算法设计与分析

Computer Algorithm Design & Analysis

赵峰
zhaof@hust.edu.cn

Chapter 4
Divide-and-Conquer

分治策略

本章研究算法的设计技术

- 在插入排序算法里，我们学到了一种方法：增量式方法。
  - 在排序子数组A[1..j-1]后，将下一个新元素A[j]插入子数组的适当位置，从而产生排序好的新子数组A[1..j]。
- 本章学习另一种算法设计方法：“分治法”（Divide and Conquer）。

分治法的基本思想：

将原问题分解为几个规模较小、但类似于原问题的子问题，递归地求解这些子问题，然后再合并这些子问题的解以建立原问题的解。

分治法遵循三个基本步骤 ：

1）分解（Divide）：将原问题分为若干个规模较小、相互独立，形式与原问题一样的子问题；

2）解决（Conquer）：若子问题规模较小、可直接求解时则直接解；否则"递归"地求解各个子问题，即继续将较大子问题递归地分解为更小的子问题，然后重复上述计算过程。

3）合并（Combine）：将子问题的解合并成原问题的解。

分治算法的实例：归并排序

对已知的含有n个元素的未排序的序列排序：

- **归并排序的基本思路**：

  分解：分解待排序的n个元素的序列成各具n/2个元素的两个子序列。

  解决：使用归并排序递归地排序两个子序列。

  合并：合并两个已排序的子序列以产生已排序的完整序列

- **归并排序的过程描述：**

  - MERGE-SORT(A,p,r)
  - MERGE(A,p,q,r)

详见2.3，P16~22

- **归并排序的时间分析：** $T(n)=2T(n/2)+cn = O(n\log n)$

认识分治：

- 当子问题足够小时，不需要再进一步分解，则称之为基本情况(base case)。基本情况的子问题可以直接求解。
- 但若子问题还足够大，不能直接求解，则需要进一步分解子问题并递归求解，称之为递归情况(recursive case)。

■分治与递归

- 子问题的性质与原问题一样，所以对子问题的求解实际上是算法的递归执行。
- 分治和递归是“一对好兄弟”。 “Recurrences go hand in hand with the divide-and-conquer paradigm”
- **分治的基本思想就是递归求解策略。**

4.1 最大子数组问题

- 一个关于炒股的story：**自学**
- 求解炒股问题的算法模型：最大子数组问题

已知数组A，在A中寻找“和”最大的非空连续子数组。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

$\underbrace{18 \quad 20 \quad -7 \quad 12}_{\text{maximum subarray}}$

——称这样的连续子数组为最大子数组（***maximum subarray***）

- 怎么求解？

方法一：**暴力求解法**（**brute-force solution**）

搜索A的每一对起止下标区间的和，和最大的子区间就是最大子数组，时间复杂度：

$$\binom{n-1}{2} = \Theta(n^2)$$

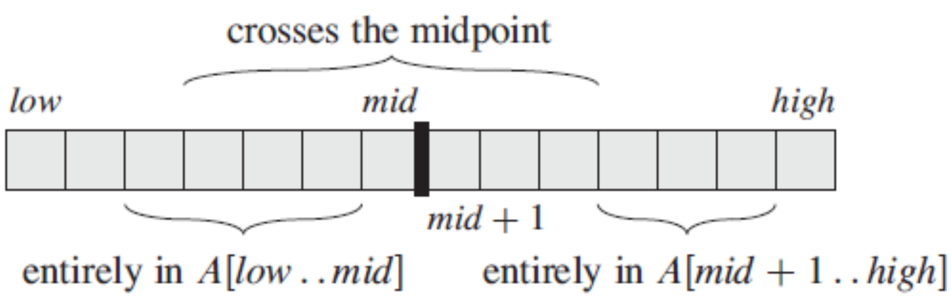
方法二：使用分治策略求解

■基本思想

设当前要寻找子数组A[low...high]的最大子数组。首先使用分治技术，将子数组A[low...high]划分为两个规模尽量相等的子子数组，分割点：

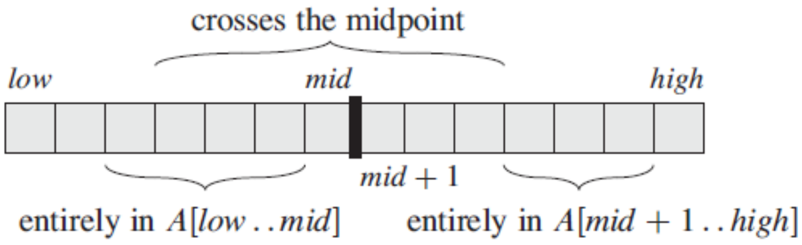
mid=（low+high）/2

然后分别求解A[low...mid]和A[mid+1...high]。



(a)

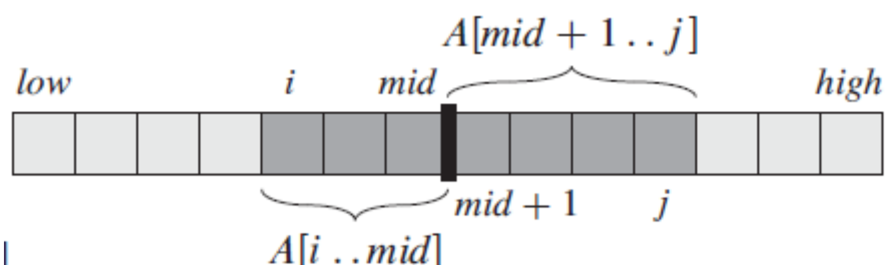
如图所示， A[low...high]的任何连续子数组A[i...j]所处的位置必然是以下三种情况之一：



(a)

A[low...high]的连续子数组A[i...j]所处的位置必是下面三种情况之一：

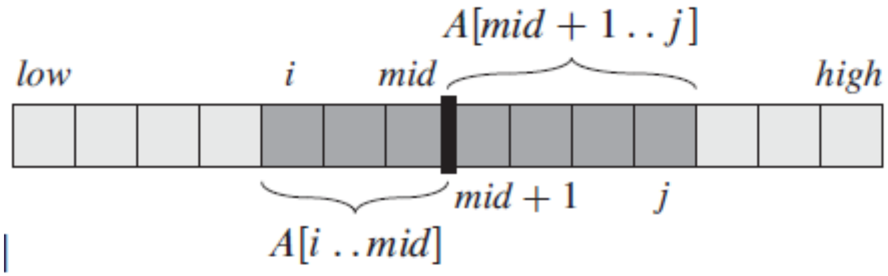
- entirely in the subarray $A[low\,..\,mid]$, so that $low \le i \le j \le mid$,
- entirely in the subarray $A[mid+1\,..\,high]$, so that $mid < i \le j \le high$, or
- crossing the midpoint, so that $low \le i \le mid < j \le high$.



