

中国计算机学会

“21世纪大学本科计算机专业系列教材”

算法设计与分析

王晓东 编著

主要内容介绍

- 第1章 算法引论
- 第2章 递归与分治策略
- 第3章 动态规划
- 第4章 贪心算法
- 第5章 回溯法
- 第6章 分支限界法

主要内容介绍（续）

- 第7章 概率算法
- 第8章 NP完全性理论
- 第9章 近似算法
- 第10章 算法优化策略

第1章 算法引论

本章主要知识点:

- 1.1 算法与程序
- 1.2 表达算法的抽象机制
- 1.3 描述算法
- 1.4 算法复杂性分析

1.1 算法与程序

算法：是满足下述性质的指令序列。

- 输入：有零个或多个外部量作为算法的输入。
- 输出：算法产生至少一个量作为输出。
- 确定性：组成算法的每条指令清晰、无歧义。
- 有限性：算法中每条指令的执行次数有限，执行每条指令的时间也有限。

程序：是算法用某种程序设计语言的具体实现。
程序可以不满足算法的性质(4)即有限性。

1.2 表达算法的抽象机制

1.从机器语言到高级语言的抽象

高级程序设计语言的主要好处是：

(1) 高级语言更接近算法语言，易学、易掌握，一般工程技术人员只需要几周时间的培训就可以胜任程序员的工作；

(2) 高级语言为程序员提供了结构化程序设计的环境和工具，使得设计出来的程序可读性好，可维护性强，可靠性高；

(3) 高级语言不依赖于机器语言，与具体的计算机硬件关系不大，因而所写出来的程序可移植性好、重用率高；

(4) 把繁杂琐碎的事务交给编译程序，所以自动化程度高，开发周期短，程序员可以集中时间和精力从事更重要的创造性劳动，提高程序质量。

1.2 表达算法的抽象机制

2.抽象数据类型

抽象数据类型是算法的一个数据模型连同定义在该模型上并作为算法构件的一组运算。

抽象数据类型带给算法设计的好处有：

- (1) 算法顶层设计与底层实现分离；
- (2) 算法设计与数据结构设计隔开，允许数据结构自由选择；
- (3) 数据模型和该模型上的运算统一在ADT中，便于空间和时间耗费的折衷；
- (4) 用抽象数据类型表述的算法具有很好的可维护性；
- (5) 算法自然呈现模块化；
- (6) 为自顶向下逐步求精和模块化提供有效途径和工具；
- (7) 算法结构清晰，层次分明，便于算法正确性的证明和复杂性的分析。

1.3 描述算法

在本书中，采用**Java语言**描述算法。

以下，对**Java语言**的若干重要特性作简要概述。

1. Java程序结构

(1) Java程序的两种类型：应用程序和applet

区别：应用程序的主方法为**main**，其可在命令行中用命令语句 **java 应用程序名** 来执行；

applet的主方法为**init**，其必须嵌入HTML文件，由Web浏览器或applet阅读器来执行。

(2) 包：java程序和类可以包(packages)的形式组织管理。

(3) import语句：在java程序中可用import语句加载所需的包。

例如，**import java.io.*;**语句加载java.io包。

1.3 描述算法

2. Java数据类型

数据类型 { 基本数据类型：详见下页表1-1
非基本数据类型：如 Byte, Integer, Boolean, String等。

Java对两种数据类型的不同处理方式：

- 对基本数据类型：在声明一个具有基本数据类型的变量时，自动建立该数据类型的对象（或称实例）。如: `int k;`
- 对非基本数据类型：语句 `String s;` 并不建立具有数据类型 `String` 的对象，而是建立一个类型 `String` 的引用对象，数据类型为 `String` 的对象可用下面的 `new` 语句建立。

```
s = new String("Welcome");  
String s = new String("Welcome");
```

1.3 描述算法

表格1-1 Java基本数据类型

类型	缺省值	分配空间 (bits)	取值范围
boolean	false	1	[true,false]
byte	0	8	[-128,127]
char	\u0000	16	[\u0000,\uFFFF]
double	0.0	64	$\pm 4.9*10^{-324} \sim \pm 1.8*10^{308}$
float	0.0	32	$\pm 1.4*10^{-45} \sim \pm 3.4*10^{38}$
int	0	32	[-2147483648,2147483647]
long	0	64	$\pm 9.2*10^{17}$
short	0	16	[-32768,32767]

1.3 描述算法

3. 方法

在Java中，执行特定任务的函数或过程统称为方法(methods)。例如，java的**Math类**给出的常见数学计算的方法如下表所示：

方法	功能	方法	功能
abs(x)	x的绝对值	max(x,y)	x和y中较大者
ceil(x)	不小于x的最小整数	min(x,y)	x和y中较小者
cos(x)	x的余弦	pow(x,y)	x^y
exp(x)	e^x	sin(x)	x的正弦
floor(x)	不大于x的最大整数	sqrt(x)	x的平方根
log(x)	x的自然对数	tan(x)	x的正切

1.3 描述算法

3. 方法

计算表达式 $\frac{a+b+|a-b|}{2}$ 值的自定义方法ab描述如下：

```
public static int ab(int a, int b)
{
    return (a+b+Math.abs(a-b))/2;
}
```

(1) **方法参数**：Java中所有方法的参数均为值参数。上述方法ab中，a和b是形式参数，在调用方法时通过实际参数赋值。

(2) **方法重载**：Java允许方法重载，即允许定义有不同签名的同名方法。上述方法ab可重载为：

```
public static double ab(double a, double b)
{
    return (a+b+Math.abs(a-b))/2.0;
}
```

1.3 描述算法

4. 异常

Java的异常提供了一种处理错误的方法。当程序发现一个错误，就引发一个异常，以便在合适地方捕获异常并进行处理。

通常用try块来定义异常处理。每个异常处理由一个catch语句组成。

```
public static void main(String [] args)
{
    try { f (); }
    catch (exception1)
    { 异常处理; }
    catch (exception2)
    { 异常处理; }
    ...
    finally
    { finally块; }
}
```

1.3 描述算法

5. Java的类

Java的类一般由4个部分组成：

(1) 类名

(2) 数据成员

(3) 方法

(4) 访问修饰 { 公有 (public)
私有 (private)
保护 (protected)

1.3 描述算法

6. 通用方法

下面的方法swap用于交换一维整型数组a的位置i和位置j处的值。

```
public static void swap(int [] a, int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

该方法只适用于
整型数组

以上方法修改如下：

```
public static void swap(object [] a, int i, int j)
{
    object temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

该方法具有通用性，适
用于Object类型及其所
有子类

1.3 描述算法

6. 通用方法

(1) Computable 界面

```
public static Computable sum(Computable [] a, int n)
{
    if (a.length == 0) return null;
    Computable sum = (Computable) a[0].zero();
    for (int i = 0; i < n; i++)
        sum.increment(a[i]);
    return sum;
}
```

利用此界面使
方法sum通用化

1.3 描述算法

6. 通用方法

(2) java.lang.Comparable 界面

Java的Comparable 界面中惟一的方法头compareTo用于比较2个元素的大小。例如java.lang.Comparable.x.compareTo(y)返回x-y的符号，当 $x < y$ 时返回负数，当 $x = y$ 时返回0，当 $x > y$ 时返回正数。

(3) Operable 界面

有些通用方法同时需要Comparable界面和Comparable 界面的支持。为此可定义Operable界面如下：

```
public interface Operable extends Comparable, Comparable  
{}
```

(4) 自定义包装类

由于Java的包装类如Integer等已定义为final型，因此无法定义其子类，作进一步扩充。为了需要可自定义包装类。 17

1.3 描述算法

7. 垃圾收集

Java的new运算用于分配所需的内存空间。

例如，`int [] a = new int [500000];` 分配2000000字节空间给整型数组a。频繁用new分配空间可能会耗尽内存。Java的垃圾收集器会适时扫描内存，回收不用的空间（垃圾）给new重新分配。

8. 递归

Java允许方法调用其自身。这类方法称为递归方法。

```
public static int sum(int [] a, int n)
{
    if (n==0) return 0;
    else return a[n-1]+sum(a,n-1);
}
```

计算一维整型数组前n个元素之和的递归方法

1.4 算法复杂性分析

算法复杂性是算法运行所需要的计算机资源的量，需要时间资源的量称为**时间复杂性**，需要的空间资源的量称为**空间复杂性**。这个量应该只依赖于算法要解的问题的规模、算法的输入和算法本身的函数。如果分别用 N 、 I 和 A 表示算法要解问题的规模、算法的输入和算法本身，而且用 C 表示复杂性，那么，应该有 $C=F(N, I, A)$ 。一般把时间复杂性和空间复杂性分开，并分别用 T 和 S 来表示，则有： $T=T(N, I)$ 和 $S=S(N, I)$ 。

（通常，让 A 隐含在复杂性函数名当中）

1.4 算法复杂性分析

最坏情况下的时间复杂性:

$$T_{\max}(N) = \max_{I \in D_N} T(N, I) = \max_{I \in D_N} \sum_{i=1}^k t_i e_i(N, I) = \sum_{i=1}^k t_i e_i(N, I^*) = T(N, I^*)$$

最好情况下的时间复杂性:

$$T_{\min}(N) = \min_{I \in D_N} T(N, I) = \min_{I \in D_N} \sum_{i=1}^k t_i e_i(N, I) = \sum_{i=1}^k t_i e_i(N, \tilde{I}) = T(N, \tilde{I})$$

平均情况下的时间复杂性:

$$T_{\text{avg}}(N) = \sum_{I \in D_N} P(I) T(N, I) = \sum_{I \in D_N} P(I) \sum_{i=1}^k t_i e_i(N, I)$$

其中 D_N 是规模为 N 的合法输入的集合; I^* 是 D_N 中使 $T(N, I^*)$ 达到 $T_{\max}(N)$ 的合法输入; \tilde{I} 是中使 $T(N, \tilde{I})$ 达到 $T_{\min}(N)$ 的合法输入; 而 $P(I)$ 是在算法的应用中出现输入 I 的概率。

1.4 算法复杂性分析

算法复杂性在渐近意义下的阶：

渐近意义下的记号： O 、 Ω 、 θ 、 o

设 $f(N)$ 和 $g(N)$ 是定义在正数集上的正函数。

O 的定义： 如果存在正的常数 C 和自然数 N_0 ，使得当 $N \geq N_0$ 时有 $f(N) \leq Cg(N)$ ，则称函数 $f(N)$ 当 N 充分大时上有界，且 $g(N)$ 是它的一个上界，记为 $f(N) = O(g(N))$ 。即 $f(N)$ 的阶不高于 $g(N)$ 的阶。

根据 O 的定义，容易证明它有如下运算规则：

- (1) $O(f) + O(g) = O(\max(f, g))$;
- (2) $O(f) + O(g) = O(f+g)$;
- (3) $O(f)O(g) = O(fg)$;
- (4) 如果 $g(N) = O(f(N))$ ，则 $O(f) + O(g) = O(f)$;
- (5) $O(Cf(N)) = O(f(N))$ ，其中 C 是一个正的常数;
- (6) $f = O(f)$ 。

1.4 算法复杂性分析

Ω 的定义: 如果存在正的常数 C 和自然数 N_0 , 使得当 $N \geq N_0$ 时有 $f(N) \geq Cg(N)$, 则称函数 $f(N)$ 当 N 充分大时下有界, 且 $g(N)$ 是它的一个下界, 记为 $f(N) = \Omega(g(N))$ 。即 $f(N)$ 的阶不低于 $g(N)$ 的阶。

θ 的定义: 定义 $f(N) = \theta(g(N))$ 当且仅当 $f(N) = O(g(N))$ 且 $f(N) = \Omega(g(N))$ 。此时称 $f(N)$ 与 $g(N)$ 同阶。

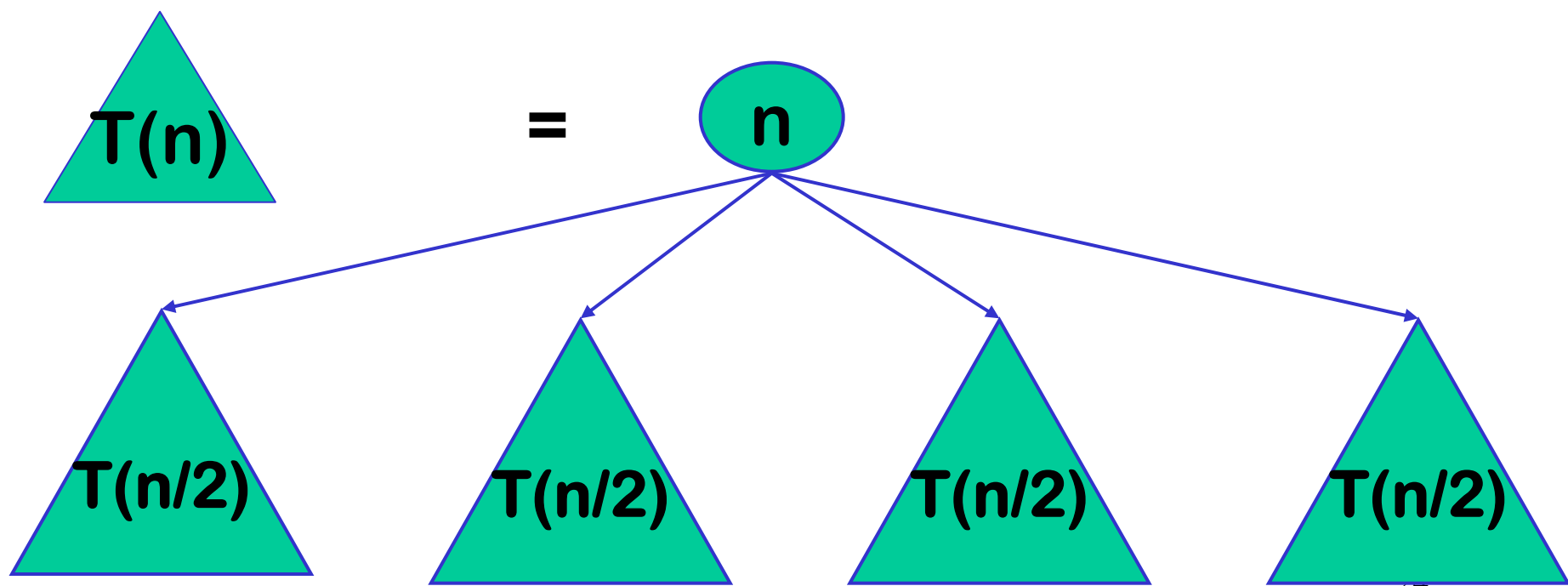
o 的定义: 对于任意给定的 $\varepsilon > 0$, 都存在正整数 N_0 , 使得当 $N \geq N_0$ 时有 $f(N) / Cg(N) \leq \varepsilon$, 则称函数 $f(N)$ 当 N 充分大时的阶比 $g(N)$ 低, 记为 $f(N) = o(g(N))$ 。

例如, $4N \log N + 7 = o(3N^2 + 4N \log N + 7)$ 。

第2章 递归与分治策略

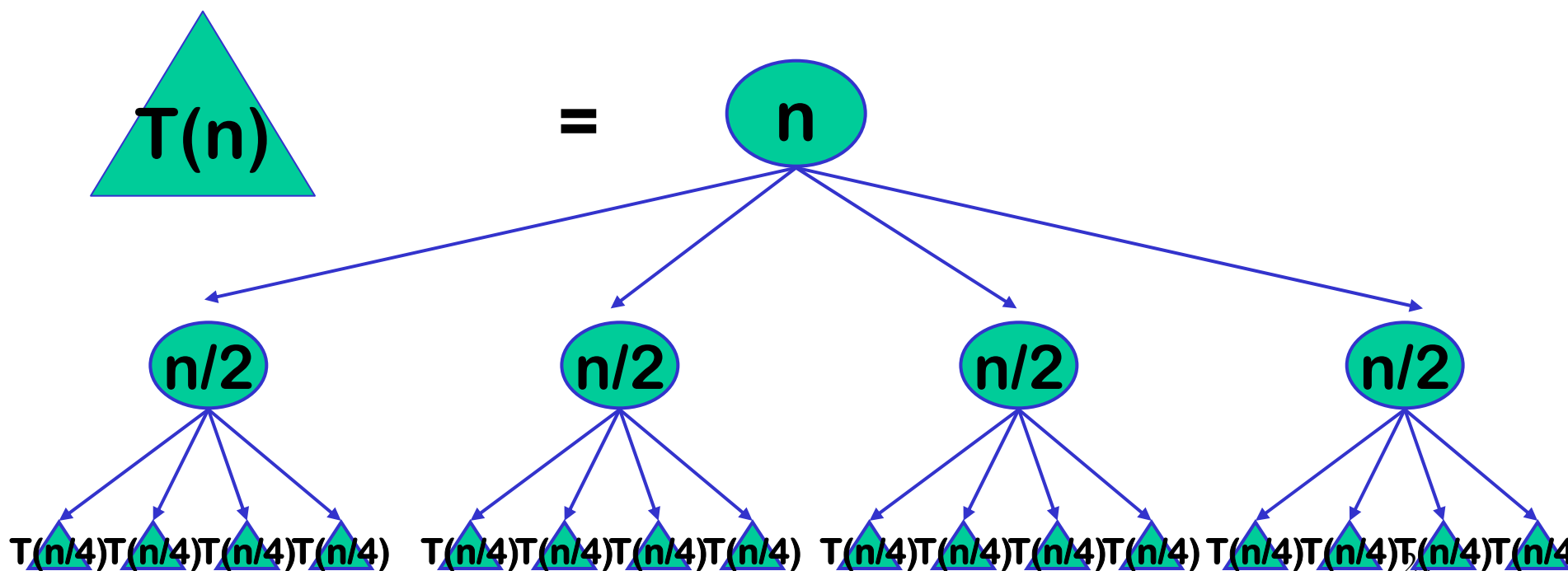
算法总体思想

- 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。



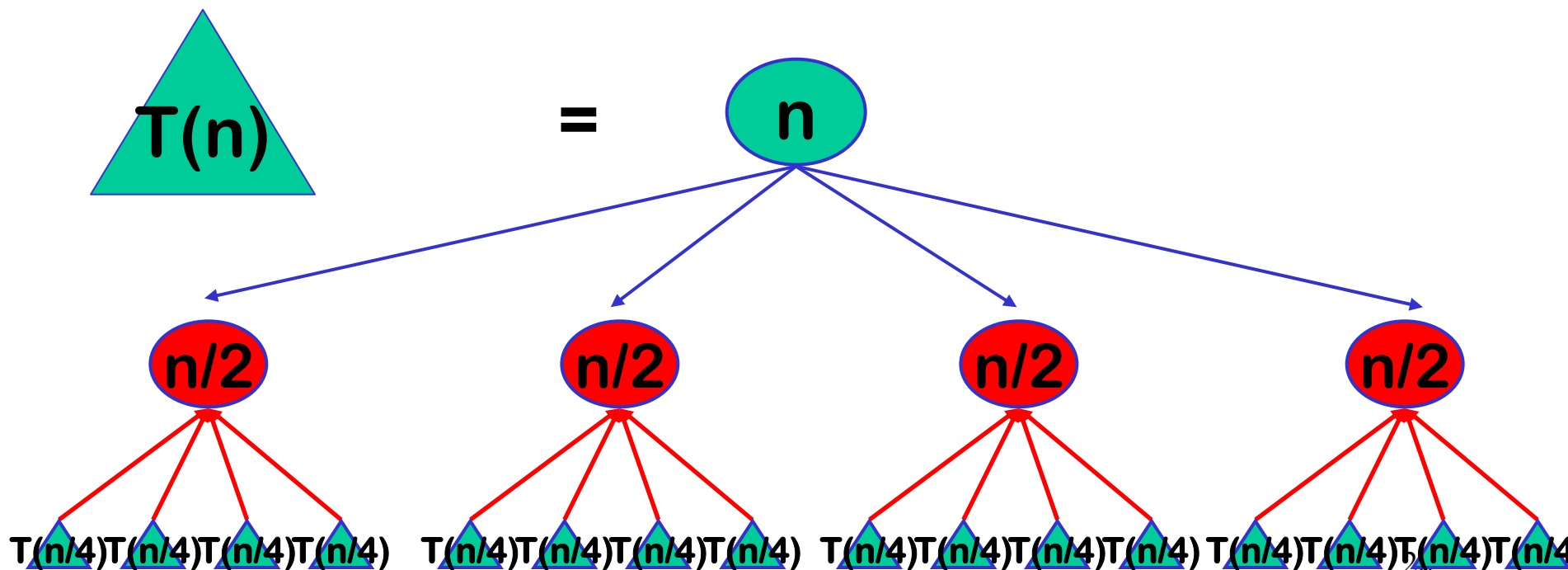
算法总体思想

- 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

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。

下面来看几个实例。

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, ..., 被称为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:

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

$$\left\{ \begin{array}{ll} A(1,0) = 2 \\ A(0,m) = 1 & m \geq 0 \\ A(n,0) = n + 2 & n \geq 2 \\ A(n,m) = A(A(n-1,m), m-1) & n, m \geq 1 \end{array} \right.$$

2.1 递归的概念

例3 Ackerman函数

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**。

Ackerman函数 $A(n, m)$ 定义如下：

$$\left\{ \begin{array}{ll} A(1,0) = 2 \\ A(0,m) = 1 & m \geq 0 \\ A(n,0) = n + 2 & n \geq 2 \\ A(n,m) = A(A(n-1,m), m-1) & n, m \geq 1 \end{array} \right.$$

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：

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

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

但本例中的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 $A(n, 1) = 2 * n$
- $M=2$ 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 $A(n, 2) = 2^n$ 。

$$2^{2^{2^{N^2}}}$$

- $M=3$ 时, 类似的可以推出
- $M=4$ 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3 Ackerman函数

- 定义单变量的Ackerman函数 $A(n)$ 为,
 $A(n) = A(n, n)$ 。
- 定义其拟逆函数 $\alpha(n)$ 为: $\alpha(n) = \min\{k \mid A(k) \geq n\}$ 。即 $\alpha(n)$ 是使 $n \leq A(k)$ 成立的最小的 k 值。
- $\alpha(n)$ 在复杂度分析中常遇到。对于通常所见到的正整数 n , 有 $\alpha(n) \leq 4$ 。但在理论上 $\alpha(n)$ 没有上界, 随着 n 的增加, 它以难以想象的慢速度趋向正无穷大。

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 $\{r_1, r_2, \dots, r_n\}$ 的全排列。

设 $R = \{r_1, r_2, \dots, r_n\}$ 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 $\text{perm}(X)$ 。

$(r_i) \text{perm}(X)$ 表示在全排列 $\text{perm}(X)$ 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， $\text{perm}(R) = (r)$ ，其中 r 是集合 R 中唯一的元素；
当 $n>1$ 时， $\text{perm}(R)$ 由 $(r_1) \text{perm}(R_1)$ ， $(r_2) \text{perm}(R_2)$ ， \dots ， $(r_n) \text{perm}(R_n)$ 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： $n=n_1+n_2+\dots+n_k$,

其中 $n_1 \geq n_2 \geq \dots \geq n_k \geq 1$, $k \geq 1$ 。

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 $q(n, m)$ 的如下递归关系。

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

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。

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

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。

2.1 递归的概念

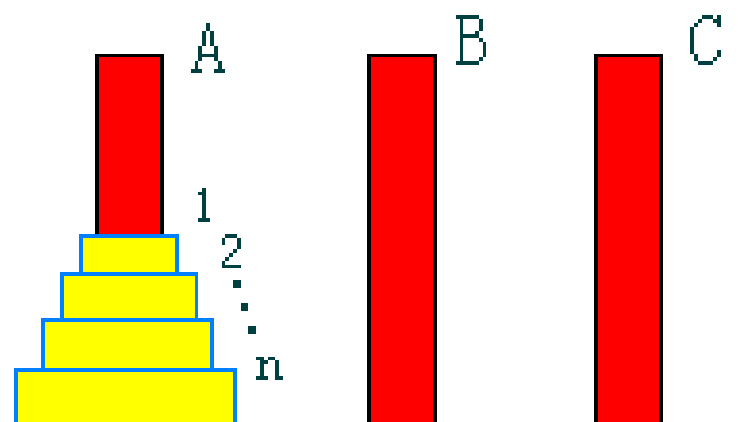
例5 整数划分问题

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

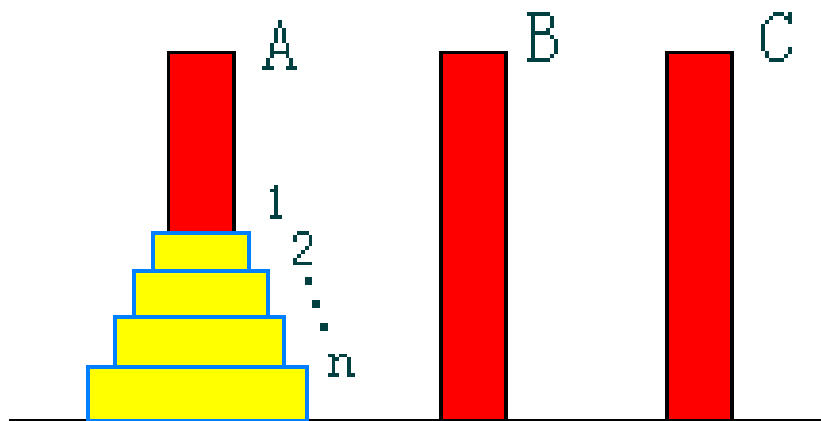
例6 Hanoi塔问题

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 $1, 2, \dots, n$ ，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。



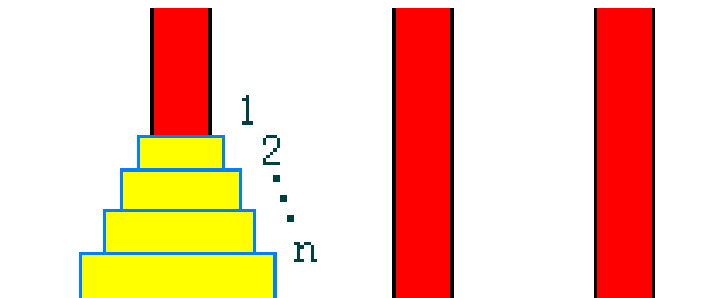
2.1 递归的概念

例6 Hanoi塔问题

```
public static 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);  
    }  
}
```

思考题：如果塔的个数变为a,b,c,d四个，现要将n个圆盘从a全部移动到d，移动规则不变，求移动步数最小的方案。



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。

递归小结

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。

1. 采用一个用户定义的栈来模拟系统的递归调用工作栈。该方法通用性强，但本质上还是递归，只不过人工做了本来由编译器做的事情，优化效果不明显。
2. 用递推来实现递归函数。
3. 通过Cooper变换、反演变换能将一些递归转化为尾递归，从而迭代求出结果。

后两种方法在时空复杂度上均有较大改善，但其适用范围有限。

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。

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。设分解阈值 $n_0=1$ ，且ad hoc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 $m^i \leq n < m^{i+1}$ 时， $T(m^i) \leq T(n) < T(m^{i+1})$ 。

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。

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：

```
public static int binarySearch(int [] a, int x, int n)
{
    // 在  $a[0] \leq a[1] \leq \dots \leq a[n-1]$  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;
        if (x > a[middle]) left = middle + 1;
        else right = middle - 1;
    }
    return -1; // 未找到 $x$ 
}
```

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 $O(\log n)$ 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 $O(\log n)$ 。

思考题：给定 a ，用二分法设计出求 a^n 的算法。

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：

$$\begin{array}{l} X \\ Y \end{array} = \begin{array}{l} \text{复杂度分析} \\ T(n) = \end{array} \begin{cases} O(1) & n = 1 \\ 4T(n/2) + O(n) & n > 1 \end{cases}$$

$$T(n) = O(n^2) \quad \text{✗没有改进} \textcircled{\ominus}$$

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

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

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

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

$T(n) = O(n^{\log_3 3}) = O(n^{1.59})$ ✓较大的改进☺

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

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

细节问题：两个 XY 的复杂度都是 $O(n^{\log_3 3})$ ，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。

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，对于大整数乘法，它能在 $O(n \log n)$ 时间内解决。

➤是否能找到线性时间的算法??? 目前为止还没有结果₅₂

Strassen矩阵乘法

◆传统方法: $O(n^3)$

A和B的乘积矩阵C中的元素 $C[i,j]$ 定义为:
$$C[i][j] = \sum_{k=1}^n A[i][k]B[k][j]$$

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

◆传统方法: $O(n^3)$

◆分治法:

使用与
个大小

复杂度分析

$$T(n) = \begin{cases} O(1) & n = 2 \\ 7T(n/2) + O(n^2) & n > 2 \end{cases}$$

$T(n) = O(n^3)$ ✖ 没有改进 ☹

由此可得:

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

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

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

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

成4

Strassen矩阵乘法

◆传统方法: $O(n^3)$

◆分治法:

为了降

复杂度分析

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

$$T(n) = O(n^{\log_2 8}) = O(n^{2.81}) \quad \checkmark \text{ 较大的改进} \text{😊}$$

$$M_1 = A_{11}B_{11}$$

$$M_2 = (A_{11} + A_{12})B_{22}$$

$$M_3 = (A_{21} + A_{22})B_{11}$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$



$$C_{11} = M_5 + M_4 - M_2 + M_6$$

$$C_{12} = M_1 + M_2$$

$$C_{21} = M_3 + M_4$$

$$C_{22} = M_5 + M_1 - M_3 - M_7$$

Strassen矩阵乘法

- ◆传统方法: $O(n^3)$
- ◆分治法: $O(n^{2.81})$
- ◆更快的方法??

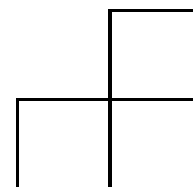
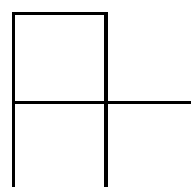
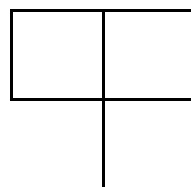
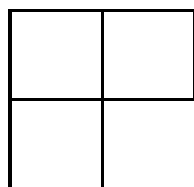
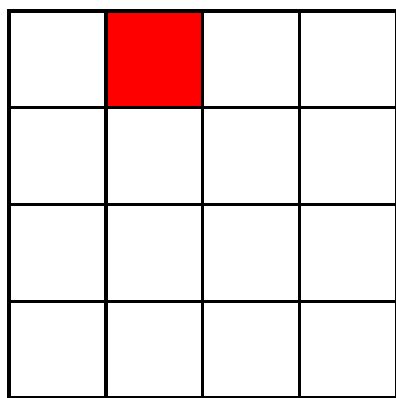
➤ Hopcroft和Kerr已经证明(1971), 计算2个 2×2 矩阵的乘积, 7次乘法是必要的。因此, 要想进一步改进矩阵乘法的时间复杂性, 就不能再基于计算 2×2 矩阵的7次乘法这样的方法了。或许应当研究 3×3 或 5×5 矩阵的更好算法。

➤ 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 $O(n^{2.376})$

➤ 是否能找到 $O(n^2)$ 的算法??? 目前为止还没有结果。

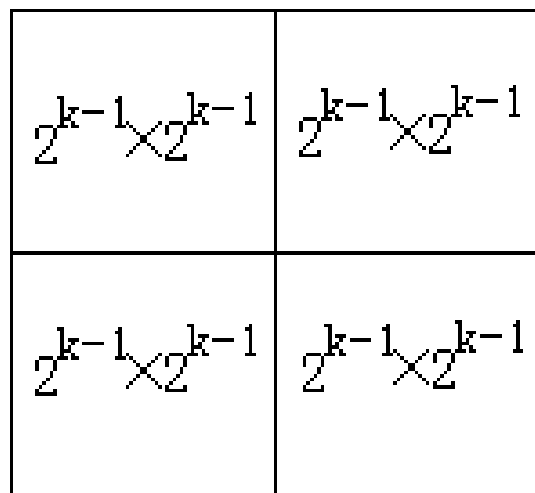
棋盘覆盖

在一个 $2^k \times 2^k$ 个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。

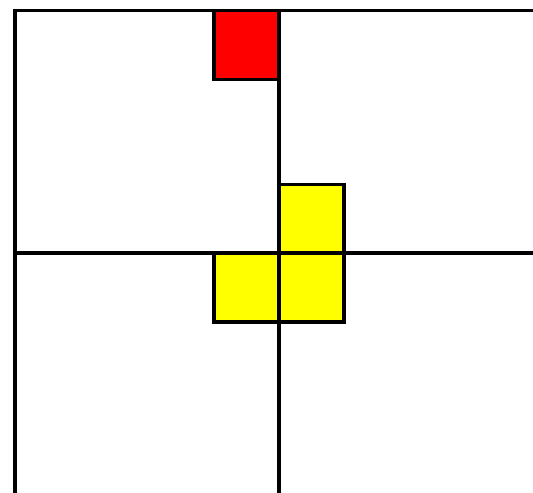


棋盘覆盖

当 $k > 0$ 时，将 $2^k \times 2^k$ 棋盘分割为4个 $2^{k-1} \times 2^{k-1}$ 子棋盘(a)所示。特殊方格必位于4个较小子棋盘之一中，其余3个子棋盘中无特殊方格。为了将这3个无特殊方格的子棋盘转化为特殊棋盘，可以用一个L型骨牌覆盖这3个较小棋盘的会合处，如(b)所示，从而将原问题转化为4个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为棋盘 1×1 。



(a)



(b)

```
public void chessBoard(int tr, int tc, int dr, int dc, int size)
```

```
{
    if (size == 1) return;
    int t = tile++; // L型骨牌号
    s = size/2; // 分割棋盘
    // 覆盖左上角子棋盘
    board[tr + s - 1][tc + s] = t;
    // 覆盖其余方格
    chessBoard(tr, tc+s, tr+s-1, tc+s, s);
    // 覆盖左下角子棋盘
```

```
    // 覆盖左上角子棋盘
```

```
    if (dr < tr + s
```

```
        // 特殊
```

```
        chessBoard
```

```
    else { // 用 t
```

```
        // 用 t
```

```
        board[tr
```

```
        // 覆盖其余方格
```

```
        chessBoard(tr, tc, tr+s-1, tc+s-1, s);
    }
    // 覆盖右上角子棋盘
```

```
    // 覆盖右上角子棋盘
```

```
    if (dr < tr + s && dc >= tc + s)
```

```
        // 特殊方格在此棋盘中
```

```
        chessBoard(tr, tc+s, dr, dc, s);
```

```
    else { // 此棋盘中无特殊方格
```

```
        // 用 t 号 L 型骨牌覆盖左下角
```

复杂度分析

$$T(k) = \begin{cases} O(1) & k = 0 \\ 4T(k-1) + O(1) & k > 0 \end{cases}$$

$T(n) = O(4^k)$ 渐进意义下的最优算法

```
    chessBoard(tr+s, tc, tr+s, tc+s, s);
    // 覆盖右下角子棋盘
```

```
    // 覆盖右下角子棋盘
```

```
    if (dr >= tr + s && dc >= tc + s)
```

```
        // 特殊方格在此棋盘中
```

```
        chessBoard(tr+s, tc+s, dr, dc, s);
```

```
    else { // 用 t 号 L 型骨牌覆盖左上角
```

```
        board[tr + s][tc + s] = t;
```

```
        // 覆盖其余方格
```

```
        chessBoard(tr+s, tc+s, tr+s, tc+s, s);
    }
```

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

复杂度分析

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

$T(n)=O(n\log n)$ 渐进意义下的最优算法

```
public
```

```
{
```

```
if (
```

```
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
```

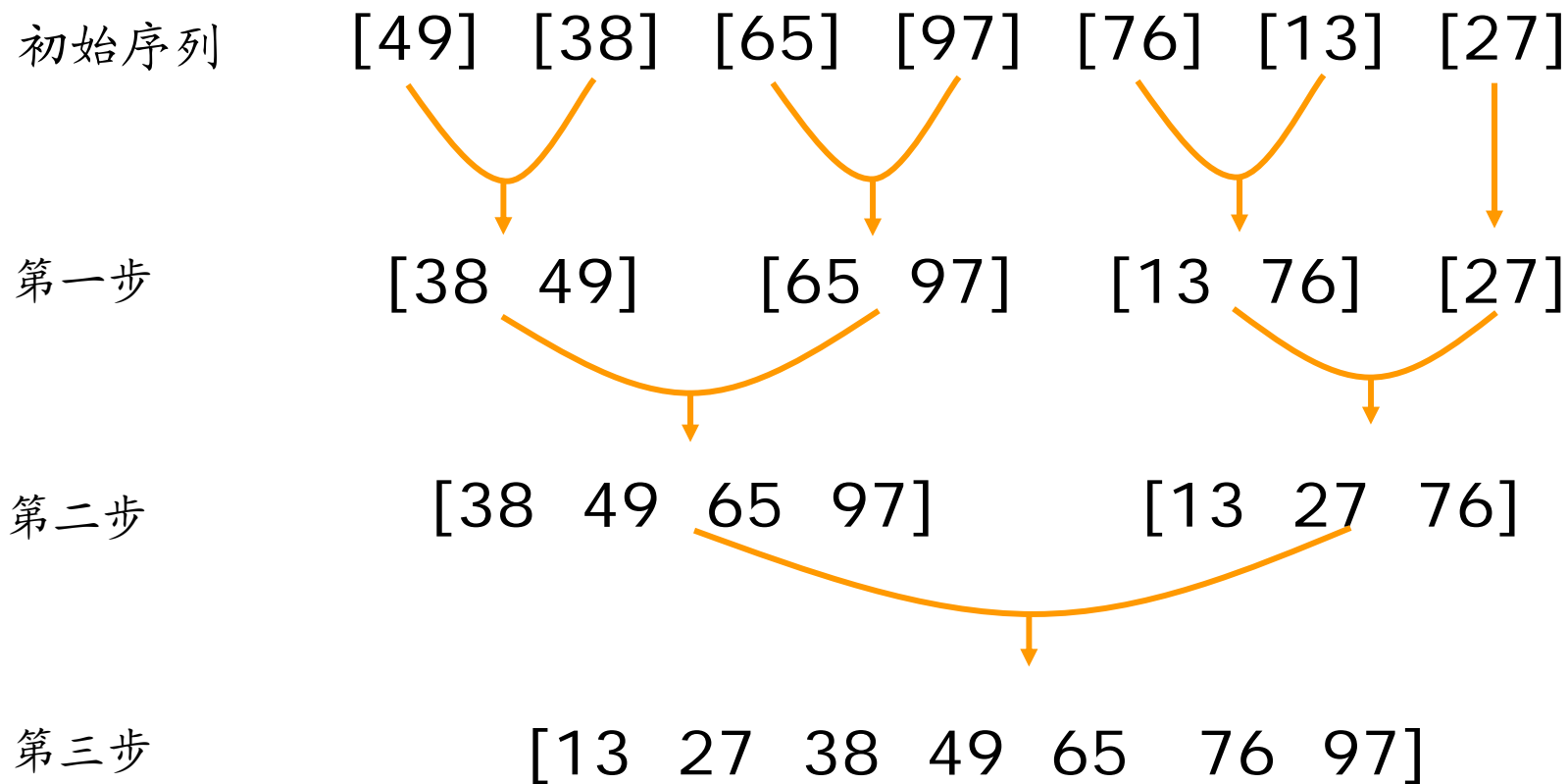
```
copy(a, b, left, right); //复制回数组a
```

```
}
```

```
}
```


