第 3 章

分　治　法

3.1　概　　念

3.1.1　分治法的基本思想

分治法是这样一种方法：对于一个输入规模为 n 的问题，用某种方法把输入分割成 k 个子集($1<k\leqslant n$)，从而产生 k 个子问题，k 个子问题解决后，再用某种方法组合成原来问题的解。

基本思想：将问题分解成若干个子问题，然后求解子问题，由此得到原问题的解，即“分而治之”。

分治法是一个递归地求解问题的过程，在每层的递归中应用如下三个步骤。

分解(Divide)：将问题划分为一些子问题，子问题的形式与原问题一样，只是规模更小。

解决(Conquer)：递归地求解出子问题。如果子问题的规模足够小，则停止递归，直接求解。

合并(Combine)：将子问题的解组合成原问题的解。

其过程如图 3-1 所示。

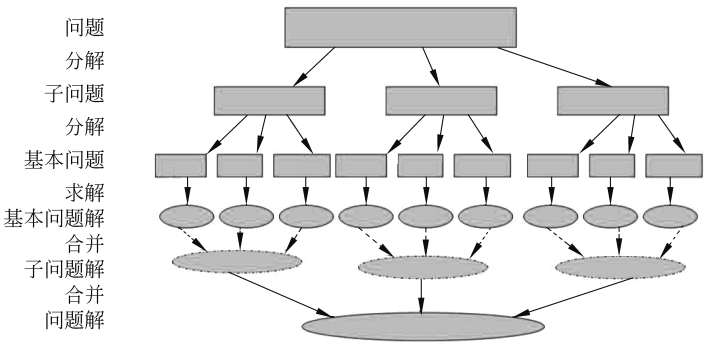

图 3-1　分治法问题求解过程

伪代码实现：

```
divide-and-conquer(S){
    if (small(S)) return(adhoc(S));
    divide S into smaller subset S1,S2,…,Si,…Sk;
    for(i=1; i<=k; i++)
        yi ← divide-and-conquer(Si);
    return merge(y1,y2,…,yi,…,yk);
}
```

其中：

small(S)是一个布尔值函数，它判断 S 的输入规模是否小到无须进一步分治就能算出其答案。

adhoc(S)是分治法的求解子算法。

merge(y_1,y_2,…,y_i,…,y_k)为解的归并子算法。

3.1.2　分治法所处理问题的基本特征

分治法虽然是一种常用的算法但并不适用于所用的问题场景，它所适用的问题应满足如下几个特征。

(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) 利用原问题分解的子问题所求出的解，应当可以合并为该问题的解。

(4) 各个子问题之间应当是相互独立的。

在以上几个特征中，最为关键的是特征(3)，这一特征直接决定该问题能否使用分治算法。当问题同时满足特征(1)和特征(2)而不满足特征(3)时，依然可以选择动态规划或者贪心算法来解决这一问题。而特征(4)则与分治算法的效率有关，当子问题之间不相互独立，使用分治法会做一些不必要的工作，此时，动态规划会是一个更好的选择。

3.1.3　分治算法的实现思路

1. 自顶向下的分治思路

此方法是从原问题出发，对原问题进行分解，在每次的分解后，所需要处理的问题相较于原来的问题规模更小。原问题的可解性依次地传递到下一层，通过依次对问题的处理，保证了原问题可以得到解决。下面以快速排序算法为例解释这一过程。

快速排序算法的思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

快速排序算法的每一次的划分，都会使一个元素处于一个正确的位置，其伪代码如下。

```
Quick_Sort(A,p,r){
    //输入：数组 A,起始位置 p,终止位置 r
    //输出：已排序数字
    if(p <r then)
        povit =A[p];              //使用子表的第一个值作为枢轴记录
        i <-p;
        j <-r;
        //从表的两端交替地向中间扫描
        while i <j do
            while i <j And A[j] >povit do
                j <-j-1;
                if i <j do
                    i <-i+1;
                    A[i] =A[j];
```

```
            end
        end
        while i <j And A[j] <=povit do
            i <-i+1;
            if i <j do
                j <-j-1;
                A[j] =A[i];
            end
        end
        A[i] <-povit;                    //枢轴记录保存到最终位置
        Quick_Sort(A, p, r-1);
        Quick_Sort(A, p+1, r);
    end
}
```

自顶向下的设计思路还同样适用于汉诺塔、二分搜索、线性时间选择等问题。

2. 自底向上的分治思路

对于使用分治法处理问题，除了在解决问题的过程中一步一步地将问题的规模缩小的同时处理问题，考虑到子问题的解的性质，还可以使用向上递推的方式，在已知解的基础上，解决规模更高的同类问题，最终使得原问题得以解决，这就是自底向上的分治策略。

这一分治策略的使用，较为经典的就是排序算法中的归并排序。在此，以二路归并算法为例进行说明。

二路归并排序的算法思想：假设初始序列包含 n 个记录，则可看成 n 个有序的子序列，每个子序列的长度为 1，然后两两归并，得到 $n/2$ 个长度为 2 或 1 的有序子序列；再两两归并，……，如此重复，直到得到一个长度为 n 的有序序列为止。

归并排序的算法实现如下。

```
MERGE(sourceArr,tempArr,sIndex,midIndex,eIndex){
    i =sIndex
    j =midIndex+1
    k =sIndex
    //取出两个有序子序列中最小的一个元素，放入新的有序数列中
    while (i! =midIndex+1 and j!=eIndex+1){
        if(sourceArr[i] <sourceArr[j]){
            tempArr[k] =sourceArr[i];
            i++;
        }else{
            tempArr[k] =sourceArr[j];
        }
        k++
    }
    while(i !=midIndex+1){          //前面的有序数列中还有元素
        tempArr[k++] =sourceArr[i++];
    }
    while(j !=eIndex+1){            //后面的有序数列中还有元素
        tempArr[k++] =sourceArr[j++];
```

```
        }
        //将有序数列复制给原数列
        for(m =sIndex to eIndex){
            sourceArr[m] =temp[m];
        }
    }
    //归一化
    MERGESORT(sourceArr,tempArr,sIndex,eIndex){
        //类似二分查找,每次取半
        if (sIndex <eIndex){
            mid =sIndex +(eIndex -sIndex)/2;
            MERGESORT(sourceArr,tempArr,sIndex,mid);
            MERGESORT(sourceArr,tempArr,mid+1,eIndex);
            MERGE(sourceArr,tempArr,sIndex,mid,eIndex);
        }
    }
```

