

算法设计与分析

级数求和与递推关系

目录

- ① 数列与级数求和
- ② 递推关系与算法分析
 - 方法一：直接迭代
 - 方法二：化简迭代
 - 方法三：递归树
- ③ 主定理及其证明
- ④ 主定理的应用

算法复杂度的数学基础

算法通常由循环和迭代结构组成

- 复杂度 \rightarrow 级数求和

级数求和的计算方法

- 通项公式 \rightarrow 精确结果
- 估计求和的上界 \rightarrow 近似结果

算法可能包含递归结构

- 复杂度 \rightarrow 递推关系

递推关系的求解方法

- 递推关系简单：直接迭代 + 换元迭代
- 递推关系复杂：化简 + 递归树
- 一般情况：主定理

数列与级数的概念

数列：一个有序的数字列表；这个有序列表中的数字称为数列的“项”。

数列与级数的概念

数列：一个有序的数字列表；这个有序列表中的数字称为数列的“项”。

级数：数列所有项的和；所得的值称为“和”或“求和”。

数列与级数的概念

数列：一个有序的数字列表；这个有序列表中的数字称为数列的“项”。

级数：数列所有项的和；所得的值称为“和”或“求和”。

例：1, 2, 3, 4 是一个数列，其项为“1”、“2”、“3”、“4”；对应的级数是和“ $1 + 2 + 3 + 4$ ”，级数的值为 10。

接下来，我们首先回顾三个经典数列。

等差数列

等差数列

$$a, a + d, \dots, a + (n - 1)d$$

$$a_i = a + (i - 1)d, \quad \text{公差} = d \neq 0$$

等差级数

$$S(n) = \sum_{i=1}^n a_i = \frac{n(a_1 + a_n)}{2} = \frac{n(2a + (n - 1)d)}{2}$$

等比数列

等比数列

$$a, ar, \dots, ar^{n-1}, \quad a_i = ar^{i-1}, \quad \text{公比} = r \neq 1$$

等比级数

$$S(n) = \sum_{i=1}^n ar^{i-1} = a + ar + ar^2 + \dots + ar^{n-1}$$

$$rS(n) = \sum_{i=1}^n ar^i = ar + ar^2 + ar^3 + \dots + ar^n$$

$$\Rightarrow S(n) - rS(n) = a - ar^n \Rightarrow S(n) = a \left(\frac{1 - r^n}{1 - r} \right)$$

$$\lim_{n \rightarrow \infty} S(n) = \frac{a}{1 - r}, \quad |r| < 1$$

等比级数的可视化

$$S(n) = \frac{1}{2} + \frac{1}{4} + \cdots$$

$$S(n) = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots$$

等比数列的应用

等比级数是具有有限和的无穷级数的最简单例子之一（尽管并非所有等比级数都具有此性质）。

等比级数在数学中广泛使用，在物理学、工程学、生物学、经济学、计算机科学、排队论和金融学中都有重要应用。

- 循环小数（如 $0.77777\ldots$ ）是有理数
- 分形几何
- 芝诺悖论
- 经济学：年金的现值
- 比特币总数 $\leq 2.1 \times 10^8$

调和数列

调和数列（其倒数构成等差数列）

$$a_i = \frac{1}{i} : 1, \frac{1}{2}, \dots, \frac{1}{n}$$

[Pythagoras 图片]

调和级数的计算：积分判别法

$$S(n) = \Theta(\ln n)$$

下界：

$$S(n) = \sum_{i=1}^n \frac{1}{i} > \int_1^{n+1} \frac{1}{x} dx = \ln(n+1)$$

上界：

$$S(n) = \sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} < 1 + \int_1^n \frac{1}{x} dx = \ln n + 1$$

这些数列的有趣性质

中间项是其相邻两项的“平均值”

- 等差数列

$$\text{算术平均: } a_{i+1} = \frac{a_i + a_{i+2}}{2}$$

这些数列的有趣性质

中间项是其相邻两项的“平均值”

- 等差数列

$$\text{算术平均: } a_{i+1} = \frac{a_i + a_{i+2}}{2}$$

- 等比数列

$$\text{几何平均: } a_{i+1} = \sqrt{a_i \cdot a_{i+2}}$$

这些数列的有趣性质

中间项是其相邻两项的“平均值”

- 等差数列

$$\text{算术平均: } a_{i+1} = \frac{a_i + a_{i+2}}{2}$$

- 等比数列

$$\text{几何平均: } a_{i+1} = \sqrt{a_i \cdot a_{i+2}}$$

- 调和数列

$$\text{调和平均: } a_{i+1} = \frac{2}{\frac{1}{a_i} + \frac{1}{a_{i+2}}}$$

精确级数求和

$$\begin{aligned}\sum_{i=1}^n i \cdot 2^{i-1} &= \sum_{i=1}^n i(2^i - 2^{i-1}) \quad // \text{拆项} \\&= \sum_{i=1}^n i \cdot 2^i - \sum_{i=1}^n i \cdot 2^{i-1} \\&= \sum_{i=1}^n i \cdot 2^i - \sum_{i=0}^{n-1} (i+1) \cdot 2^i \quad // \text{换下标} \\&= \sum_{i=1}^n i \cdot 2^i - \sum_{i=0}^{n-1} i \cdot 2^i - \sum_{i=0}^{n-1} 2^i \quad // \text{拆项} \\&= n \cdot 2^n - (2^n - 1) = (n-1)2^n + 1 \quad // \text{等比级数}\end{aligned}$$

