c\left\lceil\frac{n}{2}\right\rceil\log\left\lceil\frac{n}{2}\right\rceil+c\left\lfloor\frac{n}{2}\right\rfloor\log\left\lfloor\frac{n}{2}\right\rfloor+(d+1)n \\ &\leqslant c(\frac{n}{2}+1)\log(\frac{n}{2}+1)+c\frac{n}{2}\log\frac{n}{2}+(d+1)n \\ &= cn\log n+dn+\left[n(1+\frac{c}{2}\log(\frac{1}{4}+\frac{1}{2n}))+c\log(\frac{n}{2}+1)\right] \\ &\leqslant cn\log n+dn+\left[n(1-\frac{c}{2}\log\frac{4}{3})+c\log(\frac{n}{2}+1)\right] \end{aligned}$$

之后的证明步骤便与之前类似，只需保证 $[\cdots]<0$ 即可，此处不再赘述。

☞ Remark
这样看来两种方法似乎差别不大 · · ·

Example 2.7既可使用标准代入法求解，也可加入**添加低阶项**的技巧，但以下给出的Example 2.8仅能通过**添加低阶项**来解决：

Example 2.8

已知递推方程：$T(n)=4T(n/2)+n$，求解 $T(n)$的上界。

Solution

猜测 $T(n)=O(n^2)$，假设我们的归纳假设选取的是 $T(k)\leqslant ck^2$，则：

$$T(n)\leqslant 4c\left(\frac{n}{2}\right)^2+n=cn^2+n(\geqslant cn^2)$$

这样就无法进行归纳了。因此，我们设 $\exists c,d>0$，对 $\forall k,\ 0<k<n$，有：

$$T(k)\leqslant ck^2-dk$$

于是 $k=n$时，有：

$$T(n)\leqslant 4c\left(\frac{n}{2}\right)^2-dn+(1-d)n$$

因此，当 $d>1$时，有 $T(n)\leqslant cn^2-dn$成立，进而归纳成立，可知 $T(n)=O(n^2)$。

Example 2.9 （含有非规则项）

求解下述递推方程的上界：

$$T(n)=\begin{cases}d & n\leqslant 34\\ 2T(\frac{n}{2}+17)+n & n>34\end{cases}$$

上述递推方程中含有 $\frac{n}{2}+17$这样一个非规则项，以下，我们给出相关的 3 种求解方法：

Solution （方法 3）

直接假设 $\exists c>0$，对 $\forall k,\ k<n$成立：

$$T(k)\leqslant ck\log k$$

再使用标准的代入法求解即可，此处不做赘述。

Solution （方法 2）

使用**变量替换**的方法进行求解：
记 $n=m+34$，则 $T(m+34)=2T(m/2+34)+m+34$，记 $S(m)=T(m+34)$，则：

$$S(m)=2S(m/2)+m+34$$

于是可知 $S(m)=O(m\log m)$，进而 $T(n)=O((n-34)\log(n-34))=O(n\log n)$。

Solution

对非规则项进行**凑配**，这个方法技巧性较高，仅作参考：
假设 $\exists c>0$，对 $\forall k,\ 34<k<n$成立：

$$T(k)\leqslant c(k-34)\log(k-34)$$

于是对 $k=n$，有：

$$\begin{aligned}T(n) &\leqslant 2c(n/2-17)\log(n/2-17)+n\\ &= c(n-34)\log(n-34)+(1-c\log 2)n+34c\log 2\end{aligned}$$