3.2　折 半 查 找

3.2.1　问题描述

问题描述：在一个有序序列 S 中，查找其中是否包含元素 x，如果 x 是 S 中的元素，需要获得 x 在 S 中的位序。

3.2.2　问题分析

通过问题描述可知，带查找的序列 S 是一个有序序列，其按照由大到小或由小到大的顺序排列。

折半查找的思路是：在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键码相等，则查找成功；若给定值小于中间记录的关键码，则在中间记录的左半区继续查找；若给定值大于中间记录的关键码，则在中间记录的右半区继续查找。不断重复上述过程，直到查找成功，或所查找的区域无记录，查找失败。

例如，在一张包含 N 个记录的表(K_1，K_2，…，K_N)中，要查找 x 是否在表中。折半查找的思路是，首先将 x 与表中间记录的关键词 $K_{N/2}$ 进行比较，所得到的结果必属于下面 3 种情况之一：$K<K_{N/2}$、$K=K_{N/2}$、$K>K_{N/2}$。若本次查找不成功，则根据比较结果确定下一次应该在表的“哪一半”中去找，并对确定的“这一半”重复上述过程。如此进行下去，直到查找成功，或者直到表的长度为 0，查找以失败告终。在进行了至多 $\log_2 N$ 次比较之后，或者找到关键词等于 x 的记录，或者确定它不存在。过程如图 3-2 所示，其中，mid=(1+N)/2。

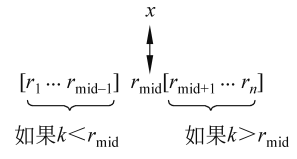

图 3-2　问题划分

3.2.3　问题求解

假设有如图 3-3 所示序列，要在其中查找值为 14 的记录。

1	2	3	4	5	6	7	8	9	10	11	12	13
7	14	18	21	23	29	31	35	38	42	46	49	52

图 3-3　待查找序列

首先，mid=$\frac{1+13}{2}$=7，以 14 比较 K_7=31，由于 14<K_7，查找范围只能在 K_7 的左侧，如图 3-4 所示。

此时 mid=$\frac{1+6}{2}$=3，以 14 比较 K_3=18，由于 14<K_3，查找范围只能在 K_3 的左侧，如图 3-5 所示。

1	2	3	4	5	6
7	14	18	21	23	29

图 3-4　一次查找后待查序列

1	2
7	14

图 3-5　二次查找后待查序列

此时 mid=$\frac{1+2}{2}$=1，以 14 比较 K_1=7，由于 14<K_1，查找范围只能在 K_1 的右侧。

比较 14 和 K_2=14，K_2 即为要查找的元素。

3.2.4　算法实现

折半查找算法递归实现如下。

```
int BinSearch2(int r[ ], int low, int high, int x){
    //数组 r[1] ～ r[n]存放查找集合
    if (low>high) return 0;
    else {
        mid=(low+high)/2;
        if(x<r[mid])
            return BinSearch2(r, low, mid-1,x);
        else if(x>r[mid])
            return BinSearch2(r, mid+1, high,x);
        else return mid;
    }
}
```

折半查找算法非递归实现如下。

```
int BinSearch2(int r[ ], int low, int high, int x){
    //数组 r[1] ～ r[n]存放查找集合
    if (low>high) return 0;
    while(low<=high){
        if(r[low].key==x)
            return low;
        if(r[high].key==x)
            return high;
        mid=low+(high-low)/2;
        if(r[mid].key==x)
```

```
            return mid;              //查找成功,返回
        if(r[mid].key<x)
            low=mid+1;               //继续在 R[mid+1..high]中查找
        else
            high=mid-1;              //继续在 R[low..mid-1]中查找
    }
    if(low>high)
        return 0;
}
```

3.2.5　折半查找判定树

折半查找的过程可以用二叉树来描述，树中的每个结点对应有序表中的一个记录，结点的值为该记录在表中的位置。通常称这个描述折半查找过程的二叉树为折半查找判定树，简称判定树。

判定树的构造过程如下。

(1) 当 $n=0$ 时，折半查找判定树为空。

(2) 当 $n>0$ 时，折半查找判定树的根结点是有序表中序号为 $\text{mid}=\dfrac{1+n}{2}$ 的记录，根结点的左子树是与有序表 $r[1]:r[\text{mid}-1]$ 相对应的折半查找判定树，根结点的右子树是与 $r[\text{mid}+1]:r[n]$ 相对应的折半查找判定树。

如图 3-6 所示为一棵折半查找判定树，其中，圆圈为内部结点，为表中位置号；方框为外部结点，表示不在表中的数据。

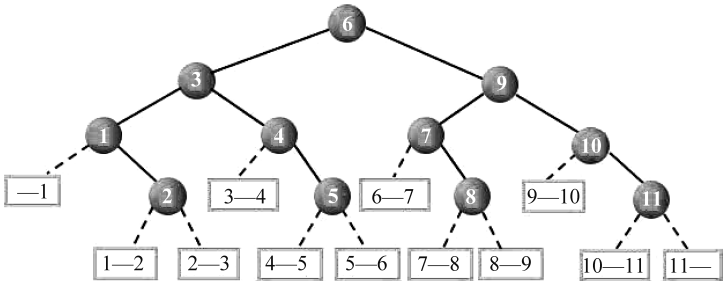

图 3-6　查找判定树

3.2.6　算法复杂度分析

因为折半查找每查找一次排除掉一半的不适合值，所以对于 n 个元素的情况如下。

一次二分后，查找范围剩下：$n_1=\dfrac{n}{2}$。

两次二分后，查找范围剩下：$n_2=\dfrac{n}{2^2}$。

…

m 次二分后，查找范围剩下：$n_m=\dfrac{n}{2^m}$。

在最坏情况下是在排除到只剩下最后一个值之后得到结果，即$\frac{n}{2^m}=1$。算法复杂度为$O(\log_2 n)$。

3.3 顺序统计

3.3.1 问题描述

在一个由 n 个元素组成的集合中，找出第 k 小元素(或第 k 大元素)。

例如，在一个元素集合中，最小值是第一个顺序统计量($i=1$)，最大值是第 n 个顺序统计量($i=n$)。

3.3.2 问题分析

如果集合中的 n 个元素已经排好序，找出第 k 小元素(或第 k 大元素)显然能够在$O(1)$的时间内完成。但要将集合中的 n 个元素已经排好序需要$O(\log_2 n)$的时间代价。

3.3.3 问题求解

下面采用分治法求解顺序统计问题。

(1) 首先从原始集合中随机选择一个元素 a，以 a 为界将原始集合中的元素放入三个新的集合 S_1，S_2，S_3。S_1 包含所有比 a 小的元素，S_2 中只有一个元素 a，S_3 中包含所有比 a 大的元素，如图 3-7 所示。要完成上述过程需要 $n-1$ 次比较操作。

