算法设计与分析

2. 递归与分治策略

主要内容

2.1 递归的概念
2.2 分治法的基本思想
2.3 二分搜索技术
2.4 大整数的乘法
2.5 Strassen矩阵乘法
2.6 棋盘覆盖
2.7 合并排序
2.8 快速排序
2.9 线性时间选择
2.10 最接近点对问题
2.11 循环赛日程表

2.0 学习要点

- 理解递归的概念。
- 会求解一些较特殊的递推方程
- 掌握设计有效算法的分治策略，能确定常见问题的复杂性的阶
- 通过典型范例，学习分治策略设计技巧。

2.1 递归的概念

- 递归函数：使用函数自身给出定义的函数
- 递归算法：一个直接或间接地调用自身的算法
- 递归方程：对于递归算法，一般可把时间代价表示为一个递归方程
    - 解递归方程最常用的方法是进行递归扩展

递归函数举例(1)

- 阶乘函数

$$
n! = \begin{cases} 1 & n=1 \\ n(n-1)! & n>1 \end{cases}
$$

- Fibonacci数列

$$
F(n) = \begin{cases} 1 & n=0 \\ 1 & n=1 \\ F(n-1)+F(n-2) & n>1 \end{cases}
$$

初始条件与递归方程是递归函数的二个要素

递归函数举例(2)

- Ackerman函数

$$
\begin{cases}
A(1,0)=2 & \\
A(0,m)=1 & m>=0 \\
A(n,0)=n+2 & n>=2 \\
A(n,m)=A(A(n-1,m),m-1) & n,m>=1
\end{cases}
$$

- Ackerman函数:双递归函数
- 双递归函数：一个函数及它的一个变量由函数自身定义

递归函数举例(2)

- Ackerman函数

$$
\begin{cases}
A(1,0)=2 & \\
A(0,m)=1 & m>=0 \\
A(n,0)=n+2 & n>=2 \\
A(n,m)=A(A(n-1,m),m-1) & n,m>=1
\end{cases}
$$

A(n，m)的自变量m的每一个值都定义了一个单变量函数：

- m=0时，$A(n,0)=n+2, n>=2$； $A(1,0)=2$
- 且 $A(1,1)=2$，故$A(n,1)=2*n+1$m=1时，$A(n,1)=A(A(n-1,1),0)=A(n-1,1)+2$，

递归函数举例(2)

- Ackerman函数

$$
\begin{cases}
A(1,0)=2 & \\
A(0,m)=1 & m>=0 \\
A(n,0)=n+2 & n>=2 \\
A(n,m)=A(A(n-1,m),m-1) & n,m>=1
\end{cases}
$$

- m=2时，$A(n,2) = A(A(n-1,2),1) = 2A(n-1,2)$，且 $A(1,2) = A(A(0,2),1) = A(1,1) = 2$，故$A(n,2) = 2\text{^}n$ 。
- m=3时，类似的可以推出 $A(n,3) = \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{n}$
- m=4时，A(n,4)的增长速度非常快，以至于没有适当的数学式子来表示这一函数。

递归函数举例(2)

- Ackerman函数

$$
\begin{cases}
A(1,0)=2 & \\
A(0,m)=1 & m>=0 \\
A(n,0)=n+2 & n>=2 \\
A(n,m)=A(A(n-1,m),m-1) & n,m>=1
\end{cases}
$$

单变量的Ackerman函数A(n)定义为 A(n)=A(n，n)。

其拟逆函数$\alpha(n)$定义为：$\alpha(n)=\min\{k \mid A(k)\geq n\}$。
$\alpha(n)$：随n的增长非常慢

递归算法举例(1)

- **排列问题：**

设R={r_1，r_2，…，r_n}是要进行排列的n个元素，R_i=R－{r_i}。集合X中元素的全排列记为Perm(X)。(r_i)Perm(X)表示在全排列Perm(X)的每一个排列前加上前缀r_i得到的排列。

R的全排列可归纳定义如下：

当n＝1时，Perm (R) = （r），其中r是集合R中唯一的元素；

当n>1时，Perm(R)由(r_1)Perm(R_1), (r_2)Perm(R_2), …, (r_n)Perm(R_n)构成。

产生Perm(R)的递归算法

```
void Swap(Type & a, Type & b)  // Swap是交换a, b的值
void Perm(Type list[], int k, int m) { // 生成全部排列
  list[k:m].
     if (k == m) {
        for(int  i  =  0;i  <  =  m;i  ++)  cout  <<
  list[i];
        cout << endl;
       }
       else // list[k: m] 有多于一个置换, 递归产生
       for (int i = k; i < = m; i++) {
             Swap(list[k], 1ist[i] );
             Perm(list, k + 1, m);
            Swap(list[ k], list[ i] );
        }
}
```

