Chapter 2

复杂度的渐近表示

本章的内容参考自《算法导论（第三版）》。

2.1 函数的渐近增长

同学们在数据结构课程上学习过大 O 记号，但是对大 O 记号没有一个具体的定义，在这一小节中，我们将对渐近复杂度表示的常用记号，即 O、Θ、Ω 给出具体的定义，在后续的课程中，我们将用这些记号来表示算法的复杂度。

首先我们介绍 Θ 记号的定义，对于一个给定的函数 $g(n)$，用 $\Theta(g(n))$ 来表示以下函数的集合：$\Theta(g(n)) = \{f(n)|\exists c_1, c_2, n_0 > 0, s.t., \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$。它的意思是，若 $f(n) \in \Theta(g(n))$（有时也简写为 $f(n) = \Theta(g(n))$），那么当 n 足够大的时候，$f(n)$ 的函数值夹在 $c_1 g(n)$ 和 $c_2 g(n)$ 之间，如图2.1(a)所示。

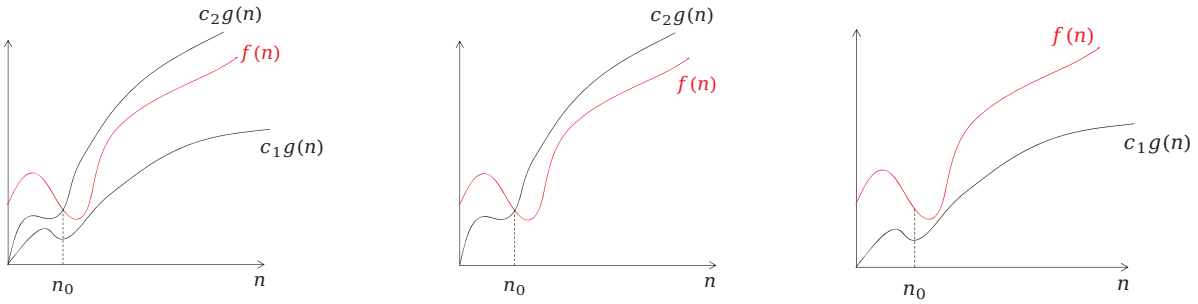
接下来是 O 记号的定义，对于一个给定的函数 $g(n)$，用 $O(g(n))$ 来表示以下函数的集合：$O(g(n)) = \{f(n)|\exists c, n_0 > 0, s.t., \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$。它的意思是，若 $f(n) \in O(g(n))$（有时也简写为 $f(n) = O(g(n))$），那么当 n 足够大的时候，$f(n)$ 的函数值在 $cg(n)$ 的下方，如图2.1(b)所示。

最后是 Ω 记号的定义，对于一个给定的函数 $g(n)$，用 $\Omega(g(n))$ 来表示以下函数的集合：$\Omega(g(n)) = \{f(n)|\exists c, n_0 > 0, s.t., \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$。它的意思是，若 $f(n) \in \Omega(g(n))$（有时也简写为 $f(n) = \Omega(g(n))$），那么当 n 足够大的时候，$f(n)$ 的函数值夹在 $c_1 g(n)$ 和 $c_2 g(n)$ 之间，如图2.1(c)所示。

通过观察定义和图我们可以得出以下结论：

- 若函数 $f(n) \in O(g(n))$，说明函数 $g(n)$ 是 $f(n)$ 的渐近上界。
- 若函数 $f(n) \in \Omega(g(n))$，说明函数 $g(n)$ 是 $f(n)$ 的渐近下界。
- 若函数 $f(n) \in \Theta(g(n))$，说明函数 $g(n)$ 是 $f(n)$ 的渐近确界。

换句话说，如果我们说“某个算法的时间复杂度是 $O(n)$”，那么我们可以认为该算法的运行时间复杂度是以 n 为上界的，即该算法的时间复杂度最差是 n，它在某些情况



