递归与分治

陈长建

计算机科学系

课前5分钟简答

- Donald E. Knuth（高德纳）因为什么获得了图灵奖?
- 复杂度包括哪两种?
- 程序和算法的区别?

第一次作业

- 1、算法分析题1-6
- 2、算法实现题1-1 统计数字问题
- 3、算法实现题1-2 字典序问题

- 提交时间：9月29日

作业要求

- 以**文档形式**提交在学习通中
- 算法分析题：
  - 要求有简要思路
- 算法实现题：
  - 要求有简要思路、代码、运行结果、分析

- 鼓励讨论、但是严禁抄袭

关于实验

- 主题三分
  - 离线测试题 (要求：有三种规模测试数据和实验报告)
  - 在线测试题（系统自动判断）

- 总计4次实验
- 验收时间：在第5、8、12、16周周末

第一次实验

- 院楼103，10月13日（周日）上午8:30-12:00（现场验收）
- 第一次实验离线题（离线准备）
  - 1. 分治法查找最大最小值
  - 2. 分治法实现合并排序
  - 3. 实现题1-3 最多约数问题
- 在线题

关于实验准备和验收

- 离线作业评分标准
  - 经典案例+课后实现题 （离线准备）
  - 1. 能够熟练讲解算法思路和程序代码。 （40%）
  - 2. 生成三种不同规模的测试数据：小、中、大（e.g., 几个、几百个、几万个甚至更多）。尝试随机数据生成方法。尝试文件读写操作（30%） （*加分项：爬取或搜索下载大规模真实数据集，抽取小、中数据，构造小中大三个规模*）
  - 3. 实验报告有不同规模数据实验的时间对比、有时间复杂度分析。 （15%）
  - 4. 完善实验报告。 （15%）

关于实验准备和验收

- 在线测试题
  - 系统：OJ系统
  - 练习：可以提前在一些在线网站上练习
    - LeetCode

本章要点

- 算法总体思想
- 递归的概念
- 递归案例分析
- 分治法的概念
- 分治案例分析

本章要点

- 算法总体思想
- 递归的概念
- 递归案例分析
- 分治法的概念
- 分治案例分析

算法总体思想

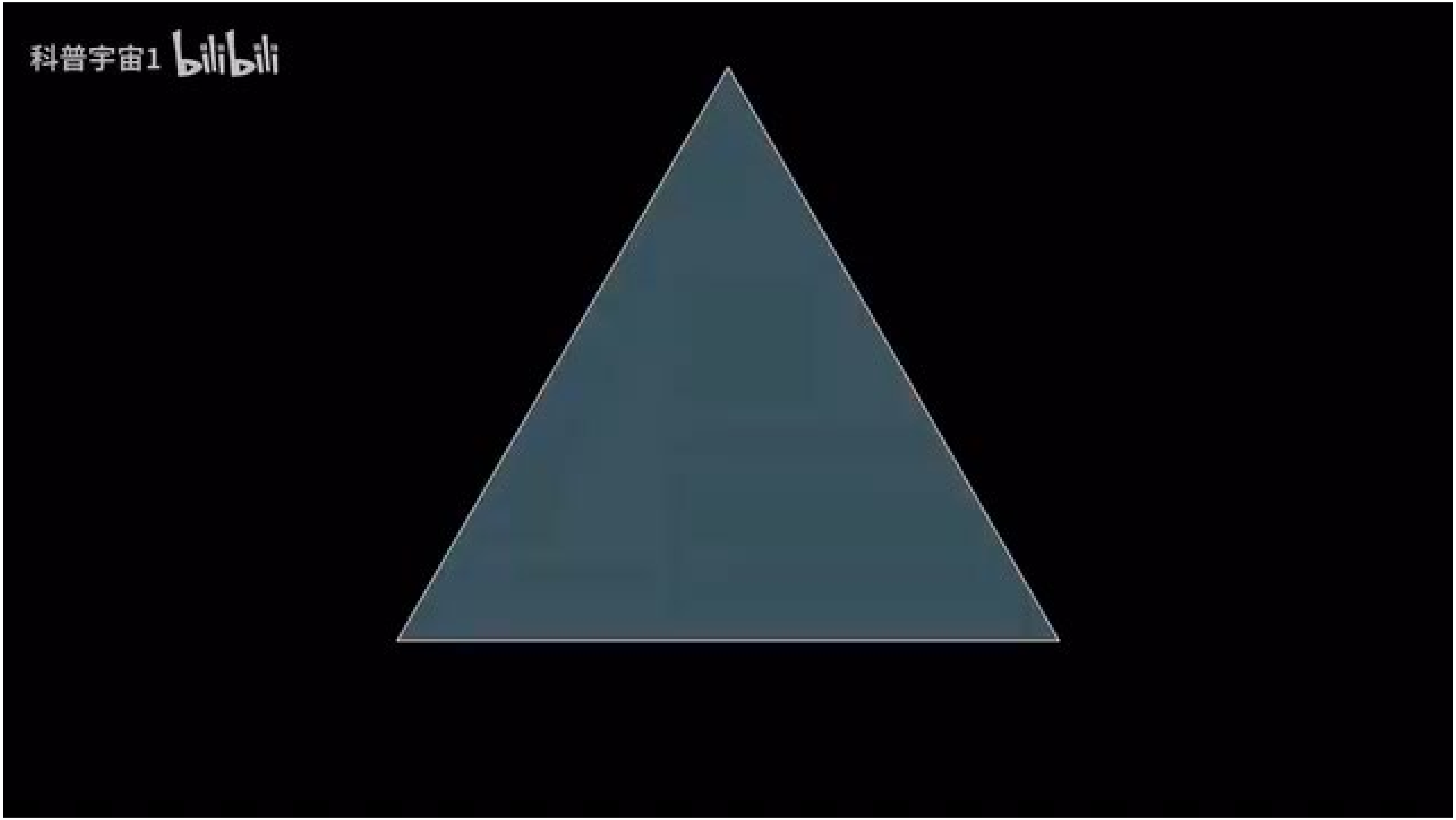
- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。