可知仅需取 c使 $(1-c\log 2)n+34c\log 2<0$即可。

有关变量替换，我们可以另举一例，如Example 2.10所示。

Example 2.10

求解递推方程：$T(n)=2T(\lfloor\sqrt{n}\rfloor)+\log n$

Solution

令 $m=\log_2 n$，则 $T(2^m)\sim 2T(2^{m/2})+m\log 2$，记 $S(m)=T(2^m)$，则有：

$$S(m)=2S(m/2)+m\log 2$$

进而知 $S(m)=\Theta(m\log m)$，可知 $T(n)=\Theta(\log n\cdot\log\log n)$。

☞ **Remark**

Example 2.10中的换元其实有一个值得讨论的地方，在置换元 $m=\log_2 n$后，我们其实只是证明了 $\{T(1),T(2),T(4),\cdots,T(2^k)\cdots\}$这个序列是呈 $\log n\cdot\log\log n$的速度增长的，但对于中间的缺项却没有讨论，这主要是因为我们一般认为：**除复杂度为常数的算法外，大部分算法的复杂度函数都会随输入规模的增加而递增**（复杂度为常数的算法则代表其复杂度与输入规模无关，而并非是一个常数）。

2.4.3 主定理法

综合 section 2.4.1与 section 2.4.2，我们似乎已经可以从主观上，对一些满足具有特定形式的递推方程的复杂度函数的渐进表达式作出估计，但我们仍缺乏相关的理论依据，而**主定理法**便给出了这一依据。

主定理（**master theorem**）给出了求解一类递推方程的普适性结论，在符合定理使用条件时可以快速地直接得出复杂度分析结果。

Theorem 2.6 （主定理）

已知递推方程：

$$T(n)=aT\left(\frac{n}{b}\right)+f(n) \tag{2.5}$$

其中 $a \geqslant 1, b > 1$，$f(n)$为非负函数，eq. (2.5) 对一切非负整数 n均成立，且我们将 n/b解释为 $\lfloor n/b \rfloor$或 $\lceil n/b \rceil$，则：

1. 若 $f(n) = O(n^{\log_b a - \varepsilon})$，其中 $\varepsilon > 0$为某个正常数，则 $T(n) = \Theta(n^{\log_b a})$。
2. 若 $f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a} \log n)$。
3. 若对某个常数 $\varepsilon > 0$，有 $f(n) = \Omega(n^{\log_b a + \varepsilon})$，且对某个常数 c与所有充分大的 n成立 $af(n/b) \leqslant cf(n)$，则 $T(n) = \Theta(f(n))$。

☞ **Remark**

主定理的第 2 条可推广为：若 $f(n) = \Theta(n^{\log_b a} \log^k n)$，其中 $k \geqslant 0$，则 $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$。

以下，我们来对主定理进行证明，证明的主要思路如下：首先对 n为 b的幂的情形证明主定理，其次再推广至所有非负整数 n的情形。

主定理证明：n为 b的幂的情形

在此情形下，我们只考虑 $T(1), T(b), T(b^2), \cdots, T(b^k) \cdots$这样一个离散序列的渐进性质，首先依 eq. (2.5) 进行迭代，不难有：

$$\begin{aligned} T(n) &= aT(n/b) + f(n) = a^2 T(n/b^2) + af(n/b) + f(n) \\ &= \cdots \\ &= a^k T(n/b^k) + \sum_{i=0}^{k-1} a^i f(n/b^i) \end{aligned}$$

记 $n = b^k$，则 $k = \log_b n$，并注意 $a^{\log_b n} = n^{\log_b a}$，于是有 eq. (2.6)：

$$T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) \tag{2.6}$$

以下，我们需要对 $g(n) = \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)$进行分析，给出如Lemma 2.1所示的引理：

Lemma 2.1

对定义在 b的幂上的函数 $g(n)$：

$$g(n) = \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)$$

有以下结论：

1. 若对某个常数 $\varepsilon > 0$有 $f(n) = O(n^{\log_b a - \varepsilon})$，则 $g(n) = O(n^{\log_b a})$。
2. 若 $f(n) = \Theta(n^{\log_b a})$，则 $g(n) = \Theta(n^{\log_b a} \log n)$。
3. 若对某个常数 c与所有充分大的 n成立 $af(n/b) \leqslant cf(n)$，则 $g(n) = \Theta(f(n))$。

证明. 对 **1**，由于 $f(n/b^i) = O((n/b^i)^{\log_b a-\varepsilon})$，于是有：

$$g(n) = O\left(\sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\varepsilon}\right)$$

