课程内容

NP完全性理论与近似算法	算法高级理论
随机化算法 / 线性规划与网络流	高级算法
递归分治 / 动态规划 / 贪心算法 / 回溯与分支限界	基础算法
算法分析与问题的计算复杂性	算法基础理论

第二章 递归与分治策略

学习要点

- 理解分治和递归的概念。
- 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。
  - 二分搜索技术；
  - 大整数乘法；
  - Strassen矩阵乘法；
  - 棋盘覆盖；
  - 合并排序和快速排序；
  - 线性时间选择；
  - 最接近点对问题；
  - 循环赛日程表。

分治法的初衷

- 任何一个问题的求解时间都与其规模有关。
- 例子：
  - n个元素排序：

    当n=1，不需计算；

    当n=2，只作一次即可；

    当n=3，三次or两次？ …

显然，随着n的增加，问题也越难处理。

分治法

- 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。
- 如果问题可分割成*k*个子问题，且这些子问题都可解，利用这些子问题可解出原问题的解，此分治法是可行的。
- 由分治法产生的子问题往往是原问题的较少模式，为递归提供了方便。

递归

- **定义：直接/间接调用自身的算法称为递归算法。**

阶乘函数

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

int Factorial(int *n*)
    if (n==0)
        return 1
    return *n*×Factorial$\boxed{(n\text{-}1)}$

递归第一式给出函数的初值，非递归定义。每个递归须有非递归初始值。

第二式是用较小自变量的函数值表示较大自变量

递归

- **前面提到的 Fibonacci数列**

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

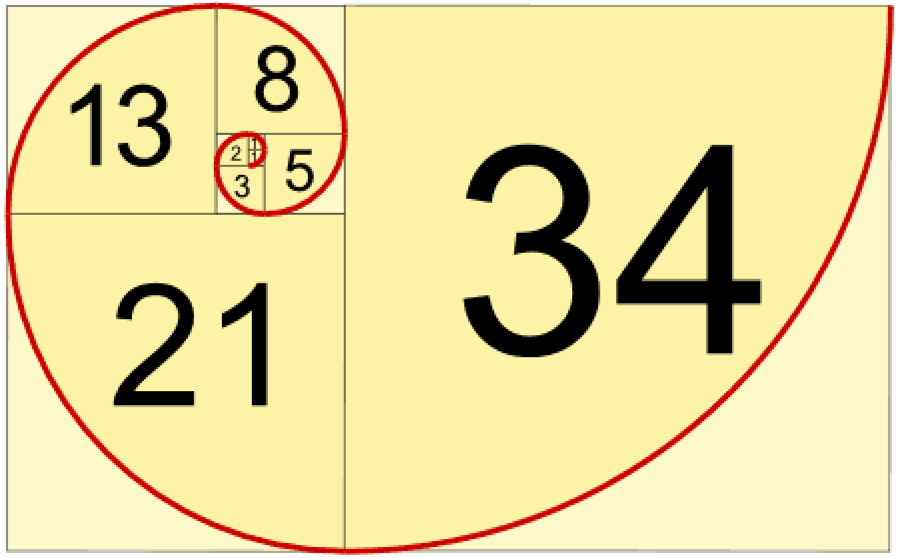
意大利数学家
Fibonacci
1170-1240

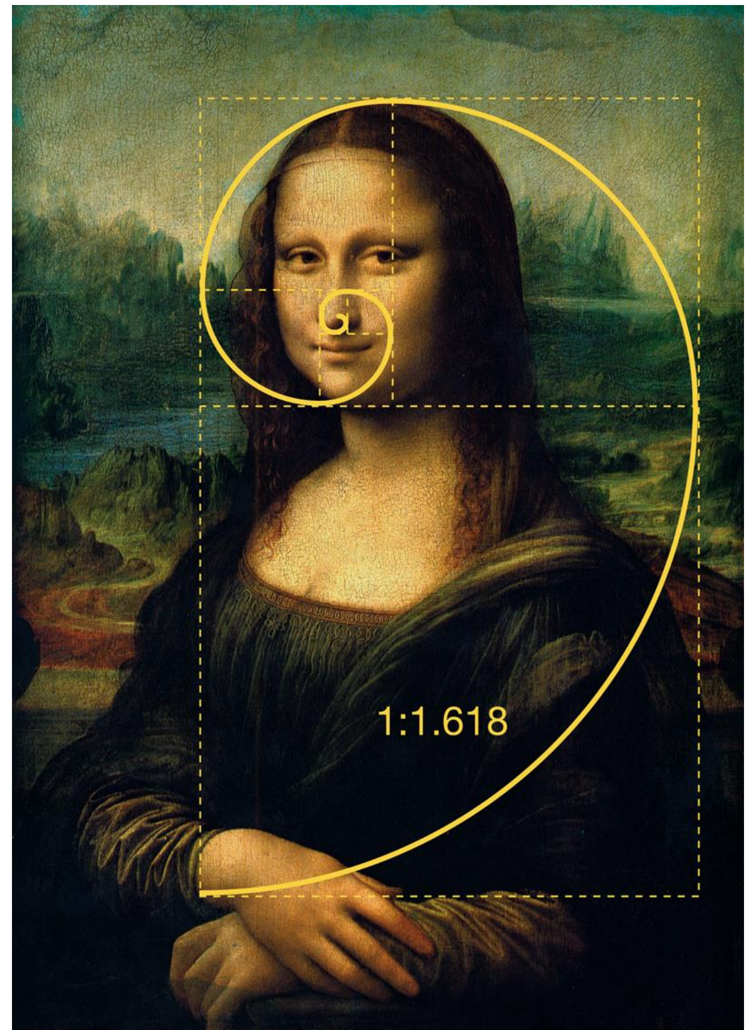
上述函数也可用非递归方式定义：

$$n!=1\times2\times3\times\cdots\times(n-1)\times n$$

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

Fibonacci数列





双递归函数

- 一个函数与它的一个变量是由函数自身定义。

Ackerman函数

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

1. $m=0,\ A(n,0)=n+2$
2. $m=1,\ A(n,1)=A(A(n-1,1),0)=A(n-1,1)+2 \qquad \therefore A(n,1)=2n$
3. $\left.\begin{array}{l} m=2,\ A(n,2)=A(A(n-1,2),1)=2A(n-1,2) \\ \because A(1,2)=A(A(0,2),1)=A(1,1)=2 \end{array}\right\} \Rightarrow A(n,2)=2^n$

双递归函数

4. $m=3$, $A(n,3)=2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ ，其中2的层数n
5. $m=4$, $A(n, 4)$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。

- **部分书上减少Ackerman函数的变量，如：**

$$A(n) \stackrel{\Delta}{=} A(n,n)$$

$$A(4)=2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$$ **（其中2的层数为65536）**

这个数非常大，无法用通常的方式来表达它！

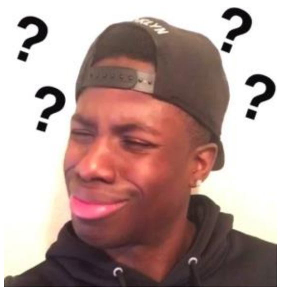
整数划分问题

- **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$n_1\geqslant n_2\geqslant\ldots\geqslant n_k\geqslant 1$，$k\geqslant 1$。
- **正整数n的这种表示称为正整数n的划分。**
  - **问题：** 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。

整数划分问题

- 仅仅考虑一个自变量：
  - 设$p(n)$为正整数n的划分数，但难以找到递归关系



- 考虑两个自变量：
  - 将最大加数n_1不大于m的划分个数记作$q(n, m)$。

整数划分问题

- 可以建立$q(n, m)$的如下递归关系。

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

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

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

最大加数n_1实际上不能大于n。因此，$q(1, m)=1$。

(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) $q(n, m)=\underline{q(n-m, m)}+\underline{q(n, m-1)},\ n>m>1;$

正整数n的最大加数n_1不大于m的划分由$\boxed{n_1=m\text{的划分}}$和$\boxed{n_1\leq m-1}$的划分组成

整数划分问题

- **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*)

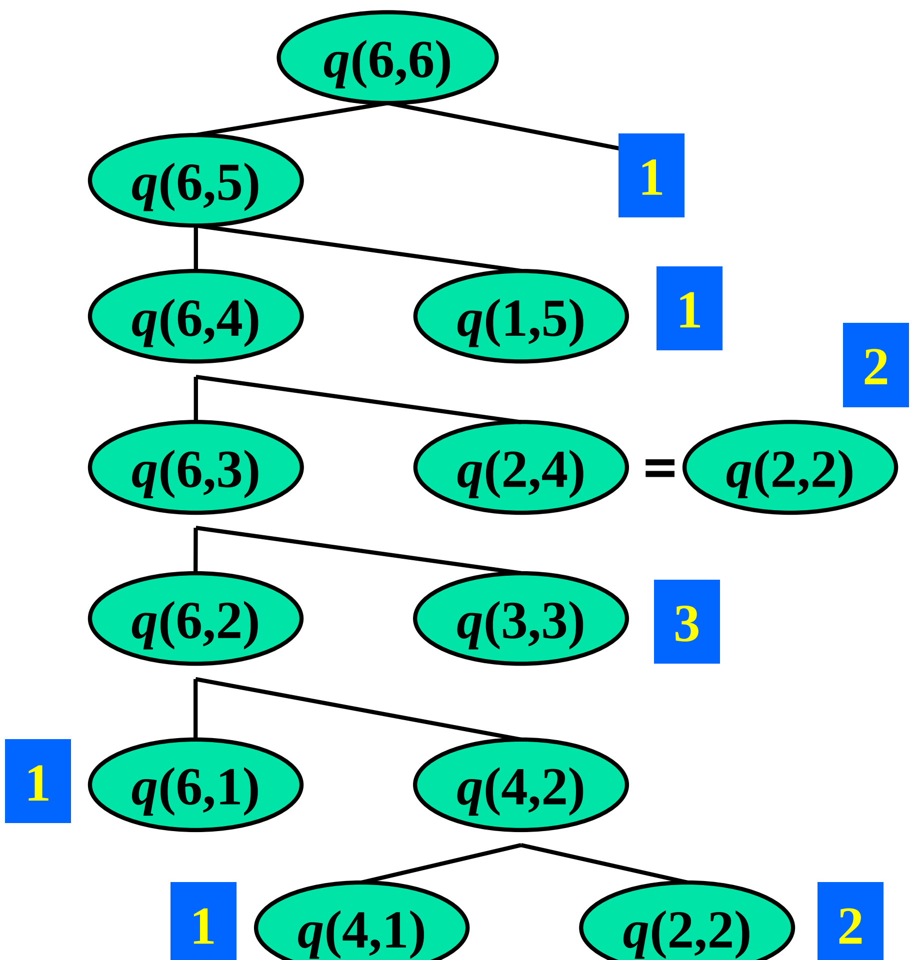
整数划分问题

- 可得算法伪代码为:

```
int q(int n, int m){
    if ((n-1)||(m<1))  return 0;
    if ((n==1)||(m==1)) return 1;
    if (n<m) return q(n, n);
    if (n==m) return q(n, m-1)+1;
    return q(n, m-1)+q(n-m, m);
}
```

