解递归方程

插入排序伪代码

```
InsertionSort (A, n)
for j←2 to n
    do key ← A[j]
    i ← j-1
    while i>0 and A[i]>key
        do A[i+1] ← A[i]
        i ← i-1
    A[i+1]=key
```

- "←"表示" 赋值 "(assignment).
- 忽略数据类型、变量的说明等与算法无关的部分.
- 允许使用自然语言表示一些相关步骤
- 伪代码突出了程序使用的算法.

插入排序分析

- 最坏情形：输入数据已按倒序排列

$$T(n)=\sum_{j=2}^{n}\Theta(j)=\Theta(n^2)$$

- 平均情形：输入数据为所有可能的排列

$$T(n)=\sum_{j=2}^{n}\Theta(j/2)=\Theta(n^2)$$

- 插入排序是否是最快的排序算法？
  - 当n很小时，结论肯定
  - 当n很大时，不一定

循环不变式的三个性质

循环不变式是证明算法的正确性的一个常用方法。

- **初始化**：循环的第一次迭代之前，它为真。
- **保持**：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。
- **终止**：在循环终止时，伴随循环终止的原因，不变式为我们提供一个有用的性质，该性质有助于证明算法的正确性。

INSERTION-SORT的正确性

- **初始化**：首先证明在第一次循环迭代之前循环不变式成立。循环开始前，子数组只有一个元素，并且是排好序的，这表明第一次循环迭代之前循环不变式成立
- **保持**：证明每次迭代保持循环不变式。非形式化的，for循环体的第3~6行将A[j-1],A[j-2],A[j-3]等向右移动一个位置，直到找到A[j]的适当位置。第7行将A[j]的知插入该位置。这时子数组A[1..j]由原来在A[1..j]中的元素组成，但已按序排列。
- **终止**：最后研究在循环终止时发生了什么。导致for循环终止的条件是j>A.length=n。因为每次循环迭代j增加1，那么必有j=n+1。在循环不变式的表述中将j用n+1代替，我们发现子数组A[1..n]由原来在A[1..n]中的元素组成，但已按序排列。因为A[1..n]就是整个数组，我们推断出整个数组已排序，因此算法正确。

最优二叉树(optimized binary tree)

for m ←2 to n
do {
    for i ←0 to n-m
        do{
            j ← i + m
            w(i, j) ←w(i, j-1) + P(i) + Q(j)
            c(i, j)←min$_{i<l\leq j}$ {c(i, l-1)+c(l, j)}+w(i, j) ← Θ(m)
            }
    }

Θ(m(n-m))

总时间复杂度

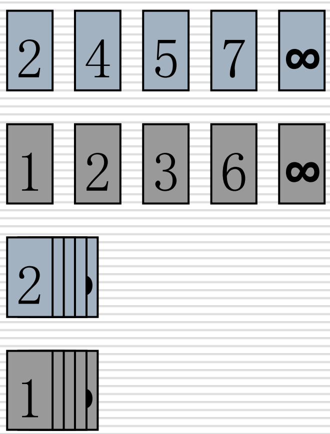
$$\Theta\left(\sum_{2\leq m\leq n} m(n-m)\right) = \Theta(n^3)$$

解递归方程的方法

- 递归树（Recursion tree）
- 代入法（Substitution method）
- Master方法（Master method）

MERGE L[1..p]和R[1..q]

- 合并两个已排序的序列<2, 4, 5, 7>和<1,2,3,6>.




```
L[p+1]=∞;R[q+1]=∞
i=1; j=1
for k=1 to p+q
    if L[i]≤R[j]
        A[k]=L[i]
        i=i+1
    else A[k]=R[j]
        j=j+1
```

<1,2,2,3,4,5,6,7>

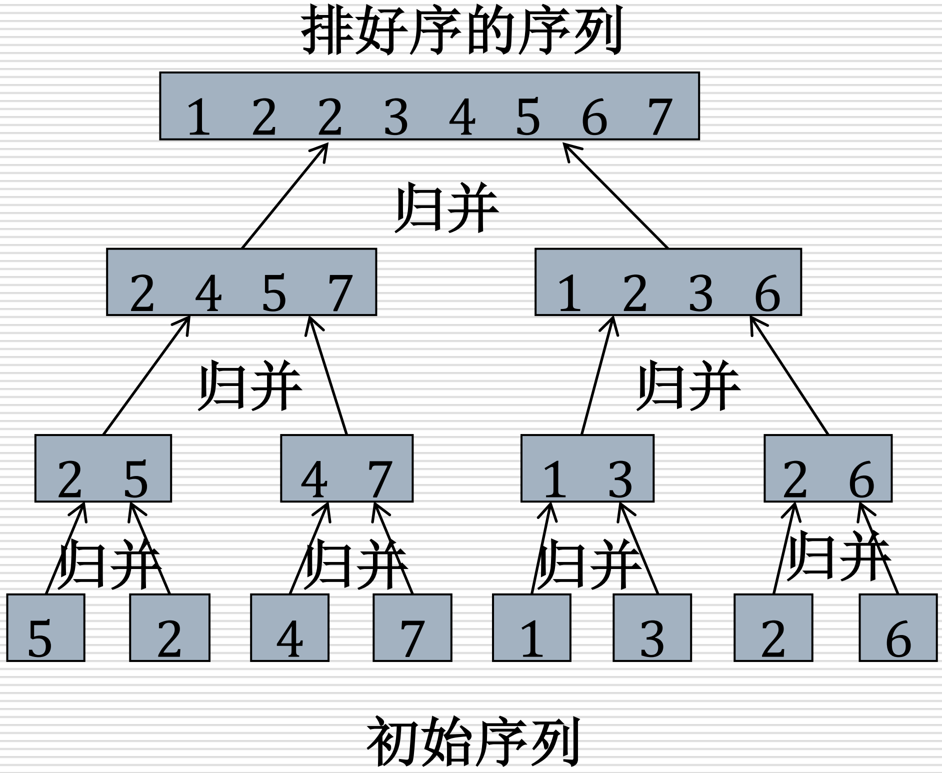
- $T(n)=\Theta(n)$ to merge a total of n elements (linear time)

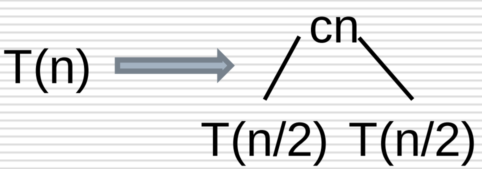
Merge算法的正确性

循环不变式：在开始第2~8行for循环的每次迭代时，子数组A[1..k-1]按从小到大的顺序包含L[1..p+1]和R[1..q+1]中的k-1个最小元素。进而，L[i]和R[j]是各自所在数组中未被复制回数组A的最小元素。