合并排序

算法**mergeSort**的递归过程可以消去。



合并排序

 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: $O(n)$

 稳定性: 稳定

思考题: 给定有序表 $A[1:n]$, 修改合并排序算法, 求出该有序表的逆序对数。

快速排序

快速排序是对气泡排序的一种改进方法

它是由C. A. R. Hoare于1962年提出的

在快速排序中，记录的比较和交换是从两端向中间进行的，关键字较大的记录一次就能交换到后面单元，关键字较小的记录一次就能交换到前面单元，记录每次移动的距离较大，因而总的比较和移动次数较少。

```
private static void qSort(int p, int r)
```

```
{
```

```
    if (p<r) {
```

```
        int q=partition(p,r); //以a[p]为基准元素将a[p:r]划分成  
        3段a[p:q-1],a[q]和a[q+1:r], 使得a[p:q-1]中任何元素小于等  
        于a[q], a[q+1:r]中任何元素大于等于a[q]。下标q在划分过  
        程中确定。
```

```
        qSort (p,q-1); //对左半段排序
```

```
        qSort (q+1,r); //对右半段排序
```

```
    }
```

```
}
```

快速排序

```
private static int partition (int p, int r)
```

```
{
    int i = p,
        j = r + 1;
    Comparable x = a[p];
    // 将 >= x 的元素交换到左边区域
    // 将 <= x 的元素交换到右边区域
    while (true) {
        while (a[++i].compareTo(x) < 0);
        while (a[--j].compareTo(x) > 0);
        if (i >= j) break;
        MyMath.swap(a, i, j);
    }
    a[p] = a[j];
    a[j] = x;
    return j;
}
```

{ 6, 7, 5, 2, 5, 8 } 初始序列

{ 6, 7, 5, 2, 5, 8 } j--;

{ 5, 7, 5, 2, 6, 8 } i++;

{ 5, 6, 5, 2, 7, 8 } j--;

{ 5, 2, 5, 6, 7, 8 } i++;

{ 5, 2, 5 } 6 { 7, 8 } 完成

快速排序具有不稳定性。

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

📖 最坏时间复杂度: $O(n^2)$

📖 平均时间复杂度: $O(n \log n)$

📖 辅助空间: $O(n)$ 或 $O(\log n)$

📖 稳定性: 不稳定

```
private static int randomizedPartition (int p, int r)
{
    int i = random(p,r);
    MyMath.swap(a, i, p);
    return partition (p, r);
}
```

线性时间选择

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

```
private static Comparable randomizedSelect(int p,int r,int k)
{
    if (p==r) return a[p];
    int i=randomizedpartition(p,r),
        j=i-p+1;
    if (k<=j) return randomizedSelect(p,i,k);
    else return randomizedSelect(i+1,r,k-j);
}
```

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。

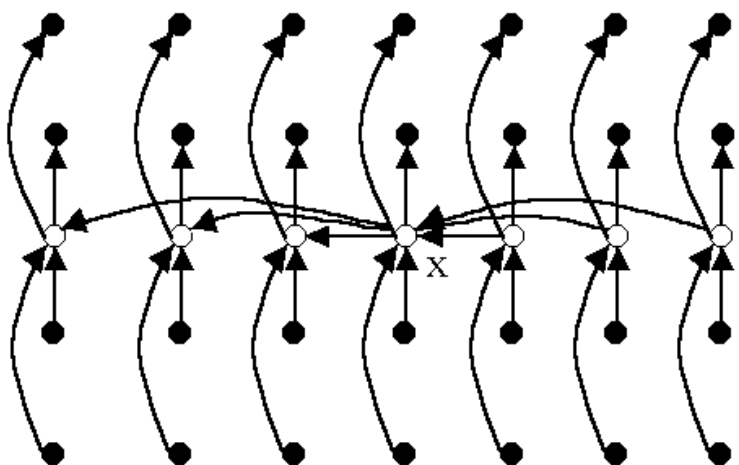
线性时间选择

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($0 < \varepsilon < 1$ 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 $\varepsilon = 9/10$ ，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 $T(n) \leq T(9n/10) + O(n)$ 。由此可得 $T(n) = O(n)$ 。

线性时间选择

- 将 n 个输入元素划分成 $\lceil n/5 \rceil$ 个组，每组5个元素，只可能有一个组不是5个元素。用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共 $\lceil n/5 \rceil$ 个。
- 递归调用 **select** 来找出这 $\lceil n/5 \rceil$ 个元素的中位数。如果 $\lceil n/5 \rceil$ 是偶数，就找它的2个中位数中较大的一个。以这个元素作为划分基准。



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 $n \geq 75$ 时， $3(n-5)/10 \geq n/4$ ，所以按此基准划分所得的2个子数组的长度都至少缩短 $1/4$ 。

```
private static Comparable select (int p, int r, int k)
```

```
{
    if (r-p<5) {
        //用某个简单排序算法对数组a[p:r]排序;
        bubbleSort(p,r);
        return a[p+k-1];
    }
```

```
//将a[
//与a[
//找中
```

```
for ( i
```

```
{
    int s=p+5*i,
```

复杂度分析

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

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

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 $n/5 + 3n/4 = 19n/20 = \varepsilon n$ ， $0 < \varepsilon < 1$ 。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。

```
j=i-p+1;
```

```
if (k<=j) return select(p,i,k);
```

```
else return select(i+1,r,k-j);
```

```
}
```

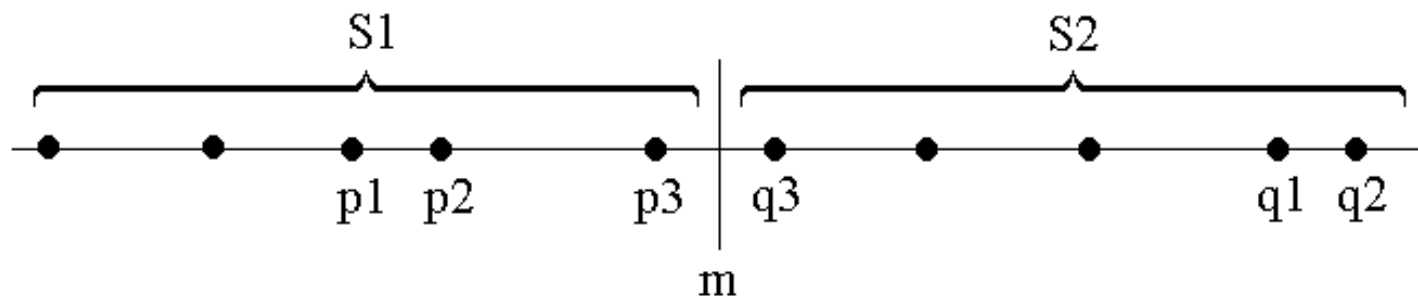
最接近点对问题

◆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 x_1, x_2, \dots, x_n 。最接近点对即为这 n 个实数中相差最小的2个实数。

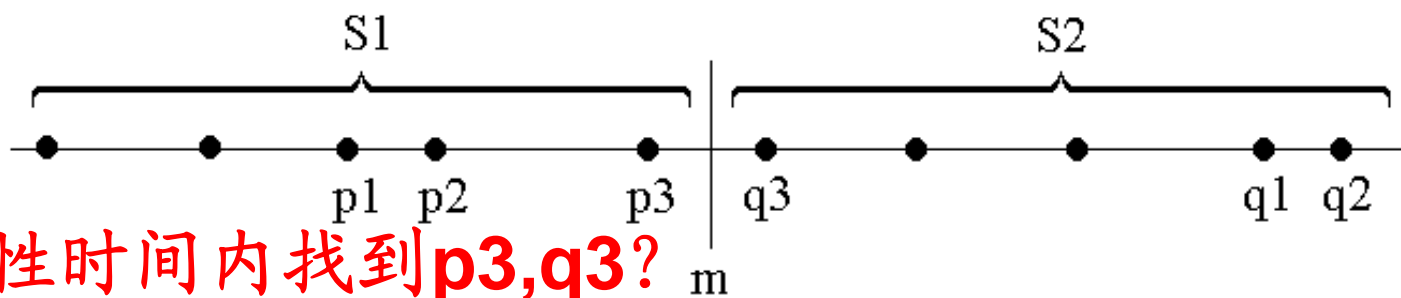
➤假设我们用 x 轴上某个点 m 将 S 划分为2个子集 S_1 和 S_2 ，基于平衡子问题的思想，用 S 中各点坐标的中位数来作分割点。

➤递归地在 S_1 和 S_2 上找出其最接近点对 $\{p_1, p_2\}$ 和 $\{q_1, q_2\}$ ，并设 $d = \min\{|p_1 - p_2|, |q_1 - q_2|\}$ ， S 中的最接近点对或者是 $\{p_1, p_2\}$ ，或者是 $\{q_1, q_2\}$ ，或者是某个 $\{p_3, q_3\}$ ，其中 $p_3 \in S_1$ 且 $q_3 \in S_2$ 。

➤能否在线性时间内找到 p_3, q_3 ?



最接近点对问题



能否在线性时间内找到 p_3, q_3 ?

- ◆如果 S 的最接近点对是 $\{p_3, q_3\}$, 即 $|p_3 - q_3| < d$, 则 p_3 和 q_3 两者与 m 的距离不超过 d , 即 $p_3 \in (m-d, m]$, $q_3 \in (m, m+d]$ 。
- ◆由于在 S_1 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 S_1 和 S_2 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 S_1 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 p_3 和 q_3 。从而我们用线性时间就可以将 S_1 的解和 S_2 的解合并成为 S 的解。

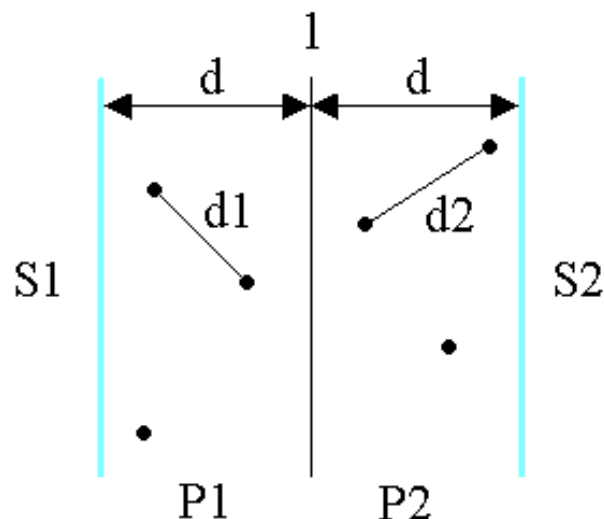
最接近点对问题

◆ 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 S_1 和 S_2 。

➤ 递归地在 S_1 和 S_2 上找出其最小距离 d_1 和 d_2 ，并设 $d = \min\{d_1, d_2\}$ ， S 中的最接近点对或者是 d ，或者是某个 $\{p, q\}$ ，其中 $p \in P_1$ 且 $q \in P_2$ 。

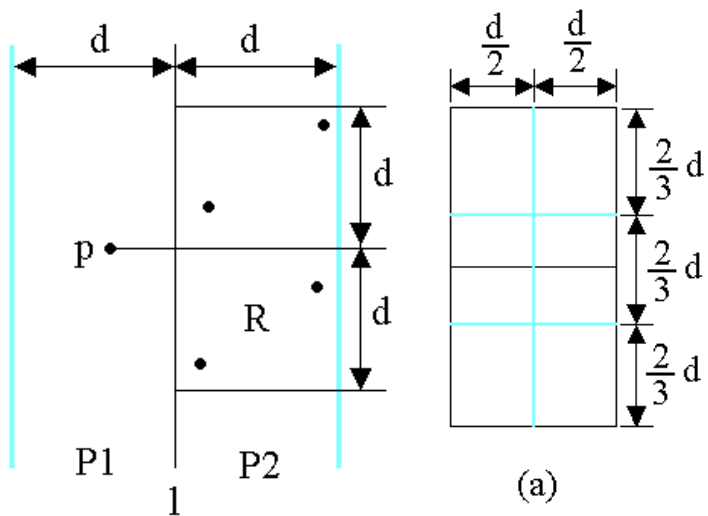
➤ 能否在线性时间内找到 p, q ?



最接近点对问题

能否在线性时间内找到 p_3, q_3 ?

- ◆考虑 P_1 中任意一点 p ，它若与 P_2 中的点 q 构成最接近点对的候选者，则必有 $\text{distance}(p, q) < d$ 。满足这个条件的 P_2 中的点一定落在一个 $d \times 2d$ 的矩形 R 中
- ◆由 d 的意义可知， P_2 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 $6 \times n/2 = 3n$ 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 $(d/2) \times (2d/3)$ 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 $(d/2) \times (2d/3)$ 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

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

$\text{distance}(u, v) < d$ 。这与 d 的意义相矛盾。

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 P_2 中所有 S_2 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 S_2 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 P_1 和 P_2 中所有 S 中点按其 y 坐标排好序，则对 P_1 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 P_1 中每一点最多只要检查 P_2 中排好序的相继6个点。

最接近点对问题

```
public static double cpair2(S)
{
```

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

```
n=|S|;
```

```
if (n <
```

复杂度分析

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

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

中与其
成合

```
构造S
```

```
//S1={p ∈ S | x(p) ≤ m},
```

```
S2={p ∈ S | x(p) > m}
```

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;
```

```
}
```

```
2. d1=cpair2(S1);
```

```
d2=cpair2(S2);
```

```
3. dm=min(d1,d2);
```

设计一个满足以下要求的比赛日程表：

- (1) 每个选手必须与其他 $n-1$ 个选手各赛一次；
- (2) 每个选手一天只能赛一次；
- (3) 循环赛一共进行 $n-1$ 天。

按分治策略，将所有的选手分为两半， n 个选手的比赛日程表就可以通过为 $n/2$ 个选手设计的比赛日程表来决定。递归地用对选手进行分割，直到只剩下2个选手时，比赛日程表的制定就变得很简单。这时只要让这2个选手进行比赛就可以了。

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

循环赛日程表

设计一个满足以下要求的比赛日程表：

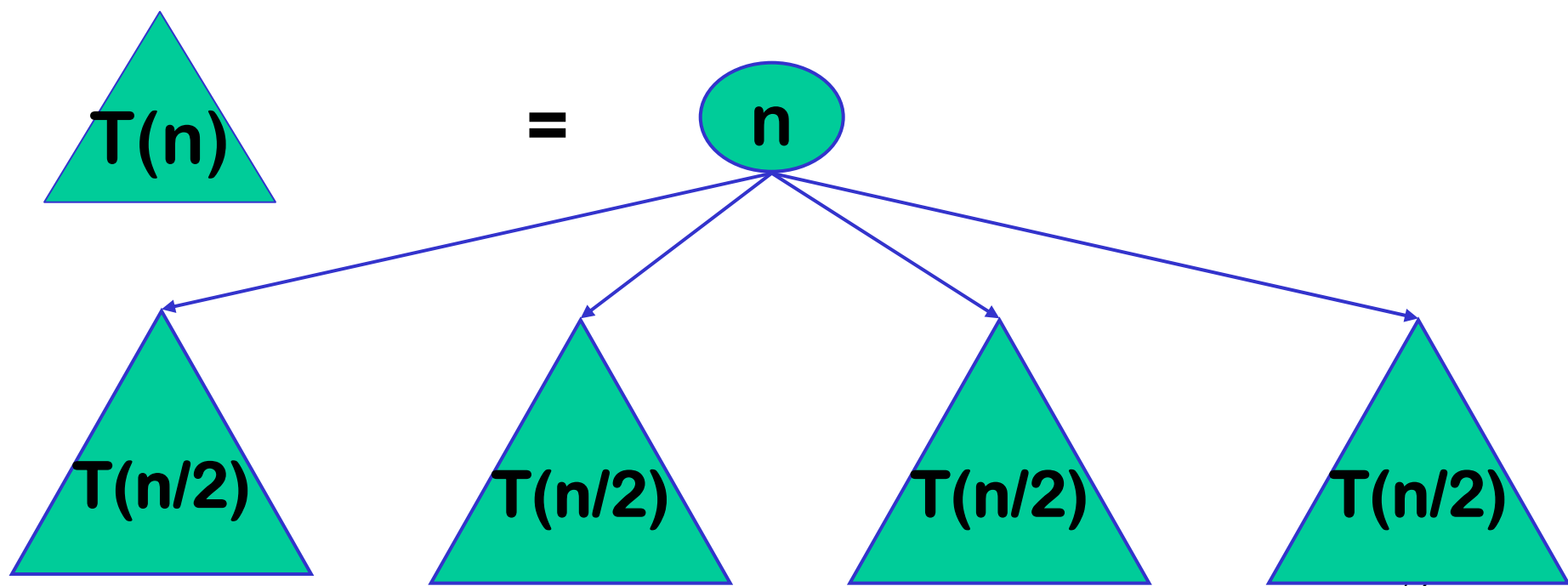
- (1) 每个选手必须与其他 $n-1$ 个选手各赛一次；
- (2) 每个选手一天只能赛一次；
- (3) 循环赛一共进行 $n-1$ 天。

按分治策略，将所有的选手分为两半， n 个选手的比赛日程表就可以通过为 $n/2$ 个选手设计的比赛日程表来决定。递归地用对选手进行分割，直到只剩下2个选手时，比赛日程表的制定就变得很简单。这时只要让这2个选手进行比赛就可以了。

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

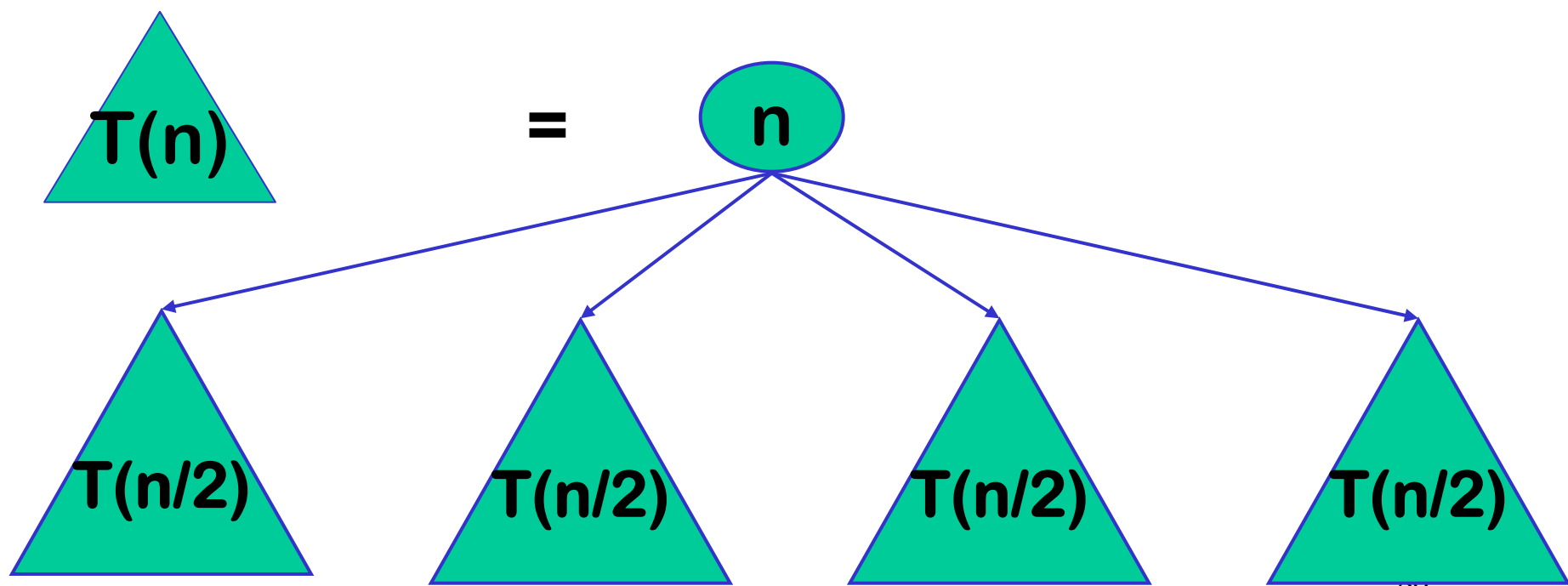
第3章 动态规划

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题



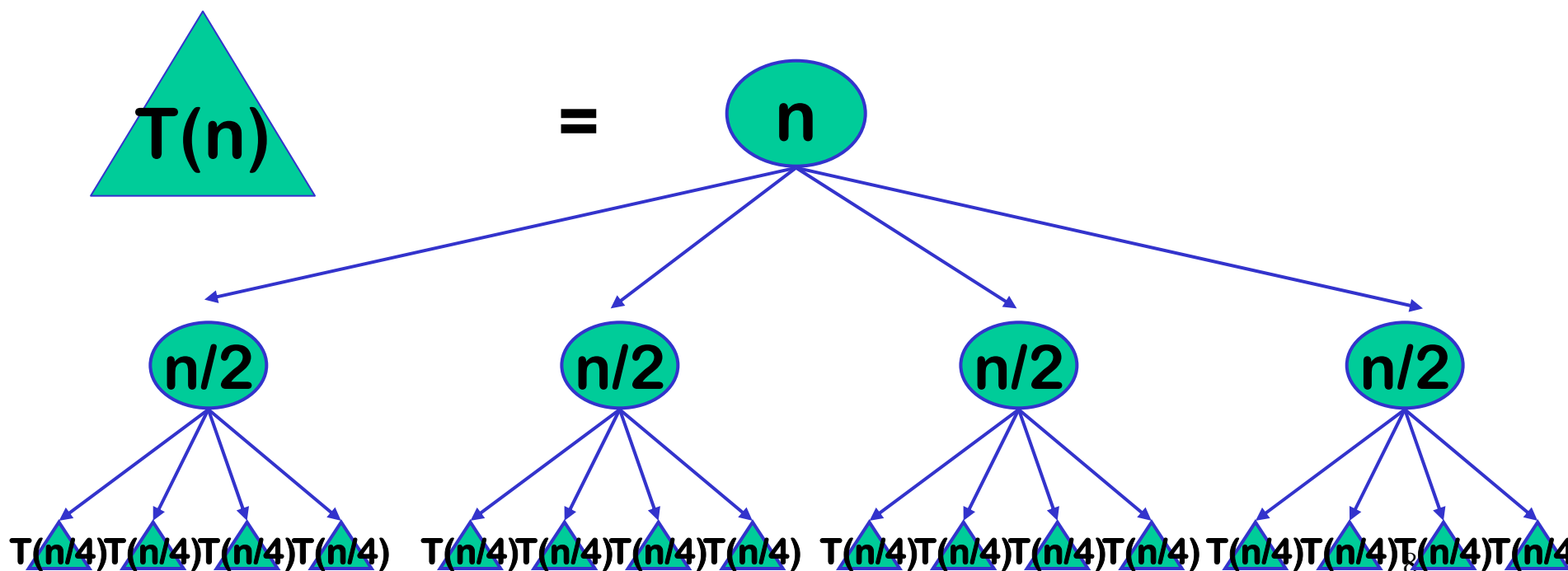
算法总体思想

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题



算法总体思想

- 但是经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次。



算法总体思想

- 如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。

**Those who cannot remember the past
are doomed to repeat it.**

-----George Santayana,
The life of Reason,
Book I: Introduction and
Reason in Common
Sense (1905)

动态规划基本步骤

- 找出最优解的性质，并刻画其结构特征。
- 递归地定义最优值。
- 以自底向上的方式计算出最优值。
- 根据计算最优值时得到的信息，构造最优解。

完全加括号的矩阵连乘积

- ◆ 完全加括号的矩阵连乘积可递归地定义为：
 - (1) 单个矩阵是完全加括号的；
 - (2) 矩阵连乘积 A 是完全加括号的，则 A 可表示为2个完全加括号的矩阵连乘积 B 和 C 的乘积并加括号，即 $A = (BC)$
- ◆ 设有四个矩阵 A, B, C, D ，它们的维数分别是：
 $A = 50 \times 10$ $B = 10 \times 40$ $C = 40 \times 30$ $D = 30 \times 5$
- ◆ 总共有五中完全加括号的方式

$$\begin{array}{lll}(A((BC)D)) & (A(B(CD))) & ((AB)(CD)) \\ (((AB)C)D) & ((A(BC))D) & \end{array}$$

$$16000, 10500, 36000, 87500, 34500$$

矩阵连乘问题

- 给定 n 个矩阵 $\{A_1, A_2, \dots, A_n\}$ ，其中 A_i 与 A_{i+1} 是可乘的， $i = 1, 2, \dots, n-1$ 。考察这 n 个矩阵的连乘积

$$A_1 A_2 \dots A_n$$

- 由于矩阵乘法满足结合律，所以计算矩阵的连乘可以有許多不同的计算次序。这种计算次序可以用加括号的方式来确定。
- 若一个矩阵连乘积的计算次序完全确定，也就是说该连乘积已完全加括号，则可以依此次序反复调用2个矩阵相乘的标准算法计算出矩阵连乘积

矩阵连乘问题

给定 n 个矩阵 $\{A_1, A_2, \dots, A_n\}$ ，其中 A_i 与 A_{i+1} 是可乘的， $i=1, 2, \dots, n-1$ 。如何确定计算矩阵连乘积的计算次序，使得依此次序计算矩阵连乘积需要的数乘次数最少。

◆**穷举法**：列举出所有可能的计算次序，并计算出每一种计算次序相应需要的数乘次数，从中找出一种数乘次数最少的计算次序。

算法复杂度分析：

对于 n 个矩阵的连乘积，设其不同的计算次序为 $P(n)$ 。

由于每种加括号方式都可以分解为两个子矩阵的加括号问题：

$(A_1 \dots A_k)(A_{k+1} \dots A_n)$ 可以得到关于 $P(n)$ 的递推式如下：

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases} \Rightarrow P(n) = \Omega(4^n / n^{3/2})$$

矩阵连乘问题

◆穷举法

◆动态规划

将矩阵连乘积 $A_i A_{i+1} \dots A_j$ 简记为 $A[i:j]$ ，这里 $i \leq j$

考察计算 $A[i:j]$ 的最优计算次序。设这个计算次序在矩阵 A_k 和 A_{k+1} 之间将矩阵链断开， $i \leq k < j$ ，则其相应完全加括号方式为 $(A_i A_{i+1} \dots A_k)(A_{k+1} A_{k+2} \dots A_j)$

计算量： $A[i:k]$ 的计算量加上 $A[k+1:j]$ 的计算量，再加上 $A[i:k]$ 和 $A[k+1:j]$ 相乘的计算量

分析最优解的结构

- 特征：计算 $A[i:j]$ 的最优次序所包含的计算矩阵子链 $A[i:k]$ 和 $A[k+1:j]$ 的次序也是最优的。
- 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为**最优子结构性质**。问题的最优子结构性质是该问题可用动态规划算法求解的显著特征。

建立递归关系

- 设计算 $A[i:j]$, $1 \leq i \leq j \leq n$, 所需要的最少数乘次数 $m[i,j]$, 则原问题的最优值为 $m[1,n]$
- 当 $i=j$ 时, $A[i:j]=A_i$, 因此, $m[i,i]=0$, $i=1,2,\dots,n$
- 当 $i < j$ 时,

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$$

这里 A_i 的维数为 $p_{i-1} \times p_i$

- 可以递归地定义 $m[i,j]$ 为:

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & i < j \end{cases}$$

k 的位置只有 $j-i$ 种可能

计算最优值

- 对于 $1 \leq i \leq j \leq n$ 不同的有序对 (i, j) 对应于不同的子问题。因此，不同子问题的个数最多只有

$$\binom{n}{2} + n = \Theta(n^2)$$

- 由此可见，在递归计算时，**许多子问题被重复计算多次**。这也是该问题可用动态规划算法求解的又一显著特征。
- 用动态规划算法解此问题，可依据其递归式以自底向上的方式进行计算。在计算过程中，保存已解决的子问题答案。每个子问题只计算一次，而在后面需要时只要简单查一下，从而避免大量的重复计算，最终得到多项式时间的算法

用动态规划法求最优解

```
public static void matrixChain(int [] p, int [][] m, int [][] s)
```

```
{
    int n=p.length-1;
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
        for (int i = 1; i <= n - r + 1; i++) {
```

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

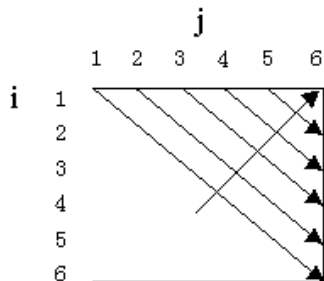
$$m[2][5] = \min \begin{cases} m[2][2] + m[3][5] + p_1 p_2 p_5 = 0 + 2500 + 35 \times 15 \times 20 = 13000 \\ m[2][3] + m[4][5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \times 5 \times 20 = 7125 \\ m[2][4] + m[5][5] + p_1 p_4 p_5 = 4375 + 0 + 35 \times 10 \times 20 = 11375 \end{cases}$$

```
        int j=i+r-1;
        m[i][j] = m[i+1][j] + p[i-1]*p[i]*p[j];
        s[i][j] = i;
        for (int k = i+1; k < j; k++) {
            int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
```

算法复杂度分析:

算法 **matrixChain** 的主要计算量取决于算法中对 r , i 和 k 的 3 重循环。循环体内的计算量为 $O(1)$, 而 3 重循环的总次数为 $O(n^3)$ 。因此算法的计算时间上界为 $O(n^3)$ 。算法所占用的空间显然为 $O(n^2)$ 。

```
        if (t < m[i][j]) {
            m[i][j] = t;
            s[i][j] = k;
        }
    }
}
```



(a) 计算次序

		j					
		1	2	3	4	5	6
i	1	0	15750	7875	9375	11875	15125
	2		0	2625	4375	7125	10500
	3			0	750	2500	5375
	4				0	1000	3500
	5					0	5000
	6						0

(b) $m[i][j]$

		j					
		1	2	3	4	5	6
i	1	0	1	1	3	3	3
	2		0	2	3	3	3
	3			0	3	3	3
	4				0	4	5
	5					0	5
	6						0

(c) $s[i][j]$

动态规划算法的基本要素

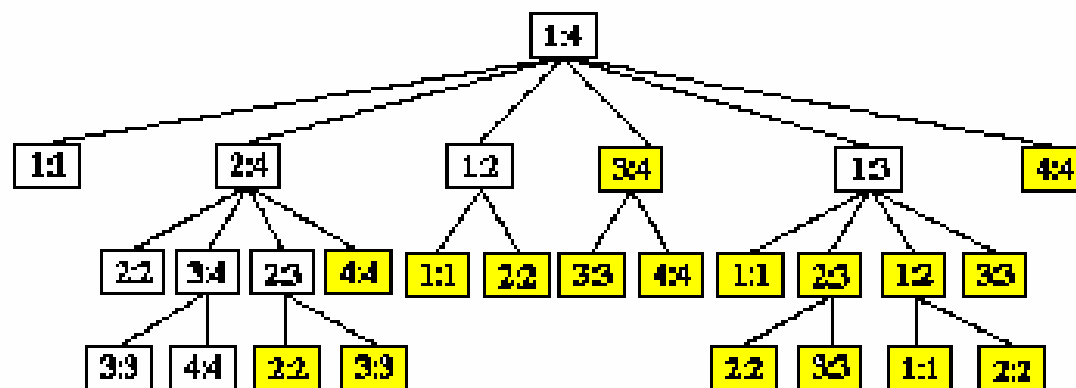
一、最优子结构

- 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为**最优子结构性质**。
- 在分析问题的最优子结构性质时，所用的方法具有普遍性：首先假设由问题的最优解导出的子问题的解不是最优的，然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。
- 利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。最优子结构是问题能用动态规划算法求解的前提。

注意：同一个问题可以有多种方式刻画它的最优子结构，有些表示方法的求解速度更快（空间占用小，问题的维度低）

二、重叠子问题

- 递归算法求解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。这种性质称为**子问题的重叠性质**。
- 动态规划算法，对每一个子问题只解一次，而后将其解保存在一个表格中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。
- 通常不同的子问题个数随问题的大小呈多项式增长。因此用动态规划算法只需要多项式时间，从而获得较高的解题效率。



三、备忘录方法

- 备忘录方法的控制结构与直接递归方法的控制结构相同，区别在于备忘录方法为每个解过的子问题建立了备忘录以备需要时查看，避免了相同子问题的重复求解。

$m \leftarrow 0$

```
private static int lookupChain(int i, int j)
```

```
{
```

```
    if ( $m[i][j] > 0$ ) return  $m[i][j]$ ;
```

```
    if ( $i == j$ ) return 0;
```

```
    int u = lookupChain(i+1,j) +  $p[i-1]*p[i]*p[j]$ ;
```

```
     $s[i][j] = i$ ;
```

```
    for (int k = i+1; k < j; k++) {
```

```
        int t = lookupChain(i,k) + lookupChain(k+1,j) +  $p[i-1]*p[k]*p[j]$ ;
```

```
        if ( $t < u$ ) {
```

```
             $u = t$ ;  $s[i][j] = k$ ;
```

```
        }
```

```
     $m[i][j] = u$ ;
```

```
    return u;
```

```
}
```

最长公共子序列

- 若给定序列 $X=\{x_1, x_2, \dots, x_m\}$ ，则另一序列 $Z=\{z_1, z_2, \dots, z_k\}$ ，是 X 的子序列是指存在一个严格递增下标序列 $\{i_1, i_2, \dots, i_k\}$ 使得对于所有 $j=1, 2, \dots, k$ 有： $z_j=x_{i_j}$ 。例如，序列 $Z=\{B, C, D, B\}$ 是序列 $X=\{A, B, C, B, D, A, B\}$ 的子序列，相应的递增下标序列为 $\{2, 3, 5, 7\}$ 。
- 给定2个序列 X 和 Y ，当另一序列 Z 既是 X 的子序列又是 Y 的子序列时，称 Z 是序列 X 和 Y 的**公共子序列**。
- 给定2个序列 $X=\{x_1, x_2, \dots, x_m\}$ 和 $Y=\{y_1, y_2, \dots, y_n\}$ ，找出 X 和 Y 的最长公共子序列。

最长公共子序列的结构

设序列 $X=\{x_1, x_2, \dots, x_m\}$ 和 $Y=\{y_1, y_2, \dots, y_n\}$ 的最长公共子序列为 $Z=\{z_1, z_2, \dots, z_k\}$ ，则

- (1) 若 $x_m = y_n$ ，则 $z_k = x_m = y_n$ ，且 z_{k-1} 是 x_{m-1} 和 y_{n-1} 的最长公共子序列。
- (2) 若 $x_m \neq y_n$ 且 $z_k \neq x_m$ ，则 Z 是 x_{m-1} 和 Y 的最长公共子序列。
- (3) 若 $x_m \neq y_n$ 且 $z_k \neq y_n$ ，则 Z 是 X 和 y_{n-1} 的最长公共子序列。

由此可见，2个序列的最长公共子序列包含了这2个序列的前缀的最长公共子序列。因此，最长公共子序列问题具有**最优子结构性质**。

子问题的递归结构

由最长公共子序列问题的最优子结构性质建立子问题最优值的递归关系。用 $c[i][j]$ 记录序列 X 和 Y 的最长公共子序列的长度。其中， $X_i = \{x_1, x_2, \dots, x_i\}$; $Y_j = \{y_1, y_2, \dots, y_j\}$ 。当 $i=0$ 或 $j=0$ 时，空序列是 X_i 和 Y_j 的最长公共子序列。故此时 $C[i][j]=0$ 。其他情况下，由最优子结构性质可建立递归关系如下：

$$c[i][j] = \begin{cases} 0 & i = 0, j = 0 \\ c[i-1][j-1] + 1 & i, j > 0; x_i = y_j \\ \max\{c[i][j-1], c[i-1][j]\} & i, j > 0; x_i \neq y_j \end{cases}$$

计算最优值

由于在所考虑的子问题空间中，总共有 $\theta(mn)$ 个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。

Algorithm lcsLength(x,y,b)