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。

凡治众如治寡，分数是也。

----孙子兵法

算法总体思想-分形



本章要点

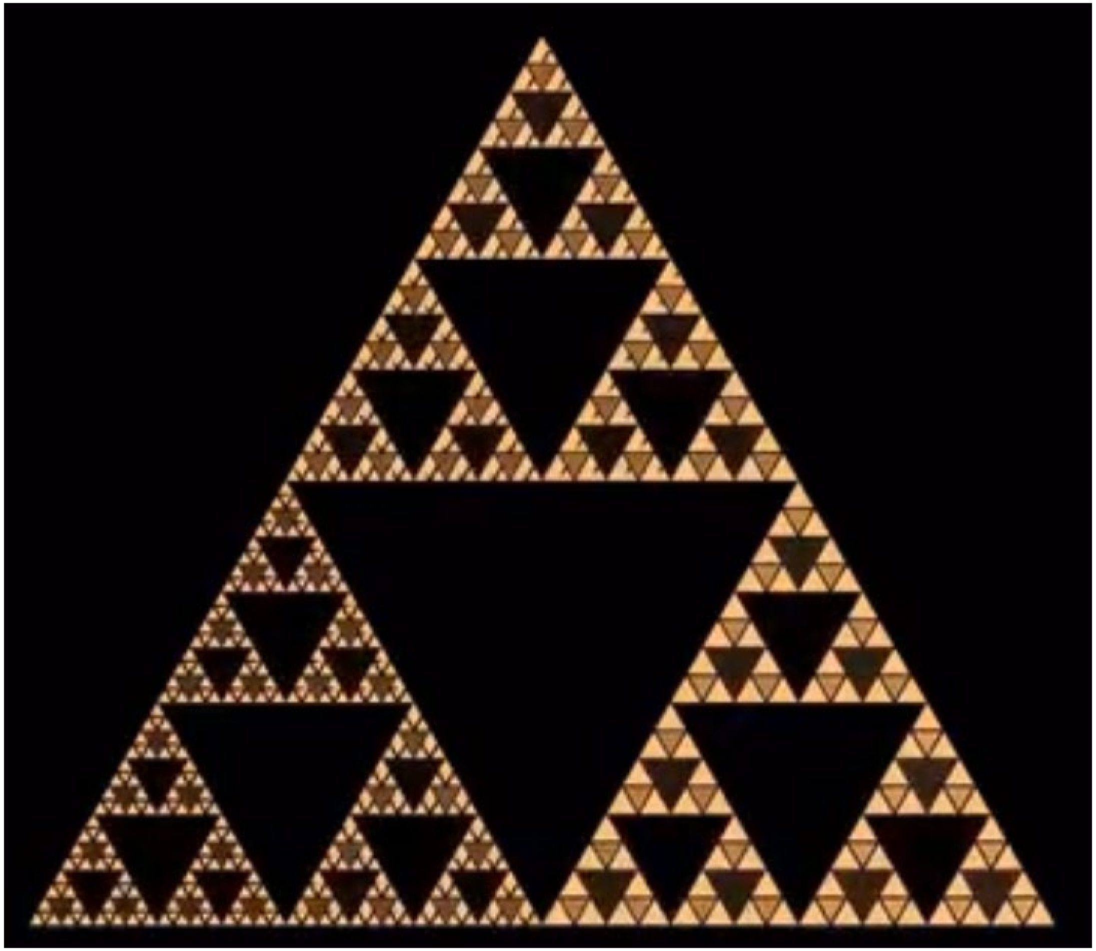
- 算法总体思想
- 递归的概念
- 递归案例分析
- 分治法的概念
- 分治案例分析

递归的概念-缘起

- 1，从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？“从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？‘从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？……’”

- 2，大雄在房里，用时光电视看着从前的情况。电视画面中的那个时候，他正在房里，用时光电视，看着从前的情况。电视画面中的电视画面的那个时候，他正在房里，用时光电视，看着从前的情况……

递归的概念-分形



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。
- 由递归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。

