第2章　递归与分治策略

学习要点

- 理解递归的概念
- 掌握设计有效算法的分治策略
- 通过下面的范例学习分治策略设计技巧

(1)二分搜索技术；

(2)大整数乘法；

(3)棋盘覆盖；

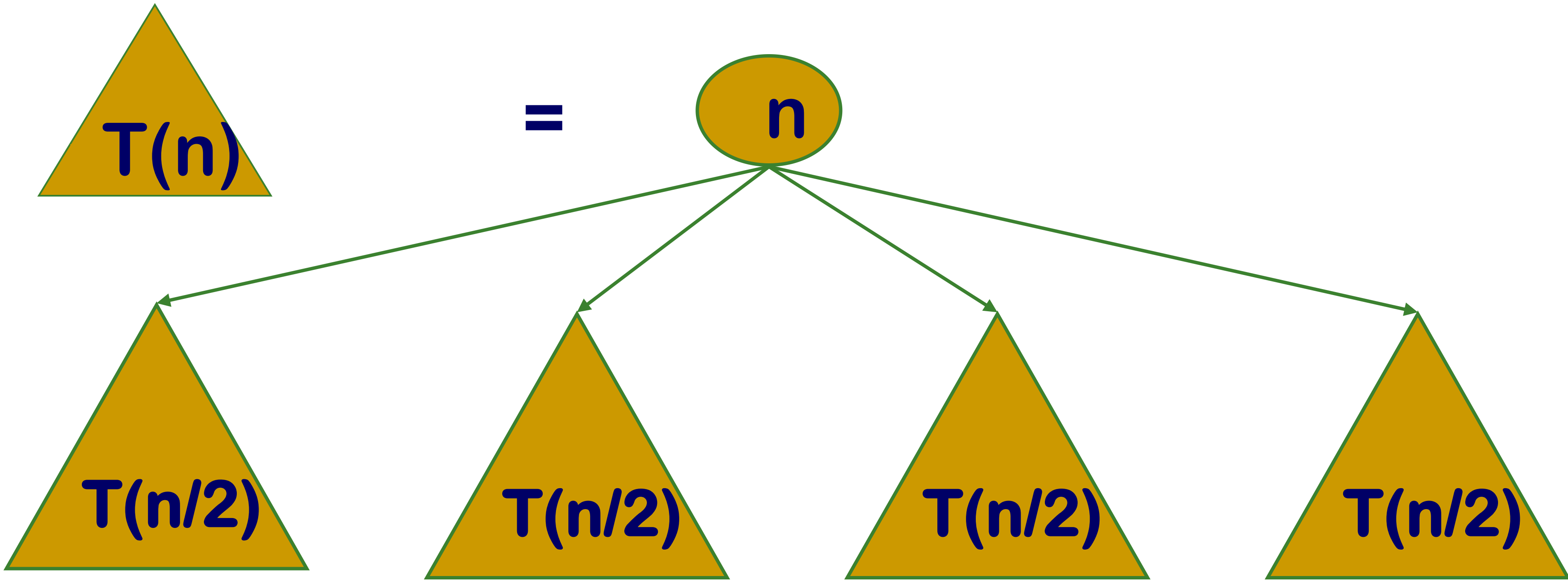
(4)合并排序和快速排序；

(5)线性时间选择；

(6)最接近点对问题；

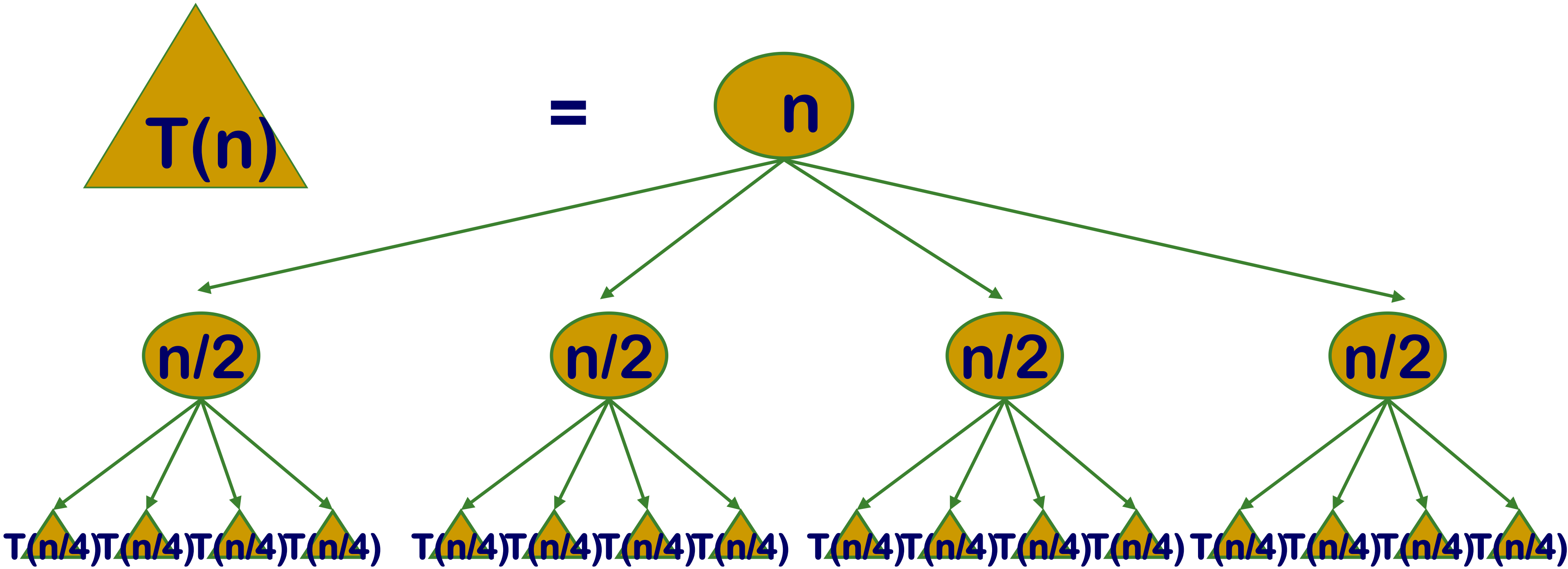
算法总体思想

对这k个子问题分别求解。如果子问题的规模仍然不够小，则再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。



