算法分析与设计

Analysis and Design of Algorithm

第4次课

课堂练习

- $\sqrt{n}-1=\Theta(\sqrt{n})$
  - 证明：令$f(n)=\sqrt{n}-1$, $g(n)=\sqrt{n}$,
  - 先证明$f(n)=O(g(n))$，显然，当$c\geq 1$时，对于任意$n>0$均有 $g(n)>f(n)$;
  - 再证明$f(n)=\Omega(g(n))$，显然，当$c<1$时，对于任意$n>\frac{1}{(1-c)^2}$均有 $g(n)<f(n)$
  - 所以命题成立。

例子：素数测试

算法PrimalityTest(n)

输入： n,大于2的奇整数

输出： true 或者false

1. $s \leftarrow \lfloor n^{1/2} \rfloor$
2. **for** $j \leftarrow 2$ **to** s
3.   **if** j 整除 n
4.   **then return false**
5. **return true**

问题：

若$n^{1/2}$ 可在$O(1)$计算，基本运算是整除,以下表示是否正确？

$W(n)=O(n^{1/2})$

$W(n)=\Theta(n^{1/2})$

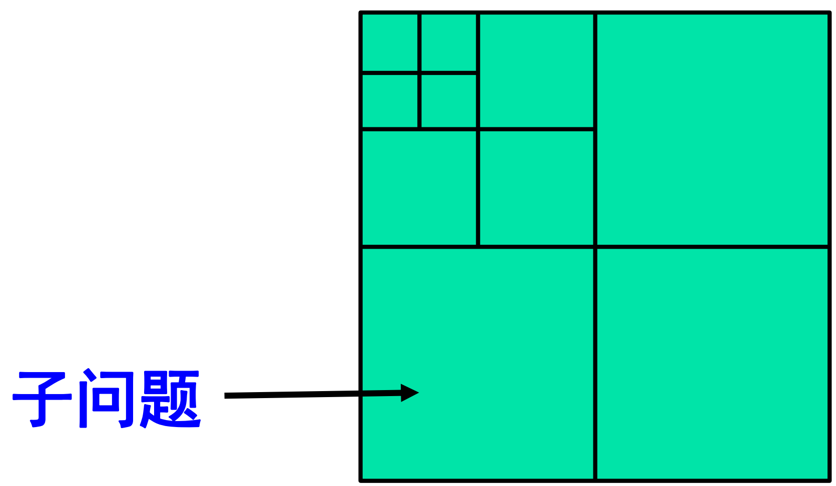
课程回顾

- 理解递归的概念
- 几个递归算法的例子
  - Fibonacci数列
  - 双递归函数
  - 整数划分
  - 汉诺塔
- 递推公式

分治策略

分治法

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。



中国MOOC：3.2 分治策略的设计思想

3.3 分治策略的一般描述和分析方法

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**。

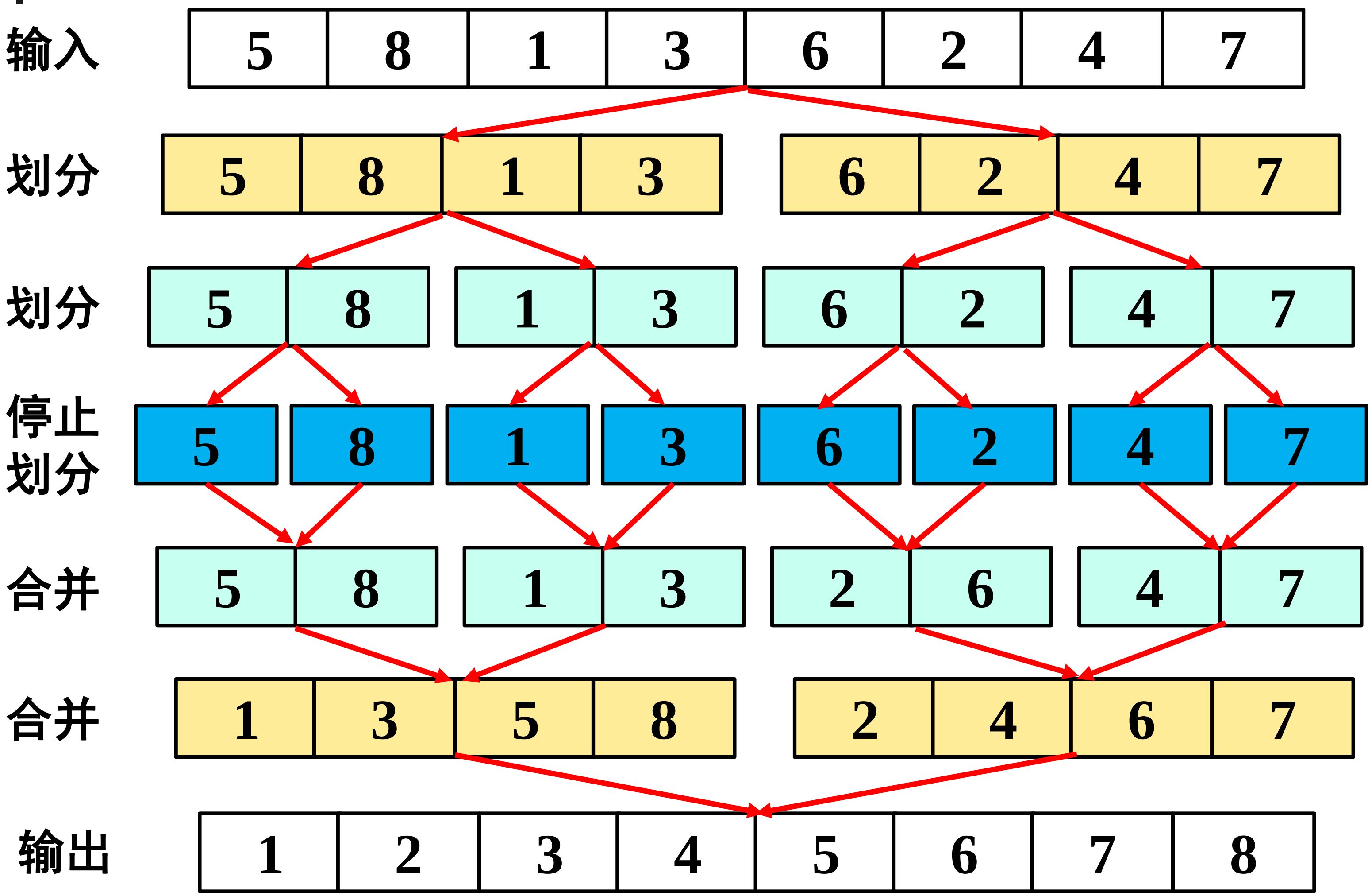
分治法的适用条件

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。

这条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然也可用分治法，但一般用**动态规划**较好。

回顾：归并排序


输入 5 8 1 3 6 2 4 7
划分 5 8 1 3 | 6 2 4 7
划分 5 8 | 1 3 | 6 2 | 4 7
停止划分 5 | 8 | 1 | 3 | 6 | 2 | 4 | 7
合并 5 8 | 1 3 | 2 6 | 4 7
合并 1 3 5 8 | 2 4 6 7
输出 1 2 3 4 5 6 7 8

分治法的基本步骤

