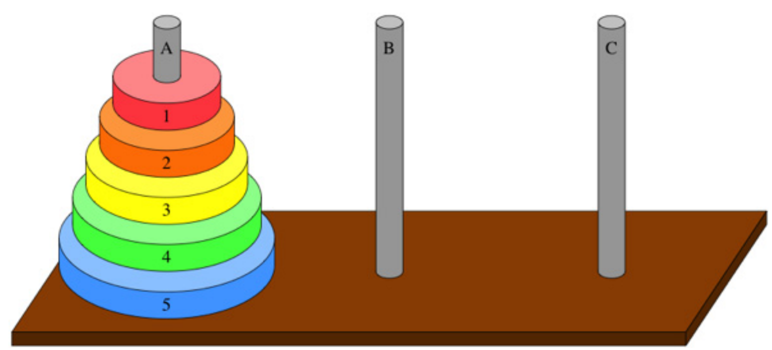
算法分析与设计

Analysis and Design of Algorithm

Lesson 05

要点回顾

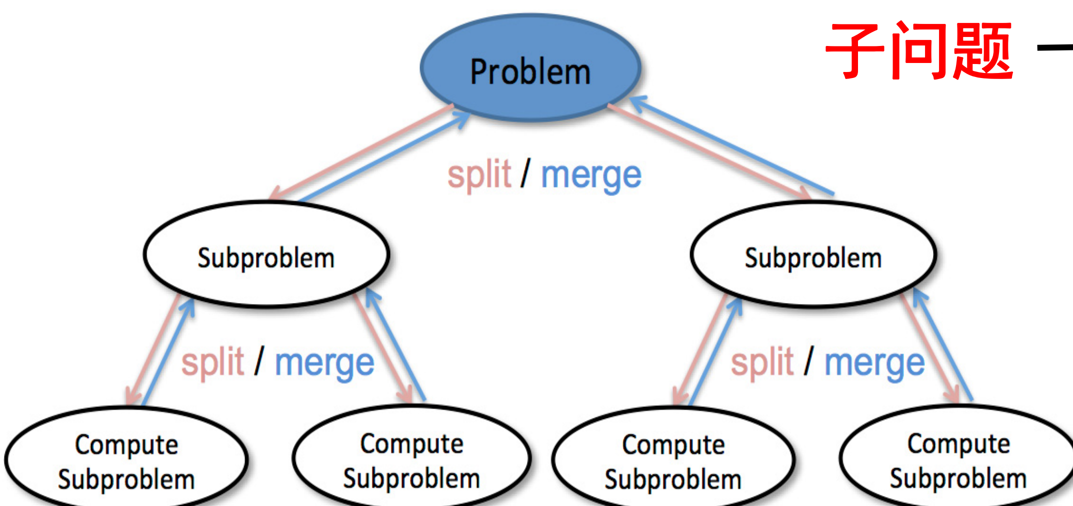
- 递归算法
  - 概念（阶乘、Fibonacci数列、双递归）
  - 例子（整数划分问题、Hanoi塔问题）
  - Hanoi塔算法、运行轨迹、分析时间复杂度
  - 递推方程（迭代法求解）
  - 递归的优缺点