近似级数求和

放大法

- ① $\sum_{i=1}^n a_i \leq n \cdot a_{\max}$ (粗略)
- ② 假设 $\exists 0 < r < 1$, 使得 $\forall k \geq 0$, 不等式 $a_{i+1}/a_i \leq r$ 成立, 则可将其放大为等比级数

$$\sum_{i=0}^n a_i \leq \sum_{i=0}^n a_0 r^i = a_0 \frac{1 - r^{n+1}}{1 - r}$$

放大法示例

估计 $\sum_{i=1}^n \frac{i}{3^i}$ 的上界

解：

$$a_i = \frac{i}{3^i}, \quad a_{i+1} = \frac{i+1}{3^{i+1}} \Rightarrow \frac{a_{i+1}}{a_i} = \frac{1}{3} \cdot \frac{i+1}{i} \leq \frac{2}{3}$$

应用放大法：

$$\sum_{i=1}^n \frac{i}{3^i} < \sum_{i=1}^{\infty} \frac{1}{3} \left(\frac{2}{3}\right)^{i-1} = \frac{1}{3} \cdot \frac{1}{1 - \frac{2}{3}} = 1$$

二分查找算法

Require: $A[l, r]$, 目标元素 x

Ensure: j

```
1:  $l \leftarrow 1, r \leftarrow n$ 
2: while  $l \leq r$  do
3:    $m \leftarrow \lfloor (l + r) / 2 \rfloor$ 
4:   if  $A[m] = x$  then
5:     return  $m$  { $x$  是中位数}
6:   else if  $A[m] > x$  then
7:      $r \leftarrow m - 1$ 
8:   else
9:      $l \leftarrow m + 1$ 
10:  end if
11: end while
12: return 0
```

二分查找演示

输入规模 n

理想情况: $n = 2^k - 1$

输入规模 n

理想情况: $n = 2^k - 1$

问: 为什么称 $n = 2^k - 1$ 为理想情况?

输入规模 n

理想情况: $n = 2^k - 1$

问: 为什么称 $n = 2^k - 1$ 为理想情况?

答: 因为子问题的规模仍然是 $2^i - 1$ 的形式

输入规模 n

理想情况: $n = 2^k - 1$

问: 为什么称 $n = 2^k - 1$ 为理想情况?

答: 因为子问题的规模仍然是 $2^i - 1$ 的形式

x 有 $2n + 1$ 种可能:

- x 在数组中: n 种
- x 不在数组中: 落入 $n + 1$ 个区间

需要 t 次比较的输入 x 的数目 ($n = 7, k = 3$)

- 对于 $t \in [k - 1]$, 需要 t 次比较的可能输入元素数为 2^{t-1}
- 对于 $t = k$, 需要 k 次比较的可能输入元素数为 $2^{k-1} + (n + 1)$

二分查找的平均情况复杂度

设 $n = 2^k - 1$ ，假设 x 以相同概率出现在每个位置：

$$\begin{aligned} T(n) &= \frac{1}{2n+1} \left(\sum_{t=1}^{k-1} t \cdot 2^{t-1} + k(2^{k-1} + n + 1) \right) \\ &= \frac{1}{2n+1} \left(\sum_{t=1}^{k-1} t \cdot 2^{t-1} + k(2^{k-1} + 2^k) \right) \\ &= \frac{1}{2n+1} \left((k-2)2^{k-1} + 1 + k2^k + k2^{k-1} \right) \\ &= \frac{k2^k - 2^k + 1 + k2^k}{2n+1} = \frac{(2k-1)2^k + 1}{2^{k+1} - 1} \\ &\approx k - \frac{1}{2} = \Theta(\log n) \end{aligned}$$

目录

① 数列与级数求和

② 递推关系与算法分析

- 方法一：直接迭代
- 方法二：化简迭代
- 方法三：递归树

③ 主定理及其证明

④ 主定理的应用

动机

递归和迭代是两种常用的编程范式

共同点：通过组合较小规模子问题的解来获得问题的解。
在这种情况下，时间复杂度函数可以表示为递推关系。

如何求解递推关系？

递推关系

定义 1（递推关系）

设 a_0, a_1, \dots, a_n 是一个数列，简记为 $\{a_n\}$ 。递推关系使用前面的项来定义数列的每一项，并且总是给出数列的初始项。

- 递推关系刻画了一项对其前面各项的依赖关系

求解：给定数列 $\{a_n\}$ 的递推关系和一些初始值，计算 a_n 的通项公式。

- 通项公式： n 的函数，不涉及其他项

递推关系示例：Fibonacci 数

Fibonacci 数： 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

递推关系： $f_n = f_{n-1} + f_{n-2}$

初始值： $f_0 = 1, f_1 = 1$

[Fibonacci 图片]

$$f_n = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^{n+1} - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^{n+1}$$