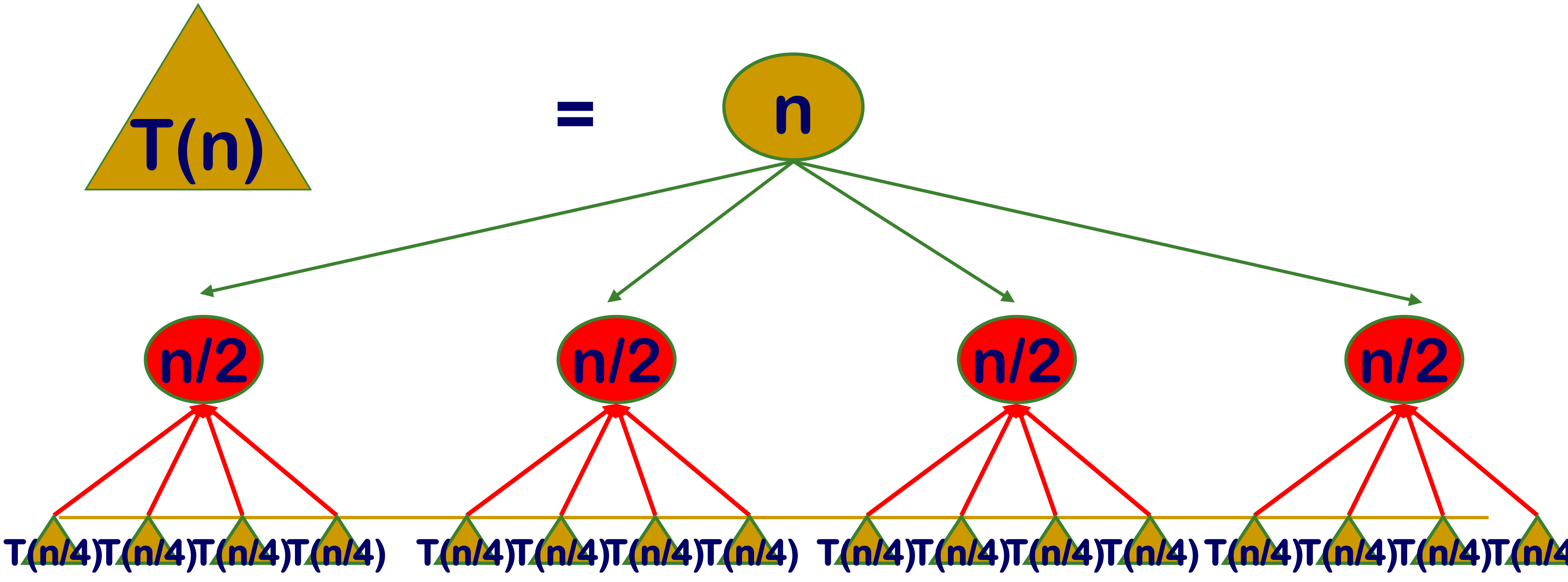
算法总体思想

将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



算法总体思想

将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



算法总体思想

将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。

分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的<u>相同问题</u>，以便各个击破，分而治之。(Divide and conqure)

