主定理 (Master Theorem)

范潇 吴靖阳 陈思同

摘 要

分治法是计算机科学中的重要算法之一, 但是其时间复杂度分析较为困难. 本文给出了主定理的内容及证明, 并举例说明其在分析以分治法为代表的一类递归算法时间复杂度时的应用.

1 背景介绍

1.1 主定理

在算法分析中, 主定理 (master theorem) 提供了用渐近符号表示许多由分治法得到的递推关系式的方法. 这种方法最初由 Jon Bentlery,Dorothea Haken 和 James B. Saxe 在 1980 年提出. 此方法经由经典算法教科书 Cormen,Leiserson,Rivest 和 Stein 的《算法导论》(introduction to algorithm) 推广. 它主要提出了求解各种递归式的方法——通过将加性项分解为某个函数乘以齐次系统的解来重写递归式. 用递归树的一般术语和分治递归的特定框架来描述这种方法. 这种方法可以通过记忆一个简单的模板 (或递归树公式) 和一个包含三个条目的表来应用.

1.2 渐近记号

渐近上界 O、渐近下界 Ω 和渐近紧确界 Θ 是最常用的三个渐近记号

定义 1. 设 f(n) 和 g(n) 是定义域为自然数集合的函数. 若存在正数 c 和正数 n_0, 使得当 $n > n_0$ 时, 有

$$0 \leq f(n) \leq cg(n)$$

则记作

$$f(n) = O(g(n))$$

定义 2. 设 f(n) 和 g(n) 是定义域为自然数集合的函数. 若存在正数 c 和正数 n_0, 使得当 $n > n_0$ 时, 有

$$0 \leq cg(n) \leq f(n)$$

则记作

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

定义 3. 设 f(n) 和 g(n) 是定义域为自然数集合的函数. 若存在正数 c_1, c_2 和正数 n_0, 使得当 $n > n_0$ 时，有

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

则记作

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

它们都满足

传递性：

$$若f(n) = O(g(n))和g(n) = O(h(n)), 则f(n) = O(h(n)).$$
$$若f(n) = \Omega(g(n))和g(n) = \Omega(h(n)), 则f(n) = \Omega(h(n)).$$
$$若f(n) = \Theta(g(n))和g(n) = \Theta(h(n)), 则f(n) = \Theta(h(n)).$$

自反性:

$$f(n) = O(f(n)),$$

$$f(n) = \Omega(f(n)),$$

$$f(n) = \Theta(f(n)).$$

1.3 分治法

在计算机科学中, 分治法是一种很重要的算法. 分治算法, 字面上的解释是“分而治之”. 分治算法主要可分为三点:

1. 将一个复杂的问题分成两个或更多的相同或相似的子问题, 再把子问题分成更小的子问题——“分”;

2. 将最后子问题可以简单的直接求解——“治”;

3. 将所有子问题的解合并起来就是原问题的解——“合”;

这三点是分治算法的主要特点，只要是符合这三个特点的问题都可以使用分治算法进行解决.

分治法所能解决的问题一般具有以下几个特征：

1 该问题的规模缩小到一定的程度就可以容易地解决;

2 该问题可以分解为若干个规模较小的相同问题, 即该问题具有最优子结构性质;

3 利用该问题分解出的子问题的解可以合并为该问题的解；

4 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题.

2 定理内容

通常, 规模为 n 的问题通过分治, 得到 a 个规模为 $\frac{n}{b}$ 的问题, 每次递归带来的额外计算规模为 f(n) 从而可以得到递归式

$$T(n) = aT(\frac{n}{b}) + f(n). \qquad (*)$$

主定理告诉我们：

1. 如果 $f(n) = O(n^{\log_b a - \epsilon})$, 其中 ϵ 为大于零的常数, 那么

$$T(n) = \Theta(n^{\log_b a}).$$

2. 如果 $f(n) = \Theta(n^{\log_b a} \lg^k n)$, 其中 $k \geq 0$, 那么

$$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n).$$

3. 如果 $f(n) = \Omega(n^{\log_b a + \epsilon})$, 其中 ϵ 为大于零的常数, 并且当 n 足够大时存在某个常数 $c < 1$ 使得 $af(n/b) \leq cf(n)$ 成立, 则