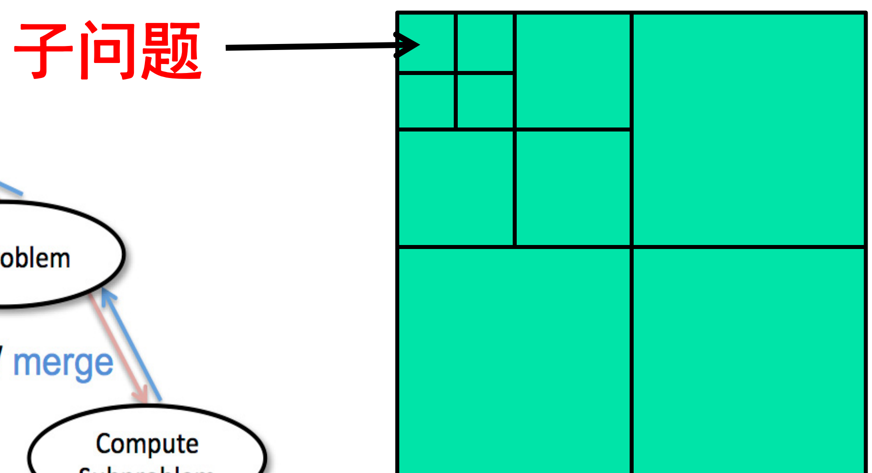


分治策略

分治法(Divide-and-Conquer)

基本思想：将一个规模为n的问题分解为k个规模较小的子问题，这些子问题**互相独立**且与原问题**相同**。递归解这些子问题，再将子问题合并得到原问题的解。





分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

- 该问题的规模缩小到一定的程度就可以容易地解决；

因为问题的计算复杂性一般是随着问题规模的增加而增加，因此大部分问题满足这个特征。

分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

- 该问题的规模缩小到一定的程度就可以容易地解决；
- 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**

这条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用。

分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

- 该问题的规模缩小到一定的程度就可以容易地解决；
- 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**
- **利用该问题分解出的子问题的解可以合并为该问题的解；**

能否利用分治法完全取决于问题是否具有这条特征，如果具备了前两条特征，而不具备第三条特征，则可以考虑**贪心算法**或**动态规划**。

分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

- 该问题的规模缩小到一定的程度就可以容易地解决；
- 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**
- 利用该问题分解出的子问题的解可以合并为该问题的解；
- 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问题。

该特征涉及到分治法效率，如果各子问题不独立，则分治法要做许多不必要的工作，重复地解公共子问题，此时虽也可用分治法，但一般用**动态规划**较好。

分治法的基本步骤

伪代码：

```
divide-and-conquer(P){
    if ( |P| <= n0) adhoc(P);     //(1)解决小规模的问题
    divide P into smaller sub instances P1,P2,...,Pk;
    //(2)分解问题
    for (i=1,i<=k,i++)
         yi=divide-and-conquer(Pi);   //(3)递归地解各子问题
    return merge(y1,...,yk);//(4)将各子问题的解合并为原问题的解
  }
```

- 其中，| P |是问题P的规模，n_0为一阈值，表示当问题P的规模不超过n_0时，问题已容易解决，不必再继续分解。
- adhoc(P)是分治的基本子算法，用于直接解小规模的问题P。
- merge(y_1, y_2, ... , y_k)是分治算法中的合并子算法。

分治法的问题

- 将问题分为多少个子问题？
- 子问题的规模是否相同/怎样才恰当？

- 研究者从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。即，将一个问题分成大小相等的k个子问题的处理方法是行之有效的。
- 这种使子问题规模大致相等的做法是出自一种叫做平衡(balancing)子问题的思想，它几乎总是比子问题规模不等的做法要好。