(b)

则， A[low..high]的一个最大子数组所处的位置也必然是这三种情况之一。

且，A[low..high]的这个“最大子数组”必然是完全位于A[low.. mid]中、完全位于A[mid+1 .. high]中或则跨越中点的所有子数组中和最大的那个。



(b)

- 求解过程分析

1）对于完全位于A[low .. mid]和A[mid+1 .. high]中的最大子数组，可以在A[low .. mid]和A[mid+1 .. high]这两个更小的子问题上递归求解。

2）怎么寻找跨越中点的最大子数组呢？

—— **可以在线性时间内求出跨越中点的最大子数组**。

- 分析：
  - 这样的子数组必然跨越中点mid；而任何跨越中点的连续子数组都由两个子数组A[i...mid]和A[mid+1...j]组成;
  - 因此只需要找出形如A[i .. mid]和A[mid+1 ..j]的最大子数组，然后合并即可得到跨越中点时的A[low ... high]中的最大子数组。

基于上述分析，可以设计以下2个过程求解最大子数组问题

过程1：FIND-MAX-CROSSING-SUBARRAY，求跨越中点的最大子数组

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    left-sum = -∞
    sum = 0
    for i = mid downto low
        sum = sum + A[i]
        if sum > left-sum
            left-sum = sum
            max-left = i
    right-sum = -∞
    sum = 0
    for j = mid + 1 to high
        sum = sum + A[j]
        if sum > right-sum
            right-sum = sum
            max-right = j
    return (max-left, max-right, left-sum + right-sum)
```

返回搜索的结果

- FIND-MAX-CROSSING-SUBARRAY(A,*low, mid, high*) takes $\Theta(n)$ time.

过程2： FIND-MAXIMUM-SUBARRAY ， 求最大子数组问题的分治算法

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

三者中的最大子数组作为问题的解返回

FIND-MAXIMUM-SUBARRAY时间分析

令 T(n)表示求解n个元素的最大子数组问题的执行时间。

1）n=1时，T(1)=Θ(1)。

2）对A[low .. mid]和A[mid+1 .. high]两个子问题的递归求解，每个子问题均约有n/2个元素，所以每个子问题的时间为T(n/2)。

—— **两个子问题递归求解的总时间是2T(n/2)**。

3）FIND-MAX-CROSSING-SUBARRAY的时间是Θ(n)。

可得FIND-MAXIMUM-SUBARRAY的执行时间T(n)的递归式：

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases} \Rightarrow T(n) = \Theta(n \lg n)$$

还有没有更快的算法？4.1-5给出了一个线性时间算法

4.2 Strassen矩阵乘法

回顾一下矩阵运算

已知两个n阶方阵： $A=(a_{ij})_{nxn}$， $B=(b_{ij})_{nxn}$

1）**矩阵加法**

$C=A+B=(c_{ij})_{nxn}$ ，其中, $c_{ij}=a_{ij}+b_{ij}$, $i,j=1,2,\ldots,n$

计算量：$\Theta(n^2)$

2）**矩阵乘法**

$C=AB=(c_{ij})_{nxn}$ ，其中，$c_{ij}=\sum_{1\le k\le n} a_{ik}b_{kj}$, $i,j=1,2,\ldots,n$

计算量：$O(n^3)$。

- 共有n^2个c_{ij}需要计算，每个c_{ij}需要n次乘运算, n−1次加法

- 实现两个n×n矩阵乘的过程

SQUARE-MATRIX-MULTIPLY(A, B)

```
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} .$$

众所周知，矩阵乘的计算时间是$\Theta(n^3)$.

能否可以用少于$\Theta(n^3)$的时间完成矩阵乘的计算?

1969年Strassen： $n^{2.81}$

Strassen矩阵乘法：基于分治的矩阵乘算法

简单的矩阵乘：2X2的两个矩阵相乘

1）直接相乘

$$A=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B=\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$C=AB=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$=\begin{bmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{bmatrix}$$

$$=\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

显然，A、B直接相乘，

需要8次乘法和4次加法

2) Strassen的计算方法：

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$C = AB = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

令： $P=(a_{11}+a_{22})(b_{11}+b_{22})$

$Q=(a_{21}+a_{22})\, b_{11}$

$R=a_{11}\,(b_{12}-b_{22})$

$S=a_{22}\,(b_{21}-b_{11})$

$T=(a_{11} + a_{12})b_{22}$

$U=(a_{11} - a_{21})\,(b_{11} + b_{12})$

$V=(a_{12} - a_{22})\,(b_{21} + b_{22})$

则，

$$\begin{aligned} c_{11} &= P+S-T+V = (a_{11}+a_{22})(b_{11}+b_{22}) + a_{22}\,(b_{21}-b_{11}) \\ &\quad -(a_{11} + a_{12})b_{22}+(a_{12} - a_{22})\,(b_{21} + b_{22}) \\ &\equiv a_{11}b_{11}+a_{12}b_{21} \\ c_{12} &= R+T \equiv a_{11}b_{12}+a_{12}b_{22} \\ c_{21} &= Q+S \equiv a_{21}b_{11}+a_{22}b_{21} \\ c_{22} &= P+R-Q-U \equiv a_{21}b_{12}+a_{22}b_{22} \end{aligned}$$

