算法

2 递归与分治策略

学习要点

同濟大學 TONGJI UNIVERSITY

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;

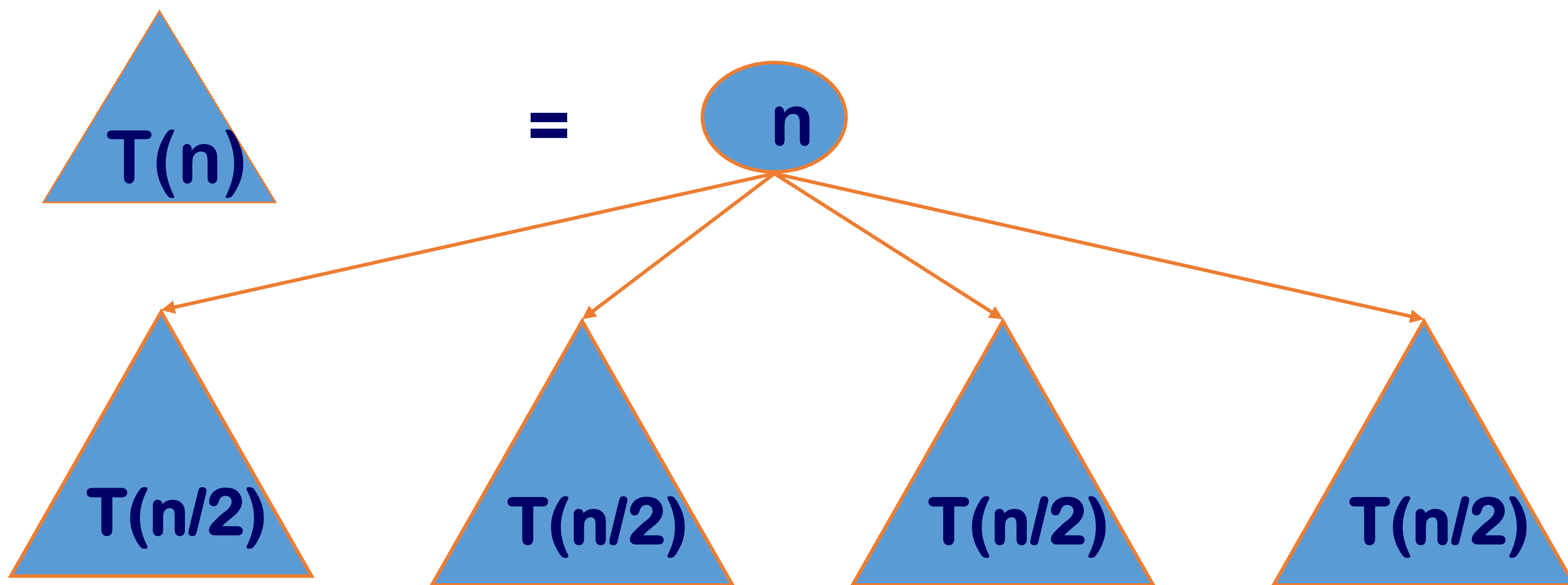
(3)合并排序和快速排序；；

(4) 线性时间选择;