```
1: m ← x.length-1;
2: n ← y.length-1;
3: c[i][0]=0; c[0][i]=0;
4: for (int i = 1; i ≤ m; i++)
5:   for (int j = 1; j ≤ n; j++)
6:     if (x[i]==y[j])
7:       c[i][j]=c[i-1][j-1]+1;
8:       b[i][j]=1;
9:     else if (c[i-1][j]>=c[i][j-1])
10:      c[i][j]=c[i-1][j];
11:      b[i][j]=2;
12:   else
13:     c[i][j]=c[i][j-1];
14:     b[i][j]=3;
```

构造最长公共子序列

Algorithm lcs(int i,int j,char [] x,int [][] b)

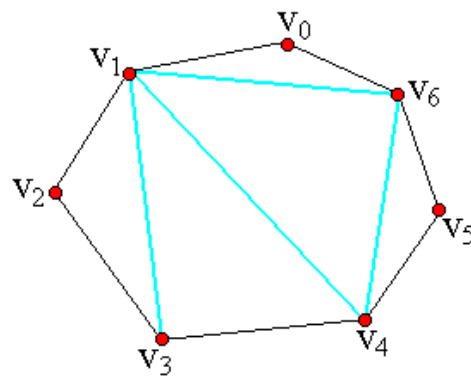
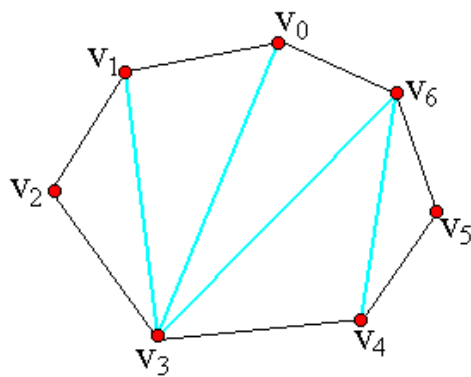
```
{
  if (i ==0 || j==0) return;
  if (b[i][j]== 1){
    lcs(i-1,j-1,x,b);
    System.out.print(x[i]);
  }
  else if (b[i][j]== 2) lcs(i-1,j,x,b);
  else lcs(i,j-1,x,b);
}
```

算法的改进

- 在算法 **lcsLength** 和 **lcs** 中，可进一步将数组 **b** 省去。事实上，数组元素 $c[i][j]$ 的值仅由 $c[i-1][j-1]$ ， $c[i-1][j]$ 和 $c[i][j-1]$ 这3个数组元素的值所确定。对于给定的数组元素 $c[i][j]$ ，可以不借助于数组 **b** 而仅借助于 **c** 本身在时间内确定 $c[i][j]$ 的值是由 $c[i-1][j-1]$ ， $c[i-1][j]$ 和 $c[i][j-1]$ 中哪一个值所确定的。
- 如果只需要计算最长公共子序列的长度，则算法的空间需求可大大减少。事实上，在计算 $c[i][j]$ 时，只用到数组 **c** 的第 *i* 行和第 *i*-1 行。因此，用2行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至 $O(\min(m,n))$ 。

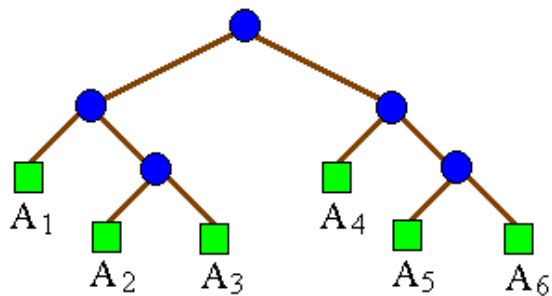
凸多边形最优三角剖分

- 用多边形顶点的逆时针序列表示凸多边形，即 $P=\{v_0, v_1, \dots, v_{n-1}\}$ 表示具有 n 条边的凸多边形。
- 若 v_i 与 v_j 是多边形上不相邻的2个顶点，则线段 $v_i v_j$ 称为多边形的一条弦。弦将多边形分割成2个多边形 $\{v_i, v_{i+1}, \dots, v_j\}$ 和 $\{v_j, v_{j+1}, \dots, v_i\}$ 。
- 多边形的三角剖分是将多边形分割成互不相交的三角形的弦的集合 T 。
- 给定凸多边形 P ，以及定义在由多边形的边和弦组成的三角形上的权函数 w 。要求确定该凸多边形的三角剖分，使得即该三角剖分中诸三角形上权之和为最小。

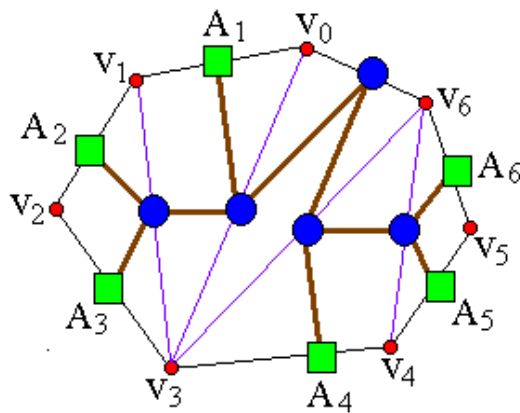


三角剖分的结构及其相关问题

- 一个表达式的完全加括号方式相应于一棵完全二叉树，称为表达式的语法树。例如，完全加括号的矩阵连乘积 $((A_1(A_2A_3))(A_4(A_5A_6)))$ 所相应的语法树如图 (a) 所示。
- 凸多边形 $\{v_0, v_1, \dots, v_{n-1}\}$ 的三角剖分也可以用语法树表示。例如，图 (b) 中凸多边形的三角剖分可用图 (a) 所示的语法树表示。
- 矩阵连乘积中的每个矩阵 A_i 对应于凸 $(n+1)$ 边形中的一条边 $v_{i-1}v_i$ 。三角剖分中的一条弦 $v_i v_j$, $i < j$, 对应于矩阵连乘积 $A[i+1:j]$ 。



(a)



(b)

最优子结构性质

- 凸多边形的最优三角剖分问题有最优子结构性质。
- 事实上，若凸 $(n+1)$ 边形 $P=\{v_0, v_1, \dots, v_n\}$ 的最优三角剖分 T 包含三角形 $v_0 v_k v_n$ ， $1 \leq k \leq n-1$ ，则 T 的权为3个部分权的和：三角形 $v_0 v_k v_n$ 的权，子多边形 $\{v_0, v_1, \dots, v_k\}$ 和 $\{v_k, v_{k+1}, \dots, v_n\}$ 的权之和。可以断言，由 T 所确定的这2个子多边形的三角剖分也是最优的。因为若有 $\{v_0, v_1, \dots, v_k\}$ 或 $\{v_k, v_{k+1}, \dots, v_n\}$ 的更小权的三角剖分将导致 T 不是最优三角剖分的矛盾。

最优三角剖分的递归结构

- 定义 $t[i][j]$, $1 \leq i < j \leq n$ 为凸子多边形 $\{v_{i-1}, v_i, \dots, v_j\}$ 的最优三角剖分所对应的权函数值, 即其最优值。为方便起见, 设退化的多边形 $\{v_{i-1}, v_i\}$ 具有权值 0。据此定义, 要计算的凸 $(n+1)$ 边形 P 的最优权值为 $t[1][n]$ 。
- $t[i][j]$ 的值可以利用最优子结构性质递归地计算。当 $j-i \geq 1$ 时, 凸子多边形至少有 3 个顶点。由最优子结构性质, $t[i][j]$ 的值应为 $t[i][k]$ 的值加上 $t[k+1][j]$ 的值, 再加上三角形 $v_{i-1}v_kv_j$ 的权值, 其中 $i \leq k \leq j-1$ 。由于在计算时还不知道 k 的确切位置, 而 k 的所有可能位置只有 $j-i$ 个, 因此可以在这 $j-i$ 个位置中选出使 $t[i][j]$ 值达到最小的位置。由此, $t[i][j]$ 可递归地定义为:

$$t[i][j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{t[i][k] + t[k+1][j] + w(v_{i-1}v_kv_j)\} & i < j \end{cases}$$

多边形游戏

多边形游戏是一个单人玩的游戏，开始时有一个由 n 个顶点构成的多边形。每个顶点被赋予一个整数值，每条边被赋予一个运算符“+”或“*”。所有边依次用整数从1到 n 编号。

游戏第1步，将一条边删除。

随后 $n-1$ 步按以下方式操作：

- (1) 选择一条边 E 以及由 E 连接着的2个顶点 V_1 和 V_2 ；
- (2) 用一个新的顶点取代边 E 以及由 E 连接着的2个顶点 V_1 和 V_2 。将由顶点 V_1 和 V_2 的整数值通过边 E 上的运算得到的结果赋予新顶点。

最后，所有边都被删除，游戏结束。游戏的得分就是所剩顶点上的整数值。

问题：对于给定的多边形，计算最高得分。

最优子结构性质

- 在所给多边形中，从顶点 i ($1 \leq i \leq n$) 开始，长度为 j (链中有 j 个顶点) 的顺时针链 $p(i, j)$ 可表示为 $v[i], op[i+1], \dots, v[i+j-1]$ 。
- 如果这条链的最后一次合并运算在 $op[i+s]$ 处发生 ($1 \leq s \leq j-1$)，则可在 $op[i+s]$ 处将链分割为2个子链 $p(i, s)$ 和 $p(i+s, j-s)$ 。
- 设 $m1$ 是对子链 $p(i, s)$ 的任意一种合并方式得到的值，而 a 和 b 分别是在所有可能的合并中得到的最小值和最大值。 $m2$ 是 $p(i+s, j-s)$ 的任意一种合并方式得到的值，而 c 和 d 分别是在所有可能的合并中得到的最小值和最大值。依此定义有 $a \leq m1 \leq b, c \leq m2 \leq d$
 - (1) 当 $op[i+s] = '+'$ 时，显然有 $a+c \leq m \leq b+d$
 - (2) 当 $op[i+s] = '*'$ 时，有 $\min\{ac, ad, bc, bd\} \leq m \leq \max\{ac, ad, bc, bd\}$
- 换句话说，主链的最大值和最小值可由子链的最大值和最小值得到。

图像压缩

图像的变位压缩存储格式将所给的象素点序列 $\{p_1, p_2, \dots, p_n\}$, $0 \leq p_i \leq 255$ 分割成 m 个连续段 S_1, S_2, \dots, S_m 。第 i 个象素段 S_i 中 ($1 \leq i \leq m$)，有 $l[i]$ 个象素，且该段中每个象素都只用 $b[i]$ 位表示。设 $t[i] = \sum_{k=1}^{i-1} l[k]$ 则第 i 个象素段 S_i 为

设 $h_i = \left\lceil \log \left(\max_{t[i]+1 \leq k \leq t[i]+l[i]} p_k + 1 \right) \right\rceil$ ，则 $h_i \leq b[i] \leq 8$ 。因此需要用 3 位表示 $b[i]$ ，如果限制 $1 \leq l[i] \leq 255$ ，则需要用 8 位表示 $l[i]$ 。因此，第 i 个象素段所需的存储空间为 $l[i] * b[i] + 11$ 位。按此格式存储象素序列 $\{p_1, p_2, \dots, p_n\}$ ，需要 $\sum_{i=1}^m l[i] * b[i] + 11m$ 位的存储空间。

图像压缩问题要求确定象素序列 $\{p_1, p_2, \dots, p_n\}$ 的最优分段，使得依此分段所需的存储空间最少。每个分段的长度不超过 256 位。

设 $l[i]$, $b[i]$, 是 $\{p_1, p_2, \dots, p_n\}$ 的最优分段。显而易见, $l[1]$, $b[1]$ 是 $\{p_1, \dots, p_{l[1]}\}$ 的最优分段, 且 $l[i]$, $b[i]$, 是 $\{p_{l[1]+1}, \dots, p_n\}$ 的最优分段。即图像压缩问题满足最优子结构性质。

设 $s[i]$, $1 \leq i \leq n$, 是像素序列 $\{p_1, \dots, p_n\}$ 的最优分段所需的存储位数。由最优子结构性质易知:

$$s[i] = \min_{1 \leq k \leq \min\{i, 256\}} \{s[i-k] + k * b_{\max(i-k+1, i)}\} + 11$$

其中 $b_{\max(i, j)} = \left\lceil \log \left(\max_{i \leq k \leq j} \{p_k\} + 1 \right) \right\rceil$

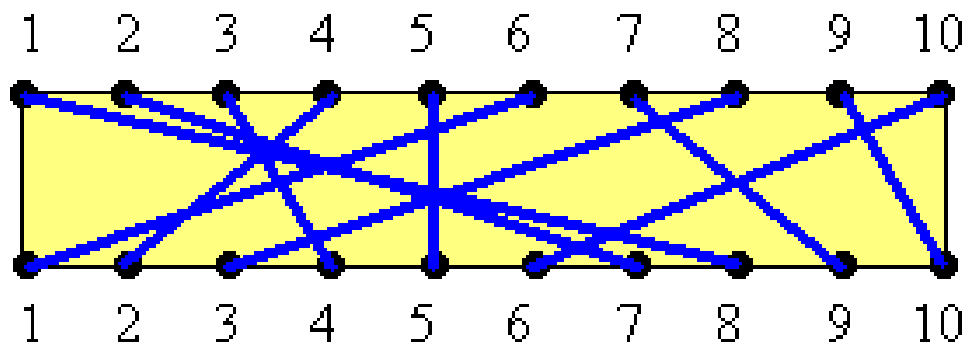
算法复杂度分析:

由于算法 **compress** 中对 k 的循环次数不超这 256, 故对每一个确定的 i , 可在时间 $O(1)$ 内完成的计算。因此整个算法所需的计算时间为 $O(n)$ 。

电路布线

在一块电路板的上、下2端分别有 n 个接线柱。根据电路设计，要求用导线 $(i, \pi(i))$ 将上端接线柱与下端接线柱相连，如图所示。其中 $\pi(i)$ 是 $\{1, 2, \dots, n\}$ 的一个排列。导线 $(i, \pi(i))$ 称为该电路板上的第 i 条连线。对于任何 $1 \leq i < j \leq n$ ，第 i 条连线和第 j 条连线相交的充分且必要的条件是 $\pi(i) > \pi(j)$ 。

电路布线问题要确定将哪些连线安排在第一层上，使得该层上有尽可能多的连线。换句话说，该问题要求确定导线集 $\text{Nets} = \{(i, \pi(i)), 1 \leq i \leq n\}$ 的最大不相交子集。



记 $N(i, j) = \{t \mid (t, \pi(t)) \in Nets, t \leq i, \pi(t) \leq j\}$ 。 $N(i, j)$ 的最大不相交子集为 $MNS(i, j)$ 。 $Size(i, j) = |MNS(i, j)|$ 。

(1) 当 $i=1$ 时, $MNS(1, j) = N(1, j) = \begin{cases} \emptyset & j < \pi(1) \\ \{(1, \pi(1))\} & j \geq \pi(1) \end{cases}$

(2) (1) 当 $i=1$ 时 $Size(1, j) = \begin{cases} 0 & j < \pi(1) \\ 1 & j \geq \pi(1) \end{cases}$

(2) 当 $i>1$ 时 $Size(i, j) = \begin{cases} Size(i-1, j) & j < \pi(i) \\ \max\{Size(i-1, j), Size(i-1, \pi(i)-1) + 1\} & j \geq \pi(i) \end{cases}$

1, $\pi(i)-1$) 的最大不相交子集。

2.3 若 $(i, \pi(i)) \notin N(i, j)$, 则对任意 $(t, \pi(t)) \in MNS(i, j)$ 有 $t < i$ 。从而 $MNS(i, j) \subseteq N(i-1, j)$ 因此, $Size(i, j) \leq Size(i-1, j)$ 。另一方面 $MNS(i-1, j) \subseteq N(i, j)$, 故又有 $Size(i, j) \geq Size(i-1, j)$, 从而 $Size(i, j) = Size(i-1, j)$ 。

流水作业调度

n 个作业 $\{1, 2, \dots, n\}$ 要在由2台机器 $M1$ 和 $M2$ 组成的流水线上完成加工。每个作业加工的顺序都是先在 $M1$ 上加工，然后在 $M2$ 上加工。 $M1$ 和 $M2$ 加工作业 i 所需的时间分别为 a_i 和 b_i 。

流水作业调度问题要求确定这 n 个作业的最优加工顺序，使得从第一个作业在机器 $M1$ 上开始加工，到最后一个作业在机器 $M2$ 上加工完成所需的时间最少。

分析：

- 直观上，一个最优调度应使机器 $M1$ 没有空闲时间，且机器 $M2$ 的空闲时间最少。在一般情况下，机器 $M2$ 上会有机器空闲和作业积压2种情况。
- 设全部作业的集合为 $N=\{1, 2, \dots, n\}$ 。 $S \subseteq N$ 是 N 的作业子集。在一般情况下，机器 $M1$ 开始加工 S 中作业时，机器 $M2$ 还在加工其他作业，要等时间 t 后才可利用。将这种情况下完成 S 中作业所需的最短时间记为 $T(S, t)$ 。流水作业调度问题的最优值为 $T(N, 0)$ 。

流水作业调度

设 π 是所给 n 个流水作业的一个最优调度，它所需的加工时间为 $a_{\pi(1)}+T'$ 。其中 T' 是在机器 M_2 的等待时间为 $b_{\pi(1)}$ 时，安排作业 $\pi(2), \dots, \pi(n)$ 所需的时间。

记 $S=N-\{\pi(1)\}$ ，则有 $T'=T(S, b_{\pi(1)})$ 。

证明：事实上，由 T 的定义知 $T' \geq T(S, b_{\pi(1)})$ 。若 $T' > T(S, b_{\pi(1)})$ ，设 π' 是作业集 S 在机器 M_2 的等待时间为 $b_{\pi(1)}$ 情况下的一个最优调度。则 $\pi(1), \pi'(2), \dots, \pi'(n)$ 是 N 的一个调度，且该调度所需的时间为 $a_{\pi(1)} + T(S, b_{\pi(1)}) < a_{\pi(1)} + T'$ 。这与 π 是 N 的最优调度矛盾。故 $T' \leq T(S, b_{\pi(1)})$ 。从而 $T' = T(S, b_{\pi(1)})$ 。这就证明了流水作业调度问题具有最优子结构的性质。

由流水作业调度问题的最优子结构性质可知，

$$T(N, 0) = \min_{1 \leq i \leq n} \{a_i + T(N - \{i\}, b_i)\}$$

$$T(S, t) = \min_{i \in S} \{a_i + T(S - \{i\}, b_i + \max\{t - a_i, 0\})\}$$

Johnson不等式

对递归式的深入分析表明，算法可进一步得到简化。

设 π 是作业集 S 在机器 M_2 的等待时间为 t 时的任一最优调度。若

$\pi(1)=i, \pi(2)=j$ 。则由动态规划递归式可得：

$$T(S,t)=a_i+T(S-\{i\},b_i+\max\{t-a_i,0\})=a_i+a_j+T(S-\{i,j\},t_{ij})$$

$$\begin{aligned}\text{其中, } t_{ij} &= b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\} \\ &= b_j + b_i - a_j + \max\{\max\{t - a_i, 0\}, a_j - b_i\} \\ &= b_j + b_i - a_j + \max\{t - a_i, a_j - b_i, 0\} \\ &= b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_i, a_i\}\end{aligned}$$

如果作业 i 和 j 满足 $\min\{b_i, a_j\} \geq \min\{b_j, a_i\}$ ，则称作业 i 和 j 满足Johnson不等式。

流水作业调度的Johnson法则

$$t_{ij} = b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_i, a_i\}$$

交换作业i和作业j的加工顺序，得到作业集S的另一调度，它所需的加工时间为 $T'(S, t) = a_i + a_j + T(S - \{i, j\}, t_{ji})$

$$\text{其中 } t_{ji} = b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_j, a_j\}$$

当作业i和j满足Johnson不等式时，有

$$\max\{-b_i, -a_j\} \leq \max\{-b_j, -a_i\}$$

$$a_i + a_j + \max\{-b_i, -a_j\} \leq a_i + a_j + \max\{-b_j, -a_i\}$$

$$\max\{a_i + a_j - b_i, a_i\} \leq \max\{a_i + a_j - b_j, a_j\}$$

$$\max\{t, a_i + a_j - b_i, a_i\} \leq \max\{t, a_i + a_j - b_j, a_j\}$$

由此可见当作业i和作业j不满足Johnson不等式时，交换它们的加工顺序后，不增加加工时间。对于流水作业调度问题，必存在最优调度 π ，使得作业 $\pi(i)$ 和 $\pi(i+1)$ 满足Johnson不等式。进一步还可以证明，调度满足Johnson法则当且仅当对任意 $i < j$ 有

$$\min\{b_{\pi(i)}, a_{\pi(j)}\} \geq \min\{b_{\pi(j)}, a_{\pi(i)}\}$$

由此可知，所有满足Johnson法则的调度均为最优调度。

流水作业调度问题的Johnson算法

- (1) 令 $N_1 = \{i \mid a_i < b_i\}$, $N_2 = \{i \mid a_i \geq b_i\}$;
- (2) 将 N_1 中作业依 a_i 的非减序排序; 将 N_2 中作业依 b_i 的非增序排序;
- (3) N_1 中作业接 N_2 中作业构成满足Johnson法则的最优调度。

算法复杂度分析:

算法的主要计算时间花在对作业集的排序。因此, 在最坏情况下算法所需的计算时间为 $O(n \log n)$ 。所需的空间为 $O(n)$ 。

0-1背包问题

给定 n 种物品和一背包。物品 i 的重量是 w_i ，其价值为 v_i ，背包的容量为 C 。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大？

0-1背包问题是一个特殊的整数规划问题。

$$\begin{aligned} & \max \sum_{i=1}^n v_i x_i \\ & \begin{cases} \sum_{i=1}^n w_i x_i \leq C \\ x_i \in \{0,1\}, 1 \leq i \leq n \end{cases} \end{aligned}$$

设所给0-1背包问题的子问题

$$\max \sum_{k=i}^n v_k x_k$$

$$\left\{ \sum_{k=i}^n w_k x_k \leq j \right.$$

算法复杂度分析:

从 $m(i, j)$ 的递归式容易看出, 算法需要 $O(nc)$ 计算时间。当背包容量 c 很大时, 算法需要的计算时间较多。例如, 当 $c > 2^n$ 时, 算法需要 $\Omega(n2^n)$ 计算时间。

品为 i ,
最优子

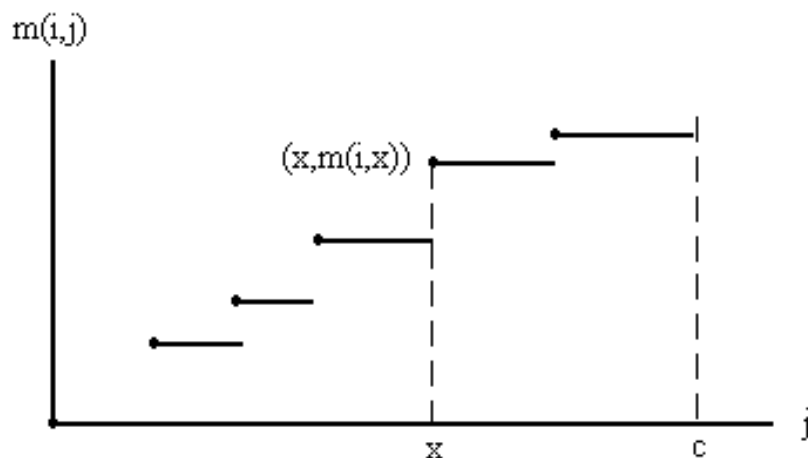
的
 $i+$
结构性质, 可以建立计算并 $m(i, j)$ 的递归式如下。

$$m(i, j) = \begin{cases} \max\{m(i+1, j), m(i+1, j-w_i) + v_i\} & j \geq w_i \\ m(i+1, j) & 0 \leq j < w_i \end{cases}$$

$$m(n, j) = \begin{cases} v_n & j \geq w_n \\ 0 & 0 \leq j < w_n \end{cases}$$

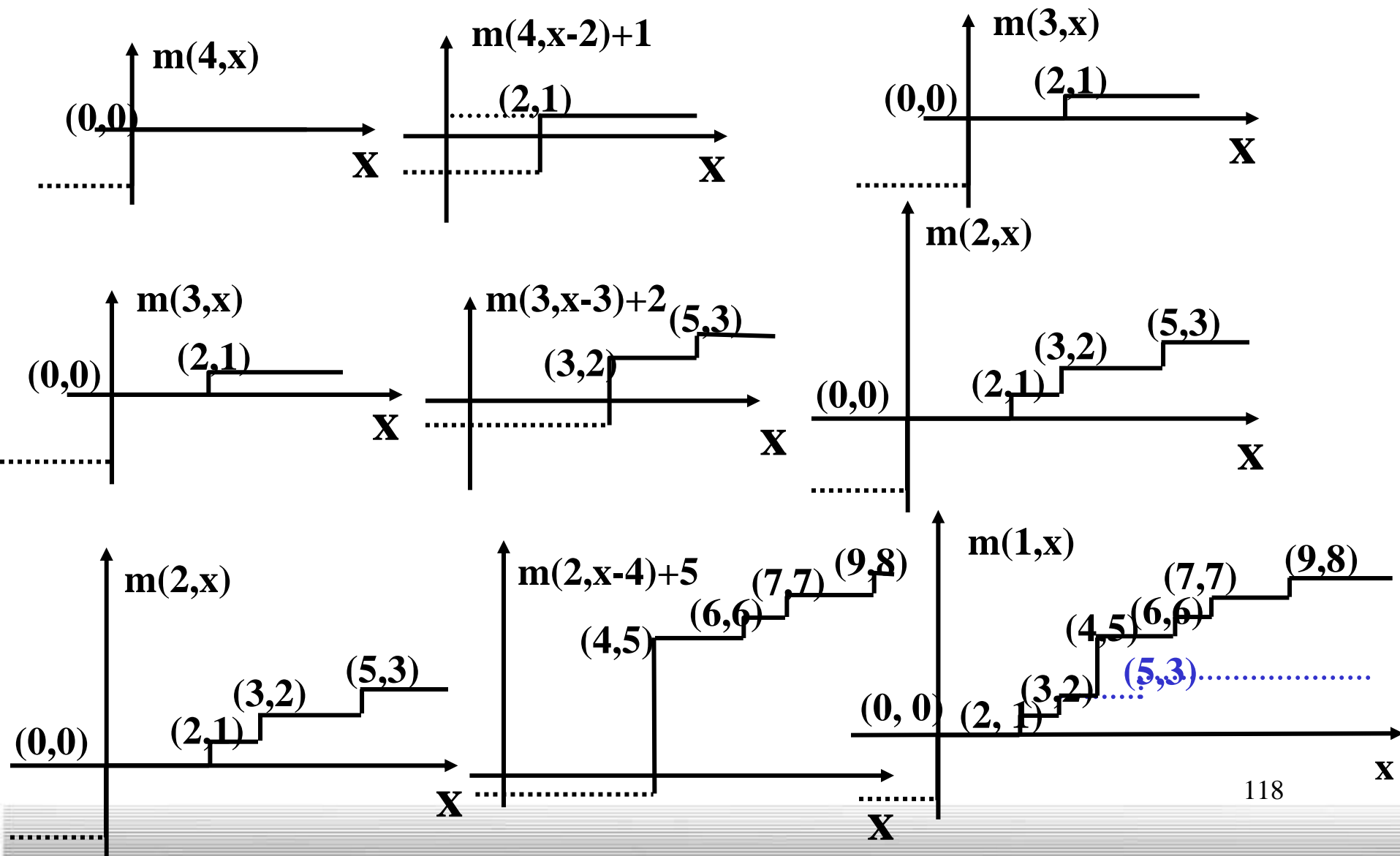
算法改进

由 $m(i,j)$ 的递归式容易证明，在一般情况下，对每一个确定的 $i(1 \leq i \leq n)$ ，函数 $m(i,j)$ 是关于变量 j 的阶梯状单调不减函数。跳跃点是这一类函数的描述特征。在一般情况下，函数 $m(i,j)$ 由其全部跳跃点惟一确定。如图所示。



对每一个确定的 $i(1 \leq i \leq n)$ ，用一个表 $p[i]$ 存储函数 $m(i, j)$ 的全部跳跃点。表 $p[i]$ 可依计算 $m(i, j)$ 的递归式递归地由表 $p[i+1]$ 计算，初始时 $p[n+1]=\{(0, 0)\}$ 。

$n=3$, $c=6$, $w=\{4, 3, 2\}$, $v=\{5, 2, 1\}$.



算法改进

- 函数 $m(i,j)$ 是由函数 $m(i+1,j)$ 与函数 $m(i+1,j-w_i)+v_i$ 作 \max 运算得到的。因此，函数 $m(i,j)$ 的全部跳跃点包含于函数 $m(i+1,j)$ 的跳跃点集 $p[i+1]$ 与函数 $m(i+1,j-w_i)+v_i$ 的跳跃点集 $q[i+1]$ 的并集中。易知， $(s,t) \in q[i+1]$ 当且仅当 $w_i \leq s \leq c$ 且 $(s-w_i, t-v_i) \in p[i+1]$ 。因此，容易由 $p[i+1]$ 确定跳跃点集 $q[i+1]$ 如下
$$q[i+1] = p[i+1] \oplus (w_i, v_i) = \{(j+w_i, m(i,j)+v_i) \mid (j, m(i,j)) \in p[i+1]\}$$
- 另一方面，设 (a, b) 和 (c, d) 是 $p[i+1] \cup q[i+1]$ 中的2个跳跃点，则当 $c \geq a$ 且 $d < b$ 时， (c, d) 受控于 (a, b) ，从而 (c, d) 不是 $p[i]$ 中的跳跃点。除受控跳跃点外， $p[i+1] \cup q[i+1]$ 中的其他跳跃点均为 $p[i]$ 中的跳跃点。
- 由此可见，在递归地由表 $p[i+1]$ 计算表 $p[i]$ 时，可先由 $p[i+1]$ 计算出 $q[i+1]$ ，然后合并表 $p[i+1]$ 和表 $q[i+1]$ ，并清除其中的受控跳跃点得到表 $p[i]$ 。

典型例子 (二)

$n=5$, $c=10$, $w=\{2, 2, 6, 5, 4\}$, $v=\{6, 3, 5, 4, 6\}$ 。

初始时 $p[6]=\{(0,0)\}$, $(w_5, v_5)=(4,6)$ 。因此,

$q[6]=p[6] \oplus (w_5, v_5) = \{(4,6)\}$ 。

$p[5]=\{(0,0), (4,6)\}$ 。

$q[5]=p[5] \oplus (w_4, v_4) = \{(5,4), (9,10)\}$ 。从跳跃点集 $p[5]$ 与 $q[5]$ 的并集

$p[5] \cup q[5] = \{(0,0), (4,6), (5,4), (9,10)\}$ 中看到跳跃点 $(5,4)$ 受控于跳跃点 $(4,6)$ 。将受控跳跃点 $(5,4)$ 清除后, 得到

$p[4]=\{(0,0), (4,6), (9,10)\}$

$q[4]=p[4] \oplus (6, 5) = \{(6, 5), (10, 11)\}$

$p[3]=\{(0, 0), (4, 6), (9, 10), (10, 11)\}$

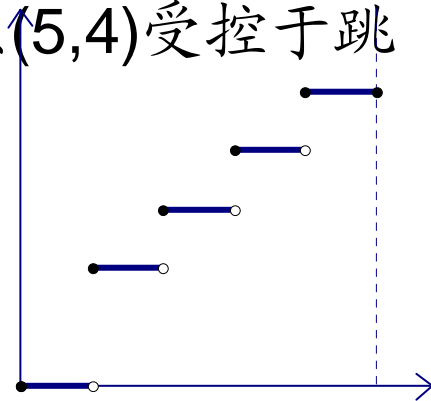
$q[3]=p[3] \oplus (2, 3) = \{(2, 3), (6, 9)\}$

$p[2]=\{(0, 0), (2, 3), (4, 6), (6, 9), (9, 10), (10, 11)\}$

$q[2]=p[2] \oplus (2, 6) = \{(2, 6), (4, 9), (6, 12), (8, 15)\}$

$p[1]=\{(0, 0), (2, 6), (4, 9), (6, 12), (8, 15)\}$

$p[1]$ 的最后的那个跳跃点 $(8,15)$ 给出所求的最优值为 $m(1,c)=15$ 。



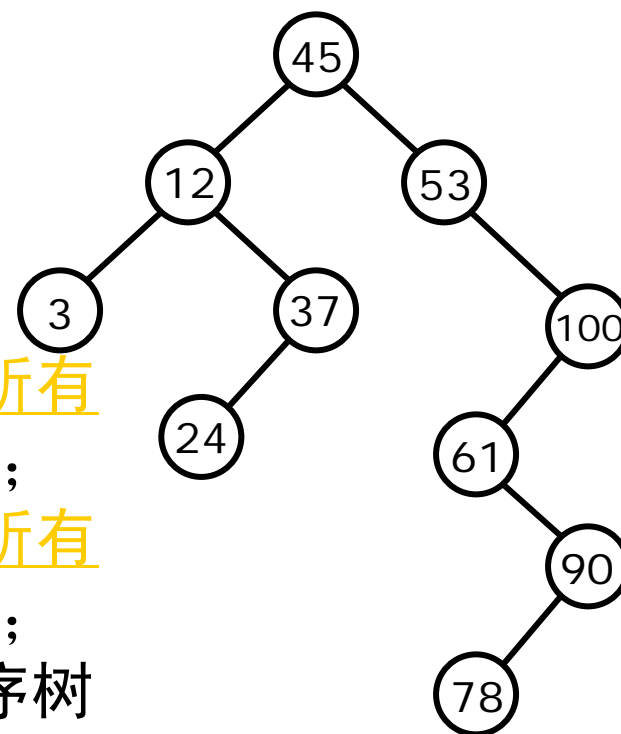
算法复杂度分析

上述算法的主要计算量在于计算跳跃点集 $p[i] (1 \leq i \leq n)$ 。由于 $q[i+1] = p[i+1] \oplus (w_i, v_i)$ ，故计算 $q[i+1]$ 需要 $O(|p[i+1]|)$ 计算时间。合并 $p[i+1]$ 和 $q[i+1]$ 并清除受控跳跃点也需要 $O(|p[i+1]|)$ 计算时间。从跳跃点集 $p[i]$ 的定义可以看出， $p[i]$ 中的跳跃点相应于 x_i, \dots, x_n 的 0/1 赋值。因此， $p[i]$ 中跳跃点个数不超过 2^{n-i+1} 。由此可见，算法计算跳跃点集 $p[i]$ 所花费的计算时间为 $O\left(\sum_{i=2}^n |p[i+1]| \right) = O\left(\sum_{i=2}^n 2^{n-i} \right) = O(2^n)$ 。从而，改进后算法的计算时间复杂性为 $O(2^n)$ 。当所给物品的重量 $w_i (1 \leq i \leq n)$ 是整数时， $|p[i]| \leq c+1, (1 \leq i \leq n)$ 。在这种情况下，改进后算法的计算时间复杂性为 $O(\min\{nc, 2^n\})$ 。

最优二叉搜索树

- 什么是二叉搜索树？

- (1) 若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
- (2) 若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
- (3) 它的左、右子树也分别为二叉排序树



在随机的情况下，二叉查找树的平均查找长度和 $\log n$ 是等数量级的

二叉查找树的期望耗费

- 查找成功与不成功的概率

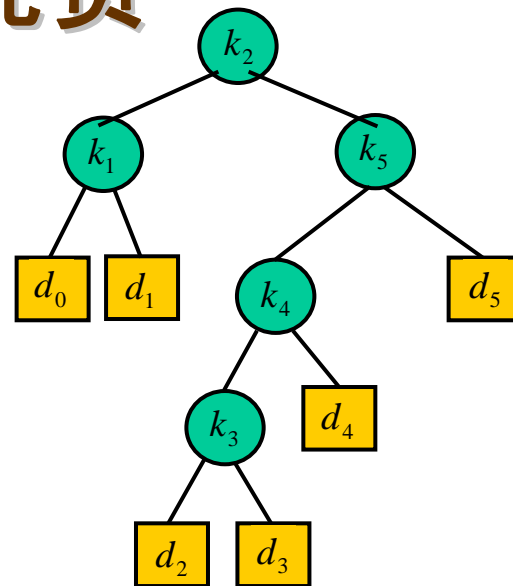
$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$$

- 二查找树的期望耗费

$E(\text{search cost in } T)$

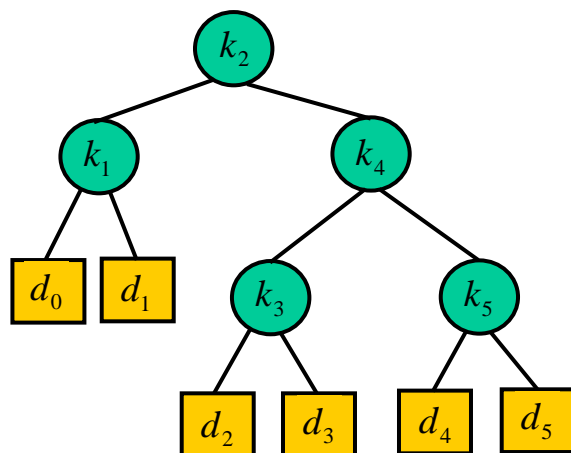
$$= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i$$



- 有 n 个节点的二叉树的个数为: $\Omega(4^n / n^{3/2})$
- 穷举搜索法的时间复杂度为指数级

二叉查找树的期望耗费示例



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

最优二叉搜索树

最优二叉搜索树 T_{ij} 的平均路长为 p_{ij} ，则所求的最优值为 $p_{1,n}$ 。
由最优二叉搜索树问题的最优子结构性质可建立计算 p_{ij} 的递归式如下

$$w_{i,j}p_{i,j} = w_{i,j} + \min_{i \leq k \leq j} \{w_{i,k-1}p_{i,k-1} + w_{k+1,j}p_{k+1,j}\}$$

记 $w_{i,j}p_{i,j}$ 为 $m(i,j)$ ，则 $m(1,n)=w_{1,n}p_{1,n}=p_{1,n}$ 为所求的最优值。计算 $m(i,j)$ 的递归式为

$$m(i, j) = w_{i,j} + \min_{i \leq k \leq j} \{m(i, k-1) + m(k+1, j)\}, \quad i \leq j$$

$$m(i, i-1) = 0, \quad 1 \leq i \leq n$$

注意到，

$$\min_{i \leq k \leq j} \{m(i, k-1) + m(k+1, j)\} = \min_{s[i][j-1] \leq k \leq s[i+1][j]} \{m(i, k-1) + m(k+1, j)\}$$

可以得到 $O(n^2)$ 的算法

第4章 贪心算法

第4章 贪心算法

顾名思义，贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。当然，希望贪心算法得到的最终结果也是整体最优的。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。如单源最短路径问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

第4章 贪心算法

本章主要知识点:

- 4.1 活动安排问题
- 4.2 贪心算法的基本要素
- 4.3 最优装载
- 4.4 哈夫曼编码
- 4.5 单源最短路径
- 4.6 最小生成树
- 4.7 多机调度问题
- 4.8 贪心算法的理论基础

4.1 活动安排问题

活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合，是可以用贪心算法有效求解的很好例子。该问题要求高效地安排一系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法使得尽可能多的活动能兼容地使用公共资源。

4.1 活动安排问题

设有 n 个活动的集合 $E = \{1, 2, \dots, n\}$ ，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。每个活动 i 都有一个要求使用该资源的起始时间 s_i 和一个结束时间 f_i ，且 $s_i < f_i$ 。如果选择了活动 i ，则它在半开时间区间 $[s_i, f_i)$ 内占用资源。若区间 $[s_i, f_i)$ 与区间 $[s_j, f_j)$ 不相交，则称活动 i 与活动 j 是相容的。也就是说，当 $s_i \geq f_j$ 或 $s_j \geq f_i$ 时，活动 i 与活动 j 相容。

4.1 活动安排问题

在下面所给出的解活动安排问题的贪心算法**greedySelector** :