整数划分问题的递归调用

运行轨迹




```
int q(int n, int m){
    if ((n-1)||(m<1))
        return 0;
    if ((n==1)||(m==1))
        return 1;
    if (n<m)
        return q(n, n);
    if (n==m)
        return q(n, m-1)+1;
    return q(n, m-1)+q(n-m, m);
}
```

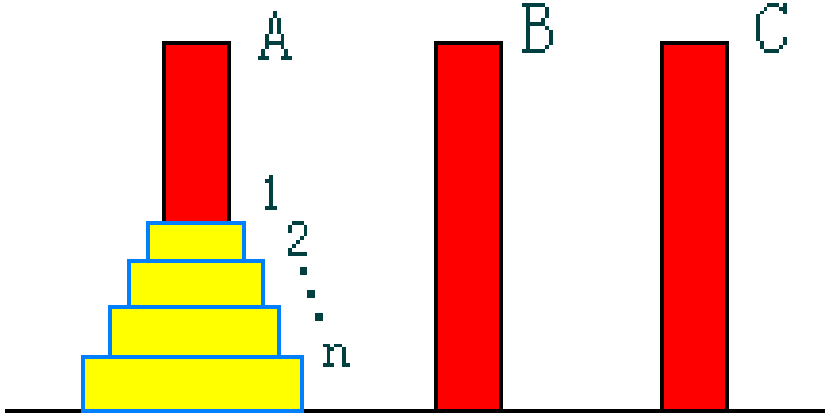
$q(6,6)=1+1+2+3+1+1+2=11$

Hanoi塔传说

在世界刚被创建的时候，有一座钻石宝塔(塔A)，其上有64个金碟。所有碟子按从大到小的次序从塔底堆放至塔顶。紧挨着这座塔有另外两个钻石宝塔(塔B和塔C)。从世界创始之日起，婆罗门的牧师们就一直在试图把塔A上的碟子移动到塔C上去，其间借助于塔B的帮助。每次只能移动一个碟子，任何时候都不能把一个碟子放在比它小的碟子上面。当牧师们完成任务时，世界末日也就到了。

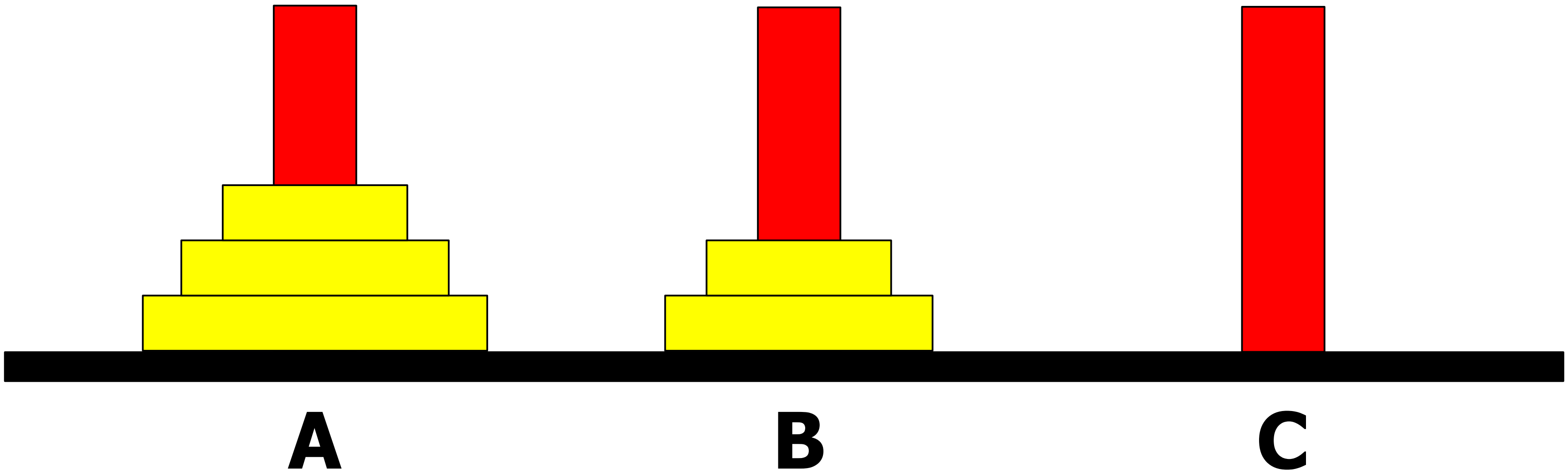
Hanoi塔问题

- 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**：满足规则1和2的前提下，可将圆盘移至A,B,C任一塔座上。



Hanoi塔问题

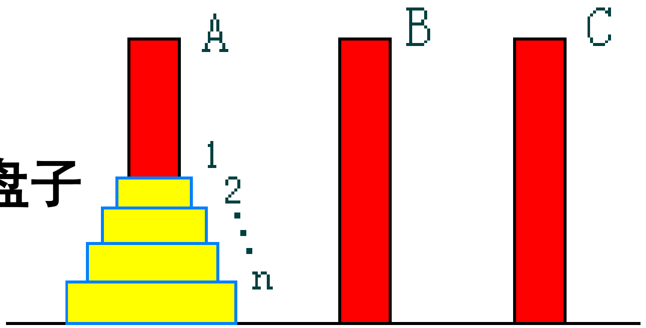
- 先来看一个简单的实例



递归算法

- 算法Hanoi(*n*, A, B, C) //*n*个盘子借助B, 从A移到C

1. **if** *n* = 1 **then** move (A, C)
2. **else** Hanoi(*n*-1, A, C, B)
3. move (A, C)//剩下的一个盘子
4. Hanoi(*n*-1, B, A, C)

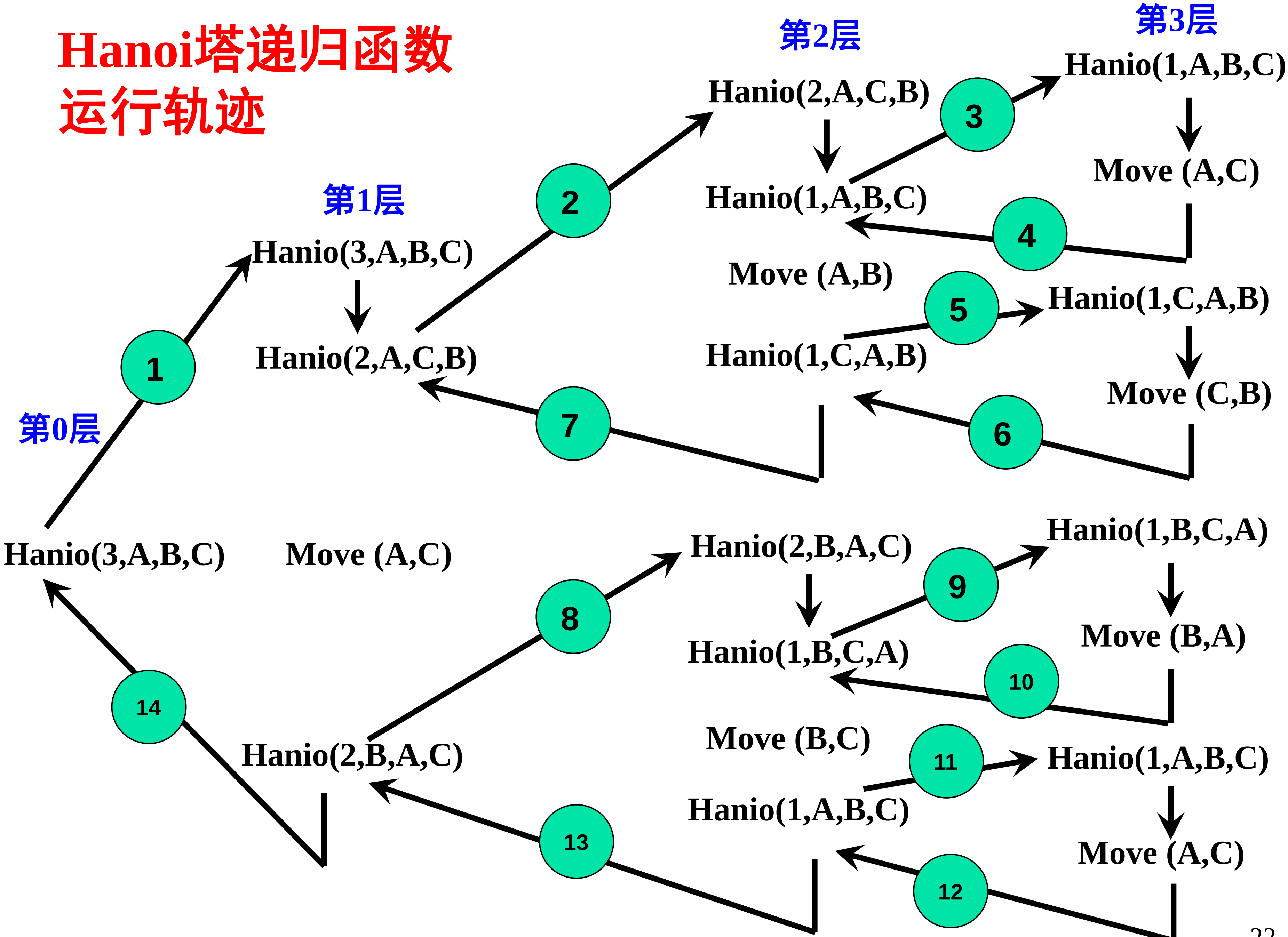


- **优点：** 简单、优雅、易于理解；
- **缺点：** 算法Hanoi以递归形式给出，每个圆盘的具体移动方式并不清楚，因此当$n \geqslant 5$以后，很难用手工移动来模拟这个算法。

递归函数的运行轨迹

- 在递归函数中，调用函数和被调用函数是同一个函数，需要注意的是递归函数的调用层次，如果把调用递归函数的主函数称为**第0层**，进入函数后，首次递归调用自身称为**第1层**调用；从第i层递归调用自身称为**第i+1层**。反之，退出第i+1层调用应该返回第i层。
- 采用**图示方法**描述递归函数的**运行轨迹**，从中可较直观地了解到各调用层次及其执行情况。

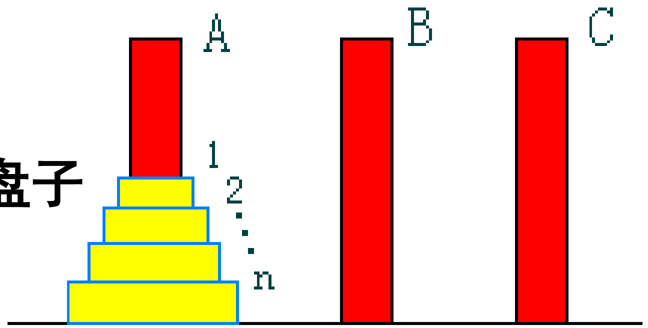
Hanoi塔递归函数运行轨迹



递归算法

- 算法Hanoi(n, A, B, C) //n个盘子借助B, 从A移到C

1. if n = 1 then move (A, C)
2. else Hanoi(n-1, A, C, B)
3. move (A, C)//剩下的一个盘子
4. Hanoi(n-1, B, A, C)