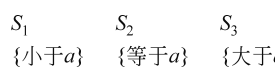

图 3-7 集合划分

(2) 假设要找出第 k 小的元素，这时判断 $m=|S_1|$ 值的大小。

① 如果 $m>k-1$，则要寻找的元素在 S_1 中，在 S_1 中继续执行步骤(1)。

② 如果 $m=k-1$，则 a 就是要找的元素。

③ 如果 $m<k-1$，则要寻找的值在 S_3 中，这时令 $k=k-(m+1)$，在 S_3 中继续执行步骤(1)。

3.3.4 算法实现

顺序统计分治法递归实现如下。

```
int OrderStatistic (int r[ ], int number){
    //数组 r[1] ～ r[n]存放查找元素集合
    n=r.length;
    //数组元素个数小于 number,第 number 小的元素不存在
    if (n<number) return 0;
    array s1=new array();
    array s3=new array();
    int k=random() * n+1;
    for(i=1;i<=n;i++){
```

```
            if(i==k) continue;
            if(r[i]<=r[k])
                s1.add(r[i]);
            else
                s3.add(r[i]);
        }
        if(s1.length>number-1)
            return OrderStatistic(s1,number);
        elesif(s1.length==number-1)
            return r[k];
        else
            return OrderStatistic(s3, number-s1.length-1);
    }
```

顺序统计分治法递归实现如下。

```
int OrderStatistic (int r[ ], int number){
    //数组 r[1] ～ r[n]存放查找元素集合
    n=r.length;
    //数组元素个数小于 number,第 number 小的元素不存在
    if (n<number) return 0;
    while(true){
        array s1=new array();
        array s3=new array();
        int k=random() * n+1;
        for(i=1;i<=n;i++){
            if(i==k) continue;
            if(r[i]<=r[k])
                s1.add(r[i]);
            else
                s3.add(r[i]);
        }
        if(s1.length>number-1){
            r=s1.copy();
            n=s1.length;
            continue;
        }
        elesif(s1.length==number-1)
            return r[k];
        else{
            r=s3.copy();
            n=s3.length;
            numner=number-s1.length-1;
        }
    }
}
```

3.3.5 算法复杂度分析

1. 平均时间复杂性分析

设集合中无相同的元素，a 是从集合 S 中选出的随机数，其是 S 中第 i 小元素，i 取值范围为 $1\sim n$。

假设 i 以相等的概率取 $1\sim n$ 中的一切值，则 i 取 $1\sim n$ 中任意值的概率都为 $p_i=\frac{1}{n}$。

用 a 把 S 划分成 S_1、S_2、S_3 共需 $n-1$ 次比较。

若 $i<k$，则在 S_3 中继续查找，在 $n-i$ 个元素范围内查找。

若 $i>k$，则在 S_1 中继续查找，在 $i-1$ 个元素上查找。

若 $i=k$，则 a 就是要找的元素。

因此算法的平均时间复杂性为：

$$T(n)\leqslant n-1+\max_k\left\{\frac{1}{n}\left[\sum_{i=1}^{k-1}T(n-i)+\sum_{i=k+1}^{n}T(i-1)\right]\right\}$$

$$=n-1+\max_k\left\{\frac{1}{n}\left[\sum_{i=n-k+1}^{n-1}T(i)+\sum_{i=k}^{n}T(i)\right]\right\} \tag{3-1}$$

可归纳证明：

$$T(n)\leqslant 4cn,\quad \text{其中，}c\text{ 为常数}$$

所以算法平均时间复杂性为 $O(n)$。

2. 最坏情况下时间复杂性分析

在最坏的情况下，每次 a 的选取均为最小值或最大值，这时每一轮操作只能排除一个元素。其时间复杂性等同于选择排序，即：

$$T(n)=O(n^2)$$

3.4 大整数乘法

3.4.1 问题描述

在科学计算特别是大规模科学计算中，浮点格式很可能无法满足计算精度的要求，为此要利用大整数来构造高精度的浮点数据类型，以保证最终输出结果的正确性。

大整数还被用来精确计算，例如，自然对数底数 e 或圆周率 π 等常数。

这时大整数无法在计算机硬件能直接表示的范围内进行处理。若用浮点数来表示它，则只能近似地表示它的大小，计算结果中的有效数字也受到限制。若要精确地表示大整数并在计算结果中要求精确地得到所有位数上的数字，就必须用软件的方法来实现大整数的算术运算。

问题：设有两个 n 位二进制整数 X 和 Y，实现二进制整数 X 和 Y 的乘法。

其中，整数超过计算机硬件能直接表示的范围。

3.4.2 问题分析

可以采用传统方法解决上述问题。

将乘数的每一位(由低位至高位)逐个去乘被乘数,每乘一次将乘积与原来的积相加,然后乘数和乘积移位一步,如此下去直至乘数的最高位运算完即得出结果。这样运算共需 n^2 次一位乘一位运算、$n(n-1)$次一位加一位运算和 n 次移位,总运算复杂性为 $O(n^2)$。

3.4.3　分治法求解问题

将 X 和 Y 各分为两段,每段的长为 $n/2$,X 分为 a 和 b 两段,Y 分为 c 和 d 两段,如图 3-8 所示。

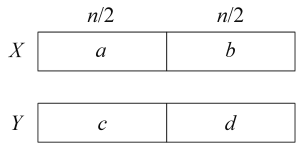

图 3-8　整数划分

由此得出,$X=2^{\frac{n}{2}}a+b$,$Y=2^{\frac{n}{2}}c+d$,X 和 Y 的乘积为:

$$XY=\left(2^{\frac{n}{2}}a+b\right)\left(2^{\frac{n}{2}}c+d\right)=2^n ac+2^{\frac{n}{2}}(ad+cb)+bd \tag{3-2}$$

使用该分治方法需要进行 4 次 $n/2$ 位的乘法,分析时间复杂度有:

$$\begin{cases}T(1)=1\\ T(n)=4T\left(\dfrac{n}{2}\right)+O(n)\end{cases} \tag{3-3}$$

根据时间复杂度主定理可以计算出:$T(n)=O(n^2)$。

3.4.4　改进的分治法

3.4.3 节中分治法的时间效率没有提升,主要问题是经过问题划分,子问题中需要 4 次 $n/2$ 位的乘法进行求解。所以算法的改进从减少 $n/2$ 位的乘法的角度来思考。

由 3.4.3 节可知:

$$XY=2^n ac+2^{\frac{n}{2}}(ad+cb)+bd \tag{3-4}$$

将式(3-4)进行变换:

$$\begin{aligned}XY&=2^n ac+2^{\frac{n}{2}}(ad-ac+cb-bd+ac+bd)+bd\\&=2^n ac+2^{\frac{n}{2}}((a-b)(d-c)+ac+bd)+bd\end{aligned} \tag{3-5}$$