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

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?

- 人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。
- 这种使子问题规模大致相等的做法是出自一种**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

分治法解规模为*n*的问题的效率：

- 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：

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

可以用换元迭代法求解

几个经典的分治算法

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
- 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。

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；**

- 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
- 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
- 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**:

```
int binarySearch(int[] a, int start, int end, int target) {
    if(start > end)
        return -1;
    int mid = start + (end - start) / 2;
    if(a[mid] == target)
        return mid;
    else if(a[mid] > target)
        return binarySearch(a, start, mid-1, target);
    else
        return binarySearch(a, mid+1, end, target);
}
```

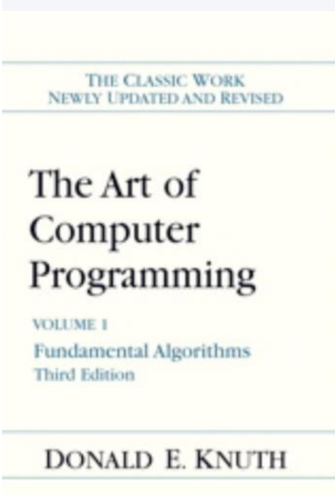
算法复杂度分析:

每执行一次算法的递归，待搜索数组的大小减少一半。因此，在最坏情况下，递归被执行了$O(\log n)$ 次。函数体内除去递归的运算需要$O(1)$ 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。

二分搜索算法(cont.)