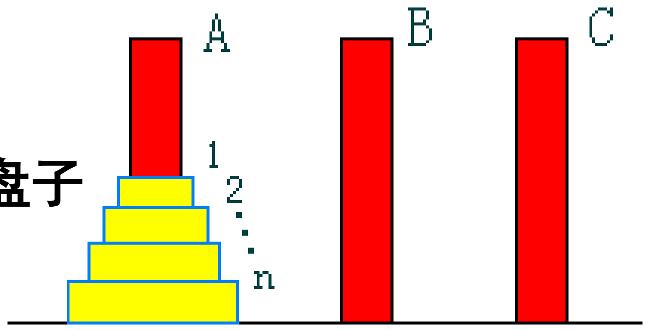


如果用了递归，就找不到while/for循环语句。
此时，该如何计算时间复杂度呢？

递归算法

- 算法Hanoi(n, A, B, C) //n个盘子借助B, 从A移到C

```
if n = 1 then move (A, C)
else Hanoi(n-1, A, C, B)
        move (A, C)//剩下的一个盘子
        Hanoi(n-1, B, A, C)
```



设n个盘子的移动次数为T(n)

$$T(n)=2T(n-1)+1$$

$$T(1)=1$$

→ 递推方程

递推方程

- 设序列$a_0,a_1,\ldots,a_n,\ldots$简记为$\{a_n\}$，一个把$a_n$与某些个$a_i(i<n)$联系起来的等式叫做关于序列$\{a_n\}$的递推方程

- **递推方程的求解：**
  给定关于序列$\{a_n\}$的递推方程和若干初值，计算a_n

迭代法求解递推方程

- 不断用递推方程的右部替换左部
- 每次替换，随着n的降低在和式中多出一项
- 直到出现初值停止迭代
- 将初值代入并对和式求和
- 可用数学归纳法验证解的正确性

Hanoi塔算法

$$T(n)=2T(n-1)+1$$
$$T(1)=1$$

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2[2T(n-2)+1]+1 \\
&= 2^2T(n-2)+2+1 \\
&= \ldots\ldots \\
&= 2^{n-1}T(1)+2^{n-2}+2^{n-3}+\ldots+2+1 \\
&= 2^{n-1}+2^{n-1}-1 \\
&= 2^n-1
\end{aligned}
$$

解的正确性-归纳验证

- 证明：下述递推方程的解是$T(n)=2^n-1$
  - $T(n)=2T(n-1)+1$
  - $T(1)=1$

- 方法：数学归纳法
- 证　$n=1$, $T(1)=2^1-1=1$

  假设对于n，解满足方程，则

  $T(n+1)$

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

Hanoi塔算法

$$T(n)=2T(n-1)+1$$
$$T(1)=1$$

$$
\begin{aligned}
T(n)&=2T(n-1)+1\\
&=2[2T(n-2)+1]+1\\
&=2^2T(n-2)+2+1\\
&=\ldots\ldots\\
&=2^{n-1}T(1)+2^{n-2}+2^{n-3}+\ldots+2+1\\
&=2^{n-1}+2^{n-1}-1\\
&=2^n-1
\end{aligned}
$$

5000亿年！！！

问题： 如果1秒移动1个，64个金蝶要多少时间？

每秒能算60万亿次

要计算三天左右

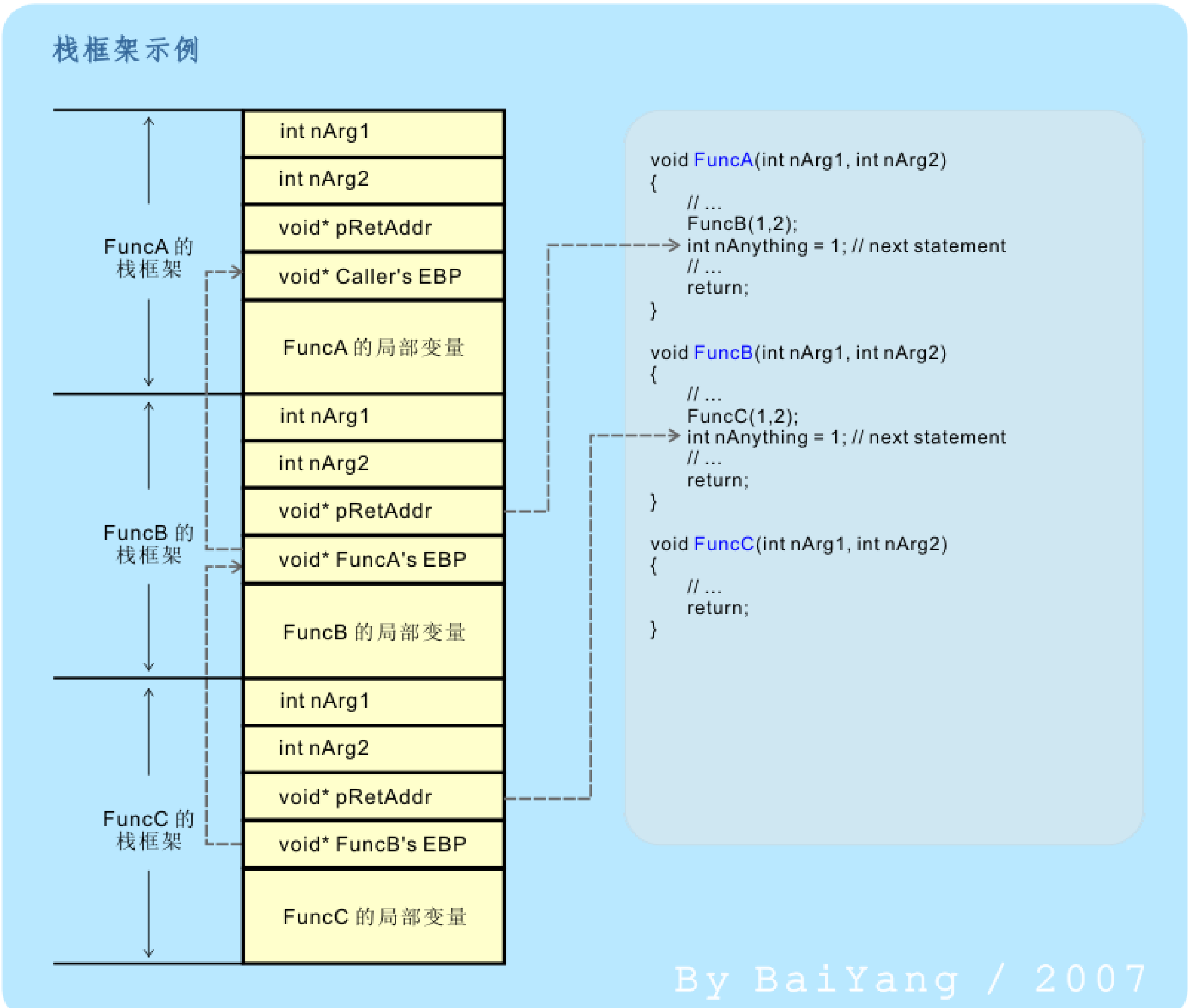
Hanoi塔算法

有没有更好的算法???

没有！！！

这是一个难解的问题，不存在多项式时间的算法！

递归的函数调用（回顾C语言）



递归小结

- 实现递归调用的关键是建立递归调用工作栈。在调用算法之前：
  - 将所有实参指针，返回地址等信息传递给被调用算法；
  - 为被调算法的局部变量分配存储区；
  - 将控制移到被调算法的入口。
- 返回调用算法时，系统要完成：
  - 保存被调算法的结果；
  - 释放分配给被调用算法的数据区；
  - 依保存的返回地址将控制转移到调用算法。

…
算法A第二层递归调用
算法A第一层递归调用
主算法递归调用算法A
…
主算法栈块

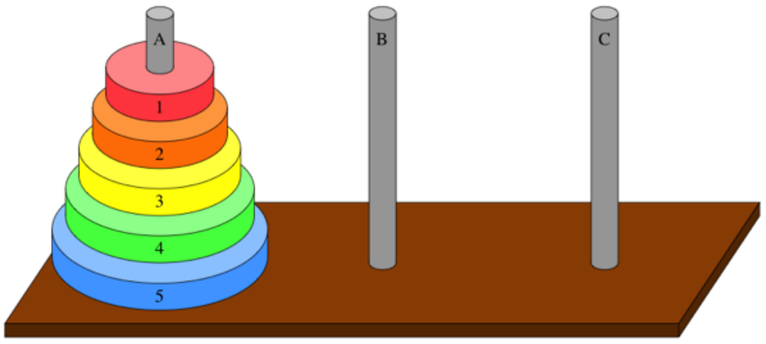
递归小结(cont.)

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。

要点回顾

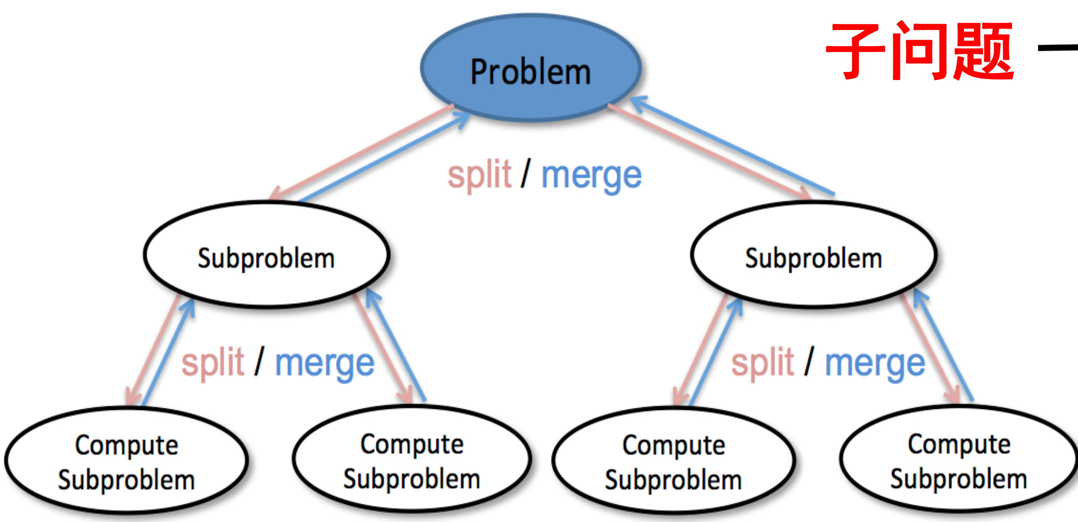
- 递归算法
  - 概念（阶乘、Fibonacci数列、双递归）
  - 例子（整数划分问题、Hanoi塔问题）
  - Hanoi塔算法、运行轨迹、分析时间复杂度
  - 递推方程（迭代法求解）
  - 递归的优缺点