(a) $f(n) \in \Theta(g(n))$ (b) $f(n) \in O(g(n))$ (c) $f(n) \in \Omega(g(n))$

Figure 2.1: 函数的渐近增长

下可能要好于 n，一个直观的例子是我们在第一章中给出的线性查找算法的时间复杂度可以表示为 $O(n)$，因为函数最差的情况下会执行 n 次循环。而反过来，如果我们说“某个算法的时间复杂度是 $\Omega(n)$，那么我们可以认为该算法的运行时间复杂度是以 n 为下界的，即该算法的时间复杂度在最好情况下是 n，它在某些情况下可能比 n 还差，最后，如果我们说“某个算法的时间复杂度是 $\Theta(n)$，那么我们可以认为该算法的运行时间复杂度是以 n 为确界，也就是所不存在要优于 n 的情况，也不存在要比 n 差的情况。

通过上面的总结我们可以知道，如果 $f(n) \in O(g(n))$，那么 $g(n) \in \Omega(f(n))$，同样的，如果 $f(n) \in \Theta(g(n))$，那么 $g(n) \in \Theta(f(n))$，同时还有 $f(n) \in O(g(n))$ 和 $f(n) \in \Omega(g(n))$。

我们举一个例子，比如:

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

要证明该式子，我们可以套用定义，即存在 n_0, c_1, c_2，使得 $\forall n \geq n_0, 0 \leq c_1n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2n^2$，同时除以 n^2，得 $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$，我们取 $n_0 = 1000, c_1 = \frac{1}{100}, c_2 = 100$，不等式成立，所以 $\frac{1}{2}n^2 - 3n = \Theta(n^2)$。同时我们也有 $\frac{1}{2}n^2 - 3n = O(n^2) = \Omega(n^2)$。

在第1章中我们讨论过，通常情况下我们关心函数运行的最差情况，因为我们希望给算法的使用者一定程度上的保证。因此，对于复杂度的三种渐近表示，我们一般情况下力求给出复杂度的确界，即 $\Theta(g(n))$，但当某些情况下确界不好给出时，我们更加希望能够给出复杂度的上界，即 $O(g(n))$。我们通常不太关心复杂度的下界是多少，即 $\Omega(g(n))$，但某些理论研究需要我们给出一类问题的复杂度下界，例如：基于比较的排序算法至少需要 $\Omega(n \lg n)$ 的时间复杂度。

另外，我们使用函数的渐近增长来表示复杂度，是基于算法在输入数据很大的前提下进行的，也就是说我们不关心输入数据比较小的情况，因为此时无论选用何种算法，运行时间都不至于太长。

最后，一般而言，我们可以把函数渐近增长由小到大排为:

1. 常数阶: $\Theta(1)$
2. 对数阶: $\Theta(\lg n)$
3. 多项式阶: $\Theta(n)$、$\Theta(n^2)$、$\Theta(n^3)$ 等

4. 指数阶: $\Theta(2^n)$、$\Theta(3^n)$ 等

5. 阶乘阶: $\Theta(n!)$，由斯特林公式 $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1+\Theta(\frac{1}{n}))$ 得到

我们在计算复杂度的时候，如果复杂度的结果是多项相加，那么我们可以简单的保留最高阶，并忽略之前的常数即可。

2.2 几种排序算法

在上一小节中我们学习了算法复杂度的渐近表示。这一小节里我们将以常见的几种排序算法为例，来实际的应用渐近增长来描述算法的计算复杂度。在这里我们定义排序算法解决的问题是输入一个有 n 个元素整型的数组，排序算法将数组由小到大排好顺序。

2.2.1 插入排序

我们首先先介绍一种最简单的排序算法，即插入排序，如 INSERTION-SORT 所示。该算法接收一个有 n 个整数的数组 A，输出一个由小到大排好顺序的数组 A，即 $A[1] \le A[2] \le \cdots \le A[n]$。