$$T(n) = \Theta(f(n)).$$

3 定理证明

在本证明中, 将递归式 $(*)$ 化简为 T(n) 仅定义在 $b > 1$ 的整数幂上, 并且设 $T(1) = \Theta(1)$. 同时, 将情况 2 化简为 $k = 0$ 的情形, 即，如果 $f(n) = \Theta(n^{\log_b a})$, 那么 $T(n) = \Theta(n^{\log_b a} \lg n)$. 每次递归把问题分为 a 个规模为 n/b 的子问题. 从根节点开始, 共有 $\log_b n$ 层, 叶子节点数为 $a^{\log_b n} = n^{\log_b a}$. 同时, 第 j 层共有 a^j 个子问题. 每个子问题的规模为 n/b^j, 每个子问题产生的运算量为 $f(n/b^j)$, 该层需要完成的总计算量为 $a^j f(n/b^j)$. 对于各层的计算量进行求和后可得分解与合并子问题的计算量为

$$\sum_{j=0}^{\log_b n - 1} a^j f(\frac{n}{b^j}),$$

而叶子节点的计算量为

$$\Theta(n^{\log_b a}),$$

总计算量等于两者之和, 即

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(\frac{n}{b^j}).$$

如果将任意一个非叶子节点看作根节点, 这都将是一个完整的分治问题的递归树, 所以在固定问题中每一层的计算代价不会无规律的变化, 复杂度的分布只会有三种情况：

1. 从树根到叶子每一层来看复杂度越来越高；

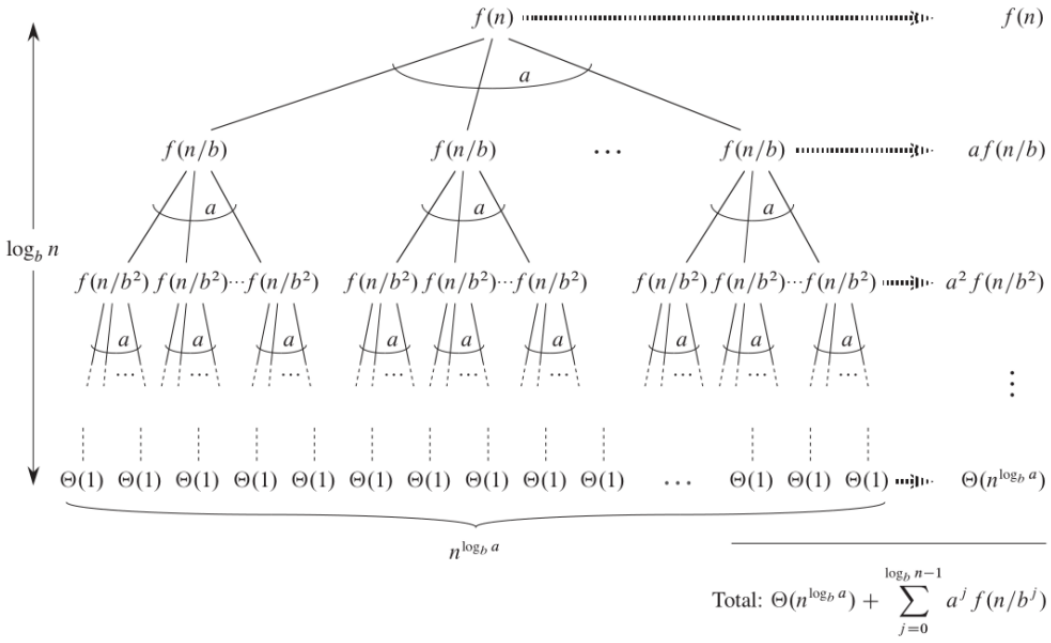

图 1

2. 从根节点到叶子节点每层来看复杂度排布均匀；

3. 从树根到叶子每一层来看复杂度越来越低.

届时我们需要考虑总体递归式的算法时间复杂度将由这两项决定，因为叶子节点已经完全确定了复杂度的渐进紧确界. 以上三种情况分别对应：

1. 整体代价叶子节点决定;

2. 整体代价均匀分布;

3. 整体代价根节点决定.

令 $g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$ 并将其与由叶子节点确定的紧确界进行比较.

1. 如果 $f(n) = O(n^{\log_b a-\epsilon})$, 其中 ϵ 为大于零的常数, 那么：

$$\begin{aligned}
g(n) &= O(\sum_{j=0}^{\log_b n-1} a^j(\frac{n}{b^j})^{\log_b a-\epsilon}) \\
&= O(n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1} a^j(\frac{1}{b^{\log_b a-\epsilon}})^j) \\
&= O(n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1} (\frac{a}{a*b^{-\epsilon}})^j) \\
&= O(n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1} b^{\epsilon j}) \\
&= O(n^{\log_b a-\epsilon})(\frac{b^{\epsilon\log_b n}-1}{b^\epsilon-1}) \\
&= O(n^{\log_b a-\epsilon}\frac{n^\epsilon-1}{b^\epsilon-1}) \\
&= O(n^{\log_b a-\epsilon})n^\epsilon \\
&= O(n^{\log_b a}).
\end{aligned}$$