调用

Perm(list,0,n-1)

递归算法举例(2)

- **整数划分问题:**

给定正整数n, 求其不同的划分个数p(n)。

若n可以表示为n＝ n1+n2+…+nk,

且n1>= n2>=…>=nk ,

我们称n1, n2, …, nk 是n的一个划分。

例：正整数6有如下11种不同的划分:

6;

5+1;

4+2，4+1+1;

3+3，3+2+1，3+1+1+1;

2+2+2，2+2+1+1，2+1+1+1+1;

1+1+1+1+1+1。

递归算法举例(2)

若n可以表示为n＝ n1+n2+…+nk,
且n1>= n2>=…>=nk ,
我们称n1, n2, …, nk 是n的一个划分。

- ~~直接找划分个数p(n)的递归关系?~~
- q(n,m):正整数n的不同的划分中，最大加数不大于m的划分个数

$$q(n,m)=\begin{cases}1 & n=1,\ m=1\\ q(n,n) & n<m \\ 1+q(n,n-1) & n=m \\ q(n,m-1)+q(n-m,m) & n>m>1\end{cases}$$

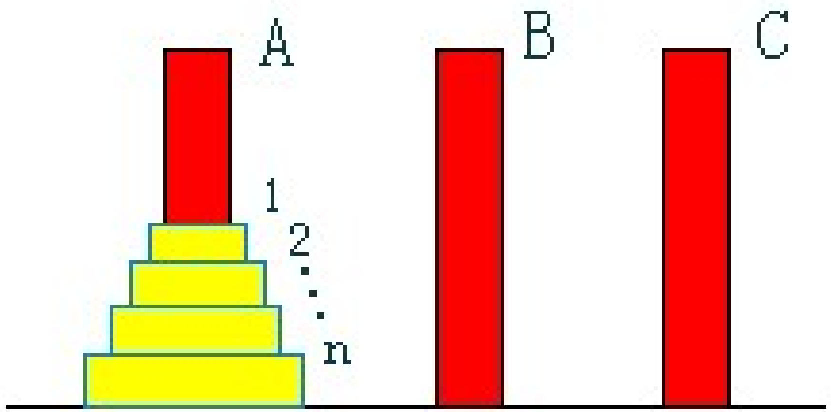
递归算法举例(3)

“Hanoi 塔”问题： 设a,b,c是3个塔座。开始时，在塔座a上有一叠共n个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为1,2,…,n,现要求将塔座a上的这一叠圆盘移到塔座b上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