```
• public static int greedySelector(int [] s, int [] f, boolean a[])  
• {  
•     int n=s.length-1;  
•     a[1]=true;  
•     int j=1;  
•     int count=1;  
•     for (int i=2;i<=n;i++) {  
•         if (s[i]>=f[j]) {  
•             a[i]=true;  
•             j=i;  
•             count++;  
•         }  
•         else a[i]=false;  
•     }  
•     return count;  
• }
```

各活动的起始时间和结束时间存储于数组s和f中且按结束时间的非减序排列

4.1 活动安排问题

由于输入的活动以其完成时间的**非减序**排列，所以算法greedySelector每次总是选择**具有最早完成时间**的相容活动加入集合A中。直观上，按这种方法选择相容活动为未安排活动留下尽可能多的时间。也就是说，该算法的贪心选择的意义是**使剩余的可安排时间段极大化**，以便安排尽可能多的相容活动。

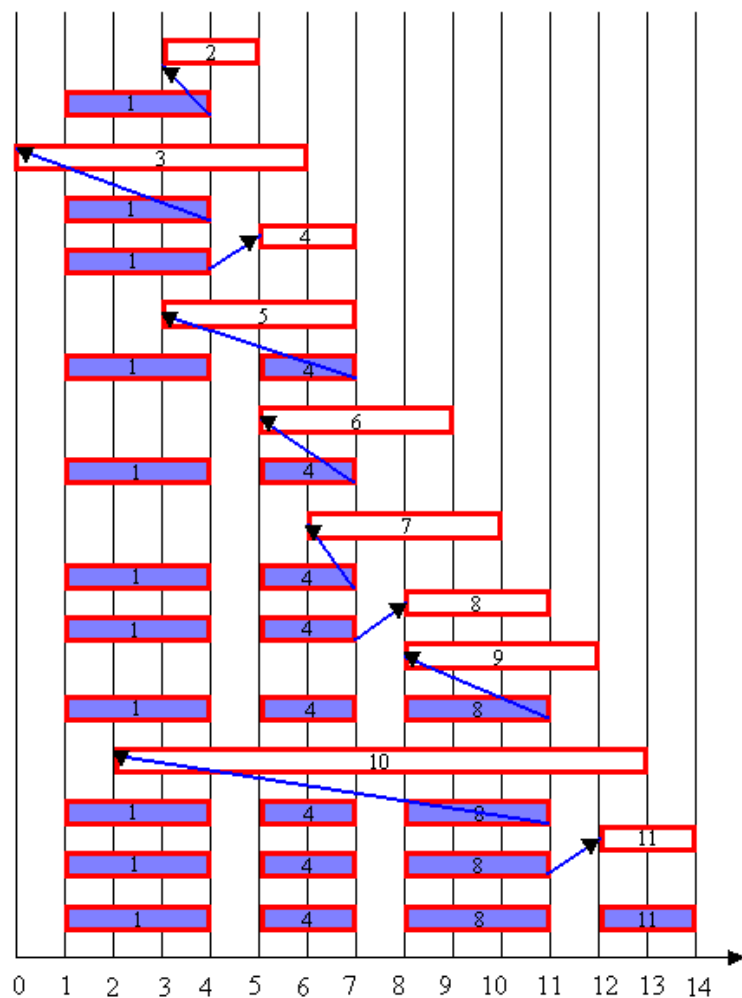
算法greedySelector的效率极高。当输入的活动已按结束时间的非减序排列，算法只需 **$O(n)$** 的时间安排n个活动，使最多的活动能相容地使用公共资源。如果所给出的活动未按非减序排列，可以用 **$O(n \log n)$** 的时间重排。

4.1 活动安排问题

例：设待安排的11个活动的开始时间和结束时间按结束时间的非减序排列如下：

i	1	2	3	4	5	6	7	8	9	10	11
S[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

4.1 活动安排问题



算法greedySelector 的计算过程如左图所示。图中每行相应于算法的一次迭代。阴影长条表示的活动是已选入集合A的活动，而空白长条表示的活动是当前正在检查相容性的活动。

4.1 活动安排问题

若被检查的活动 i 的开始时间 S_i 小于最近选择的活动 j 的结束时间 f_j ，则不选择活动 i ，否则选择活动 i 加入集合 A 中。

贪心算法并不总能求得问题的**整体最优解**。但对于活动安排问题，贪心算法greedySelector却总能求得的整体最优解，即它最终所确定的相容活动集合 A 的规模最大。这个结论可以用数学归纳法证明。

4.2 贪心算法的基本要素

本节着重讨论可以用贪心算法求解的问题的一般特征。

对于一个具体的问题，怎么知道是否可用贪心算法解此问题，以及能否得到问题的最优解呢？这个问题很难给予肯定的回答。

但是，从许多可以用贪心算法求解的问题中看到这类问题一般具有2个重要的性质：**贪心选择性质**和**最优子结构性**。

4.2 贪心算法的基本要素

1. 贪心选择性质

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。

动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。

对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。

4.2 贪心算法的基本要素

2. 最优子结构性质

当一个问题的最优解包含其子问题的最优解时，称此问题具有**最优子结构性质**。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。

4.2 贪心算法的基本要素

3. 贪心算法与动态规划算法的差异

贪心算法和动态规划算法都要求问题具有最优子结构性质，这是2类算法的一个共同点。但是，对于具有**最优子结构**的问题应该选用贪心算法还是动态规划算法求解？是否能用动态规划算法求解的问题也能用贪心算法求解？下面研究2个经典的**组合优化问题**，并以此说明贪心算法与动态规划算法的主要差别。

4.2 贪心算法的基本要素

- 0-1背包问题:

给定 n 种物品和一个背包。物品 i 的重量是 W_i ，其价值为 V_i ，背包的容量为 C 。应如何选择装入背包的物品，使得装入背包中物品的总价值最大？

在选择装入背包的物品时，对每种物品 i 只有2种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i 。

4.2 贪心算法的基本要素

- 背包问题:

与0-1背包问题类似, 所不同的是在选择物品 i 装入背包时, 可以选择物品 i 的一部分, 而不一定要全部装入背包, $1 \leq i \leq n$ 。

这2类问题都具有最优子结构性质, 极为相似, 但背包问题可以用贪心算法求解, 而0-1背包问题却不能用贪心算法求解。

4.2 贪心算法的基本要素

用贪心算法解背包问题的基本步骤:

首先计算每种物品单位重量的价值 V_i/W_i ，然后，依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。若将这种物品全部装入背包后，背包内的物品总重量未超过 C ，则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直地进行下去，直到背包装满为止。

具体算法可描述如下页：

4.2 贪心算法的基本要素

```
• public static float knapsack(float c,float [] w, float [] v,float [] x)
• {
•     int n=v.length;
•     Element [] d = new Element [n];
•     for (int i = 0; i < n; i++) d[i] = new Element(w[i],v[i],i);
•     MergeSort.mergeSort(d);
•     int i;
•     float opt=0;
•     for (i=0;i<n;i++) x[i]=0;
•     for (i=0;i<n;i++) {
•         if (d[i].w>c) break;
•         x[d[i].i]=1;
•         opt+=d[i].v;
•         c-=d[i].w;
•     }
•     if (i<n){
•         x[d[i].i]=c/d[i].w;
•         opt+=x[d[i].i]*d[i].v;
•     }
•     return opt;
• }
```

算法knapsack的主要计算时间在于将各种物品依其单位重量的价值从大到小排序。因此，算法的计算时间上界为 $O(n \log n)$ 。当然，为了证明算法的正确性，还必须证明背包问题具有贪心选择性质。

4.2 贪心算法的基本要素

对于0-1背包问题，贪心选择之所以不能得到最优解是因为在这种情况下，它无法保证最终能将背包装满，部分闲置的背包空间使每公斤背包空间的价值降低了。事实上，在考虑0-1背包问题时，应比较选择该物品和不选择该物品所导致的最终方案，然后再作出最好选择。由此就导出许多互相重叠的子问题。这正是该问题可用动态规划算法求解的另一重要特征。

实际上也是如此，动态规划算法的确可以有效地解0-1背包问题。

4.3 最优装载

有一批集装箱要装上一艘载重量为 c 的轮船。其中集装箱 i 的重量为 W_i 。最优装载问题要求确定在装载体积不受限制的情况下，将尽可能多的集装箱装上轮船。

1. 算法描述

最优装载问题可用贪心算法求解。采用重量最轻者先装的贪心选择策略，可产生最优装载问题的最优解。具体算法描述如下页。

4.3 最优装载

```
• public static float loading(float c, float [] w, int [] x)
• {
•     int n=w.length;
•     Element [] d = new Element [n];
•     for (int i = 0; i < n; i++)
•         d[i] = new Element(w[i],i);
•     MergeSort.mergeSort(d);
•     float opt=0;
•     for (int i = 0; i < n; i++) x[i] = 0;
•     for (int i = 0; i < n && d[i].w <= c; i++) {
•         x[d[i].i] = 1;
•         opt+=d[i].w;
•         c -= d[i].w;
•     }
•     return opt;
• }
```

其中Element类说明为
参见本书P115

4.3 最优装载

2. 贪心选择性质

可以证明最优装载问题具有贪心选择性质。

3. 最优子结构性质

最优装载问题具有最优子结构性质。

由最优装载问题的贪心选择性质和最优子结构性质，容易证明算法loading的正确性。

算法loading的主要计算量在于将集装箱依其重量从小到大排序，故算法所需的计算时间为 $O(n \log n)$ 。

4.4 哈夫曼编码

哈夫曼编码是广泛地用于数据文件压缩的十分有效的编码方法。其压缩率通常在20%~90%之间。哈夫曼编码算法用字符在文件中出现的频率表来建立一个用0, 1串表示各字符的最优表示方式。

给出现频率高的字符较短的编码, 出现频率较低的字符以较长的编码, 可以大大缩短总码长。

1. 前缀码

对每一个字符规定一个0, 1串作为其代码, 并要求任一字符的代码都不是其他字符代码的前缀。这种编码称为**前缀码**。

4.4 哈夫曼编码

编码的前缀性质可以使译码方法非常简单。

表示**最优前缀码**的二叉树总是一棵**完全二叉树**，即树中任一结点都有2个儿子结点。

平均码长定义为：
$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

使平均码长达到最小的前缀码编码方案称为给定编码字符集C的**最优前缀码**。

4.4 哈夫曼编码

2. 构造哈夫曼编码

哈夫曼提出构造最优前缀码的贪心算法，由此产生的编码方案称为**哈夫曼编码**。

哈夫曼算法以自底向上的方式构造表示最优前缀码的二叉树 T 。

算法以 $|C|$ 个叶结点开始，执行 $|C| - 1$ 次的“合并”运算后产生最终所要求的树 T 。

4.4 哈夫曼编码

在书上给出的算法huffmanTree中，编码字符集中每一字符 c 的频率是 $f(c)$ 。以 f 为键值的优先队列 Q 用在贪心选择时有效地确定算法当前要合并的2棵具有最小频率的树。一旦2棵具有最小频率的树合并后，产生一棵新的树，其频率为合并的2棵树的频率之和，并将新树插入优先队列 Q 。经过 $n-1$ 次的合并后，优先队列中只剩下一棵树，即所要求的树 T 。

算法huffmanTree用最小堆实现优先队列 Q 。初始化优先队列需要 $O(n)$ 计算时间，由于最小堆的removeMin和put运算均需 $O(\log n)$ 时间， $n-1$ 次的合并总共需要 $O(n \log n)$ 计算时间。因此，关于 n 个字符的哈夫曼算法的计算时间为 $O(n \log n)$ 。

4.4 哈夫曼编码

3. 哈夫曼算法的正确性

要证明哈夫曼算法的正确性，只要证明最优前缀码问题具有贪心选择性质和最优子结构性质。

(1) 贪心选择性质

(2) 最优子结构性质

4.5 单源最短路径

给定带权有向图 $G = (V, E)$ ，其中每条边的权是非负实数。另外，还给定 V 中的一个顶点，称为源。现在要计算从源到所有其他各顶点的最短路长度。这里路的长度是指路上各边权之和。这个问题通常称为单源最短路径问题。

1. 算法基本思想

Dijkstra算法是解单源最短路径问题的贪心算法。

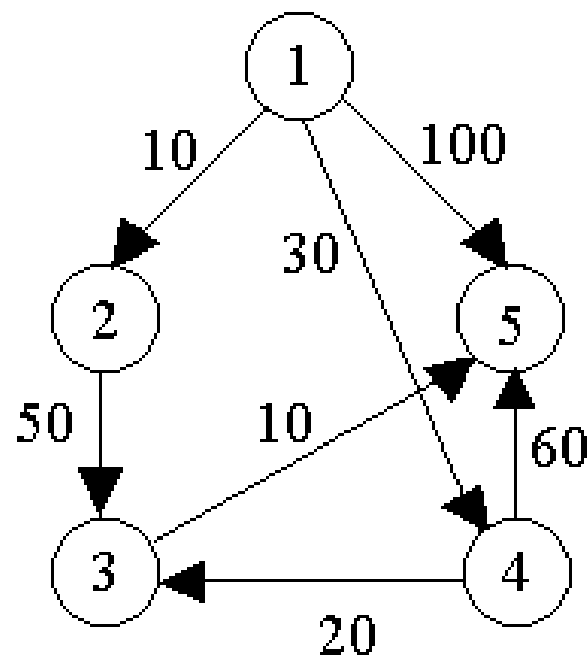
4.5 单源最短路径

其**基本思想**是，设置顶点集合 S 并不断地作**贪心选择**来扩充这个集合。一个顶点属于集合 S 当且仅当从源到该顶点的最短路径长度已知。

初始时， S 中仅含有源。设 u 是 G 的某一个顶点，把从源到 u 且中间只经过 S 中顶点的路称为从源到 u 的特殊路径，并用数组 $dist$ 记录当前每个顶点所对应的最短特殊路径长度。Dijkstra算法每次从 $V-S$ 中取出具有最短特殊路长度的顶点 u ，将 u 添加到 S 中，同时对数组 $dist$ 作必要的修改。一旦 S 包含了所有 V 中顶点， $dist$ 就记录了从源到所有其他顶点之间的最短路径长度。

4.5 单源最短路径

例如，对右图中的有向图，应用Dijkstra算法计算从源顶点1到其他顶点间最短路径的过程列在下页的表中。



4.5 单源最短路径

Dijkstra 算法的迭代过程:

迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	—	10	maxint	30	100
1	{1, 2}	2	10	60	30	100
2	{1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3, 5}	5	10	50	30	60

4.5 单源最短路径

2. 算法的正确性和计算复杂性

- (1) 贪心选择性质
- (2) 最优子结构性性质
- (3) 计算复杂性

对于具有 n 个顶点和 e 条边的带权有向图，如果用带权邻接矩阵表示这个图，那么Dijkstra算法的主循环体需要 $O(n)$ 时间。这个循环需要执行 $n-1$ 次，所以完成循环需要 $O(n^2)$ 时间。算法的其余部分所需要时间不超过 $O(n^2)$ 。

4.6 最小生成树

设 $G = (V, E)$ 是无向连通带权图，即一个网络。E中每条边 (v, w) 的权为 $c[v][w]$ 。如果 G 的子图 G' 是一棵包含 G 的所有顶点的树，则称 G' 为 G 的生成树。生成树上各边权的总和称为该生成树的耗费。在 G 的所有生成树中，耗费最小的生成树称为 G 的最小生成树。

网络的最小生成树在实际中有广泛应用。例如，在设计通信网络时，用图的顶点表示城市，用边 (v, w) 的权 $c[v][w]$ 表示建立城市 v 和城市 w 之间的通信线路所需的费用，则最小生成树就给出了建立通信网络的最经济的方案。

4.6 最小生成树

1. 最小生成树性质

用贪心算法设计策略可以设计出构造最小生成树的有效算法。本节介绍的构造最小生成树的Prim算法和Kruskal算法都可以看作是应用贪心算法设计策略的例子。尽管这2个算法做贪心选择的方式不同，它们都利用了下面的最小生成树性质：

设 $G=(V, E)$ 是连通带权图， U 是 V 的真子集。如果 $(u, v) \in E$ ，且 $u \in U$ ， $v \in V-U$ ，且在所有这样的边中， (u, v) 的权 $c[u][v]$ 最小，那么一定存在 G 的一棵最小生成树，它以 (u, v) 为其中一条边。这个性质有时也称为MST性质。

4.6 最小生成树

2. Prim算法

设 $G=(V, E)$ 是连通带权图, $V=\{1, 2, \dots, n\}$ 。

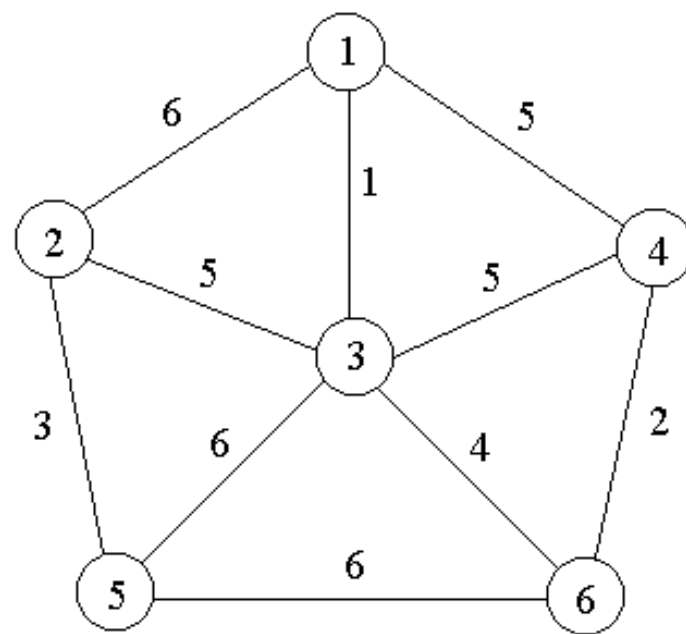
构造 G 的最小生成树的Prim算法的**基本思想**是：首先置 $S=\{1\}$ ，然后，只要 S 是 V 的真子集，就作如下的**贪心选择**：选取满足条件 $i \in S$, $j \in V-S$, 且 $c[i][j]$ 最小的边，将顶点 j 添加到 S 中。这个过程一直进行到 $S=V$ 时为止。

在这个过程中选取到的所有边恰好构成 G 的一棵**最小生成树**。

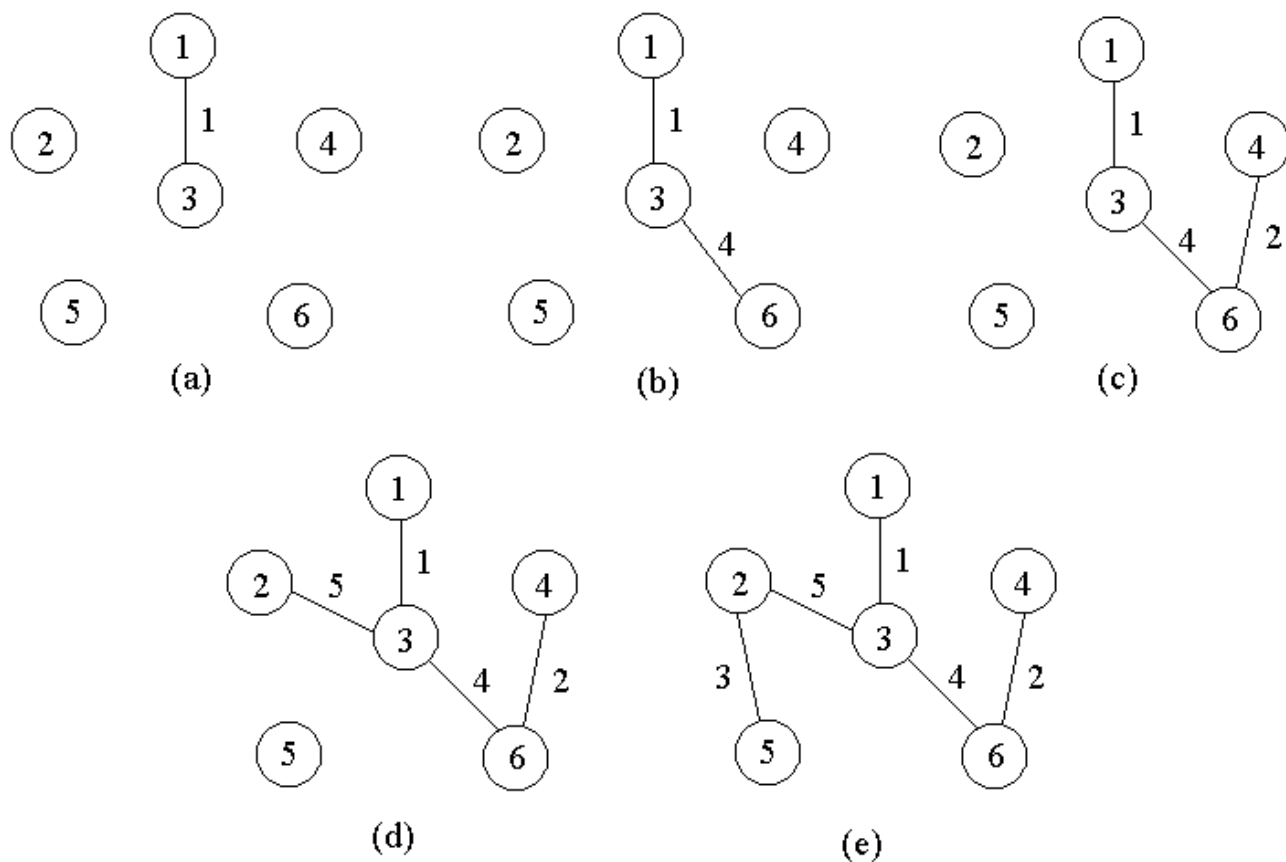
4.6 最小生成树

利用最小生成树性质和数学归纳法容易证明，上述算法中的**边集合T始终包含G的某棵最小生成树中的边**。因此，在算法结束时，T中的所有边构成G的一棵最小生成树。

例如，对于右图中的带权图，按**Prim算法**选取边的过程如下页图所示。



4.6 最小生成树



4.6 最小生成树

在上述Prim算法中，还应当考虑如何有效地找出满足条件 $i \in S, j \in V-S$ ，且权 $c[i][j]$ 最小的边 (i, j) 。实现这个目的的较简单的办法是设置2个数组closest和lowcost。

在Prim算法执行过程中，先找出 $V-S$ 中使lowcost值最小的顶点 j ，然后根据数组closest选取边 $(j, \text{closest}[j])$ ，最后将 j 添加到 S 中，并对closest和lowcost作必要的修改。

用这个办法实现的Prim算法所需的计算时间为 $O(n^2)$

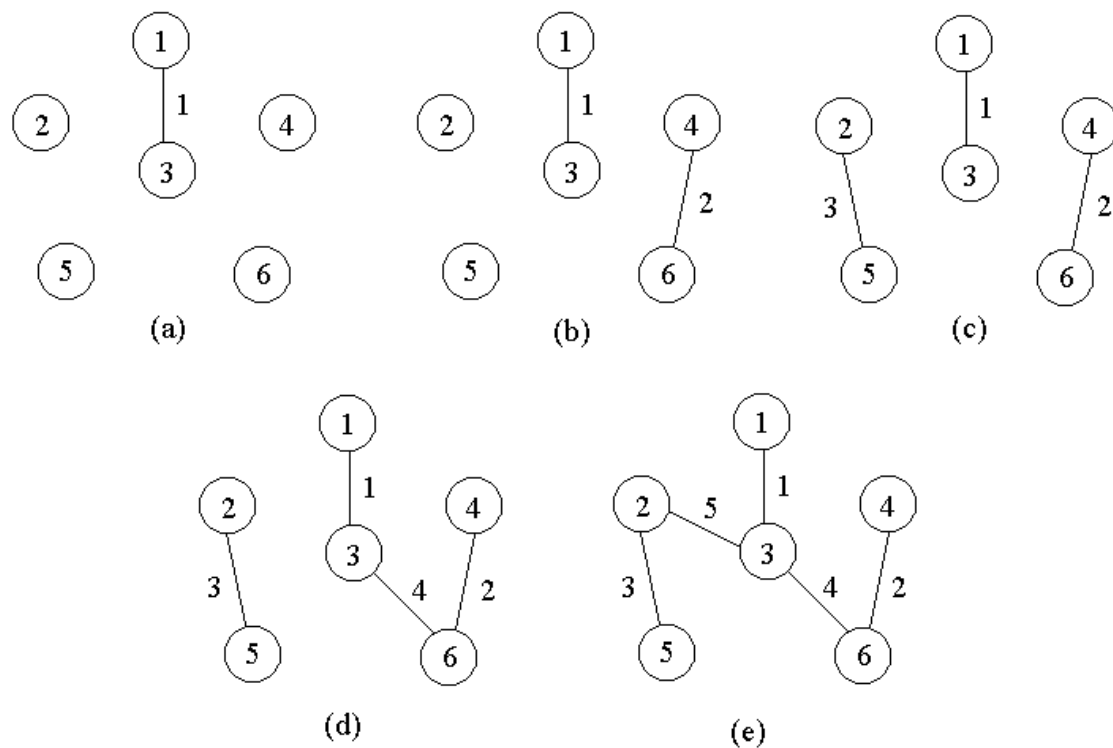
4.6 最小生成树

3. Kruskal 算法

Kruskal 算法构造 G 的最小生成树的**基本思想**是，首先将 G 的 n 个顶点看成 n 个孤立的连通分支。将所有的边按权从小到大排序。然后从第一条边开始，依边权递增的顺序查看每一条边，并按下述方法连接 2 个不同的连通分支：当查看到第 k 条边 (v, w) 时，如果端点 v 和 w 分别是当前 2 个不同的连通分支 T_1 和 T_2 中的顶点时，就用边 (v, w) 将 T_1 和 T_2 连接成一个连通分支，然后继续查看第 $k+1$ 条边；如果端点 v 和 w 在当前的同一个连通分支中，就直接再查看第 $k+1$ 条边。这个过程一直进行到只剩下一个连通分支时为止。

4.6 最小生成树

例如，对前面的连通带权图，按Kruskal算法顺序得到的最小生成树上的边如下图所示。



4.6 最小生成树

关于集合的一些基本运算可用于实现Kruskal算法。

按权的递增顺序查看等价于对优先队列执行removeMin运算。可以用堆实现这个优先队列。

对一个由连通分支组成的集合不断进行修改，需要用到抽象数据类型并查集UnionFind所支持的基本运算。

当图的边数为 e 时，Kruskal算法所需的计算时间是 $O(e \log e)$ 。当 $e = \Omega(n^2)$ 时，Kruskal算法比Prim算法差，但当 $e = o(n^2)$ 时，Kruskal算法却比Prim算法好得多。

4.7 多机调度问题

多机调度问题要求给出一种作业调度方案，使所给的 n 个作业在尽可能短的时间内由 m 台机器加工处理完成。

约定，每个作业均可在任何一台机器上加工处理，但未完工前不允许中断处理。作业不能拆分成更小的子作业。

这个问题是**NP完全问题**，到目前为止还没有有效的解法。对于这一类问题，用**贪心选择策略**有时可以设计出较好的近似算法。

4.7 多机调度问题

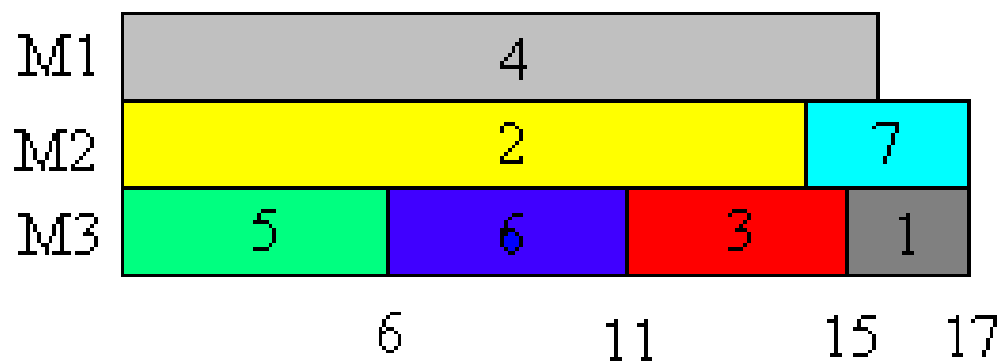
采用**最长处理时间作业优先**的贪心选择策略可以设计出解多机调度问题的较好的近似算法。

按此策略，当 $n \leq m$ 时，只要将机器 i 的 $[0, t_i]$ 时间区间分配给作业 i 即可，算法只需要 **$O(1)$** 时间。

当 $n > m$ 时，首先将 n 个作业依其所需的处理时间从大到小排序。然后依此顺序将作业分配给空闲的处理机。算法所需的计算时间为 **$O(n \log n)$** 。

4.7 多机调度问题

例如，设7个独立作业{1, 2, 3, 4, 5, 6, 7}由3台机器M1, M2和M3加工处理。各作业所需的处理时间分别为{2, 14, 4, 16, 6, 5, 3}。按算法greedy产生的作业调度如下图所示，所需的加工时间为17。



4.8 贪心算法的理论基础

借助于**拟阵**工具，可建立关于贪心算法的较一般的理论。这个理论对**确定何时使用贪心算法**可以得到问题的整体最优解十分有用。

1. 拟阵

拟阵 M 定义为满足下面3个条件的有序对 (S, I) ：

- (1) S 是非空有限集。
- (2) I 是 S 的一类具有遗传性质的独立子集族，即若 $B \in I$ ，则 B 是 S 的独立子集，且 B 的任意子集也都是 S 的独立子集。空集 \emptyset 必为 I 的成员。
- (3) I 满足交换性质，即若 $A \in I, B \in I$ 且 $|A| < |B|$ ，则存在某一元素 $x \in B - A$ ，使得 $A \cup \{x\} \in I$ 。

4.8 贪心算法的理论基础

例如，设 S 是一给定矩阵中行向量的集合， I 是 S 的线性独立子集族，则由线性空间理论容易证明 (S, I) 是一拟阵。拟阵的另一个例子是无向图 $G=(V, E)$ 的图拟阵 $M_G=(S_G, I_G)$ 。

给定拟阵 $M=(S, I)$ ，对于 I 中的独立子集 $A \in I$ ，若 S 有一元素 $x \notin A$ ，使得将 x 加入 A 后仍保持独立性，即 $A \cup \{x\} \in I$ ，则称 x 为 A 的可扩展元素。

当拟阵 M 中的独立子集 A 没有可扩展元素时，称 A 为极大独立子集。

4.8 贪心算法的理论基础

下面的关于极大独立子集的性质是很有用的。

定理4.1: 拟阵M中所有极大独立子集大小相同。

这个定理可以用反证法证明。

若对拟阵 $M=(S, I)$ 中的 S 指定权函数 W , 使得对于任意 $x \in S$, 有 $W(x) > 0$, 则称拟阵 M 为带权拟阵。依此权函数, S 的任一子集 A 的权定义为 $W(A) = \sum_{x \in A} W(x)$ 。

2. 关于带权拟阵的贪心算法

许多可以用贪心算法求解的问题可以表示为求带权拟阵的极大权独立子集问题。

4.8 贪心算法的理论基础

给定带权拟阵 $M=(S, I)$ ，确定 S 的独立子集 $A \in I$ 使得 $W(A)$ 达到最大。这种使 $W(A)$ 最大的独立子集 A 称为拟阵 M 的**最优子集**。由于 S 中任一元素 x 的权 $W(x)$ 是正的，因此，**最优子集也一定是极大独立子集**。

例如，在最小生成树问题可以表示为确定带权拟阵 M_G 的最优子集问题。求带权拟阵的最优子集 A 的算法可用于解最小生成树问题。

下面给出求**带权拟阵最优子集**的贪心算法。该算法以具有正权函数 W 的带权拟阵 $M=(S, I)$ 作为输入，经计算后输出 M 的最优子集 A 。

4.8 贪心算法的理论基础

- Set greedy (M, W)
- $\{A=\emptyset;$
- 将S中元素依权值W（大者优先）组成优先队列；
- while ($S\neq\emptyset$) {
- S.removeMax(x);
- if ($A\cup\{x\}\in I$) $A=A\cup\{x\}$;
- }
- return A
- }

4.8 贪心算法的理论基础

算法greedy的计算时间复杂性为 $O(n \log n + nf(n))$ 。

引理4.2 (拟阵的贪心选择性质)

设 $M=(S, I)$ 是具有权函数 W 的带权拟阵，且 S 中元素依权值从大到小排列。又设 $x \in S$ 是 S 中第一个使得 $\{x\}$ 是独立子集的元素，则存在 S 的最优子集 A 使得 $x \in A$ 。

算法greedy在以贪心选择构造最优子集 A 时，首次选入集合 A 中的元素 x 是单元素独立集中具有最大权的元素。此时可能已经舍弃了 S 中部分元素。可以证明这些被舍弃的元素不可能用于构造最优子集。

4.8 贪心算法的理论基础

引理4.3: 设 $M=(S, I)$ 是拟阵。若 S 中元素 x 不是空集的可扩展元素, 则 x 也不可能是 S 中任一独立子集 A 的可扩展元素。

引理4.4 (拟阵的最优子结构性质)

设 x 是求带权拟阵 $M=(S, I)$ 的最优子集的贪心算法greedy所选择的 S 中的第一个元素。那么, 原问题可简化为求带权拟阵 $M'=(S', I')$ 的**最优子集**问题, 其中:

$$S' = \{y \mid y \in S \text{ 且 } \{x, y\} \in I\}$$

$$I' = \{B \mid B \subseteq S - \{x\} \text{ 且 } B \cup \{x\} \in I\}$$

M' 的权函数是 M 的权函数在 S' 上的限制(称 M' 为 M 关于元素 x 的**收缩**)。

4.8 贪心算法的理论基础

定理4.5 (带权拟阵贪心算法的正确性)

设 $M = (S, I)$ 是具有权函数 W 的带权拟阵，算法greedy返回 M 的最优子集。

3. 任务时间表问题

给定一个单位时间任务的有限集 S 。关于 S 的一个时间表用于描述 S 中单位时间任务的执行次序。时间表中第1个任务从时间0开始执行直至时间1结束，第2个任务从时间1开始执行至时间2结束，...，第 n 个任务从时间 $n-1$ 开始执行直至时间 n 结束。

4.8 贪心算法的理论基础

具有**截止时间**和**误时惩罚**的单位时间任务时间表问题可描述如下。

- (1) n 个单位时间任务的集合 $S = \{1, 2, \dots, n\}$;
- (2) 任务 i 的截止时间 d_i , $1 \leq i \leq n$, $1 \leq d_i \leq n$, 即要求任务 i 在时间 d_i 之前结束;
- (3) 任务 i 的误时惩罚 w_i , $1 \leq i \leq n$, 即任务 i 未在时间 d_i 之前结束将招致的 w_i 惩罚; 若按时完成则无惩罚。

任务时间表问题要求确定 S 的一个时间表 (最优时间表) 使得总误时惩罚达到最小。

4.8 贪心算法的理论基础

这个问题看上去很复杂，然而借助于**拟阵**，可以用**带权拟阵的贪心算法**有效求解。

对于一个给定的S的时间表，在截止时间之前完成的任务称为**及时任务**，在截止时间之后完成的任务称为**误时任务**。

S的任一时间表可以调整成**及时优先形式**，即其中所有及时任务先于误时任务，而不影响原时间表中各任务的及时或误时性质。

类似地，还可将S的任一时间表调整成为**规范形式**，其中及时任务先于误时任务，且及时任务依其截止时间的非减序排列。

4.8 贪心算法的理论基础

首先可将时间表调整为及时优先形式，然后再进一步调整及时任务的次序。

任务时间表问题等价于确定最优时间表中及时任务子集A的问题。一旦确定了及时任务子集A，将A中各项任务依其截止时间的非减序列出，然后再以任意次序列出误时任务，即S-A中各任务，由此产生S的一个规范的最优时间表。

对时间 $t=1, 2, \dots, n$ ，设 $N_t(A)$ 是任务子集A中所有截止时间是 t 或更早的任务数。考察任务子集A的独立性。

4.8 贪心算法的理论基础

引理4.6: 对于S的任一任务子集A, 下面的各命题是等价的。

- (1) 任务子集A是独立子集。
- (2) 对于 $t=1, 2, \dots, n$, $N_t(A) \leq t$ 。
- (3) 若A中任务依其截止时间非减序排列, 则A中所有任务都是及时的。

任务时间表问题要求使总误时惩罚达到最小, 这等价于使任务时间表中的及时任务的惩罚值之和达到最大。下面的**定理**表明可用带权拟阵的贪心算法解任务时间表问题。

4.8 贪心算法的理论基础

定理4.7: 设 S 是带有截止时间的单位时间任务集, I 是 S 的所有独立任务子集构成的集合。则有序对 (S, I) 是拟阵。

由**定理4.5**可知, 用带权拟阵的贪心算法可以求得最大权(惩罚)独立任务子集 A , 以 A 作为最优时间表中的及时任务子集, 容易构造最优时间表。

任务时间表问题的贪心算法的**计算时间复杂性**是 $O(n \log n + nf(n))$ 。其中 $f(n)$ 是用于检测任务子集 A 的独立性所需的时间。用引理4.6中性质(2)容易设计一个 $O(n)$ 时间算法来检测任务子集的独立性。因此, 整个算法的**计算时间**为 $O(n^2)$ 。具体算法greedyJob可描述如P130。

4.8 贪心算法的理论基础

用抽象数据类型并查集UnionFind可对上述算法作进一步改进。如果不计预处理的时间，改进后的算法fasterJob所需的**计算时间**为 $O(n \log^* n)$ 。

第5章 回溯法

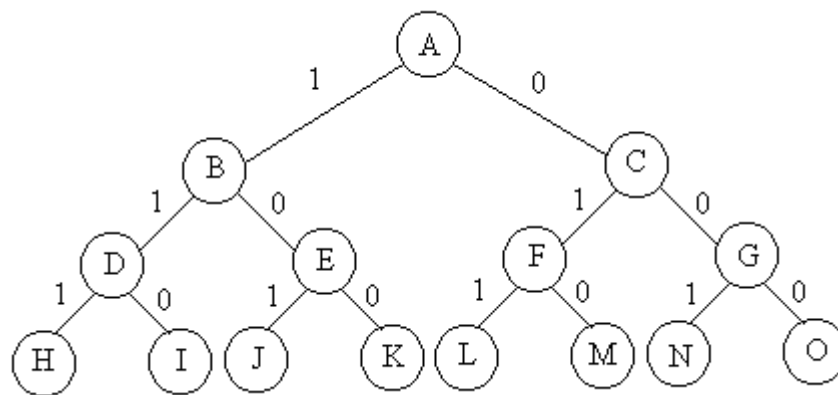
回溯法

- 有许多问题，当需要找出它的解集或者要求回答什么解是满足某些约束条件的最佳解时，往往要使用回溯法。
- 回溯法的基本做法是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法。这种方法适用于解一些组合数相当大的问题。
- 回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。算法搜索至解空间树的任意一点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

问题的解空间

- 问题的解向量：回溯法希望一个问题的解能够表示成一个 n 元式 (x_1, x_2, \dots, x_n) 的形式。
- 显约束：对分量 x_i 的取值限定。
- 隐约束：为满足问题的解而对不同分量之间施加的约束。
- 解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。



生成问题状态的基本方法

- 扩展结点:一个正在产生儿子的结点称为扩展结点
- 活结点:一个自身已生成但其儿子还没有全部生成的节点称做活结点
- 死结点:一个所有儿子已经产生的结点称做死结点
- 深度优先的问题状态生成法: 如果对一个扩展结点R, 一旦产生了它的一个儿子C, 就把C当做新的扩展结点。在完成对子树C (以C为根的子树) 的穷尽搜索之后, 将R重新变成扩展结点, 继续生成R的下一个儿子 (如果存在)
- 宽度优先的问题状态生成法: 在一个扩展结点变成死结点之前, 它一直是扩展结点
- 回溯法: 为了避免生成那些不可能产生最佳解的问题状态, 要不断地利用限界函数(bounding function)来处死那些实际上不可能产生所需解的活结点, 以减少问题的计算量。
具有限界函数的深度优先生成法称为回溯法

回溯法的基本思想

- (1) 针对所给问题，定义问题的解空间；
- (2) 确定易于搜索的解空间结构；
- (3) 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

常用剪枝函数：

用约束函数在扩展结点处剪去不满足约束的子树；
用限界函数剪去得不到最优解的子树。

用回溯法解题的一个显著特征是在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根结点到当前扩展结点的路径。如果解空间树中从根结点到叶结点的最长路径的长度为 $h(n)$ ，则回溯法所需的计算空间通常为 $O(h(n))$ 。而显式地存储整个解空间则需要 $O(2^{h(n)})$ 或 $O(h(n)!)$ 内存空间。

递归回溯

回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

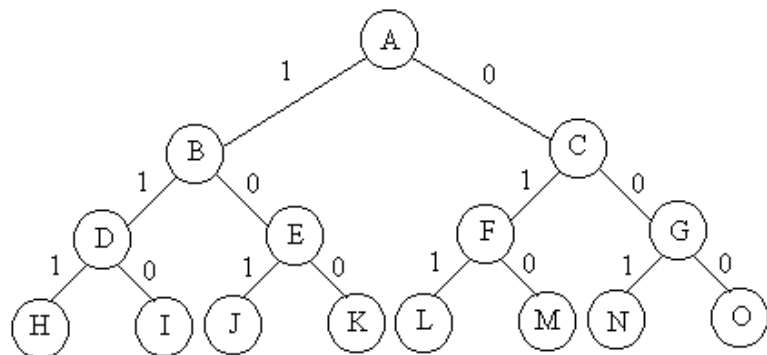
```
void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=f(n,t);i<=g(n,t);i++) {
            x[t]=h(i);
            if (constraint(t)&&bound(t)) backtrack(t+1);
        }
}
```

迭代回溯

采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

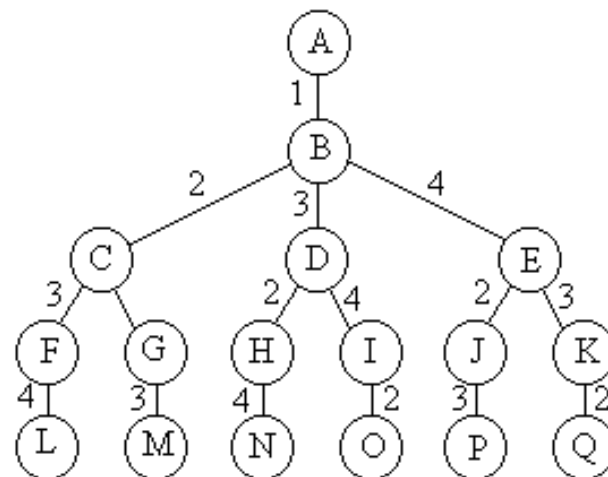
```
void iterativeBacktrack ()  
{  
    int t=1;  
    while (t>0) {  
        if (f(n,t)<=g(n,t))  
            for (int i=f(n,t);i<=g(n,t);i++) {  
                x[t]=h(i);  
                if (constraint(t)&&bound(t)) {  
                    if (solution(t)) output(x);  
                    else t++;  
                }  
            }  
        else t--;  
    }  
}
```

子集树与排列树



遍历子集树需 $O(2^n)$ 计算时间