设计的本质：Fibonacci 数与黄金分割

黄金分割的可视化

黄金分割无处不在（植物）

黄金分割无处不在（动物）

黄金分割无处不在（艺术）

黄金分割无处不在（建筑）

黄金分割无处不在（排版）

黄金分割无处不在（图像裁剪）

黄金分割无处不在（形状与符号）

接下来，我们介绍三种求解递推关系的方法。

目录

- 1 数列与级数求和
- 2 递推关系与算法分析
 - 方法一：直接迭代
 - 方法二：化简迭代
 - 方法三：递归树
- 3 主定理及其证明
- 4 主定理的应用

直接迭代的步骤

当递推关系简单时，即 $F(n)$ 仅依赖于 $F(n-1)$ ，使用直接迭代。

- ① 不断用公式的“右边的右边”替换“右边”
- ② 每次替换后，随着 n 减小，级数中出现一个新项
- ③ 替换直到达到初始值为止
- ④ 用初始值计算级数
- ⑤ 用数学归纳法验证解的正确性

注：数学归纳法可用于检验猜测是否正确。当正确性显而易见时，则不必使用。

示例：汉诺塔问题

起源：创世时，大梵天还建造了三根金刚石柱和 64 个大小不同的金盘。起初，这些盘子按大小升序放在一根柱子上，最小的在最上面，形成一个锥形。自那时起，婆罗门祭司们一直按照大梵天不可改变的规则将这些盘子从一根柱子移到另一根柱子。当最后一步完成时，世界将会终结。

作为谜题的问题

问题抽象：有三根柱子（标记为 A、B、C）和 n 个大小不同的盘子。起初，盘子按大小升序整齐地堆放在柱子 A 上。目标是求出将整个堆移到柱子 C 所需的最小移动次数 $T(n)$ ，并遵守以下规则：

- 每次只能移动一个盘子
- 每次移动是从一个堆的顶部取出盘子并放到另一个堆的顶部或空柱子上
- 大盘子不能放在小盘子上面

作为谜题的问题

问题抽象：有三根柱子（标记为 A、B、C）和 n 个大小不同的盘子。起初，盘子按大小升序整齐地堆放在柱子 A 上。目标是求出将整个堆移到柱子 C 所需的最小移动次数 $T(n)$ ，并遵守以下规则：

- 每次只能移动一个盘子
- 每次移动是从一个堆的顶部取出盘子并放到另一个堆的顶部或空柱子上
- 大盘子不能放在小盘子上面

例： $n = 1$, $T(1) = 1$; $n = 2$, $T(2) = 3$; $n = 3$, $T(3) = 7$

作为谜题的问题

问题抽象：有三根柱子（标记为 A、B、C）和 n 个大小不同的盘子。起初，盘子按大小升序整齐地堆放在柱子 A 上。目标是求出将整个堆移到柱子 C 所需的最小移动次数 $T(n)$ ，并遵守以下规则：

- 每次只能移动一个盘子
- 每次移动是从一个堆的顶部取出盘子并放到另一个堆的顶部或空柱子上
- 大盘子不能放在小盘子上面

例： $n = 1$, $T(1) = 1$; $n = 2$, $T(2) = 3$; $n = 3$, $T(3) = 7$

一般形式 $T(n) = ?$

汉诺塔的递归算法

Require: $A(n), B(0), C(0)$

Ensure: $A(0), B(0), C(n)$

```
1: if  $n = 1$  then  
2:    $\text{move}(A, C)$  {将一个盘子从 A 移到 C}  
3: else  
4:    $\text{Hanoi}(A, B, n - 1)$  {用 C 作为中转柱}  
5:    $\text{move}(A, C)$   
6:    $\text{Hanoi}(B, C, n - 1)$  {用 A 作为中转柱}  
7: end if
```

设 $T(n)$ 为移动 n 个盘子所需的移动次数

- $T(n) = 2T(n - 1) + 1$
- $T(1) = 1$

复杂度分析：直接迭代

$$\begin{cases} T(n) = 2T(n-1) + 1 \\ T(1) = 1 \end{cases} \Rightarrow T(n) = 2^n - 1$$

复杂度分析：直接迭代

$$\begin{cases} T(n) = 2T(n-1) + 1 \\ T(1) = 1 \end{cases} \Rightarrow T(n) = 2^n - 1$$