如此一来,只需要进行 ac,$(a-b)(d-c)$,bd 三次乘法操作。

算法时间复杂度为:

$$\begin{cases}T(1)=1\\ T(n)=3T\left(\dfrac{n}{2}\right)+O(n)\end{cases} \tag{3-6}$$

$$T(n)=O(n^{\log_2 3})\approx O(n^{1.59}) \tag{3-7}$$

3.5　最大子数组问题

3.5.1　问题描述

给定一个数组 $x[1\cdots n]$,对于任意一对数组下标为 p、$q(p\leqslant q)$的非空子数组,其和记为 $S(p,q)=\sum_{i=p}^{q}x[i]$,求出 $S(p,q)$的最大值。

例如，给定数组[－2,1,－3,4,－1,2,1,－5,4]，其连续最大子数组和为：S(4,7)＝[4,－1,2,1]＝6。

3.5.2 算法分析

根据问题描述，数组元素可以为正数、负数和零。如果采用枚举法求解该问题，需要列出数组 x 的所有连续子数组，共有 M 个这样的子数组。

$$M=\sum_{i=1}^{n}(n-i+1)=\sum_{j=1}^{n}j=\frac{n(n+1)}{2}=O(n^2) \tag{3-8}$$

其中，i 为子数组长度。所以枚举法的时间复杂度为 $O(n^2)$。

3.5.3 分治法求解最大子数组问题

下面用分治法求解该问题。首先将数组 $x[1\cdots n]$ 进行划分，划分为 $x[1\cdots n/2]$ 和 $x\left[\frac{n}{2}+1\cdots n\right]$ 两个子数组。

假设已求解得出：$x[1\cdots n/2]$ 的最大子数组为 S_1，$x\left[\frac{n}{2}+1\cdots n\right]$ 的最大子数组为 S_2。下面要合并子问题的解为原问题的解，有以下 3 种情况。

(1) 数组[1…$n/2$]的最大子数组即为原问题的最大子数组。

(2) 数组 $x\left[\frac{n}{2}+1\cdots n\right]$ 的最大子数组即为原问题的最大子数组。

(3) 原问题的最大子数组跨越了子问题，假设这时解为 S_3。

所以原问题的解 S 为：

$$S=\max(S_1,S_2,S_3) \tag{3-9}$$

其中，S_1，S_2 已知，现在的问题是如何求 S_3。

可以采用逐步累加的方法：从中间位置开始，分别向左和向右两个方向进行操作，通过累加找到两个方向的最大和，分别为 l_max 和 r_max，因此存在于中间的最大和为 l_max 和(l_max+r_max)。

如图 3-9 所示，$l\_max=4$，$r\_max=-1+2+1=2$，$S_3=l\_max+r\_max=4+2=6$。又因为 $S_1=4$，$S_2=4$，所以：

$$S=\max(S_1,S_2,S_3)=\max(4,4,6)=6$$

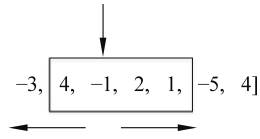

图 3-9 逐步累加求解 S_3

3.5.4 算法实现

```
int Divide(int *array,int p,int q){     //array为输入数组,p为数组起始位置,
                                        //q为数组结束位置
    if(p==q)                            //只有一个元素时,返回该元素
        return array[p];
    else{
        int m=(p+q)/2;
        int S1=MIN,S2=MIN,S3=MIN;       //MIN为数组array中最小值或足够小的值
        S1=Divide(array,p,m);           //左边和的最大值
        S2=Divide(array,m+1,q);         //右边和的最大值
        S3=MiddleMax(array,p,q,m);      //中间和的最大值
```

```
        //返回三个值中最大的一个
        if(S1>=S2 &&S1>=S3)
            S1;
        else if(S2>=S1 && S2>=S3)
            return S2;
        else
            return S3;
    }
}
```

求解 S_3：

```
int MiddleMax(int * array,int p,int q,int m){
    int l_max=MIN,r_max=MIN;        //分别用于记录左、右方向累加的最大和
    int i;
    int sum;                        //用于求和
    sum=0;
    for(i=m;i>=p;i--){              //中线开始向左寻找
        sum+=array[i];
        if(sum>l_max)
        l_max=sum;
    }
    sum=0;
    for(i=m+1;i<q;i++){             //中线开始向右寻找
        sum+=array[i];
        if(sum>r_max)
        r_max=sum;
    }
    return (l_max+r_max);           //返回左右之和
}
```

3.5.5　算法复杂性分析

求解原问题的时间复杂度等于求解 S_1、S_2、S_3 的时间之和。设求解原问题的时间复杂度为 $T(n)$，则求解 S_1、S_2 的时间复杂度均为 $T(n/2)$。

求解 S_3 的时间等于从中间元素向左右扩展元素的个数，在最坏的情况下，向左右扩展到全部数组元素，所以求解 S_3 的最坏情况下的时间复杂度为 n，所以：

$$T(n)=2T\left(\frac{n}{2}\right)+n,\quad \text{当 } n=1 \text{ 时},T(n)=1 \tag{3-10}$$

因此，

$$T(n)=O(n\log_2 n) \tag{3-11}$$

3.6　矩阵乘法

3.6.1　问题描述

矩阵乘法作为一种基本的数学运算，在计算机科学领域有着非常广泛的应用。同样地，在数据处理中，矩阵计算有着无比的优势，矩阵的计算广泛应用于大数据、机器学习等场景。

在数学中，矩阵(Matrix)是一个按照长方形阵列排列的复数或实数集合。而在计算机中，矩阵的存储可以由一个二维数组来进行存储。

矩阵相乘最重要的方法是一般矩阵乘积，只有在第一个矩阵 $\boldsymbol{A}$ 的列数和第二个矩阵 $\boldsymbol{B}$ 的行数相同时才有意义。矩阵乘法定义为：

设存在矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 分别为 $m\times p$ 和 $p\times n$ 的矩阵，存在尺度为 $m\times n$ 的矩阵 $\boldsymbol{C}$，$\boldsymbol{C}$ 中第 i 行、第 j 列的元素满足：

$$\boldsymbol{C}_{ij}=(\boldsymbol{AB})_{ij}=\sum_{k=1}^{p}a_{ik}b_{kj}=a_{i1}b_{1j}+a_{i2}b_{2j}+\cdots+a_{ip}b_{pj} \tag{3-12}$$

如果 $m=p=n$，这时有：

$$\boldsymbol{C}_{ij}=(\boldsymbol{AB})_{ij}=\sum_{k=1}^{n}a_{ik}b_{kj}=a_{i1}b_{1j}+a_{i2}b_{2j}+\cdots+a_{in}b_{nj} \tag{3-13}$$

本问题是：设 $\boldsymbol{A}$、$\boldsymbol{B}$ 是两个 $n\times n$ 的矩阵，求解 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$。