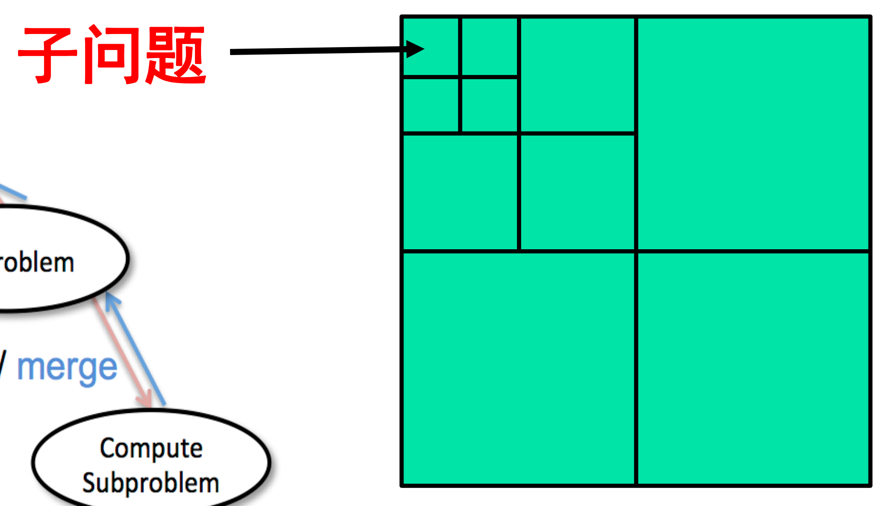


分治策略

分治法(Divide-and-Conquer)

基本思想：将一个规模为n的问题分解为k个规模较小的子问题，这些子问题**互相独立**且与原问题**相同**。递归解这些子问题，再将子问题合并得到原问题的解。





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：

- 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。

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，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。

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，即该问题具有**最优子结构性质**
- **利用该问题分解出的子问题的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。

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，即该问题具有**最优子结构性质**
- 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。

该特征涉及到分治法效率，如果各子问题不独立，则分治法要做许多不必要的工作，重复地解公共子问题，此时虽也可用分治法，但一般用动态规划较好。

分治法的基本步骤

伪代码：

```
divide-and-conquer(P){
    if ( |P| <= n0) adhoc(P);    //(1)解决小规模的问题
    divide P into smaller sub instances P1,P2,...,Pk;
    //(2)分解问题
    for (i=1,i<=k,i++)
        yi=divide-and-conquer(Pi);   //(3)递归地解各子问题
    return merge(y1,...,yk);//(4)将各子问题的解合并为原问题的解
}
```

- **其中，| P |是问题P的规模，n_0为一阈值，表示当问题P的规模不超过n_0时，问题已容易解决，不必再继续分解。**
- **adhoc(P)是分治的基本子算法，用于直接解小规模的问题P。**
- **merge(y_1, y_2, ... , y_k)是分治算法中的合并子算法。**

分治法的问题

- 将问题分为多少个子问题？
- 子问题的规模是否相同/怎样才恰当？

- 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叫做**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。

二分搜索技术

问题： 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：

1. 该问题的规模缩小到一定的程度就可以容易地解决；

如果$n=1$即只有一个元素，则只要比较这个元素和x就可以确定x是否在表中。因此这个问题满足分治法的第一个适用条件

二分搜索技术

问题： 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：

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. 分解出的子问题的解可以合并为原问题的解；

比较x和中间元素$a[mid]$，若$x=a[mid]$，则x在L中的位置就是mid；如果$x<a[mid]$，由于a是递增排序的，因此假如x在a中的话，x必然排在$a[mid]$的前面，所以只要在$a[mid]$的前面查找x即可；如果$x>a[i]$，同理只要在$a[mid]$的后面查找x即可。无论是在前面还是后面查找x，其方法都和在a中查找x一样，只不过是查找的规模缩小了。这就说明了此问题满足分治法的第二、三个适用条件

二分搜索技术

问题： 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：

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. 分解出的子问题的解可以合并为原问题的解；
4. 分解出的各个子问题是相互独立的。

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。

算法分析与设计

Analysis and Design of Algorithm

Lesson 05

要点回顾

- 递归法
  - 例子：整数划分、汉诺塔
- 递推方程的求解方法
  - 迭代法
  - 数学归纳法验证
- 分治策略
  - 基本思想、适用条件、基本步骤
  - 分治效率分析：给出了通用计算公式
  - 例子：二分搜索

二分搜索技术

问题： 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：

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. 分解出的子问题的解可以合并为原问题的解；
4. 分解出的各个子问题是相互独立的。

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。

二分搜索算法

据此容易设计出二分搜索算法的伪代码：

```
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;
}
```

算法复杂度分析：

每执行一次算法中的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$O(\log n)$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$O(\log n)$。

二分搜索算法(cont.)

- 二分搜索算法易于理解。
- 编写正确的二分搜索算法不易，90%人在2个小时内不能写出完全正确的二分算法。

第一个二分搜索算法在1946年提出，但是第一个完全正确的二分搜索算法却直到1962年才出现。

大整数的乘法

- 在复杂性计算时，都将加法和乘法运算当作基本运算处理，即加、乘法时间为常数。但上述假定仅在参加运算整数能在计算机表示范围内直接处理时才合理。
- 那么我们处理很大的整数时，怎么办？
- 要精确地表示大整数，并在计算结果中精确到所有位数，就必须用软件方法实现。

大整数的乘法(cont.)

问题：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)　　✖ 效率太低了！！！

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

大整数的乘法(cont.)

问题：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)** ✖ **效率太低了！！！**
- **分治法：**

假设：X和Y都是*n*位二进制整数

	n/2位	n/2位
X=	a	b

	n/2位	n/2位
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

则计算X，Y，需4次$n/2$位整数乘法(ac，ad，bc和bd)

需3次小于$2n$位的整数加

需2次移位(2^n, $2^{n/2}$)

共需O(n)

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

直接用之前的迭代法解有点点复杂！

换元迭代法

- 将对n的递推式换成对其他变元k的递推式
- 对k直接迭代
- 将解(关于k的函数)转换成关于n的函数

举例： 解如下递推方程

$$T(n)=2T(n/2)+n-1$$
$$T(1)=0$$

换元： 令$n=2^k$，则递推方程变换为

$$T(2^k)=2T(2^k/2)+2^k-1 =2T(2^{k-1})+2^k-1$$
$$T(0)=0$$

换元迭代法

$T(2^k)=2T(2^{k-1})+2^k-1$

$T(0)=0$

迭代求解：

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

$$=2[2T(2^{k-2})+2^{k-1}-1]+2^k-1$$

$$=2^2T(2^{k-2})+2^k-2+2^k-1$$

$$=\ldots$$

$$=2^kT(1)+k2^k-(2^{k-1}+2^{k-2}+\ldots+2+1)$$

$$=k2^k-2^k+1$$

$$=n\log n-n+1$$

最后，需用数学归纳法证明正确性！

公式法

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

基于换元迭代法，可以得到递推方程的解：

$$T(n)=n^{\log_b a}+\sum_{i=0}^{\log_b n-1} a^i f(n/b^i)$$

复杂度分析

则计算X，Y，需4次n/2位整数乘法(ac，ad，bc和bd)

需3次小于2n位的整数加

需2次移位(2^n, $2^{n/2}$)

共需O(n)

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

为了简化，设n为2的幂

换元迭代法$n=2^x$

$$T(1)=O(1)$$

$$T(2)=4O(1)+O(2) \rightarrow 4O(1)$$

$$T(4)=4(4O(1)+O(2))+O(4) \rightarrow 4^2O(1)$$

$$T(8)=4(4(4O(1)+O(2))+O(4))+O(8) \rightarrow 4^3O(1)$$

复杂度分析

$$T(16) = \cdots \quad \rightarrow \quad 4^4 O(1)$$

$$\cdots$$

$$T(2^x) = \cdots \quad \rightarrow \quad 4^x O(1)$$

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

公式法求解该算法复杂度：

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

$\rightarrow T(n) = O(n^2)$ ✖ 没有改进！！！

大整数的乘法(cont.)

为了降低时间复杂度，必须减少乘法的次数！！！

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

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

对于1式：需3次*n*/2位整数乘法((a-b)(d-c),ac,bd)
需6次加、减
需2次移位

大整数的乘法(cont.)

复杂度分析:

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

➔ $T(n)=O(n^{\log 3})=O(n^{1.59})$ ✓**改进了!!!**

细节问题: 两个XY的复杂度都是$O(n^{\log 3})$，但考虑到$a+b$, $c+d$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。

大整数的乘法(cont.)

- 还有没有更快的方法? $T(n)=O(n^{\log 3}) = O(n^{1.59})$
- 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。

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$
1966	Toom-Cook	$\Theta(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$\Theta(n \log n \log \log n)$
2007	Fürer	$n \log n \, 2^{O(\log^* n)}$

是否能找到线性时间算法? 目前为止还没有结果。

递归树法验证换元迭代法

- 递归树的概念
  - 递归树是迭代计算的模型（迭代的图形表示）
  - 递归树的生成过程与迭代过程一致
  - 树上所有项恰好是迭代之后产生和式中的项
  - 对递归树上的项求和就是迭代后方程的解

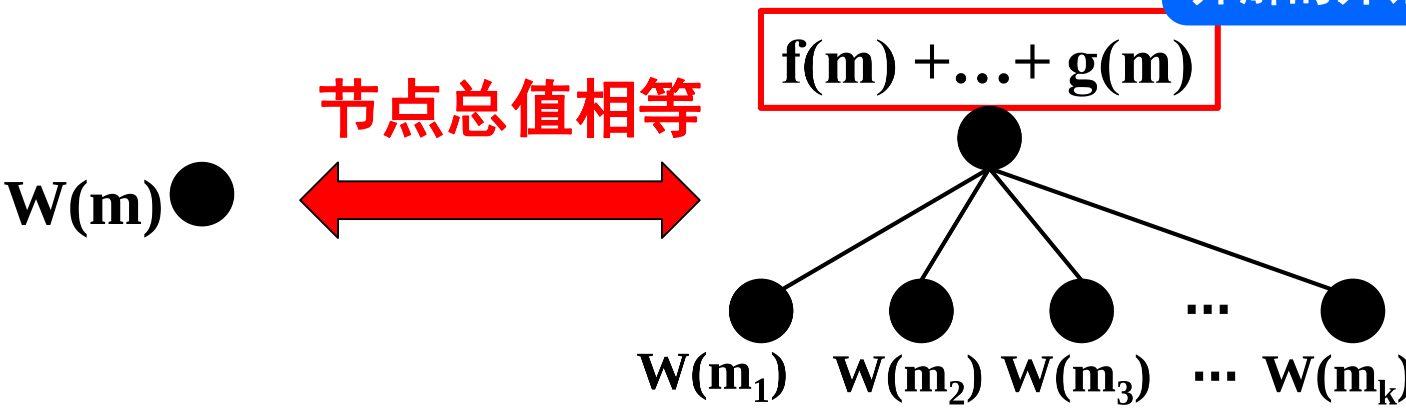
递归树法验证换元迭代法

- 迭代在递归树中的表示
  - 如果递归树上某节点标记为W(m)
  - $W(m)= \boxed{W(m_1) +\ldots+ W(m_k)}$
    $+ \boxed{f(m) +\ldots+ g(m)}, m_1,\ldots, m_k<m$