Strassen计算方法的分析：

令： $P=(a_{11}+a_{22})(b_{11}+b_{22})$

$Q=(a_{21}+a_{22})\,b_{11}$

$R=a_{11}\,(b_{12}-b_{22})$

$S=a_{22}\,(b_{21}-b_{11})$

$T=(a_{11}+a_{12})b_{22}$

$U=(a_{11}-a_{21})\,(b_{11}+b_{12})$

$V=(a_{12}-a_{22})\,(b_{21}+b_{22})$

则，

$c_{11}=P+S-T+V$

$c_{12}=R+T$

$c_{21}=Q+S$

$c_{22}=P+R-Q-U$

计算量分析：

乘法次数：7次

加（减）法次数：18次

特点：

增加了加（减）法计算量，减少了乘法计算量

带来的改进：

直观上，在用程序完成的计算中，通常认为乘法运算比加法运算需要更多的时间，Strassen矩阵乘通过减少乘法计算量、适当增加加法计算量，

矩阵乘的**分治思路**

- 设 $n = 2^k$ ，两个n阶方阵为

$$A = (a_{ij})_{nxn}$$

$$B = (b_{ij})_{nxn}$$

（注：若 $n \neq 2^k$ ，可通过在A和B中补0使之变成阶是2的幂的方阵）

首先，将A和B分成4个(n/2)x(n/2)的子矩阵：

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

方法1：朴素的分治思想 —— 简单的矩阵分块相乘

$$C = AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

8次(n/2)x(n/2) 矩阵乘
4次(n/2)x(n/2) 矩阵加

注：任意两个子矩阵块的乘可以沿用同样的规则：如果子矩阵的阶大于2，则将子矩阵分成更小的子矩阵，直到每个子矩阵只含一个元素为止。从而构造出一个递归计算过程。

简单的矩阵分块相乘时间分析：

令T(n)表示两个n×n矩阵相乘的计算时间，则首次分块时，需要：

1) 8次(n/2)×(n/2) 矩阵乘 - - - - - >8T(n/2)

2) 4次(n/2)×(n/2) 矩阵加 - - - - - >dn^2

故，

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + dn^2 & n > 2 \end{cases}$$

其中，b，d是常数

化简得：T(n) = O(n^3) ⟶ 没有变化

方法2：Strassen矩阵乘的一般方法

令

$P=(A_{11}+A_{22})(B_{11}+B_{22})$

$Q=(A_{21}+A_{22})B_{11}$

$R=A_{11}(B_{12}-B_{22})$

$S=A_{12}(B_{21}-B_{11})$

$T=(A_{11}+A_{12})B_{22}$

$U=(A_{11}-A_{21})(B_{11}+B_{12})$

$V=(A_{12}-A_{22})(B_{21}+B_{22})$

$$A=\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$B=\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

则，

$C_{11}=P+S-T+V$

$C_{12}=R+T$

$C_{21}=Q+S$

$C_{22}=P+R-Q-U$

计算量：

(n/2)x(n/2) 矩阵乘法：7次

(n/2)x(n/2) 矩阵加法：18次

注：Strassen矩阵乘也是一个递归求解过程，任意两个子矩阵块的乘可以沿用同样的规则进行。

Strassen矩阵乘法的计算复杂度分析

令T(n)表示两个$n=2^k$阶矩阵的Strassen矩阵乘所需的计算时间，则有：

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$ 其中， a和b是常数

化简：

$$\begin{aligned} T(n) &= an^2(1+7/4+(7/4)^2+\ldots+(7/4)^{k-1}) + 7^k T(1) \\ &\le cn^2(7/4)^{\log n}+7^{\log n} \\ &= cn^2 n^{\log(7/4)} + n^{\log 7} \\ &= cn^{\log 4+\log 7-\log 4} + n^{\log 7} \\ &= (c+1)n^{\log 7} \\ &= O(n^{\log 7}) \approx O(n^{2.81}) \end{aligned}$$

