课程主要内容

（一）典型问题

1. 排序
2. 查找（检索）
3. 周游
4. 优化问题

（二）算法设计方法

1. 分治法
2. 贪心法
3. 动态规划
4. 回溯法
5. 分枝限界法

（三）算法复杂度分析

1. 时间复杂度
2. 空间复杂度

（四）一类待解决的问题：NP 完全问题

（五）近似算法、随机算法简介

参考文献

1. **余祥宣，崔国华，邹海明，“计算机算法基础”，华中科技大学出版社，1998。**

2. 朱洪，陈增武，段振华，周克成，“算法设计与分析”，上海科学技术文献出版社，1989。

3. 顾立尧，霍义兴，“算法设计与分析的理论和方法”，上海交通大学出版社，1989。

4. 张益新，沈雁，“算法引论”，国防科技大学出版社，1995。

5. 吴哲辉，曹立明，蒋昌俊，“算法设计与分析”，煤炭工业出版社，1993。

6. 卢开澄，“计算机算法导引－－－设计与分析”，煤炭工业出版社，1993。

7. 郑宗汉，郑晓明，“算法设计与分析”，清华大学出版社，2005。

8. 吕帼英，任瑞征，钱宇华，“算法设计与分析”，清华大学出版社，2006。

9. Robert Sedgewick, Philippe Flajolet (冯舜玺，李学武，裴伟东译),“算法分析引论（An introduction to analysis of algorithms）”，机械工业出版社，2006。

10. 周培德，“算法设计与分析”，机械工业出版社，2006。

11. 霍红卫，“算法设计与分析”，西安电子科技大学出版社，2005。

12. 王晓东，“计算机算法设计与分析”，电子工业出版社，2005。

13. **E. Horowitz, S. Sahni, “Fundamentals of Computer Algorithms”, New York: Computer Science Press, Pitman, Inc., 1978.**

14. **Sara Baase, Allen Van Gelder, “Computer Algorithms: Introduction to Design and Analysis (Third Edition)”, Higher Education Press & Pearson Education Asia Limited., 2001.**

15. **Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest,**

Clifford Stein, “Introduction to Algorithms (Second Edition)”, Higher Education Press & The MIT Press, 2002. （有中译本）

M. I. T 开放课件：

http://www.myoops.org/cocw/mit/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/index.htm

16. D. E. Knuth, “The Art of Computer Programming (Third Edition): Fundamental Algorithms (Vol. 1)”, Addison-Wesley, 1998; 清华大学出版社（影印），2002.

17. D. E. Knuth, “The Art of Computer Programming (Third Edition): Sorting and Searching (Vol. 3)”, Addison-Wesley, 1998; 清华大学出版社（影印），2002.

18. A. Levitin, “Introduction to The design & Analysis of Algorithms”, Pearson Education Asia Limited and Tsinghua Unversity Press, 2003.（有中译本）

19. Gilles Brassard and Paul Bratley, “Fundamentals of algorithmics”, Pearson Education Asia Limited and Tsinghua Unversity Press, 2005.

20. **Jon Kleinberg and Eva Tardos, “Algorithm Design”, Pearson Education Asia Limited and Tsinghua Unversity Press, 2006**.（有中译本）

21. R. C. T. Lee, S. S. Tseng, R. C. Chang, Y. T. Tsai, “Introduction to the Design and Analysis of Algorithms”, The McGraw-Hill Education and China Machine Press(机械工业出版社), 2007。（有中译本）

第一章　引论

1. 算法

算法是众所周知的一个术语。设计一种算法的目的，总是为了解决一类特定的问题。譬如，欧几里德算法给出两个整数的最大公因子。

计算机算法是计算机科学和计算机应用的核心，无论是计算机系统，系统软件，还是计算机的各种应用课题(例如，模式识别，图像处理，计算机图形学，计算层析成像等)都可归结为算法的设计。

算法设计策略（strategy）是算法学习的重点。应用策略写成具体规范的高效算法是我们的最终目标。许多人常常感慨地说，学过的东西总是在给别人讲述的时候才能真正弄明白。同理，解决问题的策略只有在写成计算机算法，也就是说，在给计算机“讲明白”之后，你才会有透彻的理解，也才能真正应用起来。所以，具体编写算法不仅能解决特定的问题，还有助于形成缜密细致地思考问题的习惯。

另外，评估一个算法的好坏，即对算法进行分析，也是一个复杂的过程，也需要策略，也有精致的元素值得欣赏。

总之，学习计算机算法，不仅从应用的观点来看是必要的知识储备，而且对于提高个人的科学素养来说，也是极有益的心智训练。

1.1 算法的概念

算法是一个相当基本的概念，要给它一个准确的形式化定义是十分麻烦的（参考书[2]，形式地说，任何一个对所有有效输入总要停

机的图灵机是一个算法。)。从本课程涉及的内容来讲，这样的定义是不必要的。我们宁愿选择下列非形式的一种描述：算法是一组有穷的规则，它们规定了解决某一特定类型问题的一系列运算（注：这儿的运算指计算机所能提供的各种基本操作)。简言之，算法是问题的过程化解。

例 1.1 （求最大公约数的欧几里得算法）给定两个正整数m和n，$n<m$，求它们的最大公约数，即求能整除m和n的最大正整数。

解：欧几里得算法可描述如下：

E_1：（求余数）以n去除m得余数r，（我们知道$0\le r<n$)。

E_2：（验证$r=0$?）若$r=0$，算法结束，n即为所求；否则（变量递减）置$m\leftarrow n$，$n\leftarrow r$，转E_1。

算法应具备以下五个特性：

（1）输入：一个算法有 0 个或多个输入，它们是对算法给出的初始量。一组完整的输入取自特定的集合，该集合由算法将要解决的问题所确定，称为问题定义域。

例如，（排序）给n个整数排序。问题定义域为

$$\{(a_1,\ a_2,\ \cdots,\ a_n):\ a_i\in\mathbb{Z},\ i=1,\ \cdots,\ n\}$$

其中的元素$(a_1,\ a_2,\ \cdots,\ a_n)$是向量（输入形式为数组）

（2）输出：一个算法产生一个或多个输出，它们是同输入有某种特定关系的量或其他形式的结果。

例如 整数排序。输出为

$$(a_{i_1},\ a_{i_2},\ \cdots,\ a_{i_n})$$

其中$i_1, i_2, \cdots, i_n$是$1, 2, \cdots, n$的一个置换使得

$a_{i_1} \le a_{i_2} \le \cdots \le a_{i_n}$。

（3）确定性：我们一般考虑确定性算法，要求算法的每一种运算必须有确切的定义，即每一种运算应该执行何种操作必须相当明确，无歧义性，例如，不允许“将 6 或 7 与 x 相加”，“取整数n的一个因子”，“从集合S中任取一个元素”之类的运算出现在算法中。

（4）可行性：算法中有待实现的运算都相当基本，即每种运算在原理上必须能被计算机在有限的时间内完成。例如，“两个无理数的精确和”这种运算就不是可行的。

（5）有穷性：一个算法总是在执行了有限步的运算之后终止。

凡是一个算法，都必须满足以上五条特性。只满足前四条特性的一组规则称为计算过程。操作系统就是一个典型的计算过程，而不是算法。操作系统的功能是控制作业的运行，当没有作业时，系统并不终止，而是处于等待状态，等待新的作业进入。所以操作系统从原理上讲永不中止，不具备有穷性。

可以验证，欧几里得算法满足上述五个特性。

既然一个算法是用来求解一种特定类型的问题，为进一步论述起见，我们需要给出问题的形式化描述。

一个问题P是由无穷多个实例组成的一个类。其中每个实例由问题定义域上的某个实体（即输入）以及对实体的提问（或加工要求）组成。实体和输入，是分别从问题和算法的角度来指称同一个东西。

例 1.2 整数乘法问题：问题定义域是全体整数偶对，实例的实体是数

对(a, b)，提问：它们的积是多少？该问题的一个实例是：$2\times4=$？

请注意，个别的实例不认为是一个问题。

一个算法A称为解算问题P，或称A是问题P的一个算法，是指如果把P的任一实例的实体作为A的输入，A在有限步之内总输出一个关于此实例的正确答案。一个问题P称为算法可解的，如果存在一个算法解算P。

一个问题是算法可解的，是否实际可解呢？回答是并不尽然。因为解算这个问题的算法在现有的计算机上可能需要极长的时间（比方说 10 亿年），以至于无法接受，也即它们不是有效算法。这就导致对一个算法的定性与定量分析的分界线，定性是指一个算法的有穷性，即它是否总在有限步内终止，至于究竟多少步则不论，反正只要是有限步就可以了。定量是一个算法的有效性，即它到底在多少步内终止。这个步数必须要有限度，必须是可以接受的，现实可行的。对于后者的研究，就是所谓计算复杂性的问题。

算法的设计和分析主要考虑两个方面：

（一）给定一个问题，如何设计出解算它的有效算法。

（二）分析和判断一个算法的质量，主要指效率的度量。

关于第一方面的问题，随后各章将陆续介绍一些常用的设计策略。下面来讨论一下分析算法的准则和技术。

1.2 分析算法的几条准则

评价一个算法的性能质量，通常可参照下列五条准则：

（1）正确性：称解算某问题的一个算法是正确的，如果对任给的一个有效输入（即问题定义域上的一个实体），该算法总在有限时间内给出相应实例的正确答案。一个算法的正确性包括对问题的解法在逻辑上是正确的，且指令序列具有确定性，可行性，有穷性，我们主要关心前者。数学归纳法常用来证明算法的正确性。为了证明算法的正确性，必须对输入的有效性和输出的正确性作严格的定义。为证明一个大型程序的正确性，可以试图将这个程序分解成一些较小的程序段，通过证明所有这些较小的程序段是正确的来证明整个程序是正确的。