终止：终止时k=p+q+1. 根据循环不变式，子数组A[p..k-1]就是A[1..p+q]且按从小到大的顺序包含L[1..p+1]和R[1..q+1]中的k-1=p+q个最小元素。数组L和R一起包含p+q+2个元素。除两个最大的元素以外，其他所有元素都已被复制回数组A，这两个最大的元素就是哨兵。

MERGE-SORT(A, p, r)





MERGE-SORT A[1..n]

1. If n=1, done. $\Theta(1)$
2. $p=\lfloor n/2 \rfloor$ $\Theta(1)$
3. MERGE-SORT A[1..p] $T\lfloor n/2 \rfloor$
4. MERGE-SORT A[p+1..n] $T\lceil n/2 \rceil$
5. MERGE the 2 sorted lists. $\Theta(n)$

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

或

$$T(n)=\begin{cases} c & \text{if } n = 1 \\ T(n/2)+cn & \text{if } n > 1 \end{cases}$$

计算子数组的中间位置，需要常量时间，Θ(1)

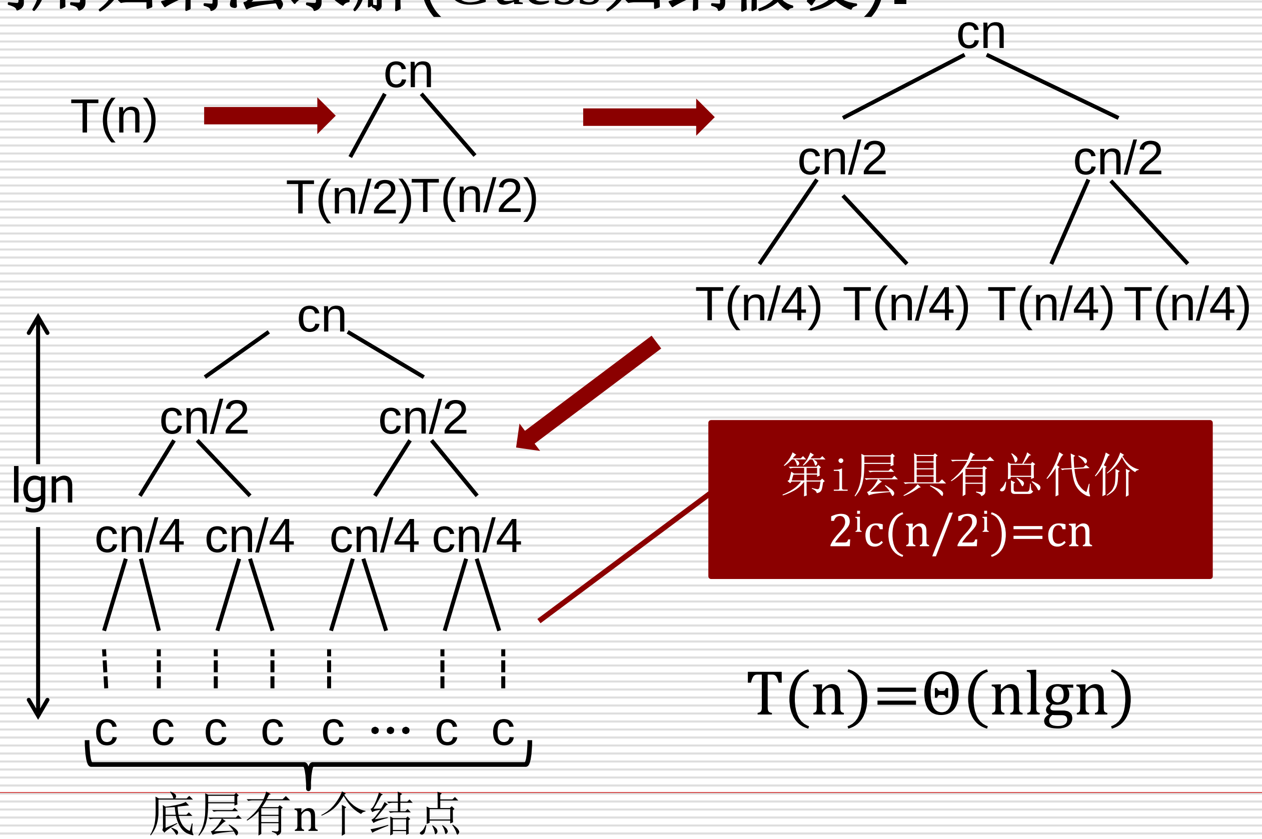
递归地求解两个规模均为n/2的子问题，将贡献2T(n/2)的运行时间

将子问题的结果进行合并。在一个具有n个元素的子数组上过程MERGE需要Θ(n)的时间

以cn代替Θ(n),不影响渐近分析的结果.

Recursion tree

- 很多递归式用递归树解不出来,但递归树能提供直觉,帮助我们用归纳法求解(Guess归纳假设).



Recurrence for merge sort

- “If n=1”,更一般的是 “if $n<n_0$, $T(n)=\Theta(1)$”. 指可找到足够大常数c_1, 使得$T(n)<c_1$ if $n<n_0$.
- 如果$n > 1$, 假定$n=2^h$, 则$T(n)=2T(n/2)+cn$.

$$
\begin{aligned}
T(n) &= 2T(n/2)+cn \\
&= 2(2T(n/2^2)+cn/2)+cn \\
&= 2^2T(n/2^2)+2cn \\
&= 2^3T(n/2^3)+3cn \\
&= 2^hT(n/2^h)+hcn \\
&= 2^hT(1)+cn\log n \quad (n=2^h) \\
&= \Theta(n\log n)
\end{aligned}
$$

当$n \neq 2^h$

当$n \neq 2^h$时，不妨设$2^h \leq n < 2^{h+1}$，则$n = \Theta(2^h)$，$h = \Theta(\log n)$。

因为时间复杂度函数是单调递增函数，有

$$T(2^h) \leq T(n) \leq T(2^{h+1}).$$

即

$$\Theta(2^h \log 2^h) \leq T(n) \leq \Theta(2^{h+1} \log 2^{h+1})。$$

因为

$$\Theta(2^{h+1} \log 2^{h+1}) = \Theta(2^h \log 2^{h+1}) = \Theta(2^h \log 2^h)$$

所以$T(2^h) = T(n) = T(2^{h+1})$.