分治法的计算效率

采用分治法解规模为n的问题的效率分析，假定：

- 分成k个规模为n/m的子问题去解。
- 解规模为1的问题耗费1个单位时间。
- 分解为k个子问题和将k个子问题的解合并需用$f(n)$个单位时间。

➔用$T(n)$表示该分治法解规模为$|P|=n$的问题所需的计算时间：

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

迭代法求方程解：

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

从游戏开始。。。

二分搜索技术

问题：给定已按升序排好序的n个元素$a[0:n-1]$，现要在这n个元素中找出一特定元素x。

分析：

1. 该问题的规模缩小到一定的程度就可以容易地解决；

如果$n=1$即只有一个元素，则只要比较这个元素和x就可以确定x是否在表中。因此这个问题满足分治法的第一个适用条件

二分搜索技术

问题：给定已按升序排好序的n个元素$a[0:n-1]$，现要在这n个元素中找出一特定元素x。

分析：

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；
3. 分解出的子问题的解可以合并为原问题的解；

比较x和中间元素$a[mid]$，若$x=a[mid]$，则x在L中的位置就是mid；如果$x<a[mid]$，由于a是递增排序的，因此假如x在a中的话，x必然排在$a[mid]$的前面，所以只要在$a[mid]$的前面查找x即可；如果$x>a[i]$，同理只要在$a[mid]$的后面查找x即可。无论是在前面还是后面查找x，其方法都和在a中查找x一样，只不过是查找的规模缩小了。这就说明了此问题满足分治法的第二、三个适用条件

二分搜索技术

问题： 给定已按升序排好序的n个元素$a[0:n-1]$，现要在这n个元素中找出一特定元素x。

分析：

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；
3. 分解出的子问题的解可以合并为原问题的解；
4. 分解出的各个子问题是相互独立的。

很显然此问题分解出的子问题相互独立，即在$a[i]$的前面或后面查找x是独立的子问题，因此满足分治法的第四个适用条件。

二分搜索算法

据此容易设计出二分搜索算法：

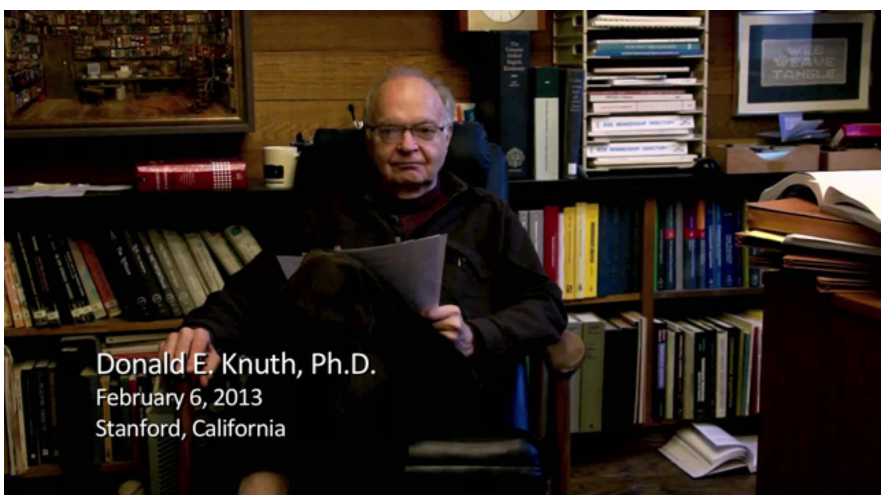
```
template<class Type>
int BinarySearch(Type a[], const Type& x, int l, int r){
     while (r >= l){
        int m = (l+r)/2;
        if (x == a[m])
             return m;
        if (x < a[m])
             r = m-1;
        else l = m+1;
        }
    return -1;
}
```

算法复杂度分析：

每执行一次算法中的while循环，待搜索数组的大小减少一半。因此，在最坏情况下，while循环被执行了$O(\log n)$次。循环体内运算需要$O(1)$时间，因此整个算法在最坏情况下的计算时间复杂性为$O(\log n)$。

二分搜索算法(cont.)

- 二分搜索算法易于理解。
- 编写正确的二分搜索算法不易，90%人在2个小时内不能写出完全正确的二分算法。



第一个二分搜索算法在1946年提出，但是第一个完全正确的二分搜索算法却直到1962年才出现。

大整数的乘法

- 在复杂性计算时，都将加法和乘法运算当作基本运算处理，即加、乘法时间为常数。但上述假定仅在参加运算整数能在计算机表示范围内直接处理时才合理。
- 那么我们处理很大的整数时，怎么办？
- 要精确地表示大整数，并在计算结果中精确到所有位数，就必须用软件方法实现。

大整数的乘法(cont.)