（2）简单性：要求算法易于理解，易于编程，易于调试。

（3）复杂度：包括时间复杂度和空间复杂度，即在计算机上运行算法时所需时间和存储空间的量度。显然，复杂度较低的算法效率较高。

（4）最优性：算法A是问题P的最优时间算法是指：解算问题P的所有算法中，A执行的基本运算次数最少。证明算法的时间最优性的一个常用方法是，先从理论上证明一个问题所需基本运算次数的下界，如果一个算法执行的基本运算次数恰是这个下界的值，则该算法是时间最优的。空间最优性也可同样分析。需要指出，空间最优性与时间最优性往往难以并存于一个算法中，根据实际情况，可以在两种资源的使用方面实施均衡。

（5）可修改可扩展性：如果问题A是解算问题P的一个算法，为了解算一个与问题P相似的问题$\overline{P}$，希望对A稍作改动就可正确运行，若算法A满足这一点，则说算法A的可修改性好；否则，如果要解决

问题$\overline{P}$，不得不对A作多处重大修改，那还不如重新写一个算法来得简便，则说A的可修改性不好。同样地，有时希望扩大算法A所解决的问题范围，即要给A增加一些功能，如果只要对A稍作修改即可，则称A的可扩展性好。反之，A的可扩展性不好。

以上说法有些术语（例如，什么是基本运算）的确切含义并未解释，待后将在有关论述中作进一步的澄清。

就理论上严格的分析来讲，我们主要关注算法的正确性，复杂度和最优性这三条评价准则。当然正确性是必要的，复杂度是评价算法的最基本量度。

下面着重介绍分析算法复杂度的基本原理，经常仅限于讨论时间复杂度。空间复杂度的分析作类似处理。

算法的时间复杂度，即算法的运行所需的时间，就输入而言，与两个因素有关：

（1）输入的规模大小，例如，在排序问题中，给n个整数排序，n即表征输入的规模。一般来说，输入规模越大，运行时间越长。

（2）即使输入规模相同，算法运行的时间还与输入的具体情况有关，例如，给1000个几乎有序的整数排序应比给1000个杂乱无章的整数排序所需时间少。

综合上述两个因素，算法的时间复杂度以函数形式来表示，自变量为表示输入规模的参数，函数值取相同规模的输入下算法运行所需的最大时间或平均时间。这两个函数分别刻画算法的最坏时间复杂度和平均时间复杂度。

输入规模通常用一个或多个非负整数来表示。例如，排序问题中用要排序的整数个数n表示输入规模，可用图模型表示的问题用图的结点数n和边数m表示输入规模。为简化起见，下面仅以输入规模是一个整数为例来作详细阐述。

表 1.1 问题和输入规模举例

问题	规模
1. 在一个表中搜索元素 x	表中元素的个数
2. 两个方阵相乘	方阵的维数
3. 整序一个表	表中元素的个数
4. 遍历一个二叉树	树中顶点（结点）个数
5. 解一线性方程组	方程个数或未知数个数或两者兼有

以 x 表示问题的输入，x 本身也是一个集合，例如排序问题的输入由若干个可比较大小的元素组成，记$|x|$为输入规模。算法A的**最坏时间复杂度**用$WT_A(n)$表示，定义为

$$WT_A(n) := \max\{t: \text{存在输入} X，|X|=n，\text{且算法} A \text{对输入} X \text{的运行时间是} t\}.$$

算法A的**平均时间复杂度**用$MT_A(n)$表示，定义为

$$MT_A(n) := \sum_{I \in \mathrm{P}_n} p(I)t(I)，$$

其中P_n表示输入规模为n的实例集D_n的一个剖分，使得P_n中任何元素I作为D_n的子集满足：I中所有实例的输入在算法A下运行的时间相同。以$t(I)$表示该时间。$p(I)$是I中元素在D_n中出现的概

率。

将规模为n的输入集X_n看作随机变量集，算法运行时间$t(X_n)$则可看作一个随机变量，算法的平均时间复杂度正是$t(X_n)$的期望，即

$MT_A(n)=E(t(X_n))$。

在不致引起混淆的情况下，$WT_A(n)$或$MT_A(n)$常简记为$T_A(n)$或$T(n)$。

上面讨论了算法的时间复杂度与输入的关系。下面来讨论算法运行时间的度量标准，即如何确定算法所包含的指令的运行时间。

算法运行时间的度量法则，应该与实际使用的计算机，程序设计语言，程序员的编程技术无关，还应该与许多实现细节或称为“薄记”操作（例如，循环下标计数，数组下标计数，或设置指针等）无关，但是它又必须能反映算法实际执行时间方面的信息。

为此，我们选定一些特定的操作，称为**基本操作**，它对被研究的问题（或对被讨论的算法类）来说是基本的。基本的含义包括这些操作是解算这一问题的关键操作，将会在算法中反复使用，操作次数将会随着输入规模增大而增大。而“薄记”操作则不一样，它也许保持常数或以较小的速度增加。因此，弃置“薄记”工作量于不顾，只计算算法所执行的基本操作次数。若这算法所执行的操作总数粗略地与基本操作数成比例（当输入规模比较大时），则这一指定是合理的。更精确地说，如果基本操作的指定是合理的，那么假设算法对输入X做$T(X)$次基本操作，则操作总数介于 $C_1T(X)$ 与 $C_2T(X)$ 之间，而实际执行时间介于$C_3T(X)$ 与 $C_4T(X)$ 之间，这里C_1，

C_2，C_3，C_4是常数，且$C_1 < C_2$，$C_3 < C_4$。C_1，C_2仅依赖于这个算法的薄记方法，C_3，C_4还依赖于实现该算法的计算机，程序设计语言以及程序员，而都与 x 的规模无关。因此，只要基本操作选取得合理，算法所执行的基本操作数就是对算法运行时间的一个好的度量。对于解算同一问题的若干算法来说，只要选取的基本操作相同，也就有了客观地比较它们优劣的准绳。

表 1.2 几个问题的基本操作

问题	基本操作
1. 在一个表中搜索元素 x	比较（x 与表中一元比较）
2. 两个实矩阵相乘	实数乘法或实数的乘法和加法
3. 整序一个表	比较（表中两元比较）
4. 遍历一个二叉树（链接结构）	访问一个链接，置一个指针

采用上述分析算法时间复杂度的原理，来考察两个例子。

例 1.3 检索问题

Procedure SEQUENSESEARCH(A, n, j, x)

// $A(1:n)$是一n元数组，x是任一给定的元素，判断x是否出现//

//在数组A中。若是，置j，使$A(j)=x$；否则，置$j=0$。//

$j \leftarrow n$

while $j \geq 1$ and not $A(j)=x$ do

$j \leftarrow j-1$

repeat

end SEQUENSESEARCH

分析：该检索问题的算法中，基本操作是两元比较，计算算法运行时间，只需计数比较次数。

算法的平均时间复杂度: 输入A, x能够依它在A中何处出现而分类，即有$n+1$个输入类要考虑，令I_i $(1\le i\le n)$ 表示x在A中第i个位置的输入类，I_0表示x不在A中的输入类。显然，算法对输入类I_i $(1\le i\le n)$ 中输入的比较次数是:$t(I_i)=n+1-i$，对输入类I_0中输入的比较次数是:$t(I_0)=n$。为计算平均比较次数，必须给出x在A中的概率和x在A中每个位置的概率。设x在A中的概率为q，且假设x在A中每个位置出现的概率相同，则

$$p(I_i)=q/n \ \ (1\le i\le n);\ \ p(I_0)=1-q.$$

所以平均时间复杂度为

$$MT(n)=\sum_{i=0}^{n}p(I_i)t(I_i)=q(n+1)/2+(1-q)n.$$

另一方面，容易看出，I_0是该算法最坏情况下的输入类，所以，

$$WT(n)=t(I_0)=n.$$

讨论：(1) 若$q=1$, 即x必在A中，则$MT(n)=(n+1)/2$，在平均意义下需要检索数组的一半。(2) 若$q=1/2$, 即x在A中与x不在A中的概率相等，则$MT(n)=(3n+1)/4$, 在平均意义上需要检索数组的四分之三；(3) 若$q=0$，即x必不在A中，则$MT(n)=n$, 即整个数组都要检索到。此时，平均时间复杂度达到了算法的最坏时间复杂度。

例 1.4 矩阵乘法问题

procedure MULMATRIX(A,B,C,n)

// 求矩阵 $A(1:n,\ 1:n)$ 与矩阵 $B(1:n,\ 1:n)$ 的乘积 $C(1:n,\ 1:n)$).//

integer $i,\ j,\ k$

for $i \leftarrow 1$ to n do

for $j \leftarrow 1$ to n do

$C(i,\ j) \leftarrow 0$

for $k \leftarrow 1$ to n do

$C(i,\ j) \leftarrow C(i,\ j) + A(i,\ k)B(k,\ j)$

repeat

repeat

repeat

end MULMATRIX

分析：该算法中，基本操作是两实数的乘法。对 C 的每一元素 $C(i,\ j)$ 需做 n 次乘法，C 有 n^2 个元素，所以共需 n^3 次乘法。由此看来，该算法的运行时间只与输入规模有关，而与输入的具体情形无关，这种现象被称为是**数据无关的**。此时，算法的平均时间复杂度与最坏时间复杂度相同：$WT(n) = MT(n)$，记为 $t(n)$。本例中，$t(n) = n^3$。