Conclusions

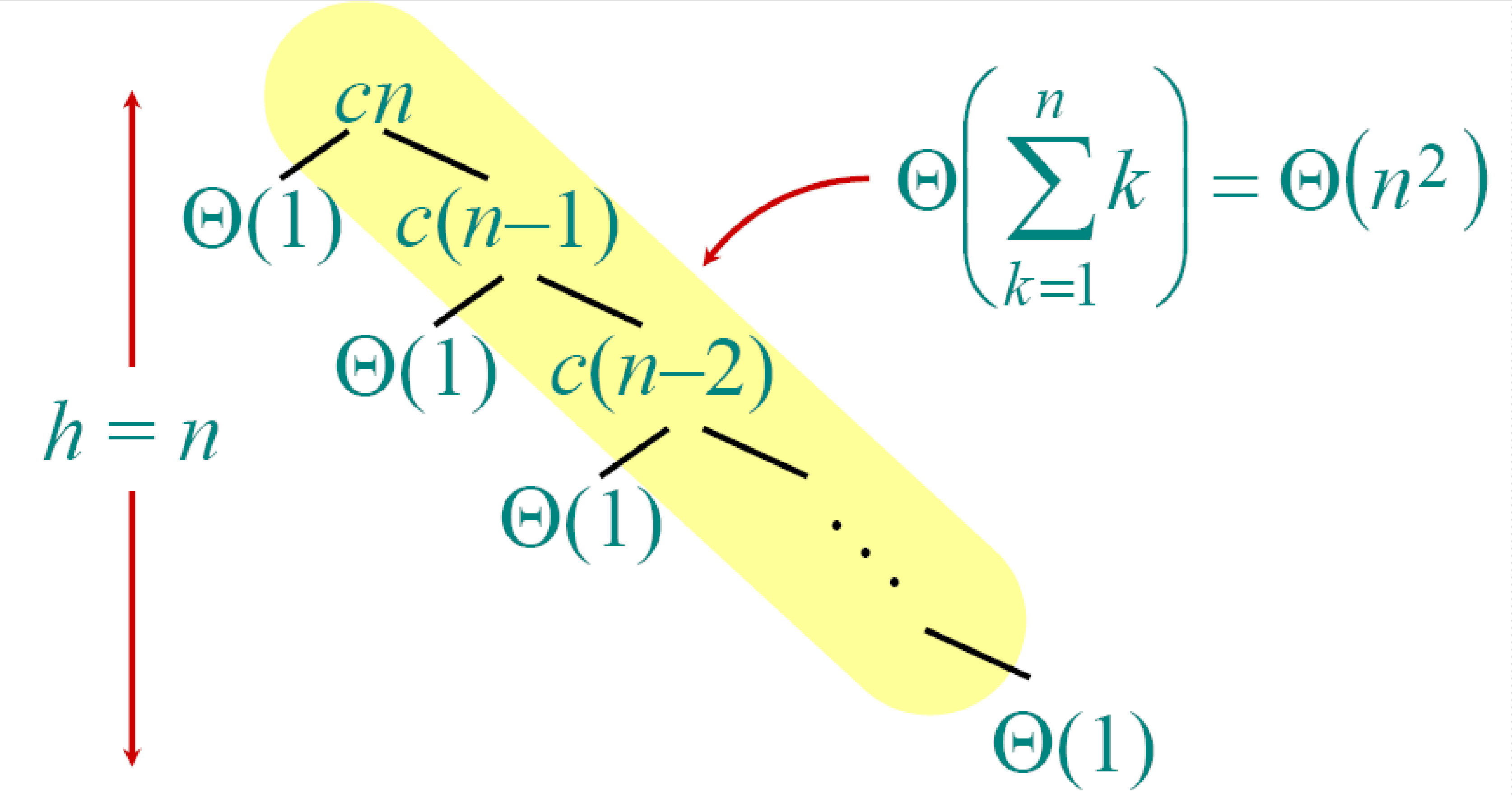
- MERGE-SORT过程的最坏情况运行时间递归式

  $T(n)=2T(n/2)+\Theta(n)=\Theta(n\lg n)$。
- 实际上, 归并排序优于插入排序$\Theta(n^2)$仅当$n > 30$ or so.
- 一个递归式(recurrence)就是一个等式或不等式，它通过更小的输入上的函数值来描述一个函数。
- 当子问题足够大，需要递归求解时，我们称之为递归情况(recursive case)。
- 当子问题变得足够小，不再需要递归时，我们说递归已经“触底”，进入了基本情况(base case)。