(5) 最接近点对

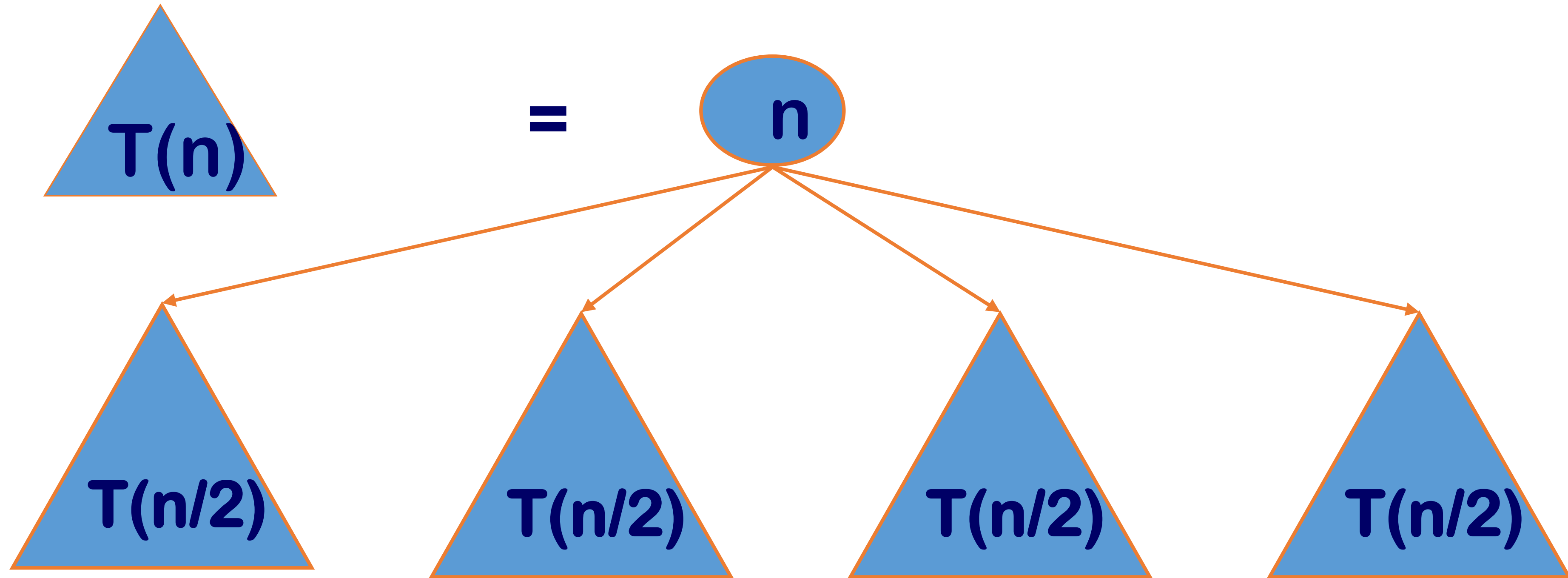
算法总体思想

将要求解的较大规模的问题分割成k个更小规模的子问题。



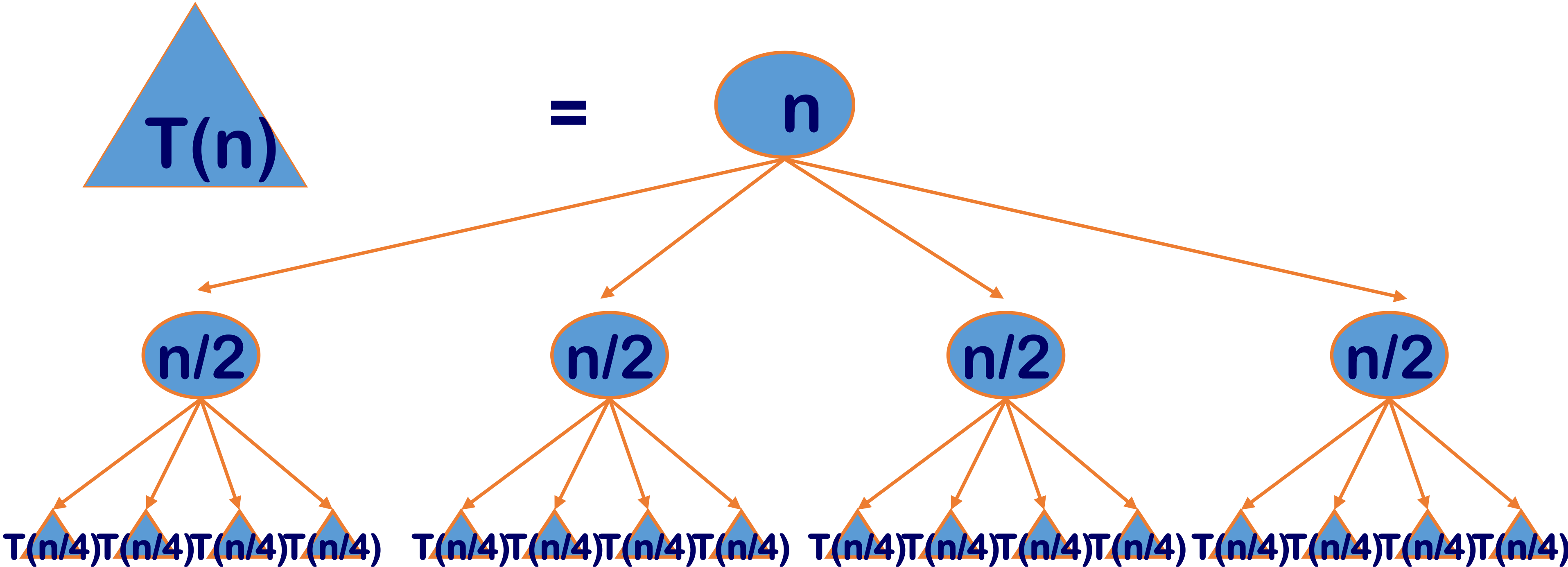
算法总体思想

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。



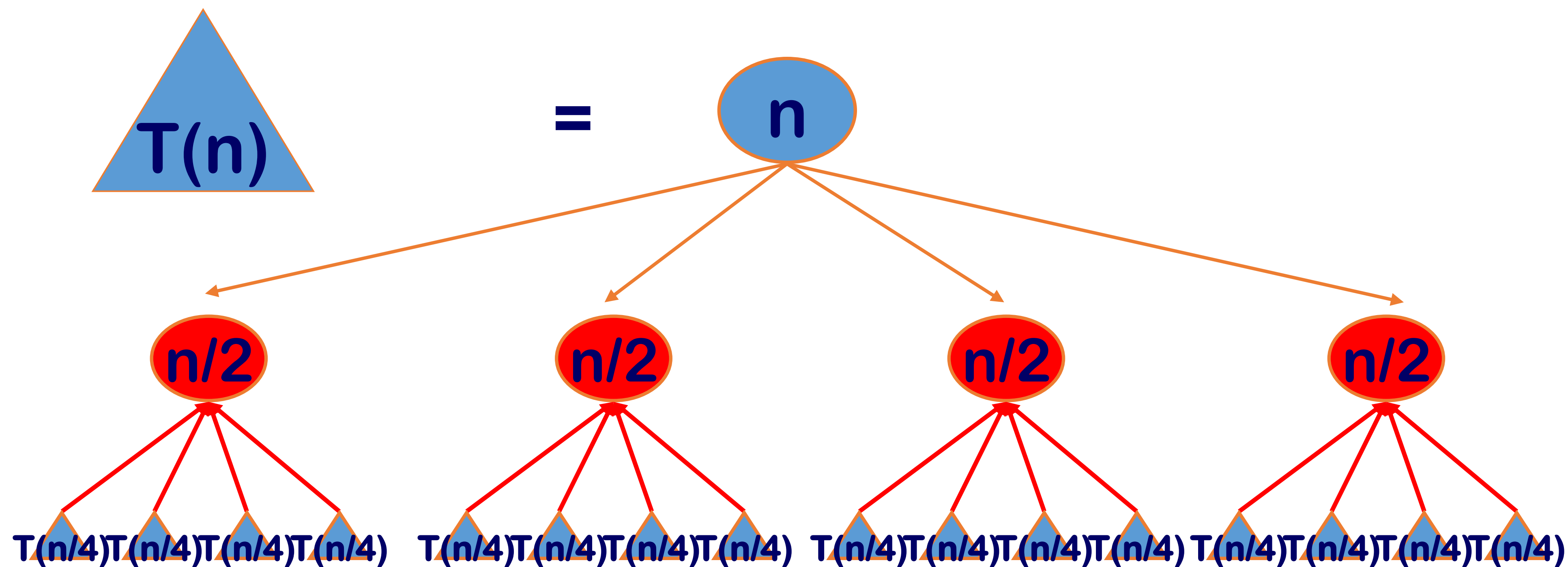
算法总体思想

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。



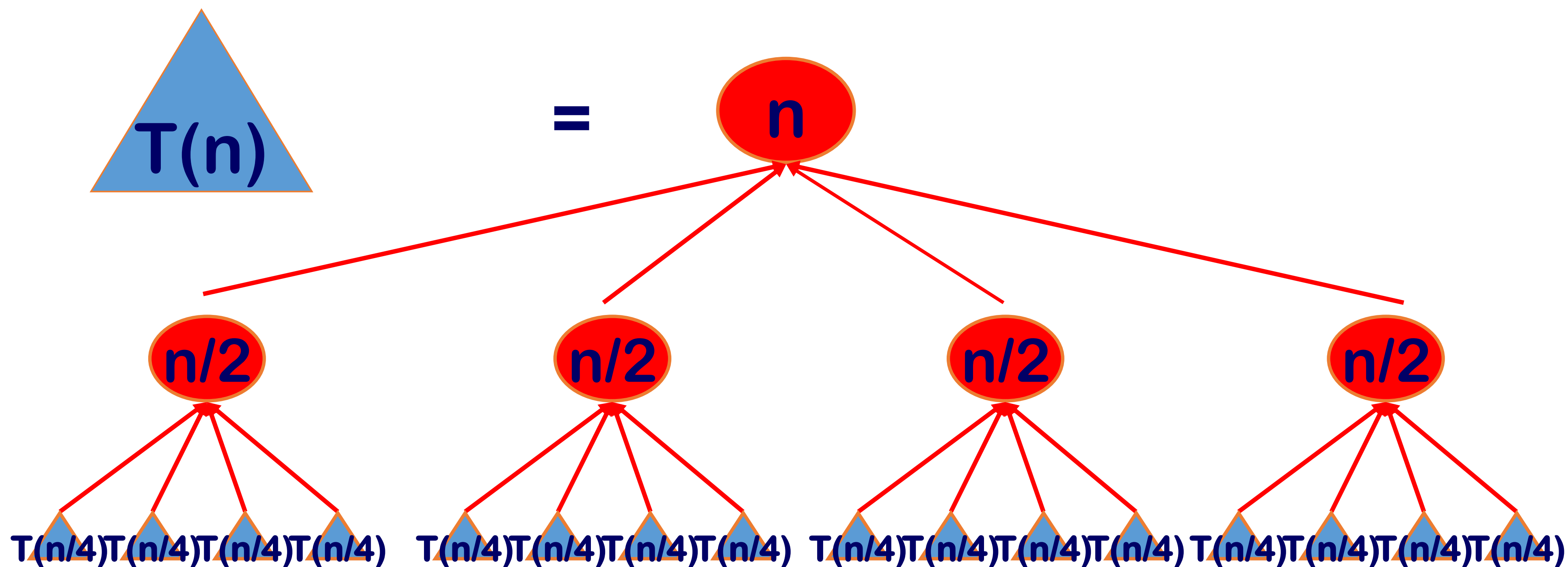
算法总体思想

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。



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相同问题，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

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。

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：

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

边界条件

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);
  }
```

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，$A(n,1)=A(A(n-1,1),0)=A(n-1,1)+2$，和$A(1,1)=2$故$A(n,1)=2n$

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

$m=3$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。

例3 Ackerman函数

Ackerman函数$A(n,m)$定义如下：

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