3.6.2 问题分析

通过式(3-13)可知，求解矩阵 $\boldsymbol{C}$ 的每个元素需要 n 次乘法和 $n-1$ 次加法，又由于矩阵 $\boldsymbol{C}$ 共有 $n\times n$ 个元素，因此两个 $n\times n$ 矩阵相乘需要 $O(n^3)$ 次乘法和 $O(n^3)$ 次加法。其时间复杂度是 $O(n^3)$ 的。

3.6.3 分治法求解矩阵相乘

假定 n 为 2 的幂，计算两个 $n\times n$ 矩阵相乘 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$ 时，采用分治法，先将每个 $n\times n$ 矩阵划分为四个 $\frac{n}{2}\times\frac{n}{2}$ 矩阵，再求矩阵的积。

$$\boldsymbol{A}=\begin{bmatrix}\boldsymbol{A}_{11} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22}\end{bmatrix},\quad \boldsymbol{B}=\begin{bmatrix}\boldsymbol{B}_{11} & \boldsymbol{B}_{12}\\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22}\end{bmatrix} \tag{3-14}$$

则：

$$\boldsymbol{C}=\begin{bmatrix}\boldsymbol{C}_{11} & \boldsymbol{C}_{12}\\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22}\end{bmatrix}=\begin{bmatrix}\boldsymbol{A}_{11} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22}\end{bmatrix}\times\begin{bmatrix}\boldsymbol{B}_{11} & \boldsymbol{B}_{12}\\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22}\end{bmatrix} \tag{3-15}$$

有：

$$\boldsymbol{C}_{11}=\boldsymbol{A}_{11}\times\boldsymbol{B}_{11}+\boldsymbol{A}_{12}\times\boldsymbol{B}_{21} \tag{3-16}$$

$$\boldsymbol{C}_{12}=\boldsymbol{A}_{11}\times\boldsymbol{B}_{12}+\boldsymbol{A}_{12}\times\boldsymbol{B}_{22} \tag{3-17}$$

$$\boldsymbol{C}_{21}=\boldsymbol{A}_{21}\times\boldsymbol{B}_{11}+\boldsymbol{A}_{22}\times\boldsymbol{B}_{21} \tag{3-18}$$

$$\boldsymbol{C}_{22}=\boldsymbol{A}_{21}\times\boldsymbol{B}_{12}+\boldsymbol{A}_{22}\times\boldsymbol{B}_{22} \tag{3-19}$$

每个公式对应两对 $\frac{n}{2}\times\frac{n}{2}$ 矩阵的乘法及一个 $\frac{n}{2}\times\frac{n}{2}$ 积的加法。其中，加法操作次数为 $4\times\frac{n^2}{4}$。

算法的时间复杂度为：

$$T(n)=\begin{cases}b & n\leqslant 2\\ 8T\left(\frac{n}{2}\right)+cn^2 & n>2\end{cases} \tag{3-20}$$

其中，b 为常数，$8T\left(\frac{n}{2}\right)$是 8 个$\frac{n}{2}\times\frac{n}{2}$相乘，$cn^2$ 是加法的运算时间。因此

$$T(n)=O(n^3) \tag{3-21}$$

我们可以利用这些公式设计的原理设计一个简单的递归分治算法。

3.6.4　Strassen 算法实现矩阵乘法

Strassen 算法也是基于分治思想的一种实现矩阵乘法的算法，其基本目标是减少子问题的矩阵相乘个数，该方法在大规模数值计算中有着广泛的应用。Strassen 算法是由 Volker Strassen 提出的，在矩阵较大时运行速度快于传统算法的矩阵相乘算法。

Strassen 算法包含以下 4 个步骤。

(1) 按式(3-15)将输入矩阵 $\boldsymbol{A}$、$\boldsymbol{B}$ 和输出矩阵 $\boldsymbol{C}$ 分解为$\frac{n}{2}\times\frac{n}{2}$的子矩阵。

(2) 构造和计算中间变量。

$$\boldsymbol{P}=(\boldsymbol{A}_{11}+\boldsymbol{A}_{22})(\boldsymbol{B}_{11}+\boldsymbol{B}_{22}) \tag{3-22}$$

$$\boldsymbol{Q}=(\boldsymbol{A}_{21}+\boldsymbol{A}_{22})\boldsymbol{B}_{11} \tag{3-23}$$

$$\boldsymbol{R}=\boldsymbol{A}_{11}(\boldsymbol{B}_{12}-\boldsymbol{B}_{22}) \tag{3-24}$$

$$\boldsymbol{S}=\boldsymbol{A}_{22}(\boldsymbol{B}_{21}-\boldsymbol{B}_{11}) \tag{3-25}$$

$$\boldsymbol{T}=(\boldsymbol{A}_{11}+\boldsymbol{A}_{12})\boldsymbol{B}_{22} \tag{3-26}$$

$$\boldsymbol{U}=(\boldsymbol{A}_{21}-\boldsymbol{A}_{11})(\boldsymbol{B}_{11}+\boldsymbol{B}_{12}) \tag{3-27}$$

$$\boldsymbol{V}=(\boldsymbol{A}_{12}-\boldsymbol{A}_{22})(\boldsymbol{B}_{21}+\boldsymbol{B}_{22}) \tag{3-28}$$

(3) 由中间变量构造问题的解。

$$\boldsymbol{C}_{11}=\boldsymbol{P}+\boldsymbol{S}+\boldsymbol{T}+\boldsymbol{V} \tag{3-29}$$

$$\boldsymbol{C}_{12}=\boldsymbol{R}+\boldsymbol{T} \tag{3-30}$$

$$\boldsymbol{C}_{21}=\boldsymbol{Q}+\boldsymbol{S} \tag{3-31}$$

$$\boldsymbol{C}_{22}=\boldsymbol{P}+\boldsymbol{R}+\boldsymbol{Q}+\boldsymbol{U} \tag{3-32}$$

(4) 时间复杂度分析。Strassen 算法子问题求解共计需要 7 次$\frac{n}{2}\times\frac{n}{2}$相乘、$O(n^2)$次加法，其时间复杂度为：

$$T(n)=\begin{cases}O(1), & n=1\\ 7T\left(\frac{n}{2}\right)+O(n^2), & n>1\end{cases} \tag{3-33}$$

计算可得：

$$T(n)=O\left(n^{\log_2^7}\right)$$

3.7　递归式求解

3.7.1　问题描述

用分治法求解问题，分析其算法时间复杂度时，经常用递归式进行表示。例如，对 n 个元素进行归并排序，归并排序采取分治策略，将待排序的 n 个元素分为两组，当组内元素无