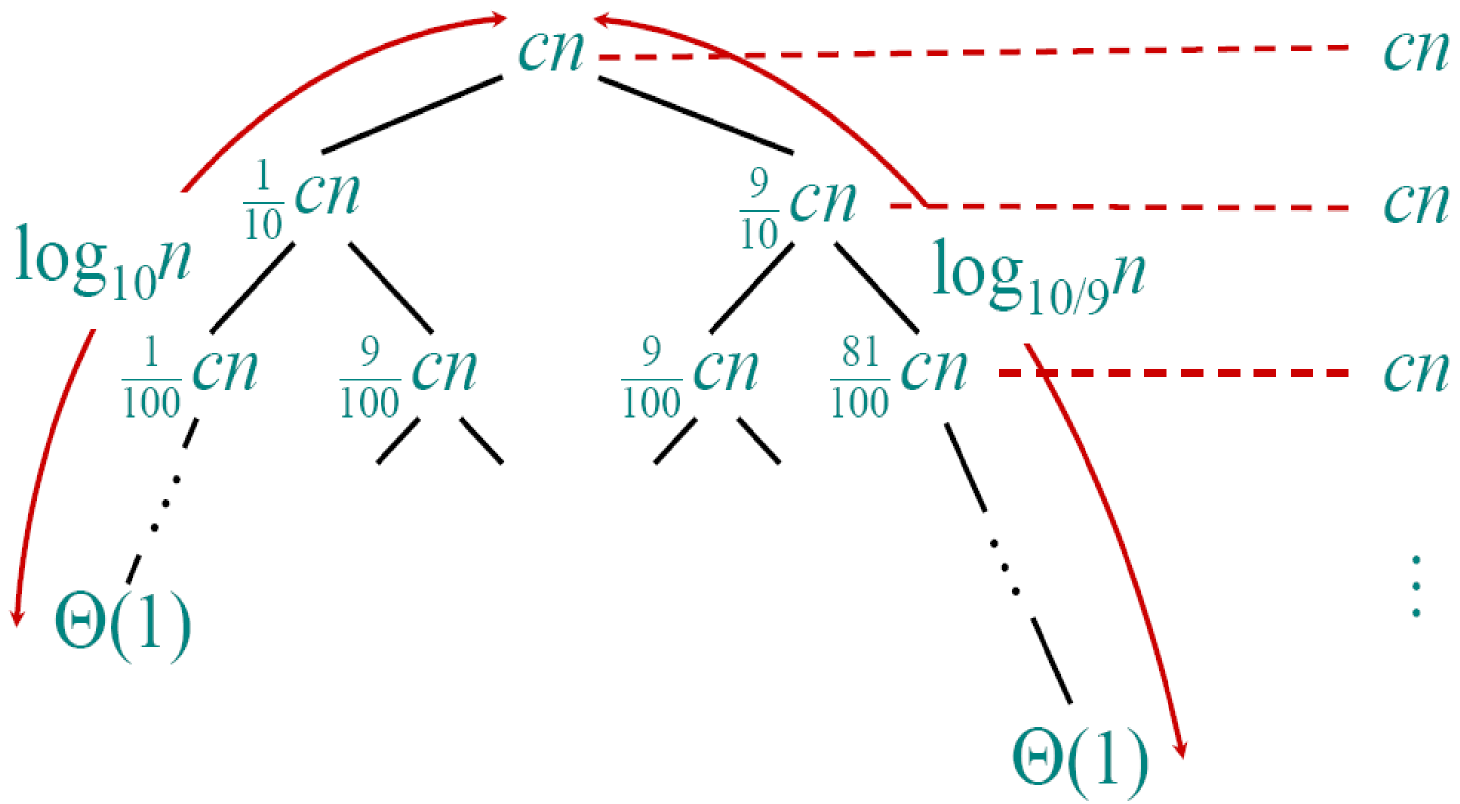
递归式可以有很多形式

- 有时，子问题划分的规模是不等的，
  - 比如$T(n)=T(2n/3)+T(n/3)+f(n)$，其中$f(n)$是分解和合并步骤的时间。
- 子问题的规模也不必是原问题规模的一个固定比例，
  - 比如LINEAR-SEARCH的递归式为$T(n)=T(n-1)+\Theta(1)$。

例1 展开递归树T(n)=T(1)+T(n−1)+cn,并做渐近分析.



例2 展开T(n)=T(0.1n)+T(0.9n)+Θ(n)的递归树并计算递归树的深度和T(n)的渐近值.



$$cn\log_{10}n \le T(n) \le cn\log_{10/9}n + O(n)$$

例3

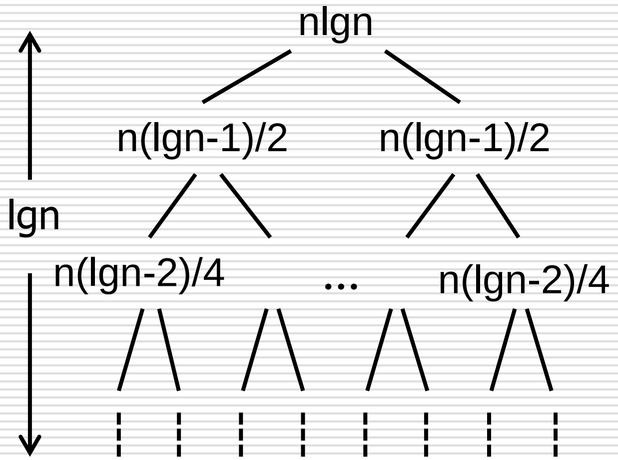
- $T(n)=2T(n/2)+n\lg n.$

  $=n\lg n+n(\lg n-1)+n(\lg n-2)+ \ldots + n(\lg n-(\lg n+1))$

  $=(n\lg n)\lg n-n(1+2+\ldots+\lg n-1)$

  $=n\lg^2 n-n\lg n(\lg n-1)/2$

  $=\Theta(n\lg^2 n)$



较一般的递归式

- 较一般的递归:$T(n)=aT(n/b)+cn$, a, b是大于1的整数,递归树方法仍可使用.
- 首先考虑$n=b^h$情形:

$$T(n)=a^hT(1)+cn(1+(a/b)+\cdots+(a/b)^{h-1})$$
$$= a^hT(1)+cb^h(1+(a/b)+\cdots+(a/b)^{h-1})$$

- 当$b^h \le n < b^{h+1}$, 仍有:

$$h=\Theta(\log_b n)$$

- 换底公式: $\log_b n=\log_2 n/\log_2 b \Rightarrow h=\Theta(\log n)$

例1 阶乘函数

- 通项公式为$n!=1\cdot 2\cdot 3\cdots(n-1)\cdot n$
- 阶乘函数可递归地定义为：

边界条件

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

递归方程

- 边界条件与递归方程是递归函数的两个要素。递归函数只有具备了这两个要素，才能在有限次计算后得出结果。
- $T(n)=\begin{cases} 0 & n = 0 \\ T(n-1)+1 & n > 0 \end{cases} \Rightarrow T(n) = \begin{cases} 0 & n = 0 \\ n & n > 0 \end{cases}$

```
Public static int factorial (int n)
{
    If (n==0) return 1;
    Return n*factorial (n-1);
}
```

例2 Fibonacci数列

- 无穷数列1，1，2，3，5，8，13，21，34，55，……，称为Fibonacci数列。第n个Fibonacci数可递归地定义为：

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

边界条件

递归方程

- 它可以递归地计算如下：

```
int fibonacci(int n)
  {
      if (n <= 1) return 1;
      return fibonacci(n-1)+fibonacci(n-2);
  }
```

例2 Fibonacci数列

- 令$t(n)$为算法执行的加法次数， 有$t(0)=0, t(1)=0, t(2)=1, t(3)=2, t(n)=t(n-1)+t(n-2)+1$.
- 因为$t(n-1)>t(n-2)$, 有$t(n)<2t(n-1)+1$, for $n>2$. 用归纳法易证$t(n)\le 2^n$.
- 又有$t(n)>2t(n-2)>\cdots>2^{k-1}t(2)=2^{k-1}$ if $n=2k$
  $>2^{k-1}t(3)=2^k$ if $n=2k+1$
- 算法有指数的时间复杂度$t(n)=O(2^n)$.
- 实际上这是因递归引起的大量的重复计算而非问题本身的难度所致. 可设计一非常简单的线性时间复杂度的迭代算法.

例2 Fibonacci数列

- $\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} F_{n-1} + F_{n-1} \\ F_{n-1} \end{bmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$
- $T(n)=\Theta(\lg n)$.
- Bottom-up: Compute F_0, F_1, F_2, ……, F_n in order, forming each number by summing the two previous. Running time is $\Theta(n)$.
- 通项公式$F(n)= \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$, $T(n)=\boldsymbol{\Theta(n)}$.