问题： 请设计一个有效的算法，可以进行两个n位大整数的乘法运算

◆小学的方法：$O(n^2)$ ✖ 效率太低了！！！

	n	$n\log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n=10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n=30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n=50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n=100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n=1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n=10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n=100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n=1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

大整数的乘法(cont.)

问题： 请设计一个有效的算法，可以进行两个n位大整数的乘法运算

◆小学的方法：$O(n^2)$　　✖ 效率太低了！！！

◆分治法：

假设：X和Y都是n位二进制整数

	n/2位	n/2位		n/2位	n/2位
X=	a	b	Y=	c	d

$$X = a\,2^{n/2} + b \qquad Y = c\,2^{n/2} + d$$

➔

$$XY = ac\,2^{n} + (ad+bc)\,2^{n/2} + bd$$

复杂度分析

则计算X，Y，需4次n/2位整数乘法(ac，ad，bc和bd)

需3次小于2n位的整数加
需2次移位(2^n, $2^{n/2}$)

共需O(n)

$$T(n)=\begin{cases} O(1) & n=1 \\ 4T(n/2)+O(n) & n>1 \end{cases}$$

直接用之前的迭代法解有点点复杂！

换元迭代法

- 将对n的递推式换成对其他变元k的递推式
- 对k直接迭代
- 将解(关于k的函数)转换成关于n的函数

举例：解如下递推方程

$$T(n)=2T(n/2)+n-1$$
$$T(1)=0$$

换元：令$n=2^k$，则递推方程变换为

$$T(2^k)=2T(2^k/2)+2^k-1=2T(2^{k-1})+2^k-1$$
$$T(0)=0$$

换元迭代法

$T(2^k)=2T(2^{k-1})+2^k-1$

$T(0)=0$

迭代求解：

$T(n)=T(2^k)=2T(2^{k-1})+2^k-1$

$=2[2T(2^{k-2})+2^{k-1}-1]+2^k-1$

$=2^2T(2^{k-2}) +2^k-2+2^k-1$

$=\ldots$

$=2^kT(1)+k2^k-(2^{k-1}+2^{k-2}+\ldots+2+1)$

$= k2^k-2^k+1$

$=n\log n-n+1$

最后，需用数学归纳法证明正确性！

复杂度分析

则计算X，Y，需4次n/2位整数乘法(ac，ad，bc和bd)

需3次小于2n位的整数加

需2次移位(2^n, $2^{n/2}$)

共需O(n)

$$T(n)=\begin{cases} O(1) & n=1 \\ 4T(n/2)+O(n) & n>1 \end{cases}$$

为了简化，设n为2的幂

换元迭代法$n=2^x$

$$T(1)=O(1)$$

$$T(2)=4O(1)+O(2) \rightarrow 4O(1)$$

$$T(4)=4(4O(1)+O(2))+O(4) \rightarrow 4^2O(1)$$

$$T(8)=4(4(4O(1)+O(2))+O(4))+O(8) \rightarrow 4^3O(1)$$

复杂度分析

$$T(16)=\cdots \rightarrow 4^4 O(1)$$

$$\cdots$$

$$T(2^x)=\cdots \rightarrow 4^x O(1)$$

➔ $T(n)=O(n^2)$

公式法求解该算法复杂度：

$$T(n)=\begin{cases} O(1) & n=1 \\ 4T(n/2)+O(n) & n>1 \end{cases}$$

➔ $T(n)=O(n^2)$ ✖ 没有改进！！！

大整数的乘法(cont.)

为了降低时间复杂度，必须减少乘法的次数！！！

1. $XY = ac\ 2^n + ((a-b)(d-c)+ac+bd)\ 2^{n/2} + bd$

2. $XY = ac\ 2^n + ((a+b)(d+c)-ac-bd)\ 2^{n/2} + bd$

对于1式：需3次n/2位整数乘法((a-b)(d-c),ac,bd)
需6次加、减
需2次移位

大整数的乘法(cont.)