INSERTION-SORT(A)

```
1: for j = 2 to A.length do
2:     key = A[j]
3:     i = j − 1
4:     while i > 0 and A[i] > key do
5:         A[i + 1] = A[i]
6:         i = i − 1
7:     A[i + 1] = key
```

跟以前一样，我们首先要证明算法的正确性，首先我们先从循环不变式的角度来理解插入排序算法。算法在设计的过程中维持的循环不变式是: 第 1 行的循环执行完毕后，前 j 个元素是排好顺序的，当循环退出条件达成的时候，整个数组也就排好了顺序了。但是我们还是要严格的证明算法的正确性，证明算法的正确性在这一章中不是重点，只是我们现在处于课程刚开始，同学们对如何证明算法的正确性不太熟练，所以我们再多给大家演示一遍。在这里我们依然使用数学归纳法来证明。第一步我们确定谓词 $P(k)$，

- $P(k)$: 算法在执行第 1 行的大循环后，令 $j = k$，此时算法可以排序好前 k 个元素构成的子数组。

虽然我们第 1 行的循环是从 2 到 $A.length$，但是我们依然可以补齐 0 和 1 的情况，在这里我们不教条地使用 0 作为基本情况，我们可以稍微变通一下，以 $P(1)$ 作为基本情况，当 $j = 1$ 时，也就是算法没有执行过第 1 行所示的循环时，算法的第 1 个元素构成的子数组当然是排好序的，因为只有一个元素。

最后我们证明一般情况，$\forall k \in \mathbb{N}(P(k) \Rightarrow P(k+1))$，假设算法运行完了 $j = k$ 的循环时，前 k 个元素构成的子数组已经排好了顺序，当运行下一次循环时，我们可以考察从第 4 行到第 6 行的循环，在循环过程中逐个不断地将比 key 大的数往后移一个单位，当循环停止时，要么 $i = 0$，此时说明没有元素是小于等于 k 的，且所有元素都往后移动了一个元素给待插入元素留了个位置，我们把它放进这个位置即可。要么是停留在了某个位置 i 上，由归纳假设，此时 $A[1..i-1]$ 的所有元素都是小于等于待插入元素的，并且大于它的元素都往后移动了一个位置，给待插入的元素留出了一个空位，这时候，我们将待插入元素放在 i 上即可。这样我们就证明完了插入排序算法的正确性。

接下来是本章的重点，衡量算法运行的时间复杂度，首先算法第 1 行的循环会执行 $n-1$ 次，每次循环中，第 4 行到第 6 行的子循环最多要循环 $j-1$ 次。那么我们的总的时间复杂度是:

$$\sum_{j=1}^{n-1} O(j) = O(\sum_{j=1}^{n-1} j) = O(\frac{(n-1)(n)}{2}) = O(\frac{n^2-n}{2}) = O(n^2)$$

接下来我们考察以下算法的空间复杂度。算法只声明了两个循环变量 i 和 j，因此占用 8 个字节，是常数个空间，因此算法的空间复杂度是 $\Theta(1)$。

2.2.2 归并排序算法

我们接下来将通过归并排序算法介绍一些新的知识。首先我们先介绍什么是归并排序算法，总得来说，归并排序算法一共有以下几个步骤:

- 首先，算法会将输入数组从中间分成两个子数组。
- 然后归并排序算法会在两个子数组上递归的运行归并排序算法，从而将两个子数组排好顺序。
- 最后归并排序会将两个子数组合并成一个排好序的完整的数组。

首先我们先介绍归并排序中的“合并”操作，如 MERGE 所示。算法接受两个排好序的数组 A 和 B 为输入参数，也就是说，$A[1] \le A[2] \le \cdots \le A[n]$，同时 $B[1] \le B[2] \le \cdots \le B[m]$，算法返回一个排好序的数组 C，数组中有 $n+m$ 个元素，并且 $C[1] \le C[2] \le \cdots \le C[n+m]$，数组中的元素就是 A 数组和 B 数组中的全部元素。此处我们将数组视作是一个对象，类似 Java 中的数组或者 C++ 中的 vector。如果使用 C 语言，需要更加仔细的操作内存。