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 2^2 T(n-2) + 2 + 1 \\ &= \dots \\ &= 2^{n-1} T(1) + 2^{n-2} + \dots + 2 + 1 \quad // \text{达到初始项} \\ &= 2^{n-1} \cdot 1 + 2^{n-1} - 1 \quad // \text{代入初始值} \\ &= 2^n - 1 \end{aligned}$$

关于汉诺塔的更多内容

有更好的算法吗？

关于汉诺塔的更多内容

有更好的算法吗？

没有！汉诺塔是一个难解问题，目前没有已知的多项式时间算法。

关于汉诺塔的更多内容

有更好的算法吗？

没有！汉诺塔是一个难解问题，目前没有已知的多项式时间算法。

问：每秒移动 1 次，移动 64 个盘子需要多长时间？

关于汉诺塔的更多内容

有更好的算法吗？

没有！汉诺塔是一个难解问题，目前没有已知的多项式时间算法。

问：每秒移动 1 次，移动 64 个盘子需要多长时间？

答：5000 亿年！对算法是坏消息，但对世界是好消息！

示例：插入排序的迭代算法

Require: 未排序的 $A[n]$

Ensure: 升序排列的 $A[n]$

```
1: for  $j \leftarrow 2$  to  $n$  do  
2:    $x \leftarrow A[j]$   
3:    $i \leftarrow j - 1$  {将  $A[j]$  插入  $A[1 \dots j - 1]$ }  
4:   while  $i > 0$  and  $x < A[i]$  do  
5:      $A[i + 1] \leftarrow A[i]$ ,  $i \leftarrow i - 1$   
6:   end while  
7:    $A[i + 1] \leftarrow x$   
8: end for
```

最坏情况复杂度

基本计算步骤：元素比较

输入规模： n

$$\begin{cases} W(n) = W(n-1) + (n-1) \\ W(1) = 0 \end{cases} \Rightarrow W(n) = n(n-1)/2$$

- 插入第 i 个元素时，算法将其与前 $i-1$ 个已排序元素比较；最大比较次数为 $i-1$

用直接迭代求解递推关系

$$\begin{aligned}W(n) &= W(n-1) + n - 1 \\&= (W(n-2) + n - 2) + n - 1 \\&= W(n-2) + n - 2 + n - 1 \\&= \dots \\&= W(1) + 1 + 2 + \dots + (n-2) + (n-1) \quad // \text{达到初始项} \\&= 0 + 1 + 2 + \dots + (n-2) + (n-1) \quad // \text{代入初始值} \\&= n(n-1)/2\end{aligned}$$

数学归纳法（可追溯到公元前 370 年，Plato 的《巴门尼德篇》）

数学归纳法是一种数学证明技术 \Rightarrow 证明性质 $P(n)$ 对每个自然数 $n \in \mathbb{N}$ 成立。

数学归纳法证明我们可以爬到梯子上任意高的地方，通过证明我们可以踏上最底层的横档（基础），并且从每一个横档我们都可以爬到下一个横档（归纳步骤）。

— 《具体数学》

数学归纳法的模板

归纳法需要证明两个事实：

归纳基础：证明性质对数字 0 成立。

归纳步骤：

- 1 证明如果性质对 n 成立，则对 $n + 1$ 也成立

$$P(0) = 1, \quad \forall n, P(n) = 1 \Rightarrow P(n + 1) = 1$$

- 2 证明如果性质对所有 $k < n$ 成立，则对 n 也成立

$$P(0) = 1, \quad \forall k < n, P(k) = 1 \Rightarrow P(n) = 1$$

两种数学归纳法的比较

归纳基础相同: $P(0) = 1$

归纳步骤不同

逻辑推理:

- ① 第一类归纳: $P(0) = 1 \Rightarrow P(1) = 1 \Rightarrow P(2) = 1$
- ② 第二类归纳: $P(0) = 1 \Rightarrow P(0) = 1 \wedge P(1) = 1 \Rightarrow \dots$

思考: 直觉相同, 何时用哪种?

- 第一类 (松耦合): 下一个数的性质仅依赖于其最近的前驱
- 第二类 (紧耦合): 下一个数的性质依赖于其所有前驱

关于数学归纳法的说明

这两步建立了性质 $P(n) = 1$ 对每个自然数 $n = 0, 1, 2, 3$ 成立。

- 基础情况不一定从 $n = 0$ 开始。它可以从任何自然数 n_0 开始，建立 $P(n) = 1$ 对所有 $n \geq n_0$ 成立。
- 该方法可以扩展到结构归纳法 \Rightarrow 证明关于更一般的良基结构（如树）的命题（在数理逻辑和计算机科学中广泛使用）。