```
int ackerman(int n,int m)
{
    if(m==0&&n==1)return 2;
    if(m>=0&&n==0)return 1;
    if(m==0&&n>=2)return n+2;
    if(m>=1&&n>=1)return ackerman(ackerman(n-1,m),m-1);
}
```

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r_i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。

对于以上nums = {1，2，3}的例子，我们对全排列的情况做一个分类：

- 以1开头，后面跟着{2，3}的全排列
- 以2开头，后面跟着{1，3}的全排列
- 以3开头，后面跟着{1，2}的全排列

而对于{2，3}、{1，3}、{1，2}，我们又可以重复使用以上的思路，下面以{2，3}为例分析：

- 以2开头，后面跟着{3}的全排列
- 以3开头，后面跟着{2}的全排列

而对于{2}、{3}这些只有一个数的集合，它们的全排列就是它们本身。这也就是递归的终止条件。

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。

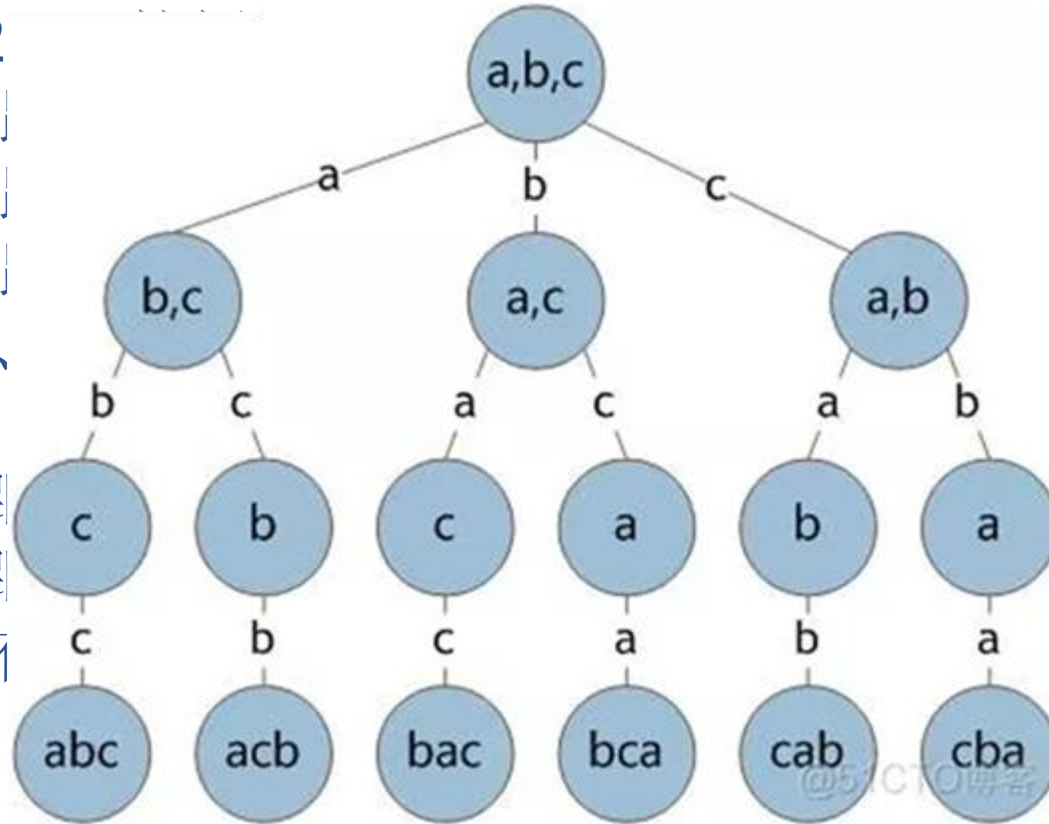
对于以上nums = {1，2 个分类：

- 以1开头，后面
- 以2开头，后面
- 以3开头，后面

而对于{2，3}、{1，3}、 路，下面以{2，3}为例分析：

- 以2开头，后面
- 以3开头，后面

而对于{2}、{3}这些只 门本身。这也就是递归的终止条件。



2.1 递归的概念

例3　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。

以上的思路，用代码按如下方式实现：

对于一个有n个元素的数组a，假设其n个元素分别为a[0]..a[n-1]。

1. 第一步：将其分成两个部分，第一个元素a[0]和后面其他元素，全排列表示以a[0]为首的所有排列，等于a[0]加上后面部分的全排列

2. 第二步：使用a[0]和后面的a[1]交换，同样，得到了以a[1]为首的所有全排列。

3. 第三步：交换回原来的序列，然后再交换a[0]和a[2]。这样能得到以a[2]为首的所有排列，然后再交换回初始状态

4. 第四步：按上面的交换--求排列--再交换回来的流程分别求出以a[3]-a[n-1]为首的所有排列。

这样就得到了所有的排列情况。

请基于上述思路，用代码实现，
https://leetcode.com/problems/permutations/

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。

(1) $q(n,1)=1, n\geq 1$;

当最大加数n_1不大于1时，任何正整数n只有一种划分形式，

即 $n=\overbrace{1+1+\cdots+1}^{n}$

(2) $q(n,m)=q(n,n), m\geq n$;

最大加数n_1实际上不能大于n。因此，$q(1,m)=1$。

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$n_1\le n-1$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$n_1\le m-1$ 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;

$f(6,6) = 1 + f(6,5);$

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。

$f(6,4) = f(2,4) + f(6,3)$;

$f(2,4) = f(2,2)$

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)。

整数划分问题

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