递归的条件

- 自己调用自己的函数称为**递归函数**(recursive function)

直接递归：F $\rightarrow$ F

间接递归：F $\rightarrow$ G $\rightarrow$ H $\rightarrow$ ... $\rightarrow$ F

完整的递归定义必须满足如下条件：

  – 包含一个基本部分，对于n的一个或多个值， F(n)必须是直接定义的（即边界条件）。
  – 在递归部分中，右侧所出现的所有F的参数都必须有一个比n小（递归方程）。

本章要点

- 算法总体思想
- 递归的概念
- 递归案例分析
- 分治法的概念
- 分治案例分析

递归-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。

递归-例1　阶乘函数

- 形式定义问题：　factorial(n)
- 边界条件：　factorial(0)=1
- 如何用小问题解决大问题（递归方程）：

  factorial(n) = n * factorial(n-1)

- 实现：

```
public static int  factorial(int n)
 {
    if (n == 0) return 1;
    return n*factorial(n-1);
 }
```

递归-例2 Fibonacci数列

- 例：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:

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

递归-例2 Fibonacci数列

- 形式定义问题： Fibonacci(n)
- 边界条件： Fibonacci(0)= Fibonacci(1)=1
- 递归方程：

  Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

- 实现

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

递归-例3 排列问题

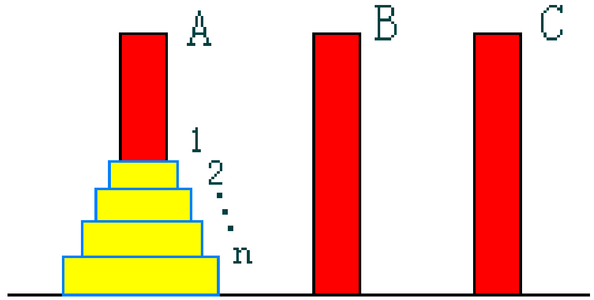
- 问题：生成 n个元素$\{r_1,r_2,\ldots,r_n\}$的全排列。
  - 例如，给定3个元素{1, 2, 3}，则全排列为：
    - {1,2,3}
    - {1,3,2}
    - {2,3,1}
    - {2,1,3}
    - {3,1,2}
    - {3,2,1}

递归-例3 排列问题

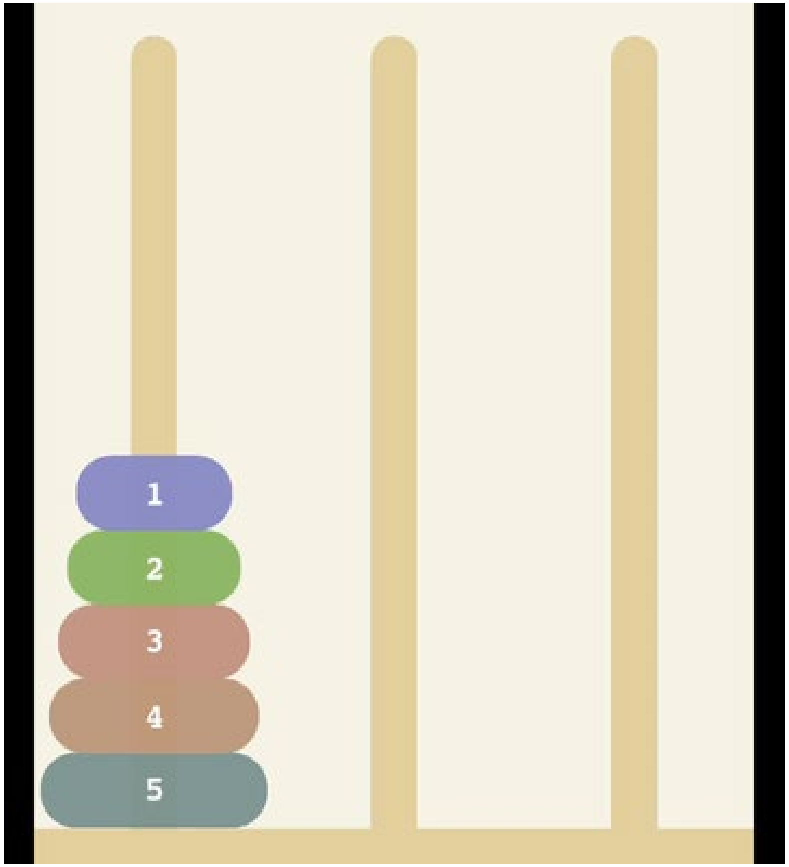
- 形式定义问题：perm(R)，其中$R=\{r_1,r_2,\ldots,r_n\}$
- 边界条件:
  - $perm(\{r_1\})=\{\{r_1\}\}$
- 递归方程:
  - 定义集合减法：$R-r_i=\{r_1,r_2,\ldots,r_{i-1}, r_{i+1}, \ldots, r_n\}$
  - $perm(R)=(r_1)perm(R-r_1) + (r_2)perm(R-r_2) + \ldots + (r_n)perm(R-r_n)$