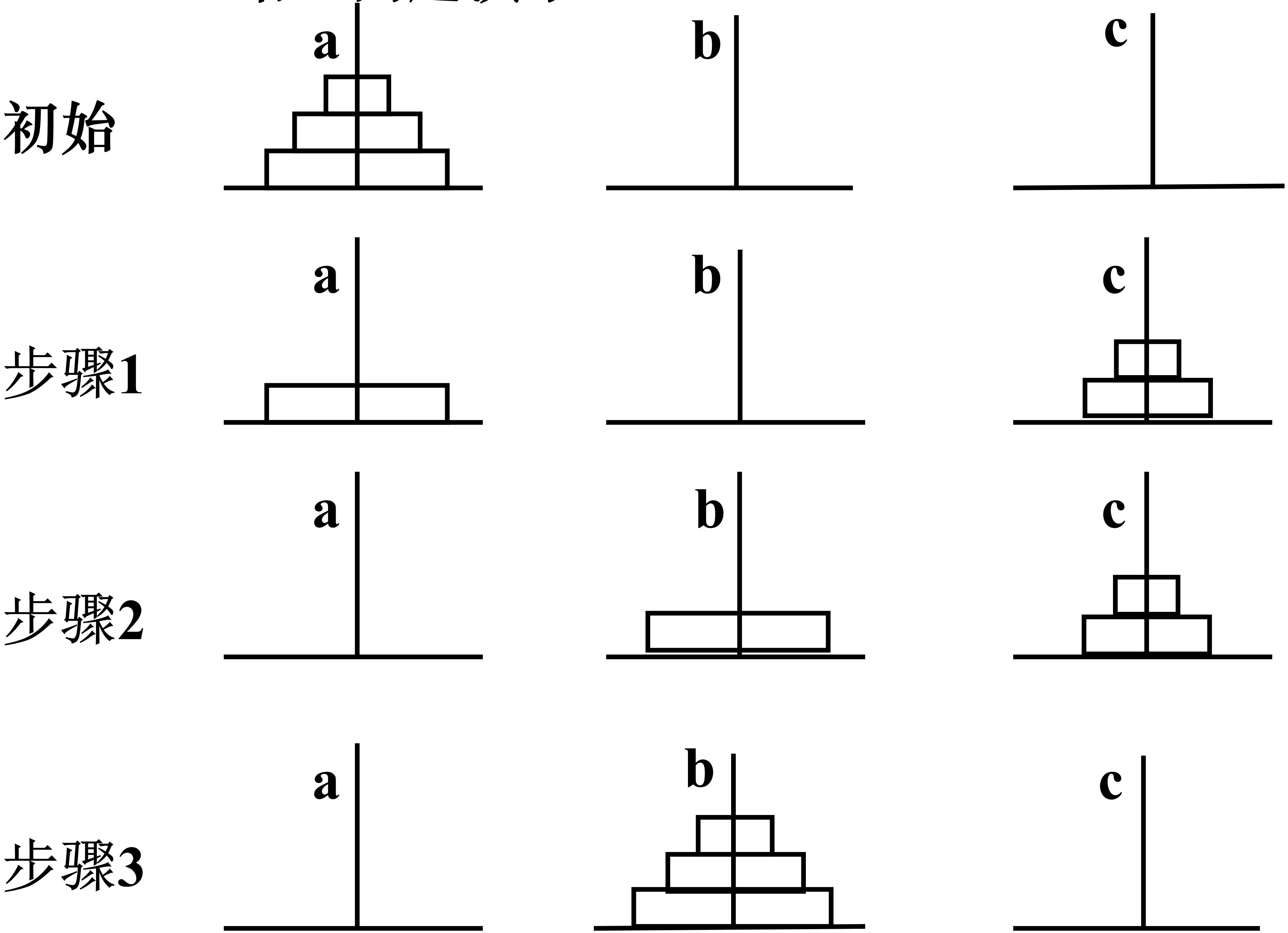
规则1：每次只能移动1个圆盘；

规则2：任何时刻都不允许将较大的圆盘压在较小的圆盘之上；

规则3：在满足移动规则1和2的前提下，可将圆盘移至a,b,c中任一塔座上。



“Hanoi 塔”问题演示



“Hanoi 塔”问题程序

```
void hanoi(int n,a,b,c) //A上n个盘借助C移到B
 {if(n == 1)move(1,a,b); //将1个盘从a移到b
       else { hanoi( n-1,a,c,b );
              move(n,a,b); //将第n个盘从a移到b
              hanoi( n-1,c,b,a) ;
            }
 }
```

“Hanoi 塔”问题移动次数递推关系

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

递归程序代价

- 递归程序每次调用需要分配不同的运行空间，一旦某一层被启用，就要为之开辟新的空间。而当一层执行完毕，释放相应空间掉，退到上一层。
- 当递归过程每层所需空间为常量C时，整个动态空间的代价就与递归的深度有关。如果递归深度为 h ，动态空间代价为C*h。
- 递归优点：结构清晰，可读性强
- 递归缺点：递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。

2.2 分治法的基本思想

- 分治策略是一种用得最多的一种有效方法
- 基本思想：将问题分解成若干子问题，然后求解子问题。子问题较原问题更容易些，由此得出原问题的解，就是所谓的“分而治之”的意思。
- 分治策略可以递归进行，即子问题仍然可以用分治策略来处理，但最后的问题要非常基本而简单。