1.3 时间复杂度的渐近表示与算法分类

前述两例中，算法的时间复杂度都可表示为简单的多项式函数。

但很多情况下，我们很难写出算法时间复杂度函数的显式表达。为此，我们采用一些简单的整数函数来渐近表示算法的时间复杂度，这样的整数函数诸如:

$$\log n,\ n,\ n\log n,\ n^k(k\geq 2),2^n,\ n!,\ n^n$$

等等都是量度算法复杂度的标准函数。下面来说明渐近表示的含义。

定义 1.1 设$T(n)$是某算法的时间复杂度，$g(n)$是一标准函数，若存在正常数c_0和正整数n_0，使得对所有的$n\geq n_0$，都有

$$T(n)\leq c_0 g(n) \tag{1.3}$$

则记

$$T(n)=O(g(n)) \tag{1.4}$$

称$g(n)$是$T(n)$的渐长率的一个上界；若存在正常数c_1和正整数n_1，使得对所有的$n\geq n_1$,都有

$$T(n)\geq c_1 g(n) \tag{1.5}$$

则记

$$T(n)=\Omega(g(n)) \tag{1.6}$$

称$g(n)$是$T(n)$的渐长率的一个下界；若既有$T(n)=O(g(n))$，又有$T(n)=\Omega(g(n))$，则记

$$T(n)=\Theta(g(n)) \tag{1.7}$$

称$T(n)$与$g(n)$的渐长率是同阶的。注意，这里的渐长率与通常所说的变化率不是一回事。

例 1.5 设$T(n)=3n^4\log n+n^3\log^3 n$，则$n^4$是它的一个下界，$n^5$是它的一个上界。$n^4\log n$与它同阶。

在实际分析算法时，我们当然希望能找到$g(n)$使得$T(n)=\Theta(g(n))$，不得已求其次，找尽量小数量级的$g(n)$,使$T(n)=O(g(n))$, 以$g(n)$作为$T(n)$的一个上界估计（或限界）。在实际应用中，我们总是希望算法复杂度不要超过某个数量级就可满意，所以给出一个恰当的上界估计也就够了。

从时间复杂度上，可以把通常遇到的算法分成两类：**多项式时间算法**和**指数时间算法**，分别指以多项式函数限界和指数函数限界的算法。

下面比较一下几种标准函数表示的时间复杂度的数量级：

(1) 表示多项式时间复杂度的常见标准函数如下

$$1 \ll \log n \ll n \ll n\log n \ll n^2 \ll n^k \quad (k>2)$$

(2) 表示指数时间复杂度的常见标准函数如下

$$2^n \ll n! \ll n^n$$

其中“$\ll$”表示远远小于。

说明：指数时间算法只有当输入规模n很小时才可以接受，以时间复杂度为2^n的算法为例，当$n=40$时，$2^{40}=1024^4>10^{12}$, 运算次数已相当可观。即使是多项式算法，只有当多项式的次数很低时，才可以接受，因为输入规模往往会达到几千，几万甚至更大。

表 1.3　不同量级复杂度之比较

时间复杂度 $T(n)$	可解决的最大实例规模(1秒) n	可解决的最大实例规模(1天) n	可解决的最大实例规模(1年) n

n	10^9	8.64×10^{13}	3.1536×10^{16}
$n\log n$	39620077	2.11038×10^{12}	6.4114×10^{14}
n^2	31623	9.2952×10^6	1.7758×10^8
n^3	1000	4.4208×10^4	3.1594×10^5
2^n	29	46	54

注：指数时间复杂度增长速度惊人的快，就$T(n)=2^n$来说，一年的运行时间能解决输入规模为 54 的实例。而输入规模从$n=54$增加1到$n=55$时，运行时间将翻一番，增加到两年。当爱因思坦被问到，他所碰到过的最伟大奇迹是什么时，他回答说，是复利的增长。他的意思就是说，指数增长速度快的不可思议。

通常来说，时间复杂度为$O(n\log n)$的算法才是实际可用的，由此导出下面说法：若问题P存在时间复杂度为$O(n\log n)$的算法，则称它是实际可解的；若解算问题P的所有算法的时间复杂度都是$\Omega(a^n)$，其中，a是某个大于1的常数，则称它是实际不可解的，例如著名的 hanoi 塔游戏问题就是实际不可解的.

例 1.6　Hanoi 塔游戏

有A，B，C三根柱子，n个大小各不相同的盘子从大到小叠放在柱子A上,要求将A上的这n个盘子借助于柱子B以同样的方式叠放在C上。盘子移动的规则如下：(1)每次只准移动一个盘子；(2)在任何时侯，都不允许大盘子叠放在小盘子上。

解：分析：当$n=1$时，直接将这一只盘子从柱子A上移放到柱子C

上；当$n>1$时，如果有成功的移动方案的话，由盘子的移动规则，当将柱子A上的最大盘子移动时，B,C柱子必有一个为空，而另一个柱子上从大而小叠放着其余的$n-1$个盘子，由柱子B,C的对称性（指起始B,C都是空的），我们当然选择C为那个空柱子，以使 A上的最大盘子移放在C上。依此事实，将问题转化为三个子问题：

(1)将A上的$n-1$个盘子借助C移放在B上；

(2)将A上的最大盘子移放到C上；

(3)将B上的$n-1$个盘子借助A移放到C上。

这就给出了解决该问题的递归算法。从以上分析可知，该算法是移动盘子次数最少的唯一方法。

算法描述如下：

```
procedure  HANOI(n, A, B, C)
// 将n个盘子从柱子A借助于柱子B移放到柱子C上。//
if  n=1  then  Write(n, A, C)   //打印信息，将n号盘子从A//
//上移到C。//
else
                call  HANOI(n-1, A, C, B)
                Write(n, A, C)
                call  HANOI(n-1, B, A, C)
endif
end  HANOI
```

Hanoi 塔问题的基本操作是移动，上述递归算法的时间复杂度

$T(n)$满足下列递归式：

$$\begin{cases} T(1)=1, \\ T(n)=2T(n-1)+1,\ n\geq 2 \end{cases} \tag{1.8}$$

可通过一个简单变换来求解该递归式。令

$$t(n)=T(n)+1,$$

则有

$$\begin{cases} t(1)=2, \\ t(n)=2t(n-1),\ n\geq 2 \end{cases} \tag{1.9}$$

故得到

$$t(n)=2^n,$$

从而

$$T(n)=2^n-1.$$

趣闻：约在十九世纪末，在欧洲的珍奇商店里出现了一种称为 hanoi 塔的游戏。这种游戏由于附有推销材料，因而更加流行起来。推销材料中说，布拉玛神庙（Temple of Bramah）里的教士们正在玩这种游戏，他们的游戏结束就标志着世界末日的来临。教士们的游戏装置很简单，一块铜板上面插着左, 中, 右 3 根金刚石针，左针上从大到小叠放着 64 个不同大小的金盘。游戏的目标是把左边针上的 64 个金盘移到右边针上，规则是一次只能移动一个金盘，可以利用中间针作为过渡，但在任何时刻均不允许大盘放在小盘上面。当时人们看了这个介绍材料均很惊恐，以为世界很快要毁灭了。

根据我们上面的论证，这 64 个金盘总共需要移动$2^{64}-1$次，即

使每秒能移动1百万次，要完成这个游戏得花费1百万年。

注：常系数线性递归式（1.9）的一般形式是

$$\begin{cases} T(1)=t_1,\ T(2)=t_2,\ \cdots,\ T(k)=t_k \\ T(n)=a_1T(n-1)+a_2T(n-2)+\cdots+a_kT(n-k),\ n\geq k+1 \end{cases} \tag{1.10}$$

其中，$t_1,\ t_2,\ \cdots,\ t_k;\ a_1,\ a_2,\ \cdots,\ a_k$ 是实常数.

式(1.10)可用母函数法求解. 以解式(1.8)为例，构造如下母函数

$$G(x)=\sum_{n=1}^{\infty}T(n)x^n \tag{1.11}$$

上式两边同乘以$2x$得

$$2xG(x)=\sum_{n=1}^{\infty}2T(n)x^{n+1}=\sum_{n=2}^{\infty}2T(n-1)x^n \tag{1.12}$$

式(1.11)减去式(1.12)得

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

进一步得到$G(x)$的如下展开式：

$$G(x)=\frac{x}{(1-x)(1-2x)}=\frac{1}{1-2x}-\frac{1}{1-x}$$

$$=\sum_{n=0}^{\infty}(2x)^n-\sum_{n=0}^{\infty}x^n=\sum_{n=0}^{\infty}(2^n-1)x^n \tag{1.13}$$

由$G(x)$展开式的唯一性推知

$$T(n)=2^n-1。$$

一般地，也可以依据下列定理采用待定系数法求解（1.10）。

定理 1.1（参考书[9]）设β是齐次常系数线性递归式

$$T(n)=a_1T(n-1)+a_2T(n-2)+\cdots+a_kT(n-k),\quad (1.14)$$

的特征方程

$$x^k-a_1x^{k-1}-a_2x^{k-2}-\cdots-a_k=0$$

的r重实根，则

$$n^s\beta^n\ （0\le s\le r-1）,\qquad (1.15)$$

是满足(1.14)的r个线性无关的解。

定理 1.2 设$qe^{i\theta}$和$qe^{-i\theta}$是齐次常系数线性递归式

$$T(n)=a_1T(n-1)+a_2T(n-2)+\cdots+a_kT(n-k),$$

的特征方程

$$x^k-a_1x^{k-1}-a_2x^{k-2}-\cdots-a_k=0$$

的r重共轭复根，则

$$n^sq^n\cos(n\theta),\ n^sq^n\sin(n\theta)\ （0\le s\le r-1）,\quad (1.16)$$

是满足(1.14)的$2r$个线性无关的解。

定理 1.3 齐次常系数线性递归式

$$T(n)=a_1T(n-1)+a_2T(n-2)+\cdots+a_kT(n-k),$$

的解空间是k维的，即（1.14）式的任何解都可以表示成k个形如（1.15）式或(1.16)式所示线性无关解的线性组合，组合系数由初始条件

$$T(1)=t_1,\ T(2)=t_2,\ \cdots,\ T(k)=t_k$$

所确定。

3. 算法语言： SPARKS

为便于表达算法所具有的特性，最好将算法用一种程序语言写成程序的样子。选择语言的原则简单说来，就是够用好用。具体地说，由该语言所写出的每一个合法的句子必须具有唯一的含义，用它写出的算法便于阅读并能容易地用人工或机器翻译成其它实际使用的程序设计语言，同时要求该语言相当简明，能反映算法本身的基本思想和基本步骤即可，不必太关注细节。为此，我们选用一种符合以上要求的 SPARKS 语言。下面对它作一简单介绍。