递归-例4 汉诺（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c上，并仍按同样顺序叠置。在移动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。

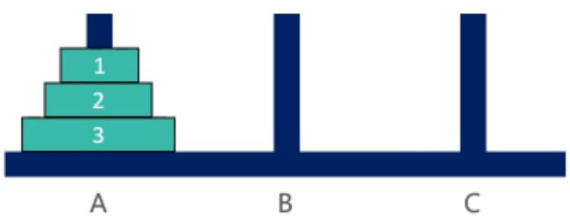


递归-例4 汉诺（Hanoi）塔问题

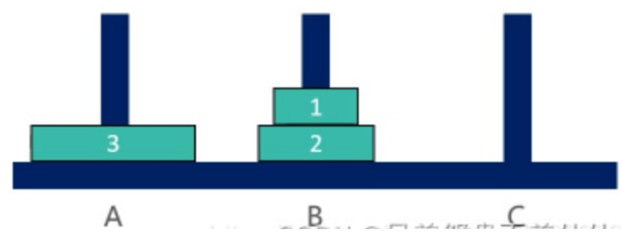


递归-例4 Hanoi塔问题

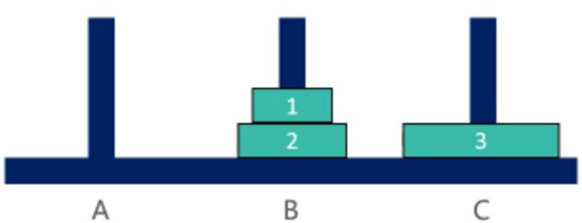
- 形式定义问题：hanoi(n, a, b, c)
  - 含义：将n个hanoi圆盘从a柱借助b柱移动到c柱
- 递归方程:



hanoi(n-1, a, c, b)



move(a,c)



hanoi(n-1, b, a, c)

递归-小结

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

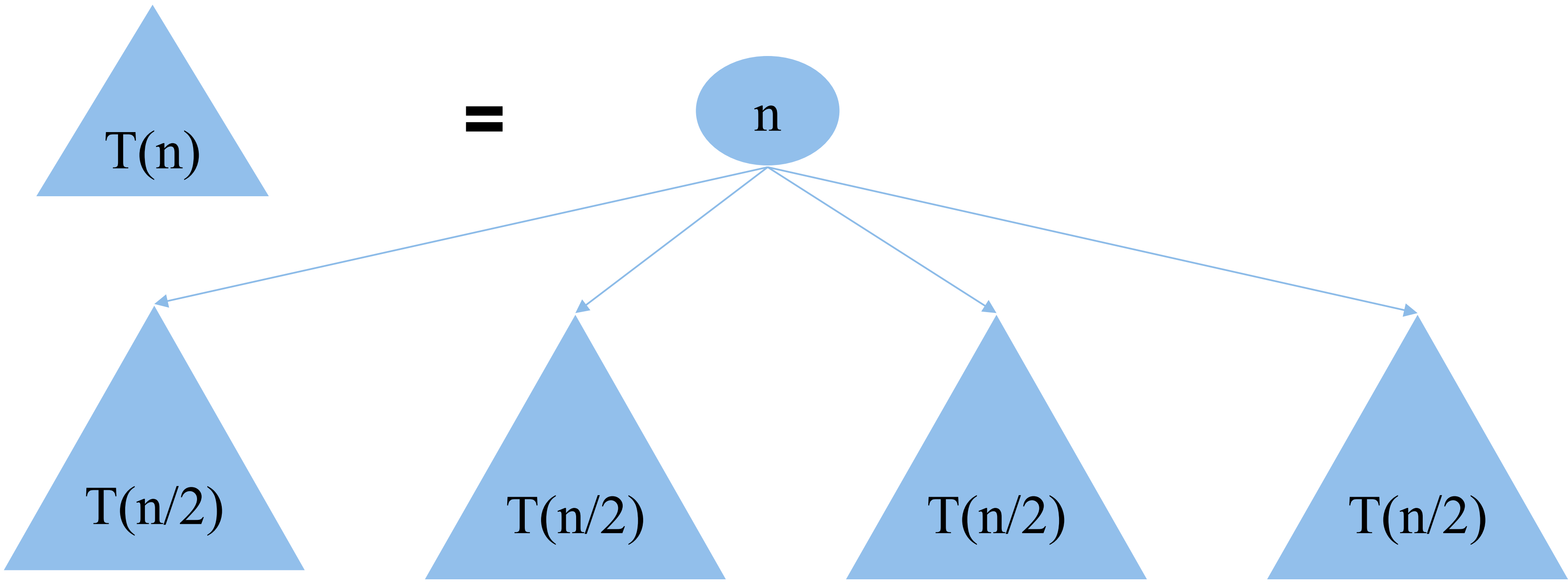
- 缺点：递归算法运行效率较低，时空消耗大。
  - 解决方法：在递归算法中消除递归调用，使其转化为非递归算法。