序时则继续分组，若组内元素有序，将两组合并，直到所有待排序元素有序。讨论归并排序的时间复杂度，每次分组都将问题的规模缩小为原来的$\frac{1}{2}$，且一次合并的时间复杂度为$O(n)$，则可以得到下面的递归式。

$$T(n)=2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+cn \tag{3-34}$$

可以用更广义的方式来定义递归式：

$$T(n)=aT\left(\frac{n}{b}\right)+f(n) \tag{3-35}$$

其中，$a\geqslant 1$，$b>1$，表示将规模为n的问题分解为a个规模为$\frac{n}{b}$的子问题，求解每个子问题花费时间$T\left(\frac{n}{b}\right)$，$f(n)$是渐近正函数，包含问题分解和子问题合并所需的代价。

递归式描述了一种算法的运行时间，递归式的求解问题不是给定n的值来求解$T(n)$的值的过程，而是计算求解$T(n)$所要花费的时间代价的问题(解是关于n的函数)。

3.7.2 代入法求解递归式

代入法是一种假设演绎方法，即先根据经验猜测解的形式，再使用数学归纳法证明解的正确性。

由于我们知道归并排序的时间复杂度是$O(n\lg n)$，所以假设递归式(3-34)的解有上界$T(n)=O(n\lg n)$，即选择常数$c>0$，有$T(n)\leqslant cn\lg n$，根据数学归纳法，假设此上界对所有正数$m<n$都成立，对于$m=\left\lfloor\frac{n}{2}\right\rfloor$也成立，有

$$T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)\leqslant c\left\lfloor\frac{n}{2}\right\rfloor\lg\left(\left\lfloor\frac{n}{2}\right\rfloor\right) \tag{3-36}$$

将公式(3-36)代入递归式(3-34)中，有

$$\begin{aligned}T(n)&\leqslant 2c\left\lfloor\frac{n}{2}\right\rfloor\lg\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+n\leqslant cn\lg\left(\frac{n}{2}\right)+n\\&=cn\lg n-cn\lg 2+n\\&=cn\lg n-cn+n\leqslant cn\lg n\quad \text{s.t.}(c\geqslant 1)\end{aligned} \tag{3-37}$$

在上述证明中，仅需保证当n足够大时假设成立即可，渐进函数仅要求我们对$n\geqslant n_0$时证明$T(n)\leqslant cn\lg n$，其中，n_0是可以选择的常数。事实上，对于边界条件$T(1)$我们的假设不成立。

代入法需要我们对解的形式给出正确的猜测，但是并不存在通用的方法来猜测递归式的正确解，经常需要依靠经验。例如，对于递归式

$$T(n)=2T\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right)+n \tag{3-38}$$

形式上与递归式(3-34)非常相似，只是在等式右边T的参数中多加了1。当n足够大时$\left\lfloor\frac{n}{2}\right\rfloor$与$\left\lfloor\frac{n}{2}\right\rfloor+1$基本没有区别，所以我们猜测$T(n)=O(n\lg n)$，通过数学归纳法仍然可以证明解是正确的。

在实际中，可以首先证明递归式较为宽松的上界和下界，然后不断缩小解的范围，直到收敛到渐近紧确界 $T(n)=O(n\lg n)$。

3.7.3　递归树法求解递归式

递归树可以通过建立递归调用与结点之间的联系来很好地展现递归过程，是描述递归算法的良好工具。

使用代入法可以简洁地证明一个解是递归式的正确解，但有些时候递归式的正确解往往难以猜测，这时可以通过绘制递归树来生成一个好的猜测，然后使用代入法来验证猜测的正确性。

递归树是迭代过程的一种图像表述。递归树往往被用于求解递归方程，它的求解表示比一般的迭代会更加简洁与清晰。递归树上所有项恰好是迭代之后产生和式中的项，递归树的生成过程与迭代过程一致，对递归树上的项求和就是迭代后方程的解。

在递归树中，每一个结点表示一个单一子问题的代价，子问题对应某次递归函数调用，我们将递归树中每层的代价求和，得到每一层的代价，将所有层的代价求和，得到递归调用的总代价。其生成过程如下。

(1) 初始：递归树只有根结点，其值为 $W(n)$。

(2) 不断继续下述过程。

① 将函数项叶结点的迭代式 $W(m)$表示成二层子树。

② 用该子树替换该叶结点。

(3) 继续递归树的生成，直至树中无函数项(只有初值)为止。

下面绘制递归式(3-34)对应的递归树。

图 3-10 绘制了一次递归调用对应的递归树，顶层结点 cn 代表递归调用顶层的代价，并将规模为 n 的问题分解为两个规模为 $\frac{n}{2}$ 的问题。图 3-11 中绘制了继续进行递归调用得到的递归树。

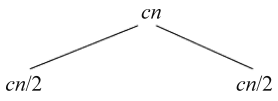

图 3-10　顶层递归树

图 3-11 将图 3-10 中代价为 $T\left(\frac{n}{2}\right)$ 的结点进一步扩展，问题被进一步分解，递归树前两层的代价之和均为 cn，将递归树的所有结点不断扩展，可以得到一棵完整的递归树，如图 3-12 所示。

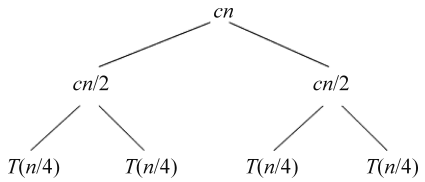

图 3-11　二层递归树

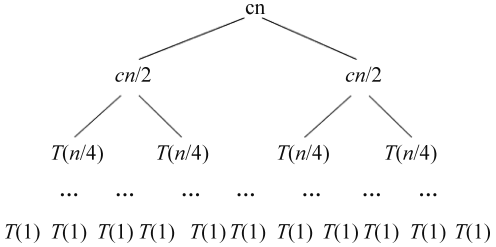

图 3-12　问题递归树

随着递归调用的进行，问题的规模不断缩小，最终子问题的规模变为 1，即原问题被分

解为 n 个规模为 1 的子问题。每一层的代价之和均为 cn，深度为 i 的结点对应问题规模为 $\frac{n}{2^i}(i=0,1,2,\cdots,\lg n)$。当 $\frac{n}{2^i}=1$ 时，有 $i=\lg n$，因此递归树有 $\lg n+1$ 层，整棵树的代价为 $cn\lg n$。

3.7.4 主方法求解递归式

主方法可以直接求出绝大多数递归式的解而无须进行复杂的运算，主方法依赖于下面的主定理。

令 $a\geqslant 1$ 和 $b>1$ 是常数，$f(n)$是一个函数，$T(n)$是定义在非负整数上的递归式：

$$T(n)=aT\left(\frac{n}{b}\right)+f(n) \tag{3-39}$$

其中，我们将 $\frac{n}{b}$ 解释为 $\left\lfloor\frac{n}{b}\right\rfloor$ 和 $\left\lceil\frac{n}{b}\right\rceil$，那么 $T(n)$有如下的渐近界。