2.1 递归的概念

- 直接或间接地调用自身的算法称为**递归算法**。用函数自身给出定义的函数称为**递归函数**。
- 由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。
- 分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

2.1　递归的概念

2.1 递归的概念

例1 阶乘函数

阶乘函数可递归地定义为：

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

边界条件

递归方程

边界条件与递归方程是递归函数的二个要素，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。

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

2.1　递归的概念

例2　Fibonacci数列

无穷数列1，1，2，3，5，8，13，21，34，55，……，称为Fibonacci数列。它可以递归地定义为：

边界条件

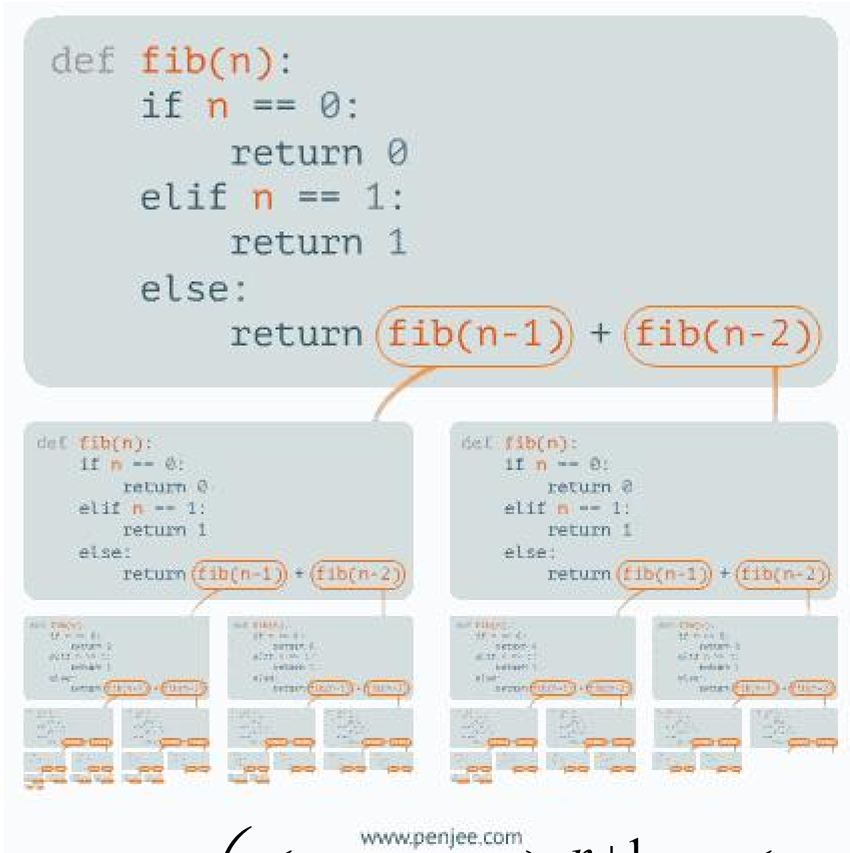
$$F(n)=\begin{cases}1 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

递归方程

第n个Fibonacci数可递归地计算如下：

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

例2 Fibonacci数列



$$F(n)=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

2.1　递归的概念

例3　Ackerman函数

Ackerman函数$A(n,m)$定义如下：

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

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**。

2.1　递归的概念

例3　Ackerman函数

前2例中的函数都可以找到相应的非递归方式定义：

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$$

$$F(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

本例中的Ackerman函数却无法找到非递归的定义。

2.1 递归的概念

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，$A(n,1)=A(A(n\text{-}1,1),0)=A(n\text{-}1,1)+2$，和$A(1,1)=2$故 $A(n,1)=2n$

$m=2$时，$A(n,2)=A(A(n\text{-}1,2),1)=2A(n\text{-}1,2)$，和 $A(1,2)=A(A(0,2),1)=A(1,1)=2$，故$A(n,2)= 2\text{^}n$ 。

$m=3$时，类似的可以推出

$$\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{n}$$

$m=4$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。

2.1 递归的概念

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。

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)由(r_1)perm(R_1)，(r_2)perm(R_2)，…，(r_n)perm(R_n)构成。

2.1 递归的概念

例5 整数划分问题

将正整数n表示成一系列正整数之和：$n=n_1+n_2+\ldots+n_k$，其中$n_1 \geq n_2 \geq \ldots \geq n_k \geq 1$，$k \geq 1$。