（一）数据类型与结构

SPARKS 的基本数据类型包括整型，实型，布尔型和字符型，变量只能存放单一类型的值，它可以用下述形式来说明其类型：

integer i, j; real x, y; boolean a, b; char c, d;

有特殊含义的标识符作为保留字，给变量命名的规则是：以字母起头，不许使用特殊字符，不要太长，不许与任何保留字重复。

SPARKS 使用带有任意整数下界和上界的多维数组。例如，一个n维整型数组可用以下形式说明：

integer $A(l_1:u_1, \cdots, l_n:u_n)$

其中l_i, u_i $(1 \le i \le n)$为整数或整型变量，分别表示第i维的下界和上界。如果某一维的下界l_i为1，在数组说明中，此l_i可以不写出。例如,

integer $A(5, 3:10)$

与

integer $A(1:5,\ 3:10)$

同义。为保持 SPARKS 语法的简明性，只使用数组作为基本结构单元来构造所有数据对象，而没有引进记录等结构类型。

（二）语句

1. 赋值语句：（变量）←（表达式）

2. 逻辑运算符：and, or, not (布尔值共有两个：true 和 false)

3. 关系运算符：＜，≤，＝，≠，＞，≥

4. 条件语句：

if cond then S_1

else S_2

endif

或

if cond then S

endif

其中 cond 表示条件，$S, S_i\ (i=1,2)$ 表示 SPARKS 语句组。

5. case 语句

case

:cond1: S_1

:cond2: S_2

……

:condn: S_n

:else: S_{n+1} (可缺省)

endcase

6. 循环语句

(1) while cond do

S

repeat

(2) loop

S

(until cond) repeat

（3）for $vble \leftarrow start$ to $finish$ (by $increment$) do

S

repeat

其中 vble 是一个变量，start, finish 和 increment 是算术表达式。一个整型或实型变量或一个常数都是算术表达式的简单形式。子句 by increment 缺省时, increment 自动取$+1$。

补充说明：上述循环体的语句组S中可加入某种检测条件导致一个出口。例如，可在S中加入

if cond then go to label endif

语句，它将控制转移到附标号"label"的语句：label: S。go to label 语句的两种特殊形式是

Exit 和 cycle

例如

loop

S_1

if cond1 then exit endif

S_2

until cond2 repeat

意谓当 cond1=true 时，退出整个循环。而

loop

S_1

if cond1 then cycle endif

S_2

until cond2 repeat

与

loop

S_1

if cond1 then go to label endif

S_2

label: until cond2 repeat

等效

（三）SPARKS 程序

一个完整的 SPARKS 程序是一个或多个过程的集合，第一个过程作为主程序，执行从主程序开始。单个的 SPARKS 过程有以下形式：

procedure NAME([参数表])\

(说明部分)

S

end NAME

一个 SPARKS 过程可以是一个纯过程（又称为子例行程序），也可以是一个函数。在函数中，返回值由放在紧接 return 的一对括号中的值来表示。例如

return([参数表])

其中表达式的值作为函数的值来传送。对于纯过程而言，end 的执行意味着执行一条没有值与其相联系的 return 语句，为了停止程序的执行，可以使用 stop 语句。

执行任一 SPARKS 过程，譬如过程 A，当到达 end 或 return 语句时，控制返回到调用过程 A 的那个 SPARKS 过程。如果过程 A 是主程序，控制则返回到操作系统。调用过程 A 的语句为

call A

SPARKS 过程允许递归调用，包括直接和间接递归。

（四）补充说明

输入，输出对拟定算法模型是不必要的，所以 SPARKS 对输入，输出只采用两个过程：

read（[参数表]）；　　print（[参数表]）.

首尾均带有双斜线的注解可以放在程序中的任何地方，例如

// 检索过程开始//

最后，当用自然语言或数学表示能较好地描述算法模型时，我们也毫

不犹豫地这样做。

至此，SPARKS 语言说明完毕。

作业

1. 求满足下列递归式的$T(n)$的渐近表示:

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

(2) $\begin{cases} T(1)=T(2)=1, \\ T(n)=T(n-1)+T(n-2),\ n\geq 3 \end{cases}$

(3) $T(n)=T(\sqrt{n})+1$

2. (有序拆分问题) 给定正整数n，要求写出n的所有有序拆分. n的一个有序拆分是指一个正整数数组$(n_1,\ \cdots,\ n_i)(1\leq i\leq n)$, 使得

$$n=n_1+\cdots+n_i$$

例如, 3的所有有序拆分为

$$(1,\ 1,\ 1),\ (1,\ 2),\ (2,\ 1),\ (3).$$

请设计求解该问题的算法, 并分析其时间复杂度。

第二章 分治法(Divide and Conquer)

1．概述

分治法是广为人知的一种算法设计技术。顾名思义，分治法是分而治之的方法（行政制度就是一种典型的分治法），是指对于一个输入规模为n的问题，采用某种方式把输入分割成k（$1<k\le n$）个子集，从而产生k个不同的可分别求解的同类型问题，解出这k个子问题后，再用适当的方法将其组合成原问题的解（当输入规模n较小时，可直接求解），据此思路设计的算法很自然地可用一个递归过程来表示。

procedure Solve(I)

$n \leftarrow$ size(I)

if(n = smallsize)) then solution $\leftarrow$ Directlysolve(I)

else

{

divide I into $I_1, \cdots, I_k$

for $i \leftarrow 1$ to k do

$S_i \leftarrow$ Solve(I_i)

repeat

solution $\leftarrow$ Combine($S_1, \cdots, S_k$)

}

return(solution)

end Solve

借助分治法，产生了许多高效算法，众多实际可行的$O(n\log n)$时间复杂度的算法是利用分治法设计的。

我们先从简单例子讲起。

1.1 典型例子选介

例 2.1 整数乘法问题

设X和Y是两个n位的二进制整数，按照传统方法对X和Y相乘，需要$\Theta(n^2)$次”位运算”使用分治法可将其降低到$\Theta(n^{\log 3})(\log 3\approx 1.59)$。

为简化讨论，假定n是2的幂，我们分别将X和Y按位均分成两部分，如下图所示：

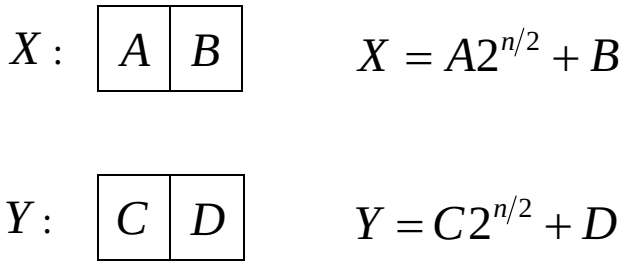


图 **2.1**

其中，A, B, C, D 都是$n/2$位的二进制整数，X与Y的乘积可表示为

$$XY=(A2^{n/2}+B)(C2^{n/2}+D)=AC2^{n}+(AD+BC)2^{n/2}+BD \tag{2.1}$$

若依照式(2.1)分而治之，得到四个子问题：AC，AD，BC，BD，此外，还有一些加法和移位运算。不难看出，加法与移位所耗费的位运算为cn，因此，若设两个n位数相乘的时间复杂度（可以认为是

数据无关的)为$T(n)$，则

$$\begin{cases} T(1)=1, \\ T(n)=4T(n/2)+cn,\ n\geq 2 \end{cases} \tag{2.2}$$

设 $n=2^k$，则

$$T(n)=4T(2^{k-1})+c2^k=4(4T(2^{k-2})+c2^{k-1})+c2^k$$

$$=4^2T(2^{k-2})+c4\cdot 2^{k-1}+c2^k=\cdots=4^kT(1)+c\sum_{i=1}^{k}4^{k-i}\cdot 2^i$$

$$=4^k+c4^k\sum_{i=1}^{k}(1/2)^i=\Theta(n^2)。 \tag{2.3}$$

时间复杂度并没降低，但将下式

$$AD+BC=(A-B)(D-C)+AC+BD \tag{2.4}$$

代入(2.1)中，得到：

$$XY=AC2^n+\{(A-B)(D-C)+AC+BD\}2^{n/2}+BD \tag{2.5}$$

将4次乘法降低为3次乘法，而加，减和移位次数仍为cn,于是由式(2.5)确定的算法时间复杂度满足下列递归式

$$\begin{cases} T(1)=1, \\ T(n)=3T(n/2)+cn,\ n\geq 2 \end{cases} \tag{2.6}$$

其解为（仍设 $n=2^k$）

$$T(n)=3^k+c3^k\sum_{i=1}^{k}(2/3)^i=\Theta(n^{\log 3})\ (3^k=2^{k\log 3}) \tag{2.7}$$

从而降低了时间复杂度。

该算法可写成下列递归过程:

算法 2.1 MTBN (Multiplication of Two Bit Number)算法

procedure $\mathrm{MTBN}(X,Y,n)$

//X 和Y 是绝对值小于2^n 的整数，n 是2的幂，函数值等于X 乘Y //

integer $S, A, B, C, D, M_1, M_2, M_3$

//S 存放XY 的符号， A, B分别存放X 的左半部分和右半部分,//

//C, D分别存放Y 的左半部分和右半部分,M_1, M_2, M_3分别//

//存放三个乘积AC, $(A-B)(D-C)$, BD.//

$S \leftarrow \mathrm{sign}(X)\mathrm{sign}(Y)$

$X \leftarrow |X|$

$Y \leftarrow |Y|$

if $n=1$then

if($X=1$)and($Y=1$) then return(S)

else return(0)

endif

else

$A \leftarrow X$的左边$n/2$位

$B \leftarrow X$的右边$n/2$位

$C \leftarrow Y$的左边$n/2$位

$D \leftarrow Y$的右边$n/2$位

$M_1 \leftarrow \mathrm{MTBN}(A, C, n/2)$

$M_2 \leftarrow \mathrm{MTBN}(A-B, D-C, n/2)$

$M_3 \leftarrow \text{MTBN}(B,\ D,\ n/2)$

return($S(M_1 2^n + (M_1 + M_2 + M_3)2^{n/2} + M_3)$)

endif

end MTBN

在此例中，分割成的子问题大小相等，实际上这是设计算法策略中的一条重要原则------平衡.而且，分治法还常需辅之以代数变换才能得到高效算法。下面再举两例，进一步阐明这些思想。

例 2.2 矩阵乘法问题

设A，B是两个n阶方阵，则$C = AB$也是n阶方阵，C中元素

$$C(i,\ j) = \sum_{k=1}^{n} A(i,k)B(k,\ j) \quad ,(1 \le i,\ j \le n) \qquad (2.8)$$

依此定义式直接计算C的算法时间复杂度为 $\Theta(n^3)$.