2. 如果 $f(n) = \Theta(n^{\log_b a})$, 那么类似的有：

$$\begin{aligned}
g(n) &= \Theta(\sum_{j=0}^{\log_b n-1} a^j(\frac{n}{b^j})^{\log_b a}) \\
&= \Theta(n^{\log_b a}\sum_{j=0}^{\log_b n-1} a^j(\frac{1}{b^{\log_b a}})^j) \\
&= \Theta(n^{\log_b a}\sum_{j=0}^{\log_b n-1} 1) \\
&= \Theta(n^{\log_b a}\log_b n) \\
&= \Theta(n^{\log_b a}\lg n).
\end{aligned}$$

3. 如果 $f(n) = \Omega(n^{\log_b a+\epsilon})$, 其中 ϵ 为大于零的常数, 并且当 n 足够大时存在某个常数 $c < 1$ 使得 $af(n/b) \le cf(n)$ 成立, 那么：

$$\begin{aligned}
af(\frac{n}{b}) &\le cf(n), \\
f(\frac{n}{b}) &\le \frac{c}{a}f(n), \\
&\cdots \\
f(\frac{n}{b^j}) &\le (\frac{c}{a})^j f(n), \\
a^j f(\frac{n}{b^j}) &\le c^j f(n),
\end{aligned}$$

因而：

$$\begin{aligned} g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(\frac{n}{b^j}) \\ &\leq \sum_{j=0}^{\log_b n-1} c^j f(n) \\ &\leq f(n)\sum_{j=0}^{\infty} c^j + O(1) \\ &\leq f(n)\frac{1}{1-c} + O(1) \\ &= O(f(n)). \end{aligned}$$

又因为

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

因此有：

1.

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}). \end{aligned}$$

2.

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a}\lg n) \\ &= \Theta(n^{\log_b a}\lg n). \end{aligned}$$

3.

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)). \end{aligned}$$

得证.

4 定理应用

4.1 大整数乘法的分析

大整数乘法是分治法的具体应用. 设 X 和 Y 是 n 位的二进制整数, 为求它们的乘积 XY, 将 X 和 Y 分为 2 段, 每段长为 $n/2$ 位 (假设 n 是 2 的幂). 由此, 可将 X、Y 分别表示为 $A2^{n/2}+b, C2^{n/2}+D$,X

和 Y 的乘积为

$$\begin{aligned} XY &= (A2^{n/2} + B)(C2^{n/2} + D) \\ &= AC2^n + (AD + BC)2^{n/2} + BD \\ &= AC2^n + ((A - B)(D - C) + AC + BD)2^{n/2} + BD, \end{aligned}$$

这样只需作 3 次 $n/2$ 位乘法，因此，

$$T(n) = \begin{cases} O(1), & n = 1; \\ 3T(n/2) + O(n), & n > 1. \end{cases}$$

使用主定理求解,$a = 3, b = 2$, 则

$$n^{\log_b a} = n^{\log_2 3} = \Theta(n^{1.59}), f(n) = O(n^{\log_2 3 - \varepsilon}),$$

其中 $\varepsilon = 1$. 第一种情况成立, 故递归式的解为

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{1.59}).$$

4.2 二路并归算法的分析

并归算法的思路是将两个有序表合并成一个有序表. 对于拥有 n 个元素的表, 其合并表的时间花费为 n.[3] 因此, 二次并归算法的时间复杂度可使用递归式 $T(n) = 2T(n/2) + n$ 表示. 使用主定理求解,$a = 2, b = 2, f(n) = n$, 则

$$n^{\log_b a} = n^{\log_2 2} = \Theta(n),$$

第二种情况成立, 故递归式的解为

$$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n \lg n).$$

4.3 满足第三种情况的案例

已知递归式 $T(n) = 3T(n/4) + a \log n$. 使用主定理求解,$a = 3, b = 4, f(n) = n \log n$, 则

$$n^{\log_4 3} = O(n^{0.793}), f(n) = \Omega(n^{\log_4 3 + \varepsilon}), \varepsilon \approx 0.2,$$

且当 $c = 0.75, n$ 充分大时，有

$$af(n/b) = 3(n/4) \log(n/4) \leq (3/4)n \log n = cf(n),$$

即第三种情况成立, 故递归式的解为

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

参考文献

[1] Alfred V. Aho and John E. Hopcroft. The Design and Analysis of Computer Algorithms. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1974.

[2] Jon Louis Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. SIGACT News, 12(3):36–44, sep 1980.

[3] 李卿. 递归算法分析中主定理的应用. 黑龙江科技信息, pages 193+83, 2011.