正整数n的这种表示称为正整数n的划分。求正整数n的不同划分个数。

例如正整数6有如下11种不同的划分：

6；

5+1；

4+2，4+1+1；

3+3，3+2+1，3+1+1+1；

2+2+2，2+2+1+1，2+1+1+1+1；

1+1+1+1+1+1。

2.1 递归的概念

例5 整数划分问题

前面的几个例子中，问题本身都具有比较明显的递归关系，因而容易用递归函数直接求解。

在本例中，如果设 $p(n)$ 为正整数 n 的划分数，则难以找到递归关系，因此考虑增加一个自变量：

将最大加数 n_1 不大于 m 的划分个数记作 $q(n,m)$。可以建立 $q(n,m)$的递归关系。

(3) $q(n,n)=1+q(n,n-1)$;

正整数n的划分由$n_1=n$的划分和$n_1\leq n-1$的划分组成。

(4) $q(n,m)=q(n,m-1)+q(n-m,m),n>m>1$;

正整数n的最大加数n_1不大于m的划分由$n_1=m$的划分和$n_1\leq n-1$ 的划分组成。

(3)当n=m时，q(n,n),根据划分中是否包含n，可以分为两种情况：

(a)划分中包含n的情况，只有一个即{n}；

(b)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);

6；

5+1；

4+2，4+1+1；

3+3，3+2+1，3+1+1+1；

2+2+2，2+2+1+1，2+1+1+1+1；

1+1+1+1+1+1。

(4)当n>m时，根据划分中是否包含最大值m，可以分为两种情况：

(a)划分中包含m的情况，即{m, {x1,x2,…xi}}, 其中{x1,x2,…xi} 的和为n-m，因此这情况下为f(n-m,m)

(b)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 $f(n, m) = f(n-m, m)+f(n,m-1)$;

6；

5+1；

4+2，4+1+1；

3+3，3+2+1，3+1+1+1；

2+2+2，2+2+1+1，2+1+1+1+1；

1+1+1+1+1+1。

2.1　递归的概念

例5　整数划分问题

设p(n)为正整数n的划分数，则难以找到递归关系，因此考虑增加一个自变量：将最大加数n_1不大于m的划分个数记作q(n,m)。可以建立q(n,m)的如下递归关系。

$$q(n,m)=\begin{cases} 1 & n=1,m=1 \\ q(n,n) & n<m \\ 1+q(n,n-1) & n=m \\ q(n,m-1)+q(n-m,m) & n>m>1 \end{cases}$$