```
int divide(int n, int m)
{
    int res;
    if (n == 1 || m == 1)
        res = 1;
    else if (n < m)
        res = divide(n, n);
    else if (n == m)
        res = 1 + divide(n, n - 1);
    else
        res = divide(n, m - 1) + divide(n - m, m);
    return res;
}
```

2.1 递归的概念

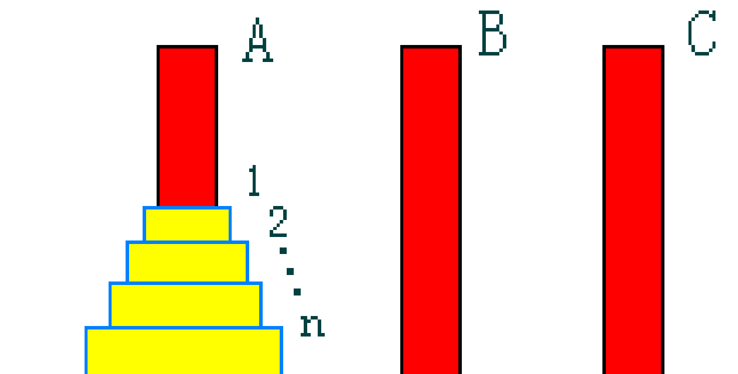
例6 Hanoi塔问题

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；

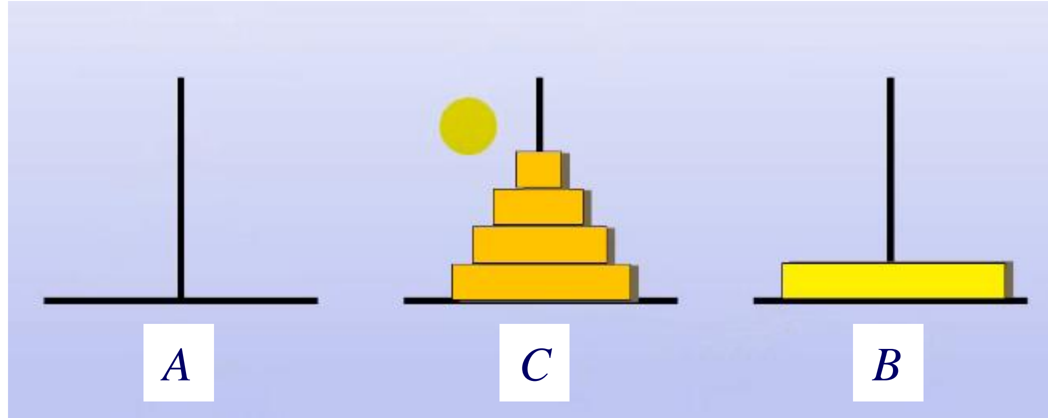
规则2：任何时刻都不允许将较大的圆盘压在较小的圆盘之上；

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:

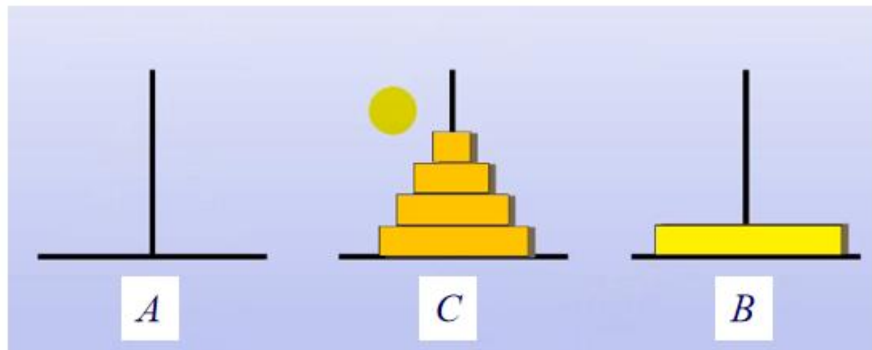


问题就依赖于C柱子上的4个盘子如何移动，也就是n-1个盘子如何从C移动到B上面

2.1 递归的概念

例6 Hanoi塔问题

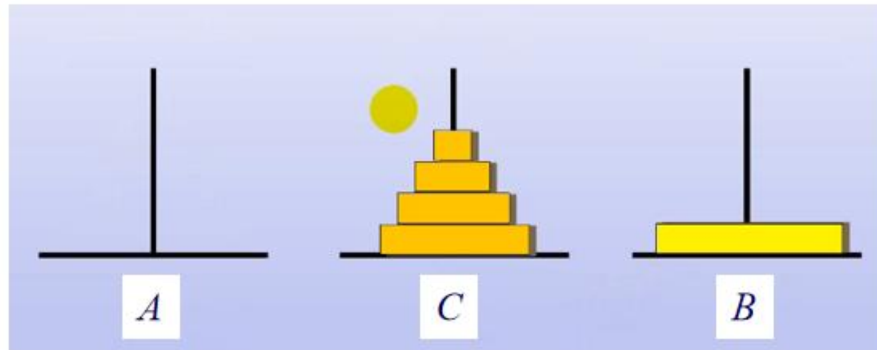
- 在问题规模较大时，较难找到一般的方法，因此我们尝试用递归技术来解决这个问题。
- 当n=1时，问题比较简单。此时，只要将编号为1的圆盘从塔座a直接移至塔座b上即可。
- 当n＞1时，需要利用塔座c作为辅助塔座。此时若能设法将n-1个较小的圆盘依照移动规则从塔座a移至塔座c，然后，将剩下的最大圆盘从塔座a移至塔座b，最后，再设法将n-1个较小的圆盘依照移动规则从塔座c移至塔座b。
- 由此可见，n个圆盘的移动问题可分为2次n-1个圆盘的移动问题，这又可以递归地用上述方法来做。由此可以设计出解Hanoi塔问题的递归算法如下。



2.1 递归的概念

例6　Hano i 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：

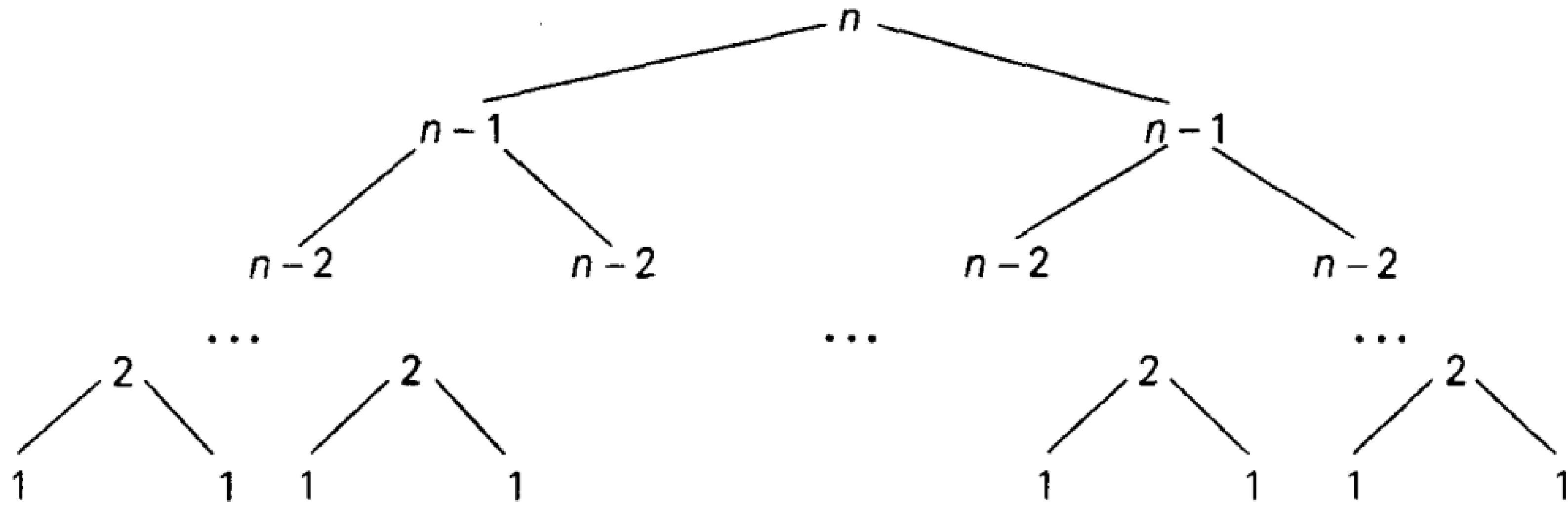
$M(n)=2^iM(n-i)+2^{i-1}+...+2+1=2^iM(n-i)+2^i-1$

令$i=n-1$

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

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。

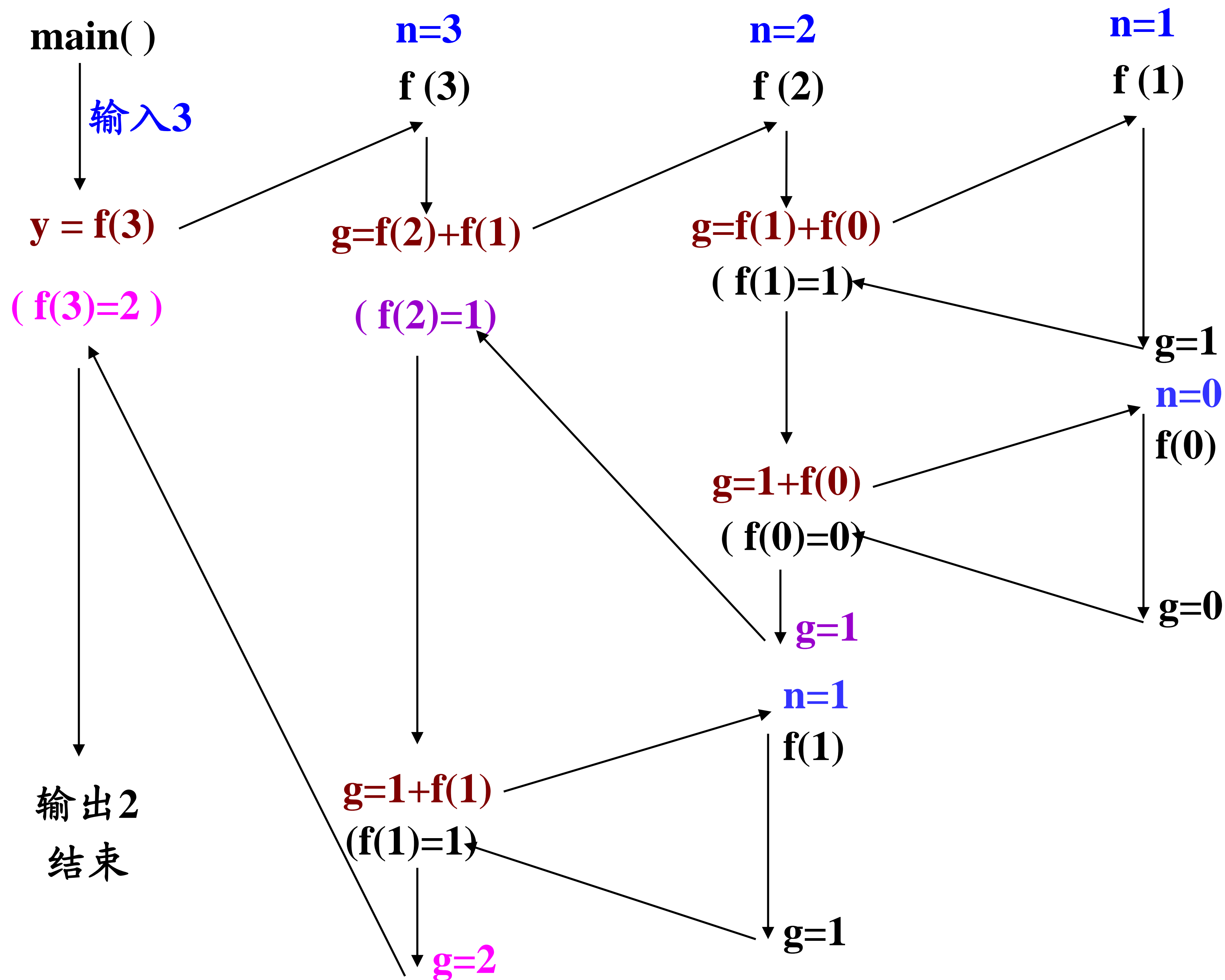
如果一个递归算法会不止一次地调用它本身，处于分析的目的，构造一棵它的递归调用树是很有用的



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：

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

递归程序执行过程（fibonacci)




```
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。

2.2 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。

$T(n)=T(|P1|)+T(|P2|)+\ldots\ldots T(|Pk|)+f(n)$

$T(n_0)=c$

f(n)：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。

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)表示该分治法求解规模为|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(Binary Search)

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。

据此设计出**二分搜索算法**:

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

2.3 二分搜索技术

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

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

```
          1 1 1
        1 0 1 1
    -----------
          1 1 1
        1 1 1
      0 0 0
    1 1 1
  -------------
  1 0 0 1 1 0 1
```

小学的方法　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 =	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

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

$T(n)=O(n^2)$ ✘没有改进

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

分治法:

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

为了降低时间复杂度，必须减少乘法的次数。

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

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

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

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

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。

2.5 合并排序

基本思想：将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合。

```
void MergeSort(Type a[], int left, int right) {
    if (left<right) {//至少有2个元素
    int i=(left+right)/2;  //取中点
    mergeSort(a, left, i);
    mergeSort(a, i+1, right);
    merge(a, b, left, i, right);  //合并到数组b
    copy(a, b, left, right);    //复制回数组a
    }
}
```

时间复杂度分析

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

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

渐进意义下的最优算法

2.5 合并排序

📖最坏时间复杂度：O(nlogn)
📖平均时间复杂度：O(nlogn)

合并排序
优点：稳定性。在最坏情况下的键值比较次数十分接近于任何基于比较的排序算法在理论上能够达到的最少次数。
缺点：算法需要线性的额外空间O(n)

2.6 快速排序

合并排序：按照元素在数组的位置进行划分

快速排序：按照元素的值对他们进行划分

$$\underbrace{A[0]\ldots A[s-1]}_{\text{都}\le A[s]}\quad A[s]\quad \underbrace{A[s+1]\ldots A[n-1]}_{\text{都}\ge A[s]}$$