这里，k=log*n*

Strassen矩阵乘法是通过递归实现的，C++实现代码如下：

```cpp
#include <iostream>

using namespace std;

const int N = 6;     //Define the size of the Matix

template<typename T>
void Strassen(int n, T A[][N], T B[][N], T C[][N])

template<typename T>
void input(int n, T p[][N]);

template<typename T>
void output(int n, T C[][N]);

int main() {
    //Define three Matices
    int A[N][N],B[N][N],C[N][N];

    //对A和B矩阵赋值，随便赋值都可以，测试用
    for(int i=0;i<N;i++){
      for(int j=0; j<N; j++) {
        A[i][j] =i*j;
        B[i][j] =i*j;
      }
    }

    //调用Strassen方法实现C=A*B
    Strassen(N, A, B, C);

    //输出矩阵C中值
    output(N, C);

    system("pause");
    return 0;
}

/**The Input Function of Matrix*/
template<typename T>
void input(int n, T p[][N]){
     for(int i=0; i<n; i++){
       cout<<"Please Input Line"<<i+1<<endl;
       for(int j=0;j<n;j++){
         cin>>p[i][j]
       }
     }
}

/**The Output Fanction of Matrk*/
template<typename T>
void output(int n, T C[][N]){
     cout<<"The Output Matrix is :"<<endl;
     for(int i=0; i<n; i++){
       for(int j=0;j<n;j++){
         cout<<C[i][j]<<" "<<endl;
       }
     }
}

/**Matrix Multplication as the normal algorithm*/
template<typename T>
void Matrix_Multiply(T A[][N],T B[][N],T C[][N]){ //Calculating A*B->C
     for(int i=0; i<2;i++){
       for(int j=0;j<2;j++) {
         C[i][j]= 0;
         for(int t=0; t<2; t++) {
          C[i][j] =C[i][j]+A[i][t]*B[t][j]
         }
       }
     }
}

/**Matrix Addition*/
template <typename T>
void Matrix_Add(int n, T X[][N],T Y[][N], T Z[][N]) {
     for(int i=0; i<n; i++){
       for(int j=0;j<n;j++){
         Z[i][j]=X[i][j]+Y[i][j]
       }
     }
}

/**Matrix Subtraction*/
template <typename T>
void Matrix_Sub(int n, T X[][N],T Y[][N],T Z[][N]) {
     for(int i=0; i<n; i++){
       for(int j=0;j<n;j++){
         Z[i][j]=X[i][j]-Y[i][j];
       }
     }
}

/**
* 参数n指定矩阵A,B,C的阶数，因为随着递归调用Strassen函数
* 矩阵A,B,C的阶数是递减的N只是预留足够空间而已
*/
template <typename T>
void Strassen(int n, T A[][N],T B[][N],T C[][N]){
     T A11[N][N], A12[N][N],A21[N][N],A22[N][N];
     T B11[N][N], B12[N][N],B21[N][N],B22[N][N];
     T C11[N][N], C12[N][N],C21[N][N],C22[N][N];
     T M1[N][N], M2[N][N],M3[N][N],M4[N][N],M5[N][N], M6[N][N],M7[N][N];
     T AA[N][N], BB[N][N];

     if(n == 2){ //2-order
       Matix_Multiply(A,B, C);
     } else {
       //将矩阵A和B分成阶数相同的四个子矩阵，即分治思想。
       for(int i=0;i<n/2; i++) {
         for(int j=0; j<n/2;j++){
          A11[i][j]=A[i][j]
          A12[i][j]=A[i][j+n/2]
          A21[i][j]=A[i+n/2][j]
          A22[i][j]=A[i+n/2][j+n/2]

          B11[i][j]=B[i][j]
          B12[i][j]=B[i][j+n/2]
          B21[i][j]=B[i+n/2][j]
          B22[i][j]=B[i+n/2][j+n/2]
         }
       }

       //Calculate M1 =(A0 + A3) × (B0 +B3)
       Matix_Add(n/2,A11, A22, AA);
       Matix_Add(n/2,B11, B22, BB);
       Strassen(n/2,AA, BB, M1)

       //Calculate M2 =(A2 + A3) × B0
       Matix_Add(n/2,A21, A22, AA);
       Strassen(n/2,AA, B11, M2);

       //Calculate M3 =A0 × (B1 - B3)
       Matix_Sub(n/2,B12, B22, BB);
       Strassen(n/2,A11, BB, M3);

       //Calculate M4 =A3 × (B2 - B0)
       Matix_Sub(n/2,B21, B11, BB);
       Strassen(n/2,A22, BB, M4);

       //Calculate M5 =(A0 + A1) × B3
       Matix_Add(n/2,A11, A12, AA);
       Strassen(n/2,AA, B22, M5);

       //Calculate M6 =(A2 - A0) ×(B0 + B1)
       Matix_Sub(n/2,A21, A11, AA);
       Matix_Add(n/2,B11, B12, BB);
       Strassen(n/2,AA, BB, M6)

       //Calculate M7 =(A1 - A3) ×(B2 + B3)
       Matix_Sub(n/2,A12, A22, AA);
       Matix_Add(n/2,B21, B22, BB);
       Strassen(n/2,AA, BB, M7)

       //Calculate C0 = M1+ M4-M5 +M7
       Matix_Add(n/2,M1,M4,AA);
       Matix_Sub(n/2,M7,M5,BB);
       Matix_Add(n/2,AA, BB, C11);

       //Calculate C1 = M3+ M5
       Matix_Add(n/2,M3,M5,C12);

       //Calculate C2 = M2+ M4
       Matix_Add(n/2,M2,M4,C21);

       //Calculate C3 = M1- M2 +M3 +M6
       Matix_Sub(n/2,M1,M2,AA);
       Matix_Add(n/2,M3,M6,BB);
       Matix_Add(n/2,AA, BB, C22);

       //Set the result to C[][N]
       for(int i=0;i<n/2; i++) {
         for(int j=0; j<n/2;j++){
          C[i][j] =C11[i][j]
          C[i][j+n/2] =C12[i][j]
          C[i+n/2][j] =C21[i][j]
          C[i+n/2][j+n/2] =C22[i][j]
         }
       }
     }
}
```

- Strassen算法的发表引起很大的轰动。
- 但从实用的角度看，Strassen算法并不是解决矩阵乘法的最好选择：

（1）隐藏在Strassen算法运行时间$\Theta(n^{\log 7})$中的常数因子比直接过程的$\Theta(n^3)$的常数因子大。

（2）对于稀疏矩阵，有更快的专用算法可用。

（3）Strassen算法的数值稳定性不如直接过程，其计算过程中引起的误差积累比直接过程大。

（4）递归过程生成的子矩阵会消耗更多的存储空间。

- 不断地在改进。见P63的分析讨论。
- 目前已知的n╳n矩阵乘的最优时间是$O(n^{2.376})$

4.3 求解递归式

- 分治和递归是"一对好兄弟"。

设开始时，问题的规模为n，之后被分解为两个子问题，子问题的规模分别n_1和n_2。

令$T(n)$表示对规模为n时问题求解的时间，则规模分别为n_1和n_2的子问题的求解时间可表示为$T(n_1)$和$T(n_2)$。

一般，$T(n)$ 和$T(n_1)$、$T(n_2)$的关系可表示为

$$T(n) = T(n_1)+T(n_2)+f(n)$$

如果n1=n2≈n/2，则T(n)可表示为：T(n) = 2T(n/2)+f(n)
如归并排序的时间：
T(n) = 2T(n/2)+cn
也可能如二分查找，时间为：
T(n) =T(n/2)+1

- 其中f(n)是指除子问题递归求解以外的、其它必要处理所花费的时间。

因此，**分治算法的计算时间表达式也往往是递归式。**

- 如何化简递归式，以得到形式简单的限界函数？

这里，所谓求解递归式就是化简递归式，以得到形式简单的函数表示。

递归式求解的结果是得到形式简单的渐进限界函数表示（用O、Ω、Θ表示的函数式）。

介绍三种常用的递归式求解方法：

- 代换法
- 递归树法
- 主方法

对表达式细节的简化

为便于处理，通常做如下假设和简化处理

(1) 运行时间函数T(n)的定义中，一般假定自变量为正整数。

(2) 忽略递归式的边界条件，即n较小时函数值的表示。

- 原因在于，虽然递归式的解会随着T(1)值的改变而改变，但此改变不会超过常数因子，对函数的阶没有根本影响。

(3) 对上取整、下取整运算做合理简化，

如：$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + f(n)$$

通常忽略上、下取整函数，直接写作：

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

注：被简化的细节并不是不重要，只是对这些细节的处理不影响算法分析的渐近界，是在“无穷大”分析中的合理假设和简化。

在细节被简化处理的同时，也要知道它们在什么情况下是“实际”重要的。这样就可以了解算法在各种情况的具体执行情况。

1) 代换法(The substitution method for solving recurrences)

- 用代换法解递归式基本思想：

先猜测解的形式，然后用数学归纳法求出解中的常数，并证明解是正确的。

此时，用猜测的解作为归纳假设，在推论证明时作为较小值代入函数（因此得名“代入法”），然后证明推论的正确性。

- 用代换法解递归式的步骤：

(1) 猜测解的形式

(2) 用数学归纳法证明猜测的正确性

例：用代换法确定下式的上界

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

分析：该式与 $T(n) = 2T(n/2) + n$ 类似，故猜测其解为*O(nlogn)*。

1）代入法要证明的是：恰当选择常数c，可使得*T(n)≤cnlogn*成立。

所以现在设法证明*T(n)≤cnlogn*，并确定常数c的存在。

2）假设该界对$\lfloor n/2 \rfloor$成立，即 $T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$, 然后在数学归纳法推论证明阶段对递归式做代换，有：