正整数n的划分数p(n)=q(n,n)。

2.1 递归的概念

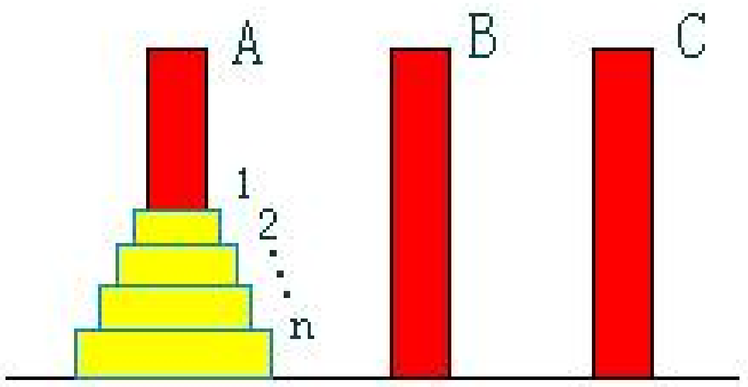
例6 Hanoi塔问题

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：

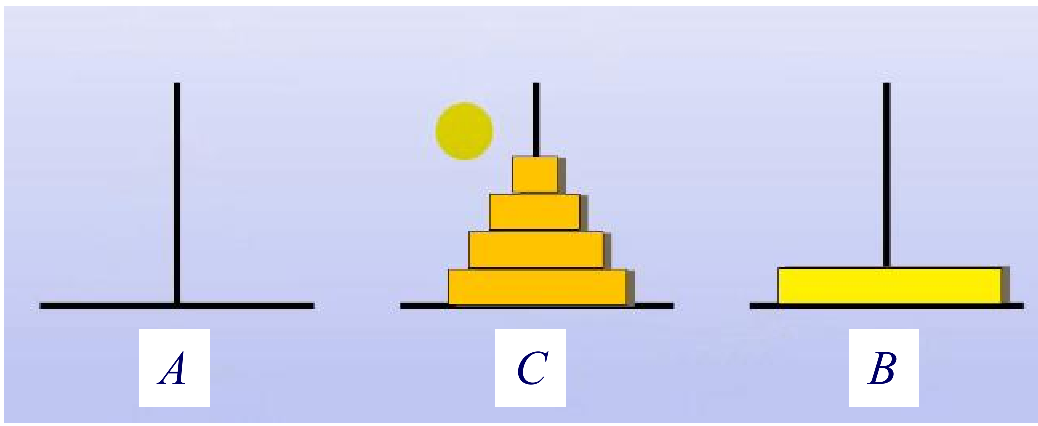
规则1：每次只能移动1个圆盘；

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。



以5个盘子为例，进行分析。
先将上面4个盘子看成一个整体，那么第一步，需要借助B柱子，将上面4个盘子放在C柱子上，然后将A柱子最底的第5个盘子放到B柱子上，如下图所示：