```
void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=0;i<=1;i++) {
            x[t]=i;
            if (legal(t)) backtrack(t+1);
        }
}
```



遍历排列树需要 $O(n!)$ 计算时间

```
void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=t;i<=n;i++) {
            swap(x[t], x[i]);
            if (legal(t)) backtrack(t+1);
            swap(x[t], x[i]);
        }
}
```


装载问题

有一批共 n 个集装箱要装上2艘载重量分别为 c_1 和 c_2 的轮船，其中集装箱 i 的重量为 w_i ，且 $\sum_{i=1}^n w_i \leq c_1 + c_2$

装载问题要求确定是否有一个合理的装载方案可将这个集装箱装上这2艘轮船。如果有，找出一种装载方案。

容易证明，如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。

(1)首先将第一艘轮船尽可能装满；

(2)将剩余的集装箱装上第二艘轮船。

将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近。由此可知，装载问题等价于以下特殊的0-1背包问题。

$$\begin{aligned} \max \quad & \sum_{i=1}^n w_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^n w_i x_i \leq c_1 \\ & x_i \in \{0,1\}, 1 \leq i \leq n \end{aligned}$$

用回溯法设计解装载问题的 $O(2^n)$ 计算时间算法。在某些情况下该算法优于动态规划算法。

- 解空间：子集树
- 可行性约束函数(选择当前元素): $\sum_{i=1}^n w_i x_i \leq c_1$
- 上界函数(不选择当前元素):

当前载重量 cw +剩余集装箱的重量 $r \leq$ 当前最优载重量 $bestw$

```
private static void backtrack (int i)
```

```
{// 搜索第i层结点
```

```
    if (i > n) // 到达叶结点
```

```
        更新最优解bestx,bestw;return;
```

```
    r -= w[i];
```

```
    if (cw + w[i] <= c) {// 搜索左子树
```

```
        x[i] = 1;
```

```
        cw += w[i];
```

```
        backtrack(i + 1);
```

```
        cw -= w[i];    }
```

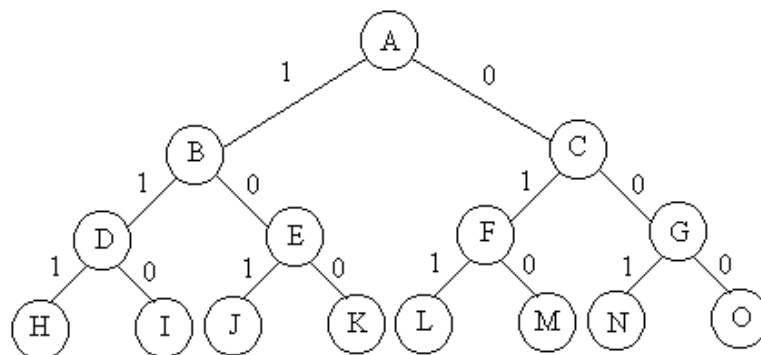
```
    if (cw + r > bestw) {
```

```
        x[i] = 0; // 搜索右子树
```

```
        backtrack(i + 1);    }
```

```
    r += w[i];
```

```
}
```



给定 n 个作业的集合 $\{J_1, J_2, \dots, J_n\}$ 。每个作业必须先由机器1处理，然后由机器2处理。作业 J_i 需要机器 j 的处理时间为 t_{ji} 。对于一个确定的作业调度，设 F_{ji} 是作业 i 在机器 j 上完成处理的时间。所有作业在机器2上完成处理的时间和称为该作业调度的完成时间和。批处理作业调度问题要求对于给定的 n 个作业，制定最佳作业调度方案，使其完成时间和达到最小。

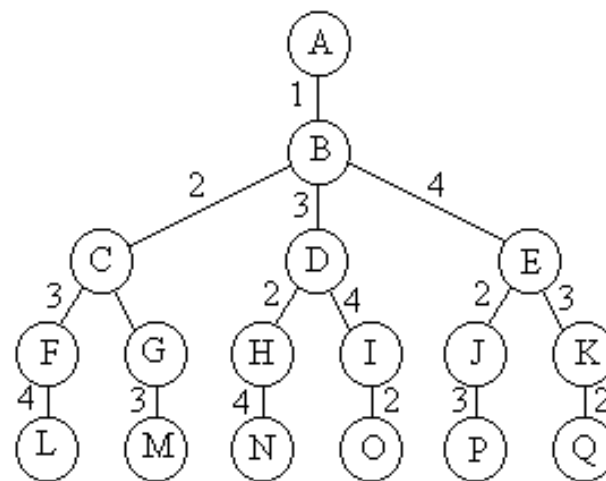
t_{ji}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3

这3个作业的6种可能的调度方案是1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2; 3,2,1; 它们所相应的完成时间和分别是19, 18, 20, 21, 19, 19。易见，最佳调度方案是1,3,2，其完成时间和为18。

•解空间：排列树

```
private static void backtrack(int i)
```

```
{
    if (i > n) {
        for (int j = 1; j <= n; j++) bestx[j] = x[j];
        bestf = f;
    } else
        for (int j = i; j <= n; j++) {
            f1+=m[x[j]][1];
            f2[i]=((f2[i-1]>f1)?f2[i-1]:f1)+m[x[j]][2];
            f+=f2[i];
            if (f < bestf) {
                MyMath.swap(x,i,j);
                backtrack(i+1);
                MyMath.swap(x,i,j);
            }
            f1-=m[x[j]][1];
            f-=f2[i];
        }
}
```



```
public class FlowShop
```

```
    static int n,    // 作业数
```

```
    f1,    // 机器1完成处理时间
```

```
    f,    // 完成时间和
```

```
    bestf; // 当前最优值
```

```
    static int [][] m; // 各作业所需的处理时间
```

```
    static int [] x;   // 当前作业调度
```

```
    static int [] bestx; // 当前最优作业调度
```

```
    static int [] f2;   // 机器2完成处理时间
```

符号三角形问题

下图是由14个“+”和14个“-”组成的符号三角形。2个同号下面都是“+”，2个异号下面都是“-”。

```

+  +  -  +  -  +  +
  +  -  -  -  -  +
    -  +  +  +  -
      -  +  +  -
        -  +  -
          -  -
            +

```

在一般情况下，符号三角形的第一行有 n 个符号。符号三角形问题要求对于给定的 n ，计算有多少个不同的符号三角形，使其所含的“+”和“-”的个数相同。

符号三角形问题

- 解向量：用 n 元组 $x[1:n]$ 表示符号三角形的第一行。
- 可行性约束函数：当前符号三角形所包含的“+”个数与“-”个数均不超过 $n*(n+1)/4$
- 无解的判断： $n*(n+1)/2$ 为奇数

```

+ + - + - + +
+ - - - - +

```

```
private static void backtrack (int t)
```

```
{
```

```
if ((co
```

```
if (t>r
```

```
else t
```

```
p[1]
```

```
cou
```

```
for (int j=2;j<=t;j++) {
```

```
    p[j][t-j+1]=p[j-1][t-j+1]^p[j-1][t-j+2];
```

```
    count+=p[j][t-j+1];
```

```
}
```

```
backtrack(t+1);
```

```
for (int j=2;j<=t;j++) count-=p[j][t-j+1];
```

```
count-=i;
```

```
}
```

```
}
```

复杂度分析

计算可行性约束需要 $O(n)$ 时间，在最坏情况下有 $O(2^n)$ 个结点需要计算可行性约束，故解符号三角形问题的回溯算法所需的计算时间为 $O(n2^n)$ 。

n后问题

在 $n \times n$ 格的棋盘上放置彼此不受攻击的 n 个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。 n 后问题等价于在 $n \times n$ 格的棋盘上放置 n 个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。

1				Q				
2						Q		
3								Q
4		Q						
5							Q	
6	Q							
7			Q					
8					Q			
	1	2	3	4	5	6	7	8

n后问题

- 解向量: (x_1, x_2, \dots, x_n)

- 显约束: $x_i=1, 2, \dots, n$

- 隐约束:

- 1)不同列: $x_i \neq x_j$

- 2)不处于同一正、反对角线: $|i-j| \neq |x_i - x_j|$

```
private static boolean place (int k)
```

```
{
```

```
    for (int j=1;j<k;j++)
```

```
        if ((Math.abs(k-j)==Math.abs(x[j]-x[k]))||(x[j]==x[k])) return false;
```

```
    return true;
```

```
}
```

```
private static void backtrack (int t)
```

```
{
```

```
    if (t>n) sum++;
```

```
    else
```

```
        for (int i=1;i<=n;i++) {
```

```
            x[t]=i;
```

```
            if (place(t)) backtrack(t+1);
```

```
        }
```

0-1背包问题

- 解空间：子集树
- 可行性约束函数： $\sum_{i=1}^n w_i x_i \leq c_1$
- 上界函数：

```
private static double bound(int i)
```

```
{// 计算上界
```

```
    double cleft = c - cw; // 剩余容量
```

```
    double bound = cp;
```

```
    // 以物品单位重量价值递减序装入物品
```

```
    while (i <= n && w[i] <= cleft)
```

```
    {
```

```
        cleft -= w[i];
```

```
        bound += p[i];
```

```
        i++;
```

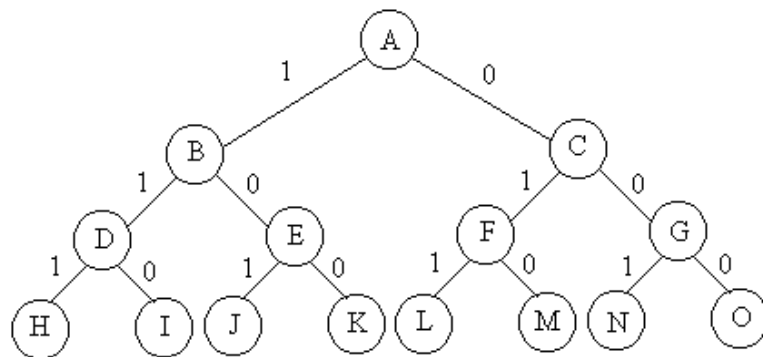
```
    }
```

```
    // 装满背包
```

```
    if (i <= n)    bound += p[i] / w[i] * cleft;
```

```
    return bound;
```

```
}
```



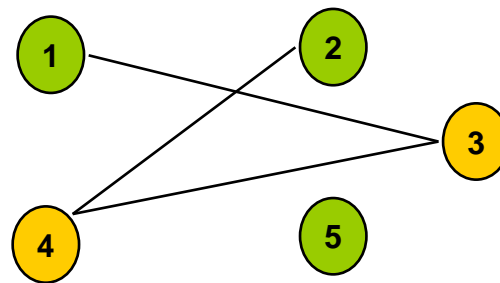
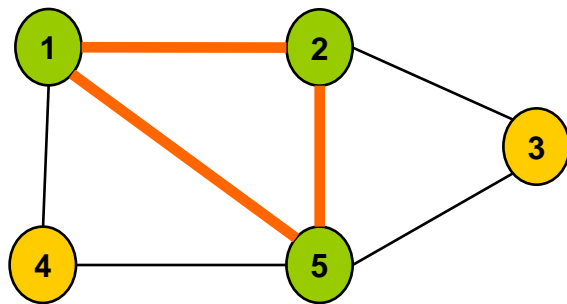
最大团问题

给定无向图 $G=(V, E)$ 。如果 $U \subseteq V$ ，且对任意 $u, v \in U$ 有 $(u, v) \in E$ ，则称 U 是 G 的完全子图。 G 的完全子图 U 是 G 的团当且仅当 U 不包含在 G 的更大的完全子图中。 G 的最大团是指 G 中所含顶点数最多的团。

如果 $U \subseteq V$ 且对任意 $u, v \in U$ 有 $(u, v) \notin E$ ，则称 U 是 G 的空子图。 G 的空子图 U 是 G 的独立集当且仅当 U 不包含在 G 的更大的空子图中。 G 的最大独立集是 G 中所含顶点数最多的独立集。

对于任一无向图 $G=(V, E)$ 其补图 $\bar{G}=(V, E_1)$ 定义为： $V_1=V$ ，且 $(u, v) \in E_1$ 当且仅当 $(u, v) \notin E$ 。

U 是 G 的最大团当且仅当 U 是 \bar{G} 的最大独立集。



- 解空间：子集树
- 可行性约束函数：顶点*i*到已选入的顶点集中每一个顶点都有边相连。
- 上界函数：有足够多的可选择顶点使得算法有可能在右子树中找到更大的团。

```
private static void backtrack(int i)
```

```
{  
    if (i > n) { // 到达叶结点  
        for (int j = 1; j <= n; j++) bestx[j] = x[j];  
        bestn = cn; return;    }  
    // 检
```

```
    boo
```

```
    for (
```

```
        if
```

复杂度分析

最大团问题的回溯算法**backtrack**所需的计算时间显然为 $O(n2^n)$ 。

```
    if (ok) { // 进入左子树
```

```
        x[i] = 1; cn++;
```

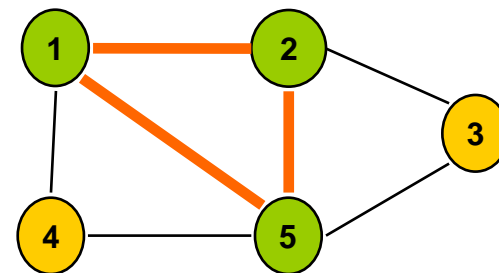
```
        backtrack(i + 1);
```

```
        cn--;
```

```
    if (cn + n - i > bestn) { // 进入右子树
```

```
        x[i] = 0;
```

```
        backtrack(i + 1);    } }
```

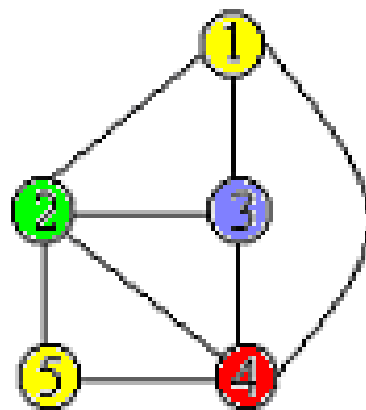
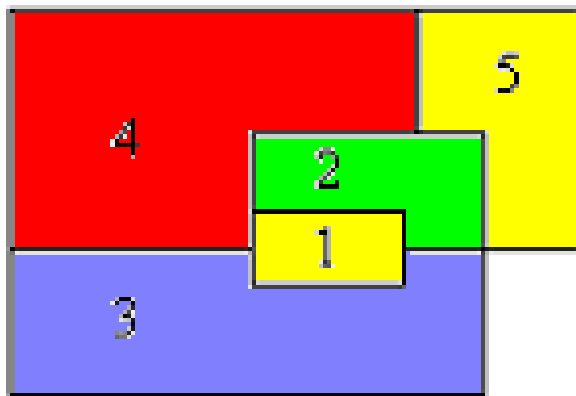


进一步改进算法的建议

- 选择合适的搜索顺序，可以使得上界函数更有效的发挥作用。例如在搜索之前可以将顶点按度从小到大排序。这在某种意义上相当于给回溯法加入了启发性。
- 定义 $S_i = \{v_i, v_{i+1}, \dots, v_n\}$ ，依次求出 S_n, S_{n-1}, \dots, S_1 的解。从而得到一个更精确的上界函数，若 $cn + S_i \leq \max$ 则剪枝。同时注意到：从 S_{i+1} 到 S_i ，如果找到一个更大的团，那么 v_i 必然属于找到的团，此时有 $S_i = S_{i+1} + 1$ ，否则 $S_i = S_{i+1}$ 。因此只要 \max 的值被更新过，就可以确定已经找到最大值，不必再往下搜索了。

图的 m 着色问题

给定无向连通图 G 和 m 种不同的颜色。用这些颜色为图 G 的各顶点着色，每个顶点着一种颜色。是否有一种着色法使 G 中每条边的2个顶点着不同颜色。这个问题是图的 m 可着色判定问题。若一个图最少需要 m 种颜色才能使图中每条边连接的2个顶点着不同颜色，则称这个数 m 为该图的色数。求一个图的色数 m 的问题称为图的 m 可着色优化问题。

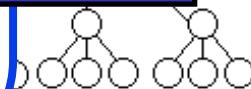


- 解向量: (x_1, x_2, \dots, x_n) 表示顶点 i 所着颜色 $x[i]$
- 可行性约束函数: 顶点 i 与已着色的相邻顶点颜色不重复。

思考: 图的 m 着色问题与图的最大团问题有何关系, 你能否利用这个关系改进最大团问题的上

界?

$$\sum_{i=0}^m m^i (mn) = nm(m^n - 1) / (m - 1) = O(nm^n)$$



```
private static boolean OK(int k)
```

```
{// 检查颜色可用性
```

```
    for (int j=1;j<=n;j++)
```

```
        if (a[k][j] && (x[j]==x[k])) return false;
```

```
    return true;
```

```
}
```

```
}
```

旅行售货员问题

- 解空间：排列树

```
private static void backtrack(int i)
```

```
{
    if (i == n) {
        if (a[x[n - 1]][x[n]] < Float.MAX_VALUE && a[x[n]][1] < Float.MAX_VALUE &&
            ...)) {
```

复杂度分析

算法**backtrack**在最坏情况下可能需要更新当前最优解 $O((n-1)!)$ 次，每次更新**bestx**需计算时间 $O(n)$ ，从而整个算法的计算时间复杂性为 $O(n!)$ 。

```
// 是否可进入x[j]子树?
```

```
if (a[x[i - 1]][x[j]] < Float.MAX_VALUE &&
```

```
(bestc == Float.MAX_VALUE || cc+a[x[i - 1]][x[j]]<bestc)) { // 搜索子树
```

```
    MyMath.swap(x, i, j);
```

```
    cc+=a[x[i - 1]][x[i]];
    backtrack(i + 1);
```

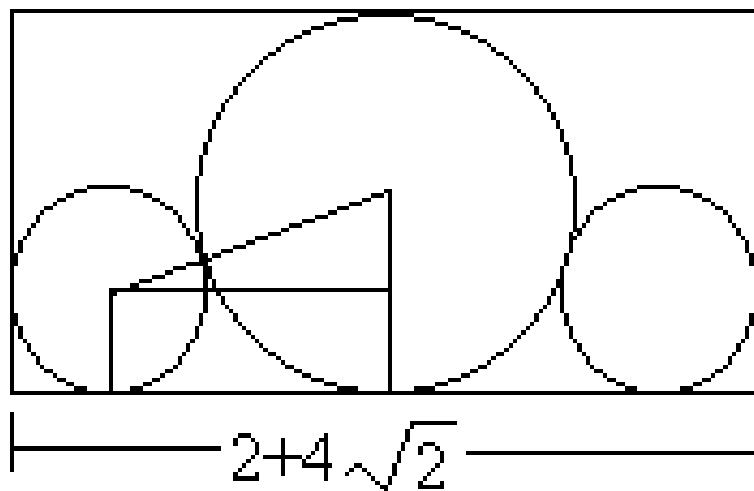
```
    cc-=a[x[i - 1]][x[i]];
    MyMath.swap(x, i, j);
```

```
}
```

```
}
```

圆排列问题

给定 n 个大小不等的圆 c_1, c_2, \dots, c_n ，现要将这 n 个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列。例如，当 $n=3$ ，且所给的3个圆的半径分别为1, 1, 2时，这3个圆的最小长度的圆排列如图所示。其最小长度为 $2+4\sqrt{2}$

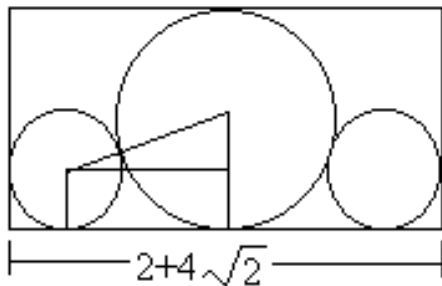



```
private static void backtrack(int t)
{
    if (t>n) compute();
    else
```

```
private static float center(int t)
{ // 计算当前所选择圆的圆心横坐标
  float temp=0;
  for (int j=1;j<t;j++) {
```

- 上述算法尚有许多改进的余地。例如，象 $1, 2, \dots, n-1, n$ 和 $n, n-1, \dots, 2, 1$ 这种互为镜像的排列具有相同的圆排列长度，只计算一个就够了，可减少约一半的计算量。另一方面，如果所给的 n 个圆中有 k 个圆有相同的半径，则这 k 个圆产生的 $k!$ 个完全相同的圆排列，只计算一个就够了。

```
    }
    MyMath.swap(r, t, j);
  }
}
```



```
    // 计算当前圆排列的长度
    float low=0,
          high=0;
    for (int i=1;i<=n;i++) {
        if (x[i]-r[i]<low) low=x[i]-r[i];
        if (x[i]+r[i]>high) high=x[i]+r[i];
    }
    if (high-low<min) min=high-low;
}
```


连续邮资问题

假设国家发行了 n 种不同面值的邮票，并且规定每张信封上最多只允许贴 m 张邮票。连续邮资问题要求对于给定的 n 和 m 的值，给出邮票面值的最佳设计，在1张信封上可贴出从邮资1开始，增量为1的最大连续邮资区间。

例如，当 $n=5$ 和 $m=4$ 时，面值为 $(1,3,11,15,32)$ 的5种邮票可以贴出邮资的最大连续邮资区间是1到70。

连续邮资问题

- 解向量：用 n 元组 $x[1:n]$ 表示 n 种不同的邮票面值，并约定它们从小到大排列。 $x[1]=1$ 是惟一的选择。
- 可行性约束函数：已选定 $x[1:i-1]$ ，最大连续邮资区间是 $[1:r]$ ，接下来 $x[i]$ 的可取值范围是 $[x[i-1]+1:r+1]$ 。

如何确定 r 的值？

计算 $X[1:i]$ 的最大连续邮资区间在本算法中被频繁使用到，因此势必要找到一个高效的方法。考虑到直接递归的求解复杂度太高，我们不妨尝试计算用不超过 m 张面值为 $x[1:i]$ 的邮票贴出邮资 k 所需的最少邮票数 $y[k]$ 。通过 $y[k]$ 可以很快推出 r 的值。事实上， $y[k]$ 可以通过递推在 $O(n)$ 时间内解决：

```
for (int j=0; j<= x[i-2]*(m-1);j++)  
    if (y[j]<m)  
        for (int k=1;k<=m-y[j];k++)  
            if (y[j]+k<y[j+x[i-1]*k]) y[j+x[i-1]*k]=y[j]+k;  
while (y[r]<maxint) r++;
```

回溯法效率分析

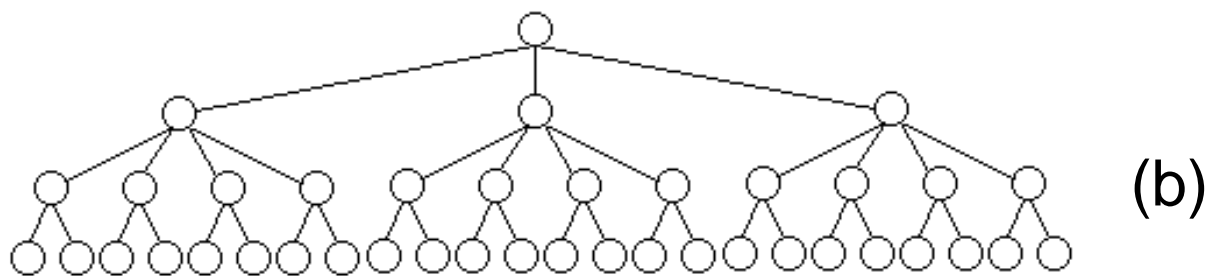
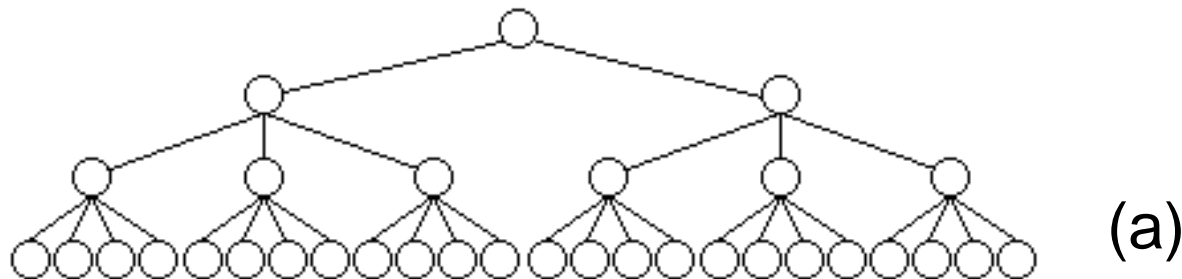
通过前面具体实例的讨论容易看出，回溯算法的效率在很大程度上依赖于以下因素：

- (1) 产生 $x[k]$ 的时间；
- (2) 满足显约束的 $x[k]$ 值的个数；
- (3) 计算约束函数**constraint**的时间；
- (4) 计算上界函数**bound**的时间；
- (5) 满足约束函数和上界函数约束的所有 $x[k]$ 的个数。

好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算量较大。因此，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷。

重排原理

对于许多问题而言，在搜索试探时选取 $x[i]$ 的值顺序是任意的。
在其他条件相当的前提下，让可取值最少的 $x[i]$ 优先。从图中关于同一问题的2棵不同解空间树，可以体会到这种策略的潜力。



图(a)中，从第1层剪去1棵子树，则从所有应当考虑的3元组中一次消去12个3元组。对于图(b)，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。

第六章 分支限界法

第六章 分支限界法

本章主要知识点

- 6.1 分支限界法的基本思想
- 6.2 单源最短路径问题
- 6.3 装载问题
- 6.4 布线问题
- 6.5 0—1背包问题
- 6.6 最大团问题
- 6.7 旅行售货员问题
- 6.8 电路板排列问题
- 6.9 批处理作业调度

6.1 分支限界法的基本思想

1. 分支限界法与回溯法的不同

- (1) 求解目标：回溯法的求解目标是找出解空间树中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出在某种意义下的最优解。
- (2) 搜索方式的不同：回溯法以深度优先的方式搜索解空间树，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树。

6.1 分支限界法的基本思想

2. 分支限界法基本思想

分支限界法常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树。

在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中。

此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。

6.1 分支限界法的基本思想

3. 常见的两种分支限界法

(1) 队列式(FIFO)分支限界法

按照队列先进先出(FIFO)原则选取下一个节点为扩展节点。

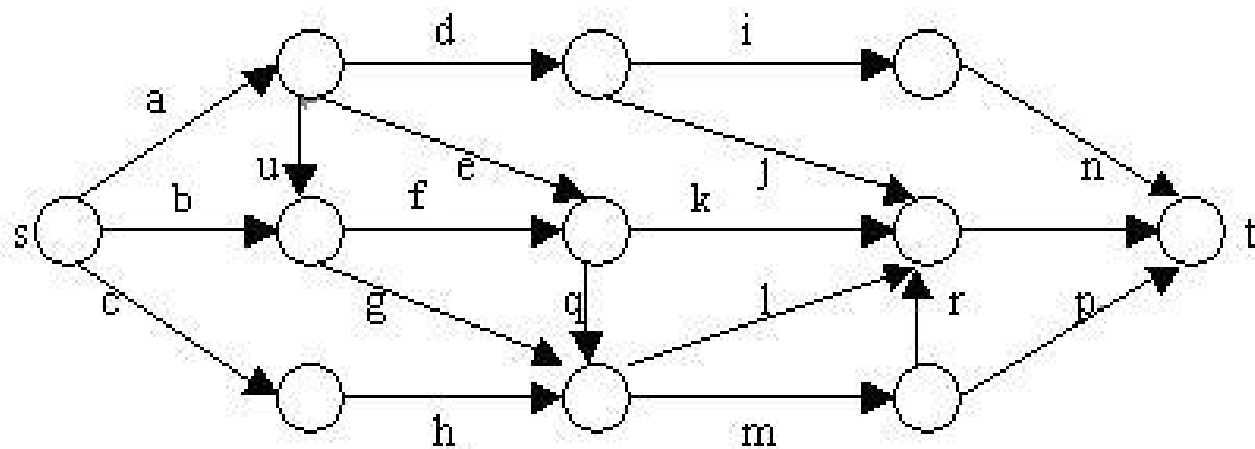
(2) 优先队列式分支限界法

按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点。

6.2 单源最短路径问题

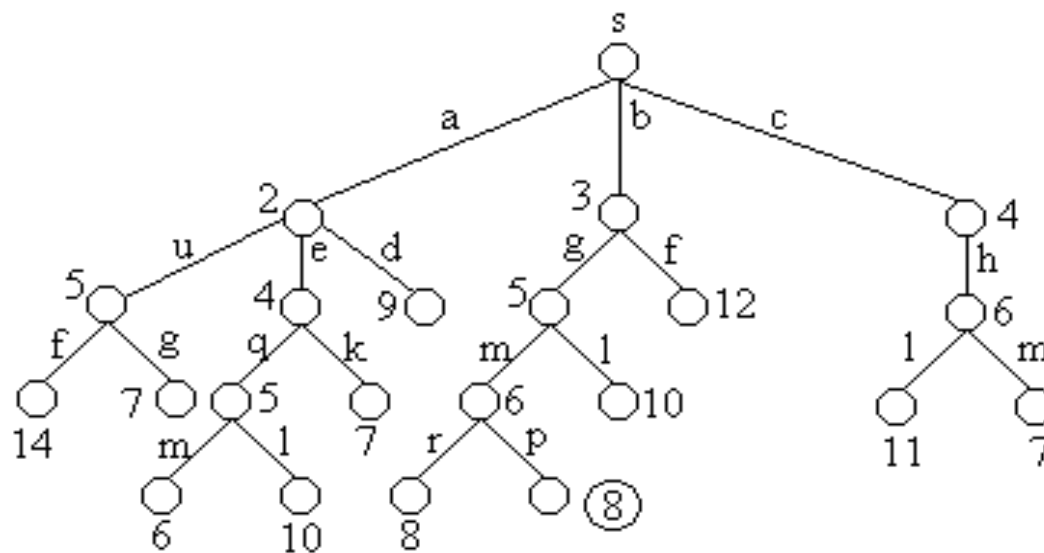
1. 问题描述

下面以一个例子来说明单源最短路径问题：在下图所给的有向图G中，每一边都有一个非负边权。要求图G的从源顶点s到目标顶点t之间的最短路径。



6.2 单源最短路径问题

下图是用优先队列式分支限界法解有向图G的单源最短路径问题产生的解空间树。其中，每一个结点旁边的数字表示该结点所对应的当前路长。



6.2 单源最短路径问题

2. 算法思想

解单源最短路径问题的优先队列式分支限界法用一极小堆来存储活结点表。其优先级是结点所对应的当前路长。

算法从图 G 的源顶点 s 和空优先队列开始。结点 s 被扩展后，它的儿子结点被依次插入堆中。此后，算法从堆中取出具有最小当前路长的结点作为当前扩展结点，并依次检查与当前扩展结点相邻的所有顶点。如果从当前扩展结点 i 到顶点 j 有边可达，且从源出发，途经顶点 i 再到顶点 j 的所相应的路径的长度小于当前最优路径长度，则将该顶点作为活结点插入到活结点优先队列中。这个结点的扩展过程一直继续到活结点优先队列为空时为止。

6.2 单源最短路径问题

3. 剪枝策略

在算法扩展结点的过程中，一旦发现一个结点的下界不小于当前找到的最短路长，则算法剪去以该结点为根的子树。

在算法中，利用结点间的控制关系进行剪枝。从源顶点 s 出发，2条不同路径到达图 G 的同一顶点。由于两条路径的路长不同，因此可以将路长长的路径所对应的树中的结点为根的子树剪去。

6.2 单源最短路径问题

```
while (true)
{
    // 搜索问题的解空间
    for (int j=1;j<=n;j++)
        if(a[enode.i][j] < Float.MAX_VALUE && enode.length+a[enode.i][j] < dist[j])
        {
            // 顶点i到顶点j可达, 且满足控制约束
            dist[j]=enode.length+a[enode.i][j];
            p[j]=enode.i;
            HeapNode node = new HeapNode(j,dist[j]);
            heap.put(node);    // 加入活结点优先队列
        }
    if (heap.isEmpty()) break;
    else enode = (HeapNode) heap.removeMin();
}
```

顶点i和j间有边, 且此
路径长小于原先从原点
到j的路径长

6.3 装载问题

1. 问题描述

有一批共个集装箱要装上2艘载重量分别为 C_1 和 C_2 的轮船，其中集

装箱 i 的重量为 W_i ，且
$$\sum_{i=1}^n w_i \leq c_1 + c_2$$

装载问题要求确定是否有一个合理的装载方案可将这个集装箱装上这2艘轮船。如果有，找出一种装载方案。

容易证明：如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。

- (1) 首先将第一艘轮船尽可能装满；
- (2) 将剩余的集装箱装上第二艘轮船。

6.3 装载问题

2. 队列式分支限界法

在算法的while循环中，首先检测当前扩展结点的左儿子结点是否为可行结点。如果是则将其加入到活结点队列中。然后将其右儿子结点加入到活结点队列中(右儿子结点一定是可行结点)。2个儿子结点都产生后，当前扩展结点被舍弃。

活结点队列中的队首元素被取出作为当前扩展结点，由于队列中每一层结点之后都有一个尾部标记-1，故在取队首元素时，活结点队列一定不空。当取出的元素是-1时，再判断当前队列是否为空。如果队列非空，则将尾部标记-1加入活结点队列，算法开始处理下一层的活结点。

6.3 装载问题

2. 队列式分支限界法

```
while (true)
{
    if (ew + w[i] <= c)    enqueue(ew + w[i], i);    // 检查左儿子结点
    enqueue(ew, i);        // 右儿子结点总是可行的
    ew = ((Integer) queue.remove()).intValue();    // 取下一扩展结点
    if (ew == -1)
    { if (queue.isEmpty()) return bestw;
      queue.put(new Integer(-1));                // 同层结点尾部标志
      ew = ((Integer) queue.remove()).intValue(); // 取下一扩展结点
      i++;                                        // 进入下一层    }    }
```

6.3 装载问题

3. 算法的改进

节点的左子树表示将此集装箱装上船，右子树表示不将此集装箱装上船。设 $bestw$ 是当前最优解； ew 是当前扩展结点所相应的重量； r 是剩余集装箱的重量。则当 $ew+r \leq bestw$ 时，可将其右子树剪去，因为此时若要船装最多集装箱，就应该把此箱装上船。

另外，为了确保右子树成功剪枝，应该在算法每一次进入左子树的时候更新 $bestw$ 的值。

6.3 装载问题

3. 算法的改进

// 检查左儿子结点

```
int wt = ew + w[i];
```

```
if (wt <= c)
```

```
{ // 可行结点
```

```
    if (wt > bestw) bestw = wt;
```

```
    // 加入活结点队列
```

```
    if (i < n)
```

```
        queue.put(new Integer(wt));
```

```
}
```

提前更新
bestw

// 检查右儿子结点

```
if (ew + r > bestw && i < n)
```

```
// 可能含最优解
```

```
    queue.put(new Integer(ew));
```

```
    ew=((Integer)queue.remove())
```

```
        .intValue();
```

```
// 取下一扩展结点
```

右儿子剪枝

6.3 装载问题

4. 构造最优解

为了在算法结束后能方便地构造出与最优值相应的最优解，算法必须存储相应子集树中从活结点到根结点的路径。为此目的，可在每个结点处设置指向其父结点的指针，并设置左、右儿子标志。

```
private static class QNode
```

```
{   QNode parent;      // 父结点  
    boolean leftChild; // 左儿子标志  
    int weight;        // 结点所相应的载重量
```

6.3 装载问题

找到最优值后，可以根据parent回溯到根节点，找到最优解。

// 构造当前最优解

```
for (int j = n; j > 0; j--)  
{  
    bestx[j] = (e.leftChild) ? 1 : 0;  
    e = e.parent;  
}
```

6.3 装载问题

5. 优先队列式分支限界法

解装载问题的优先队列式分支限界法用最大优先队列存储活结点表。活结点 x 在优先队列中的优先级定义为从根结点到结点 x 的路径所相应的载重量再加上剩余集装箱的重量之和。

优先队列中优先级最大的活结点成为下一个扩展结点。以结点 x 为根的子树中所有结点相应的路径的载重量不超过它的优先级。子集树中叶结点所相应的载重量与其优先级相同。

在优先队列式分支限界法中，一旦有一个叶结点成为当前扩展结点，则可以断言该叶结点所相应的解即为最优解。此时可终止算法。

6.4 布线问题

算法的思想

解此问题的队列式分支限界法从起始位置 a 开始将它作为第一个扩展结点。与该扩展结点相邻并且可达的方格成为可行结点被加入到活结点队列中，并且将这些方格标记为1，即从起始方格 a 到这些方格的距离为1。

接着，算法从活结点队列中取出队首结点作为下一个扩展结点，并将与当前扩展结点相邻且未标记过的方格标记为2，并存入活结点队列。这个过程一直继续到算法搜索到目标方格 b 或活结点队列为空时为止。即加入剪枝的广度优先搜索。

6.4 布线问题