Golden ratio

例3　Ackerman函数

- 当一个函数及它的一个变量是由函数自身定义时，称这个函数是**双递归函数**。
- Ackerman函数A(n，m)定义如下：

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

- 找不到相应的非递归公式。

例3 Ackerman函数

- 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，$A(n,0)=n+2$
- m=1时，$A(1,1)=A(A(0,1),0)=A(1,0)= 2$;
- $A(n,1)=A(A(n-1,1),0)=A(n-1,1)+2\ (n>1)=2\cdot n\ (n\geq 1)$
- m=2时，$A(1,2)=A(A(0,2),1)=A(1,1)=2$，$A(n,2)=A(A(n-1,2),1)=2A(n-1,2)$，和故$A(n,2)= 2^n$ 。
- m=3时，类似的可以推出$A(n,3)=2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$，其中2的层数为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。

例4 排列问题

- 设计一个递归算法生成n个元素$\{r_1,r_2,\ldots,r_n\}$的全排列(n个元素的全排列共n！个）。
- 设$R=\{r_1,r_2,\ldots,r_n\}$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)=(r_1)perm(R_1), (r_2)perm(R_2), …, (r_n)perm(R_n) 。

例4 排列问题

```
Public static void perm(Object[] list, intk, int
m)
{//产生list[k:m]的所有排列
    if {k==m)
    {//只剩一个元素
        for (int i=0;i<=m; i++)
            System. out. print(list[i]);
            System. Out. Println();
    }
else
//还有多个元素，递归产生排列
    for (int i=k; i<=m; i++)
    {
        MyMath. Swap(list, k, i);
        perm (list, k+1, m);
        MyMath. Swap(list, k, i);
    }
}
```

例5　整数划分问题

- 所谓整数划分，是指把一个正整数n写成如下形式：$n=m_1+m_2+...+m_i$（其中m_i为正整数，并且$1\le m_i \le n$），则$\{m_1,m_2,...,m_i\}$为n的一个划分。
- 如果$\{m_1, m_2, ..., m_i\}$中的最大值不超过m，即$\max(m_1, m_2, ..., m_i)\le m$，则称它属于n的一个m划分。这里我们记n的m划分的个数为f(n,m)；
- 例如当n=4时，他有5个划分，{4}, {3,1}, {2,2}, {2,1,1}, {1,1,1,1}; 注意4=1+3和4=3+1被认为是同一个划分。
- 该问题是求出n的所有划分个数，即f(n, n)。

例5 整数划分问题

- 根据n和m的关系，考虑以下几种情况：
  - 当$n=1$时, 不论m的值为多少($m>0$), 只有一种划分即{1};
  - 当$m=1$时, 不论n的值为多少, 只有一种划分即n个1: {1, 1, 1, ..., 1};
  - 当$n=m$时, 根据划分中是否包含n, 可以分为两种情况:
    - 划分中包含n的情况, 只有一个即{n};
    - 划分中不包含n的情况, 这时划分中最大的数字也一定比n小, 即n的所有(n-1)划分. 因此$f(n,n) =1 + f(n,n-1)$.

例5 整数划分问题

- 当n<m时，由于划分中不可能出现负数，因此就相当于f(n,n);
- 但n>m时，根据划分中是否包含最大值m，可以分为两种情况：
  - 划分中包含m的情况，即{m, {x_1,x_2,...x_i}}, 其中{x_1,x_2,... x_i} 的和为n-m，因此这种情况下为f(n-m,m);
  - 划分中不包含m的情况，则划分中所有值都比m小，即n的(m-1)划分，个数为f(n,m-1) ; 因此f(n, m) = f(n-m, m)+ f(n,m-1).
- $$f(n,m)=\begin{cases}1 & n=1 \text{ or } m=1\\ f(n,n) & n<m\\ 1+f(n,n-1) & n=m\\ f(n,m-1)+f(n-m,m) & n>m>1\end{cases}$$

例5 整数划分问题

```
Public static int q(int n, int m)
{
        if ((n<1)||(m<1)) return 0;
        if ((n==1)||(m==1)) return 1;
        if (n<m) return q(n,n);
        if (n==m) return q(n, m-1)+1;
        return q(n,m-1)+q(n-m,m);
}
```

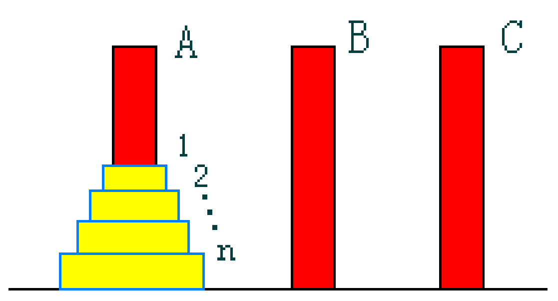
例6　Hanoi塔问题

- 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：

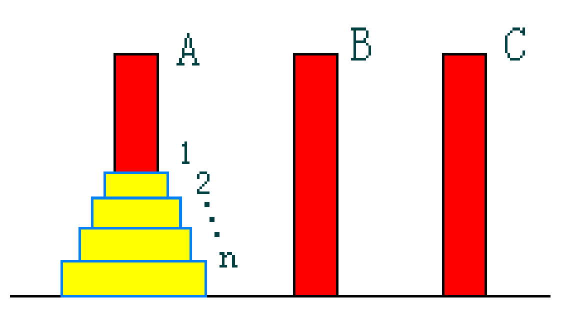
规则1：每次只能移动1个圆盘；

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。



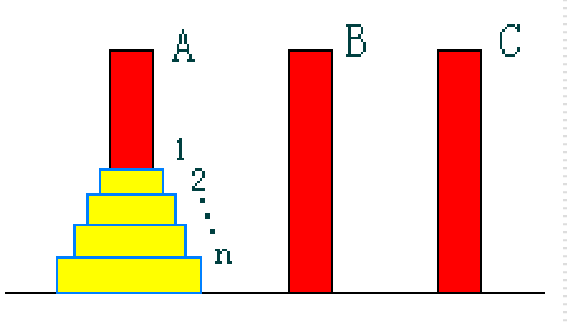
例6 Hanoi塔问题



- 在问题规模较大时，较难找到一般的方法，因此我们尝试用递归技术来解决这个问题。
- 当n=1时，问题比较简单。此时，只要将编号为1的圆盘从塔座a直接移至塔座b上即可。
- 当n>1时，需要利用塔座c作为辅助塔座。此时若能设法将n-1个较小的圆盘依照移动规则从塔座a移至塔座c，然后，将剩下的最大圆盘从塔座a移至塔座b，最后，再设法将n-1个较小的圆盘依照移动规则从塔座c移至塔座b。
- 由此可见，n个圆盘的移动问题可分为2次n-1个圆盘的移动问题，这又可以递归地用上述方法来做。由此可以设计出解Hanoi塔问题的递归算法如下。