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
             low = mid + 1
         else if (midVal > key)
             high = mid - 1;
         else
             return mid; // key found
     }
     return -(low + 1);  // key not found.
 }
```

分治法算法的时间复杂度

中国MOOC：02-04 迭代法求解递推方程
02-05 差消法求
02-06 递归树
02-07 主定理及其证明（选学）
02-08 主定理的应用（选学）

基本方法

- **递推方程的求解（以大整数的乘法为例）**

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

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

- **求解方法**
  - **迭代法**
  - **递归树**
  - **公式法**

迭代法

迭代法算出该算法的复杂度：

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

$$\begin{aligned} T(n) &= 4T(n/2)+O(n) \\ &= 4(4T(n/4)+O(n/2))+O(n) \\ &= 4^{\log_2 n}+O(n)+2O(n)+\cdots+2^{\log_2 n\text{-}1}O(n) \\ &= O(n^2) \end{aligned}$$

太抽象啦！有没有更形象的方法呢？

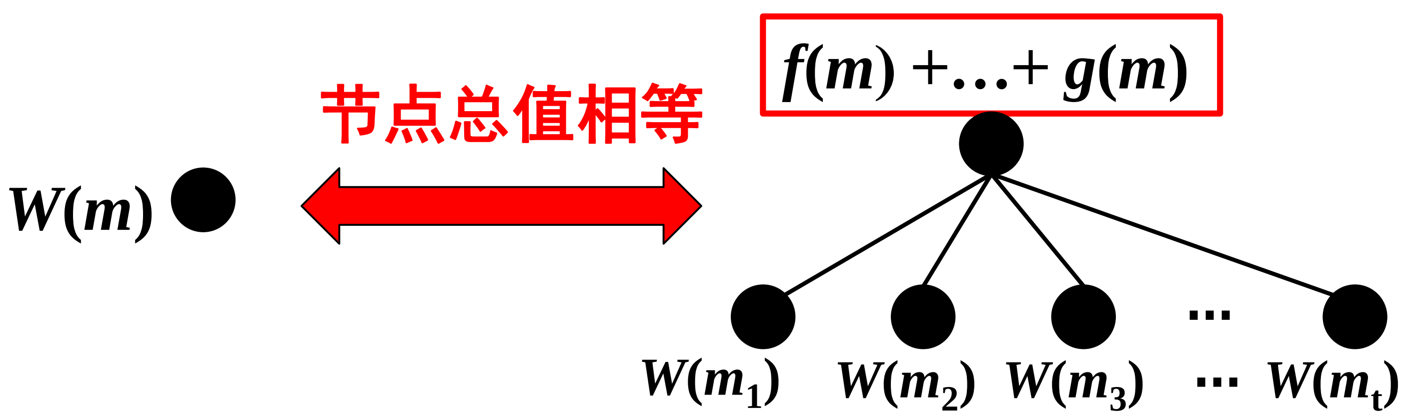
递归树验证迭代法

- 递归树的概念
  - 递归树是迭代计算的模型（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

递归树验证迭代法

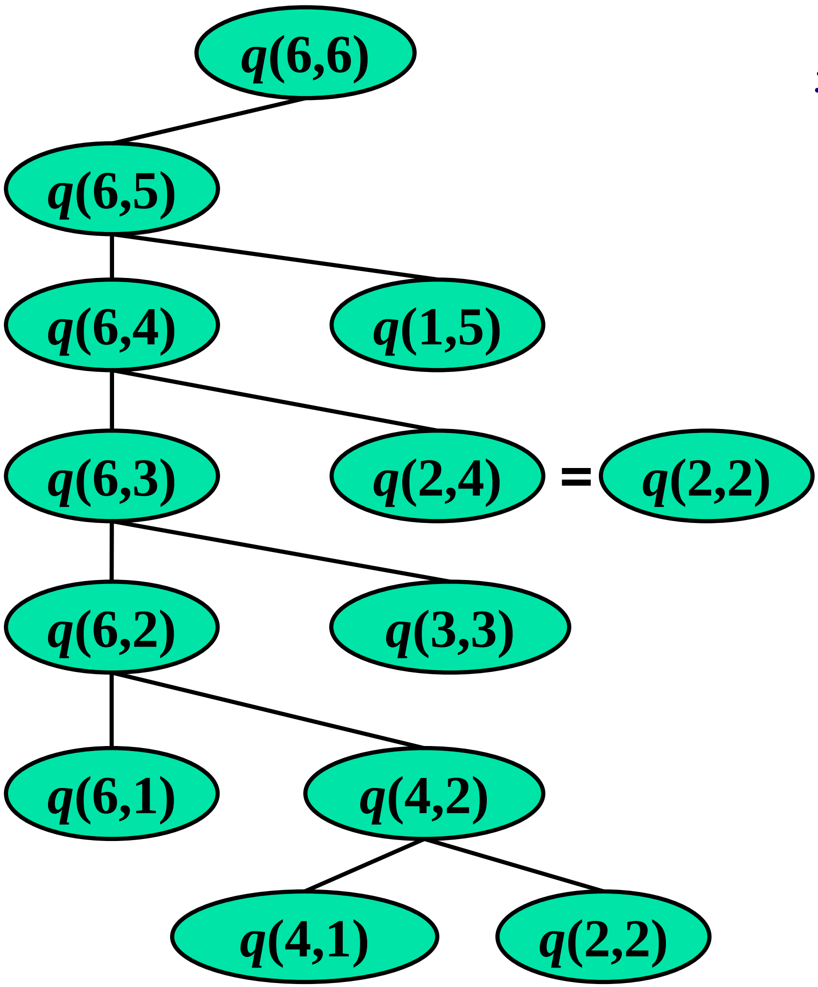
- 迭代在递归树中的表示
  - 如果递归树上某节点标记为$W(m)$
  - $W(m)= W(m_1) +\ldots+ W(m_t)$
    $+ \boxed{f(m) +\ldots+ g(m)}, m_1,\ldots, m_t<m$

  其中$W(m_1) ,\ldots,W(m_t)$称为函数项



每个叶节点是一个函数项

回顾：整数划分问题

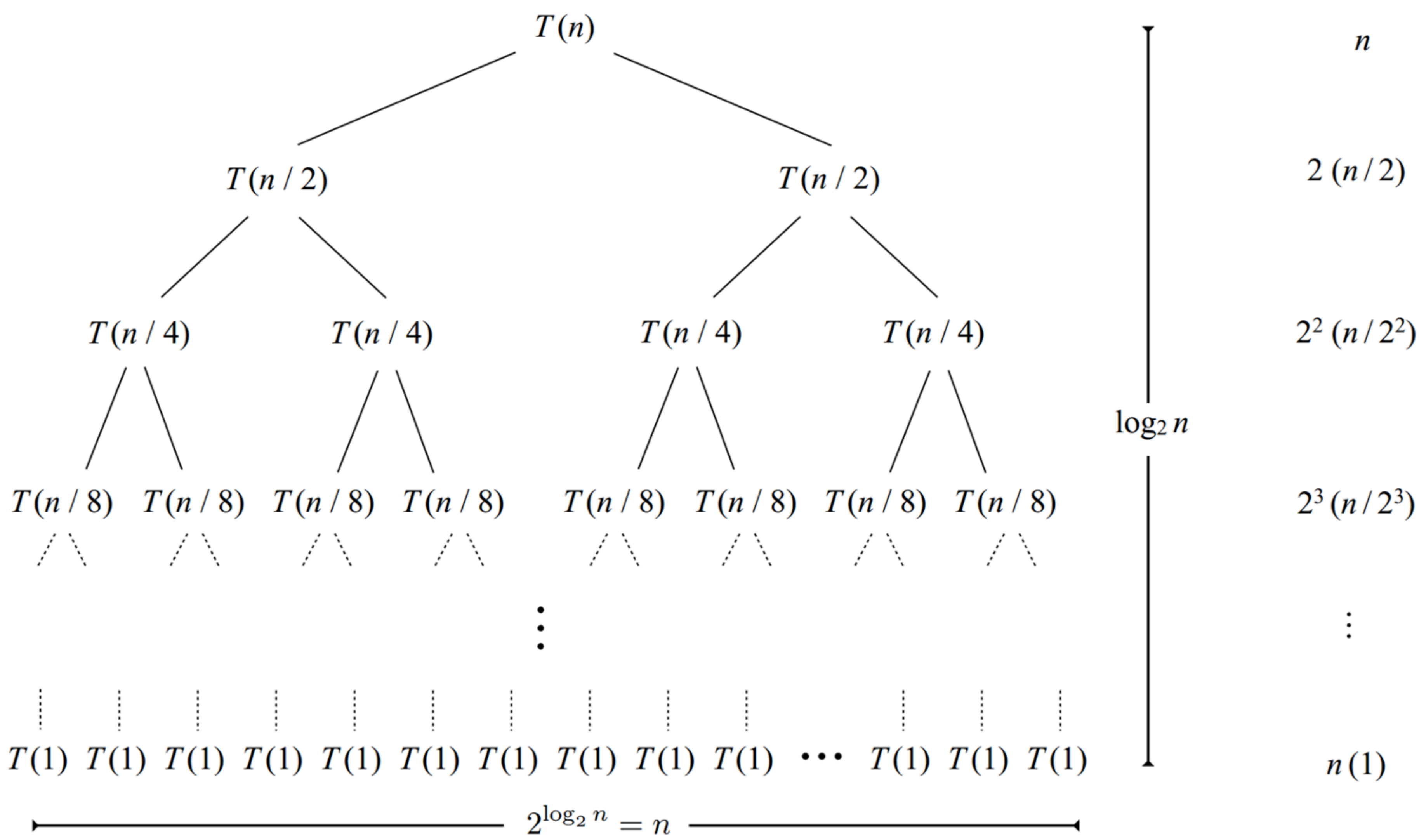