复杂度分析：

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

➔ $T(n)=O(n^{\log 3})=O(n^{1.59})$ ✓改进了！！！

细节问题：两个XY的复杂度都是$O(n^{\log 3})$，但考虑到***a*+*b***, ***c*+*d*** 可能得到***n*+1**位的结果，使问题的规模变大，故不选择第2种方案。

大整数的乘法(cont.)

- 还有没有更快的方法？ $T(n)=O(n^{\log 3})=O(n^{1.59})$
- 如果将大整数分成更多段，用更复杂的方式把它们组合起来，将有可能得到更优的算法。
- 最终的，这个思想导致了快速傅利叶变换(Fast Fourier Transform)的产生。该方法也可以看作是一个复杂的分治算法。

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$
1966	Toom-Cook	$\Theta(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$\Theta(n \log n \log \log n)$
2007	Fürer	$n \log n \, 2^{O(\log^* n)}$

是否能找到线性时间算法? 目前为止还没有结果。

递归树法验证换元迭代法

- 递归树的概念
  - 递归树是迭代计算的模型（迭代的图形表示）
  - 递归树的生成过程与迭代过程一致
  - 树上所有项恰好是迭代之后产生和式中的项
  - 对递归树上的项求和就是迭代后方程的解

递归树法验证换元迭代法

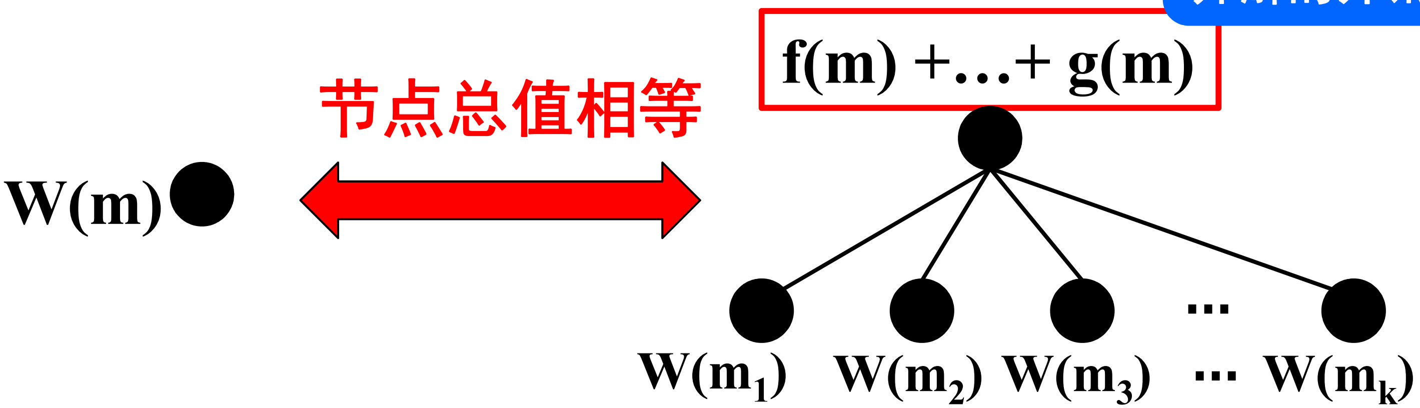
- 迭代在递归树中的表示
  - 如果递归树上某节点标记为W(m)
  - $W(m) = \boxed{W(m_1) + \ldots + W(m_k)}$