```
Position [] offset = new Position [4];  
offset[0] = new Position(0, 1);    // 右  
offset[1] = new Position(1, 0);    // 下  
offset[2] = new Position(0, -1);   // 左  
offset[3] = new Position(-1, 0);   // 上
```

定义移动方向的
相对位移

```
for (int i = 0; i <= size + 1; i++)  
{  
    grid[0][i] = grid[size + 1][i] = 1; // 顶部和底部  
    grid[i][0] = grid[i][size + 1] = 1; // 左翼和右翼  
}
```

设置边界的围墙

6.4 布线问题

```
for (int i = 0; i < numOfNbrs; i++)  
{  
    nbr.row = here.row + offset[i].row;  
    nbr.col = here.col + offset[i].col;  
    if (grid[nbr.row][nbr.col] == 0)  
    { // 该方格未标记  
        grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;  
        if ((nbr.row == finish.row) && (nbr.col == finish.col)) break;  
        q.put(new Position(nbr.row, nbr.col));  
    }  
}
```

找到目标位置后，可以通过回溯方法找到这条最短路径。

6.5 0-1背包问题

• 算法的思想

首先，要对输入数据进行预处理，将各物品依其单位重量价值从大到小进行排列。

在下面描述的优先队列分支限界法中，节点的优先级由已装袋的物品价值加上剩下的最大单位重量价值的物品装满剩余容量的价值和。

算法首先检查当前扩展结点的左儿子结点的可行性。如果该左儿子结点是可行结点，则将它加入到子集树和活结点优先队列中。当前扩展结点的右儿子结点一定是可行结点，仅当右儿子结点满足上界约束时才将它加入子集树和活结点优先队列。当扩展到叶节点时为问题的最优值。

6.5 0-1背包问题

上界函数

```
while (i <= n && w[i] <= cleft) // n表示物品总数, cleft为剩余空间
{
    cleft -= w[i];                //w[i]表示i所占空间
    b += p[i];                    //p[i]表示i的价值
    i++;
}
if (i <= n) b += p[i] / w[i] * cleft; // 装填剩余容量装满背包
return b;                        //b为上界函数
```

6.5 0-1背包问题

```
while (i != n + 1)
{
    // 非叶结点
    double wt = cw + w[i];
    if (wt <= c)
    {
        // 左儿子结点为可行结点
        if (cp + p[i] > bestp)    bestp = cp + p[i];
        addLiveNode(up, cp + p[i], cw + w[i], i + 1, enode, true);
    }
    up = bound(i + 1);
    if (up >= bestp) //检查右儿子节点
        addLiveNode(up, cp, cw, i + 1, enode, false);
    // 取下一个扩展节点 (略)
}
```

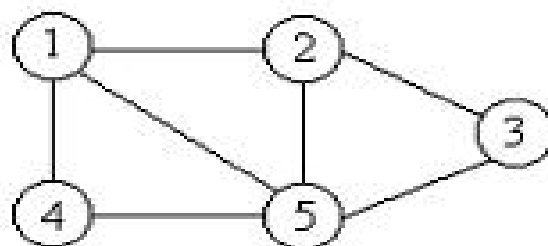
分支限界搜索
过程

6.6 最大团问题

1. 问题描述

给定无向图 $G=(V, E)$ 。如果 $U \subseteq V$ ，且对任意 $u, v \in U$ 有 $(u, v) \in E$ ，则称 U 是 G 的完全子图。 G 的完全子图 U 是 G 的团当且仅当 U 不包含在 G 的更大的完全子图中。 G 的最大团是指 G 中所含顶点数最多的团。

下图 G 中，子集 $\{1, 2\}$ 是 G 的大小为2的完全子图。这个完全子图不是团，因为它被 G 的更大的完全子图 $\{1, 2, 5\}$ 包含。 $\{1, 2, 5\}$ 是 G 的最大团。 $\{1, 4, 5\}$ 和 $\{2, 3, 5\}$ 也是 G 的最大团。



6.6 最大团问题

2. 上界函数

用变量`cliqueSize`表示与该结点相应的团的顶点数；`level`表示结点在子集空间树中所处的层次；用`cliqueSize + n - level + 1`作为顶点数上界`upperSize`的值。

在此优先队列式分支限界法中，`upperSize`实际上也是优先队列中元素的优先级。算法总是从活结点优先队列中抽取具有最大`upperSize`值的元素作为下一个扩展元素。

6.6 最大团问题

3. 算法思想

子集树的根结点是初始扩展结点，对于这个特殊的扩展结点，其cliqueSize的值为0。

算法在扩展内部结点时，首先考察其左儿子结点。在左儿子结点处，将顶点 i 加入到当前团中，并检查该顶点与当前团中其他顶点之间是否有边相连。当顶点 i 与当前团中所有顶点之间都有边相连，则相应的左儿子结点是可行结点，将它加入到子集树中并插入活结点优先队列，否则就不是可行结点。

接着继续考察当前扩展结点的右儿子结点。当 $upperSize > bestn$ 时，右子树中可能含有最优解，此时将右儿子结点加入到子集树中并插入到活结点优先队列中。

6.6 最大团问题

算法的while循环的终止条件是遇到子集树中的一个叶结点(即 $n+1$ 层结点)成为当前扩展结点。

对于子集树中的叶结点, 有 $\text{upperSize} = \text{cliqueSize}$ 。此时活结点优先队列中剩余结点的 upperSize 值均不超过当前扩展结点的 upperSize 值, 从而进一步搜索不可能得到更大的团, 此时算法已找到一个最优解。

6.7 旅行售货员问题

1. 问题描述

某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)。他要选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总的路程(或总旅费)最小。

路线是一个带权图。图中各边的费用(权)为正数。图的一条周游路线是包括 V 中的每个顶点在内的一条回路。周游路线的费用是这条路线上所有边的费用之和。

旅行售货员问题的解空间可以组织成一棵树，从树的根结点到任一叶结点的路径定义了图的一条周游路线。旅行售货员问题要在图 G 中找出费用最小的周游路线。

6.7 旅行售货员问题

2. 算法描述

算法开始时创建一个最小堆，用于表示活结点优先队列。堆中每个结点的子树费用的下界 $lcost$ 值是优先队列的优先级。接着算法计算出图中每个顶点的最小费用出边并用 $minout$ 记录。如果所给的有向图中某个顶点没有出边，则该图不可能有回路，算法即告结束。如果每个顶点都有出边，则根据计算出的 $minout$ 作算法初始化。

算法的 $while$ 循环体完成对排列树内部结点的扩展。对于当前扩展结点，算法分2种情况进行处理：

6.7 旅行售货员问题

1、首先考虑 $s=n-2$ 的情形，此时当前扩展结点是排列树中某个叶结点的父结点。如果该叶结点相应一条可行回路且费用小于当前最小费用，则将该叶结点插入到优先队列中，否则舍去该叶结点。

2、当 $s < n-2$ 时，算法依次产生当前扩展结点的所有儿子结点。由于当前扩展结点所相应的路径是 $x[0:s]$ ，其可行儿子结点是从剩余顶点 $x[s+1:n-1]$ 中选取的顶点 $x[i]$ ，且 $(x[s], x[i])$ 是所给有向图 G 中的一条边。对于当前扩展结点的每一个可行儿子结点，计算出其前缀 $(x[0:s], x[i])$ 的费用 cc 和相应的下界 $lcost$ 。当 $lcost < bestc$ 时，将这个可行儿子结点插入到活结点优先队列中。

6.7 旅行售货员问题

算法中while循环的终止条件是排列树的一个叶结点成为当前扩展结点。当 $s=n-1$ 时，已找到的回路前缀是 $x[0:n-1]$ ，它已包含图 G 的所有 n 个顶点。因此，当 $s=n-1$ 时，相应的扩展结点表示一个叶结点。此时该叶结点所相应的回路的费用等于 cc 和 $lcost$ 的值。剩余的活结点的 $lcost$ 值不小于已找到的回路的费用。它们都不可能导致费用更小的回路。因此已找到的叶结点所相应的回路是一个最小费用旅行售货员回路，算法可以结束。

算法结束时返回找到的最小费用，相应的最优解由数组 v 给出。

6.8 电路板排列问题

• 算法描述

算法开始时，将排列树的根结点置为当前扩展结点。在do-while循环体内算法依次从活结点优先队列中取出具有最小cd值的结点作为当前扩展结点，并加以扩展。

首先考虑 $s=n-1$ 的情形，当前扩展结点是排列树中的一个叶结点的父结点。 x 表示相应于该叶结点的电路板排列。计算出与 x 相应的密度并在必要时更新当前最优值和相应的当前最优解。

当 $s < n-1$ 时，算法依次产生当前扩展结点的所有儿子结点。对于当前扩展结点的每一个儿子结点 $node$ ，计算出其相应的密度 $node.cd$ 。当 $node.cd < bestd$ 时，将该儿子结点 N 插入到活结点优先队列中。

6.8 电路板排列问题

算法描述

```
do
    if (enode.s == n - 1)
        { // 仅一个儿子结点
            int ld = 0; // 最后一块电路板的密度
            for (int j = 1; j <= m; j++)
                ld += board [enode.x[n]][j];
            if (ld < bestd)
                { // 找到密度更小的电路板排列
                    x = enode.x;
                    bestd = Math.max(ld, enode.cd); }
        }
```

$S=n-1$ 的情况，计算出此时的密度和bestd进行比较。

6.8 电路板排列问题

算法描述

else

```
{// 产生当前扩展结点的所有儿子结点
```

```
for (int i = enode.s + 1; i <= n; i++) {
```

```
    HeapNode node = new HeapNode(0, new int [m + 1], 0,  
new int [n + 1]);
```

```
for (int j = 1; j <= m; j++)
```

```
    // 新插入的电路板
```

```
    node.now[j] = enode.now[j] + board [enode.x[i]][j];
```

6.8 电路板排列问题

算法描述

```
int ld = 0; // 新插入电路板的密度
for (int j = 1; j <= m; j++)
    if (node.now[j] > 0 && total[j] != node.now[j]) ld++;
node.cd = Math.max(ld, enode.cd);
if (node.cd < bestd)
    { // 可能产生更好的叶结点
        node.s = enode.s + 1;
        for (int j = 1; j <= n; j++) node.x[j] = enode.x[j];
        node.x[node.s] = enode.x[i];
        node.x[i] = enode.x[node.s];
        heap.put(node);    }    }    }
```

计算出每一个儿子结点的密度与bestd进行比较
大于bestd时加入队列

6.9 批处理作业问题

1. 问题的描述

给定 n 个作业的集合 $J = \{J_1, J_2, \dots, J_n\}$ 。每一个作业 J_i 都有2项任务要分别在2台机器上完成。每一个作业必须先由机器1处理，然后再由机器2处理。作业 J_i 需要机器 j 的处理时间为 t_{ji} ， $i=1, 2, \dots, n$ ； $j=1, 2$ 。对于一个确定的作业调度，设是 F_{ji} 是作业 i 在机器 j 上完成处理的时间。则所有作业在机器2上完成处理的时间和 $f = \sum_{i=1}^n F_{2i}$

称为该作业调度的完成时间和。批处理作业调度问题要求对于给定的 n 个作业，制定最佳作业调度方案，使其完成时间和达到最小。

6.9 批处理作业问题

2. 限界函数

在结点E处相应子树中叶结点完成时间和的下界是:

$$f \geq \sum_{i \in M} F_{2i} + \max\{S_1, S_2\}$$

注意到如果选择 P_k , 使 t_{1pk} 在 $k \geq r+1$ 时依非减序排列, S_1 则取得极小值。同理如果选择 P_k 使 t_{2pk} 依非减序排列, 则 S_2 取得极小值。

$$f \geq \sum_{i \in M} F_{2i} + \max\{\hat{S}_1, \hat{S}_2\}$$

这可以作为优先队列式分支限界法中的限界函数。

6.9 批处理作业问题

3. 算法描述

算法的while循环完成对排列树内部结点的有序扩展。在while循环体内算法依次从活结点优先队列中取出具有最小bb值（完成时间和下界）的结点作为当前扩展结点，并加以扩展。

首先考虑 $enode.s=n$ 的情形，当前扩展结点 $enode$ 是排列树中的叶结点。 $enode.sf2$ 是相应于该叶结点的完成时间和。当 $enode.sf2 < bestc$ 时更新当前最优值 $bestc$ 和相应的当前最优解 $bestx$ 。

当 $enode.s < n$ 时，算法依次产生当前扩展结点 $enode$ 的所有儿子结点。对于当前扩展结点的每一个儿子结点 $node$ ，计算出其相应的完成时间和的下界 bb 。当 $bb < bestc$ 时，将该儿子结点插入到活结点优先队列中。而当 $bb \geq bestc$ 时，可将结点 $node$ 舍去。

6.9 批处理作业问题

3. 算法描述

```
do
{
    if (enode.s == n ) { // 叶结点
        if (enode.sf2 < bestc) {
            bestc = enode.sf2;
            for (int i = 0; i < n; i++)
                bestx[i] = enode.x[i];
        }
    }
}
```

当 $\text{enode.sf2} < \text{bestc}$ 时，更新当前最优值 bestc 和相应的最优解 bestx

6.9 批处理作业问题

3. 算法描述

else // 产生当前扩展结点的儿子结点

```
for (int i = enode.s; i < n; i++) {
```

```
    MyMath.swap(enode.x, enode.s,i);
```

```
    int [] f= new int [3];
```

```
    int bb=bound(enode,f);
```

```
    if (bb < bestc )    {
```

```
        HeapNode node=new HeapNode(enode,f,bb,n);
```

```
        heap.put(node);    }
```

```
    MyMath.swap(enode.x, enode.s,i);
```

```
} // 完成结点扩展
```

当 $bb < bestc$ 时，将儿子结点插入到活结点优先队列中

第7章 概率算法

随机数

随机数在概率算法设计中扮演着十分重要的角色。在现实计算机上无法产生真正的随机数，因此在概率算法中使用的随机数都是一定程度上随机的，即伪随机数。

线性同余法是产生伪随机数的最常用的方法。由线性同余法产生的随机序列 a_0, a_1, \dots, a_n 满足

$$\begin{cases} a_0 = d \\ a_n = (ba_{n-1} + c) \bmod m \end{cases} \quad n = 1, 2, \Lambda$$

其中 $b \geq 0$ ， $c \geq 0$ ， $d \leq m$ 。 d 称为该随机序列的种子。如何选取该方法中的常数 b 、 c 和 m 直接关系到所产生的随机序列的随机性能。这是随机性理论研究的内容，已超出本书讨论的范围。从直观上看， m 应取得充分大，因此可取 m 为机器大数，另外应取 $\gcd(m, b) = 1$ ，因此可取 b 为一素数。

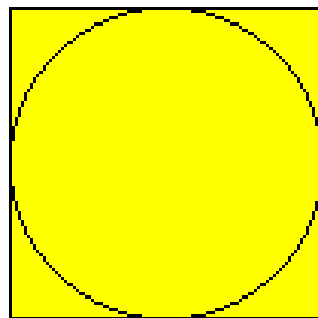
数值概率算法

用随机投点法计算 π 值

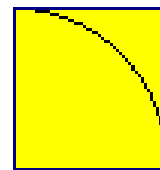
设有一半径为 r 的圆及其外切正方形。向该正方形随机地投掷 n 个点。设落入圆内的点数为 k 。由于所投入的点在正方形上均匀分布，因而所投入的点落入圆内的概率为 $\frac{\pi r^2}{4r^2} = \frac{\pi}{4}$ 。所以当 n 足够大

时， k 与 n 之比就逼近这一概率。从而 $\pi \approx \frac{4k}{n}$ 。

```
public static double darts(int n)
{ // 用随机投点法计算 $\pi$ 值
  int k=0;
  for (int i=1; i <=n; i++) {
    double x=dart.fRandom();
    double y=dart.fRandom();
    if ((x*x+y*y)<=1) k++;
  }
  return 4*k/(double)n;
}
```



(a)

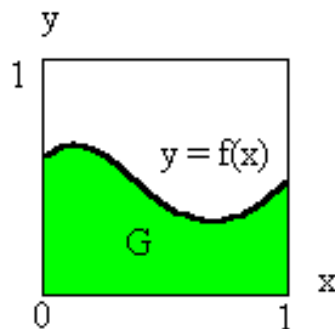


(b)

计算定积分

设 $f(x)$ 是 $[0, 1]$ 上的连续函数，且 $0 \leq f(x) \leq 1$ 。

需要计算的积分为 $I = \int_0^1 f(x) dx$ ，积分 I 等于图中的面积 G 。



在图所示单位正方形内均匀地作投点试验，则随机点落在曲线下方的概率为

$$P_r\{y \leq f(x)\} = \int_0^1 \int_0^{f(x)} dy dx = \int_0^1 f(x) dx$$

假设向单位正方形内随机地投入 n 个点 (x_i, y_i) 。如果有 m 个点落入

G 内，则随机点落入 G 内的概率 $I \approx \frac{m}{n}$

解非线性方程组

求解下面的非线性方程组

$$\begin{cases} f_1(x_1, x_2, \Lambda, x_n) = 0 \\ f_2(x_1, x_2, \Lambda, x_n) = 0 \\ \Lambda \wedge \Lambda \wedge \Lambda \wedge \Lambda \wedge \Lambda \wedge \Lambda \wedge \Lambda \\ f_n(x_1, x_2, \Lambda, x_n) = 0 \end{cases}$$

其中, x_1, x_2, \dots, x_n 是实变量, f_i 是未知量 x_1, x_2, \dots, x_n 的非线性实函数。要求确定上述方程组在指定求根范围内的一组解 $x_1^*, x_2^*, \Lambda, x_n^*$

在指定求根区域 D 内, 选定一个随机点 x_0 作为随机搜索的出发点。在算法的搜索过程中, 假设第 j 步随机搜索得到的随机搜索点为 x_j 。在第 $j+1$ 步, 计算出下一步的随机搜索增量 Δx_j 。从当前点 x_j 依 Δx_j 得到第 $j+1$ 步的随机搜索点。当 $x < \varepsilon$ 时, 取为所求非线性方程组的近似解。否则进行下一步新的随机搜索过程。

舍伍德(Sherwood)算法

设A是一个确定性算法，当它的输入实例为x时所需的计算时间记为 $t_A(x)$ 。设 X_n 是算法A的输入规模为n的实例的全体，则当问题的输入规模为n时，算法A所需的平均时间为

$$\bar{t}_A(n) = \sum_{x \in X_n} t_A(x) / |X_n|$$

这显然不能排除存在 $x \in X_n$ 使得 $t_A(x) \gg \bar{t}_A(n)$ 的可能性。希望获得一个概率算法B，使得对问题的输入规模为n的每一个实例均有

$$t_B(x) = \bar{t}_A(n) + s(n)$$

这就是舍伍德算法设计的基本思想。当 $s(n)$ 与 $\bar{t}_A(n)$ 相比可忽略时，舍伍德算法可获得很好的平均性能。

舍伍德(Sherwood)算法

复习学过的Sherwood算法:

(1) 线性时间选择算法

(2) 快速排序算法

有时也会遇到这样的情况, 即所给的确定性算法无法直接改造成舍伍德型算法。此时可借助于随机预处理技术, 不改变原有的确定性算法, 仅对其输入进行随机洗牌, 同样可收到舍伍德算法的效果。例如, 对于确定性选择算法, 可以用下面的洗牌算法**shuffle**将数组**a**中元素随机排列, 然后用确定性选择算法求解。这样做所收到的效果与舍伍德型算法的效果是一样的。

```
public static void shuffle(Comparable []a, int n)
```

```
{// 随机洗牌算法
```

```
    rnd = new Random();
```

```
    for (int i=1;i<n;i++) {
```

```
        int j=rnd.random(n-i+1)+i;
```

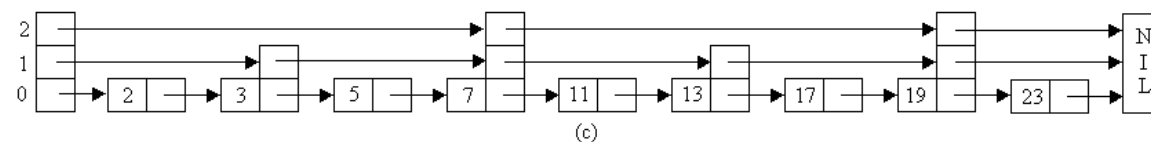
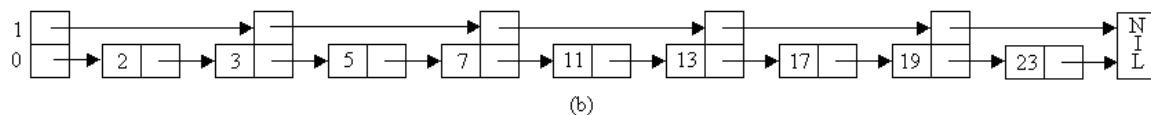
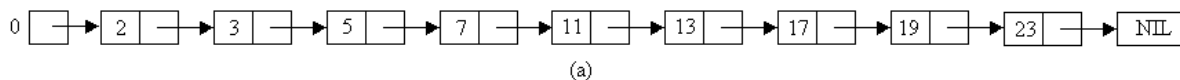
```
        MyMath.swap(a, i, j);
```

```
    }
```

```
}
```

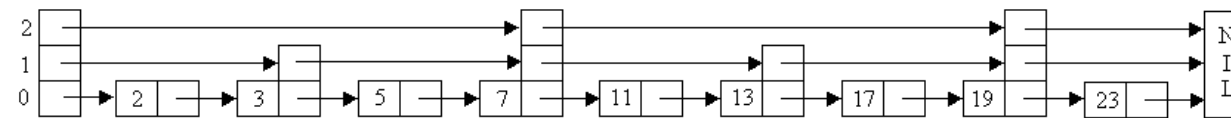
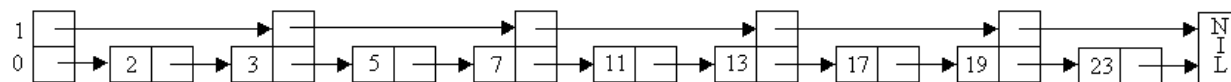
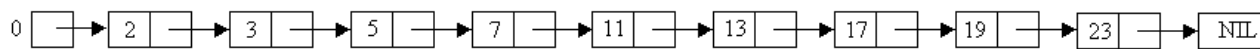
跳跃表

- 舍伍德型算法的设计思想还可用于设计高效的数据结构。
- 如果用有序链表来表示一个含有 n 个元素的有序集 S ，则在最坏情况下，搜索 S 中一个元素需要 $\Omega(n)$ 计算时间。
- 提高有序链表效率的一个技巧是在有序链表的部分结点处增设附加指针以提高其搜索性能。在增设附加指针的有序链表中搜索一个元素时，可借助于附加指针跳过链表中若干结点，加快搜索速度。这种增加了向前附加指针的有序链表称为跳跃表。
- 应在跳跃表的哪些结点增加附加指针以及在该结点处应增加多少指针完全采用随机化方法来确定。这使得跳跃表可在 $O(\log n)$ 平均时间内支持关于有序集 S 的搜索、插入和删除等运算。



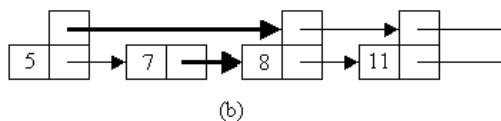
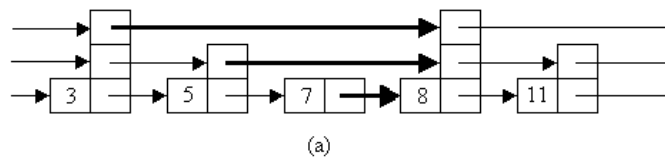
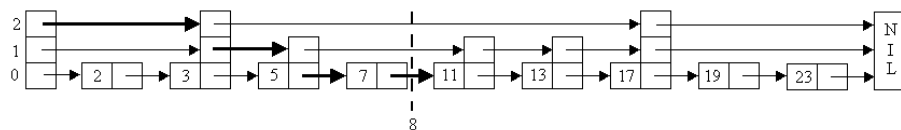
跳跃表

在一般情况下，给定一个含有 n 个元素的有序链表，可以将它改造成一个完全跳跃表，使得每一个 k 级结点含有 $k+1$ 个指针，分别跳过 $2^k-1, 2^{k-1}-1, \dots, 2^0-1$ 个中间结点。第 i 个 k 级结点安排在跳跃表的位置 $i2^k$ 处， $i \geq 0$ 。这样就可以在时间 $O(\log n)$ 内完成集合成员的搜索运算。在一个完全跳跃表中，最高级的结点是 $\lceil \log n \rceil$ 级结点。



完全跳跃表与完全二叉搜索树的情形非常类似。它虽然可以有效地支持成员搜索运算，但不适应于集合动态变化的情况。集合元素的插入和删除运算会破坏完全跳跃表原有的平衡状态，影响后继元素搜索的效率。

为了在动态变化中维持跳跃表中附加指针的平衡性，必须使跳跃表中 k 级结点数维持在总结点数的一定比例范围内。注意到在一个完全跳跃表中，50%的指针是0级指针；25%的指针是1级指针；...； $(100/2^{k+1})\%$ 的指针是 k 级指针。因此，在插入一个元素时，以概率 $1/2$ 引入一个0级结点，以概率 $1/4$ 引入一个1级结点，...，以概率 $1/2^{k+1}$ 引入一个 k 级结点。另一方面，一个 i 级结点指向下一个同级或更高级的结点，它所跳过的结点数不再准确地维持在 2^{i-1} 。经过这样的修改，就可以在插入或删除一个元素时，通过对跳跃表的局部修改来维持其平衡性。



注意到，在一个完全跳跃表中，具有 i 级指针的结点中有一半同时具有 $i+1$ 级指针。为了维持跳跃表的平衡性，可以事先确定一个实数 $0 < p < 1$ ，并要求在跳跃表中维持在具有 i 级指针的结点中同时具有 $i+1$ 级指针的结点所占比例约为 p 。为此目的，在插入一个新结点时，先将其结点级别初始化为0，然后用随机数生成器反复地产生一个 $[0, 1]$ 间的随机实数 q 。如果 $q < p$ ，则使新结点级别增加1，直至 $q \geq p$ 。由此产生新结点级别的过程可知，所产生的新结点的级别为0的概率为 $1-p$ ，级别为1的概率为 $p(1-p)$ ，...，级别为 i 的概率为 $p^i(1-p)$ 。如此产生的新结点的级别有可能是一个很大的数，甚至远远超过表中元素的个数。为了避免这种情况，用 $\log_{1/p} n$ 作为新结点级别的上界。其中 n 是当前跳跃表中结点个数。当前跳跃表中任一结点的级别不超过 $\log_{1/p} n$

拉斯维加斯(Las Vegas)算法

拉斯维加斯算法的一个显著特征是它所作的随机性决策有可能导致算法找不到所需的解。

```
public static void obstinate(Object x, Object y)
```

```
{// 反复调用拉斯维加斯算法LV(x,y), 直到找到问题的一个解y
```

```
    boolean success= false;
```

```
    while (!success) success=lv(x,y);
```

```
}
```

设 $p(x)$ 是对输入 x 调用拉斯维加斯算法获得问题的一个解的概率。

一个正确的拉斯维加斯算法应该对所有输入 x 均有 $p(x)>0$ 。

设 $t(x)$ 是算法**obstinate**找到具体实例 x 的一个解所需的平均时间, $s(x)$ 和 $e(x)$ 分别是算法对于具体实例 x 求解成功或求解失败所需的平均时间, 则有: $t(x) = p(x)s(x) + (1 - p(x))(e(x) + t(x))$

解此方程可得:

$$t(x) = s(x) + \frac{1 - p(x)}{p(x)} e(x)$$

n后问题

对于n后问题的任何一个解而言，每一个皇后在棋盘上的位置无任何规律，不具有系统性，而更象是随机放置的。由此容易想到下面的拉斯维加斯算法。

在棋盘上相继的各行中随机地放置皇后，并注意使新放置的皇后与已放置的皇后互不攻击，直至n个皇后均已相容地放置好，或已没有下一个皇后的可放置位置时为止。

如果将上述随机放置策略与回溯法相结合，可能会获得更好的效果。可以先在棋盘的若干行中随机地放置皇后，然后在后继行中用回溯法继续放置，直至找到一个解或宣告失败。随机放置的皇后越多，后继回溯搜索所需的时间就越少，但失败的概率也就越大。

stopVegas	p	s	e	t
0	1.0000	262.00	--	262.00
5	0.5039	33.88	47.23	80.39
12	0.0465	13.00	10.20	222.11

整数因子分解

设 $n > 1$ 是一个整数。关于整数 n 的因子分解问题是找出 n 的如下形式的惟一分解式：
$$n = p_1^{m_1} p_2^{m_2} \wedge p_k^{m_k}$$

其中， $p_1 < p_2 < \dots < p_k$ 是 k 个素数， m_1, m_2, \dots, m_k 是 k 个正整数。

如果 n 是一个合数，则 n 必有一个非平凡因子 x ， $1 < x < n$ ，使得 x 可以整除 n 。给定一个合数 n ，求 n 的一个非平凡因子的问题称为整数 n 的因子分割问题。

```
private static int split(int n)
```

```
{  
    int m = (int) Math.floor(Math.sqrt((double)n));  
    for (int i=2; i<=m; i++)  
        if (n%i==0) return i;  
    return 1;  
}
```

事实上，算法**split(n)**是对范围在 $1 \sim x$ 的所有整数进行了试除而得到范围在 $1 \sim x^2$ 的任一整数的因子分割。

Pollard算法

在开始时选取 $0 \sim n-1$ 范围内的随机数，然后递归地由

$$x_i = (x_{i-1}^2 - 1) \bmod n$$

产生无穷序列 $x_1, x_2, \dots, x_k, \dots$

对于 $i=2^k$ ，以及 $2^k < j \leq 2^{k+1}$ ，算法计算出 $x_j - x_i$ 与 n 的最大公因子

$d = \gcd(x_j - x_i, n)$ 。如果 d 是 n 的非平凡因子，则实现对 n 的一次分割，算法输出 n 的因子 d 。

```
private static void pollard(int n)
```

```
{// 求整数n因子分割的拉斯维加斯算法
```

```
    rnd = new Random(); // 初始化随机数
```

```
    int i=1,k=2;
```

```
    int x=rnd.random(n),y=x; // 随机整数
```

```
    while (true) {
```

```
        i++;
```

```
        x=(x*x-1)%n;
```

```
        int d=gcd(y-x,n); // 求n的非平凡因子
```

```
        if ((d>1) && (d<n)) System.out.println(d);
```

```
        if (i==k) {
```

```
            y=x;
```

```
            k*=2;} } }
```

对Pollard算法更深入的分析可知，执行算法的while循环约 \sqrt{p} 次后，Pollard算法会输出 n 的一个因子 p 。由于 n 的最小素因子 $p \leq \sqrt{n}$ ，故Pollard算法可在 $O(n^{1/4})$ 时间内找到 n 的一个素因子。

蒙特卡罗(Monte Carlo)算法

- 在实际应用中常会遇到一些问题，不论采用确定性算法或概率算法都无法保证每次都能得到正确的解答。蒙特卡罗算法则在一般情况下可以保证对问题的所有实例都以高概率给出正确解，但是通常无法判定一个具体解是否正确。
- 设 p 是一个实数，且 $1/2 < p < 1$ 。如果一个蒙特卡罗算法对于问题的任一实例得到正确解的概率不小于 p ，则称该蒙特卡罗算法是 p 正确的，且称 $p-1/2$ 是该算法的优势。
- 如果对于同一实例，蒙特卡罗算法不会给出2个不同的正确解答，则称该蒙特卡罗算法是一致的。
- 有些蒙特卡罗算法除了具有描述问题实例的输入参数外，还具有描述错误解可接受概率的参数。这类算法的计算时间复杂性通常由问题的实例规模以及错误解可接受概率的函数来描述。

蒙特卡罗(Monte Carlo)算法

对于一个一致的 p 正确蒙特卡罗算法，要提高获得正确解的概率，只要执行该算法若干次，并选择出现频次最高的解即可。如果重复调用一个一致的 $(1/2+\varepsilon)$ 正确的蒙特卡罗算法 $2m-1$ 次，得到正确解的概率至少为 $1-\delta$ ，其中，

$$\delta = \frac{1}{2} - \varepsilon \sum_{i=0}^{m-1} \binom{2i}{i} \left(\frac{1}{4} - \varepsilon^2\right)^i \leq \frac{(1-4\varepsilon^2)^m}{4\varepsilon\sqrt{\pi m}}$$

对于一个解所给问题的蒙特卡罗算法 $MC(x)$ ，如果存在问题实例的子集 X 使得：

- (1) 当 $x \notin X$ 时， $MC(x)$ 返回的解是正确的；
 - (2) 当 $x \in X$ 时，正确解是 y_0 ，但 $MC(x)$ 返回的解未必是 y_0 。
- 称上述算法 $MC(x)$ 是偏 y_0 的算法。

重复调用一个一致的， p 正确偏 y_0 蒙特卡罗算法 k 次，可得到一个 $O(1-(1-p)^k)$ 正确的蒙特卡罗算法，且所得算法仍是一个一致的偏 y_0 蒙特卡罗算法。

主元素问题

设 $T[1:n]$ 是一个含有 n 个元素的数组。当 $|\{i \mid T[i]=x\}| > n/2$ 时，称元素 x 是数组 T 的主元素。

```
public static boolean majority(int[]t, int n)
```

```
{// 判定主元素的蒙特卡罗算法
```

```
    rnd = new Random();
```

```
    int i=rnd.random(n)+1;
```

```
    int x=t[i]; // 随机选择数组元素
```

```
    int k=0;
```

```
    for (int j=1;j<=n;j++)
```

```
        if (t[j]==x) k++;
```

```
    return (k>n/2); // k>n/2 时t含有主元素
```

```
}
```

对于任何给定的 $\varepsilon > 0$ ，算法

majorityMC重复调用 $\lceil \log(1/\varepsilon) \rceil$ 次算法**majority**。它是一个偏真蒙特卡罗算法，且其错误概率小于 ε 。

算法**majorityMC**所需的计算时间显然是 $O(n \log(1/\varepsilon))$ 。

```
public static boolean majorityMC(int[]t, int n, double e)
```

```
{// 重复 $\lceil \log(1/e) \rceil$ 次调用算法majority
```

```
    int k= (int) Math.ceil(Math.log(1/e)/Math.log(2));
```

```
    for (int i=1;i<=k;i++)
```

```
        if (majority(t,n)) return true;
```

```
    return false;
```

```
}
```


素数测试

Wilson定理: 对于给定的正整数 n , 判定 n 是一个素数的充要条件是 $(n-1)! \equiv -1 \pmod{n}$ 。

费尔马小定理: 如果 p 是一个素数, 且 $0 < a < p$, 则 $a^{p-1} \equiv 1 \pmod{p}$ 。

二次探测定理: 如果 p 是一个素数, 且 $0 < x < p$, 则方程 $x^2 \equiv 1 \pmod{p}$ 的解为 $x=1, p-1$ 。

```
private static int power(int a, int p, int n)    public static boolean prime(int n)
```

```
{// 计算  $a^p \bmod n$ , 并实施对 $n$ 的二次探测    {// 素数测试的蒙特卡罗算法
```

```
    int x, result;
```

```
    if (p==0) result=1;
```

```
    else {
```

```
        x=power(a,p/2,n); // 递归计算
```

```
        result=(x*x)%n;    //
```

```
        if ((result==1)&&(x!=1)&&
```

```
            composite=true;
```

```
        if ((p%2)==1)    //  $p$ 是奇数
```

```
            result=(result*a)%n;
```

```
    }
```

```
    return result;}

```

```
    rnd = new Random();
```

```
    int a, result;
```

```
    composite=false;
```

```
    a=rnd.nextInt(p-2)+2;
```

算法**prime**是一个偏假3/4正确的蒙特卡罗算法。通过多次重复调用错误概率不超过 $(1/4)^k$ 。这是一个很保守的估计, 实际使用的效果要好得多。