本章要点

- 算法总体思想
- 递归的概念
- 递归案例分析
- 分治法的概念
- 分治案例分析

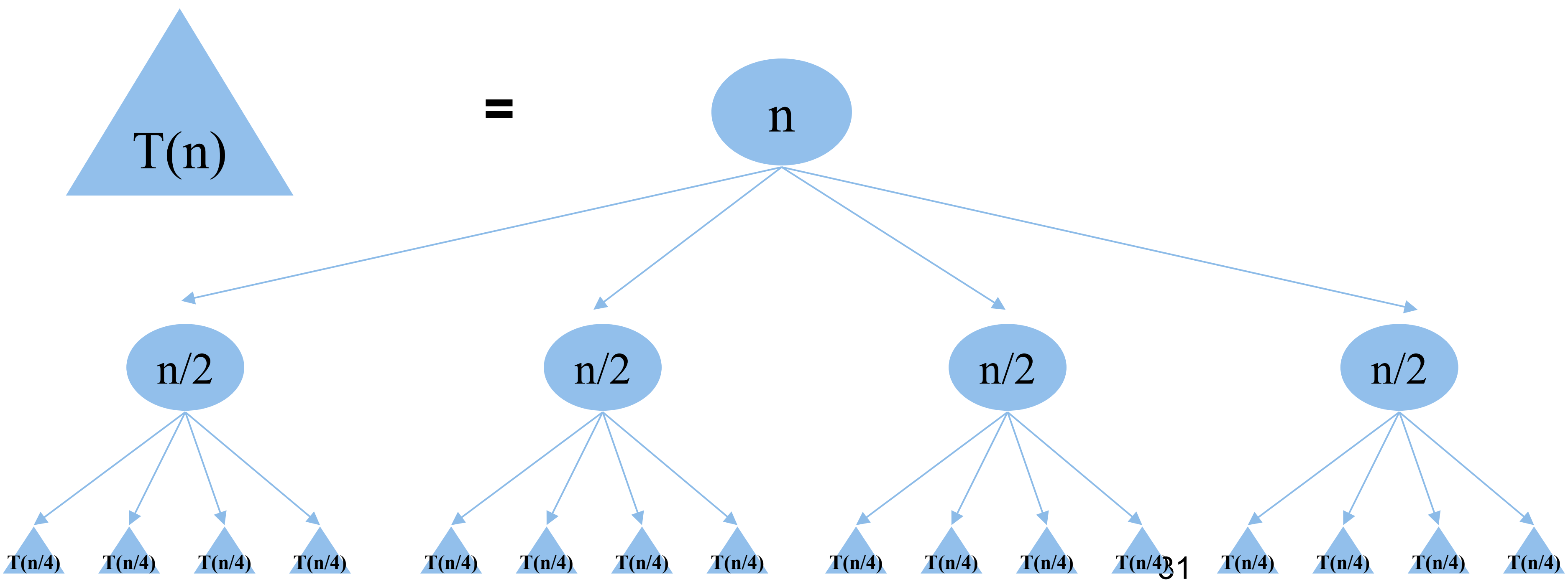
算法总体思想

- 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。



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：
  - 1. 该问题的规模缩小到一定的程度就可以容易地解决

因为问题的计算复杂性一般是随着问题规模的增加而增加，因此大部分问题满足这个特征。

- 例：排序算法
  - 给5个数排序
  - 给10^5个数排序

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：
  - 2. 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质

这条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用

- 例：排序算法
- 反例：一元n次问题

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：
  - 3. 利用该问题分解出的子问题解可以合并为该问题的解

能否利用分治法完全取决于问题是否具有这条特征，如果具备了前两条特征，而不具备第三条特征，则可以考虑贪心算法或动态规划。

- 例：排序算法
  - 合并两个有序的数组复杂度是O(n)

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：
  - 4. 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

- 例：Fibonacci

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
 }
```

- 在用分治法时，最好使子问题规模大致相同。
  - 这种使子问题规模大致相等的做法是出自一种平衡(balancing)子问题的思想，它几乎总是比子问题规模不等的做法要好。

分治法的复杂性分析

- 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：

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

本章要点

- 算法总体思想
- 递归的概念
- 递归案例分析
- 分治法的概念
- 分治案例分析
  - 二分搜索、大整数法、棋盘覆盖、合并排序、快速排序、线性时间选择 ...

分治-例1 二分搜索技术

- 问题的描述、分析
- 二分搜索算法
- 算法的正确性证明
- 性能分析

问题描述

- 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。
  - 若是，则找出x在表中的位置并返回其所在下标
  - 若非，则返回0值。

分治求解策略分析

定义问题的形式描述：$I=(n, a_1, a_2, ..., a_n, x)$

- 问题分解：选取下标k，由此将I分解为3个子问题：

  $I_1=(k-1, a_1, a_2, ..., a_{k-1}, x)$

  $I_2=(1, a_k, x)$

  $I_3=(n-k, a_{k+1}, a_2, ..., a_n, x)$

- 对于I_2，若$a_k=x$，则有解k，对I_1、I_3不用再求解；否则，
- 若$x<a_k$，则只在I_1中求解，对I_3不用求解；
- 若$x>a_k$，则只在I_3中求解，对I_1不用求解；

I_1、I_3上的求解可再次采用分治方法划分后求解（递归过程）

二分搜索算法