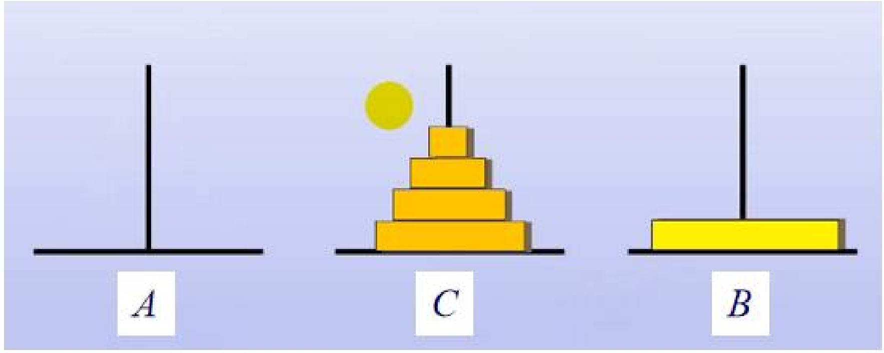


问题就依赖于C柱子上的4个盘子如何移动，也就是n-1个盘子如何从C移动到B上面

2.1 递归的概念

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);
     }
  }
```



显然，我们可以选择盘子的数量 n 作为输入规模的一个指标，盘子的移动也可以作为该算法的基本操作。

移动的次数M(n)有下列递推等式：

$M(n)=M(n-1)+1+M(n-1)=2M(n-1)+1, n>1$

$M(1)=1$，

因此，对于移动次数M(n)我们建立了下面的递推关系：

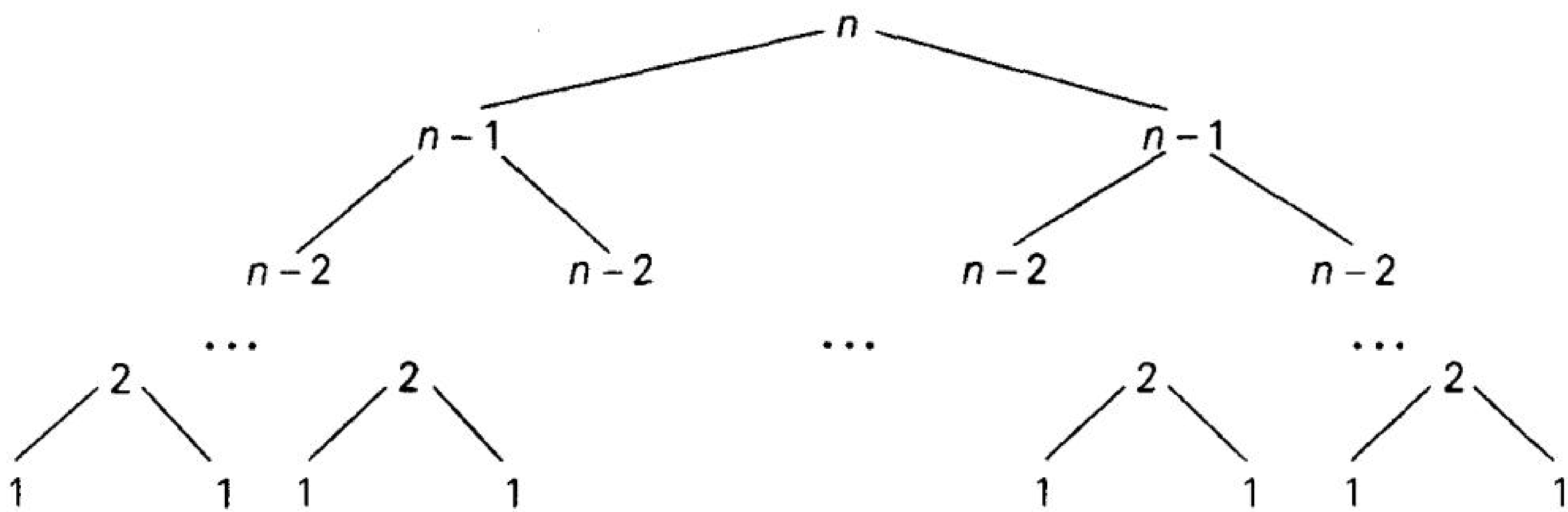
$M(n)=2^iM(n-i)+2^{i-1}+...+2+1=2^iM(n-i)+2^i-1$

令$i=n-1$

$M(n)=2^iM(n-(n-1))+2^{n-1}-1=2^n-1$

我们应该谨慎使用递归算法，因为它们的简洁可能会掩盖它们的低效率

如果一个递归算法会不止一次地调用它本身，处于分析的目的，构造一棵它的递归调用树是很有用的



通过计算树中的节点数，我们可以得到汉诺塔算法所做调用的全部次数：

$$M(n) = \sum_{l=0}^{n-1} 2^{l} = 2^{n} - 1$$

递归程序执行过程...

- 函数调用机制

  函数调用为了在调用函数后能正确地使用函数参数和正确地返回到调用时的位置,必须将一些数据存储在栈中。这个处理过程称为函数调用机制。

- 函数调用时,需要做下面一些工作:

(1) 建立被调函数的栈空间

(2) 保护调用函数的运行状态和返回地址