```
    } else;
```

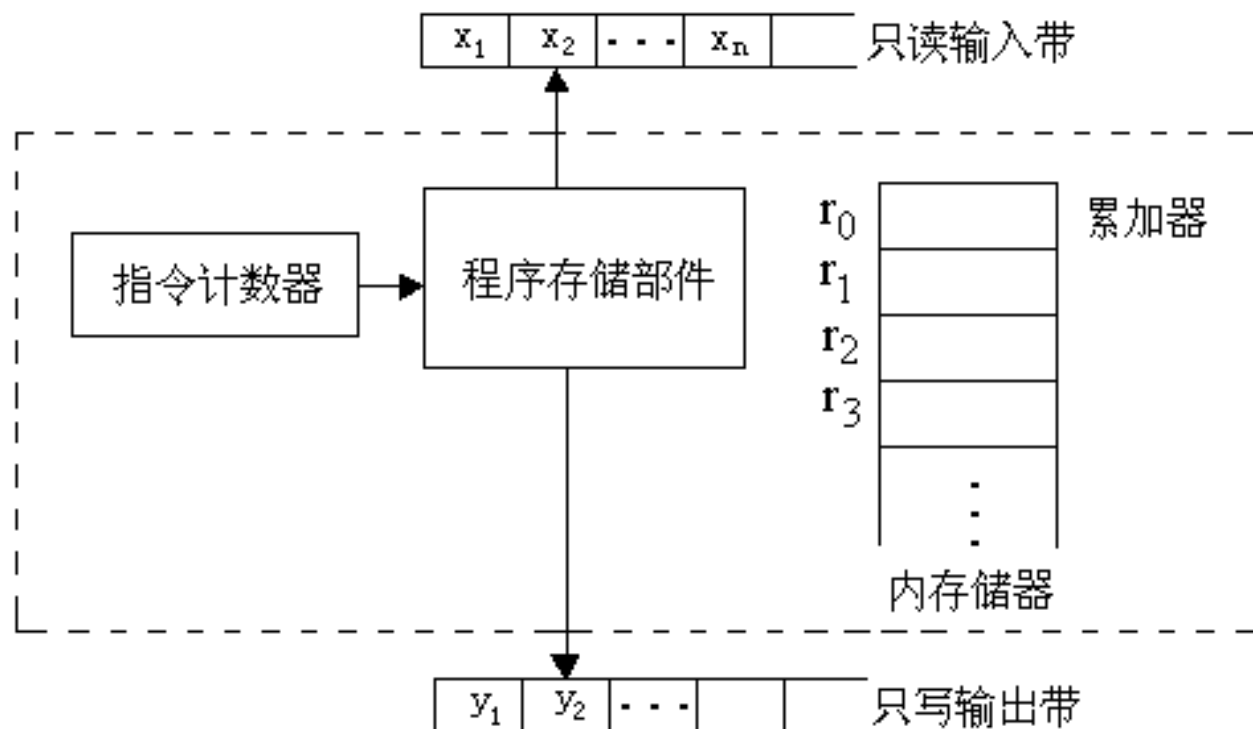
第8章 NP完全性理论

8.1 计算模型

- 8.1.1 随机存取机RAM
- 8.1.2 随机存取存储程序机RASP
- 8.1.3 RAM模型的变形与简化
- 8.1.4 图灵机
- 8.1.5 图灵机模型与RAM模型的关系
- 8.1.6 问题变换与计算复杂性归约

8.1.1 随机存取机RAM

1. RAM的结构



8.1.1 随机存取机RAM

2. RAM程序

一个RAM程序定义了从输入带到输出带的一个映射。可以对这种映射关系作2种不同的解释。

解释一：把RAM程序看成是计算一个函数

若一个RAM程序P总是从输入带前 n 个方格中读入 n 个整数 x_1, x_2, \dots, x_n ，并且在输出带的第一个方格上输出一个整数 y 后停机，那么就说程序P计算了函数 $f(x_1, x_2, \dots, x_n) = y$

解释二：把RAM程序当作一个语言接受器。

将字符串 $S = a_1 a_2 \dots a_n$ 放在输入带上。在输入带的第一个方格中放入符号 a_1 ，第二个方格中放入符号 a_2 ， \dots ，第 n 个方格中放入符号 a_n 。然后在第 $n+1$ 个方格中放入0，作为输入串的结束标志符。如果一个RAM程序P读了字符串 S 及结束标志符0后，在输出带的第一格输出一个1并停机，就说程序P接受字符串 S 。

8.1.1 随机存取机RAM

3. RAM程序的耗费标准

标准一：均匀耗费标准

在均匀耗费标准下，每条RAM指令需要一个单位时间；每个寄存器占用一个单位空间。以后除特别说明，RAM程序的复杂性将按照均匀耗费标准衡量。

标准二：对数耗费标准

对数耗费标准是基于这样的假定，即执行一条指令的耗费与以二进制表示的指令的操作数长度成比例。在RAM计算模型下，假定一个寄存器可存放一个任意大小的整数。因此若设 $l(i)$ 是整数 i 所占的二进制位数，则

$$l(i) = \begin{cases} \lfloor \log |i| \rfloor & i \neq 0 \\ 1 & i = 0 \end{cases}$$

8.1.2 随机存取存储程序机RASP

1. RASP的结构

RASP的整体结构类似于RAM，所不同的是RASP的程序是存储在寄存器中的。每条RASP指令占据2个连续的寄存器。第一个寄存器存放操作码的编码，第二个寄存器存放地址。RASP指令用整数进行编码。

2. RASP程序的复杂性

不管是在均匀耗费标准下，还是在对数耗费标准下，RAM程序和RASP程序的复杂性只差一个常数因子。在一个计算模型下 $T(n)$ 时间内完成的输入-输出映射可在另一个计算模型下模拟，并在 $kT(n)$ 时间内完成。其中 k 是一个常数因子。空间复杂性的情况也是类似的。

8.1.3 RAM模型的变形与简化

1. 实随机存取机 RRAM

在RRAM模型下，一个存储单元可以存放一个实数。下列的各运算为基本运算且每个运算只耗费单位时间。

- (1) 算术运算 $+$, $-$, \times , $/$ 。
- (2) 2个实数间的比较($<$, \leq , $=$, \neq , \geq , $>$)。
- (3) 间接寻址(整数地址)。
- (4) 常见函数的计算, 如三角函数, 指数函数, 对数函数等。

优点：能够方便处理实数；
适合于用FORTRAN, PASCAL等高级语言写的算法。

8.1.3 RAM模型的变形与简化

2. 直线式程序

对于许多问题，所设计的RAM程序中的转移指令仅用于重复一组指令，而且重复的次数与问题的输入规模 n 成比例。在这种情况下，可以用重复地写出相同指令组的方法来消除程序中的循环。由此，对每一个固定的 n 得到一个无循环的直线式程序。

经过对RAM模型的简化，得到直线式程序的指令系统如下：

$x \leftarrow y+z$

$x \leftarrow y-z$

$x \leftarrow y*z$

$x \leftarrow y/z$

$x \leftarrow i$

每条指令耗费一个单位时间。

其中 x ， y 和 z 是符号地址(或变量)，而 i 是常数。

8.1.3 RAM模型的变形与简化

3. 位式计算

直线式程序计算模型显然是基于均匀耗费标准的。在对数耗费标准下，使用另一个RAM的简化计算模型，称之为位式计算 (Bitwise Computation) 模型。

除了下列2点外，该计算模型与直线式程序计算模型基本相同：

(1) 假设所有变量取值0或1，即为位变量。

(2) 所用的运算是逻辑运算而不是算术运算。

用 \wedge 代表与， \vee 代表或， \oplus 代表异或， \neg 代表非。

在位式计算模型下，每个逻辑运算指令耗费一个单位时间。

8.1.3 RAM模型的变形与简化

4. 位向量运算 (Bit Vector Operations)

若在直线式程序计算模型中，假设所有变量均为位向量，而且所用的运算均为位操作指令，则得到位向量运算计算模型。

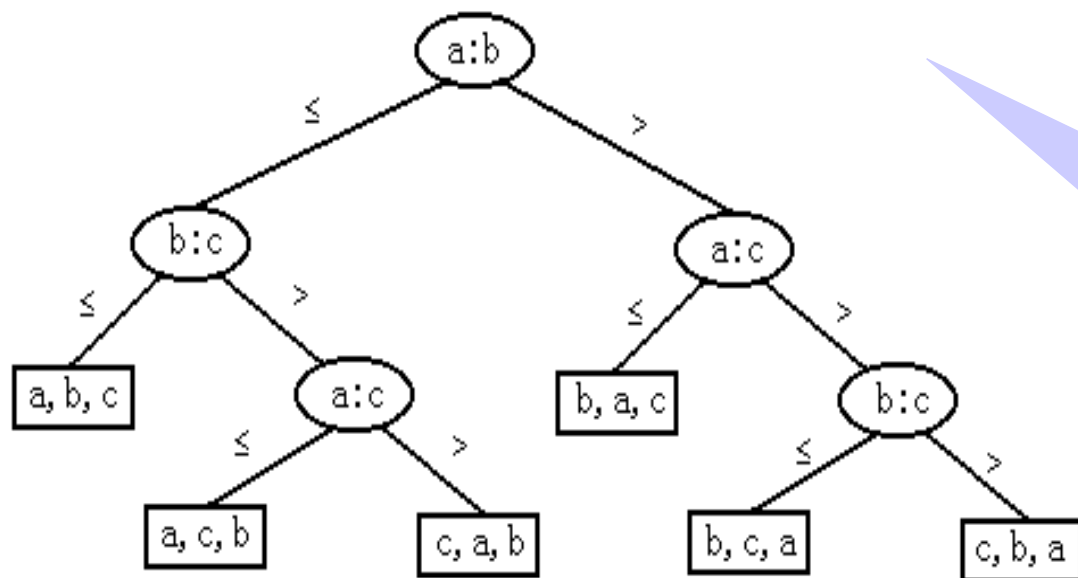
例如，要表示一个有100个顶点的图中从顶点 v 到其余各顶点间有没有边相连，可以用100位的一个位向量表示。若顶点 v 到顶点 v_j 之间有边相连，则该位向量的第 j 位为1，否则为0。

缺点：所需的机器字长要远大于其他模型。

8.1.3 RAM模型的变形与简化

5. 判定树

判定树是一棵二叉树。它的每个内结点表示一个形如 $x:y$ 的比较。指向该结点左儿子的边相应于 $x \leq y$ ，标号为 \leq 。指向该结点右儿子的边相应于 $x > y$ ，标号为 $>$ 。每一次比较耗费一个单位时间。下图是对 a, b, c 三个数进行排序的一棵判定树。



在判定树模型下，算法的时间复杂性可用判定树的高度衡量。最大的比较次数是从根到叶的最长路径的长度。

8.1.3 RAM模型的变形与简化

6. 代数计算树ACT

以 $x = (x_1, x_2, \dots, x_n)$ 为输入的一棵代数计算树 T 是一棵二叉树，且：

(1) 每个叶结点表示一个输出结果 YES 或 NO。

(2) 每个单儿子内部结点 (简单结点) v 表示下列形式运算指令：

$$f_v = f_{v1} \text{ op } f_{v2} \text{ 或 } f_v = c \text{ op } f_{v1} \text{ 或 } f_v = \sqrt{f_{v1}}$$

其中， f_{v1} 和 f_{v2} 分别是结点 v 在树 T 中的祖先结点 $v1$ 和 $v2$ 处得到的结果值，或是 x 的分量； $\text{op} \in \{+, -, \times, /\}$ ； c 是一个常数。

(3) 每个有 2 个儿子的内部结点 (分支结点) v ，表示下列形式的测试指令：

$$f_{v1} > 0 \text{ 或 } f_{v1} \geq 0 \text{ 或 } f_{v1} = 0$$

其中， f_{v1} 是结点 v 在树 T 中的祖先结点 $v1$ 处得到的结果值，或是 x 的分量。

8.1.3 RAM模型的变形与简化

7. 代数判定树ADT (Algebraic Decision Tree)

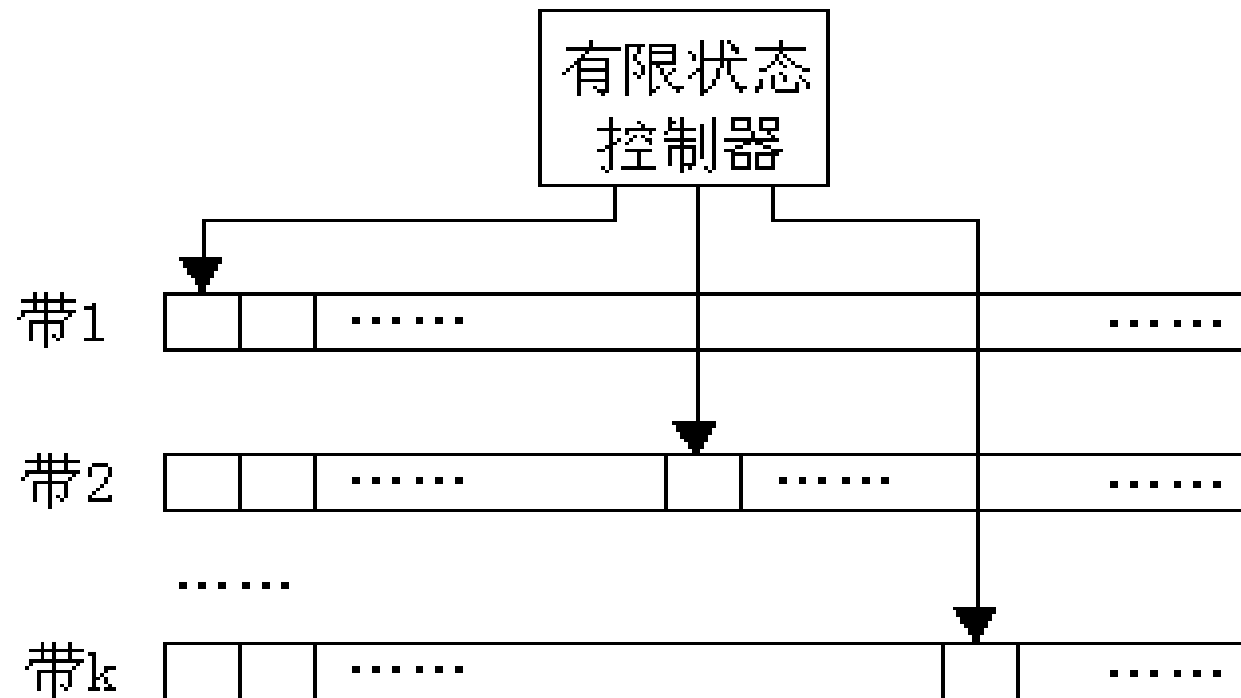
在代数计算树 T 中，若将所有的简单结点都压缩到其最近的子孙分支结点处，并将简单结点处的计算在压缩后的分支结点处同时完成，则计算结果可看作是输入 x 的一个代数函数 $f_v(x)$ 。由此引出另一个称为代数判定树的计算模型。

代数判定树 T 是一棵二叉树，且

- (1) 每个叶结点表示输出结果YES或NO。
- (2) 每个内部结点 v 表示一个形如 $f_v(x_1, x_2, \dots, x_n) : 0$ 的比较。其中， $x = (x_1, x_2, \dots, x_n)$ 是输入， f_v 是一个代数函数。

8.1.4 图灵机

1. 多带图灵机



8.1.4 图灵机

1. 多带图灵机

根据有限状态控制器的当前状态及每个读写头读到的带符号，

图灵机的一个计算步可实现下面3个操作之一或全部。

- (1) 改变有限状态控制器中的状态。
 - (2) 清除当前读写头下的方格中原有带符号并写上新的带符号。
 - (3) 独立地将任何一个或所有读写头，向左移动一个方格(L)或向右移动一个方格(R)或保持不动(S)。
- 带图灵机可形式化地描述为五元组 $(Q; T, I, \delta, b, q_0, q_f)$ ，其中：

- (1) Q 是有限个状态的集合。
- (2) T 是有限个带符号的集合。
- (3) I 是输入符号的集合， $I \subseteq T$ 。
- (4) b 是惟一的空白符， $b \in T - I$ 。
- (5) q_0 是初始状态。
- (6) q_f 是终止(或接受)状态。
- (7) δ 是移动函数。它是从 $Q \times T^k$ 的某一子集映射到 $Q \times (T \times \{L, R, S\})^k$ 的函数。

8.1.4 图灵机

1. 多带图灵机

与RAM模型类似，图灵机既可作为语言接受器，也可作为计算函数的装置。

图灵机 M 的时间复杂性 $T(n)$ 是它处理所有长度为 n 的输入所需的最大计算步数。如果对某个长度为 n 的输入，图灵机不停机， $T(n)$ 对这个 n 值无定义。

图灵机的空间复杂性 $S(n)$ 是它处理所有长度为 n 的输入时，在 k 条带上所使用过的方格数的总和。如果某个读写头无限地向右移动而不停机， $S(n)$ 也无定义。

8.1.5 图灵机模型与RAM模型的关系

图灵机模型与RAM模型的关系是指同一问题在这2种不同计算模型下的复杂性之间的关系。

定理8-3 对于问题P的任何长度为 n 的输入，设求解问题P的算法A在 k 带图灵机模型TM下的时间复杂性为 $T(n)$ ，那么，算法A在RAM模型下的时间复杂性为 $O(T^2(n))$ 。

定理8-4 对于问题P的任何长度为 n 的输入，设求解问题P的算法A在RAM模型下，不含有乘法和除法指令，且按对数耗费标准其时间复杂性为 $T(n)$ ，那么，算法A在 k 带图灵机模型TM下的时间复杂性为 $O(T^2(n))$ 。

8.1.6 问题变换与计算复杂性归约

通过问题变换的技巧，可以将2个不同问题的计算复杂性联系在一起。这样就可以将一个问题的计算复杂性归结为另一个问题的计算复杂性，从而实现问题的计算复杂性归约。

具体地说，假设有2个问题A和B，将问题A变换为问题B是指：

(1) 将问题A的输入变换为问题B的适当输入。

(2) 解出问题B。

(3) 把问题B的输出变换为问题A的正确解。

若用 $O(\tau(n))$ 时间能完成上述变换的第(1)步和第(3)步，则称问题A是 $\tau(n)$ 时间可变换到问题B，且简记为 $A \propto_{\tau(n)} B$ 。其中的 n 通常为问题A的规模(大小)。

当 $\tau(n)$ 为 n 的多项式时，称问题A可在多项式时间内变换为问题B。特别地，当 $\tau(n)$ 为 n 的线性函数时，称问题A可线性地变换为问题B。

8.1.6 问题变换与计算复杂性归约

问题的变换与问题的计算复杂性归约的关系：

命题1 (计算时间下界归约)： 若已知问题A的计算时间下界为 $T(n)$ ，且问题A是 $\tau(n)$ 可变换到问题B，即 $A \propto_{\tau(n)} B$ ，则 $T(n) - o(\tau(n))$ 为问题B的一个计算时间下界。

命题2 (计算时间上界归约)： 若已知问题B的计算时间上界为 $T(n)$ ，且问题A是 $\tau(n)$ 可变换到问题B，即 $A \propto_{\tau(n)} B$ ，则 $T(n) + o(\tau(n))$ 是问题A的一个计算时间上界。

在命题1和命题2中，当 $\tau(n) = o(T(n))$ 时，问题A的下界归约为问题B的下界，问题B的上界归约为问题A的上界。

8.1.6 问题变换与计算复杂性归约

通过问题变换获得问题的计算时间下界的例子：

(1) 判别函数问题：给定 n 个实数 x_1, x_2, \dots, x_n ，计算其判别函数 $\prod_{i \neq j} (x_i - x_j)$ 。

元素惟一性问题可以在 $O(1)$ 时间内变换为判别函数问题。任何一个计算判别函数的算法，计算出判别函数值后，再作一次测试，判断其值是否为0，即可得到元素惟一性问题的解。由命题1即知，元素惟一性问题的计算时间下界 $\Omega(n \log n)$ 也是判别函数问题的一个计算时间下界。

(2) 最接近点对问题：给定平面上 n 个点，找出这 n 个点中距离最近的2个点。

在元素惟一性问题中，将每一个实数 x_i ， $1 \leq i \leq n$ ，变换为平面上的点 $(x_i, 0)$ ， $1 \leq i \leq n$ ，则元素惟一性问题可以在线性时间内变换为最接近点对问题。

8.2 P类与NP类问题

- 8.2.1 非确定性图灵机
- 8.2.2 P类与NP类语言
- 8.2.3 多项式时间验证

8.2.1 非确定性图灵机

在图灵机计算模型中，移动函数 δ 是单值的，即对于 $Q \times T^k$ 中的每一个值，当它属于 δ 的定义域时， $Q \times (T \times \{L, R, S\})^k$ 中只有惟一的值与之对应，称这种图灵机为**确定性图灵机**，简记为DTM (Deterministic Turing Machine)。

非确定性图灵机 (NDTM)：一个 k 带的非确定性图灵机 M 是一个7元组： $(Q, T, I, \delta, b, q_0, q_f)$ 。与确定性图灵机不同的是非确定性图灵机允许移动函数 δ 具有不确定性，即对于 $Q \times T^k$ 中的每一个值 $(q; x_1, x_2, \dots, x_k)$ ，当它属于 δ 的定义域时， $Q \times (T \times \{L, R, S\})^k$ 中有惟一的一个子集 $\delta(q; x_1, x_2, \dots, x_k)$ 与之对应。可以在 $\delta(q; x_1, x_2, \dots, x_k)$ 中随意选定一个值作为它的函数值。

8.2.2 P类与NP类语言

P类和NP类语言的定义:

$P = \{L \mid L \text{ 是一个能在多项式时间内被一台DTM所接受的语言}\}$

$NP = \{L \mid L \text{ 是一个能在多项式时间内被一台NDTM所接受的语言}\}$

由于一台确定性图灵机可看作是非确定性图灵机的特例，所以可在多项式时间内被确定性图灵机接受的语言也可在多项式时间内被非确定性图灵机接受。故 $P \subseteq NP$ 。

8.2.2 P类与NP类语言

NP类语言举例——无向图的团问题。

该问题的输入是一个有 n 个顶点的无向图 $G=(V, E)$ 和一个整数 k 。要求判定图 G 是否包含一个 k 顶点的完全子图(团)，即判定是否存在 $V' \subseteq V$ ， $|V'|=k$ ，且对于所有的 $u, v \in V'$ ，有 $(u, v) \in E$ 。

若用邻接矩阵表示图 G ，用二进制串表示整数 k ，则团问题的一个实例可以用长度为 $n^2 + \log k + 1$ 的二进位串表示。因此，团问题可表示为语言：

$\text{CLIQUE} = \{w\#v \mid w, v \in \{0, 1\}^*, \text{以} w \text{为邻接矩阵的图} G \text{有一个} k \text{顶点的团，其中} v \text{是} k \text{的二进制表示。}\}$

8.2.2 P类与NP类语言

接受该语言CLIQUE的非确定性算法：用非确定性选择指令选出包含 k 个顶点的候选顶点子集 V ，然后确定性地检查该子集是否是团问题的一个解。算法分为3个阶段：

算法的第一阶段将输入串 $w\#v$ 分解，并计算出 $n = \sqrt{|w|}$ ，以及用 v 表示的整数 k 。若输入不具有形式 $w\#v$ 或 $|w|$ 不是一个平方数就拒绝该输入。显而易见，第一阶段可 $O(n^2)$ 在时间内完成。

在算法的第二阶段中，非确定性地选择 V 的一个 k 元子集 $V' \subseteq V$ 。

算法的第三阶段是确定性地检查 V' 的团性质。若 V' 是一个团则接受输入，否则拒绝输入。这显然可以在 $O(n^4)$ 时间内完成。因此，整个算法的时间复杂性为 $O(n^4)$ 。

非确定性算法在多项式时间内接受语言CLIQUE，故CLIQUE \in NP。

8.2.3 多项式时间验证

多项式时间可验证语言类VP可定义为:

$VP = \{L \mid L \in \Sigma^*, \Sigma \text{ 为一有限字符集, 存在一个多项式 } p \text{ 和一个多项式时间验证算法 } A(X, Y) \text{ 使得对任意 } X \in \Sigma^*, X \in L \text{ 当且仅当存在 } Y \in \Sigma^*, |Y| \leq p(|X|) \text{ 且 } A(X, Y) = 1\}$ 。

定理8-5: $VP = NP$ 。(证明见书本)

例如(哈密顿回路问题): 一个无向图G含有哈密顿回路吗?

无向图G的哈密顿回路是通过G的每个顶点恰好一次的简单回路。
可用语言HAM-CYCLE 定义该问题如下:

$HAM-CYCLE = \{G \mid G \text{ 含有哈密顿回路}\}$

8.3 NP完全问题

- 8.3.1 多项式时间变换
- 8.3.2 Cook定理

8.3.1 多项式时间变换

设 $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$ 是2个语言。所谓语言 L_1 能在多项式时间内变换为语言 L_2 (简记为 $L_1 \propto_p L_2$) 是指存在映身 $f: \Sigma_1^* \rightarrow \Sigma_2^*$, 且 f 满足:

- (1) 有一个计算 f 的多项式时间确定性图灵机;
- (2) 对于所有 $x \in \Sigma_1^*$, $x \in L_1$, 当且仅当 $f(x) \in L_2$ 。

定义: 语言 L 是NP完全的当且仅当

- (1) $L \in \text{NP}$;
- (2) 对于所有 $L' \in \text{NP}$ 有 $L' \propto_p L$ 。

如果有一个语言 L 满足上述性质(2), 但不一定满足性质

(1), 则称该语言是NP难的。所有NP完全语言构成的语言类称为NP完全语言类, 记为NPC。

8.3.1 多项式时间变换

定理8-6: 设 L 是NP完全的, 则

- (1) $L \in P$ 当且仅当 $P = NP$;
- (2) 若 $L \propto_p L_1$, 且 $L_1 \in NP$, 则 L_1 是NP完全的。

定理8-6的(2)用来证明问题的NP完全性。但前提是: 要有第一个NP完全问题 L 。

8.3.2 Cook定理

定理8-7 (Cook定理): 布尔表达式的可满足性问题SAT是NP完全的。

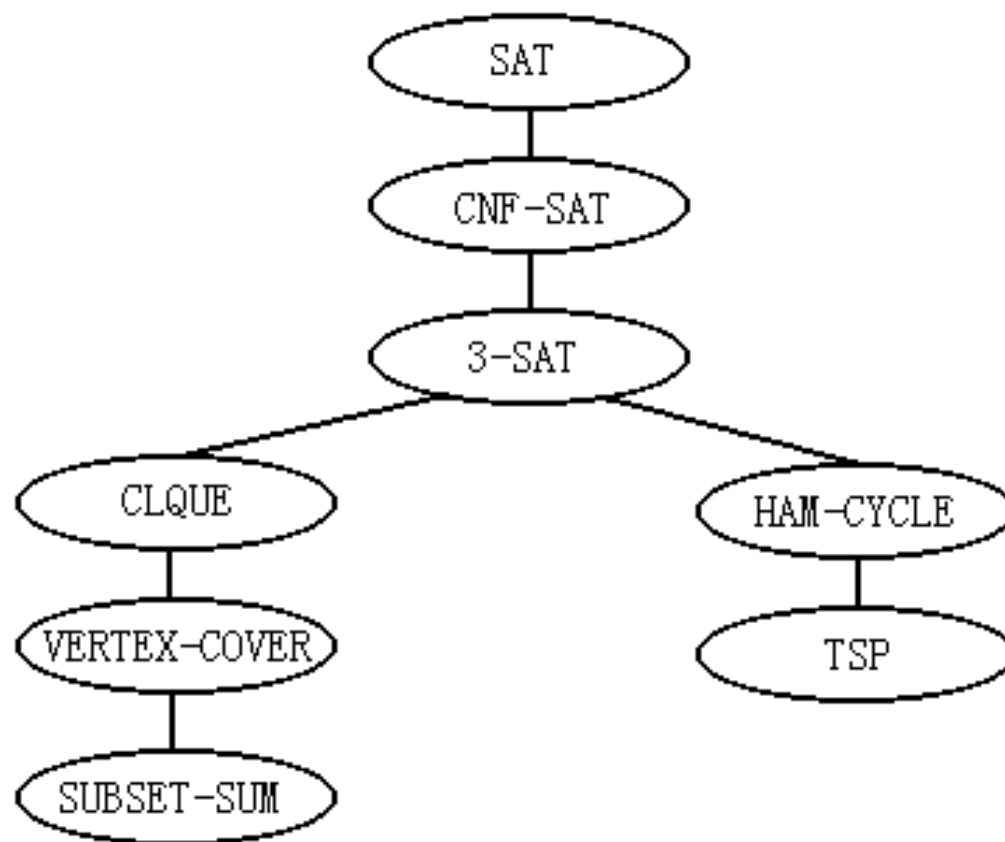
证明: SAT的一个实例是 k 个布尔变量 x_1, \dots, x_k 的 m 个布尔表达式 A_1, \dots, A_m 。若存在各布尔变量 x_i ($1 \leq i \leq k$)的0, 1赋值, 使每个布尔表达式 A_i ($1 \leq i \leq m$)都取值1, 则称布尔表达式 $A_1 A_2 \dots A_m$ 是可满足的。

SAT \in NP是很明显的。对于任给的布尔变量 x_1, \dots, x_k 的0, 1赋值, 容易在多项式时间内验证相应的 $A_1 A_2 \dots A_m$ 的取值是否均为1。因此, SAT \in NP。

现在只要证明对任意的 $L \in \text{NP}$ 有 $L \leq_p \text{SAT}$ 即可。

(详细证明见书本P307 ~ 310)

8.4 一些典型的NP完全问题



部分NP完全问题树

8.4.1 合取范式的可满足性问题 (CNF-SAT)

问题描述： 给定一个合取范式 α ，判定它是否可满足。

如果一个布尔表达式是一些因子和之积，则称之为合取范式，简称CNF (Conjunctive Normal Form)。这里的因子是变量 x 或 \bar{x} 。例如： $(x_1 + x_2)(x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + x_3)$ 就是一个合取范式，而 $x_1x_2 + x_3$ 就不是合取范式。

要证明CNF-SAT \in NPC，只要证明在Cook定理中定义的布尔表达式A，...，G或者已是合取范式，或者有的虽然不是合取范式，但可以用布尔代数中的变换方法将它们化成合取范式，而且合取范式的长度与原表达式的长度只差一个常数因子。

8.4.2 3元合取范式的可满足性问题 (3-SAT)

问题描述： 给定一个3元合取范式 α ，判定它是否可满足。

证明思路：

3-SAT \in NP是显而易见的。为了证明3-SAT \in NPC，只要证明CNF-SAT \propto_p 3-SAT，即合取范式的可满足性问题可在多项式时间内变换为3-SAT。

8.4.3 团问题CLIQUE

问题描述： 给定一个无向图 $G=(V, E)$ 和一个正整数 k ，判定图 G 是否包含一个 k 团，即是否存在， $V' \subseteq V$ ， $|V'|=k$ ，且对任意 $u, w \in V'$ 有 $(u, w) \in E$ 。

证明思路：

已经知道 $\text{CLIQUE} \in \text{NP}$ 。通过 $3\text{-SAT} \leq_p \text{CLIQUE}$ 来证明 CLIQUE 是 NP 难的，从而证明团问题是 NP 完全的。

8.4.4 顶点覆盖问题 (VERTEX-COVER)

问题描述: 给定一个无向图 $G=(V, E)$ 和一个正整数 k , 判定是否存在 $V' \subseteq V$, $|V'|=k$, 使得对于任意 $(u, v) \in E$ 有 $u \in V'$ 或 $v \in V'$ 。如果存在这样的 V' , 就称 V' 为图 G 的一个大小为 k 顶点覆盖。

证明思路:

首先, $\text{VERTEX-COVER} \in \text{NP}$ 。因为对于给定的图 G 和正整数 k 以及一个“证书” V' , 验证 $|V'|=k$, 然后对每条边 $(u, v) \in E$, 检查是否有 $u \in V'$ 或 $v \in V'$, 显然可在多项式时间内完成。

其次, 通过 $\text{CLIQUE} \propto_p \text{VERTEX-COVER}$ 来证明顶点覆盖问题是NP难的。

8.4.5 子集和问题

(SUBSET-SUM)

问题描述: 给定整数集合 S 和一个整数 t , 判定是否存在 S 的一个子集 $S' \subseteq S$, 使得 S' 中整数的和为 t 。例如, 若 $S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ 且 $t = 3754$, 则子集 $S' = \{1, 16, 64, 256, 1040, 1093, 1284\}$ 是一个解。

证明思路:

首先, 对于子集和问题的一个实例 $\langle S, t \rangle$, 给定一个“证书” S' , 要验证 $t = \sum_{i \in S'} i$ 是否成立, 显然可在多项式时间内完成。因此, $\text{SUBSET-SUM} \in \text{NP}$;

其次, 证明 $\text{VERTEX-COVER} \propto_p \text{SUBSET-SUM}$ 。

8.4.6 哈密顿回路问题 (HAM-CYCLE)

问题描述： 给定无向图 $G=(V, E)$ ，判定其是否含有一哈密顿回路。

证明思路：

首先，已知哈密顿回路问题是一个NP类问题。

其次，通过证明 $3\text{-SAT} \propto_p \text{HAM-CYCLE}$,

得出： $\text{HAM-CYCLE} \in \text{NPC}$ 。

8.4.7 旅行售货员问题TSP

问题描述: 给定一个无向完全图 $G=(V, E)$ 及定义在 $V \times V$ 上的一个费用函数 c 和一个整数 k , 判定 G 是否存在经过 V 中各顶点恰好一次的回路, 使得该回路的费用不超过 k 。

首先, 给定TSP的一个实例 (G, c, k) , 和一个由 n 个顶点组成的顶点序列。验证算法要验证这 n 个顶点组成的序列是图 G 的一条回路, 且经过每个顶点一次。另外, 将每条边的费用加起来, 并验证所得的和不超过 k 。这个过程显然可在多项式时间内完成, 即 $TSP \in NP$ 。

其次, 旅行售货员问题与哈密顿回路问题有着密切的联系。哈密顿回路问题可在多项式时间内变换为旅行售货员问题。即 $HAM-CYCLE \propto_p TSP$ 。从而, 旅行售货员问题是NP难的。

因此, $TSP \in NPC$ 。

第9章 近似算法

第9章 近似算法

迄今为止，所有的NP完全问题都还没有多项式时间算法。对于这类问题，通常可采取以下几种解题策略。

- (1) 只对问题的特殊实例求解
- (2) 用动态规划法或分支限界法求解
- (3) 用概率算法求解
- (4) 只求近似解
- (5) 用启发式方法求解

本章主要讨论解NP完全问题的近似算法。

9.1 近似算法的性能

若一个最优化问题的最优值为 c^* ，求解该问题的一个近似算法求得的近似最优解相应的目标函数值为 c ，则将**该近似算法的性能比**定义为 $\eta = \max \left\{ \frac{c}{c^*}, \frac{c^*}{c} \right\}$ 。在通常情况下，该性能比是问题输入规模 n 的一个函数 $\rho(n)$ ，即

$$\max \left\{ \frac{c}{c^*}, \frac{c^*}{c} \right\} \leq \rho(n)。$$

该**近似算法的相对误差**定义为 $\lambda = \left| \frac{c - c^*}{c^*} \right|$ 。若对问题的输入规模 n ，有一函数 $\varepsilon(n)$ 使得 $\left| \frac{c - c^*}{c^*} \right| \leq \varepsilon(n)$ ，则称 $\varepsilon(n)$ 为该**近似算法的相对误差界**。近似算法的性能比 $\rho(n)$ 与相对误差界 $\varepsilon(n)$ 之间显然有如下关系： $\varepsilon(n) \leq \rho(n) - 1$ 。

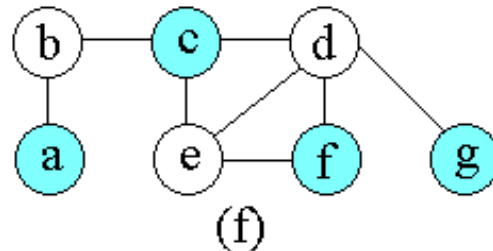
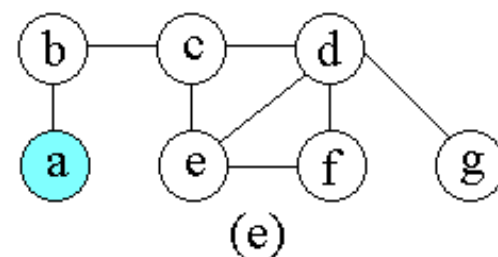
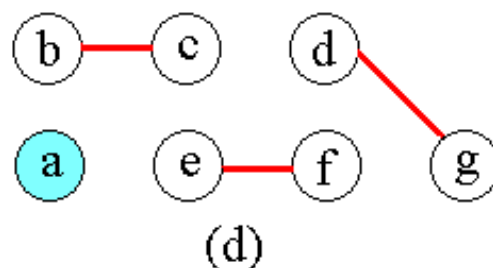
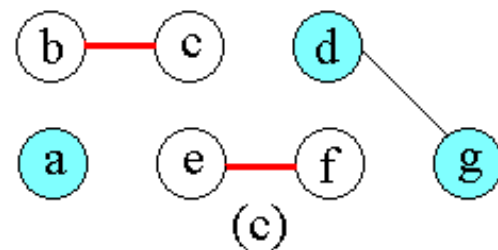
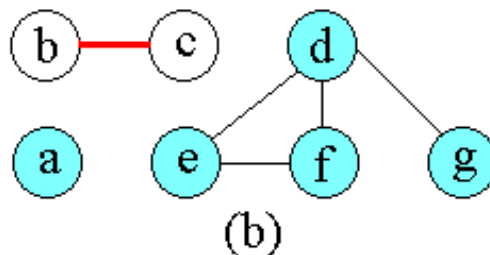
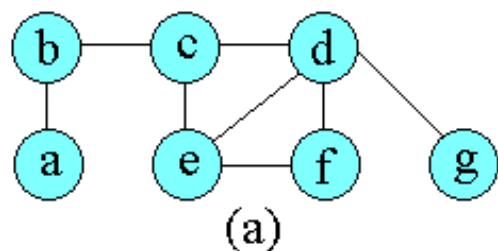
9.2 顶点覆盖问题的近似算法

问题描述：无向图 $G=(V, E)$ 的顶点覆盖是它的顶点集 V 的一个子集 $V' \subseteq V$ ，使得若 (u, v) 是 G 的一条边，则 $v \in V'$ 或 $u \in V'$ 。顶点覆盖 V' 的大小是它所包含的顶点个数 $|V'|$ 。

```
VertexSet approxVertexCover ( Graph g )  
{  
    cset= $\emptyset$ ;  
    e1=g.e;  
    while (e1  $\neq \emptyset$ ) {  
        从e1中任取一条边(u, v);  
        cset=cset  $\cup$  {u, v};  
        从e1中删去与u和v相关联的所有边;  
    }  
    return c  
}
```

Cset用来存储顶点覆盖中的各顶点。初始为空，不断从边集e1中选取一边 (u, v) ，将边的端点加入cset中，并将e1中已被u和v覆盖的边删去，直至cset已覆盖所有边。即e1为空。

9.2 顶点覆盖问题的近似算法



图(a)~(e)说明了算法的运行过程及结果。(e)表示算法产生的近似最优顶点覆盖cset，它由顶点b, c, d, e, f, g所组成。(f)是图G的一个最小顶点覆盖，它只含有3个顶点：b, d和e。

算法approxVertexCover的性能比为2。

9.3 旅行售货员问题近似算法

问题描述：给定一个完全无向图 $G=(V, E)$ ，其每一边 $(u, v) \in E$ 有一非负整数费用 $c(u, v)$ 。要找出 G 的最小费用哈密顿回路。

旅行售货员问题的一些特殊性质：

比如，费用函数 c 往往具有三角不等式性质，即对任意的3个顶点 $u, v, w \in V$ ，有： $c(u, w) \leq c(u, v) + c(v, w)$ 。当图 G 中的顶点就是平面上的点，任意2顶点间的费用就是这2点间的欧氏距离时，费用函数 c 就具有三角不等式性质。

9.3.1 具有三角不等式性质的旅行售货员问题

对于给定的无向图G，可以利用找图G的最小生成树的算法设计找近似最优的旅行售货员回路的算法。

```
void approxTSP (Graph g)
```

```
{
```

```
    (1) 选择g的任一顶点r;
```

```
    (2) 用Prim算法找出带权图g的一棵以r为根的最小生成树T;
```

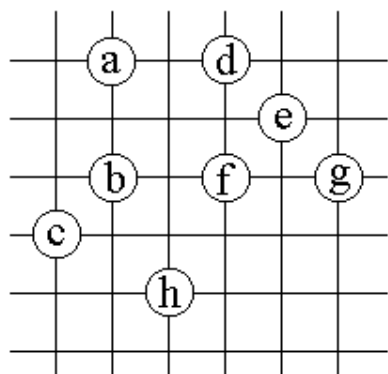
```
    (3) 前序遍历树T得到的顶点表L;
```