例6 Hanoi塔问题

```
void hanoi(int n, int a, int b, int c)
  {
      if (n > 0)
      {
          hanoi(n-1, a, c, b);
          move(a,b);
          hanoi(n-1, c, b, a);
      }
  }
```



求解递归式的方法

- 代入法(Substitution method)：我们猜测一个界，然后用数学归纳法证明这个界是正确的。
- 递归树法(Recursion tree)：将递归式转换为一棵树，其结点表示不同层次的递归调用产生的代价。然后采用边界和技术来求解递归式。
- 主方法(Master method)：可求解形如$T(n)=aT(n/b)+f(n)$的递归式的界。其中$a\geq 1$，$b>1$，$f(n)$是一个给定的函数。这种递归式刻画了这样一个分治算法：生成a个子问题，每个子问题的规模是原问题规模的$1/b$，分解和合并步骤总共花费时间为$f(n)$。

代入法

The most general methods:

1. Guess the form of the solution.
2. Verify by induction.
3. Solve for constants.

Example1

- 以下面的时间复杂度函数为例：

$$T(n)=\begin{cases}\Theta(1) & n=1\\ 4T(n/2)+n & n>1\end{cases}, \text{其中} n=2^h\text{。}$$

- Guess $O(n^3)$:归纳假定为$T(n)\le cn^3$. c是待定常数.
- 应用假定有:$T(k) \le ck^3$ for $k < n$.
- 归纳证明: $T(n) \le cn^3$ 并确定常数c.

Example(continued)

- 初始条件为: $T(1)=\Theta(1)$。可取c足够大,使得$T(1)\le c$, 即对$n=1$, $T(n) \le cn^3$成立.
- 应用假定:$T(k) \le ck^3$ for $k < n$.
- 则 $T(n)=4T(n/2)+n \le 4c(n/2)^3+n$

  $$=(c/2)n^3+n=cn^3-((c/2)n^3-n)$$
- 取 $c \ge 2$ and $n \ge 1$, 则不等式$(c/2)n^3-n \ge 0$成立, 所以$T(n) \le cn^3$成立。
- $T(n) \le cn^3$这个界不是紧的!

例(续)

- We shall prove that $T(n)=O(n^2)$.
- Assume that $T(k)\le ck^2$ for $k<n$:
- $T(\mathrm{n})=4T(\mathrm{n}/2)+\mathrm{n}$

  $\le 4\mathrm{c}(\mathrm{n}/2)^2+\mathrm{n}$

  $=\mathrm{cn}^2+\mathrm{n}$
- 归纳不能往下进行！

代入法

- 有时你可能正确猜出了递归式解的渐进界，但莫名其妙地在归纳证明时失败了。问题常常出在归纳假设不够强，无法证出准确的界。
- 当遇到这种障碍时，如果修改猜测，将它减去一个低阶的项，数学证明常常能顺利进行。

Example(continued)

- **方法：**减去一个低阶的项。
- 假设 $T(k) \le c_1k^2 - c_2k$ for $k<n$.
- $T(n) = 4T(n/2)+n$

  $\le 4[c_1(n/2)^2 - c_2(n/2)]+n$

  $= c_1n^2 - c_2n - (c_2n - n)$

  $\le c_1n^2 - c_2n$ (取 $c_2>1$)
- For $1 \le n < n_0$, we have "$\Theta(1)$" $\le c_1n^2 - c_2n$, if we pick c_1 big enough.

代入法

- 考虑递归式：$T(n)=T(\lfloor n/2\rfloor)+T(\lceil n/2\rceil)+1$
- 我们猜测解为$T(n)=O(n)$, 并尝试证明对某个恰当选出的常数c, $T(n)\le cn$成立. 将我们的猜测代入递归式, 得到

$$T(n)\le c\lfloor n/2\rfloor+c\lceil n/2\rceil+1=cn+1$$

- 这并不意味着对任意c都有$T(n)\le cn$。
- 我们可能忍不住尝试猜测一个更大的界，比如$T(n)=O(n^2)$. 虽然从这个猜测也能推出结果，但原来的猜测$T(n)=O(n)$是正确的。如何证明它是正确的？

代入法

- 解决方法：从原来的猜测中减去一个低阶项！
- 新的猜测为$T(n)\le cn-d$, d是大于等于0的一个常数. 有

  $$T(n)\le c(\lfloor n/2\rfloor-d)+c(\lceil n/2\rceil-d)+1=cn-2d+1\le cn-d$$
- 只要$d\ge 1$，上式就成立。这样，只要选择足够大的c，就可以满足边界条件。

代入法

- 有时, 一个小的代数运算可以将一个未知的递归式变成你所熟悉的形式.
- 考虑递归式: $T(n)=2T(\lfloor\sqrt{n}\rfloor)+\lg n$.
- 为方便起见, 我们只考虑$\sqrt{n}$是整数的情形.
- 令$m=\lg n$, 得到$T(2^m)=2T(2^{m/2})+m$. 现在重命名$S(m)=T(2^m)$, 得到新的递归式$S(m)=2S(m/2)+m=O(m\lg m)$ (回忆MERGE-SORT的时间复杂度).
- $T(n)=T(2^m)=S(m)=O(m\lg m)=O(\lg n(\lg\lg n))n$.

The Master Method

- The master method用来解下述递归

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

式中$a\geq 1$, $b>1$为整数, $f(n)>0$.

- 按$f(n)$相对于$n^{\log_b a}$的渐近性质, 分三种情形进行分析.

The Master Method:情形1

- 情形1:
  - $f(n)=O(n^{\log_b a-\varepsilon})$, $\varepsilon>0$为某一常数,
  - $f(n)$的增长渐近地慢于$n^{\log_b a}$ (慢n^{ε}倍).
- Solution: $T(n)=\Theta(n^{\log_b a})$.

Examples

$T(n)=9T(n/3)+\Theta(n)$.

$a=9, b=3 \Rightarrow n^{\log_b a}=n^2$, $f(n)=cn$,

$f(n)=O(n^{\log_3 9-\varepsilon})$, 其中$\varepsilon=1$,

满足情况1,

$T(n)=\Theta(n^2)$。

$T(n)=4T(n/2)+n$

$a=4, b=2 \Rightarrow n^{\log_b a}=n^2$; $f(n)=n$.

$f(n)=O(n^{2-\varepsilon})$ for $\varepsilon=1$.

满足情形1: $T(n)=\Theta(n^2)$.

The Master Method:情形2

- 情形2:
  - $f(n)=\Theta(n^{\log_b a})$ $k\geq 0$ 为某一常数.
  - $f(n)$和$n^{\log_b a}$几乎有相同的渐近增长率.
- Solution: $T(n)=\Theta(n^{\log_b a}\lg n)$.
- 情形2的一般形式:
  - $f(n)=\Theta(n^{\log_b a}\lg^s n)$，$s\geq 0$ 为某一常数.
  - $f(n)$和$n^{\log_b a}$几乎有相同的渐近增长率.
- Solution: $T(n)=\Theta(n^{\log_b a}\lg^{s+1} n)$.