分治法提供矩阵相乘的另外一种方法。仍限定n是2的方幂，分别划分A和B为4个$n/2$阶方阵

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad (2.9)$$

则A和B的积为

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \qquad (2.10)$$

如果$n = 2$，则式(2.10)将对矩阵的元素直接进行计算；当$n > 2$时，C可通过计算$n/2$阶方阵的乘积与和运算得到。因为n是2的方

幂，因而方阵的乘积可递归计算。

就(2.10)式而言，分成的子问题共有8个$n/2$阶方阵相乘，此外，两个$n/2$阶方阵相加可在cn^2时间内完成，因此这个思路设计的算法时间复杂度$T(n)$满足下列递归式

$$\begin{cases} T(1)=1, \\ T(n)=8T(n/2)+cn^2,\ n\geq 2 \end{cases}$$

该递归式的解是：$T(n)=\Theta(n^3)$，与常规方法比较，没有得到改进。由于方阵的乘法复杂度比加减法高（$\Theta(n^3)$对$\Theta(n^2)$），如果能通过增加加减法的次数以减少乘法的次数，算法的时间复杂度将会得以改善。1969 年，Volker Strassen 发现了一种方法，用7个乘法和18个加减法实现了上述设想，具体步骤如下：

Step1: 记

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

Step2: 通过$M_i(1\leq i\leq 7)$的加减法计算各$C_{ij}(1\leq i,\ j\leq 2)$.

$$C_{11}=M_1+M_3-M_4+M_6,$$
$$C_{12}=M_4+M_5,$$

$$C_{21}=M_6+M_7,$$

$$C_{22}=M_1-M_2+M_5-M_7.$$

上述步骤形成算法的时间复杂度$T(n)$满足下列递归式

$$\begin{cases} T(1)=1, \\ T(n)=7T(n/2)+cn^2,\ n\geq 2 \end{cases} \tag{2.11}$$

解之得（$n=2^k$）：

$$T(n)=7T(2^{k-1})+cn^2=7(7T(2^{k-2})+cn^2/4)+cn^2$$

$$=7^2T(2^{k-2})+(7/4)cn^2+cn^2=\cdots$$

$$=7^kT(1)+cn^2\sum_{i=0}^{k-1}(7/4)^i=7^k+4cn^2/3((7/4)^k-1)$$

$$=(1+4c/3)7^k-4cn^2/3 \qquad (4^k=n^2)$$

$$=\Theta(n^{\log 7}) \qquad (7^k=2^{k\log 7}=n^{\log 7},\ \log 7\approx 2.81)$$

最后需要指出的是，斯特拉森矩阵乘法只有当n相当大时才优于通常的矩阵乘法。

注：斯特拉森算法的另一种形式见“计算机算法基础”（余祥宣，崔国华，邹海明，66 页习题 25）：递归求解C_{ij}共用7次乘法和15次加减法。

例 2.3 快速傅里叶变换(FFT)

离散 Fourier 变换 DFT(Discrete Fourier Transform)广泛应用于数字信号处理等工程领域，它的快速算法 FFT 的设计也是基于分治法。

N维实向量$(a_0,\ a_1,\ \cdots,\ a_{N-1})$的 Fourier 变换定义如下：

$$\mathrm{b_m} = \sum_{k=0}^{N-1} a_k e^{2mk\pi i/N}, \quad 0 \le m \le N-1$$

称为该向量的频谱。记

$$\omega_N = e^{2\pi i/N},$$

我们有

$$\mathrm{b_m} = \sum_{k=0}^{N-1} a_k \omega_N^{mk}, \quad 0 \le m \le N-1。$$

构作对应于 N 维实向量 $(a_0, a_1, \cdots, a_{N-1})$ 的多项式

$$a(x) = \sum_{k=0}^{N-1} a_k x^k,$$

则

$$b_m = a(\omega_N^m).$$

这样，N 元实向量的 Fourier 变换等价于求以向量分量为系数的 $N-1$ 次多项式在 N 个点

$$\omega_N^m, \quad 0 \le m \le N-1$$

处的值。

在实际应用中，N 通常取为 2 的方幂。记

$$M = N/2, \quad y = x^2,$$

则 $a(x)$ 可改写为

$$a(x) = \sum_{k=0}^{M-1} a_{2k} x^{2k} + \sum_{k=0}^{M-1} a_{2k+1} x^{2k+1} = \sum_{k=0}^{M-1} a_{2k} y^k + x \sum_{k=0}^{M-1} a_{2k+1} y^k$$

$$= b(y) + xc(y)$$

其中

$$b(y)=\sum_{k=0}^{M-1}a_{2k}y^k,\quad c(y)=\sum_{k=0}^{M-1}a_{2k+1}y^k$$

可看作分别对应于两个M维向量：

$$(a_0,\ a_2,\ \cdots,\ a_{2(M-1)}),\ (a_1,\ a_3,\ \cdots,\ a_{2(M-1)+1})$$

的多项式。因此有：当$0\le j\le M-1$时，

$$a(\omega_N^j)=b(\omega_N^{2j})+\omega_N^j c(\omega_N^{2j})=b(\omega_M^j)+\omega_N^j\cdot c(\omega_M^j)$$

$$(\omega_N^2=\omega_M)$$

$$a(\omega_N^{M+j})=b(\omega_N^{N+2j})+\omega_N^{M+j}c(\omega_N^{N+2j})=b(\omega_M^j)-\omega_N^j\cdot c(\omega_M^j)$$

$$(\omega_N^N=1,\ \omega_N^M=-1)$$

这样将输入规模为N的问题分割成输入规模为$N/2$的两个子问题，照此设计的算法时间复杂度$T(n)$满足下列递归式：

$$\begin{cases}T(1)=1,\\ T(N)=2T(N/2)+cN,\ N\ge 2\end{cases}\qquad(2.12)$$

其中cN是分割和组合所需的基本操作数，基本操作为加法，乘法及赋值。设$N=2^r$，解此递归式得：

$$T(N)=2T(N/2)+cN=2^2T(N/2^2)+2cN$$

$$=\cdots=2^rT(1)+rcN=\Theta(N\log N)$$

而由定义式直接计算的时间复杂度显然为$\Theta(N^2)$.

由上述步骤，我们可以给出一个快速傅里叶变换的递归算法。

算法 2.2　快速傅里叶变换

procedure FFT(N, $a(x)$, ω, A)

//多项式$a(x)=a_{N-1}x^{N-1}+\cdots+a_0$，$N$是2的方幂，a($\omega^j$)//

//（$0\le j\le N-1$）置于数组$A(0:N-1)$中，初始调用FFT//

//时，将ω置为ω_N.//

if $N=1$ then $A(0)\leftarrow a_0$

else

{

$M\leftarrow N/2$

$b(x)\leftarrow a_0+a_2x+\cdots+a_{2(M-1)}x^{M-1}$

$c(x)\leftarrow a_1+a_3x+\cdots+a_{2(M-1)+1}x^{M-1}$

call FFT(M, $b(x)$, ω^2, B)

call FFT(M, $c(x)$, ω^2, C)

$t\leftarrow 1$

for $j\leftarrow 0$ to $M-1$ do

$A(j)=B(j)+t\cdot C(j)$

$A(M+j)=B(j)-t\cdot C(j)$

$t=\omega\cdot t$

repeat

endif

end FFT

1.2 master 方法

采用分治法设计的算法时间复杂度$T(n)$总是满足形如：

$$T(n) = aT(n/b) + f(n) \qquad (2.13)$$

的递归式. 式(2.13)表述了(a, b)型分治算法的时间复杂度，(a, b)型分治算法指递归地将输入规模为n的问题分解为a个输入规模为n/b的子问题，$f(n)$表示分解和合成的时间复杂度。下面定理基本解决了式(2.13)的渐近解。

Master Theorem 设常数$a \geq 1$, $b>1$，$f(n)$是定义在非负整数上的非负函数，$T(n)$是由下列递归式定义的非负整数上的非负函数

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

其中n/b表示$\lfloor n/b \rfloor$或$\lceil n/b \rceil$. 则$T(n)$有下面渐近界：

1. 如果

$$f(n) = O(n^{\log_b a-\varepsilon}),$$

其中，ε是某正常数, 则

$$T(n) = \Theta(n^{\log_b a}); \qquad (2.14)$$

2. 如果

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

则

$$T(n) = \Theta(n^{\log_b a} \log n); \qquad (2.15)$$

3. 如果

$$f(n) = \Omega(n^{\log_b a+\varepsilon}),$$

ε是某正常数, 且存在常数$c<1$, 使得当n充分大时，

$$af(n/b) \leq cf(n),$$

则

$$T(n)=\Theta(f(n)). \qquad (2.16)$$

注：n 并不总是只取 b 的方幂，当 n 是一般整数时，递归式中的 $aT(n/b)$ 应替之以

$$a_1T\left(\lfloor n/b\rfloor\right)+a_2T\left(\lceil n/b\rceil\right),\quad a_1+a_2=a$$