```
    (4) 将r加到表L的末尾，按表L中顶点次序组成回路H，作为计算结果返回;
```

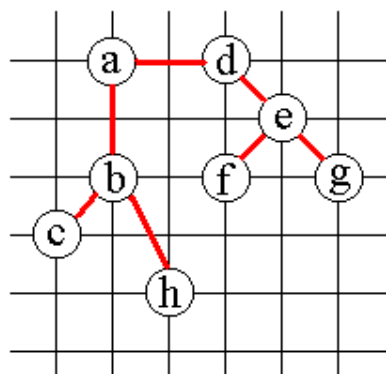
```
}
```

当费用函数满足三角不等式时，算法找出的旅行售货员回路的费用不会超过最优旅行售货员回路费用的2倍。

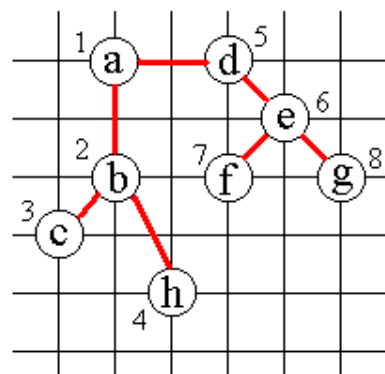
9.3.1 具有三角不等式性质的旅行售货员问题举例



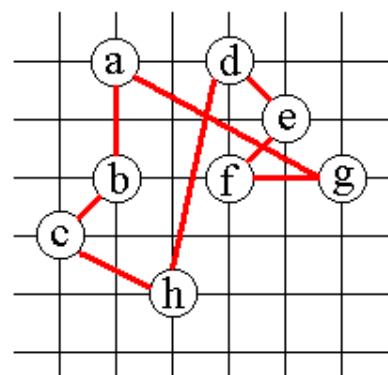
(a)



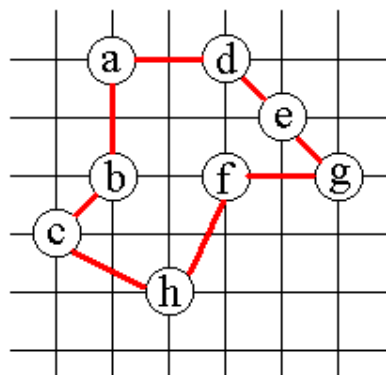
(b)



(c)



(d)



(e)

(b) 表示找到的最小生成树T; (c) 表示对T作前序遍历的次序; (d) 表示L产生的哈密顿回路H; (e) 是G的一个最小费用旅行售货员回路。

9.3.2 一般的旅行售货员问题

在费用函数不一定满足三角不等式的一般情况下，不存在具有常数性能比的解TSP问题的多项式时间近似算法，除非 $P=NP$ 。换句话说，若 $P \neq NP$ ，则对任意常数 $\rho > 1$ ，不存在性能比为 ρ 的解旅行售货员问题的多项式时间近似算法。

9.4 集合覆盖问题的近似算法

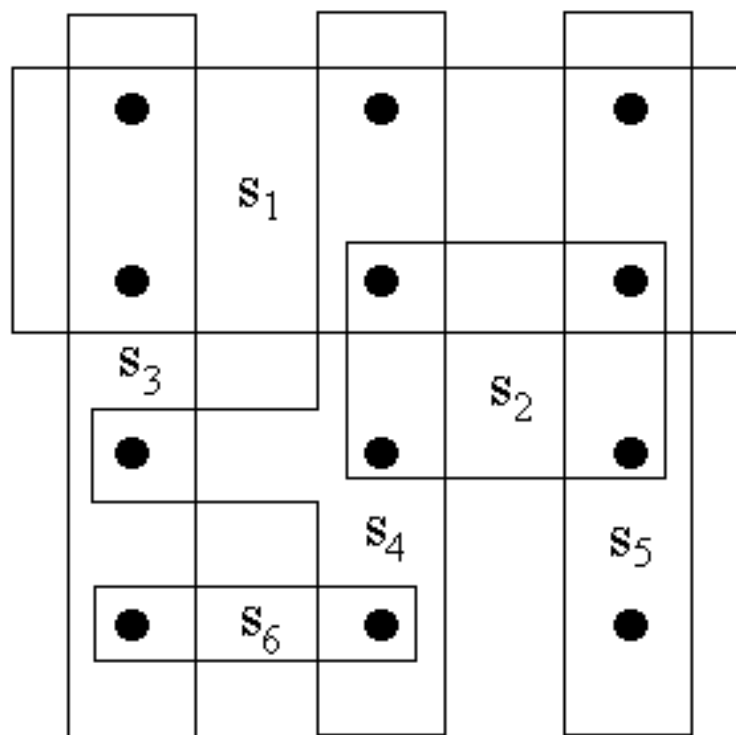
问题描述：给定一个完全无向图 $G=(V, E)$ ，其每一边 $(u, v) \in E$ 有一非负整数费用 $c(u, v)$ 。要找出 G 的最小费用哈密顿回路。

集合覆盖问题的一个实例 $\langle X, F \rangle$ 由一个有限集 X 及 X 的一个子集族 F 组成。子集族 F 覆盖了有限集 X 。也就是说 X 中每一元素至少属于 F 中的一个子集，即 $X = \bigcup_{S \in F} S$ 。对于 F 中的一个子集 $C \subseteq F$ ，若 C 中的 X 的子集覆盖了 X ，即 $X = \bigcup_{S \in C} S$ ，则称 C 覆盖了 X 。集合覆盖问题就是要找出 F 中覆盖 X 的最小子集 C^* ，使得

$$|C^*| = \min \{ |C| \mid C \subseteq F \text{ 且 } C \text{ 覆盖 } X \}$$

9.4 集合覆盖问题的近似算法

集合覆盖问题举例：



用12个黑点表示集合X。

$F = \{S_1, S_2, S_3, S_4, S_5, S_6, \}$ ，如图所示。

容易看出，对于这个例子，最小集合覆盖为：

$C = \{S_3, S_4, S_5, \}$ 。

9.4 集合覆盖问题的近似算法

集合覆盖问题近似算法——贪心算法

```
Set greedySetCover (X, F)
{
    U=X;
    C=∅;
    while (U !=∅) {
        选择F中使 $|S \cap U|$ 最大的子集S;
        U=U-S;
        C=C ∪ {S};
    }
    return C;
}
```

算法的循环体最多执行 $\min\{|X|, |F|\}$ 次。而循环体内的计算显然可在 $O(|X||F|)$ 时间内完成。因此，算法的计算时间为 $O(|X||F|\min\{|X|, |F|\})$ 。由此即知，该算法是一个多项式时间算法。

9.5 子集合问题的近似算法

问题描述：设子集问题的一个实例为 $\langle S, t \rangle$ 。其中， $S = \{x_1, x_2, \dots, x_n\}$ 是一个正整数的集合， t 是一个正整数。子集和问题判定是否存在 S 的一个子集 S_1 ，使得 $\sum_{x \in S_1} x = t$ 。

9.5.1 子集合问题的指数时间算法

```
int exactSubsetSum (S, t)
{
    int n=|S|;
    L[0]={0};
    for (int i=1; i<=n; i++) {
        L[i]=mergeLists (L[i-1], L[i-1]+S[i]);
        删去L[i]中超过t的元素;
    }
    return max (L[n]);
}
```

算法以集合 $S=\{x_1, x_2, \dots, x_n\}$ 和目标值 t 作为输入。算法中用到将2个有序表 $L1$ 和 $L2$ 合并成为一个新的有序表的算法 $\text{mergeLists}(L1, L2)$ 。

9.5.2 子集合问题的完全多项式时间近似格式

基于算法exactSubsetSum, 通过对表 $L[i]$ 作适当的修整建立一个子集和问题的完全多项式时间近似格式。

在对表 $L[i]$ 进行修整时, 用到一个修整参数 δ , $0 < \delta < 1$ 。用参数 δ 修整一个表 L 是指从 L 中删去尽可能多的元素, 使得每一个从 L 中删去的元素 y , 都有一个修整后的表 L_1 中的元素 z 满足 $(1-\delta)y \leq z \leq y$ 。可以将 z 看作是被删去元素 y 在修整后的新表 L_1 中的代表。

举例: 若 $\delta = 0.1$, 且 $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$, 则用 δ 对 L 进行修整后得到 $L_1 = \langle 10, 12, 15, 20, 23, 29 \rangle$ 。其中被删去的数11由10来代表, 21和22由20来代表, 24由23来代表。

9.5.2 子集合问题的完全多项式 时间近似格式

对有序表L修整算法

```
List trim(L,  $\delta$ )
{
    int m=|L|;
    L1=  $\langle L[1] \rangle$  ;
    int last=L[1];
    for (int i=2; i<=m; i++) {
        if (last<(1- $\delta$ )*L[i]) {
            将L[i]加入表L1的尾部;
            last=L[i];
        }
    }
    return L1;
}
```

子集和问题近似格式

```
int approxSubsetSum(S, t,  $\epsilon$ )
{
    n=|S|;
    L[0]=  $\langle 0 \rangle$  ;
    for (int i=1; i<=n; i++) {
        L[i]=Merge-Lists(L[i-1],
                        L[i-1]+S[i]);
        L[i]=Trim(L[i],  $\epsilon/n$ );
        删去L[i]中超过t的元素;
    }
    return max(L[n]);
}
```

第10章 算法优化策略

算法设计策略的比较与选择

给定由 n 个整数(可能为负整数)组成的序列 a_1, a_2, \dots, a_n , 求该序列形如 $\sum_{k=i}^j a_k$ 的子段和的最大值。当所有整数均为负整数时定义其最大子段和为 0。依此定义, 所求的最优值为:

$$\max \left\{ 0, \max_{1 \leq i \leq j \leq n} \sum_{k=i}^j a_k \right\}$$

例如:

$$A = (-2, 11, -4, 13, -5, -2)$$

$$\text{最大子段和为 } \sum_{k=2}^4 a_k = 20$$

简单算法

```
public static int maxSum()  
{  
    int n=a.length-1;  
    int sum=0;  
    for (int i=1;i<=n;i++) {  
        int thissum=0;  
        for (int j=i;j<=n;j++) {  
            thissum+=a[j];  
            if (thissum>sum) {  
                sum=thissum;  
                besti=i;  
                bestj=j;  
            }  
        }  
    }  
    return sum;  
}
```

注意到 $\sum_{k=i}^j a_k = a_j + \sum_{k=i}^{j-1} a_k$ ，则可将算法中的最后一个for循环省去，避免重复计算只需要 $O(n^2)$ 的计算时间。

分治算法

如果将所给的序列 $a[1:n]$ 分为长度相等的2段 $a[1:n/2]$ 和 $a[n/2+1:n]$ ，分别求出这2段的最大子段和，则 $a[1:n]$ 的最大子段和有2种情况

(1) **复杂度分析**

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

$$(3) a \quad T(n) = O(n \log n)$$

同；

对于情形(3)。容易看出， $a[n/2]$ 与 $a[n/2+1]$ 在最优子序列中。因此，可以在 $a[1:n/2]$ 中计算出 $s1 = \max_{1 \leq i \leq n/2} \sum_{k=i}^{n/2} a[k]$ ，并在 $a[n/2+1:n]$ 中计算出 $s2 = \max_{n/2+1 \leq i \leq n} \sum_{k=n/2+1}^i a[k]$ 。则 $s1 + s2$ 即为出现情形(3)时的最优值。据此可设计出求最大子段和的分治算法。

动态规划算法

记 $b[j] = \max_{1 \leq i \leq j} \{ \sum_{k=i}^j a[k] \}$ $1 \leq j \leq n$, 则所求的最大子段和为

$$\max_{1 \leq i \leq j \leq n} \sum_{k=i}^j a[k] = \max_{1 \leq j \leq n} \max_{1 \leq i \leq j} \sum_{k=i}^j a[k] = \max_{1 \leq j \leq n} b[j]$$

当 $b[j-1] > 0$ 时 $b[j] = b[j-1] + a[j]$, 否则 $b[j] = a[j]$ 。由此可得计算 $b[j]$ 的动态规划递归式

$$b[j] = \max\{b[j-1] + a[j], a[j]\}, \quad 1 \leq j \leq n$$

```
public static int maxSum()
```

```
{  
    int n=a.length-1;  
    int sum=0,  
        b=0;  
    for (int i=1;i<=n;i++) {  
        if (b>0) b+=a[i];  
        else b=a[i];  
        if (b>sum)sum=b;  
    }  
    return sum;  
}
```

算法显然需要 $O(n)$ 计算时间和 $O(n)$ 空间。

最大子矩阵和问题

给定一个 m 行 n 列的整数矩阵 a ，试求矩阵 a 的一个子矩阵，使其各元素之和为最大。

$$\text{记 } s(i1, i2, j1, j2) = \sum_{i=i1}^{i2} \sum_{j=j1}^{j2} a[i][j]$$

最大子矩阵和问题的最优值为 $\max_{\substack{1 \leq i1 \leq i2 \leq m \\ 1 \leq j1 \leq j2 \leq n}} s(i1, i2, j1, j2)$

$$\text{由于 } \max_{\substack{1 \leq i1 \leq i2 \leq m \\ 1 \leq j1 \leq j2 \leq n}} s(i1, i2, j1, j2) = \max_{1 \leq i1 \leq i2 \leq m} \{ \max_{1 \leq j1 \leq j2 \leq n} s(i1, i2, j1, j2) \} = \max_{1 \leq i1 \leq i2 \leq m} t(i1, i2)$$

$$\text{其中, } t(i1, i2) = \max_{1 \leq j1 \leq j2 \leq n} s(i1, i2, j1, j2) = \max_{1 \leq j1 \leq j2 \leq n} \sum_{j=j1}^{j2} \sum_{i=i1}^{i2} a[i][j]$$

$$\text{设 } b[j] = \sum_{i=i1}^{i2} a[i][j], \text{ 则 } t(i1, i2) = \max_{1 \leq j1 \leq j2 \leq n} \sum_{j=j1}^{j2} b[j]$$

由于解最大子段和问题的动态规划算法需要时间 $O(n)$ ，故算法的双重for循环需要计算时间 $O(m^2n)$ 。

最大m子段和问题

给定由n个整数(可能为负整数)组成的序列 a_1, a_2, \dots, a_n , 以及一个正整数m, 要求确定序列的m个不相交子段, 使这m个子段的总和达到最大。

设 $b(i, j)$ 表示数组a的前j项中i个子段和的最大值, 且第i个子段含 $a[j]$ ($1 \leq i \leq m, 1 \leq j \leq n$)。则所求的最优值显然为 $\max_{m \leq j \leq n} b(m, j)$

与最大子段和问题类似地, 计算 $b(i, j)$ 的递归式为

$$b(i, j) = \max \{b(i, j-1) + a[j], \max_{i-1 \leq t < j} b(i-1, t) + a[j]\} \quad (1 \leq i \leq m, 1 \leq j \leq n)$$

初始时, $b(0, j)=0, (1 \leq j \leq n); b(i, 0)=0, (1 \leq i \leq m)$ 。

优化: 注意到在上述算法中, 计算 $b[i][j]$ 时只用到数组b的第i-1行和第i行的值。因而算法中只要存储数组b的当前行, 不必存储整个数组。另一方面, $b(i-1, t)$ 的值可以在计算第i-1行时预先计算并保存起来。计算第i行的值时不必重新计算, 节省了计算时间和空间。

动态规划加速原理

四边形不等式

货物储运问题

在一个铁路沿线顺序存放着 n 堆装满货物的集装箱。货物储运公司要将集装箱有次序地集中成一堆。规定每次只能选相邻的2堆集装箱合并成新的一堆，所需的运输费用与新的一堆中集装箱数成正比。给定各堆的集装箱数，试制定一个运输方案，使总运输费用最少。

设合并 $a[i:j]$ ， $1 \leq i \leq j \leq n$ ，所需的最少费用为 $m[i,j]$ ，则原问题的最优值为 $m[1, n]$ 。由最优子结构性质可知，

$$m[i, j] = \begin{cases} 0 & i = j \\ \min_{i < k \leq j} \{m[i, k-1] + m[k, j] + \sum_{t=i}^j a[t]\} & i < j \end{cases}$$

根据递归式，按通常方法可设计计算 $m(i,j)$ 的 $O(n^3)$ 动态规划算法

四边形不等式

货物储运问题的动态规划递归式是下面更一般的递归计算式的特殊情形。

$$m[i, j] = \begin{cases} 0 & i = j \\ w(i, j) + \min_{i < k \leq j} \{m[i, k-1] + m[k, j]\} & i < j \end{cases}$$

对于 $i \leq i' < j \leq j'$, 当函数 $w(i, j)$ 满足

$$w(i, j) + w(i', j') \leq w(i', j) + w(i, j')$$

时称 w 满足 **四边形不等式**。

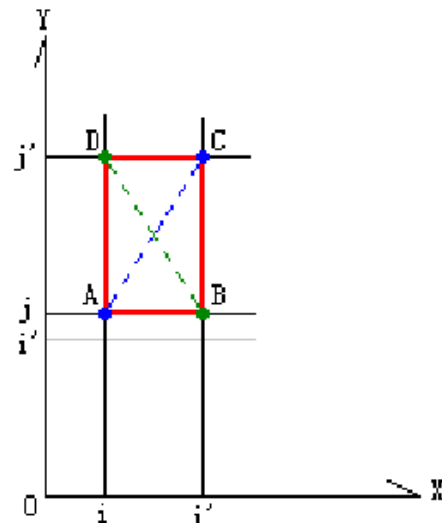
当函数 $w(i, j)$ 满足

$$w(i', j) \leq w(i, j')$$

时称 w **关于区间包含关系单调**

对于满足四边形不等式的单调函数 w , 可推知由递归式定义的函数 $m(i, j)$ 也满足四边形不等式, 即

$$m(i, j) + m(i', j') \leq m(i', j) + m(i, j')$$



定义 $s(i, j) = \max\{k \mid m(i, j) = m(i, k-1) + m(k, j) + w(i, j)\}$

由函数 $m(i, j)$ 的四边形不等式性质可推出函数 $s(i, j)$ 的单调性，即

$$s(i, j) \leq s(i, j+1) \leq s(i+1, j+1), \quad i \leq j$$

根据前面的讨论，当 w 是满足四边形不等式的单调函数时，函数 $s(i, j)$ 单调，从而

$$\min_{i < k \leq j} \{m(i, k-1) + m(k, j)\} = \min_{s(i, j-1) \leq k \leq s(i+1, j)} \{m(i, k-1) + m(k, j)\}$$

改进后算法 **speedDynamicProgramming** 所需的计算时间为

$$\begin{aligned} & O\left(\sum_{r=0}^{n-1} \sum_{i=1}^{n-r} (1 + s(i+1, i+r) - s(i, i+r-1))\right) \\ &= O\left(\sum_{r=0}^{n-1} (n-r + s(n-r, n) - s(1, r))\right) \\ &= O\left(\sum_{r=0}^{n-1} n\right) \\ &= O(n^2) \end{aligned}$$

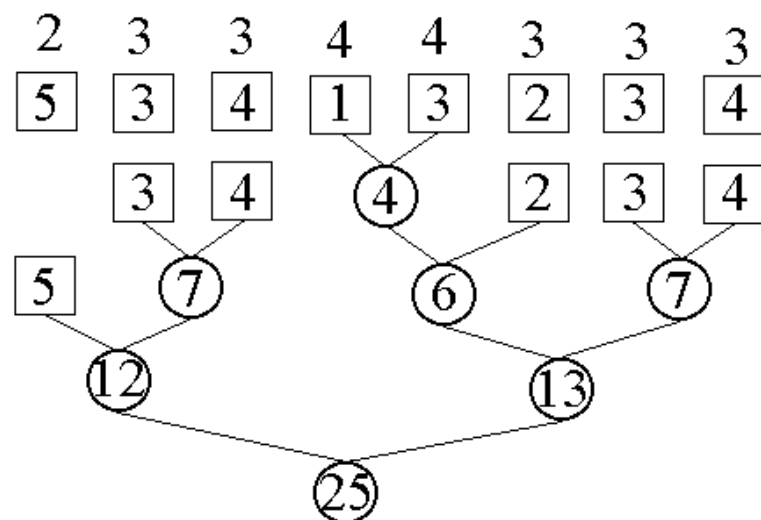
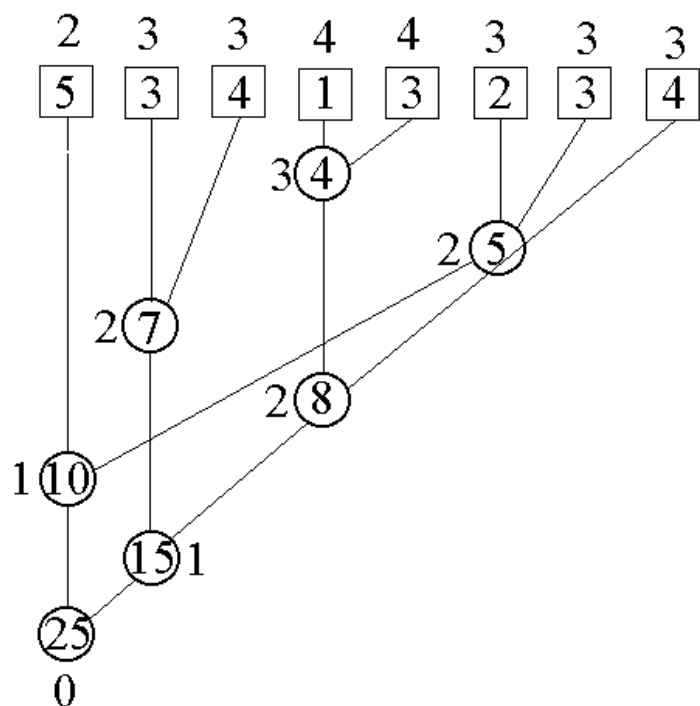
问题的算法特征

-

相同层序定理

相同层序定理：存在货物储运问题的最优合并树，其各原始结点在最优合并树中所处的层序与相应的原始结点在相容合并树中所处的层序相同。

根据上述定理，容易从各原始结点在相容合并树中所处的层序构造出相应的最优合并树，如图所示。

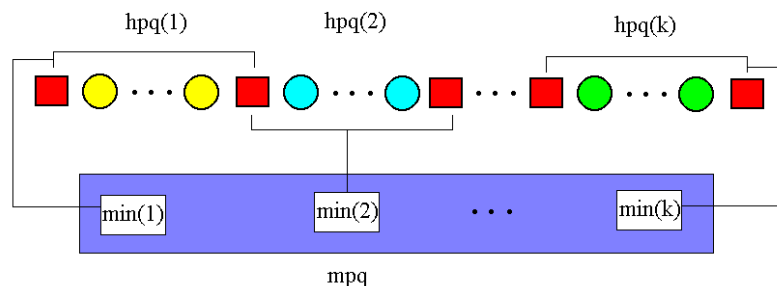


$$4+7+6+7+12+13+25=74$$

1. 组合阶段: 将给定的 n 个数作为方形结点依序从左到右排列, $a[1], a[2], \dots, a[n]$ 。反复删除序列中最小相容结点对 $a[i]$ 和 $a[j]$, ($i < j$), 并在位置 i 处插入值为 $a[i] + a[j]$ 的圆形结点, 直至序列中只剩下1个结点。 $O(n \log n)$
2. 标记层序阶段: 将第一阶段结束后留下的惟一结点标记为第0层结点。然后以与第一阶段相反的组合顺序标记其余结点的层序。 $O(n)$
3. 重组阶段: 根据标记层序阶段计算出的各结点的层序, 按下述规则重组。 $O(n)$

结点 $a[i]$ 和 $a[j]$ 重组为新结点应满足:

- (1) $a[i]$ 和 $a[j]$ 在当前序列中相邻;
- (2) $a[i]$ 和 $a[j]$ 均为当前序列中最大层序结点;
- (3) 在所有满足条件(1)和(2)的结点中, 下标 i 最小。



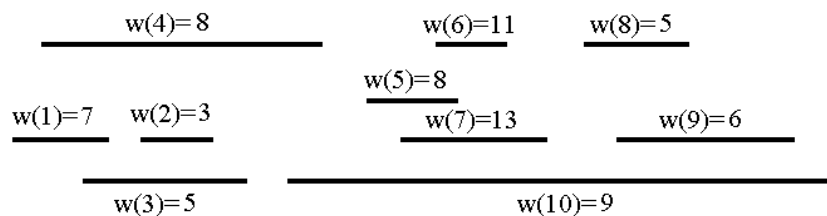
优化数据结构

带权区间最短路问题

S 是直线上 n 个带权区间的集合。从区间 $I \in S$ 到区间 $J \in S$ 的一条路是 S 的一个区间序列 $J(1), J(2), \dots, J(k)$, 其中 $J(1) = I, J(k) = J$, 且对所有 $1 \leq i \leq k-1$, $J(i)$ 与 $J(i+1)$ 相交。这条路的长度定义为路上各区间权之和。在所有从 I 到 J 的路中, 路长最短的路称为从 I 到 J 的最短路。带权区间图的单源最短路问题要求计算从 S 中一个特定的源区间到 S 中所有其他区间之间的最短路。

区间集 $S(i)$ 的扩展定义为: $S(i) \cup T$, 其中 T 是满足下面条件的另一区间集。 T 中任意区间 $I=[a,b]$ 均有 $b > b(i)$ 。

设区间 $I(k)$ ($k \leq i$) 是区间集 $S(i)$ 中的一个区间, $1 \leq i \leq n$ 。如果对于 $S(i)$ 的任意扩展 $S(i) \cup T$, 当区间 $J \in T$ 且在 $S(i) \cup T$ 中有从 $I(1)$ 到 J 的路时, 在 $S(i) \cup T$ 中从 $I(1)$ 到 J 的任一最短路都不含区间 $I(k)$, 则称区间 $I(k)$ 是 $S(i)$ 中的无效区间。若 $S(i)$ 中的区间 $I(k)$ 不是无效区间则称其为 $S(i)$ 中的有效区间。



带权区间最短路问题

性质1: 区间 $I(k)$ 是 $S(i)$ 中的有效区间, 则对任意 $k \leq j \leq i$, 区间 $I(k)$ 是 $S(j)$ 中的有效区间。另一方面, 若区间 $I(k)$ 是 $S(i)$ 中的无效区间, 则对任意 $j > i$, 区间 $I(k)$ 是 $S(j)$ 中的无效区间。

性质2: 集合 $S(i)$ 中所有有效区间的并覆盖从 $a(1)$ 到 $b(j)$ 的线段, 其中 $b(j)$ 是 $S(i)$ 的最右有效区间的右端点。

性质3: 区间 $I(i)$ 是集合 $S(i)$ 中的有效区间当且仅当在 $S(i)$ 中有一条从 $I(1)$ 到 $I(i)$ 的路。

性质4: 当 $i > k$ 且 $\text{dist}(i, i) < \text{dist}(k, i)$ 时, $I(k)$ 是 $S(i)$ 中的无效区间。

性质5: 设 $I(j(1)), I(j(2)), \dots, I(j(k))$ 是 $S(i)$ 中的有效区间, 且 $j(1) < j(2) < \dots < j(k) \leq i$, 则 $\text{dist}(j(1), i) \leq \text{dist}(j(2), i) \leq \dots \leq \text{dist}(j(k), i)$ 。

性质6: 如果区间 $I(i)$ 包含区间 $I(k)$ (因此 $i > k$), 且 $\text{dist}(i, i) < \text{dist}(k, i)$, 则 $I(k)$ 是 $S(i)$ 中的无效区间。

性质7: 当 $i > k$ 且 $\text{dist}(i, i) < \text{dist}(k, i-1)$ 时, $I(k)$ 是 $S(i)$ 中的无效区间。

性质8: 如果区间 $I(k)$ ($k > 1$) 不包含 $S(k-1)$ 中任一有效区间 $I(j)$ 的右端点 $b(j)$, 则对任意 $i \geq k$, $I(k)$ 是 $S(i)$ 中的无效区间。

带权区间图的最短路算法

算法 **shortestIntervalPaths**

步骤1: $\text{dist}(1,1) \leftarrow w(1);$

步骤2:

for ($i=2; i \leq n; i++$) {

(2.1):

$j = \min\{ k \mid a(i) < b(k); 1 \leq k < i \};$

if (j 不存在) $\text{dist}(i,i) \leftarrow +\infty;$

else $\text{dist}(i,i) \leftarrow \text{dist}(j,i-1) + w(i);$

(2.2):

for ($k < i$) {

if ($\text{dist}(i,i) < \text{dist}(k,i-1)$) $\text{dist}(k,i) \leftarrow +\infty;$

else $\text{dist}(k,i) \leftarrow \text{dist}(k,i-1);$

}

}

步骤3:

for ($i=2; i \leq n; i++$) {

if ($\text{dist}(i,n) = +\infty$) {

$j = \min\{ k \mid$

$(\text{dist}(k,n) < +\infty) \&\& (a(i) < b(k)) \};$

$\text{dist}(i,n) = \text{dist}(j,n) + w(i);$

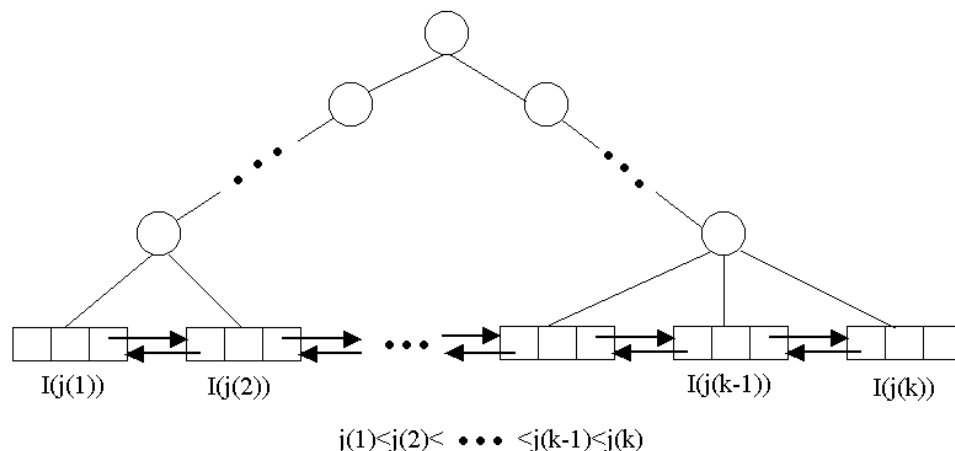
}

}

上述算法的关键是有效地实现步骤(2.1)和(2.2)

实现方案1

用一棵平衡搜索树（2-3树）存储当前区间集 $S(i)$ 中的有效区间。以区间的右端点的值为序。如图所示。



(2.1)的实现对应于平衡搜索树从根到叶的一条路径上的搜索，在最坏情况下需要时间 $O(\log n)$ 。

(2.2)的实现对应于反复检查并删除平衡搜索树中最右叶结点的前驱结点。在最坏情况下，每删除一个结点需要时间 $O(\log n)$ 。

综上，算法**shortestIntervalPaths**用平衡搜索树的实现方案，在最坏情况下的计算时间复杂性为 $O(n \log n)$ 。

实现方案2

采用并查集结构。用整数 k 表示区间 $I(k)$, $1 \leq k \leq n$ 。初始时每个元素 k 构成一个单元元素集, 即集合 k 是 $\{k\}$, $1 \leq k \leq n$ 。

(1) 每个当前有效区间 $I(k)$ 在集合 k 中。

(2) 对每个集合 $S(i)$, 设

$L(S(i)) = \{I(k) \mid I(k) \text{ 是 } S(i) \text{ 的无效区间, 且 } I(k) \text{ 与 } S(i) \text{ 的任一有效区间均不相交}\}$, $L(S(i))$ 中所有区间均位于 $S(i)$ 的所有有效区间并的右侧。

(3) 用一个栈 AS 存放当前有效区间 $I(i(1))$, $I(i(2))$, ..., $I(i(k))$ 。

$I(i(k))$ 是栈顶元素。该栈称为当前有效区间栈。

(4) 对于 $1 \leq k \leq n$, 记 $\text{prev}(I(k)) = \min\{j \mid a(k) < b(j)\}$ 。对给定的区间序列做一次线性扫描确定 $\text{prev}(I(k))$ 的值。

(5) 对于当前区间集 $S(i)$, 用一维数组 dist 记录 $\text{dist}(j, i)$ 的值。

(6) 用 $\text{dist}[k] = -1$ 标记区间 $I(k)$ 为无效区间。

在最坏情况下, 算法需要 $O(n\alpha(n))$ 计算时间, 其中 $\alpha(n)$ 是单变量Ackerman函数的逆函数

优化搜索策略

最短加法链问题

给定一个正整数和一个实数，如何用最少的乘法次数计算出 x^n 。

例如，可以用6次乘法逐步计算 x^{23} 如下： $x, x^2, x^3, x^5, x^{10}, x^{20}, x^{23}$

可以证明计算最少需要6次乘法。计算的幂序列中各幂次1, 2, 3, 5, 10, 20, 23组成了一个关于整数23的加法链。在一般情况下，计算 x^n 的幂序列中各幂次组成正整数的一个加法链

$$1 = a_0 < a_1 < a_2 < \dots < a_r = n$$

$$a_i = a_j + a_k, k \leq j < i, i = 1, 2, \dots, r$$

上述最优求幂问题相应于正整数的最短加法链问题，即求 n 的一个加法链使其长度达到最小。正整数的最短加法链长度记为 $l(n)$ 。

回溯法

问题的状态空间树如图所示。其中第 i 层结点 a_i 的儿子结点 $a_{i+1} > a_i$ 由 $a_j + a_k$, $k \leq j \leq i$ 所构成。

```
private static void backtrack(int step)
```

```
{// 解最短加法链问题的标准回溯法
```

```
int i,j,k;
```

```
if (a[step]==n) // 找到一条加法链
```

```
{
```

```
    if (step<best) 更新最优值
```

```
    return;
```

```
}
```

```
// 对当前结点a[step]的每一个儿子结点递归搜索
```

```
for (i=step;i>=1;i--)
```

```
    if (2*a[i]>a[step])
```

```
        for (j=i;j>=1;j--){
```

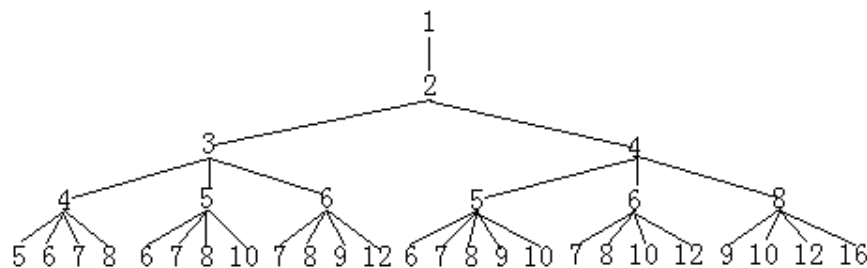
```
            k=a[i]+a[j];
```

```
            a[step+1]=k;
```

```
            if ((k>a[step])&&(k<=n)) backtrack(step+1);
```

```
        }
```

```
}
```



由于加法链问题的状态空间树的每一个第 k 层结点至少有 $k+1$ 个儿子结点，因此从根结点到第 k 层的任一结点的路径数至少是 $k!$ 。用标准的回溯法只能对较小的构造出最短加法链。

- 深度优先搜索: 算法所搜索到的第一个加法链不一定是最短加法链。
- 广度优先搜索: 算法找到的第一个加法链就是最短加法链, 但这种方法的空间开销太大。
- 迭代搜索算法: 既能保证算法找到的第一个加法链就是最短加法链, 又不需要太大的空间开销。其基本思想是控制回溯法的搜索深度 d , 从 $d=1$ 开始搜索, 每次搜索后使 d 增1, 加深搜索深度, 直到找到一条加法链为止。

```
private static void iterativeDeepening()
```

```
{// 逐步深化的迭代搜索算法
```

```
    best=n+1;
```

```
    found=false;
```

```
    lb=2; // 初始迭代搜索深度
```

```
    while (!found){
```

```
        a[1]=1;
```

```
        backtrack(1);
```

```
        lb++; // 加深搜索深度
```

```
    }
```

```
}
```

对于正整数, 记 $\lambda(n)=\lfloor \log n \rfloor$,
 $v(n)=n$ 的2进制表示中1的个数。
迄今为止所知道的 $l(n)$ 的最好下界
是 $l(n) \geq lb(n) = \lambda(n) + \lceil \log v(n) \rceil$ 。利用这个下界, 可以从深度 $lb(n)$ 开始搜索, 大大加快了算法的搜索进程。

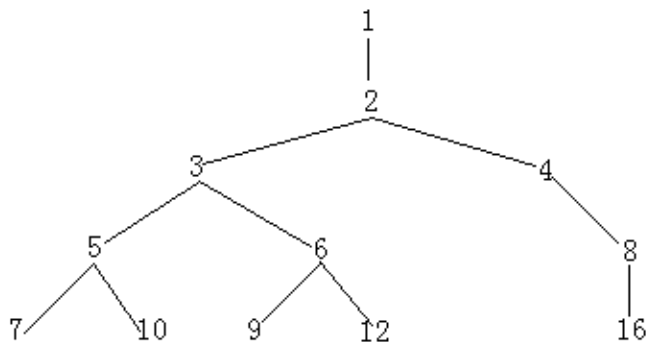
剪枝函数

- 设 a_i 和 a_j 是加法链中的两个元素，且 $a_i > 2^m a_j$ 。由于加倍是加法链中元素增大的最快的方式，即 $a_i \leq 2a_{i-1}$ ，所以从 a_j 到 a_i 至少需要 $m+1$ 步。如果预期在状态空间树 T 的第 d 层找到关于 n 的一条加法链，则以状态空间树第 i 层结点 a_i 为根的子树中，可在第 d 层找到一条加法链的必要条件是 $2^{d-i}a_i \geq n$ 。
- 当 $3 \times 2^{d-(i+2)}a_i < n$ 时，状态空间树中以结点 a_i 为根的子树中不可能在第 d 层之前找到最短加法链。
- 设在求正整数 n 的最短加法链的逐步深化迭代搜索算法中，当前搜索深度为 d 。且正整数可表示为 $n = 2^t(2k+1)$ ， $k > 0$ ，则在状态空间树的第 i 层结点 a_i 处的一个剪枝条件是

$$\begin{cases} \log(n / 3a_i) + i + 2 > d & 0 \leq i \leq d - t - 2 \\ \log(n / a_i) + i > d & d - t - 1 \leq i \leq d \end{cases}$$

最短加法链长的上界

与加法链问题密切相关的幂树给出了 $l(n)$ 的更精确的上界。



假设已定义了幂树 T 的第 k 层结点，则 T 的第 $k+1$ 层结点可定义如下。依从左到右顺序取第 k 层结点 a_k ，定义其按从左到右顺序排列的儿子结点为 $a_k + a_j$ ， $0 \leq j \leq k$ 。其中 a_0, a_1, \dots, a_k ，是从 T 的根到结点 a_k 的路径。且 $a_k + a_j$ 在 T 中未出现过。

含正整数 n 的部分幂树 T 容易在线性时间内用迭代搜索方式构造出来。

优化算法

综合前面的讨论，对构造最短加法链的标准回溯法作如下改进。

- (1) 采用逐步深化迭代搜索策略；
- (2) 利用 $l(n)$ 的下界 $lb(n)$ 对迭代深度作精确估计；
- (3) 采用剪枝函数对问题的状态空间树进行剪枝搜索，加速搜索进程；
- (4) 用幂树构造 $l(n)$ 的精确上界 $ub(n)$ 。

当 $lb(n)=ub(n)$ 时，幂树给出的加法链已是最短加法链。

当 $lb(n)<ub(n)$ 时，用改进后的逐步深化迭代搜索算法，从深度 $d=lb(n)$ 开始搜索。