```
int q(int n, int m){
    if ((n<1)||(m<1))
       return 0;
    if ((n==1)||(m==1))
       return 1;
    if (n<m)
       return q(n, n);
    if (n==m)
       return q(n, m-1)+1;
    return q(n, m-1)+q(n-m, m);
}
```

例子：画出归并排序的递归树

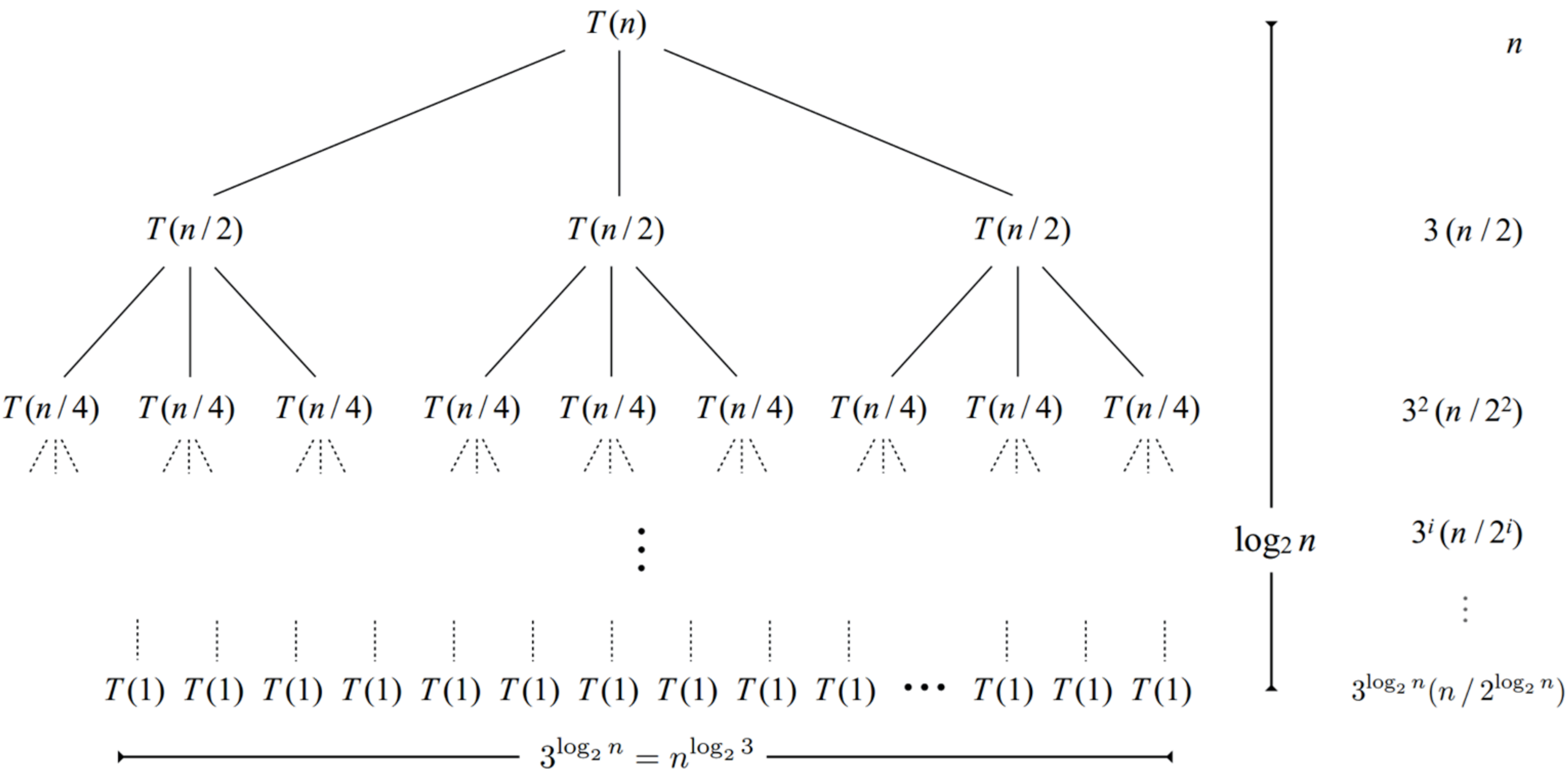
$$T(n)=\begin{cases} O(1) & n=1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

$T(n)=2T(n/2)+n$ $\quad$ $k=2,\ m=2$ $\quad$ $r=k/m$



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

递归树求解：$T(n)=3T(n/2)+O(n)$



$$r = 3/2 > 1 \qquad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n = \frac{r^{1+\log_2 n} - 1}{r - 1}\, n = 3n^{\log_2 3} - 2n$$

公式法

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

方程的解：

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

回到公式法

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

方程的解：

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

- 第一项为所有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合并解的工作量

哪一项更主要？

主定理(Master定理)

定理： 设$a \geqslant 1$，$b > 1$为常数，$f(n)$为函数，$T(n)$为非负数，且$T(n)=kT(n/m)+f(n)$，则：

1. 若 $f(n)=O(n^{(\log_m k)-\varepsilon})$，存在$\varepsilon>0$ 是常数，则有$T(n)=\Theta(n^{\log_m k})$
2. 若 $f(n)=\Theta(n^{\log_m k})$,则$T(n)=\Theta(n^{\log_m k}\log n)$
3. 若 $f(n)=\Omega(n^{(\log_m k)+\varepsilon})$，存在$\varepsilon>0$ 是常数，且对所有充分大的 n 有 $kf(\frac{n}{m})\le cf(n)$，$c<1$ 是常数，则有$T(n)=\Theta(f(n))$

Master定理

- **例子1：**求解 $T(n)=9T\left(\frac{n}{3}\right)+n$

$k=9, m=3, f(n)=n, n^{\log_m k}=n^2$