.当然，这样的变化不影响定理的结论。

证明：为了对定理的内容获得一个直观的理解，我们先来看 n 取 b 的方幂的情形：

$$T(b^k)=aT(b^{k-1})+f(b^k)=a^2T(b^{k-2})+af(b^{k-1})+f(b^k)$$

$$=\cdots=a^k f(1)+\sum_{i=0}^{k-1}a^i f(b^{k-i})=\Theta(n^{\log_b a})+\sum_{i=0}^{\log_b n-1}a^i f(b^{k-i})$$

$$=\Theta(n^{\log_b a})+\sum_{i=0}^{\log_b n-1}a^i f(n/b^i) \qquad (2.17)$$

其中最后的等式利用下面数量关系

$$k=\log_b n,$$

和

$$a^k=n^{k\log_n a}=n^{(\log_b n)\cdot(\log_n a)}=n^{\log_b a}$$

基于(2.17)，我们有

若 $f(n)=O(n^{\log_b a-\varepsilon})$（$\varepsilon$ 是正常数), 则

$$\sum_{i=0}^{\log_b n-1}a^i\cdot f(n/b^i)=\sum_{i=0}^{\log_b n-1}a^i\cdot O((n/b^i)^{\log_b a-\varepsilon})$$

$$=O(n^{\log_b a-\varepsilon})\sum_{i=0}^{\log_b n-1}b^{\varepsilon i}=O(n^{\log_b a-\varepsilon})\cdot\frac{b^{\varepsilon\log_b n}-1}{b^{\varepsilon}-1}$$

$$= O(n^{\log_b a})$$

所以，

$$T(n) = \Theta(n^{\log_b a})$$

1. 若 $f(n) = \Theta(n^{\log_b a})$, 类似地

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

$$= \Theta(n^{\log_b a}) \sum_{i=0}^{\log_b n-1} 1 = \Theta(n^{\log_b a} \log n)$$

所以，

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

3. 如果 $f(n) = \Omega(n^{\log_b a+\varepsilon})$) ($\varepsilon$ 是正常数), 且存在常数 $c < 1$，使得当 n 充分大时，

$$af(n/b) \le cf(n),$$

则

$$f(n) \le \sum_{i=0}^{\log_b n-1} a^i \cdot f(n/b^i) \le \sum_{i=0}^{\log_b n-1} c^i f(n)$$

$$\le \sum_{i=0}^{\infty} c^i f(n) = \frac{f(n)}{1-c}$$

所以，

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

对于任意正整数 n，以 n/b 取为 $\lceil n/b \rceil$ 为例来考虑，记

$$\begin{cases} n_0 = n, \\ n_i = \lceil n_{i-1}/b \rceil, \ i \geq 1 \end{cases}$$

则

$$n_1 \leq n/b + 1,$$

$$n_2 \leq n_1/b + 1 \leq n/b^2 + 1/b + 1$$

$$n_3 \leq n_2/b + 1 \leq n/b^3 + 1/b^2 + 1/b + 1$$

$$\cdots\cdots$$

$$n_i \leq n/b^i + 1/b^{i-1} + \cdots + 1/b + 1$$

$$< n/b^i + \sum_{j=0}^{\infty} 1/b^j = n/b^i + b/(b-1)$$

另一方面，

$$n_1 \geq n/b$$

$$n_2 \geq n_1/b \geq n/b^2$$

$$\cdots\cdots$$

$$n_i \geq n_{i-1}/b \geq n/b^i$$

取$i = \lfloor \log_b n \rfloor$, 则

$$n_{\lfloor \log_b n \rfloor} < n/b^{\lfloor \log_b n \rfloor} + b/(b-1) < n/b^{\log_b n - 1} + b/(b-1)$$

$$= b + b/(b-1) = O(1)$$

从而类似于(2.17)式，我们有

$$T(n) = aT(n_1) + f(n) = a^2 T(n_2) + af(n_1) + f(n)$$

$$= \cdots = a^{\lfloor \log_b n \rfloor} f(n_{\lfloor \log_b n \rfloor}) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f(n_i)$$

$$= \Theta(n^{\log_b a}) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f(n_i).$$

再利用

$$n_i = \Theta(n/b^i)$$

则可完成定理的证明。

例 2.4 求满足下列递归式的$T(n)$的渐近表示

(1) $T(n) = 4T(n/2) + n$

(2) $T(n) = 4T(n/2) + n^2$

(3) $T(n) = 4T(n/2) + n^3$

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

解: (1) 递归式中

$$a = 4,\ b = 2,\ \log_b a = 2,\ \ f(n) = n = O(n^{\log_b a - 1/2}),$$

可归入 **master** 定理中的第一种情形，所以

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2);$$

（2） $f(n) = \Theta(n^{\log_b a})$，可归入 **master** 定理中的第二种情形，所以

$$T(n) = \Theta(n^{\log_b a}\log n) = \Theta(n^2\log n)$$

（3） $f(n) = \Omega(n^{\log_b a + 1/2})$, 且

$$af(n/b) = 4f(n/2) = n^3/2 = f(n)/2,$$

可归入 **master** 定理中的第三种情形，所以

$$T(n) = \Theta(f(n)) = \Theta(n^3)$$

（4）递归式中

$$a=1,\ b=2,\ \log_b a=0,\quad f(n)=c\log n$$

不能归入 **master** 定理中的任何一种情形，所以采用定理中的证明方法将递归式展开到底：(先设$n=2^k$)

$$T(n)=T(2^{k-1})+ck=T(2^{k-2})+c(k-1)+ck=\cdots$$

$$=T(1)+c\sum_{i=1}^{k} i=T(1)+ck(k+1)/2=\Theta(\log^2 n)$$

（$k=\log n$）

作业：

1. 求满足下列递归式的$T(n)$的渐近表示

(1) $T(n)=T(9n/10)+n$

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

(3) $T(n)=16T(n/4)+n^2$

(4) $T(n)=7T(n/3)+n^2$

(5) $T(n)=7T(n/2)+n^2$

(6) $T(n)=2T(n/4)+\sqrt{n}$

2.（猜谜游戏）给定n种图案和m个位置。出谜者在每个位置上任意摆放一种图案作为谜底，请猜谜者以尽可能少的次数猜中谜底。规则：猜谜者必须对每个位置设定一种图案算猜一次，反馈信息是：在正确的位置摆放了正确图案的数目c和除此之外在不正确的位置上摆放了正确图案的数目d. 例：$n=4,\ m=5,$ 设图案为：

$$A,\ B,\ C,\ D,$$

谜底为：

$$\underline{A} \quad \underline{B} \quad \underline{B} \quad \underline{C} \quad \underline{D}$$

如果猜测:

$$\underline{B} \quad \underline{A} \quad \underline{D} \quad \underline{C} \quad \underline{D}$$

则反馈：

$$c=2,\ d=2.$$

请设计解决该问题的算法，并分析你的算法最坏情况下需猜多少次才能猜中谜底。

下面各节，就利用分治法解决检索，选择，排序等问题分别进行相关讨论。

2. 二分检索

问题：给定一个n元表$A(n)$，且表中元素可比较大小，来判定与表中元素同类型的元素x是否在表中出现。假设表$A(n)$已按递增顺序排列：

$$A(1)<A(2)<\cdots<A(n).$$

注：要检索的表中不可能有两元素相同，且该表总要先排序，以提高检索的效率。

算法 2.4 二分检索 (Binary Search)

procedure BINSRCH($A,\ low,\ high,\ x,\ j$)

//给定按递增顺序排列的元素表$A(low,\ high)$（$low\geq 1$,//

//$high\geq low-1$; 当$high=low-1$时，$A(low,\ high)$是虚拟//

//数组，表示空表)本过程旨在判断 x 是否在表中出现. 若是，置 j ,//

//使得 $A(j)=x$; 否则置 $j=0$. //

integer low, $high$, j, mid

while $low \le high$ do

$\text{mid} \leftarrow \lfloor (\text{low+high})/2 \rfloor$

//$\lfloor x \rfloor$表示不大于 x 的最大整数//

case

$: x < A(mid)$: $high \leftarrow mid - 1$

$: x > A(mid)$: $low \leftarrow mid + 1$

: else: $\text{j} \leftarrow \text{mid}$; return

endcase

repeat

$j \leftarrow 0$

end BINSRCH

下面先来看算法的正确性。

定理 2.1 过程$\text{BINSRCH}(A, low, high, x, j)$能正确地运行。

证明：对表中元素数目 (即 $high-low+1$) 进行归纳，当 $high-low+1=0$ (即$high=low-1$)时，表A是空表，此时过程不进入 while 循环，j 被置成0, 过程终止, 运行正确；假设当 $0 \le high-low+1<n$ ($n \ge 1$)时，过程能正确运行，来考查 $high-low+1=n$时过程的运行情况。

记 $\text{mid}=\lfloor(\text{low}+\text{high})/2\rfloor$，则以输入而论，可分成下列三种情况：

(1) $x < A(mid)$，此时，由表的顺序性，问题归结为在表 $A(low, mid-1)$ 中检索 x．在这样的输入下，过程 $\text{BINSRCH}(A, low, high, x, j)$ 第一次执行到 while 语句时，因 $low = high - n + 1 \le high$，满足 while 中的条件语句，所以进入 while 循环体，本次循环的结果是将 $high$ 的值更改为 $mid-1$，然后转回 while 语句，j 的值完全由过程此后的运行情况决定，即相当于执行过程 $\text{BINSRCH}(A, low, mid-1, x, j)$．因为 $low \le mid \le high$，得知 $0 \le mid - 1 - low + 1 < high - low + 1 = n$，由归纳假设，过程运行正确；