$A[s]$已经位于有序数组中的最终位置，对$A[s]$前后子数组进行排序

2.6 快速排序

合并排序：将问题划分成两个子问题很快，算法主要工作在于合并子问题的解

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

快速排序：算法主要工作在于划分节点，而不需要再去合并子问题的解

```
QUICKSORT(A, p, r)
1 if p < r
2     // Partition the subarray around the pivot, which ends up in A[q].
3     q = PARTITION(A, p, r)
4     QUICKSORT(A, p, q - 1)      // recursively sort the low side
5     QUICKSORT(A, q + 1, r)      // recursively sort the high side
```

2.6 快速排序

快速排序算法的性能取决于划分的对称性。通过修改算法**partition**，可以设计出采用随机选择策略的快速排序算法。在快速排序算法的每一步中，当数组还没有被划分时，可以在a[p:r]中随机选出一个元素作为划分基准，这样可以使划分基准的选择是随机的，从而可以期望划分是较对称的。

```
template<class Type>
int RandomizedPartition (Type a[], int p, int r){
    int i = Random(p,r);
    Swap(a[i], a[p]);
    return Partition (a, p, r);
}
```

最坏时间复杂度：**$O(n^2)$**

平均时间复杂度：**O(nlogn)**

辅助空间：**O(n)或O(logn)**

2.6 快速排序

快速排序中分治思想三个要点

1、划分步：把输入的问题划分为子问题

2、治理步：调用处理方法来处理问题

3、组合步：把各个子问题的解组合起来

2.7 线性时间选择

给定线性序集中n个元素和一个整数k，1≤k≤n，要求找出这n个元素中第k小的元素?

k = 1: 最小值;
k = n: 最大值;
k = n/2 : 中位数

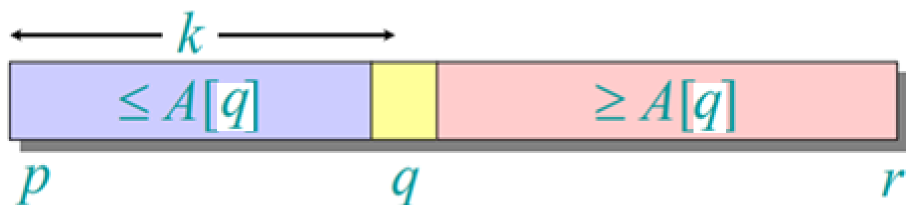
最简单的方法: 排序+选择.
$T(n) = \theta(n\log n) + \theta(1) = \theta(n\log n)$

时间复杂性有没有可能更低?

2.7 线性时间选择

给定线性序集中n个元素和一个整数k，1≤k≤n，要求找出这n个元素中第k小的元素

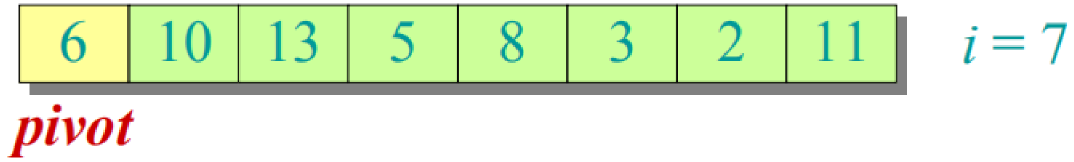
```
RANDOMIZED-SELECT(A, p, r, i)
1 if p == r
2     return A[p]// 1 ≤ i ≤ r – p + 1 when p == r means that i = 1
3 q = RANDOMIZED-PARTITION(A, p, r)
4 k = q – p + 1
5 if i == k
6     return A[q]// the pivot value is the answer
7 elseif i < k
8     return RANDOMIZED-SELECT(A, p, q – 1, i)
9 else return RANDOMIZED-SELECT(A, q + 1, r, i – k)
```



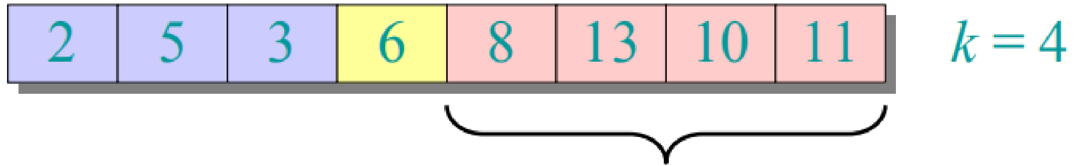
在最坏情况下，算法**randomizedSelect**需要$O(n^2)$计算时间
但可以证明，算法**randomizedSelect**可以在$O(n)$平均时间内找出n个输入元素中的第k小元素。

2.7 线性时间选择

找到第7个最小的数



划分



7 – 4 = 3rd

Lucky:

T(n) =O(n) =O(n)

Unlucky:

$T(n) = T(n – 1) + O(n)= O(n^2)$

比合并排序？

2.7 线性时间选择

平均时间计算

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n\text{–}k\text{–}1 \text{ split} \\ 0 & \text{otherwise.} \end{cases}$$

假设第 k 元素均处于最大的划分数组中

$$T(n) = \begin{cases} T(\max\{0, n\text{–}1\}) + \Theta(n) & \text{if } 0 : n\text{–}1 \text{ split,} \\ T(\max\{1, n\text{–}2\}) + \Theta(n) & \text{if } 1 : n\text{–}2 \text{ split,} \\ \vdots & \\ T(\max\{n\text{–}1, 0\}) + \Theta(n) & \text{if } n\text{–}1 : 0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right).$$

两边取均值

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k (T(\max\{k, n-k-1\}) + \Theta(n))]$$

均值操作的线性性!

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)]$$

不同选择的独立性

2.7 线性时间选择

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$
$$= \sum_{k=0}^{n-1} E[X_k(T(\max\{k, n-k-1\}) + \Theta(n))]$$
$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)]$$
$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n}\sum_{k=0}^{n-1}\Theta(n)$$
$$\le \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n)$$

$E[X_k] = 1/n$.

证明： 存在足够大的常数 c 满足 $E[T(n)] \le cn$

利用结论：

$$\sum_{k=\lfloor n/2 \rfloor}^{n-1} k \le \frac{3}{8}n^2$$

采用归纳法证：

$$E[T(n)] \le \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n)$$
$$\le \frac{2c}{n}\left(\frac{3}{8}n^2\right) + \Theta(n)$$
$$= cn - \left(\frac{cn}{4} - \Theta(n)\right)$$
$$\le cn,$$