$\because f(n)=n< O\left(n^{(\log_m k)}\right)=n^2$ 取$\varepsilon=1$即可

$\therefore T(n)=\Theta\left(n^{\log_m k}\right)=\Theta\left(n^2\right)$

Master定理

- **例子2：** 求解$T(n) = T\left(\frac{2n}{3}\right) + 1$

$$k = 1, m = \frac{3}{2}, f(n) = 1, n^{\log_m k} = n^{\log_{3/2} 1} = 1$$

$$\because f(n) = 1 = \Theta(n^{\log_m k}),$$

$$\therefore T(n) = \Theta\left(n^{\log_m k} \log n\right) = \Theta(\log n)$$

Master定理

- **例子3：** 求解$T(n) = 3T\left(\frac{n}{4}\right) + n\log n$

$k = 3, m = 4, f(n) = n\log n$， $n^{\log_m k} = n^{\log_4 3} \approx n^{0.793}$

$f(n) = n\log n = \Omega(n^{\log_4 3+\varepsilon}) \approx \Omega(n^{0.793+\varepsilon})$ 取ε =0.2即可

Master定理

- 例子3：求解$T(n) = 3T\left(\frac{n}{4}\right) + n\log n$

  条件验证 $kf(\frac{n}{m}) \le cf(n)$

  $k = 3, m = 4, f(n) = n\log n$，代入上式

  $3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \le c \times n\log n$ 只要c≥3/4即可

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

递归算法分析

- **二分检索：** $T(n)=T(n/2)+1, T(1)=1$

$$\boldsymbol{k=1, m=2}, \qquad \boldsymbol{n^{\log_2 1}=1}, \qquad \boldsymbol{f(n)=1}$$

属于第二种情况

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

不能使用主定理的例子

- **例如：求解** $T(n) = 2T(n/2) + n\log n$

$$m = k = 2, n^{\log_2 2} = n, f(n) = n\log n$$

不存在$\varepsilon > 0$使右式成立 $n\log n = \Omega(n^{1+\varepsilon})$

不存在$c < 1$使$af(\frac{n}{b}) \le cf(n)$对所有充分大的$n$成立

$$2(n/2)\log(n/2) = n(\log n - 1) \le cn\log n$$

可以考虑递归树或公式法！！！