(2) $x > A(mid)$，类似(1)可得正确性；

(3) $x = A(mid)$，则过程执行一次 while 循环，置 $j = mid$ 后结束，执行正确。

综之，过程 $\text{BINSRCH}(A, low, high, x, j)$ 总能正确地运行。

下面来分析算法的时间复杂度，基本操作是比较，每次 while 循环包括两次比较，我们不妨把这两次比较看作一次大比较。下面所计算的比较次数是指大比较次数。

记 $n = high - low + 1$，即表中元素个数．当 $n = 0$ 时，仅用一方框表示 x 不在表 A 中；当 $n \ge 1$ 时，可用下图来表示在表 $A(low, high)$ 中检索 x 的过程：

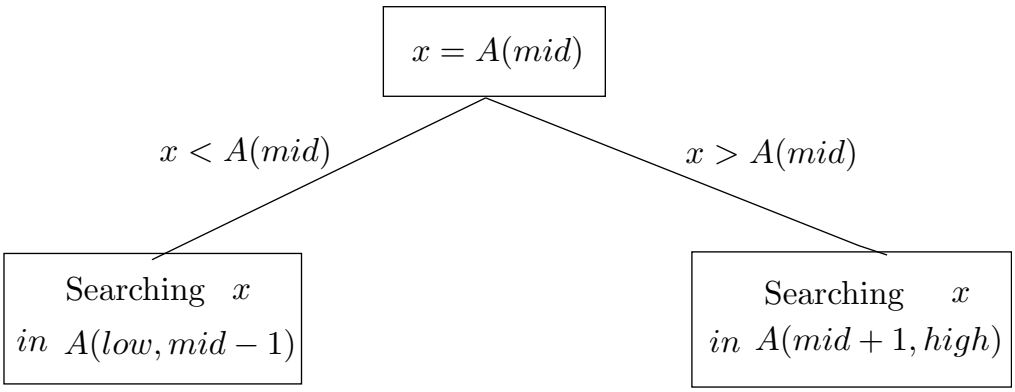


Fig 2.1

由此图结合二元树的递归定义来看，在$A(low,\ high)$中检索x的过程BINSRCH($A,\ low,\ high,\ x,\ j$)可用一个二元树来表示，此二元树称为二元比较树。

例 2.5 在表$A(1:7)$，$A(1:9)$中检索x的算法执行过程对应的二元检索树如下图:

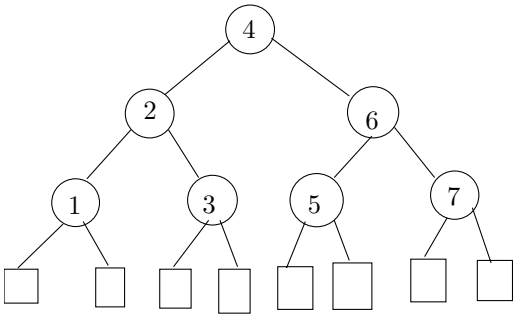


Fig 2.2 (a)

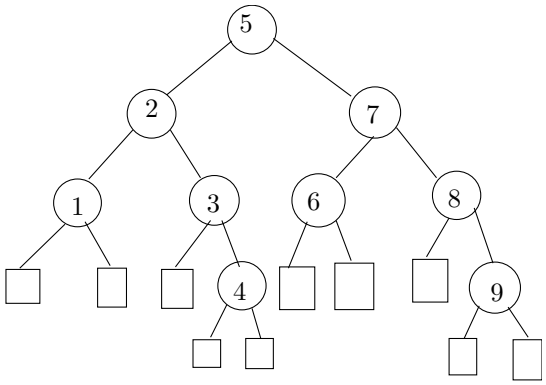


Fig 2.2 (b)

注：1. 二元比较树包含n个内结点（用圆框表示）和$n+1$个外结点（叶子）（用方框表示），内结点的最高级为$\lceil \log(n+1) \rceil$（当$2^k - 1 < n \le 2^{k+1} - 1$时，$\lceil \log(n+1) \rceil = k+1$）,外结点的最高级为$\lceil \log(n+1) \rceil + 1$，外结点只可能在$\lceil \log(n+1) \rceil + 1$级或$\lceil \log(n+1) \rceil$级上。

2. 当$n=0$时，二元比较树退化为一片叶子，表示表A是空的，x不在空表A中；当$n\geq 1$时，二元比较树的n个内结点表示x可能所在的表A中的n个位置，$n+1$个外结点表示x不在A中的$n+1$种情形。

3. $x=A(i)(1\leq i\leq n)$时过程所花费的比较次数，恰是其对应内结点的级数；当$A(i)<x<A(i+1)$ $(0\leq i\leq n$, 其中$A(0)$表示虚拟最小值, $A(n+1)$表示虚拟最大值) 时，过程所化费的比较次数，则是表示此种情形的叶子的级数减去1，其值为$\lceil\log(n+1)\rceil$或$\lceil\log\ (n+1)\rceil-1$.

由以上事实知，算法的最坏时间复杂度为

$$WT(n)=\lceil\log(n+1)\rceil \qquad (2.18)$$

下面来看算法的平均时间复杂度$MT(n)$，假定x在表A中的概率是q，且假设x在A中每个位置出现的概率相同，即x在表A中的n种情况出现的概率都是$1/n$. 记x在表A中的n种情况的平均时间复杂度为$S(n)$；同样记x不在表A中的$n+1$种情况的平均时间复杂度为$U(n)$. 则

$$MT(n)=q\cdot S(n)+(1-q)\cdot U(n) \qquad (2.19)$$

此处，利用了下面概率论中的基本结论之一：

定理 2.2 设X，Y是随机变量，则对于任意函数$f(x,y)$（(x,y)在(X,Y)的样本空间中取值），都有

$$E(f(X,Y))=\int_y p(Y=y)E(f(X,Y=y))dy.$$

或等价地

$$E(f(X,Y))=E(g(Y)),$$

其中

$$g(y)=E(f(X,Y=y)).$$

显然, 不论x不在表A中的$n+1$种情况的概率分布是什么，都有

$$\lceil\log(n+1)\rceil-1<U(n)\le\lceil\log(n+1)\rceil. \qquad (2.20)$$

下面利用$U(n)$来估算$S(n)$. 需要这样一个事实：含n个内结点的二元树中，内部路径长度I和外部路径长度E满足下列关系：

$$E=I+2n \qquad (2.21)$$

因为结点到根结点的内部路径是结点的级数减 1，所以

$$S(n)=\frac{I}{n}+1\ =\frac{E}{n}-1=(\frac{n+1}{n})(\frac{E}{n+1})-1, \qquad (2.22)$$

由(2.20)知，

$$\lceil\log(n+1)\rceil-1<\frac{E}{n+1}\le\lceil\log(n+1)\rceil. \qquad (2.23)$$

将(2.23)代入(2.22) 得：

$$\frac{n+1}{n}\lceil\log(n+1)\rceil-2-\frac{1}{n}<S(n)\le\frac{n+1}{n}\lceil\log(n+1)\rceil-1,$$

化简为（考虑$n\ge1$）

$$\lceil\log(n+1)\rceil-2<S(n)\le\lceil\log(n+1)\rceil. \qquad (2.24)$$

将(2.24)和(2.20)代入(2.19)得到

$$\lceil\log(n+1)\rceil-2<MT(n)\le\lceil\log(n+1)\rceil.$$

这说明算法的平均时间复杂度也是$\Theta\left(\lceil\log\ (n+1)\rceil\right)$，且最坏时间复杂度比平均时间复杂度不超过两个基本操作.

下面说明算法 2.3 是以比较为基础的检索算法类中最坏时间复

杂度为最佳的算法。只允许进行元素间的比较而不允许对它们实施其它运算的检索算法称为以比较为基础的检索算法。

在一个有序表A中检索x，比较操作只需在x和表A中的元素之间进行，这种比较可分成两种情况：

其一：

$x = A(j_0)$;

$x < A(j_0)$，归结为在表$A(low, j_0 - 1)$中检索x;

$x > A(j_0)$，归结为在表$A(j_0 + 1, high)$中检索x;

其二：

$x < A(j_0)$，归结为在表$A(low, j_0 \text{-} 1)$中检索x;

$x \geq A(j_0)$，归结为在表$A(j_0, high)$中检索x;

或

$x \leq A(j_0)$，归结为在表$A(low, j_0)$中检索x;

$x > A(j_0)$，归结为在表$A(j_0 + 1, high)$中检索x;

我们仅对采取第一类比较的算法来进行分析，采取第二类比较的算法也有类似的结果.

依照对算法 2.3 的分析，任何以比较为基础的算法都可用二元比较树来表示算法执行的过程．这样的二元比较树都含n个内结点(每个内结点都表示x在表A中某个位置时的算法执行终止处)，$n+1$个外结点(每个外结点都表示x在表A外某个位置时算法执行终止处)，算法A在输入规模为n的情况下所化费的最大比较次数$WT_A(n)$等于对应的二元比较树内结点的最高级数.

我们知道，内结点最高级为k的二元树含内结点的数目至多为2^k-1，所以$n\le 2^k-1$，即$k\ge\lceil\log(n+1)\rceil$，即$WT_n\ge\lceil\log(n+1)\rceil$, 这说明算法 2.4 在最坏情况下是最佳的.

综之得到：

定理 2.3 设表$A(n)$含有n个不同的元素，它们被排序成

$$A(1)<A(2)<\cdots<A(n),$$

又设以比较为基础判断某元素x是否在表$A(n)$的任何算法在最坏情况下所需的最小比较次数是$\mathrm{FIND}(n)$, 那末

$$\mathrm{FIND}(n)\ge\lceil\log(n+1)\rceil。$$

作业：

1. 假设n元序列

$$E(1),\ E(2),\ \cdots,\ E(n)$$

呈现单峰分布，即存在指标m $(1\le m\le n)$, 使得

$$E(1)<\cdots<E(m),\ E(m)>\cdots>E(n).$$

但我们不知道m的值. 问题：如何通过$\Theta(\log n)$次元素比较操作确定m?