（此处可以举个例子）

MERGE(A, B)

```
1: i = 1, j = 1, k = 1
2: Let C[1..A.length+B.length] be a new array.
3: while i ≤ A.length and j ≤ B.length do
4:     if A[i] ≤ B[j] then
5:         C[k] = A[i]
6:         i = i + 1
7:     else
8:         C[k] = B[j]
9:         j = j + 1
10:    k = k + 1
11: while i ≤ A.length do
12:    C[k] = A[i]
13:    k = k + 1
14:    i = i + 1
15: while j ≤ B.length do
16:    C[k] = B[j]
17:    k = k + 1
18:    j = j + 1
19: return C
```

我们接下来要证明 MERGE 算法的正确性，并且分析算法的时间复杂度和空间复杂度。首先 MERGE 算法的执行过程是将 A 和 B 数组中的元素逐个放进 C 数组中的，所以 MERGE 算法是将 A 和 B 合并在一起的。接下来我们将使用数学归纳法来 MERGE 算法得到的数组是排好序的，首先第一步，确定谓词 $P(n)$:

- $P(n)$: MERGE 操作填入 C 数组中 n 个元素的时候，这 n 个元素是从小到大排好顺序的，未被放入数组中的元素都是大于等于数组 C 中的元素。

第二步我们证明基本情况 $P(0)$，当 C 数组中有 0 个元素时，自然是排好了顺序的。

第三步我们证明一般情况 $\forall n \in \mathbb{N}(P(n) \Rightarrow P(n+1))$，归纳假设 $P(n)$ 成立，即 C 数组已经有 n 个排好顺序的元素，数组外的元素都是大于等于数组 C 中的元素，此时，我们要分情况讨论:

第一种情况是数组 A 和 B 里的元素都没有被全部加入 C 元素中，这种情况下，算法的第 4 行到第 10 行会将 A 数组和 B 数组中的最小的元素放入 C 数组中，此时 $n+1$ 个元素是排好顺序的，且 C 数组外的元素都大于等于 C 数组中的元素。

第二种情况是数组 A 和 B 中只有一个数组有元素了，那么第 11 行和第 15 行的两个循环只有一个循环会被运行，鉴于 A 数组和 B 数组是已经排好顺序的，放入的元素

将是剩余元素中的最小的那个，且 C 数组外的元素是大于等于数组 C 中的元素的。

算法的时间复杂度是 $\Theta(n+m)$，因为每个元素都被复制且仅被复制一次，空间复杂度也是 $\Theta(n+m)$，因为我们开了数组 C 的大小是 $n+m$。

有了 MERGE 操作以后，我们就可以使用它来完善归并排序算法了。算法接受一个数组作为参数，返回一个排好序的数组，即 $A[1] \le A[2] \le \cdots \le A[n]$。

```
MERGE-SORT(A)
 1: if A.length ≤ 1 then
 2:     return A
 3: mid = A.length/2
 4: Let B[1..mid] is a new array.
 5: Let C[1..A.length - mid] is a new array.
 6: for i = 1 to mid do
 7:     B[i] = A[i]
 8: for i = mid + 1 to A.length do
 9:     C[i-mid] = A[i]
10: B = MERGE-SORT(B)
11: C = MERGE-SORT(C)
12: return MERGE(B, C)
```

接下来我们依旧是证明算法的正确性和分析算法的时间复杂度、空间复杂度。首先，我们要知道的是，归并排序算法 MERGE-SORT 使用了递归，要更简单的证明递归算法的正确性，我们可以应用归纳公理的一个变种，即强归纳公理：

- 强归纳公理:对于谓词 $P(n)$,若 $P(0)$ 为真,且 $\forall n \in \mathbb{N}(P(0) \wedge P(1) \wedge \cdots \wedge P(n) \Rightarrow P(n+1))$ 为真，那么 $\forall n \in \mathbb{N}$, $P(n)$ 都为真。