验证解的正确性：数学归纳法

命题： $W(n) = n(n-1)/2$ 是递推关系的通项公式

$$\begin{cases} W(n) = W(n-1) + n - 1 \\ W(1) = 0 \end{cases}$$

方法：数学归纳法

- ① 基础： $n = 1, W(1) = 1 \times (1-1)/2 = 0 \checkmark$
- ② 归纳步骤： $P(n) = 1 \Rightarrow P(n+1) = 1$:

$$\begin{aligned} W(n+1) &= W(n) + n \\ &= n(n-1)/2 + n \quad // \text{前提} \\ &= n((n-1)/2 + 1) = n(n+1)/2 \checkmark \end{aligned}$$

直接迭代的变体：换元迭代

当 n 本身是另一个变量（如 k ）的函数，且每次迭代后 k 减少 1 时，我们首先将 n 换元为 k 的函数，然后对 k 应用迭代方法。

- ① 将关于 n 的递推公式转换为关于 k 的递推公式
- ② 对 k 进行迭代
- ③ 将关于 k 的通项公式转换回关于 n 的通项公式

归并排序算法

Require: 未排序的 $A[n]$

Ensure: 升序排列的 $A[n]$

- 1: $l \leftarrow 1, r \leftarrow n$
- 2: **if** $l < r$ **then**
- 3: $k \leftarrow \lfloor (l + r) / 2 \rfloor$
- 4: MergeSort(A, l, k)
- 5: MergeSort($A, k + 1, r$)
- 6: Merge(A, l, k, r)
- 7: **end if**

换元迭代示例 (1/2)

假设 $n = 2^k$, 递推关系为:

$$\begin{cases} W(n) = 2W(n/2) + n - 1 \\ W(1) = 0 \end{cases}$$

- $n - 1$ 是归并的代价

换元: $n \rightarrow 2^k$

$$\begin{cases} W(2^k) = 2W(2^{k-1}) + 2^k - 1 \\ W(2^0 = 1) = 0 \end{cases}$$

换元迭代示例 (2/2)

$$\begin{aligned} W(n) &= 2W(2^{k-1}) + 2^k - 1 \quad // \text{换元并对 } k \text{ 迭代} \\ &= 2(2W(2^{k-2}) + 2^{k-1} - 1) + 2^k - 1 \quad // \text{第 1 轮迭代} \\ &= 2^2 W(2^{k-2}) + 2^k - 2 + 2^k - 1 \\ &= 2^2(2W(2^{k-3}) + 2^{k-2} - 1) + 2^k - 2 + 2^k - 1 \quad // \text{第 2 轮迭代} \\ &= \dots \\ &= 2^k W(2^0 = 1) + k \cdot 2^k - (2^{k-1} + 2^{k-2} + \dots + 2 + 1) \\ &= 0 + k \cdot 2^k - 2^k + 1 \\ &= n \log n - n + 1 \quad // \text{换元回去} \end{aligned}$$

目录

- 1 数列与级数求和
- 2 递推关系与算法分析
 - 方法一：直接迭代
 - 方法二：化简迭代
 - 方法三：递归树
- 3 主定理及其证明
- 4 主定理的应用

化简迭代

动机

- 求解递推关系的基本方法是迭代
- 当原始递推关系复杂时，需要化简
 - ▶ 将高阶方程转化为一阶方程（减少依赖），然后换元

快速排序示例

快速排序回顾

假设 $A[n]$ 中元素互不相同, 设 $l \leftarrow 1$, $r \leftarrow n$, 用第一个元素 $A[1] = x$ 对 $A[l \dots r]$ 进行划分, 使得:

- 小于 x 的元素存储在 $A[l \dots k-1]$
- 大于 x 的元素存储在 $A[k+1 \dots r]$
- $A[1]$ 放在 $A[k]$

递归地对 $A[l \dots k-1]$ 和 $A[k+1 \dots r]$ 排序

总体复杂度:

- 子问题的复杂度
- 划分的复杂度

输入与子问题规模

根据第一个元素 x 在最终排序数组中的位置，可将输入分为 n 种情况

x 的最终位置	子问题 1 的规模	子问题 2 的规模
1	0	$n - 1$
2	1	$n - 2$
3	2	$n - 3$
\vdots	\vdots	\vdots
$n - 1$	$n - 2$	1
n	$n - 1$	0

对于每个输入，划分所需的比较次数恰好是 $n - 1$ （想想为什么？）

复杂度求和

$$\begin{aligned} &T(0) + T(n-1) + n-1 \\ &T(1) + T(n-2) + n-1 \\ &T(2) + T(n-3) + n-1 \\ &\quad \vdots \\ &T(n-1) + T(0) + n-1 \end{aligned}$$