递归求解划分后的子问题的开销

归约到子问题及合并的开销

其中$W(m_1),\ldots,W(m_k)$称为函数项

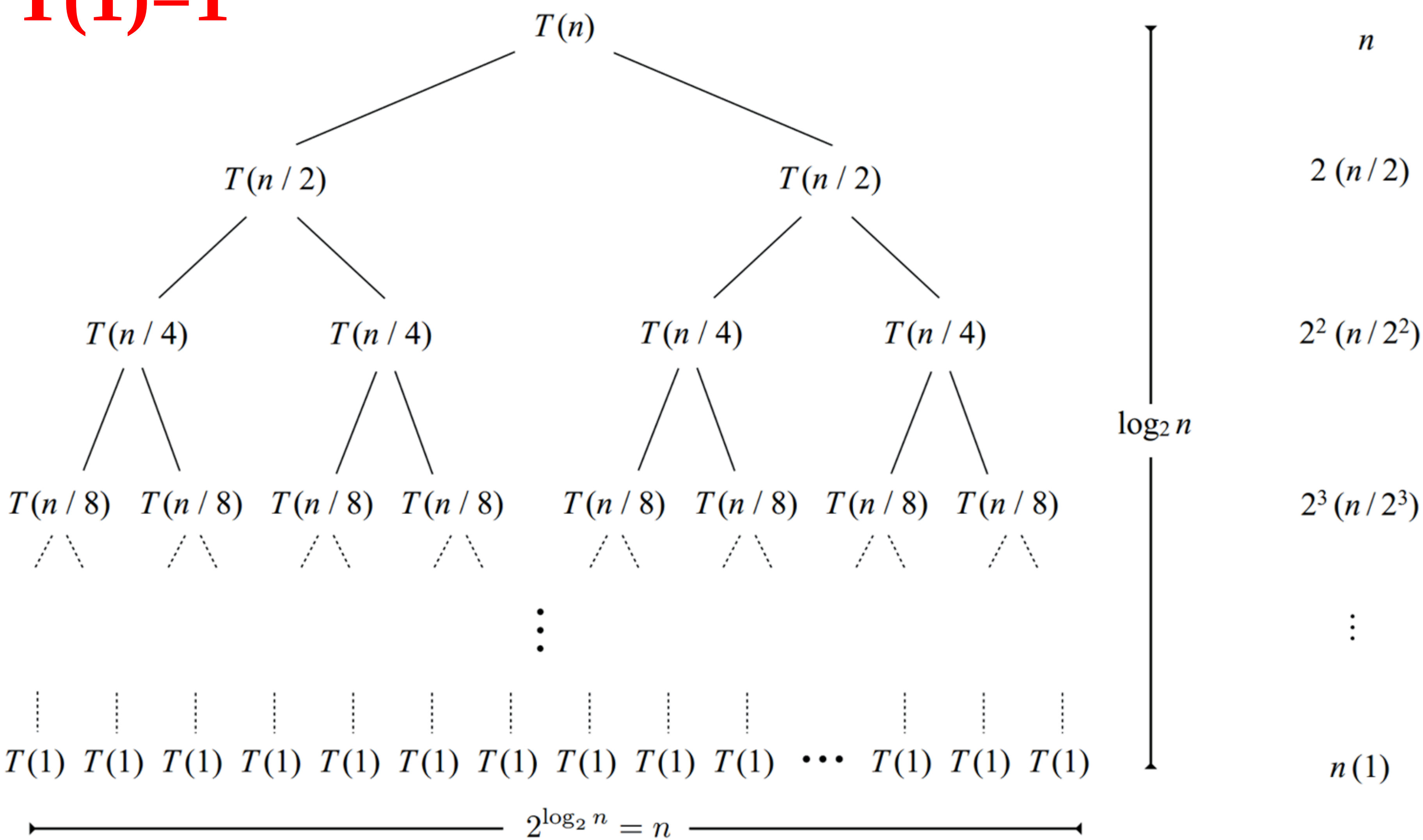


每个叶节点是一个函数项

例子： 画出$T(n)=2T(n/2)+n$递归树

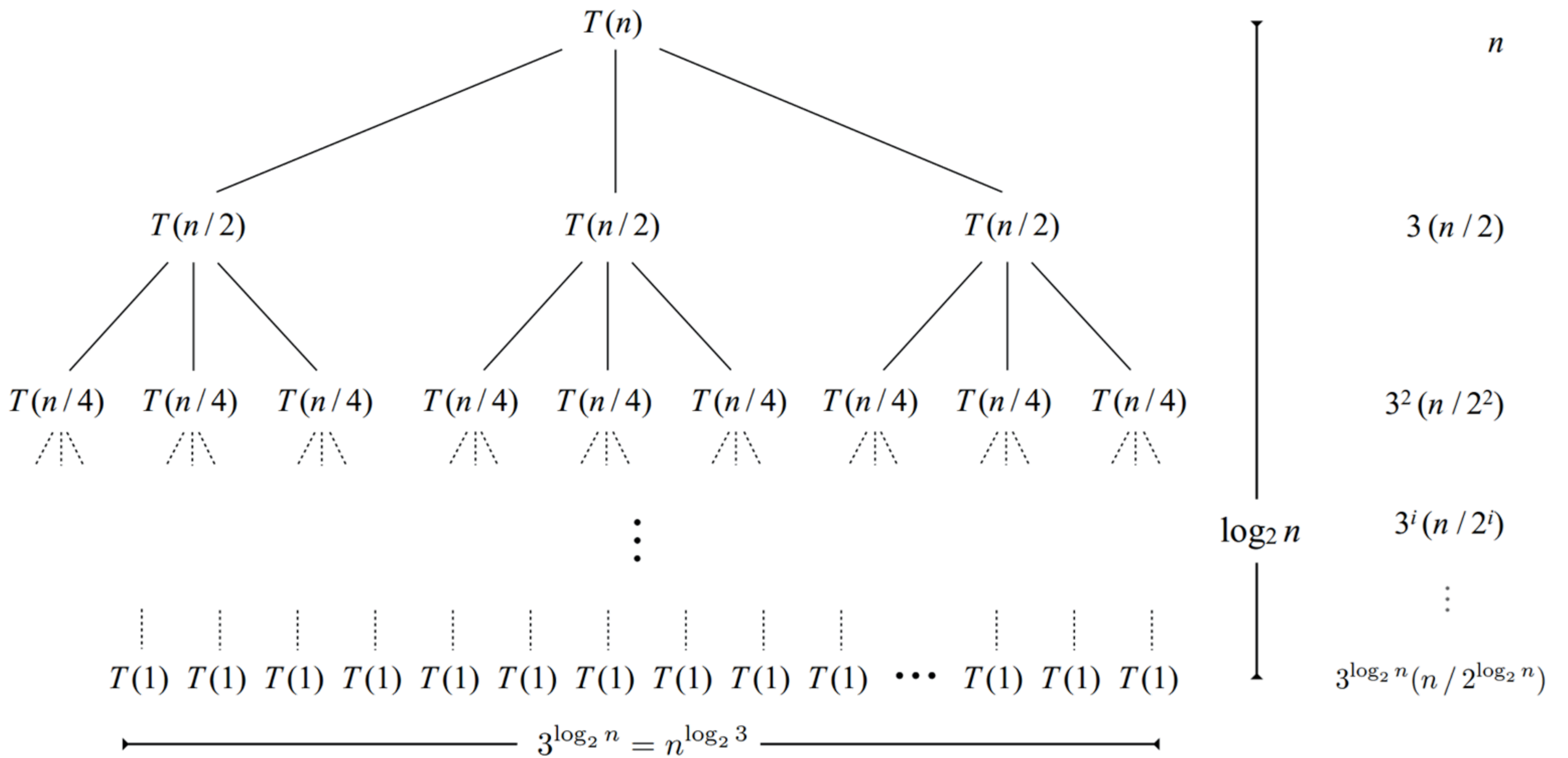
$T(1)=1$

$T(n)=2T(n/2)+n$
$T(1)=1$



$r = 1$ $\qquad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n = n\,(\log_2 n + 1)$

递归树求解：$T(n)=3T(n/2)+n$ ，$T(1)=1$



$r = 3/2 > 1$ $\quad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n = \dfrac{r^{1+\log_2 n} - 1}{r - 1}\, n = 3n^{\log_2 3} - 2n$

再举个例子

- **用递归树求解方程：**

$$T(n)=T(n/3)+T(2n/3)+n$$

再次回到公式

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

方程的解：

$$T(n)=n^{\log_b a}+\sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i)$$

- **第一项为所有最小子问题的计算工作量**
- **第二项为迭代过程归约到子问题及合并解的工作量**

哪一项更主要？

主定理(Master定理)

定理： 设$a \geqslant 1$，$b > 1$为常数，$f(n)$为函数，$T(n)$为非负数，且$T(n)=aT(n/b)+f(n)$，则：

1. 若 $f(n)=O(n^{(\log_b a)-\varepsilon})$，存在$\varepsilon > 0$ 是常数，则有 $T(n)=\Theta(n^{\log_b a})$
2. 若 $f(n)=\Theta(n^{\log_b a})$, 则有$T(n)=\Theta(n^{\log_b a}\log n)$
3. 若 $f(n)=\Omega(n^{(\log_b a)+\varepsilon})$，存在$\varepsilon > 0$ 是常数，且对所有充分大的 n 有 $af(\frac{n}{b}) \leq cf(n)$，$c < 1$ 是常数，则有$T(n)=\Theta(f(n))$

Master定理

- 例子1：求解 $T(n) = 9T\left(\frac{n}{3}\right) + n$

$$a = 9, b = 3, f(n) = n, n^{\log_b a} = n^2$$

$$\because f(n) = n < O\left(n^{(\log_b a)}\right) = n^2$$ 取$\varepsilon = 1$即可

$$\therefore T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^2\right)$$

Master定理

- 例子2：求解$T(n) = T\left(\frac{2n}{3}\right) + 1$

$a = 1, b = \frac{3}{2}, f(n) = 1, n^{\log_b a} = n^{\log_{3/2} 1} = 1$

$\because f(n) = 1 = \Theta(n^{\log_b a}),$

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

Master定理

- **例子3：** 求解$T(n) = 3T\left(\frac{n}{4}\right) + n\log n$

$a = 3, b = 4, f(n) = n\log n$，$n^{\log_b a} = n^{\log_4 3} \approx n^{0.793}$

$f(n) = n\log n = \Omega(n^{\log_4 3+\varepsilon}) \approx \Omega(n^{0.793+\varepsilon})$ 取ε =0.2即可

Master定理

- **例子3：求解**$T(n) = 3T\left(\frac{n}{4}\right) + n\log n$

  条件验证 $af\left(\frac{n}{b}\right) \leq cf(n)$

  $a = 3, b = 4, f(n) = n\log n$，代入上式

  $3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \leq c \times n\log n$　**只要c≥3/4即可**

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

递归算法分析

- **二分检索：** $T(n)=T(n/2)+1, T(1)=1$

$$a=1, b=2, n^{\log_2 1}=1, f(n)=1$$

属于第二种情况

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

不能使用主定理的例子

- **例如：求解** $T(n)=2T(n/2)+n\log n$

$$a=b=2, n^{\log_2 2}=n, f(n)=n\log n$$

不存在$\varepsilon>0$使右式成立 $n\log n=\Omega(n^{1+\varepsilon})$

不存在$c<1$使$af(\frac{n}{b})\le cf(n)$对所有充分大的n成立

$$2(n/2)\log(n/2)=n(\log n-1)\le cn\log n$$

可以考虑递归树！！！

递推方程求解方法小结

- 迭代法
- 换元迭代法
- 递归树
- 公式法
- 主定理

Strassen矩阵乘法

A和B的乘积矩阵C中的元素C[i,j]定义为:

$$C[i][j]=\sum_{k=1}^{n}A[i][k]B[k][j]$$

分析：若依此定义来计算A和B的乘积矩阵C，则每计算C的一个元素C[i][j]，需要做n次乘法和n-1次加法。因此，算出矩阵C的n^2个元素所需的计算时间为O(n^3)

Strassen矩阵乘法

分治法:

使用与大整数相乘类似的技术，将矩阵A，B和C中每一矩阵都分块成4个大小相等的子矩阵。由此可将方程C=AB重写为:

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

由此可得:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

复杂度分析

1）n=2，子矩阵阶为1，8次乘和4次加，直接求出；
2）子矩阵阶大于2，为求子矩阵积可继续分块，直到子矩阵阶降为2。

此想法就产生了一个分治降阶递归算法。

两个n阶方阵的积➔8个n/2阶方阵积和4个n/2阶方阵加。

可在O(n^2)时间内完成

计算时间耗费 $$T(n)=\begin{cases} O(1) & n=2 \\ 8T(n/2)+O(n^2) & n>2 \end{cases}$$

所以 $T(n)=O(n^3)$**，与原始定义计算相比并不有效。**

复杂度分析(cont.)