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\le cn\log(n/2) + n \\ &= cn\log n - cn\log 2 + n \\ &= cn\log n - (c-1)n \end{aligned}$$

故，要使*T(n)≤cnlogn成立，只要c≥1就可以，这样的c是存在的、合理的。*

上面的证明过程，证明了当n足够大时猜测的正确性。但边界值呢？

即： **$T(n) \le cn\log n$ 的结论对于小n成立吗？**

分析：

事实上，对n=1，上述结论存在问题：

- 作为边界条件，我们有理由假设T(1)=1，但对n=1，$T(1) \le c \cdot 1 \cdot \log 1 = 0$，与T(1)=1不相符。

也即， $T(n) \le cn\log n$ 对于归纳证明的基本情况不成立。

- 怎么处理？

从n_0的定义出发：

只需要存在常数n_0，使得$n \geq n_0$时结论成立即可，所以n_0不一定取1。

所以，这里，我们不取$n_0=1$，而取$n_0=2$，并将T(2)、T(3)作为归纳证明中的边界条件代替T(1) ，使得T(2)、T(3) 满足$T(n) \leq cnlogn$。

- 但需要注意的是，依旧要合理地假设T(1) =1。

而$n>3$时，递归计算不再直接依赖T(1)，而使用T(2)、T(3)即可完成。

进一步分析：

带入T(1)=1，通过递归式有，T(2) = 4，T(3)=5，

- **如何使T(2)、T(3)满足$T(n) \le cn\log n$？**

只要c取足够大的常数，就会有 $T(2) \le c2\log 2$ 和$T(3) \le c3\log 3$ 成立即可。这样的c是什么？

答案：只要$c \ge 2$即可。

综上所述，取常数$c \ge 2$，最终的结论$T(n) \le cn\log n$就成立。命题得证。

如何猜测递归式的解呢？

遗憾的是，并不存在通用的方法来猜测递归式的正确解。

1）主要靠经验

- 尝试1：看有没有形式上类似的表达式，以此推测新递归式解的形式。
- 尝试2：先猜测一个较宽的界，然后再缩小不确定范围，逐步收缩到紧确的渐近界。
- 避免盲目推测

如：$T(n) \le 2(c\lfloor n/2 \rfloor) + n \le cn + n = O(n)$

原因：并未证出一般形式T(n)≤cn。（cn+n$\not\le$cn）

必要的时候要做一些技术处理

1）去掉一个低阶项

2）变量代换：对陌生的递归式做些简单的代数变换，使之变成较熟悉的形式。

例：化简 $T(n) \le 2T(\lfloor \sqrt{n} \rfloor) + \log n$

分析：原始形态比较复杂

做代数代换：令 $m = \log n$ ，则n=2m，$\sqrt{n} = 2^{m/2}$

同时，为简单起见，忽略下取整细节 $\lfloor \sqrt{n} \rfloor$，

直接使用 $\sqrt{n}$ ，

得：$T(2^m) \le 2T(2^{m/2}) + m$

$$T(2^m) \le 2T(2^{m/2}) + m$$

再设**S(m) = T(2^m)**，得以下形式递归式：

$$S(m) \le 2S(m/2) + m$$

从而获得形式上熟悉的递归式。

根据前面的一些讨论，可得新的递归式的上界是：$O(m \log m)$

再将S(m)带回T(n)，有，

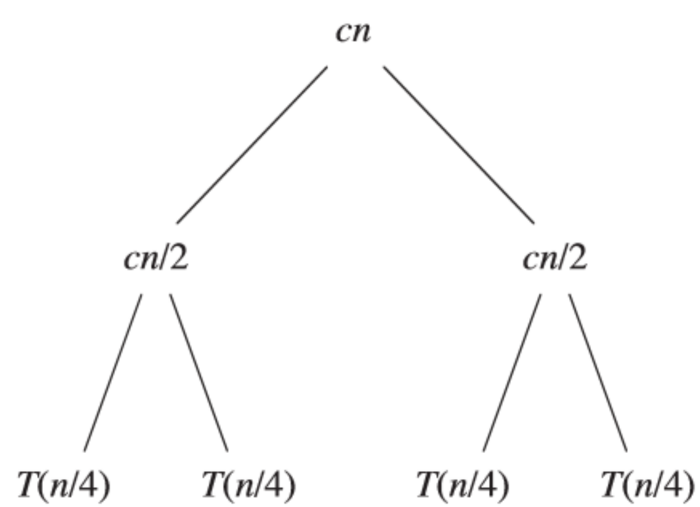
$$\begin{aligned} T(n) &= T(2^m) \\ &= S(m) = O(m \log m) \\ &= O(\log n \log \log n) \end{aligned}$$

这里， $m = \log n$

2）递归树法(The recursion-tree method for solving recurrences)

- 根据递归式的定义，可以画一棵递归树，来帮助我们猜测递归式的解。
- 递归树：反应递归的执行过程。每个节点表示一个单一子问题的代价，子问题对应某次递归调用，根节点代表顶层调用的代价，子节点分别代表各层递归调用的代价。