```
public static int binarySearch(int [] a, int x, int n)
  { // 在 a[0] <= a[1] <= ... <= a[n-1] 中搜索 x
   // 找到x时返回其在数组中的位置，否则返回-1
  int left = 0;  int right = n - 1;
   while (left <= right) {
      int middle = (left + right)/2;
      if (x == a[middle]) return middle;   //找到x, 返回位置
                                            //middle
      if (x > a[middle]) left = middle + 1;
      else right = middle - 1;
      }
  return -1; // 未找到x
  }
```

示例

例2.3.1：设a[0:8] = (-15，-6，0，7，9，23，54，82，101)

在a中检索x=101，-14，82。执行轨迹见下表

x=101			x=-14			x=82		
left	right	middle	left	right	middle	left	right	middle
0	8	4	0	8	4	0	8	4
5	8	6	0	3	1	5	8	6
7	8	7	0	0	0	7	8	7
8	8	8	1	0	找不到			找到
		找到						
成功的检索			不成功的检索			成功的检索		

分析

- 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。
  - 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。

算法的正确性证明

定理2.1 算法binarySearch(a,x,n)能正确运行

证明:

1）在具体指定a中的数据元素及x的数据类型后，算法中的所有运算都能按要求正确运行——即首先满足确定性和能行性

2）终止性

算法初始部分置left←0, right←n-1

① 若n=0，不进入循环，返回-1，算法终止

② 否则，进入循环，计算middle,

- 如果 x==a[middle]，返回middle，算法终止；
- 如果x> a[middle]，置left←middle+1，进入下次循环；搜索范围实际缩小为[middle+1, right]，对[left, middle-1]区间不做进一步搜索；
- 如果x< a[middle]，置right←middle-1，搜索范围实际缩小为[left, middle-1]，进入下次循环，对[middle+1, right]区间不做进一步搜索；

因left, right, middle都是整型变量，故按照上述规则，在有限步内，或找到某个middle，有a[middle] == x；或变得left>right，在a中没有找到任何元素等于x，算法终止。

性能分析

1） 空间特性

n+5个空间位置——O（n）

2） 时间特性

区分以下情况，并进行相应的分析

- 成功检索：所检索的x出现在a中。
  - 成功检索情况共有n种：x恰好是A中的某个元素，a中共有n个元素，故有n种可能的情况
- 不成功检索：所检索的x不出现在a中。
  - 不成功检索情况共有n+1种：
  - x<a[0],或a[i]<x<a[i+1],0≤i<n-2或x>a[n-1]
- 成功/不成功检索的最好情况：执行步数最少，计算时间最短
- 成功/不成功检索的最坏情况：执行步数最多，计算时间最长
- 成功/不成功检索的平均情况：一般情况下的计算时间