总结

优点：线性期望时间，实际应用效果较好

缺点：最坏情况下效果很差

有没有最坏情况下依然为线性的算法？

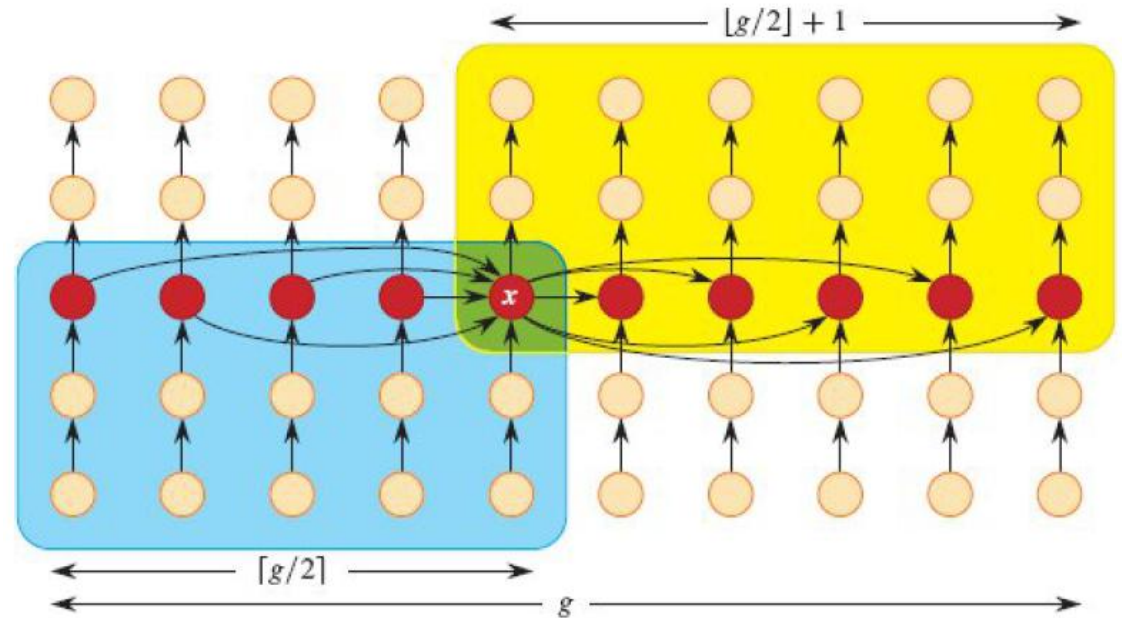
2.7 线性时间选择

改进：通过使划分更均匀，实现在最坏情况下用O(n)时间完成选择任务。

Select算法：找到第**i**小的元素

1.将n个输入元素划分成⌈n/5⌉个组，每组5个元素，至多有一组由剩下的n mod 5个元素组成。

2.用插入排序，将每组中的元素排好序，取出每组的中位数，共⌈n/5⌉个。



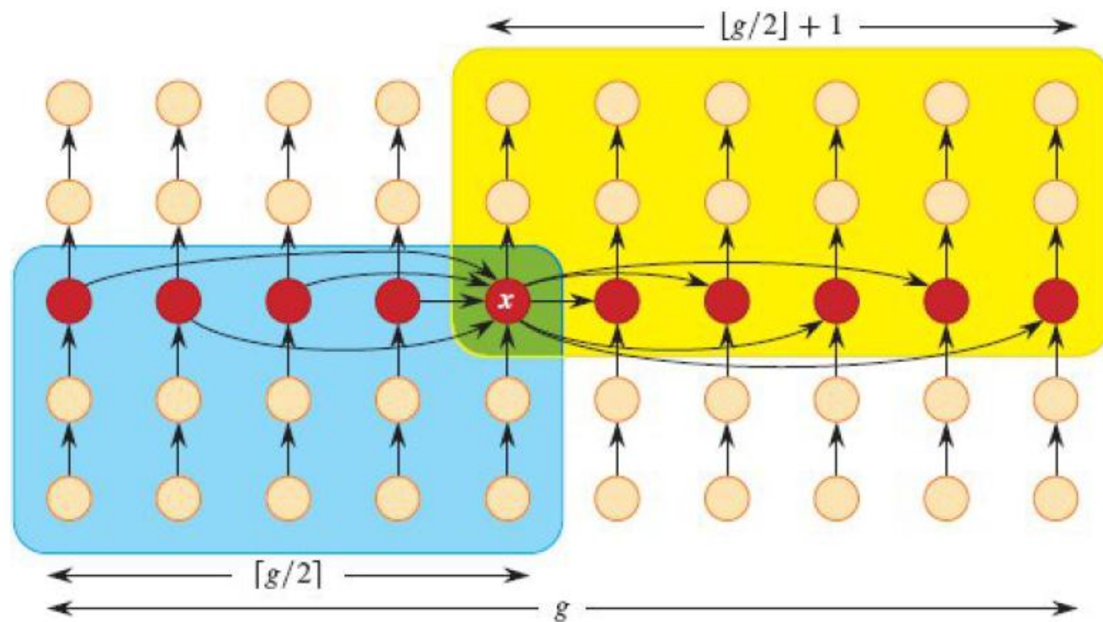
2.7 线性时间选择

改进：通过使划分更均匀，实现在**最坏情况下**用O(n)时间完成选择任务。

3.**调用Select**来找出这⌈n/5⌉个中位数的中位数**x**。如果⌈n/5⌉是偶数，就找它的2个中位数中较大的一个。

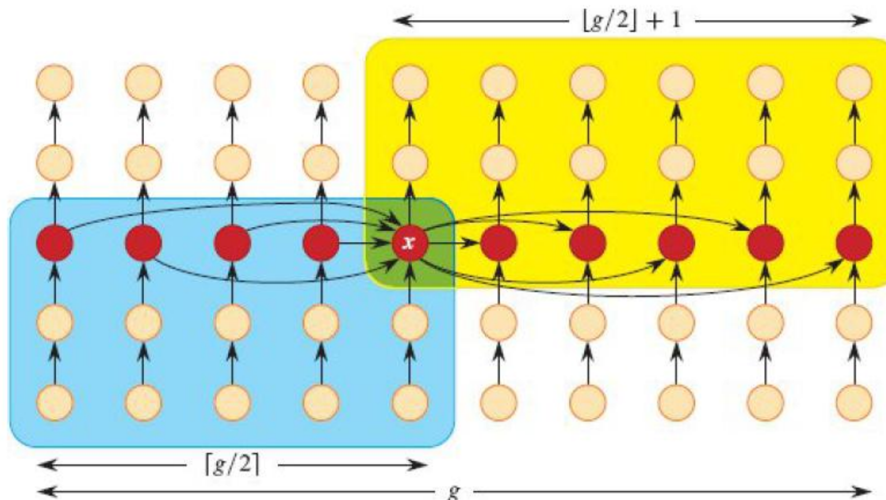
4.按上一步中找到的x作为中轴，对全部 n个元素进行划分，有k-1个数小于x。

5.若i==k，返回x。若i小于k，在低区**调用Select**找出第i小的元素，否则在高区**调用Select**。



2.7 线性时间选择

计算耗时分析



$T(n)$	SELECT(i, n)
$\Theta(n)$	1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
$T(n/5)$	2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
$\Theta(n)$	3. Partition around the pivot x. Let $k = \text{rank}(x)$.
$T(3n/4)$	4. **if** $i = k$ **then return** x **elseif** $i < k$ **then** recursively SELECT the ith smallest element in the lower part **else** recursively SELECT the $(i{-}k)$th smallest element in the upper part

设所有元素互不相同。在这种情况下，找出的基准x至少比3(n-5)/10个元素大，因为在每一组中有2个元素小于本组的中位数，而n/5个中位数中又有(n-5)/10个小于基准x。同理，基准x也至少比3(n-5)/10个元素小。而当n≥75时，3(n-5)/10≥n/4所以按此基准划分所得的2个子数组的长度都至少缩短1/4。

复杂度分析

$$T(n) \le \begin{cases} C_1 & n < 75 \\ C_2 n + T(n/5) + T(3n/4) & n \ge 75 \end{cases}$$

T(n)=**O(n)**

归纳法

$T(n) \le cn$

$$\begin{aligned} T(n) &\le \frac{1}{5}cn + \frac{3}{4}cn + \Theta(n) \\ &= \frac{19}{20}cn + \Theta(n) \\ &= cn - \left(\frac{1}{20}cn - \Theta(n)\right) \\ &\le cn \end{aligned}$$

,

- 补充：实际上，该算法运行缓慢，因为n前面的常数很大。

```
Type Select(Type a[], int p, int r, int k){
    if (r-p<75) {
      用某个简单排序算法对数组a[p:r]排序;
      return a[p+k-1];
      }
    for ( int i = 0; i<=(r-p-4)/5; i++ )
       //将a[p+5*i]至a[p+5*i+4]的第3小元素
        与a[p+i]交换位置;
      //找中位数的中位数，r-p-4即上面所说的n-5
      Type x = Select(a, p, p+(r-p-4)/5, (r-p-4)/10);
      int i=Partition(a,p,r, x), j=i-p+1;
      if (k<=j) return Select(a,p,i,k);
      else return Select(a,i+1,r,k-j);
   }
}
```

上述算法将每一组的大小定为5，并选取75作为是否作递归调用的分界点。这2点保证了T(n)的递归式中2个自变量之和n/5+3n/4=19n/20=εn，0<ε<1。这是使T(n)=O(n)的关键之处。当然，除了5和75之外，还有其他选择。

2.7 线性时间选择

如果能在线性时间内找到一个划分基准，使得按这个基准所划分出的2个子数组的长度都至少为原数组长度的ε倍(0<ε<1是某个正常数)，那么就可以**在最坏情况下**用O(n)时间完成选择任务。

例如，若ε=9/10，算法递归调用所产生的子数组的长度至少缩短1/10。所以，在最坏情况下，算法所需的计算时间T(n)满足递归式
$T(n) \leqslant T(9n/10)+O(n)$ 。由此可得$T(n)=O(n)$。

2.8 最接近点对问题

计算几何学中研究的基本问题之一。

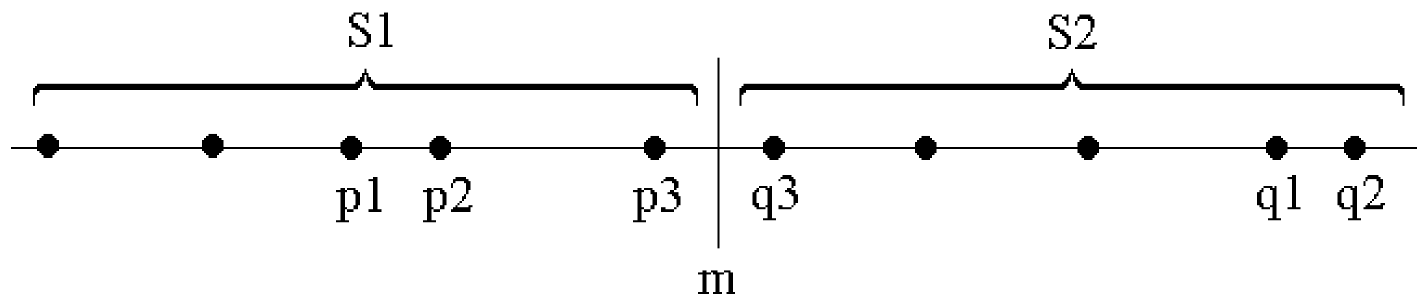
在涉及几何对象的问题中，常需要了解其邻域中其他几何对象的信息。例如，在空中交通控制问题中，若将飞机作为空间中移动的一个点来看待，则具有最大碰撞危险的两架飞机，就是这个空间中最接近的一对点。