接下来我们应用强归纳公理来证明归并排序算法的正确性。首先第一步，确定谓词 $P(n)$:

- $P(n)$: 归并排序算法 MERGE-SORT 能够成功将一个长度为 n 的数组排序好。

第二步是证明基本情况 $P(0)$（和 $P(1)$），数组为 0（或 1）时，算法直接将原数组返回，原数组就是排好序的数组。

第三步是证明一般情况 $\forall n \in \mathbb{N}(P(0) \wedge P(1) \wedge \cdots \wedge P(n) \Rightarrow P(n+1))$，假设算法能够成功排序数组长度为 0、1、2、$\cdots$、n 的数组，对于任意一个长度为 $n+1$ 的数组，算法首先将它分成两个数组，其中每个数组长度都位于 0 到 n 之间。由归纳假设，算法能够成功将这两个数组排好顺序，在排好顺序之后，MERGE 操作将两个数组合并成一个排好序的数组。

接下来是计算算法的时间复杂度。我们假设算法接受的数组元素个数为 n，算法运行的时间复杂度是 $T(n)$，算法的运行过程中首先将数组切分成两个元素个数为 $n/2$ 的子数组，然后分别在上面运行 MERGE-SORT，最后再合并在一起，因此算法的时间复杂度满足以下关系:

$$T(n) = 2T(n/2) + \Theta(n)$$

我们首先使用代入法来解上面的递归式，首先我们先猜测一个可能的解，由于插入排序的时间复杂度是 $\Theta(n^2)$，所以我们在这里先猜测上面的递归式的解为 $O(n^2)$，也就是说我们先估一个上界，然后再看看这个上界是不是满足 O 记号的定义。我们代入 $T(n/2) \le c_1(n/2)^2$ 和 $\Theta(n) \le c_2 n$，有:

$$T(n) = 2T(n/2) + \Theta(n) \le 2 \times c_1 n^2/4 + c_2 n = c_1 n^2/2 + c_2 n \le c_1 n^2/2 + c_2 n^2 \le c_1 n^2$$

当且仅当 $c_1 \ge 2c_2$，则有 $T(n) \le c_1 n^2$，即 $T(n) = O(n^2)$。

我们马上就可以看见我们估计的这个上界不够好，这也显示出了使用代入法来求解上面的递归式的问题在于，你很难凭空估计一个准确的上下界，因此我们要考虑使用别的办法来解决这个问题。

我们接下来使用递归树法来求解 $T(n)$，如图2.2所示。

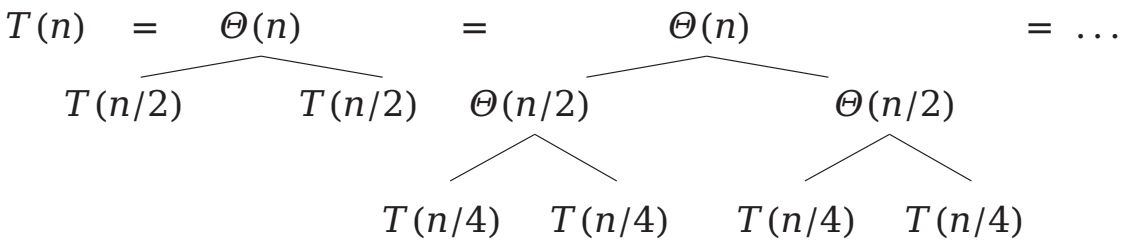


Figure 2.2: 递归树的展开过程

如果我们将该递归树完全展开，则会得到如图2.3所示的样子，接下来我们对整棵递归树求和，递归树的每一层的和都是 $\Theta(n)$，递归树的高度是 $\log_2 n$，因此递归树的总和为 $\Theta(n \lg n)$，即递归算法的时间复杂度是 $\Theta(n \lg n)$。