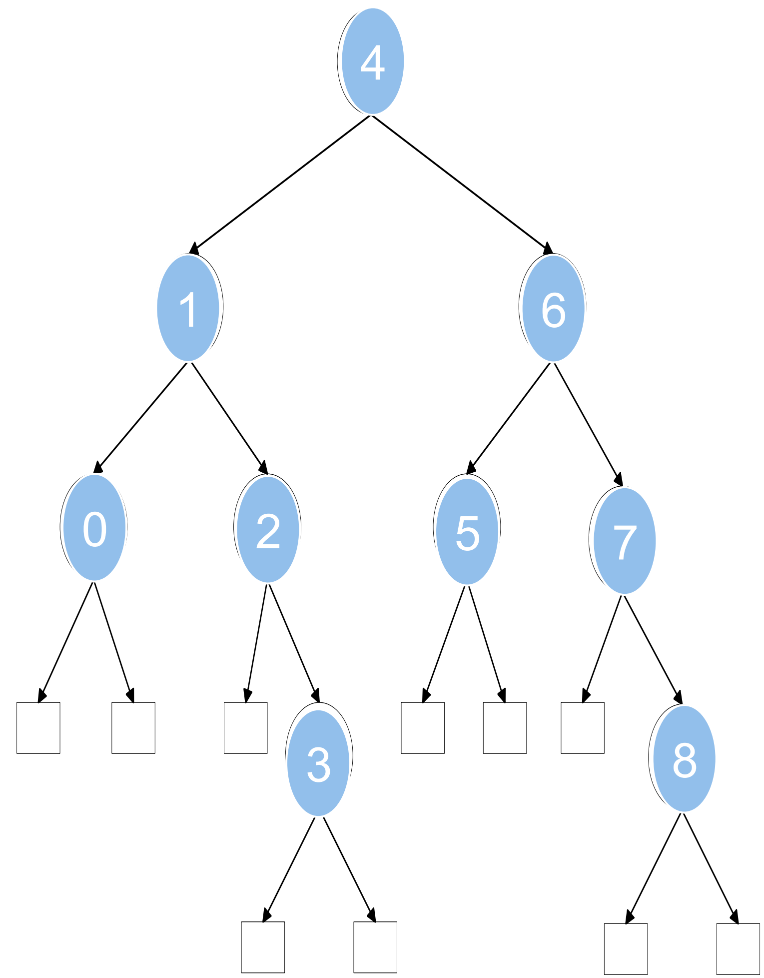
示例

- 频率计数特征
  - while循环体外语句的频率计数为1
  - 集中考虑while循环中的x与a中元素的比较（其它运算的频率计数与之有相同的数量级）
  - 假定只需一次比较就可确定if语句控制是三种情况的哪一种。查找每个元素所需的元素比较次数统计如下：

a	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	
元素	-15	-6	0	7	9	23	54	82	101	
成功检索比较次数	3	2	3	4	1	3	2	3	4	
不成功检索比较次数	3	3	3	4	4	3	3	3	4	4

二元比较树

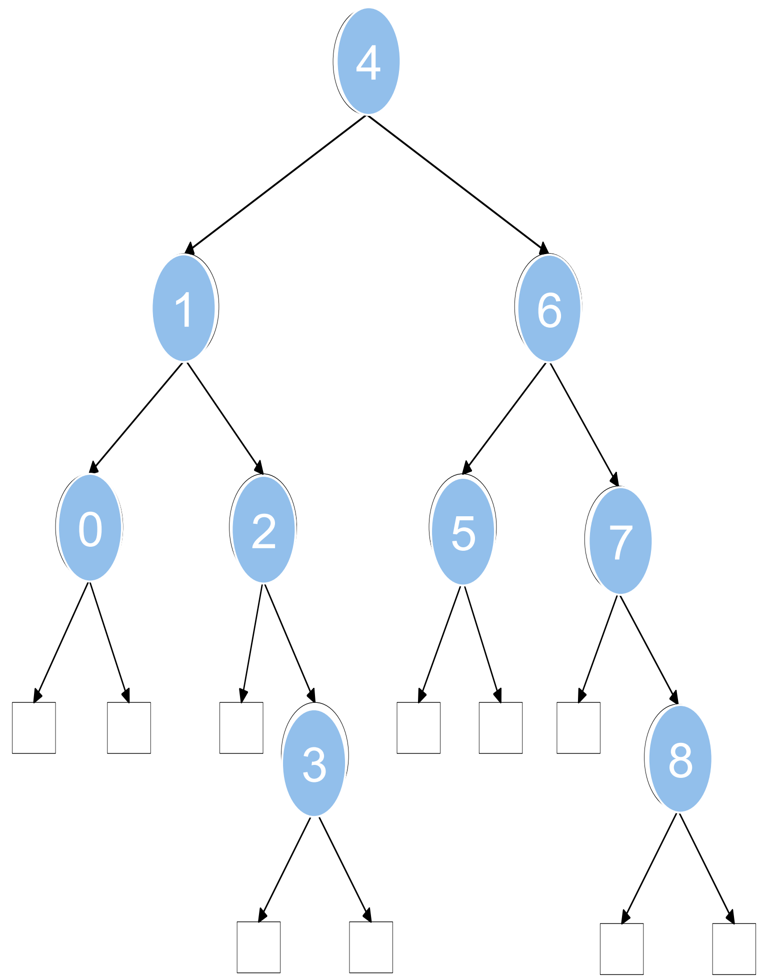
- 算法执行过程的主体是x与一系列中间元素a[middle]比较。可用一棵二元树描述这一过程，并称之为二元比较树。
- 构造：比较树由称为内结点和外结点的两种结点组成：
  - 内结点：表示一次元素比较，用圆形结点表示，存放一个middle值；代表一次成功检索；
  - 外结点：在二分检索算法中表示一种不成功检索的情况，用方形结点表示。
  - 路　径：表示一个元素的比较序列。



基于二元比较树的分析

- 若x在a中出现，则算法的执行过程在一个圆形的内结点处结束。
- 若x不在a中出现，则算法的执行过程在一个方形的外结点处结束

——外结点不代表元素的比较，因为比较过程在该外结点的上一级的内结点处结束。



基于二元比较树分析时间复杂度

1）不成功检索的最好、最坏和平均情况的计算时间均为 $\Theta(\log n)$ ——外结点处在最末的两级上；

2）最好情况下的成功检索的计算时间为 $\Theta(1)$

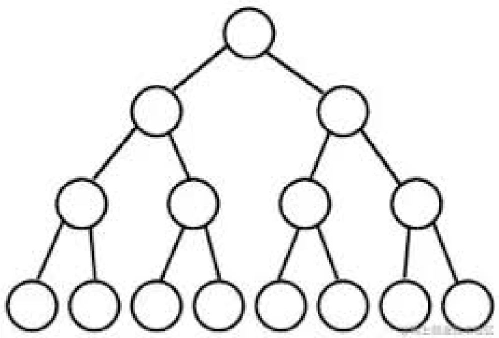
最坏情况下的成功检索的计算时间为 $\Theta(\log n)$

（注：对数均以2为底）

基于二元比较树分析时间复杂度

3）平均情况下的成功检索的计算时间分析

利用外部结点和内部结点到根距离和之间的关系进行推导：



- 由根到所有内结点的距离之和称为内部路径长度，记为I；
- 由根到所有外部结点的距离之和称为外部路径长度，记为E。

则有，$E=I+2n$

证明：$I=1*2^1+2*2^2+\ldots\ldots+k*2^k$

$E=(k+1)*2^{k+1}$

$E-I=(k+1)*2^{k+1}-(1*2^1+2*2^2+3*2^3\ldots\ldots+k*2^k)$

设 $F(k)=E-I$

$F(k)/2=(k+1)*2^k-(1*2^0+2*2^1+\ldots\ldots+k*2^{k-1})$

$F(k)-F(k)/2=(k+1)*2^{k+1}+2^0+2^1+\ldots+2^{K-1}-(k+1)*2^K-k*2^k$

$=(K+1)*2^K+2^0+2^1+\ldots+2^{K-1}-k*2^k$

$=2^0+2^1+\ldots+2^{K-1}+2^k=n$

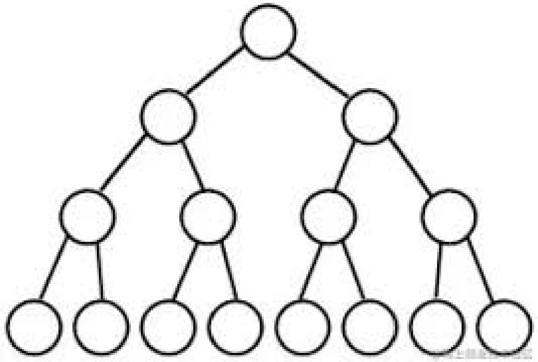
$F(k)=E-I=2n$ $E=I+2n$

基于二元比较树分析时间复杂度

- U(n)是平均情况下不成功检索的计算时间，则$U(n) = E/(n+1)$
- S(n)是平均情况下成功检索的计算时间，则$S(n) = I/n+1$
- 利用上述公式，可有: $S(n) = (1+1/n)U(n) -1$

当$n\to\infty$, $S(n) \propto U(n)$ ，而$U(n) =\Theta(\log n)$

所以 $S(n) = \Theta(\log n)$



分治-例2 大整数乘法