求和: $2(T(1) + \cdots + T(n-1)) + n(n-1)$

快速排序的平均复杂度

假设：第一个元素 x 最终以相同概率出现在每个位置：

$$T(n) = \frac{2}{n} \sum_{i=1}^{n-1} T(i) + O(n), \quad n \geq 2$$

$$T(1) = 0, \quad T(0) = 0$$

观察与思路

- 递推关系复杂：第 n 项依赖于所有前面的项 \rightarrow 直接迭代会非常复杂
- 思路：化简复杂方程，然后迭代

通过相减化简

改写并迭代一次得到两个递推关系，然后尝试化简右边的项。

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

$$nT(n) = 2 \sum_{i=1}^{n-1} T(i) + n(n-1)$$

$$(n-1)T(n-1) = 2 \sum_{i=1}^{n-2} T(i) + (n-1)(n-2)$$

通过相减化简（续）

相减

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1)$$

化简

$$nT(n) = (n+1)T(n-1) + \Theta(n)$$

改写

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{\Theta(n)}{n(n+1)} = \frac{T(n-1)}{n} + \frac{\Theta(1)}{n+1}$$

迭代

$$\begin{aligned}\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{\Theta(1)}{n+1} = \dots \\ &= \Theta(1) \left(\frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{3} \right) + \frac{T(1)}{2} \quad // \text{达到初始项} \\ &= \Theta(1) \left(\frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{3} \right) \quad // \text{代入初始值} \\ &= \Theta(\ln n)\end{aligned}$$

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

目录

- ① 数列与级数求和
- ② 递推关系与算法分析
 - 方法一：直接迭代
 - 方法二：化简迭代
 - 方法三：递归树
- ③ 主定理及其证明
- ④ 主定理的应用

递归树的概念

当 $F(n)$ 依赖于若干不连续的前项时，我们可以尝试用递归树求解递推关系。

- 递归树是递归计算的模型，也是递推关系的迭代
- 递归树的生成与递归过程相同
- 递归树上的节点正是递归级数中的项
- 递归树上所有节点（包括内部节点和叶节点）的和就是递推关系的解

递归树中迭代的表示

递归树是递归的模型 \Rightarrow 与求解递推关系密切相关
假设递推关系如下：

$$T(n) = T(n_1) + \cdots + T(n_t) + f(n), \quad |n_1|, \dots, |n_t| < |n|$$

- $T(n_1), \dots, T(n_t)$ ：函数项
- $f(n)$ ：分解代价 + 合并代价

如何在递归树上表示 $T(n)$?

递归树的可视化

- 根节点是合并和分解的代价
- 每个叶节点是一个函数项
- 所有节点之和 = $T(n)$

2 层递归树示例

归并排序的递推关系

$$\begin{cases} T(n) = 2T(n/2) + (n - 1) \\ T(2) = 1 \\ T(1) = 0 \end{cases}$$

递归树的生成规则

- ① 最初，递归树中只有根节点，其值为 $T(n)$
- ② 重复以下步骤：
 - ▶ 将叶节点中的函数项 $T(n)$ 表示为 2 层子树
 - ▶ 用这个子树替换叶节点
- ③ 继续生成递归树，直到树中没有函数项为止
 - ▶ 达到叶节点——初始值

平衡递归树生成演示

完整的递归树

计算递归树的和（平衡情况）

$$\begin{cases} T(n) = 2T(n/2) + n - 1, & n = 2^k \\ T(1) = 0 \end{cases}$$

$$\begin{aligned} T(n) &= \underbrace{(n-1)}_{\text{第 0 层}} + \underbrace{(n-2)}_{\text{第 1 层}} + \cdots + \underbrace{(n-2^{k-1})}_{\text{第 } (k-1) \text{ 层}} \\ &= kn - (2^k - 1) \quad // k = \log n \\ &= n \log n - n + 1 \end{aligned}$$

递归树的应用（非平衡情况）

计算通项公式

$$T(n) = T(n/3) + T(2n/3) + n$$

不同路径到达初始值的速率不同

- 左边路径最快——估计下界
- 右边路径最慢——估计上界

计算递归树的和（非平衡情况）

递推关系： $T(n) = T(n/3) + T(2n/3) + n$

递归树的深度为 k ，每层的和为 $O(n)$

估计最长路径以计算上界

$$n \left(\frac{2}{3}\right)^k = 1 \Rightarrow \left(\frac{3}{2}\right)^k = n \Rightarrow k = \log_{3/2} n$$

$$T(n) < \log_{3/2} n \times n = O(n \log n)$$

估计最短路径以计算下界

$$T(n) > \log_3 n \times n = \Omega(n \log n)$$

综合以上， $T(n) = \Theta(n \log n)$

说明

为简单起见，表示初始值的叶节点不包含在求和中。

- 初始值通常不能用 $f(n)$ 表示
- 初始值通常是常数，如 0 或 1，因此可以单独计算

目录

① 数列与级数求和

② 递推关系与算法分析

- 方法一：直接迭代
- 方法二：化简迭代
- 方法三：递归树

③ 主定理及其证明

④ 主定理的应用

主定理的应用

求解递推关系

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

- a : 分解后的子问题数
- n/b : 子问题的规模
- $f(n)$: 分解和合并子问题的代价

例

- 二分查找: $T(n) = T(n/2) + 1$
- 归并排序: $T(n) = 2T(n/2) + n - 1$

主定理

设 $a \geq 1$, $b \geq 1$ 为常数, $T(n)$ 和 $f(n)$ 为函数, 且

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

- ① 若 $\exists \varepsilon > 0$ 使得 $f(n) = O(n^{(\log_b a) - \varepsilon})$, 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 $\exists \varepsilon > 0$ 使得 $f(n) = \Omega(n^{(\log_b a) + \varepsilon})$, 且 $\exists r < 1$ 使得对所有 n (可放宽为对足够大的 n) 不等式 $af(n/b) \leq rf(n)$ 成立, 则:

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

如何证明主定理?