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



基于递归树的时间分析

节点代价：在递归树中，每个节点有求解相应（子）问题的代价(cost)。

层代价：每一层各节点代价之和。

总代价：整棵树的各层代价之和

目　标：利用树的性质，获取对递归式解的猜测，然后用代换法或其它方法加以验证。

例：已知递归式 $T(n)=3T(\lfloor n/4 \rfloor)+\Theta(n^2)$，求其上界

准备性工作：为简单起见，对一些细节做必要、合理的简化和假设，这里为：

（1）去掉底函数的表示

- 理由：底函数和顶函数对递归式求解并不“重要”。

（2）假设n是4的幂，即$n=4^k$，$k=\log_4 n$。

- 一般，当证明$n=4^k$成立后，再加以适当推广，就可以把结论推广到n不是4的幂的一般情况了。

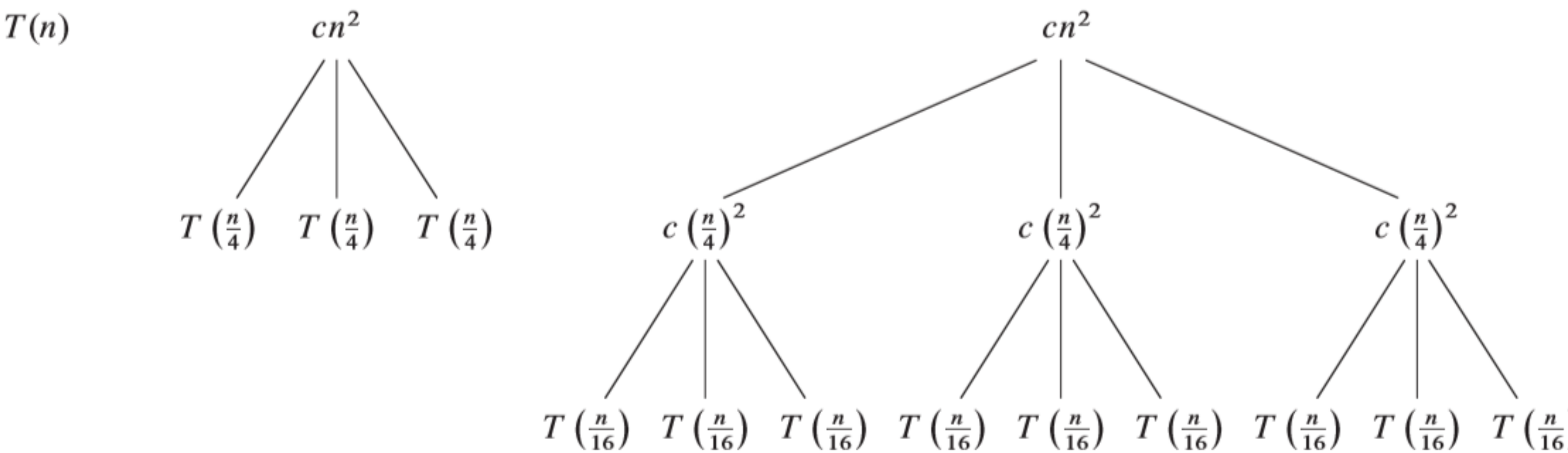
（3）展开 $\Theta(n^2)$, 代表递归式中非重要项。

- 假设其常系数为c，c>0，从而去掉 符号Θ转变成cn2的形式，便于后续的公式化简。

最终得以下形式的递归式： $T(n)=3T(n/4)+cn^2$

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

用递归树描述T(n)的演化过程:



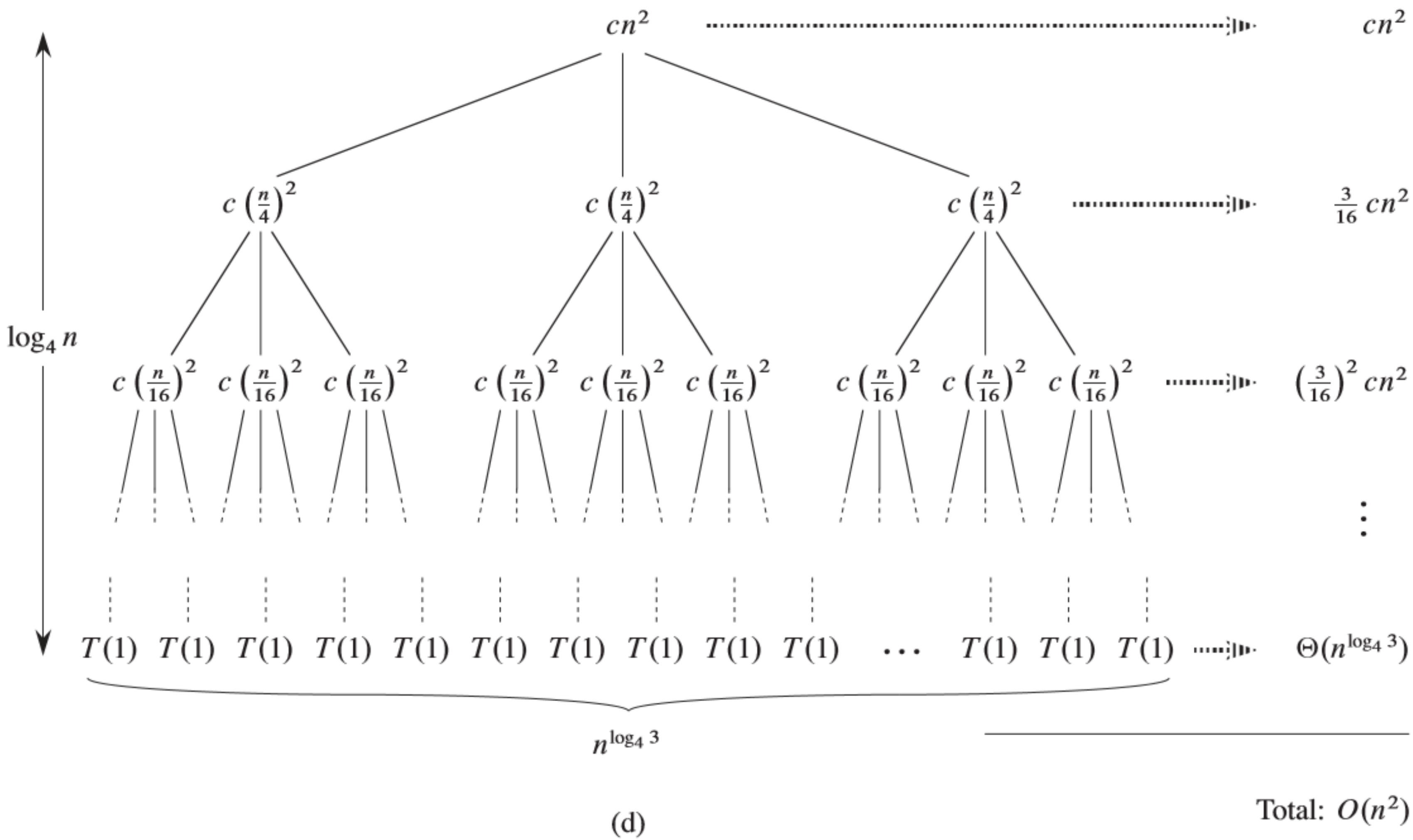
(a) (b) (c)

a) 对原始问题T(n)的描述。

b) 第一层递归调用的分解情况，cn^2是顶层计算除递归以外的代价，T(n/4)是分解出来的规模为n/4的子问题的代价，总代价$T(n)=3T(n/4)+cn^2$。

c) 第二层递归调用的分解情况。$c(n/4)^2$是三棵二级子树除递归以外的代价。

继续扩展下去，直到递归的最底层，得到如下形式的递归树：



d) 完全扩展的递归树，递归树高度为$\log_4 n$(共有$\log_4 n+1$层)