接下来我们考察算法的空间复杂度，假设算法的空间复杂度是 $S(n)$，算法在执行过程中会开两个数组，数组的总大小是 n，然后递归的执行两个归并排序的子过程，每个子过程的输入参数的大小是 $n/2$，最后 MERGE 函数还会开个长度为 n 的数组，即:

$$S(n) = 2S(n/2) + \Theta(n) = \Theta(n \lg n)$$

所以算法的空间复杂度也是 $\Theta(n \lg n)$。

由我们之前的讨论，$\Theta(n \lg n)$ 的时间复杂度是要比 $\Theta(n^2)$ 的时间复杂度要快的，也就是说，归并排序算法是比插入排序算法运行的速度要快。并且，即使我们在一个运算速度比较快的处理器上运行插入排序算法，在一个运算速度比较慢的处理器上运行归并排序算法，当我们的数组中的元素超过一定个数的时候，在比较慢的那个处理器上运行的归并排序算法也会更早的排完顺序。

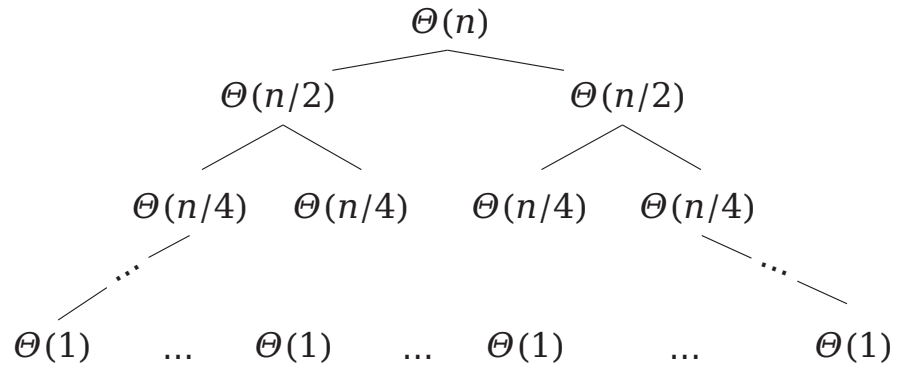


Figure 2.3: 递归树的完全展开

2.3 主定理

本小节我们介绍主定理，可以用于求解递归算法的时间复杂度。

主定理 (Master Theorem)：令 $a \geq 1$ 和 $b > 1$ 是常数，$f(n)$ 是一个函数，$T(n)$ 是定义在非负整数上的递归式:

$$T(n) = aT(n/b) + f(n)$$

那么 $T(n)$ 有如下渐近界:

1. 若对某个常数 $\epsilon > 0$ 有 $f(n) = O(n^{\log_b a - \epsilon})$，则 $T(n) = \Theta(n^{\log_b a})$。
2. 若 $f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a} \lg n)$。
3. 若对某个常数 $\epsilon > 0$ 有 $f(n) = \Omega(n^{\log_b a + \epsilon})$，且对某个常数 $c < 1$ 和所有足够大的 n 有 $af(n/b) \leq cf(n)$，则 $T(n) = \Theta(f(n))$。

一种方便记忆主定理的方法就是用 $f(n)$ 和 $n^{\log_b a}$ 作比较。如果说 $f(n)$ 是多项式意义上的小于 $n^{\log_b a}$，那么解就是 $T(n) = \Theta(n^{\log_b a})$。如果 $f(n)$ 是渐近意义上的等于 $n^{\log_b a}$，那么解就是 $T(n) = \Theta(n^{\log_b a} \lg n)$。最后，如果 $f(n)$ 是多项式意义上的大于 $n^{\log_b a}$，且满足一个正则条件 $af(n/b) \leq cf(n)$，那么解是 $T(n) = \Theta(f(n))$。

例 1：归并排序的递归式是 $T(n) = 2f(n/2) + \Theta(n)$，其中 $f(n) = \Theta(n)$，$a = 2$，$b = 2$，此时 $n^{\log_b a} = n = \Theta(n) = f(n)$，满足主定理的第 2 个情况，于是有 $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n \lg n)$。