2.2.1 分治法概述

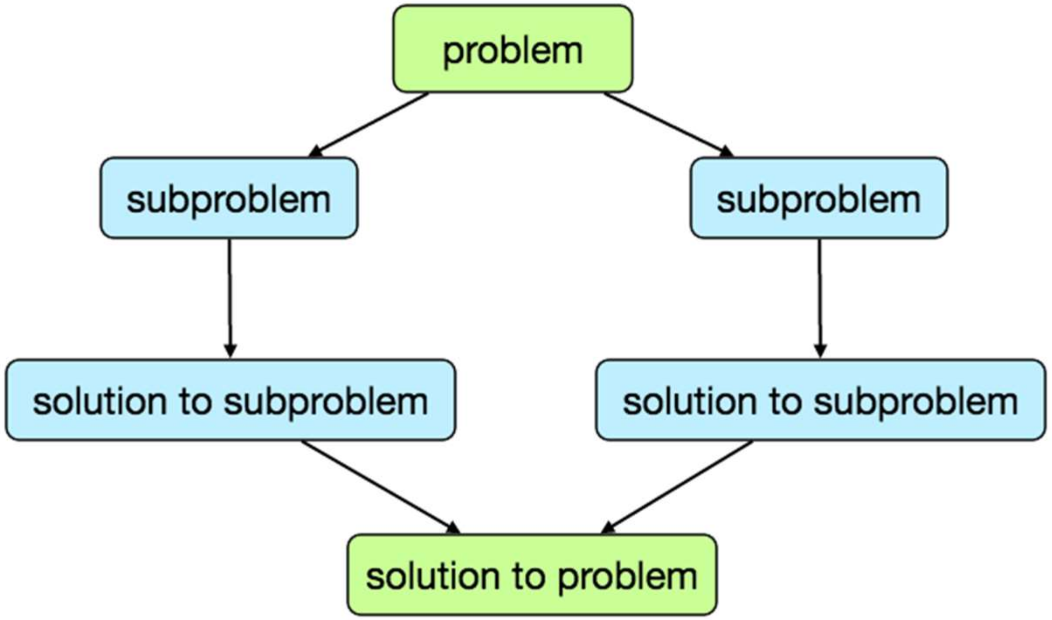
设：被求解问题的输入规模为n。

步骤1：把问题分解为k个性质相同、但规模较小的子问题（$1 \le k \le n$），并求解这些子问题。

（如果这些子问题的规模还不够“小”，则可以进一步分解，直到可以求解为止）

步骤2：逐步合并子问题的解，直到获得原问题的解。

2.2.1 分治法概述


problem
subproblem
subproblem
solution to subproblem
solution to subproblem
solution to problem

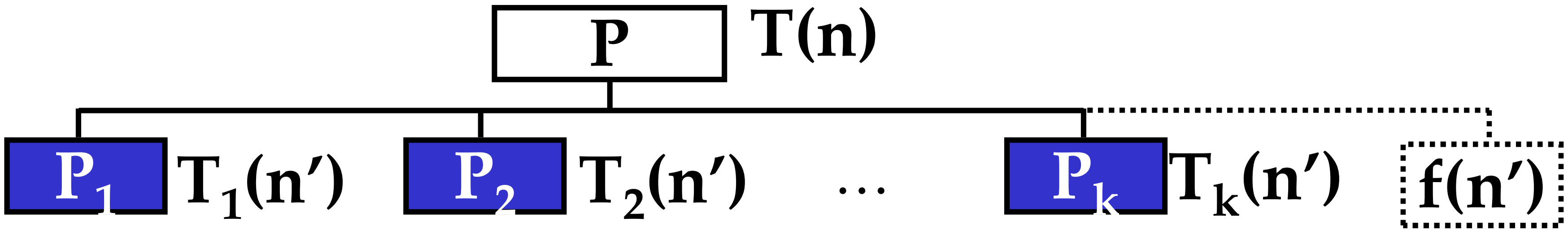
2.2.1 分治法概述

problem
divide
subproblem
subproblem
divide
divide
subproblem
subproblem
subproblem
subproblem
divide
divide
divide
divide
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
conquer
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
combine
combine
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
solution to subproblem
solution to subproblem
combine
solution to problem

2.2.2 分治法算法构架

```
divide-and-conquer(P)
{
    if ( | P | <= n0)  adhoc(P);   //直接解决小规模的问题
    将 P分解为子问题 P1,P2,...,Pk; //分解问题
    for (i=1,i<=k,i++)
      yi=divide-and-conquer(Pi); //递归地解各子问题
    return merge(y1,...,yk); //将各子问题的解合并为原问
  题的解
 }
```