直接迭代

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

为方便起见, 设 $n = b^k$

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + f(n) \\ &= a\left(aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right) + f(n) \\ &= a^2 T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) \\ &= \dots \end{aligned}$$

迭代结果

$$= a^k T\left(\frac{n}{b^k}\right) + a^{k-1} f\left(\frac{n}{b^{k-1}}\right) + \cdots + a f\left(\frac{n}{b}\right) + f(n)$$

$$= a^k T(1) + \sum_{j=0}^{k-1} a^j f\left(\frac{n}{b^j}\right) \quad // \text{达到初始项}$$

$$= c_1 n^{\log_b a} + \sum_{j=0}^{k-1} a^j f\left(\frac{n}{b^j}\right) \quad // \text{假设 } T(1) = c_1$$

$$k = \log_b n = \log_b a \cdot \log_a n$$

- 第一项是所有基础子问题的总代价
- 第二项是所有分解和合并步骤的总代价

对应的递归树

- 分支因子 a
- b : 每递归一层, 子问题规模减少 b 倍
- $k = \log_b n$: 经过 k 层后达到基础情况, 即递归树的高度
- 树的第 j 层由 a^j 个子问题组成, 每个规模为 n/b^j

情况 1

$\exists \varepsilon > 0$ 使得 $f(n) = O(n^{(\log_b a) - \varepsilon})$

$$\begin{aligned} T(n) &= c_1 n^{\log_b a} + \sum_{j=0}^{(\log_b n) - 1} a^j f\left(\frac{n}{b^j}\right) \\ &= c_1 n^{\log_b a} + O\left(\sum_{j=0}^{(\log_b n) - 1} a^j \left(\frac{n}{b^j}\right)^{(\log_b a) - \varepsilon}\right) \quad // \text{代入前提} \\ &= c_1 n^{\log_b a} + O\left(n^{(\log_b a) - \varepsilon} \sum_{j=0}^{(\log_b n) - 1} \frac{a^j}{(b^{(\log_b a) - \varepsilon})^j}\right) \end{aligned}$$

情况 1: 继续化简

$$\frac{1}{(b^{\log_b a - \varepsilon})^j} = \frac{b^{\varepsilon j}}{(b^{\log_b a})^j} = \frac{b^{\varepsilon j}}{a^j}$$

$$= c_1 n^{\log_b a} + O\left(n^{(\log_b a) - \varepsilon} \sum_{j=0}^{\log_b n - 1} (b^{\varepsilon})^j\right) \quad // \text{化简}$$

$$= c_1 n^{\log_b a} + O\left(n^{(\log_b a) - \varepsilon} \cdot \frac{b^{\varepsilon \log_b n} - 1}{b^{\varepsilon} - 1}\right) \quad // \text{等比级数}$$

$$= c_1 n^{\log_b a} + O\left(n^{(\log_b a) - \varepsilon} \cdot n^{\varepsilon}\right) = \Theta(n^{\log_b a})$$

情况 2

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

$$\begin{aligned} T(n) &= c_1 n^{\log_b a} + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) \\ &= c_1 n^{\log_b a} + \Theta\left(\sum_{j=0}^{(\log_b n) - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) \quad // \text{代入前提} \\ &= c_1 n^{\log_b a} + \Theta\left(n^{\log_b a} \sum_{j=0}^{(\log_b n) - 1} \frac{a^j}{a^j}\right) \\ &= c_1 n^{\log_b a} + \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \log n) \end{aligned}$$

情况 3

$$\exists \varepsilon > 0, f(n) = \Omega(n^{(\log_b a) + \varepsilon}) \quad (1)$$

$$af(n/b) \leq rf(n) \quad (2)$$

反复应用条件 (2):

$$a^j f\left(\frac{n}{b^j}\right) \leq a^{j-1} rf\left(\frac{n}{b^{j-1}}\right) \leq \cdots \leq r^j f(n)$$

$$\begin{aligned} T(n) &= c_1 n^{\log_b a} + \sum_{j=0}^{(\log_b n)-1} a^j f\left(\frac{n}{b^j}\right) \\ &= c_1 n^{\log_b a} + f(n) + \sum_{j=1}^{(\log_b n)-1} a^j f\left(\frac{n}{b^j}\right) \end{aligned}$$