例 2：对于递归式 $T(n) = 9T(n/3) + n$，有 $f(n) = n$，$a = 9$，$b = 3$，此时 $n^{\log_b a} = n^2$。由于 $f(n) = n = O(n^{2-\epsilon})$，其中 $\epsilon = 0.5$（或 0.6、1 都可以），于是满足主定理的第 1 个情况，于是有 $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$。

例 3：对于递归式 $T(n) = 3T(n/4) + n \lg n$，有 $f(n) = n \lg n$，$a = 3$，$b = 4$，此时 $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$，由于 $f(n) = n \lg n = \Omega(n^{\log_4 3 + \epsilon})$，$\epsilon \approx 0.2$，因此我们要判断是不是符合正则条件，如果符合则可以适用主定理的第 3 个情况，由于对于足够大的 n，有

$$af(n/b) = 3f(n/4) = 3\frac{n}{4}\lg\frac{n}{4} = \frac{3}{4}n\lg\frac{n}{4} \leq \frac{3}{4}n\lg n = cn\lg n$$

即 $c = \frac{3}{4}$，因此符合正则条件，可以适用主定理的第 3 个情况，于是有 $T(n) = \Theta(n \lg n)$。

例 4：对于递归式 $T(n) = 2T(n/2) + n \lg n$，有 $f(n) = n \lg n$，$a = b = 2$，$n^{\log_b a} = n$，这时我们可以发现 $n \lg n = \Omega(n)$，但我们不能适用主定理的第三个情况，原因是不存在 $\epsilon > 0$ 使得 $n \lg n = \Omega(n^{1+\epsilon})$，因为假设存在 $\epsilon > 0, c > 0$，使得 $n \lg n \geq cn^{1+\epsilon}$，我们有:

$$n \lg n \geq cn^{1+\epsilon} \Leftrightarrow \lg n \geq cn^{\epsilon} \Leftrightarrow c \leq \frac{\lg n}{n^{\epsilon}}$$

然而由洛必达法则:

$$\lim_{n\to\infty} \frac{\lg n}{n^{\epsilon}} = \lim_{n\to\infty} \frac{1/n}{\epsilon n^{\epsilon-1}} = \lim_{n\to\infty} \frac{1}{\epsilon n^{\epsilon}} = 0$$

即 $c \leq 0$，这就出现了矛盾，所以不存在 $c > 0, n > n_0$，使得 $n \lg n \geq cn^{1+\epsilon}$。因此对于这个递归式，我们不能适用主定理，应该使用递归树法来求解。

例 5：解递归式 $T(n) = 2T(\sqrt{n}) + 1$。该递归式中存在根号，因此要解这个递归式我们要想办法将根号转换为我们熟悉的形式。我们熟悉的形式是最好是将 $T(n^{1/2})$ 转换为 $W(x/2)$ 的形式，因此我们希望在把变量 “移动” 到指数上。于是我们令 $n = e^x$，即: $x = \ln n$，于是原式可以化为: $T(e^x) = 2T(e^{x/2}) + 1$。令 $W(x) = T(e^x)$，于是原式等价于 $W(x) = 2W(x/2) + 1$，此时 $x^{\log_b a} = x$，且 $f(x) = 1 = O(x^{1-\epsilon})$，可以取 $\epsilon = 0.2$，于是符合主定理的情况 1，于是有 $W(x) = \Theta(x) = \Theta(\ln n) = T(e^x) = T(n)$，因此 $T(n) = \Theta(\lg n)$。

2.4 主定理的直观理解

我们接下来将使用递归树的方法来直观的理解主定理，这里我们不去严格的证明主定理。如图2.4所示，是递归式 $T(n) = af(n/b) + f(n)$ 的递归树展开。

首先我们认为 $f(1) = \Theta(1)$，因为我们假设对规模为 1 的输入处理时间是常数。那么树的高度就是 $\log_b n$，所以树的最后一层有 $a^h = a^{\log_b n} = n^{\log_b a}$ 个叶子节点。