- 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：$O(n^2)$
  - 分治法：

X=

n/2位	n/2位
a	b

Y=

n/2位	n/2位
c	d

- $X = ab$
- $Y = cd$
- $X = a \cdot 10^{n/2} + b$
- $Y = c \cdot 10^{n/2} + d$
- $XY = (a*c) \cdot 10^{n} + (a*d+b*c) \cdot 10^{n/2} + b*d$

时间复杂度分析

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

- $T(n) = 4T(n/2) = 4^2T(n/2^2) = \ldots = O(4^{\log_2 n})$
- 对数运算规则 $a^{\log_b c} = c^{\log_b a}$
- $T(n) = O(4^{\log_2 n}) = O(n^{\log_2 4}) = O(n^2)$

×没有改进☹

简单实例

- $23 * 14=332$
- $23=2 \cdot 10^1+3 \cdot 10^0$　　　　$14=1 \cdot 10^1+4 \cdot 10^0$
- $23 * 14= (2 \cdot 10^1+3 \cdot 10^0)*(1 \cdot 10^1+4 \cdot 10^0)$
- $=(2*1) \cdot 10^2+(3*1+2*4) \cdot 10^1 +(3*4) \cdot 10^0$
- $3*1+2*4=(2+3)*(1+4)- (2*1)- (3*4)$
- 4次乘法减为3次乘法

改进大整数乘法

- 分治法的改进:
  - $XY = ac \cdot 10^n + (ad+bc) \cdot 10^{n/2} + bd$
  - 为了降低时间复杂度，必须减少乘法的次数。
  - $XY = ac \cdot 10^n + ((a-c)(b-d)+ac+bd) \cdot 10^{n/2} + bd$
  - $XY = ac \cdot 10^n + ((a+c)(b+d)-ac-bd) \cdot 10^{n/2} + bd$

细节问题：两个XY的复杂度都是O(nlog3)，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。

时间复杂度分析

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

$T(n)=O(n^{\log 3})\ =O(n^{1.585})$ ✓**较大的改进**☺

大整数乘法总结

- 小学的方法：$O(n^2)$　　×效率太低
- 分治法: $O(n^{1.585})$　　✓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，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，对于大整数乘法，它能在O(nlogn)时间内解决。
- 是否能找到线性时间算法？目前为止还没有结果。

小结-分治法时间复杂度分析

- 递归树方法
  - 二分搜索算法
- 主定理法

$$T(n)=\begin{cases} c & n=1 \\ aT\left(\dfrac{n}{b}\right)+cn & n>1 \end{cases}$$

$$T(n)=\begin{cases} \Theta(n) & a<b \\ \Theta(n\log_b n) & a=b \\ \Theta\left(n^{\log_b a}\right) & a>b \end{cases}$$

- 代换法（数学归纳）