情况 3 (续)

$$\begin{aligned}T(n) &\leq c_1 n^{\log_b a} + f(n) + \sum_{j=1}^{(\log_b n)-2} r^j f(n) \\&= c_1 n^{\log_b a} + f(n) + f(n)r \cdot \frac{1 - r^{(\log_b n)-2}}{1 - r} \quad // \text{等比级数: } r < 1 \\&= c_1 n^{\log_b a} + \Theta(f(n))\end{aligned}$$

$$\text{条件 1} \Rightarrow \text{order}(f(n)) \geq \text{order}(n^{\log_b a})$$

因此:

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

回顾: 条件 2 用于证明 $f(n)$ 的系数被常数上界。

主定理的简化形式

定义 $h(n) = n^{\log_b a}$ ，我们将主定理重述如下：

$$T(n) = \begin{cases} \Theta(h(n)) & \text{若 } f(n) = o(h(n)) \\ \Theta(h(n) \log n) & \text{若 } f(n) = \Theta(h(n)) \\ \Theta(f(n)) & \text{若 } f(n) = \omega(h(n)) \wedge \exists r < 1 \text{ 使得 } af(n/b) < rf(n) \end{cases}$$

目录

1 数列与级数求和

2 递推关系与算法分析

- 方法一：直接迭代
- 方法二：化简迭代
- 方法三：递归树

3 主定理及其证明

4 主定理的应用

求解递推关系示例 1

计算递推关系的通项公式：

$$T(n) = 9T(n/3) + n$$

应用主定理

- $a = 9, b = 3, h(n) = n^2, \varepsilon = 1$
- $f(n) = n, n^{\log_3 9} = n^2, f(n) = O(n^{\log_3 9 - 1}) = o(n^2)$

主定理（情况 1） $\Rightarrow T(n) = \Theta(n^2)$

求解递推关系示例 2

计算递推关系的通项公式：

$$T(n) = T(2n/3) + 1$$

应用主定理

- $a = 1, b = 3/2, h(n) = n^{\log_b a} = n^0 = 1$
- $f(n) = 1, n^{\log_{3/2} 1} = n^0 = 1, f(n) = \Theta(1)$

主定理（情况 2） $\Rightarrow T(n) = \Theta(\log n)$

求解递推关系示例 3

计算递推关系的通项公式：

$$T(n) = 3T(n/4) + n \log n$$

应用主定理

- $a = 3, b = 4, h(n) = n^{\log_4 3}$
- $f(n) = n \log n = \Omega(n^{\log_4 3 + \varepsilon}) \approx \Omega(n^{0.793 + \varepsilon})$, 取 $\varepsilon = 0.2$
- 检验附加条件 $af(n/b) \leq rf(n)$ 对所有 n 成立
 - ▶ 检验 $f(n) = n \log n \Rightarrow af(n/b) = 3(n/4) \log(n/4) \leq rn \log n$ 对某个 $r < 1$ 成立
 - ▶ 取 $r = 3/4 < 1$, 此不等式对所有 n 成立

主定理 (情况 3) $\Rightarrow T(n) = \Theta(f(n)) = \Theta(n \log n)$

递归算法的复杂度分析

二分查找: $T(n) = T(n/2) + 1$, $T(1) = 1$

- $a = 1$, $b = 2$, $h(n) = n^{\log_2 1} = 1$, $f(n) = 1$

主定理 (情况 2) $\Rightarrow T(n) = \Theta(\log n)$

归并排序: $T(n) = 2T(n/2) + (n - 1)$, $T(1) = 0$

- $a = 2$, $b = 2$, $h(n) = n^{\log_2 2} = n$, $f(n) = n - 1$

主定理 (情况 2) $\Rightarrow T(n) = \Theta(n \log n)$

主定理不适用的情况

例：计算通项公式

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

应用主定理： $a = b = 2$, $h(n) = n^{\log_b a} = n$, $f(n) = n \log n$

只有情况 3 可能适用，但 $\nexists r < 1$ 使得 $af(n/b) \leq rf(n)$ 对所有 n 成立。

$$\begin{aligned} af(n/b) - rf(n) &= 2(n/2) \log(n/2) - rn \log n \\ &= n(\log n - 1) - rn \log n \\ &= (1 - r)n \log n - n > 0 \quad \text{若 } r < 1 \end{aligned}$$

通过递归树求解

求和

$$\begin{aligned}T(n) &= n \log n + n(\log n - 1) + n(\log n - 2) \\&\quad + \cdots + n(\log n - k + 1) \\&= (n \log n) \log n - n(1 + 2 + \cdots + k - 1) \\&= n \log^2 n - nk(k - 1)/2 \quad // \text{代入 } k = \log n \\&= \Theta(n \log^2 n)\end{aligned}$$

小结

- 经典数列与级数
- 复杂度分析：求解递推关系
 - ▶ 直接迭代
 - ▶ 化简迭代
 - ▶ 递归树
- 主定理及其证明
- 主定理的应用