(3) 保护函数传递的参数

(4) 将控制转交被调函数

…递归程序执行过程…

- 实际上函数被调用时执行的代码是函数的一个副本,与调用函数的代码无关;
- 当一个函数被调用两次,则函数就会有两个副本在内存中运行,每个副本都有自己的栈空间,且与调用函数的栈空间不同,因此不会相互影响;
- 这种调用机制决定了函数是可以递归调用的。

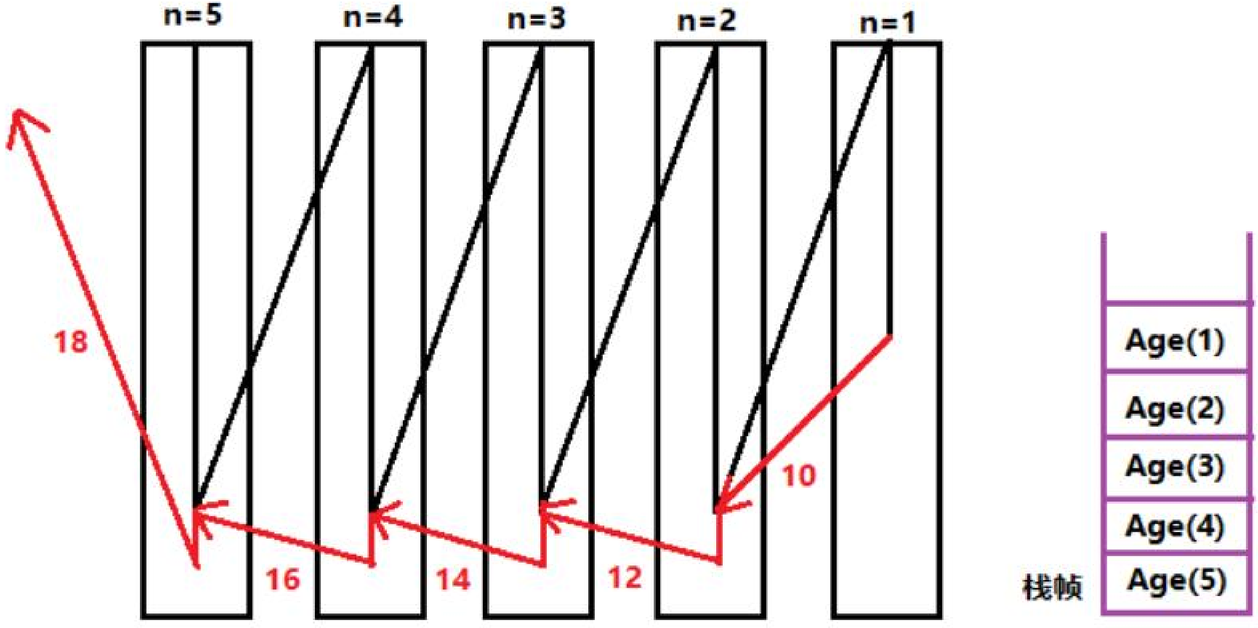
…递归程序执行过程…

- 递归函数调用同样遵守函数调用机制;
- 当函数调用自己时也要将函数状态、返回地址、函数参数、局部变量压入栈中进行保存;
- 在函数的递归调用过程中,为了保证递归调用的正确执行,系统要建一个递归调用工作栈,为各层的调用分配数据存储区;
- 每一层递归调用所需的信息构成一个工作记录。每进入一层递归调用,就产生一个新的工作记录压入栈顶。每退出一层递归调用,就从栈顶弹出一个工作记录。

递归算法的实际运算情况

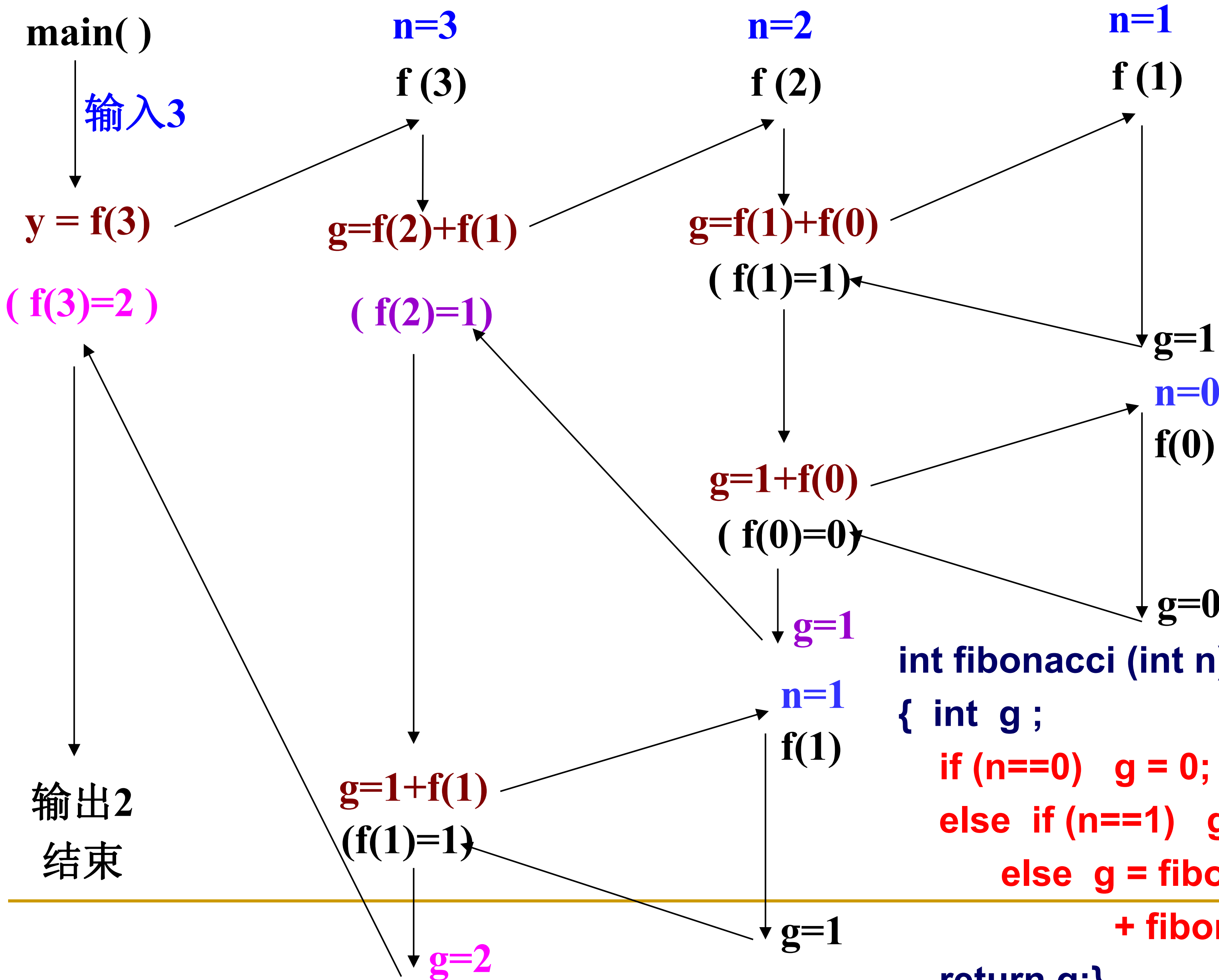
已知有五个人,第一个人年龄为10岁,第二个人比第一个人年龄大两岁,第三个人比第二个人大两岁,第四个人比第三个人大两岁,第五个人比第四个人大两岁.求第五个人的年龄

//Age(5):第5个人的年龄
//Age(4):第4个人的年龄
//Age(3):第3个人的年龄
//Age(2):第2个人的年龄
//Age(1):第1个人的年龄
//Age(n) = Age(n-1) + 2



n=5
n=4
n=3
n=2
n=1
18
16
14
12
10
栈帧
Age(1)
Age(2)
Age(3)
Age(4)
Age(5)

递归程序fibonacci执行过程：