(1) 若存在常数 $\varepsilon>0$ 满足 $f(n)=O\left(n^{\log_b^a-\varepsilon}\right)$，则 $T(n)=\Theta\left(n^{\log_b^a}\right)$。

(2) 若 $f(n)=\Theta\left(n^{\log_b^a}\right)$，则 $T(n)=\Theta\left(n^{\log_b^a}\lg n\right)$。

(3) 若存在常数 $\varepsilon>0$ 满足 $f(n)=\Omega\left(n^{\log_b^a+\varepsilon}\right)$，且对某个常数 $c<1$ 和足够大的 n 有 $af\left(\frac{n}{b}\right)\leqslant cf(n)$，则 $T(n)=\Theta(f(n))$。

主方法将 $T(n)$的解分为三种情况，直观上理解，我们将函数 $f(n)$和函数 $n^{\log_b^a}$ 进行比较，两个函数较大者决定了递归式的解。若 $n^{\log_b^a}$ 更大，则属于情况(1)，解为 $T(n)=\Theta\left(n^{\log_b^a}\right)$，若函数 $f(n)$更大，则属于情况(3)，解为 $T(n)=\Theta(f(n))$。若两个函数大小相当，则属于情况(2)，解需要乘以一个对数因子，解为 $T(n)=\Theta\left(n^{\log_b^a}\lg n\right)$。

除此之外，在第一种情况中，$f(n)$小于 $n^{\log_b^a}$ 是多项式意义上的小于，即 $f(n)$渐近小于 $n^{\log_b^a}$，还需要相差一个因子 n^{ε}。在第三种情况中，还需要满足条件 $af\left(\frac{n}{b}\right)\leqslant cf(n)$。

主方法并不适用于全部的递归式，$f(n)$可能小于 $n^{\log_b^a}$，但不是多项式意义上的小于，$f(n)$可能大于 $n^{\log_b^a}$，但同样不是多项式意义上的大于。所以情况(1)和情况(2)之间存在间隙，情况(2)和情况(3)之间同样存在间隙，如果函数 $f(n)$落在两个间隙中，或者 $f(n)$不满足 $af\left(\frac{n}{b}\right)\leqslant cf(n)$，就不能使用主方法求解。

下面通过具体例子来分析主方法的实际应用。

(1) $T(n)=9T\left(\frac{n}{3}\right)+n$，其中，$a=9$，$b=3$，$f(n)=n$，因此求得 $n^{\log_b^a}=n^2$，有 $\varepsilon=1$ 使得 $f(n)=O\left(n^{\log_b^a-\varepsilon}\right)$，满足主定理情况(1)，从而得到解 $T(n)=\Theta(n^2)$。

(2) $T(n)=T\left(\frac{2n}{3}\right)+1$，其中，$a=1$，$b=\frac{2}{3}$，$f(n)=1$，因此求得 $n^{\log_b^a}=n^0=1$，$f(n)=\Theta(n^{\log_b^a})=\Theta(1)$，满足主定理情况(2)，从而得到解 $T(n)=\Theta(\lg n)$。

(3) $T(n)=3T\left(\frac{n}{4}\right)+n\lg n$，其中，$a=3$，$b=4$，$f(n)=n\lg n$，因此求得 $n^{\log_b^a}=n^{\log_4^3}$，由于 $f(n)=\Omega(n^{\log_4^3+\varepsilon})$，其中，$\varepsilon=1-\log_4^3$。且当 n 足够大时 $af\left(\frac{n}{b}\right)=\frac{3n}{4}\lg\left(\frac{n}{4}\right)\leqslant\frac{3}{4}n\lg n=\frac{3}{4}f(n)$，则当 $c=\frac{3}{4}$ 时，满足主定理情况(3)，可以得到递归式的解为 $T(n)=\Theta(n\lg n)$。

(4) $T(n)=2T\left(\frac{n}{2}\right)+n\lg n$，其中，$a=2$，$b=2$，$f(n)=n\lg n$，因此求得 $n^{\log_b^a}=n^{\log_2^2}=n$，但对于任意的 $\varepsilon>0$ 有 $\frac{f(n)}{n^{\log_b^a}}=\lg n<n^{\varepsilon}$，因此不存在常数 $\varepsilon>0$ 满足 $f(n)=\Omega\left(n^{\log_b^a+\varepsilon}\right)$，故此递归式不能使用主方法解决。

3.8　证明主定理

3.7 节中采用三种方法求解递归式，其中包括主方法。主方法求解递归式依赖于主定理，本节将证明主定理。

3.8.1　主定理

在此，重复描述 3.7.4 节中的主定理内容。

令 $a\geqslant1$ 和 $b>1$ 是常数，$f(n)$是一个函数，$T(n)$是定义在非负整数上的递归式：

$$T(n)=aT\left(\frac{n}{b}\right)+f(n) \tag{3-40}$$

其中，n 是问题规模大小；a 是原问题的子问题个数；$\frac{n}{b}$是每个子问题的大小，这里假设每个子问题有相同的规模大小；$f(n)$是将原问题分解成子问题和将子问题的解合并成原问题的解的时间。

我们将$\frac{n}{b}$解释为$\left\lfloor\frac{n}{b}\right\rfloor$和$\left\lceil\frac{n}{b}\right\rceil$，那么 $T(n)$有如下的渐进界。

(1) 若存在常数 $\varepsilon>0$ 满足 $f(n)=O\left(n^{\log_b^a-\varepsilon}\right)$，则 $T(n)=\Theta\left(n^{\log_b^a}\right)$。

(2) 若 $f(n)=\Theta\left(n^{\log_b^a}\right)$，则 $T(n)=\Theta\left(n^{\log_b^a}\lg n\right)$。

(3) 若存在常数 $\varepsilon>0$ 满足 $f(n)=\Omega\left(n^{\log_b^a+\varepsilon}\right)$，且对某个常数 $c<1$ 和足够大的 n 有 $af\left(\frac{n}{b}\right)\leqslant cf(n)$，则 $T(n)=\Theta(f(n))$。

3.8.2　主定理递归树表示

主定理可以用 3.7.3 节中的递归树进行表示，如图 3-13 所示。

树的深度为 $\log_b n$，加上根结点共计 $\log_b n+1$ 层，叶子结点数为 $a^{\log_b n}$。第 i 层共有 a^i 个子问题，每个问题规模为$\frac{n}{b^i}$。