最接近点问题：给定平面上的n个点，找其中的一对点，使得在n个点组成的所有点对中，该点距离最小

2.8 最接近点对问题

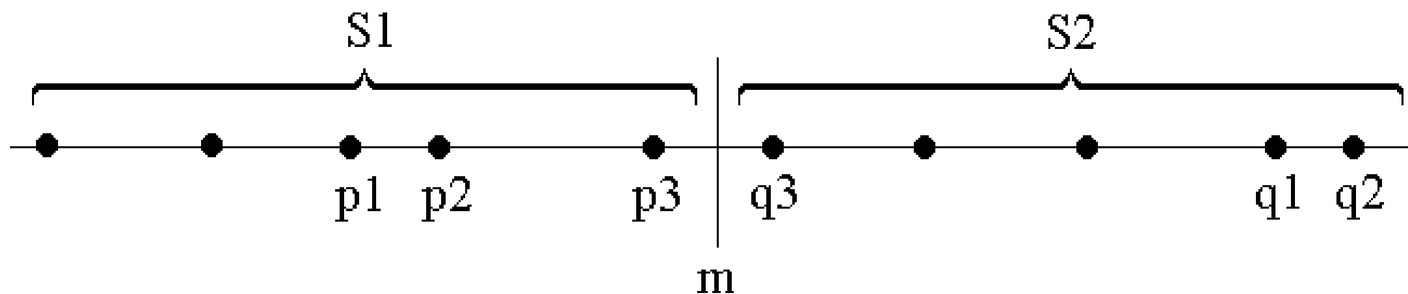
给定平面上n个点的集合S，找其中的一对点，使得在n个点组成的所有点对中，该点对间的距离最小。

为了使问题易于理解和分析，先来考虑**一维**的情形。此时，S中的n个点退化为x轴上的n个实数 x1,x2,…,xn。最接近点对即为这n个实数中相差最小的2个实数。



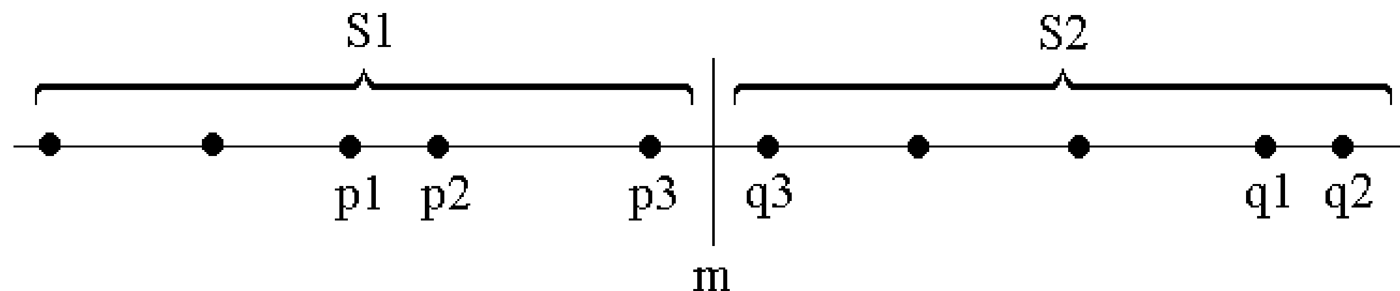
2.10 最接近点对问题

假设我们用x轴上某个点m将S划分为2个子集S1和S2，基于平衡子问题的思想，用S中各点坐标的中位数来作分割点。
递归地在S1和S2上找出其最接近点对{p1,p2}和{q1,q2}，并设$d=\min\{|p1-p2|,|q1-q2|\}$，S中的最接近点对或者是{p1,p2}，或者是{q1,q2}，或者是某个{p3,q3}，其中$p3 \in S1$且$q3 \in S2$。
能否在线性时间内找到p3,q3?



为什么不用均值

2.8 最接近点对问题



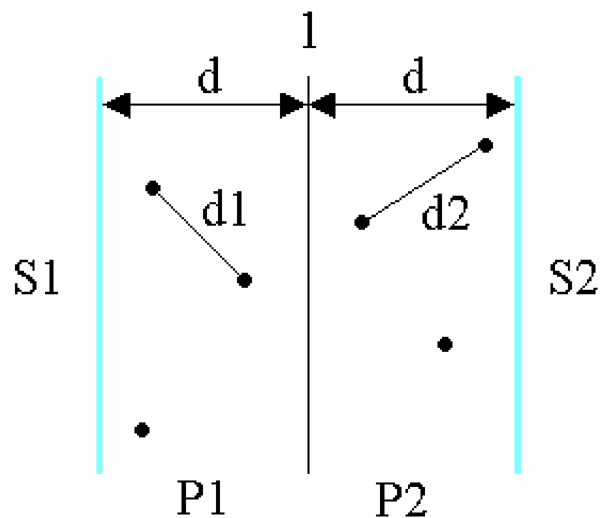
能否在线性时间内找到p3,q3?

- 如果S的最接近点对是{p3,q3}，即|p3-q3|<d，则p3和q3两者与m的距离不超过d，即**p3∈(m-d,m]**，**q3∈(m,m+d]**。
- 由于在S1中，每个长度为d的半闭区间至多包含一个点(否则必有两点距离小于d)，并且m是S1和S2的分割点，因此(m-d,m]中至多包含S中的一个点。由图可以看出，**如果(m-d,m]中有S中的点，则此点就是S1中最大点。**
- 因此，我们用线性时间就能找到区间(m-d,m]和(m,m+d]中所有点，即p3和q3。**从而我们用线性时间就可以将S1的解和S2的解合并成为S的解。**

2.8 最接近点对问题

- 下面来考虑二维的情形。

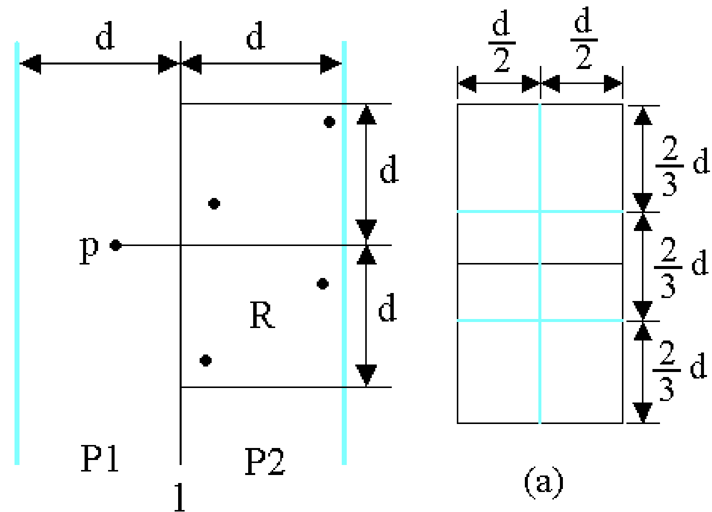
- 选取一垂直线l:x=m来作为分割直线。其中m为S中各点x坐标的中位数。由此将S分割为S1和S2。
- 递归地在S1和S2上找出其最小距离d1和d2，并设d=min{d1,d2}，S中的最接近点对或者是d，或者是某个{p,q}，其中p∈S1且q∈S2。
- 能否在线性时间内找到p,q?



2.8 最接近点对问题

能否在线性时间内找到p,q?