原因是此法没有减少矩阵的乘法次数！！！

下面从计算2个2阶方阵乘开始，研究减少乘法次数(小于8次)

$$M_1 = A_{11}(B_{12} - B_{22})$$
$$M_2 = (A_{11} + A_{12})B_{22}$$
$$M_3 = (A_{21} + A_{22})B_{11}$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

7次乘 $\Rightarrow$

$$C_{11} = M_5 + M_4 - M_2 + M_6$$
$$C_{12} = M_1 + M_2$$
$$C_{21} = M_3 + M_4$$
$$C_{22} = M_5 + M_1 - M_3 - M_7$$

复杂度分析(cont.)

所以 需要7次乘法 18次加减法 ➔ $T(n)=\begin{cases} O(1) & n=2 \\ 7T(n/2)+O(n^2) & n>2 \end{cases}$

➔ $T(n)=O(n^{\log 7})=O(n^{2.81})$ ✓较大的改进

Strassen矩阵乘法

- 还有没有更快的方法?
- Hopcroft和Kerr已经证明(1971)，计算2个2×2矩阵的乘积，7次乘法是必要的。因此，要想进一步改进矩阵乘法的时间复杂性，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究3×3或5×5矩阵的更好算法。
- 在Strassen之后又有许多算法改进了矩阵乘法的计算时间复杂性。目前最好的计算时间上界是 $O(n^{2.3727})$

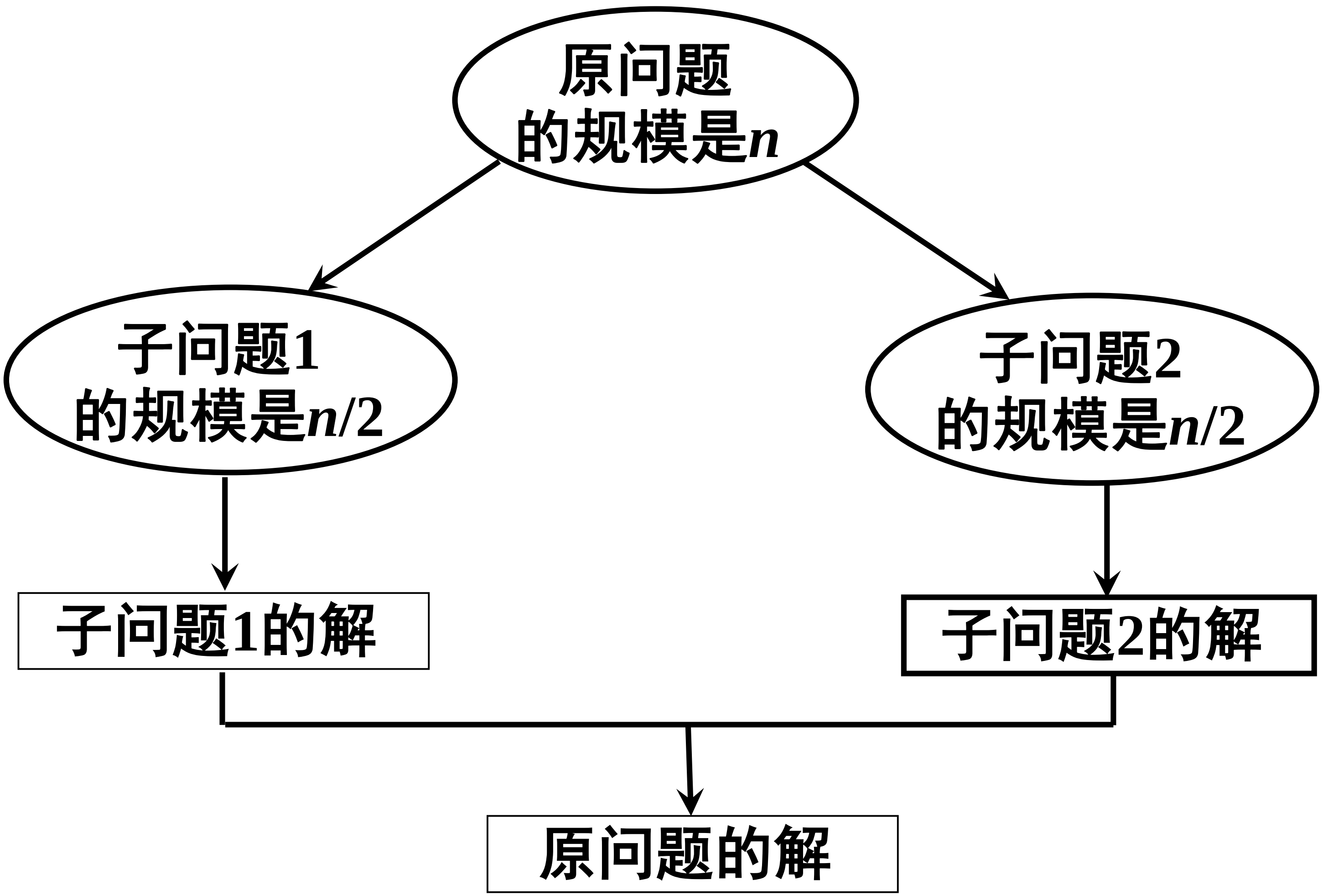
year	algorithm	order of growth
?	brute force	$O(n^3)$
1969	Strassen	$O(n^{2.808})$
1978	Pan	$O(n^{2.796})$
1979	Bini	$O(n^{2.780})$

是否能找到$O(n^2)$的算法?

year	algorithm	order of growth
1982	Romani	$O(n^{2.517})$
1982	Coppersmith-Winograd	$O(n^{2.496})$
1986	Strassen	$O(n^{2.479})$
1989	Coppersmith-Winograd	$O(n^{2.376})$
2010	Strother	$O(n^{2.3737})$
2011	Williams	$O(n^{2.3727})$

排序问题中的分治法

分治法的典型情况



二分归并排序/合并排序

二分归并排序的分治策略是：

- **划分**：将待排序序列r_1，r_2，…，r_n划分为两个长度相等的子序列$r_1, \ldots, r_{n/2}$和$r_{n/2+1}, \ldots, r_n$；
- **求解子问题**：分别对这两个子序列进行排序，得到两个有序子序列；
- **合并**：将这两个有序子序列合并成一个有序序列。

二分归并排序/合并排序

```
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
       }
}
```

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

二分归并排序/合并排序

改进：算法mergeSort的递归过程可以消去。

初始序列　[49] [38] [65] [97] [76] [13] [27]

第一步　[38 49] [65 97] [13 76] [27]

第二步　[38 49 65 97] [13 27 76]

第三步　[13 27 38 49 65 76 97]

二分归并排序/合并排序复杂度

- 最坏时间复杂度： $O(n\log n)$
- 平均时间复杂度：$O(n\log n)$
- 辅助空间：$O(n)$

可以看出，不论是最坏情况还是平均情况，二分归并排序的**复杂度是最好的！**且是个**稳定**排序。

但它的一个重大缺点是，它不是一个就地操作的算法，需要$O(n)$个额外存储单元，当n很大的时候是一个很大的开销。

快速排序

快速排序的分治策略是：

- **划分**：选定一个记录作为轴值，以轴值为基准将整个序列划分为两个子序列$r_1 \ldots r_{i-1}$和$r_{i+1} \ldots r_n$，前一个子序列中记录的值均小于或等于轴值，后一个子序列中记录的值均大于或等于轴值；
- **求解子问题**：分别对划分后的每一个子序列递归处理；
- **合并**：由于对子序列$r_1 \ldots r_{i-1}$和$r_{i+1} \ldots r_n$的排序是就地进行的，所以合并不需要执行任何操作。

快速排序

$$[\, r_1 \;\; \dots \;\; \dots \;\; r_{i\text{-}1} \,] \; r_i \; [\, r_{i+1} \;\; \dots \;\; \dots \;\; r_n \,]$$