2.2.3 分治法示例



有：$T(n) = T_1(n') + T_2(n') + \ldots + T_k(n') + f(n')$

2.2.4 代价分析

$$T(n) = \begin{cases} O(1) & n=1 \\ kT(n/m) + f(n) & n>1 \end{cases}$$

记$n=m^t$, 则$T(n)=k^tT(1)+\sum_{j=0}^{\log_m n-1} k^j f(n/m^j)$

$$=k^{\log_m n} + \sum_{j=0}^{\log_m n-1} k^j f(n/m^j)$$

注意：递归方程解只给出n等于m的方幂时T(n)的值，但是如果认为T(n)足够平滑，那么由$n= m^t$时T(n)的值可以估计T(n)的增长速度。通常假定T(n)是单调上升的，从而当$m^i \le n < m^{i+1}$时，$T(m^i) \le T(n) < T(m^{i+1})$。该处理方式有一般性。

主定理的应用背景

求解递推方程

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

a:归约后的子问题个数
n/b:归约后子问题的规模
f(n):规约过程及组合子问题的解的工作量

二分检索： $T(n) = T(n/2) + 1$

二分归并排序： $T(n) = 2T(n/2) + n - 1$

主定理

定理： 设 $\boldsymbol{a} \geq \mathbf{1}, \boldsymbol{b} > \mathbf{1}$为常数。$f(n)$为函数，$\mathrm{T}(n)$为非负整数，且$\boldsymbol{T}(\boldsymbol{n}) = \boldsymbol{aT}(\boldsymbol{n}/\boldsymbol{b}) + \boldsymbol{f}(\boldsymbol{n})$, 则

1. 若 $f(n) = O\left(n^{\log_b a-\varepsilon}\right), \varepsilon > 0$, 那么

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

2. 若 $f(n) = O\left(n^{\log_b a}\right)$, 那么

$$T(n) = \Theta\left(n^{\log_b a}\log n\right)$$

3. 若$f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right), \varepsilon > 0$, 且对于某个常数 $c < 1$和充分大的n有$af\left(\frac{n}{b}\right) \leq cf(n)$, 那么

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

存在ε

存在ε

存在c和n_0

迭代

$$\boxed{T(n) = aT(n/b) + f(n)}$$

设$n = b^k$

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

迭代结果

$$\boldsymbol{a^{\log_b n} = n^{\log_b a}}$$

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

$$= \underline{a^k T(1)} + \sum_{j=0}^{k-1} a^j f\left(\frac{n}{b^j}\right)$$

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

- 第一项为所有子最小子问题的计算工作量
- 第二项为迭代过程归约到子问题及综合解的工作量

Case1

$$\boxed{f(n) = O(n^{\log_b a - \varepsilon})}$$

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

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

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

Case1(续)

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

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

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

$$=c_1 n^{\log_b a}+O\left(n^{\log_b a-\varepsilon}\frac{b^{\varepsilon\log_b n}-1}{\underline{b^{\varepsilon}-1}}\right)$$

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

Case2

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

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

$$= c_1 n^{\log_b a} + \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right)$$

$$= c_1 n^{\log_b a} + \Theta\left(n^{\log_b a} \sum_{j=0}^{\log_b n - 1} \frac{a^j}{a^j}\right)$$

$$= c_1 n^{\log_b a} + \Theta\left(n^{\log_b a} \log n\right) = \Theta\left(n^{\log_b a} \log n\right)$$

Case3

$$\boldsymbol{f(n) = \Omega(n^{\log_b a+\varepsilon})} \quad (1)$$
$$\boldsymbol{af(n/b) \le cf(n)} \quad (2)$$

$$T(n) = c_1\, n^{\log_b a} + \sum_{j=0}^{k-1} a^j f\left(\frac{b}{b^j}\right)$$
$$\le c_1\, n^{\log_b a} + \sum_{j=0}^{\log_b n-1} c^j f(n)$$

$$a^j f\left(\frac{n}{b^j}\right) \le a^{j-1} c f\left(\frac{n}{b^{j-1}}\right)$$
$$\le c a^{j-1} f\left(\frac{n}{b^{j-1}}\right) \le \cdots \le c^j f(n)$$

Case3（续）

$$T(n) \le c_1 n^{\log_b a} + \sum_{j=0}^{\log_b n-1} c^j f(n)$$

$$= c_1 n^{\log_b a} + f(n)\frac{c^{\log_b n} - 1}{c-1}$$
$$= c_1 n^{\log_b a} + \Theta\big(f(n)\big)$$
$$= \Theta\big(f(n)\big)$$