- 考虑P1中任意一点p，它若与P2中的点q构成最接近点对的候选者，则必有distance(p，q)＜d。满足这个条件的P2中的点一定落在一个d × 2d的矩形R中
- 由d的意义可知，P2中任何2个S中的点的距离都不小于d。由此可以推出矩形R中最多只有6个S中的点。
- 因此，在分治法的合并步骤中最多只需要检查6 × n/2=3n个候选者



证明: 将矩形R的长为2d的边3等分，将它的长为d的边2等分，由此导出6个(d/2) × (2d/3)的矩形。若矩形R中有多于6个S中的点，则由鸽舍原理易知至少有一个(d/2) × (2d/3)的小矩形中有2个以上S中的点。设u，v是位于同一小矩形中的2个点，则

$$(x(u)-x(v))^2+(y(u)-y(v))^2 \le (d/2)^2+(2d/3)^2 = \frac{25}{36}d^2$$

distance(u,v)<d。这与d的意义相矛盾。

2.8 最接近点对问题

- 为了确切地知道要检查哪6个点，可以将p和P2中所有S2的点投影到垂直线l上。由于能与p点一起构成最接近点对候选者的S2中点一定在矩形R中，所以它们在直线l上的投影点距p在l上投影点的距离小于d。由上面的分析可知，这种投影点最多只有6个。
- 因此，若将P1和P2中所有S中点按其y坐标排好序，则对P1中所有点，对排好序的点列作一次扫描，就可以找出所有最接近点对的候选者。对P1中每一点最多只要检查P2中排好序的相继6个点。

2.8 最接近点对问题

```
double cpair2(S)
{
    n=|S|;
    if (n < 2) return    ;
1、m=S中各点x间坐标的中位数;
    构造S1和S2;
    //S1={p∈S|x(p)<=m},
    S2={p∈S|x(p)>m}
2、d1=cpair2(S1);
    d2=cpair2(S2);
3、dm=min(d1,d2);
4、设P1是S1中距垂直分割线l的距离在dm之内的所有点组成的集合;
    P2是S2中距分割线l的距离在dm之内所有点组成的集合;
    将P1和P2中点依其y坐标值排序;
    并设X和Y是相应的已排好序的点列;
5、通过扫描X以及对于X中每个点检查Y中与其距离在dm之内的所有点(最多6个)可以完成合并;
    当X中的扫描指针逐次向上移动时，Y中的扫描指针可在宽为2dm的区间内移动;
    设dl是按这种扫描方式找到的点对间的最小距离;
6、d=min(dm,dl);
    return d;
}
```

复杂度分析

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

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

2.8 最接近点对问题

2.9 主定理 求解递推方程

主定理：设$a≥1, b>1$为常数，$f(n)$为函数，$T(n)$为非负整数，且

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

则有以下结果：

1. 若$f(n) = O(n^{\log_b a-\varepsilon}), \varepsilon > 0$，那么$T(n) = \Theta(n^{\log_b a})$
2. 若$f(n) = \Theta(n^{\log_b a})$，那么$T(n) = \Theta(n^{\log_b a} \log n)$
3. 若$f(n) = \Omega(n^{\log_b a+\varepsilon}), \varepsilon > 0$，且对于某个常数 $c < 1$和充分大的n有

$$a f(n/b) \le cf(n)，那么 T(n)=\Theta(f(n))$$

第二条修正：

If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$
then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

(1) 在第一种情况，$f(n)$不仅小于$n^{\log_b a}$，必须多项式地小于，即对于一个常数$\varepsilon > 0$，$f(n) = O(\frac{n^{\log_b a}}{n^{\varepsilon}})$.

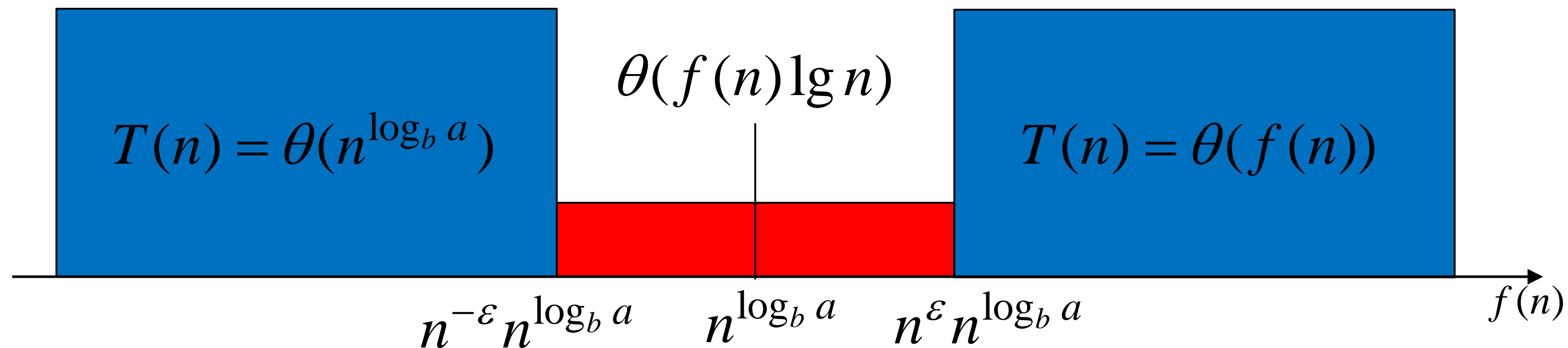
(2) 在第三种情况，$f(n)$不仅大于$n^{\log_b a}$，必须多项式地大于，即对于一个常数$\varepsilon > 0$，$f(n) = \Omega\left(n^{\log_b a} \cdot n^{\varepsilon}\right)$.

*直观地：我们用$f(n)$与$n^{\log_b a}$比较

(1). 若$n^{\log_b a}$大，则$T(n) = \theta(n^{\log_b a})$

(2). 若$f(n)$大，则$T(n) = \theta(f(n))$

(3). 若$f(n)$与$n^{\log_b a}$同阶，则$T(n) = \theta\left(n^{\log_b a} \lg n\right) = \theta(f(n) \lg n)$.



主定理的应用

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

*直观地：我们用$f(n)$与$n^{\log_b a}$比较

(1). 若$n^{\log_b a}$大，则$T(n) = \theta(n^{\log_b a})$

(2). 若$f(n)$大，则$T(n) = \theta(f(n))$

(3). 若$f(n)$与$n^{\log_b a}$同阶，则$T(n) = \theta(n^{\log_b a} \lg n) = \theta(f(n) \lg n)$.

例　求解递推方程 $T(n) = 9T(n/3) + n$

解 上述递推方程中的 $a = 9, b = 3, f(n) = n,$　　那么

$$n^{\log_3 9} = n^2, \quad f(n) = O(n^{\log_3 9 - 1}),$$

相当于主定理的第一种情况，其中ε=1. 根据定理得到

$$T(n) = \Theta(n^2)$$

例　求解递推方程 $T(n) = T(2n/3) + 1$

解　上述递推方程中的 a=1, b=3/2, $f(n)$=1，那么

$$n^{\log_{3/2} 1} = n^0 = 1, \quad f(n) = 1$$

相当于主定理的第二种情况. 根据定理得到 .

$$T(n) = \Theta(n^0 \log n) = \Theta(\log n)$$

主定理的应用

实例

例 求解递推方程 $T(n)=3T(n/4)+n\log n$

上述递推方程中的a=3, b=4, $f(n)$=nlog$n$，那么

$$n\log n=\Omega(n^{\log_4 3+\varepsilon})=\Omega(n^{0.793+\varepsilon}) \qquad \varepsilon\approx 0.2$$

此外，要使 $af(n/b)\le cf(n)$成立，代入$f(n)$=nlog$n$，得到

$$\frac{3n}{4}\log\frac{n}{4}\le cn\log n$$

显然只要 $c\ge 3/4$，上述不等式就可以对充分大的n成立. 相当于主定理的第三种情况. 因此有

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

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

*直观地：我们用$f(n)$与$n^{\log_b a}$比较

(1). 若$n^{\log_b a}$大，则$T(n)=\theta(n^{\log_b a})$

(2). 若$f(n)$大，则$T(n)=\theta(f(n))$

(3). 若$f(n)$与$n^{\log_b a}$同阶，则$T(n)=\theta(n^{\log_b a}\lg n)=\theta(f(n)\lg n)$.

主定理的应用

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

*直观地：我们用$f(n)$与$n^{\log_b a}$比较

(1). 若$n^{\log_b a}$大，则$T(n) = \theta(n^{\log_b a})$

(2). 若$f(n)$大，则$T(n) = \theta(f(n))$

(3). 若$f(n)$与$n^{\log_b a}$同阶，则$T(n) = \theta(n^{\log_b a}\lg n) = \theta(f(n)\lg n)$.

不能使用主定理的例子

$$T(n) = 2T(n/2) + n\log n$$

$$T(1) = 1$$

- 不能使用主定理

  $a = 2,\ b = 2,\ n^{\log_b a} = n,\quad f(n) = n\log n$

  不存在$c<1$使得 $2(n/2)\log(n/2) \le c\, n\log n$

  不存在 $\varepsilon > 0$ 使得 $f(n) = n\log n = \Omega(n^{1+\varepsilon})$
- 只能使用递归树，结果等于 $T(n)=n\log^2 n$

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

符合主定理修正的第二条：If there exists a constant $k \ge 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$ then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.