```
int fibonacci (int n)
{  int  g ;
    if (n==0)    g = 0;
    else  if (n==1)   g = 1;
         else  g = fibonacci (n-1)
                  + fibonacci (n-2);
    return g;}
```

递归小结

- 优点：结构清晰，可读性强，而且容易用数学归纳法来证明算法的正确性，因此它为设计算法、调试程序带来很大方便。

- 缺点：递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。

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;
- 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。

分治法的基本步骤

```
divide-and-conquer(P)
 {
  if ( | P | <= n0) adhoc(P);   //解决小规模的问题
  divide P into smaller subinstances 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
 }
```

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。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**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。

分治法的复杂性分析

一个分治法将规模为n的问题分成k个规模为n／m的子问题去解。设分解阀值n0=1，且adhoc解规模为1的问题耗费1个单位时间。再设将原问题分解为k个子问题以及用merge将k个子问题的解合并为原问题的解需用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，则有：

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

通过迭代法求得方程的解： $T(n)=n^{\log_m k}+\sum_{j=0}^{\log_m n-1} k^j f(n/m^j)$

注意：递归方程及其解只给出n等于m的方幂时T(n)的值，但是如果认为T(n)足够平滑，那么由n等于m的方幂时T(n)的值可以估计T(n)的增长速度。通常假定T(n)是单调上升的，从而当$m^i \le n < m^{i+1}$时，$T(m^i) \le T(n) < T(m^{i+1})$。

2.3 二分搜索技术

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: **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;

分解出的子问题的解可以合并为原问题的解;

分解出的各个子问题是相互独立的。

分析：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。

2.3 二分搜索技术

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。

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;
}
```

2.3 二分搜索技术

$$
\begin{aligned}
T(n) &\le T(n/2) + c \\
&\le T(n/4) + c + c \\
&\le T(n/8) + c + c + c \\
&\le T(n/2^k) + kc \\
&\le T(1) + c \log n \quad \text{where } k = \log n \\
&\le b + c \log n \quad = \quad O(\log n)
\end{aligned}
$$

算法复杂度分析：

每执行一次算法的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(logn) 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(logn) 。

2.4 大整数的乘法

请设计一个有效的算法，可以进行两个n位二进制大整数的乘法运算

$$
\begin{array}{ccccccc}
 & & & & 1 & 1 & 1 \\
 & & & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 1 & 1 \\
 & & & 1 & 1 & 1 & \\
 & & 0 & 0 & 0 & & \\
 & 1 & 1 & 1 & & & \\
\hline
1 & 0 & 0 & 1 & 1 & 0 & 1
\end{array}
$$

复杂度：$O(n^2)$　　　　✘效率太低

2. 4 大整数的乘法

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

分治法

X =

Y =

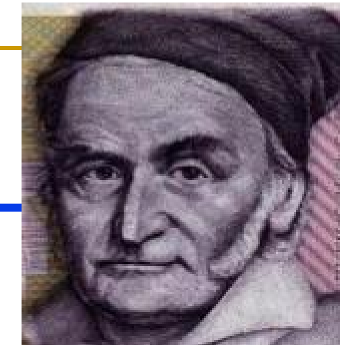
复杂度分析

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

$T(n)=O(n^2)$ ✘**没有改进**

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

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



2. 4

请设计

小学的

分治法

复杂度分析

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

$T(n)=O(n^{\log 3}) =O(n^{1.59})$ ✓ **较大的改进**

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

为了降低时间复杂度，必须减少乘法的次数。

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

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

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

2.4 大整数的乘法

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

分治法: $O(n^{1.59})$ ✓较大的改进

更快的方法??

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。

2.4 大整数的乘法

X =	a	b
Y =	c	d

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

随堂测验，写出递归函数的代码

2.4 大整数的乘法

```
int divideConquer(int x, int y, int n){
    if(x == 0 || y == 0){
        return 0;
    }else if(n == 1){
        return x * y;
    }else{
        int A = (int) x / pow(2, (int)(n / 2));
        int B = x - A * pow(2, n / 2);
        int C = (int) y / pow(2, (int)(n / 2));
        int D = y - C * pow(2, n / 2);
        int AC = divideConquer(A, C, n / 2);
        int BD = divideConquer(B, D, n / 2);
        int ABDC = divideConquer((A - B), (D - C), n / 2) + AC + BD;
        return sign * (AC * pow(2 , n) + ABDC * pow(2, (int)(n / 2)) + BD);
    }
}
```

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