对 i等比求和后有：

$$g(n) = O\left(n^{\log_b a-\varepsilon}\left(\frac{n^\varepsilon-1}{b^\varepsilon-1}\right)\right) = O(n^{\log_b a-\varepsilon}\cdot n^\varepsilon) = O(n^{\log_b a})$$

因此可知 $g(n) = O(n^{\log_b a})$。

对 **2**，由于：

$$\Theta\left(a_i\left(\frac{n}{b^i}\right)^{\log_b a}\right) = \Theta(n^{\log_b a})$$

因此可知 $g(n) = \Theta(n^{\log_b a}(\log_b n-1)) = \Theta(n^{\log_b a}\log n)$。

对 **3**，由于 $g(n) \geqslant f(n)$恒成立，因此 $g(n) = \Omega(f(n))$。以下，我们假设对某个常数 c与所有充分大的 n成立 $af(n/b) \leqslant cf(n)$，则：

$$a^2f(n/b^2) \leqslant caf(n/b) \leqslant c^2f(n)$$

一般地就有：

$$a^if(n/b^i) \leqslant c^if(n)$$

于是就有：

$$g(n) = \sum_{i=0}^{\log_b n-1} a^if(n/b^i) \leqslant \left(\sum_{i=0} c^i\right)f(n) + O(1)$$

其中 $O(1)$为充分大的 n所未覆盖的有限项，因此可知 $g(n) = O(f(n))$，利用Theorem 2.4，可知 $g(n) = \Theta(f(n))$，证毕。 ■

使用Lemma 2.1立即可以证得原结论，此处不再赘述。

主定理证明：一般情形

我们将要把 n为 b的幂的情形推广至 $n \in \mathbb{N}$的情形上去，于是我们需考查：

$$T(n) = aT\left(\left\lfloor\frac{n}{b}\right\rfloor\right) + f(n) \tag{2.7}$$

与

$$T(n) = aT\left(\left\lceil\frac{n}{b}\right\rceil\right) + f(n) \tag{2.8}$$

eq. (2.7) 与 eq. (2.8) 将递推方程推广定义至 $\mathbb{N}$上，于是我们仅需证明对 eq. (2.7) 与 eq. (2.8)，其满足的渐进性质不变即可，我们以下以 eq. (2.8) 为例进行说明。

对 eq. (2.8) 进行迭代，有：

$$T(n) = a^kT(n_k) + \sum_{i=0}^{k-1} a^if(n/b^i)$$

这里的 n_k 定义为：

$$n_k = \begin{cases} n & k = 0 \\ \lceil n_{k-1}/b \rceil & k \geqslant 1 \end{cases}$$

利用该定义立即可得：

$$n_k \leqslant \frac{n}{b^k} + \frac{1}{b^{k-1}} + \cdots + \frac{1}{b} + 1 < \frac{n}{b^k} + \frac{b}{b-1}$$

而我们记 $k = \lfloor \log_b n \rfloor$，可知：

$$n_{\lfloor \log_b n \rfloor} < \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} = O(1)$$

于是将 $k = \lfloor \log_b n \rfloor$ 代入 $T(n)$，有：

$$T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f(n/b^i)$$

之后对 $g(n) = \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f(n/b^i)$ 的分析便与之前完全类似，此处便不再赘述。

对主定理的理解

主定理的实质是将 $f(n)$ 与 $n^{\log_b a}$ 的大小进行比较，若 $f(n)$ 在**多项式意义**上小于 $n^{\log_b a}$，则 $T(n) = \Theta(n^{\log_b a})$；若 $f(n)$ 在**多项式意义**上大于 $n^{\log_b a}$，且满足条件 $af(n/b) \leqslant cf(n)$，则 $T(n) = \Theta(f(n))$；若 $f(n)$ 与 $n^{\log_b a}$ 大小相当，则 $T(n) = \Theta(n^{\log_b a} \log n)$。

☞ **Remark**

多项式意义上的大于与小于是一个较为严格的条件，它要求 $f(n)$ 与 $n^{\log_b a}$ 至少要在 n 充分大时有一个多项式 n^{ε} 的差距。