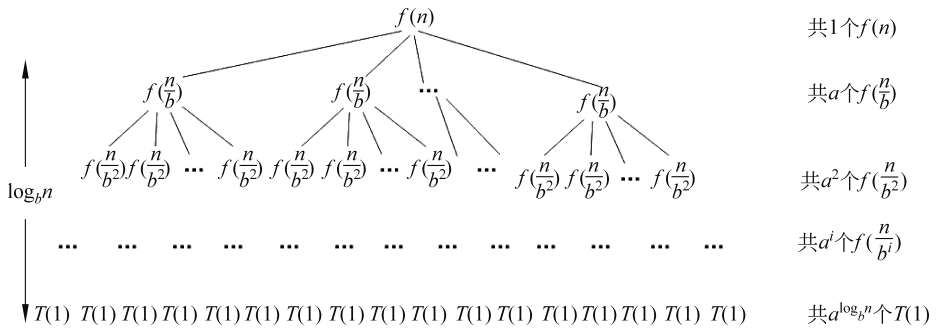

图 3-13　主定理递归树

3.8.3　主定理证明

为了便于分析假定 n 是 b 的幂次，即 $n=b^m$，m 为正整数，$m=\log_b n$。

1. 对 n 为 b 的幂主定理证明

假设 n 是 b 的幂次，即 $n=b^m$，m 为正整数，$m=\log_b n$。

引理 3.1：令 $a\geqslant 1$，$b>1$ 是常数，$f(n)$是一个定义在 b 的幂上的非负函数。$T(n)$是定义在 b 的幂上的递归式：

$$T(n)=\begin{cases}\Theta(1), & n=1\\ aT\left(\dfrac{n}{b}\right)+f(n), & n=b^i\end{cases} \tag{3-41}$$

其中，i 是正整数，那么

$$T(n)=\Theta(n^{\log_b a})+\sum_{j=0}^{\log_b n-1}a^j f\left(\frac{a}{b^j}\right) \tag{3-42}$$

证明：

由图 3-13 可知，在层次 i 的运算量为 $f\left(\dfrac{n}{b^i}\right)\times a^i$，则 $T(n)$为：

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

等式成立。

注：$a^{\log_b n}=n^{\log_b a}$ 的证明可根据对数运算公式，也可直接在等式两端求以 b 为底的对数进行证明。

由公式(3-42)可知，$T(n)$由两部分组成，$\Theta(n^{\log_b a})$为求解叶结点的代价，$\sum_{j=0}^{\log_b n-1}a^j f\left(\dfrac{a}{b^j}\right)$为分解子问题和合并子问题解的代价。

引理 3.2：令 $a\geqslant 1$，$b>1$ 是常数，$f(n)$是一个定义在 b 的幂上的非负函数。$g(n)$是定义在 b 的幂上的递归式：

$$g(n)=\sum_{j=0}^{\log_b n-1}a^j\times f\left(\frac{n}{b^j}\right) \tag{3-43}$$

对 b 的幂，$g(n)$有如下渐进界。

(1) 若对某个常数 $\varepsilon>0$ 有 $f(n)=O\left(n^{\log_b^a-\varepsilon}\right)$，则 $g(n)=O\left(n^{\log_b^a}\right)$。

(2) 若 $f(n)=\Theta\left(n^{\log_b^a}\right)$，则 $g(n)=\Theta\left(n^{\log_b^a}\lg n\right)$。

(3) 若对某个常数 $c<1$ 和所有足够大的 n 有 $af\left(\frac{n}{b}\right)\leqslant cf(n)$，则 $g(n)=\Theta(f(n))$。

证明：

(1) 对于情况 1，有 $f(n)=O\left(n^{\log_b^a-\varepsilon}\right)$，则：

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

因此，

$$\begin{aligned}g(n)&=\sum_{j=0}^{\log_b n-1}a^j\times f\left(\frac{n}{b^j}\right)=\sum_{j=0}^{\log_b n-1}a^j\times O\left(\left(\frac{n}{b^j}\right)^{\log_b^a-\varepsilon}\right)\\&=O\left(n^{\log_b^a-\varepsilon}\times\sum_{j=0}^{\log_b n-1}\left(\frac{ab^\varepsilon}{b^{\log_b^a}}\right)^j\right)=O\left(n^{\log_b^a-\varepsilon}\times\sum_{j=0}^{\log_b n-1}(b^\varepsilon)^j\right)\\&=O\left(n^{\log_b^a-\varepsilon}\times\left(\frac{b^{\varepsilon\log_b n}-1}{b^\varepsilon-1}\right)\right)=O\left(n^{\log_b^a-\varepsilon}\times\left(\frac{n^\varepsilon-1}{b^\varepsilon-1}\right)\right)\end{aligned}\tag{3-44}$$

由于 b 和 ε 是常数，因此

$$g(n)=O\left(n^{\log_b^a-\varepsilon}\times\left(\frac{n^\varepsilon-1}{b^\varepsilon-1}\right)\right)=O\left(n^{\log_b^a}\right)\tag{3-45}$$

因此对情况 1 成立。

(2) 对于情况 2，假定 $f(n)=\Theta\left(n^{\log_b^a}\right)$，有：

$$f\left(\frac{n}{b^j}\right)=\Theta\left(\left(\frac{n}{b^j}\right)^{\log_b^a}\right)\tag{3-46}$$

代入公式(3-43)，得到：

$$\begin{aligned}g(n)&=\sum_{j=0}^{\log_b n-1}a^j\times f\left(\frac{n}{b^j}\right)=\sum_{j=0}^{\log_b n-1}a^j\times\Theta\left(\left(\frac{n}{b^j}\right)^{\log_b^a}\right)\\&=\Theta\left(\sum_{j=0}^{\log_b n-1}a^j\times\left(\frac{n}{b^j}\right)^{\log_b^a}\right)=\Theta\left(n^{\log_b^a}\times\sum_{j=0}^{\log_b n-1}a^j\times\left(\frac{1}{b^{\log_b^a}}\right)^j\right)\\&=\Theta\left(n^{\log_b^a}\times\sum_{j=0}^{\log_b n-1}1\right)=\Theta\left(n^{\log_b^a}\times\log_b n\right)=\Theta\left(n^{\log_b^a}\times\lg n\right)\end{aligned}\tag{3-47}$$

因此对情况 2 成立。

(3) 对于情况 3，由于 $af\left(\frac{n}{b}\right)\leqslant cf(n)$，所以 $f\left(\frac{n}{b}\right)\leqslant\frac{c}{a}f(n)$，进而 $f\left(\frac{n}{b^j}\right)\leqslant\left(\frac{c}{a}\right)^j f(n)$，代入公式(3-43)，得到：

$$\begin{aligned}g(n)&=\sum_{j=0}^{\log_b n-1}a^j\times f\left(\frac{n}{b^j}\right)\leqslant\sum_{j=0}^{\log_b n-1}a^j\times\left(\frac{c}{a}\right)^j f(n)+O(1)\\&=f(n)\sum_{j=0}^{\log_b n-1}c^j+O(1)=f(n)\frac{1}{1-c}+O(1)=O(f(n))\end{aligned}\tag{3-48}$$

因此对情况 3 成立。