Examples

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

$a=2, b=2 \Rightarrow n^{\log_b a}=n,\ f(n)=cn$,

满足情况2,

$T(n)=\Theta(n\lg n)$。

$T(n)=4T(n/2)+n^2\lg n$

$a=4, b=2 \Rightarrow n^{\log a}=n^2; f(n)=n^2\lg n$.

满足情形 2: $f(n)=\Theta(n^2\lg n)$, $k = 1$.

$T(n)=\Theta(n^2\lg n)$.

The Master Method:情形3

- 情形3
  - $f(n)=\Omega(n^{\log a+\varepsilon})$，$\varepsilon>0$为一常数.
  - f(n)多项式地快于$n^{\log_b a}$ (by an n^{ε} factor),
- f(n) 满足以下规则性条件:

  $af(n/b)\le cf(n)$, $0<c<1$ 为常数.
- Solution：$T(n) = \Theta(f(n))$.

Examples

$T(n)=3T(n/4)+n\lg n.$

$a=3, b=4, f(n)=n\lg n, n^{\log_b a}= n^{\log_4 3}<n^{0.793}.$

$f(n)=n\lg n>n>n^{\log_4 3+0.2}$, 其中$\varepsilon=0.2$.

当n足够大时， 对于$c=3/4$,

$af(n/b)= 3(n/4)\lg(n/4)\le(3/4)n\lg n=cf(n)$,

正则条件成立,

满足情况3,

递归式的解为$T(n)=\Theta(n\lg n)$.

Examples

$T(n)=4T(n/2)+n^3$

$a=4$, $b=2 \Rightarrow n^{\log_b a}=n^2$; $f(n)=n^3$.

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon=1$

and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c=1/2$.

$\therefore$ $T(n)=\Theta(n^3)$.

The Master Method

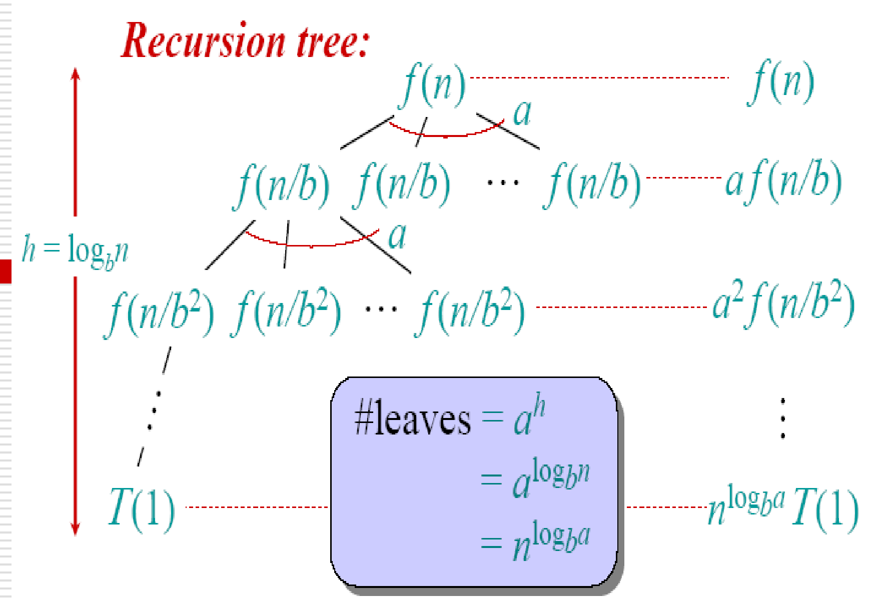
主定理　令$a\geq 1$和$b>1$是常数，$f(n)$是一个函数，

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

是定义在非负整数上的递归式，其中n/b为$\lfloor n/b \rfloor$或$\lceil n/b \rceil$。那么$T(n)$有如下渐进界：

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

主方法



- 从递归树可知:

$$\begin{aligned} T(n) &= a^h T(1) + \textstyle\sum_{k=0}^{h-1} a^k f(b^{h-k}) \\ &= a^h T(1) + \textstyle\sum_{k=0}^{h-1} a^k f(b^{h-k}) \\ &= n^{\log_b a} T(1) + \textstyle\sum_{k=0}^{h-1} a^k f(b^{h-k}) \end{aligned}$$

- $T(n)$的渐近性质由$n^{\log_b a}$和$\sum_{k=0}^{h-1} a^k f(b^{h-k})$的渐近性质决定: 两者中阶较高的决定!
- 这就是为什么要比较$f(n)$和$n^{\log_b a}$ 的原因!

引理

- Let a, b, and c be nonnegative constants. The solution to the recurrence

$$T(n)=\begin{cases} c & \text{for } n = 1 \\ aT(n/b) + cn & \text{for } n > 1 \end{cases}$$

for n a power of b is

$$T(n)=\begin{cases} O(n) & \text{if } a < b \\ O(n\log n) & \text{if } a = b \\ O\left(n^{\log_b a}\right) & \text{if } a > b \end{cases}$$

引理证明

- 经过迭代得到

$$
\begin{aligned}
T(n) &= aT(n/b)+cn \\
&= a[aT(n/b^2)+cn/b]+cn \\
&= a^2T(n/b^2)+acn/b+cn \\
&= \cdots\cdots \\
&= a^kT(n/b^k)+a^{k-1}cn/b^{k-1}+\cdots+acn/b+cn \\
&= a^kT(1)+\sum_{j=0}^{k-1} a^j cn/bj
\end{aligned}
$$

因为$n=b^k$, $k=\log_b n$, 所以$a^k=a^{\log_b n}=n^{\log_b a}$(换底公式). 所以

$$T(n)=cn^{\log_b a}+cn\cdot\sum_{j=0}^{\log_b n-1}(aj/bj)$$

引理证明

$$T(n)=cn^{\log_b a}+ cn \cdot \sum_{j=0}^{\log_b n-1}(a/b)^j$$

- If a<b, then $\sum_{i=0}^{\infty}(a/b)^i$ converges, so T(n) is O(n);
- If a=b, then each term in the sum is unity, and there are O(logn) terms. Thus T(n) is O(nlogn).
- Finally, if a>b then

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

Which is $O(a^{\log_b n})$ or equivalently $O(n^{\log_b a})$.