树的深度：子问题的规模按1/4的方式减小，在递归树中，深度为i的节点，子问题的大小为$n/4^i$。

当$n/4^i=1$时，子问题规模仅为1，达到边界值。

所以，

- 节点分布层：$0 \sim \log_4 n$
- 树共有$\log_4 n+1$层
- 从第2层起，每一层上的节点数为上层节点数的3倍
- 深度为i的层节点数为3^i。

代价计算

（1）内部节点：位于0~$\log_4 n-1$ 层

深度为i的节点的局部代价为 $c(n/4^i)^2$ ，

i层节点的总代价为：$3^i c(n/4^i)^2=(3/16)^i cn^2$ 。

（2）叶子节点：位于$\log_4 n$层，共有 $3^{\log_4 n}=n^{\log_4 3}$ 个，

每个叶子节点的代价为T(1),

总代价为 $n^{\log_4 3}T(1)=\Theta(n^{\log_4 3})$

（3）树的总代价

整棵树的总代价等于各层代价之和，则有

$$T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \ldots\ldots + (\frac{3}{16})^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n-1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})$$

利用等比数列化简上式。

- 对于实数x≠1，和式 $\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n$ 是一个几何级数（等比数列），其值为 $\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$
- 当和是无穷的且|x|<1时，得到无穷递减几何级数，此时

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$$

- T(n)中，cn^2项的系数构成一个递减的等比级数。
- 将T(n)扩展到无穷，即有

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)
\end{aligned}
$$

至此，获得T(n)解的一个猜测：$T(n)=O(n^2)$，成立吗？

用代换法证明猜测的正确：

- 将T(n)≤dn²作为归纳假设，d是待确定的常数，带入推论证明过程，有

$$
\begin{aligned}
T(n) &= 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \\
&\le 3T(\lfloor n/4 \rfloor) + cn^2 \le 3d\lfloor n/4 \rfloor^2 + cn^2 \\
&\le 3d(n/4)^2 + cn^2 \\
&= \frac{3}{16}dn^2 + cn^2
\end{aligned}
$$

c是引入的另一个常量

显然，要使得T(n)≤dn²成立，只要d≥(16/13)c即可。所以，T(n)≤dn²的猜测成立。

定理得证（边界条件的讨论略）。

另： O(n²)是T(n)的一个紧确界，为什么？

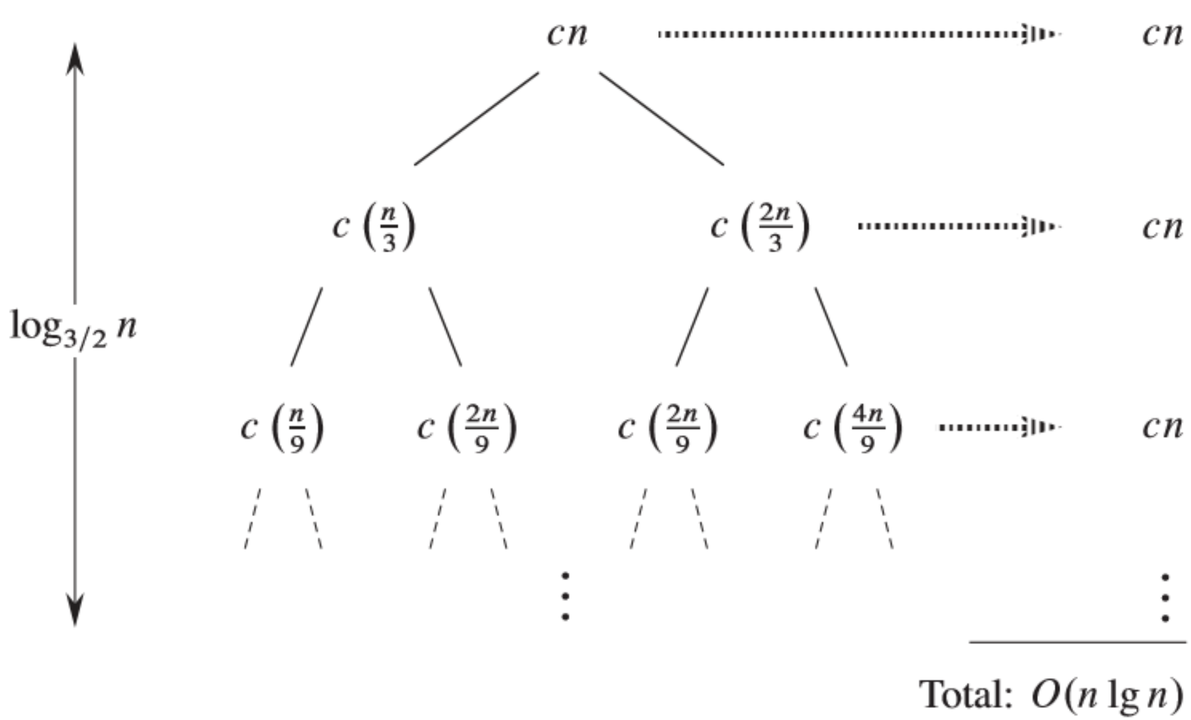
例 求表达式的 $T(n)=T(n/3)+T(2n/3)+O(n)$ 上界

（这里，表达式中直接省略了下取整和上取整函数）

- 进一步地，引入常数c，展开O(n)，得：

$$T(n) \le T(n/3) + T(2n/3) + cn$$

递归树为



分析：

- 该树并不是一个完全的二叉树。
  - 从根往下，越来越多的内节点在左侧消失(1/3分叉上)，因此每层的代价并不都是cn，而是≤cn的某个值。
- 树的深度：
  - 在上述形态中，最长的路径是最右侧路径，由