现在我们考察主定理的情况 3，即 $af(n/b) \leq cf(n)$ 且 $f(n) = \Omega(n^{\log_b a+\epsilon})$，也就是说每一层的总和都是呈几何级数的减少的，这样下来对这个级数求和，$f(n)$ 将成为主导，也就是说，最终的结果是 $\Theta(f(n))$。

（此处从 i 层展开到 $i+1$ 层如果是减少的，那么从 $i+1$ 层到 $i+2$ 层也是减少的，因为是用同样的方法在解决子问题。）

而若 $f(n) = O(n^{\log_b a-\epsilon})$，那么每一层的综合都是呈几何级数增加的，这样下来对这个级数求和，$\Theta(n^{\log_b a})$ 将成为主导，那么最终的结果是 $\Theta(n^{\log_b a})$，就是情况 1。

最后如果 $f(n) = \Theta(n^{\log_b a})$,那么每一层都是相同的,最终的求和结果就是 $\Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \lg n)$。

这样以来我们就从递归树的角度直观的理解了主定理。

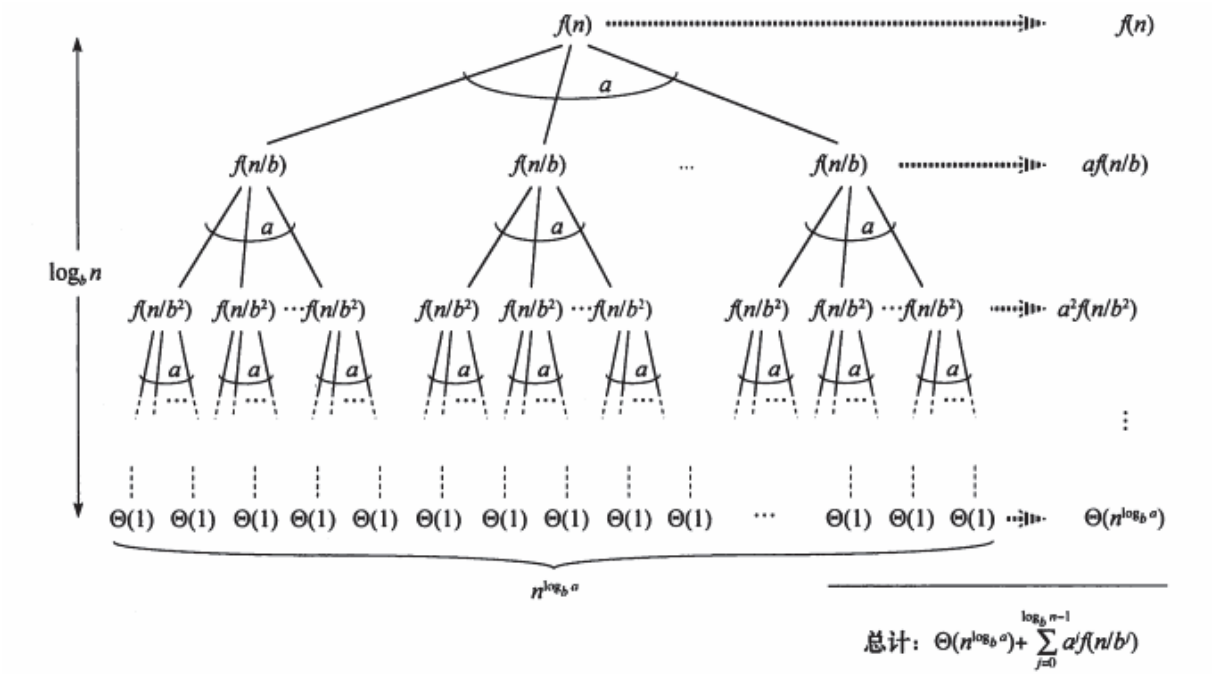


Figure 2.4: 使用递归树的展开来直观的理解主定理