主定理的证明

- 不妨设$n=b^k$, $T(1)=c_1$, 经过迭代得到

$$
\begin{aligned}
T(n)&=aT(n/b)+f(n)\\
&=a[aT(n/b^2)+f(n/b)]+f(n)\\
&=a^2T(n/b^2)+af(n/b)+f(n)\\
&=\cdots\cdots\\
&=a^kT(n/b^k)+a^{k-1}f(n/b^{k-1})+\cdots+af(n/b)+f(n)\\
&=a^kT(1)+\sum_{j=0}^{k-1} a^j f(n/bj)\\
&=c_1 n^{\log_b a}+\sum_{j=0}^{k-1} a^j f(b^{k-j})
\end{aligned}
$$

(注：由于$n=b^k$，$k=\log_b n$，所以$a^k=a^{\log_b n}=n^{\log_b a}$（换底公式）. ）

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

第一种情况,显然$T(n)=\Omega(n^{\log_b a})$. 由于$f(n)=O(n^{\log_b a-\varepsilon})$,

$$f\left(\frac{n}{b^j}\right)=O\left(\left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon}\right)=O\left(n^{\log_b a-\varepsilon}\left(\underbrace{\frac{1}{b^{\log_b a-\varepsilon}}}_{b^{\varepsilon}/a}\right)^j\right)$$

又由于$k=\log_b n$，所以

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

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

所以，$T(n)=c_1 n^{\log_b a}+O\left(n^{\log_b a-\varepsilon}\cdot n^{\varepsilon}\right)=O(n^{\log_b a})$.

$$\sum_{j=0}^{\log_b n-1}(b^{\varepsilon})^j=\frac{b^{\varepsilon\log_b n}-1}{b^{\varepsilon}-1}=\frac{n^{\varepsilon}-1}{b^{\varepsilon}-1}=O(n^{\varepsilon})$$

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

第二种情况，当$f(n)=\Theta(n^{\log_b a})$时，有

$$f\left(\frac{n}{b^j}\right)=\Theta\left(\left(\frac{n}{b^j}\right)^{\log_b a}\right)=\Theta\left(n^{\log_b a}\left(\frac{1}{b^{\log_b a}}\right)^j\right)=\Theta\left(n^{\log_b a}\frac{1}{a^j}\right)$$

代入求和公式中，有

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

所以

$$T(n)=c_1 n^{\log_b a}+\Theta\Big(n^{\log_b a}\underbrace{\log_b n}_{k=\log_b n}\Big)=\Theta\left(n^{\log_b a}\log_b n\right)$$

第二种情况(common case)

- $T(n)=a^kT(1)+\sum_{j=0}^{k-1} a^j f(n/b^j)$
- 由$f(n)=\Theta(n^{\log_b a}\lg^s n)$, $n=b^k$, $n^{\log_b a}=a^k$得

$$f(b^k) \le c \cdot a^k(\lg b^k)^s = c \cdot a^k \cdot k^s(\lg b)^s$$

同理, $f(b^i) \le c \cdot a^i \cdot i^s(\lg b)^s$. 又由于$f(n/b^j)=f(b^{k-j})$,

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

所以 $T(n) \le a^kT(1)+c \cdot a^k \cdot \sum_{i=1}^{k} i^s (\lg b)^s$

$\le a^kT(1)+c \cdot a^k \cdot \underline{k^{s+1}} \cdot (\lg b)^s$

$1^s+\ldots+k^s=\Theta(k^{s+1})$

$=a^kT(1)+c \cdot a^k \cdot k^{s+1} \cdot (\lg b)^{s+1}/\lg b$

$=c_1 n^{\log_b a}+c_2 \cdot n^{\log_b a} \cdot \lg^{s+1} n$

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

第三种情况，当$f(n)=\Omega(n^{\log_b a+\varepsilon})$, 且对某个常数$c<1$和所有足够大的$n$有$af(n/b)\le cf(n)$时, 则$T(n)=\Theta(f(n))$.

由$af(n/b)\le cf(n)$有

$$f(n)\ge \frac{a}{c} f\left(\frac{n}{b}\right)\ge \frac{a^2}{c^2} f\left(\frac{n}{b^2}\right)\ge\cdots\ge \frac{a^j}{c^j} f\left(\frac{n}{b^j}\right),$$

所以

$c<1$, $c^{\log_b n}\to 0$

$$\sum_{j=0}^{k-1} a^j f(n/b^j)\le \sum_{j=0}^{k-1} c^j f(n)=f(n)\frac{c^{\log_b n}-1}{c-1}=\Theta(f(n))$$

又因为$f(n)=\Omega(n^{\log_b a+\varepsilon})$，所以

$$T(n)\le c_1 n^{\log_b a}+\Theta(f(n))=\Theta(f(n))$$

显然，$T(n)\ge f(n)$，于是$T(n)=\Theta(f(n))$。

主定理的使用方法

f(n)必须渐进小于$n^{\log_b a}$. 要相差一个因子n^{ε}，其中ε是大于0的常数。

主定理中的三种情况的每一种，都是将函数f(n)与函数$n^{\log_b a}$进行比较，两个函数较大者决定了递归式的解。

- 若f(n)多项式意义上小于$n^{\log_b a}$，如情况1，则解为$T(n)=\Theta(n^{\log_b a})$。
- 若f(n)多项式意义上大于$n^{\log_b a}$，如情况3，并且满足”正则条件”$af(n/b)\le cf(n)$，则解为$T(n)=\Theta(f(n))$。
- 若两个函数大小相当，如情况2，则乘上一个对数因子，解为$T(n)=\Theta(n^{\log_b a}\lg n)=\Theta(f(n)\lg n)$。

主定理的使用方法

主定理的三种情况并未覆盖f(n)的所有可能性。

情况1和情况2之间有一定空隙：f(n)可能小于$n^{\log_b a}$但不是多项式意义上的小于。

情况2和情况3之间也有一定空隙： f(n)可能大于$n^{\log_b a}$但不是多项式意义上的大于。

如果函数f(n)落在这两个空隙中，或者情况3的正则条件不成立，就不能使用主方法来求解递归式。

Homework(2)

- 1.用归纳法证明

$$T(n) \le \begin{cases} c & \text{if } n = 1 \\ T\lfloor n/2 \rfloor + T\lceil n/2 \rceil + cn & \text{if } n > 1 \end{cases}$$

$$\le cnlogn$$

- 2.应用master方法求解$T(n)=2T(n/2)+\Theta(n^{1/2})$
- 3.展开递归树:$T(n)=T(2)+T(n-2)+cn$,并做渐近分析
- 展开$T(n)=T(0.2n)+T(0.8n)+\Theta(n)$的递归树并计算递归树的深度和$T(n)$的渐近值.
- 14章练习33-(a),(b),(c),(d),(e),(f),(g),(h),(i),(j)