$$n \rightarrow (2/3)n \rightarrow (2/3)^2 n \rightarrow \dots \rightarrow 1$$

组成。
  - 当$k=\log_{3/2}n$时，$(3/2)^k/n=1$，所以树的深度为$\log_{3/2}n$。

- 递归式解的猜测：
  - 至此，我们可以合理地猜测该树的总代价至多是层数乘以每层的代价，并鉴于上面关于层代价的讨论，我们可以假设递归式的上界为：

$$O(cn\log_{3/2}n)=O(n\log n)$$

注：这里，我们假设每层的代价为cn，事实上，cn为每层代价的上界，这一假设是合理的细节简化处理。

猜测的证明：证明O(nlogn)是递归式的上界

即证明：T(n)≤dnlog$n$，d是待确定的合适正常数。

$$
\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3)\log(n/3) + d(2n/3)\log(2n/3) + cn \\
&= (d(n/3)\log n - d(n/3)\log 3) + (d(2n/3)\log n - d(2n/3)\log 3/2)) + cn \\
&= dn\log n - d((n/3)\log 3 + (2n/3)\log(3/2)) + cn \\
&= dn\log n - d((n/3)\log 3 + (2n/3)\log 3 - (2n/3)\log 2) + cn \\
&= dn\log n - dn(\log 3 - 2/3) + cn \le \underline{dn\log n}
\end{aligned}
$$

成立吗？

上式的成立条件：$d \ge c/(\log 3 - (2/3))$，存在！

∴ 猜测正确，递归式解得证。

3）主方法（The master method for solving recurrences ）

如果递归式有如下形式，在满足一定的条件下，可以用主方法直接给出渐近界：

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

其中，a、b是常数，且a≥1，b>1；f(n)是一个渐近正的函数。

含义：规模为n的原问题被分为a个子问题，每个子问题的规模是n/b，a和b是正常数。子问题被递归地求解，T(n)是原始问题的时间，每个子问题的时间为T(n/b)；问题分解及子问题解合并及其它有关运算的代价由函数f(n)描述。

上式给出了算法总代价与子问题代价之间的关系。

注：这里采用了细节的简化，没有考虑n/b的取整问题，省略了下取整、上取整，但本质上不影响对递归式渐近行为的分析。

对上述形式的递归式渐近界的求解可用称之为“主定理”的结论给出的。

定理2.1 主定理

设a≥1和b>1为常数，设f(n)为一函数，T(n)是定义在非负整数上的递归式：

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

其中n/b指$\lfloor n/b \rfloor$或$\lceil n/b \rceil$。

则T(n)可能有如下的渐近界：

1）若对于某常数ε>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 $T(n)=\Theta(n^{\log_b a}\log n)$

3）若对某常数ε>0，有 $f(n)=\Omega(n^{\log_b a+\varepsilon})$，且对常数c<1与所有足够大的n，有 $af(n/b)\le cf(n)$，则 $T(n)=\Theta(f(n))$。

理解主定理：

1）T(n)的解似乎与f(n)和 $n^{\log_b a}$ 有“密切关联”：

f(n)和 $n^{\log_b a}$ 比较，T(n)取了其中较大的一个。

如：第一种情况，函数 $n^{\log_b a}$ 比较大，所以 $T(n)=\Theta(n^{\log_b a})$

第三种情况，函数f(n) 比较大，所以 $T(n)=\Theta(f(n))$

第二种情况，两个函数一样大，则乘以对数因子，得

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

2) 在第一种情况中，f(n)要**多项式**地小于 $n^{\log_b a}$。即，对某个常量ε>0，f(n)必须渐近地小于 $n^{\log_b a}$，两者相差了一个 n^{ε} 因子。

3) 在第三种情况中，f(n)不仅要大于 $n^{\log_b a}$ ，而且要多项式地大于 $n^{\log_b a}$, 还要满足一个“规则性”条件 $af(n/b) \le cf(n)$ 。

4) 若递归式中的f(n)与 $n^{\log_b a}$ 的关系不满足上述性质：

- f(n)小于等于 $n^{\log_b a}$ ，但不是多项式地小于。
- f(n)大于等于 $n^{\log_b a}$ ，但不是多项式地大于。

则不能用主方法求解该递归式。

使用主方法：分析递归式满足主定理的哪种情形，即可得到解（无需证明，保证正确）。

例2.6 解递归式 $T(n)=9T(n/3)+n$

分析：这里，a=9，b=3，f(n)=n。

则$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$ 。

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

所以对应主定理的第一种情况。

于是有：$T(n) = O(n^2)$

例2.7 解递归式 $T(n) = T(2n / 3) + 1$

分析：这里，a=1，b=3/2，f(n)=1，因此有$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$,

且有$f(n) = \Theta(n^{\log_b a}) = \Theta(1)$，故主定理第二种情况成立，

T(n)=Θ(log$n$)

例2.8 解递归式 $T(n)=3T(n/4)+n\log n$

分析：这里，a=3，b=4，f(n)=nlog$n$，

$$n^{\log_b a}=n^{\log_4 3}=O(n^{0.793})$$

故，$f(n)=\Omega(n^{\log_4 3+\varepsilon})$ 成立，其中ε≈**0.2**。

同时，对足够大的n，

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

其中c = 3/4。

所以第三种情况成立，T（n） = Θ（nlogn）。

例2.9 递归式$T(n)=2T(n/2)+n\log n$ 不能用主方法求解

分析：这里，a=2，b=2，

$$n^{\log_b a}=n^{\log_2 2}=O(n)$$

且，f(n)=nlog$n$ 渐进大于 $n^{\log_b a}=O(n)$

第三种情况成立吗？

$n^x (\log n)^y < n^{x+\varepsilon}$

事实上不成立，因为对于任意正常数ε，

$$f(n)/n^{\log_b a}=(n\log n)/n=\log n<n^{\varepsilon}$$

不满足 $f(n)=\Omega(n^{\log_b a+\varepsilon})$

$f(n)=\Omega(n^{\log_b a+\varepsilon})$

注：要想 $f(n)=\Omega(n^{\log_b a+\varepsilon})$，应有$f(n)/n^{\log_b a}>n^{\varepsilon}$。

因此该递归式落在情况二和情况三之间，条件不成立，不能用主定理求解。

4.6 证明主定理

为什么主定理是正确的？

主定理证明：（略，P55）