均$\leqslant r_i$　　　轴值　　　均$\geqslant r_i$

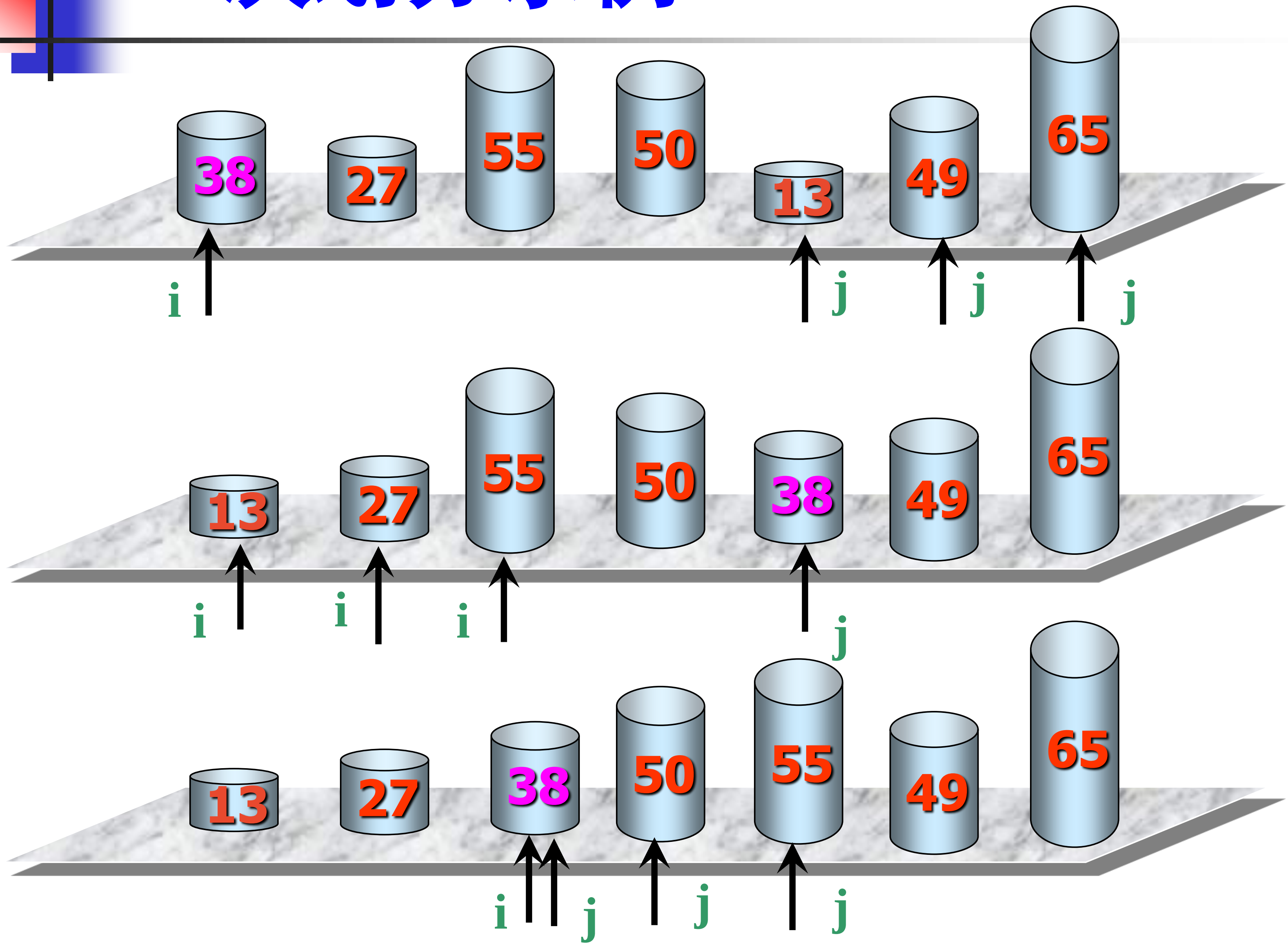
位于最终位置

- 归并排序按照元素在序列中的位置对序列进行划分，
- 快速排序按照元素的值对序列进行划分。

以第一个元素/记录作为轴值，对待排序序列进行划分的过程为：

1. 初始化：取第一个记录作为基准，设置两个参数i，j分别用来指示将要与基准记录进行比较的左侧记录位置和右侧记录位置，也就是本次划分的区间；
2. 右侧扫描过程：将基准记录与j指向的记录进行比较，如果j指向记录的关键码大，则j前移一个记录位置。重复右侧扫描过程，直到右侧的记录小（即反序）。若i<j，则将基准记录与j指向的记录进行交换；
3. 左侧扫描过程：将基准记录与i指向的记录进行比较，如果i指向记录的关键码小，则i后移一个记录位置。重复左侧扫描过程，直到左侧的记录大（即反序）。若i<j，则将基准记录与i指向的记录交换；
4. 重复2、3步，直到i与j指向同一位置，即基准记录最终的位置。

一次划分示例

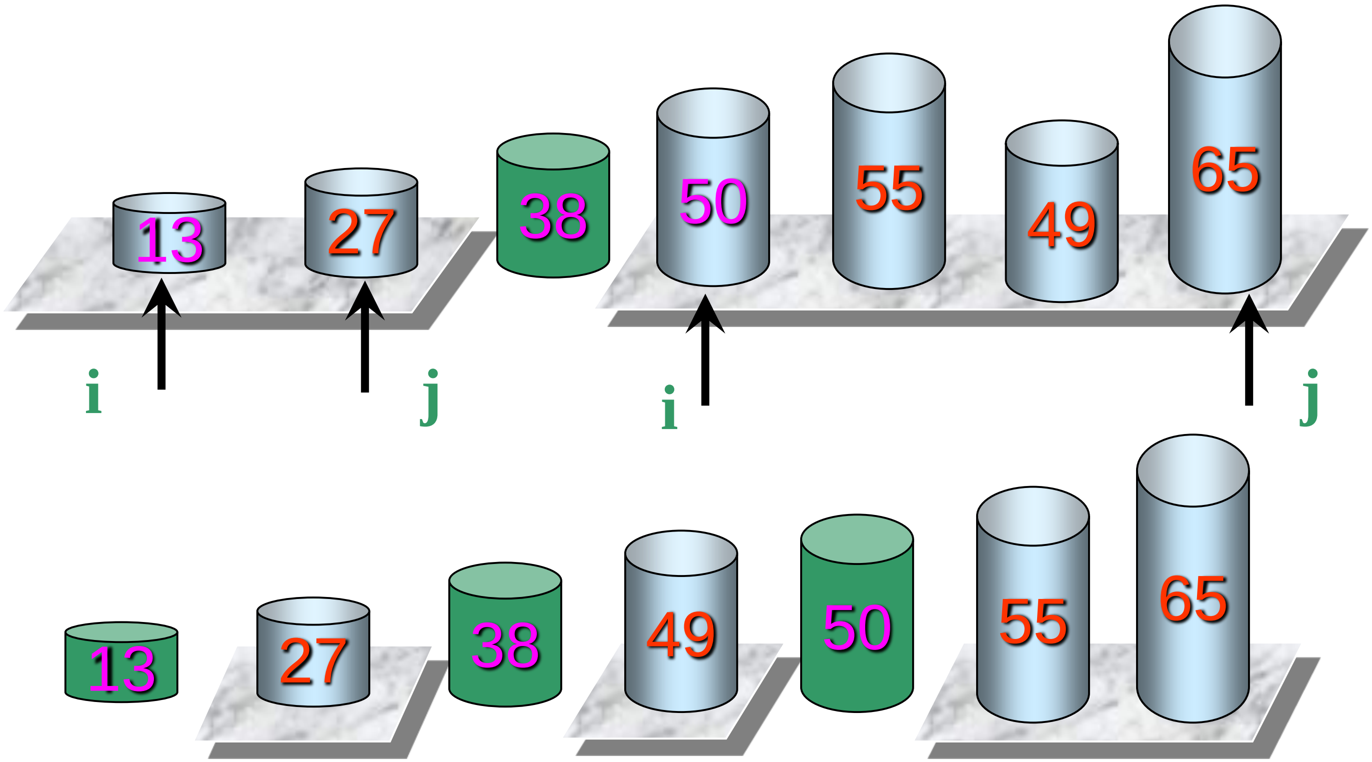

38 27 55 50 13 49 65
i j j j
13 27 55 50 38 49 65
i i i j
13 27 38 50 55 49 65
i j j j

C++描述

一次划分算法伪代码

```
int Partition(int r[ ], int first, int end)
{
    i=first; j=end;          //初始化
    while (i<j){
        while (i<j && r[i]<= r[j]) j--;    //右侧扫描
        if (i<j) {
          r[i]←→r[j];          //将较小记录交换到前面
          i++;
        }
       while (i<j && r[i]<= r[j]) i++;    //左侧扫描
       if (i<j) {
         r[j]←→r[i];          //将较大记录交换到后面
         j--;
       }
    }
    return i;    // i为轴值记录的最终位置
}
```

以轴值为基准将待排序序列划分为两个子序列后，对每一个子序列分别递归进行排序。



C++描述

快速排序算法伪代码

```
void QuickSort(int r[ ], int first, int end){
 if (first<end) {
    pivot=Partition(r, first, end);
      //问题分解，pivot是轴值在序列中的位置
    QuickSort(r, first, pivot-1);
      //递归地对左侧子序列进行快速排序
    QuickSort(r, pivot+1, end);
     //递归地对右侧子序列进行快速排序
  }
}
```

在最好情况下，每次划分对一个记录定位后，该记录的左侧子序列与右侧子序列的长度相同。在具有n个记录的序列中，一次划分需要对整个待划分序列扫描一遍，则所需时间为$O(n)$。设$T(n)$是对n个记录的序列进行排序的时间，每次划分后，正好把待划分区间划分为长度相等的两个子序列，则有：

$$T(n) \leqslant 2\,T(n/2) + n$$

$$\leqslant 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n$$

$$\leqslant 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n$$

$$\cdots\ \cdots\ \cdots$$

$$\leqslant nT(1) + n\log_2 n = O(n\log n)$$

因此，时间复杂度为$O(n\log n)$。

在最坏情况下，待排序记录序列正序或逆序，每次划分只得到一个比上一次划分少一个记录的子序列（另一个子序列为空）。此时，必须经过n-1次递归调用才能把所有记录定位，而且第i趟划分需要经过n-i次关键码的比较才能找到第i个记录的基准位置，因此，总的比较次数为：

$$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}n(n-1)=O(n^2)$$

因此，时间复杂度为$O(n^2)$。

在平均情况下，设基准记录的关键码第***k***小(**1**≤***k*** ≤***n***)，则有：

$$T(n)=\frac{1}{n}\sum_{k=1}^{n-1}(T(k)+T(n-k))+n-1=\frac{2}{n}\sum_{k=1}^{n-1}T(k)+n-1$$

这是快速排序的平均时间性能，其数量级也为***O*****(*****n*****log*****n*****)**。

这是一个关于全部历史的递推方程，求解需要技巧！ ---差消法

快速排序复杂度分析小结

- 快速排序算法的性能取决于划分的对称性。
- 快速排序时间与划分是否对称有关，最坏情况一边n-1个，一边1个。
  - 如不对称，则：$T(n)=\begin{cases} O(1) & n\le 1 \\ T(n-1)+O(n) & n>1 \end{cases}$

    解得 $T(n)=O(n^2)$
  - 最好情况，对称，则：$T(n)=\begin{cases} O(1) & n\le 1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$

    解得 $T(n)=O(n\log n)$

结论：
最坏时间复杂度：$O(n^2)$
平均时间复杂度：$O(n\log n)$

递推方程求解方法小结

- 迭代法
- 换元迭代法
- 递归树
- 公式法
- 主定理

课程QQ群

LIVES

711174算法作业交流群

扫一扫二维码，加入群聊。

- 布置作业
- 上传课件
- 答疑解惑
- 课后交流

助教：杨明璇

220181709@seu.edu.cn

几何问题中的分治法

最接近点对问题

设$p_1=(x_1, y_1), p_2=(x_2, y_2), \ldots, p_n=(x_n, y_n)$是平面上$n$个点构成的集合$S$，最近对问题就是找出集合$S$中距离最近的点对。

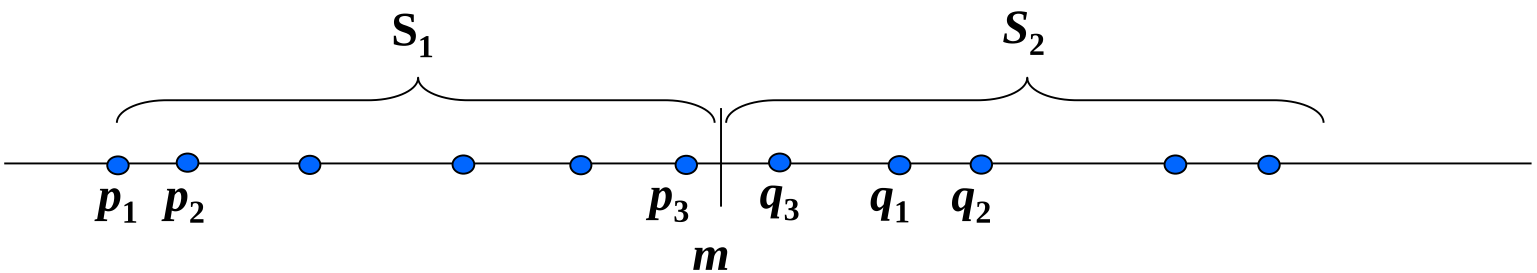
严格地讲，最接近点对可能多于一对，简单起见，只找出其中的一对作为问题的解。

最接近点对问题

用分治法解决最近对问题，很自然的想法就是将集合S分成两个子集 $S1$和 $S2$，每个子集中有$n/2$个点。然后在每个子集中递归地求其最接近的点对，在求出每个子集的最接近点对后，在合并步中，如果集合 S 中最接近的两个点都在子集 $S1$ 或 $S2$中，则问题很容易解决，如果这两个点分别在 $S1$和 $S2$中，问题就比较复杂了。