    $+ \boxed{f(m) + \ldots + g(m)}, \; m_1, \ldots, m_k < m$

递归求解划分后的子问题的开销

归约到子问题及合并的开销

其中$W(m_1), \ldots, W(m_k)$称为函数项

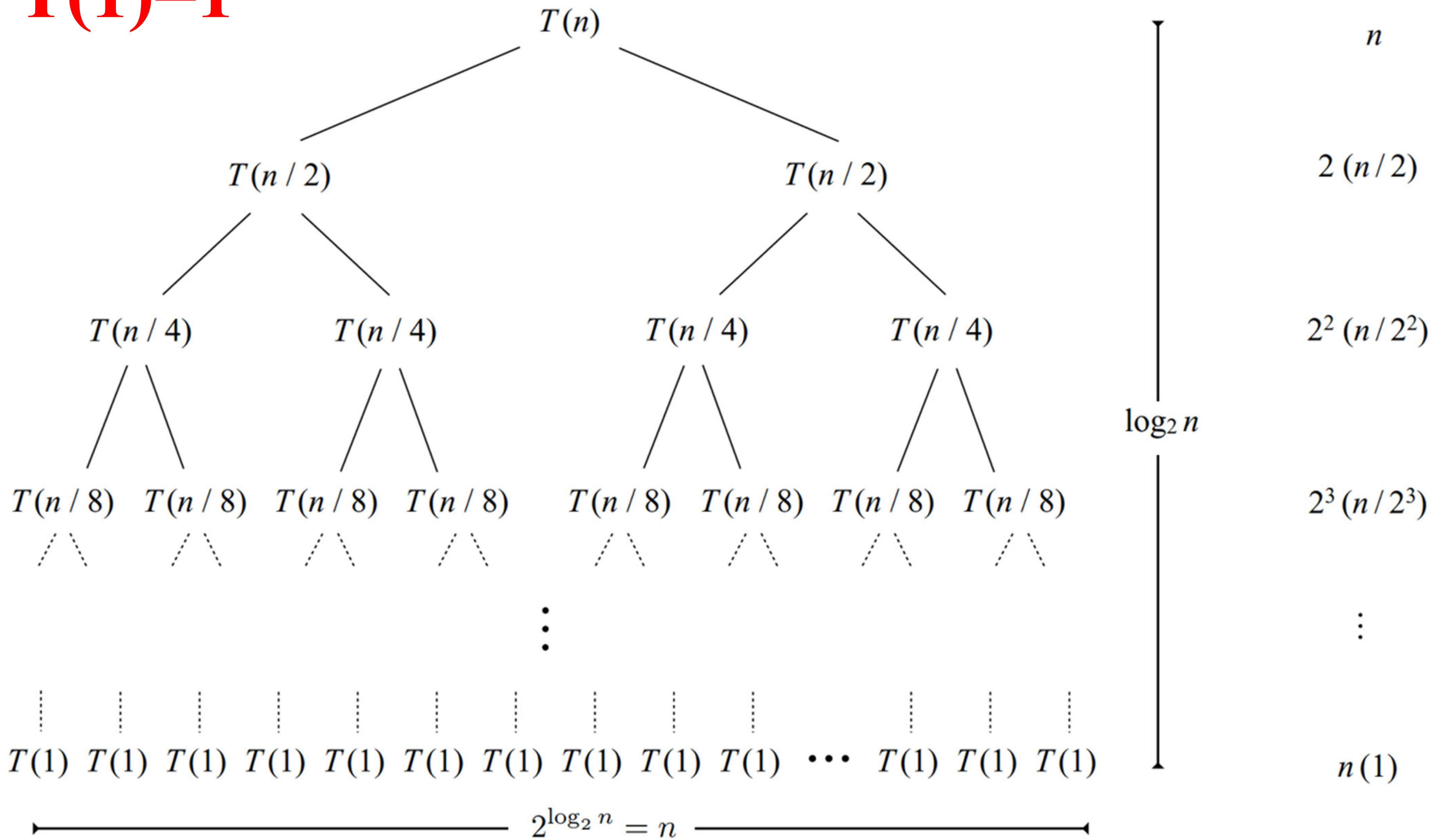


每个叶节点是一个函数项

例子： 画出$T(n)=2T(n/2)+n$递归树

$T(1)=1$

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



$r = 1$ $$T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n = n\,(\log_2 n + 1)$$