为了使问题易于理解，先考虑一维的情形。

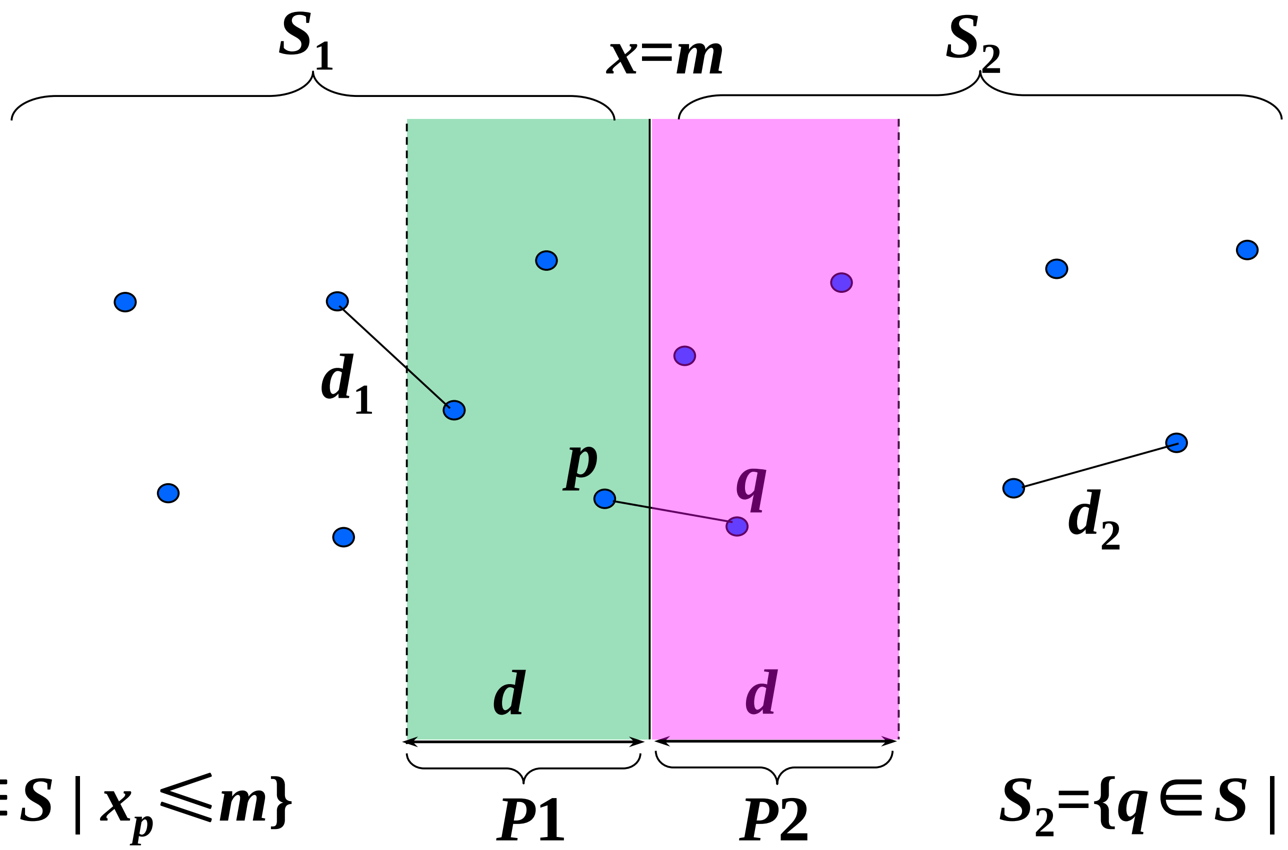
此时，S中的点退化为x轴上的n个点$x_1, x_2, \ldots, x_n$。用x轴上的某个点m将S划分为两个集合S_1和S_2，并且S_1和S_2含有点的个数相同。递归地在S_1和S_2上求出最接近点对 (p_1, p_2) 和 (q_1, q_2)，如果集合S中的最接近点对都在子集S_1或S_2中，则$d=\min\{(p_1, p_2), (q_1, q_2)\}$即为所求，如果集合$S$中的最接近点对分别在$S_1$和$S_2$中，则一定是$(p_3, q_3)$，其中，$p_3$是子集$S_1$中的最大值，$q_3$是子集$S_2$中的最小值。



下面考虑**二维**的情形，此时S中的点为平面上的点。

- 为了将平面上的点集S 分割为点的个数大致相同的两个子集S_1和S_2，选取垂直线$x=m$来作为分割线，其中，m为S中各点x坐标的中位数。
- 由此将S分割为$S_1=\{p\in S \mid x_p\leqslant m\}$和$S_2=\{q\in S \mid x_q>m\}$。
- 递归地在S_1和S_2上求解最近对问题，分别得到S_1中的最近距离d_1和S_2中的最近距离d_2。
- 令$d=\min(d_1, d_2)$，若S的最近对(p, q)之间的距离小于d，则p和q必分属于S_1和S_2，不妨设$p\in S_1$，$q\in S_2$，则p和q距直线$x=m$的距离均小于d，所以，可以将求解限制在以$x=m$为中心、宽度为$2d$的垂直带$P1$和$P2$中，垂直带之外的任何点对之间的距离都一定大于d。

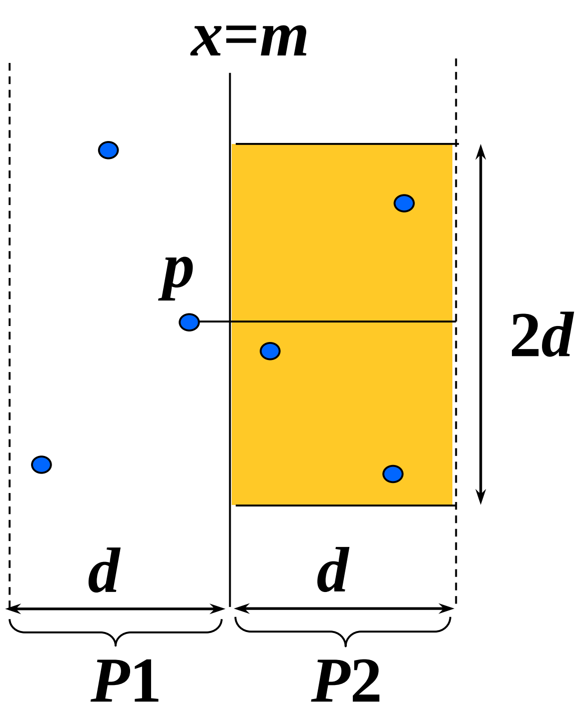
最近对问题的分治思想



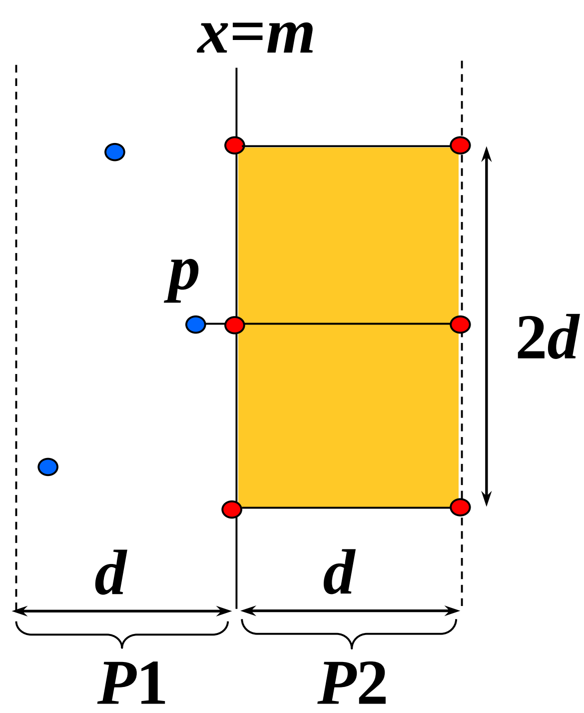
$S_1=\{p\in S \mid x_p\leqslant m\}$

$S_2=\{q\in S \mid x_q > m\}$

对于点$p \in P1$，需要考察$P2$中的各个点和点p之间的距离是否小于d，显然，$P2$中这样点的y轴坐标一定位于区间$[y-d, y+d]$之间。而且，这样的点不会超过?个。



(a) 包含点q的$d\times 2d$的矩形区域



(b) 最坏情况下需要检查的6个点

最接近点对问题复杂度分析

应用分治法求解含有n个点的最近对问题，其时间复杂性可由下面的递推式表示：

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

合并子问题的解的时间$f(n)=O(n)$，根据主定理，可得$T(n)=O(n\log n)$。

课后作业

- 上传群里
  - LaTeX版本：homework-01.tex
- 提交要求：
  - 一份答案电子版（分析题+实现题算法描述）
    - 命名：学号-homework-01.tex
  - 一份实现题的可执行源代码（C++）
  - 打包成一个文件：学号-姓名-次数.zip
  - 电子邮件地址：mxyang@seu.edu.cn
- 截止时间：
  - 一周内完成（如：周二课堂布置，下周一前提交；周四课堂布置，下周三前提交）

LaTex教程

- TEX是斯坦福大学的教授Donald E. Knuth开发的一个功能强大的幕后排版系统。当时在撰写名为《The Art of Computer Programming》的书，由于出版商把书中的数学式子排版得很难看，决定推迟出版，自行研发一套排版系统进行排版。这个系统就是TEX系统。
- LaTeX：
  - TEX是很低阶的排版语言，对于绝大多数人来说，学起来会很吃力，而且排版工作也会变得相当繁复，难以被更多人使用，效率也不是很高。所以，一些经常用到的功能，如果我们事先定义好，到要用的时候只引用一小段代码就可以实现一个相对复杂的功能，那不仅提高了排版效率，而且版面也会清晰很多。这种事先定义好的功能，叫做宏集(macro)。
  - LaTeX就是TEX的众多宏集之一，是由Leslie Lamport编写的。

LaTex教程

- **CTeX：**
  - **CTeX是利用TEX排版系统中文套装的简称**
  - **CTeX中文套装在MiKTeX的基础上增加了对中文的完整支持，集成了编辑器WinEdt和PostScript处理软件Ghostscript 和 GSview 等主要工具**
  - **支持CCT和CJK两种中文TeX处理方式**
  - **http://www.ctex.org**