2. 设$E_1(1:n)$和$E_2(1:n)$是增序排列的n元数组，要找出这$2n$个元素中的第n小元素。请设计时间复杂度为$O(\log n)$的算法求解该问题，并证明该问题的最坏时间复杂度为$\Omega(\log n)$。

3. 在本节的假设条件下（x在表A中的n种情况的概率相同，x不在表A中的$n+1$种情况的概率也相同），证明：在以比较为基础的所

有检索算法中二分检索的平均比较次数最少。

3. 找最大和最小元素

问题 1: 在n个不同的元素中找最大(小)元。

以找最大元为例（找最小元是类似的），简单的方法是将元素逐个进行比较，具体步骤可描述如下。

算法 2.4 直接找最大元

Procedure $\mathrm{MAX}(A, n, \max)$

//将$A(1:n)$中的最大元置于$\max$，$n \geq 2$//

integer i, n

$\max \leftarrow A(1)$

for $i \leftarrow 2$ to n do

if $A(i) > \max$ then $\max \leftarrow A(i)$ endif

repeat

end MAX

算法中的基本操作是比较，显然该算法是数据无关的，$T(n) = n - 1$.事实上，该算法在以比较为基础的找最大元算法类中是最优的。

定理 2.4 任何以比较为基础的在n个元素中寻找最大元的算法，必须至少作$n-1$次比较。

证明: 算法中的每一次比较只能确定出某一元素小于等于其他元素中的某一个，如果算法包含k次比较，则至多能确定k个元素不是最大

元，因此若$k \le n-2$,则还有$m(m \ge n-k \ge 2)$个元素未发现小于任何其他元素，最大元应在这m个元素中间产生，但算法并不能最终确定这m个元中究竟哪个最大，这说明$k \ge n-1$.

注：也可考虑用连通图中结点数n与边数m的关系: $m \ge n-1$来证明.

但同时找最大元与最小元却并不需要$2(n-1)$次比较。

算法 2.6 直接找最大和最小元素

procedure MAXMIN(A, n, max, min)

//将$A(1:n)$中的最大元素置于 max, 最小元置于 min, $n \ge 2$//

integer i, n

$\max \leftarrow \min \leftarrow A(1)$

for $i \leftarrow 2$ to n do

  if $A(i) > \max$ then $\max \leftarrow A(i)$

  else

    if $A(i) < \min$ then $\min \leftarrow A(i)$ endif

  endif

repeat

end MAXMIN

注：在循环体的第一个 if 语句中，只有当$A(i) < \max$时，才可能有$A(i) < \min$, 所以关于 min 寻找之比较可嵌入其中。

该算法的最好情况在元素按递增顺序排列时出现，元素比较数是$n-1$, 最坏情况在元素按递减次序排列时出现，元素比较数是

$2(n-1)$. 至于在平均情况下，即假设$n!$种排列具有相同概率的情况下，平均比较次数是

$$\sum_{i=2}^{n}(1/i+2(1-1/i))=2(n-1)-H(n)+1=2(n-1)-\ln\ n+\alpha_n$$

其中$H(n)$为n阶调和数：

$$H(n)=\sum_{i=1}^{n}1/i=\ln\ n+E_n,$$

$$\alpha_n=1-E_n,$$

E_n有极限，令

$$E=\lim_{n\to\infty}E_n$$

E是欧拉常数,

$$E\approx 0.577.$$

注：比较数设为随机数$x(n)$，则平均比较次数为$E(x(n))$. 设第i次循环需作的比较数为随机数$x^i(n)$，则

$$x(n)=\sum_{i=2}^{n}x^i(n), \qquad (2.25)$$

从而，

$$E(x(n))=\sum_{i=2}^{n}E(x^i(n)) \qquad (2.26)$$

又因为

$$E(x^i(n))=1/i+2(1-1/i) \qquad (2.27)$$

将(2.27)代入(2.26)得

$$E(x(n)) = \sum_{i=2}^{n} (1/i + 2(1-1/i)).$$

下面用分治法来设计一个算法，基本考虑是将任一较大实例：

$$I = (n,\ A(1),\ \cdots,\ A(n)) \quad (\mathrm{n} \geq 3)$$

分成两个实例：

$$I_1 = \left(\lceil n/2 \rceil;\ A(1),\ \cdots,\ A\left(\lceil n/2 \rceil\right)\right)$$

和

$$I_2 = \left(\lfloor n/2 \rfloor;\ A\left(\lceil n/2 \rceil + 1\right),\ \cdots,\ A(n)\right)$$

则

$$\max(I) = \max\{\max(I_1),\ \max(I_2)\},$$

$$\min(I) = \min\{\min(I_1),\ \min(I_2)\}$$

而当$n \leq 2$时，不作任何分割直接得到其解。

算法 2.6 递归求取最大和最小元素

procedure $\mathrm{MAXMIN}(i,\ j,\ \max_1,\ \min_1)$

// $A(1:n)$是含有n个元素的数组，参数i, j是整数, $1 \leq i \leq j \leq n$,//

//该过程把$A(1:n)$中的最大和最小元素分别赋给$\max_1$和$\min_1$. //

integer $i,\ j$; global $n,\ A(1:n)$

case

$:i = j:\ \max_1 \leftarrow \min_1 \leftarrow A(i)$

$:i = j-1:$ if $A(i) < A(j)$

then $\max_1 \leftarrow A(j),\ \min_1 \leftarrow A(i)$

else $\max_1 \leftarrow A(i),\ \min_1 \leftarrow A(j)$

endif

: else: $mid \leftarrow \lfloor (i+j)/2 \rfloor$

call MAXMIN(i, mid, $\max_2$, $\min_2$)

call MAXMIN($mid+1$, j, $\max_3$, $\min_3$)

$\max_1 \leftarrow \max(\max_2, \max_3)$

$\min_1 \leftarrow \min(\min_2, \min_3)$

endcase

end MAXMIN

这个过程最初由 call MAXMIN(1, n, max, min) 所调用。

分析比较数: 该过程所需比较数 $T(n)$ ($n = j - i + 1$)满足下列递归式:

$$\begin{cases} T(1)=0,\ T(2)=1, \\ T(n)=T\left(\lfloor n/2 \rfloor\right)+T\left(\lceil n/2 \rceil\right)+2,\ n \geq 3 \end{cases} \tag{2.28}$$

当 n 是 2 的幂时,

$$T(n)=\lceil 3n/2 \rceil - 2. \tag{2.29}$$

可证明如下：记 $n=2^k$. 当 $k=1$ 时，$T(2)=1$, 等式成立；假设 $k=k_0$ ($k_0 \geq 1$)时，等式成立，即 $T(2^{k_0})=3\cdot 2^{k_0-1}-2$，来看 $k=k_0$ 时的情况:

$$T(2^{k_0+1})=2T(2^{k_0})+2=2(3\cdot 2^{k_0-1}-2)+2=3\cdot 2^{k_0}-2$$

等式仍成立。证毕。

但当 n 不是 2 的幂时，(2.29)不一定成立. 例如: $T(6)=8$, $\lceil 3\times 6/2 \rceil - 2 = 7$.

(2.29)中的"＝"替换成"≥", 就对任何整数成立了。稍后我们将会证明，以元素比较为基础的找最大和最小的算法，在最坏情况下元素比较下界为$\lceil 3n/2 \rceil - 2$.

所以该算法就元素比较数而言是比较好的（甚至当n是 2 的幂时，在最坏情况下是最佳的），但由于递归算法中，i, j, max, min 进出栈（栈深度为$\lceil \log n \rceil$）所带来的开销，再考虑到i, j间比较与元素比较时间相类时，此算法可能反而比直接比较算法花销的时间更多.

下面另外给出一个用分治策略设计的算法，基本思想是将实例：

$$I = (n;\ A(1),\ \cdots,\ A(n))\quad (n > 2)$$

分割成$\lfloor n/2 \rfloor$个实例:

$$I_i = (2;\ A(i),\ A(n+1-i))\quad (1 \le i \le \lfloor n/2 \rfloor),$$

分别将其中的最大元和最小元赋给$A(i)$和$A(n+1-i)$. 然后直接求

$$\mathrm{A}(1),\ \cdots,\ \mathrm{A}\left(\lceil \mathrm{n}/2 \rceil\right)$$

中的最大元和

$$A\left(n+1-\lceil n/2 \rceil\right)\ \cdots,\ A(n)$$

中的最小元，就分别得到实例I的最大元和最小元。

算法 2.8 非递归分治求取最大和最小元素

```
procedure BMAXMIN(A, n, max, min)
integer i, n, N
if n = 2 then
        if A(1) > A(2) then max ← A(1); min ← A(2)
```

else $\max \leftarrow A(2)$; $\min \leftarrow A(1)$

endif

else

$N \leftarrow \lfloor n/2 \rfloor$

for $i \leftarrow 1$ to N do

if $A(i) < A(n+1-i)$

then interchange($A(i)$, $A(n+1-i)$)

endif

repeat

$N \leftarrow \lceil n/2 \rceil$; $\max \leftarrow A(1)$; $\min \leftarrow A(n)$

for $i \leftarrow 2$ to N do

if $A(i) > \max$ then $\max \leftarrow A(i)$ endif

if $A(n+1-i) < \min$ then $\min \leftarrow A(n+1-i)$ endif

repeat

endif

end BMAXMIN

下面来说明该算法元素比较数$T(n) = \lceil 3n/2 \rceil - 2$.

证明：当$n = 2$时，等式成立；当$n > 2$时

$$T(n) = \lfloor n/2 \rfloor + 2\left(\lceil n/2 \rceil - 1\right)$$
$$= \left(\lfloor n/2 \rfloor + \lceil n/2 \rceil\right) + \lceil n/2 \rceil - 2$$
$$= n + \lceil n/2 \rceil - 2$$
$$= \lceil 3n/2 \rceil - 2$$

该算法的比较数达到了最坏情况下的下界，且不需要递归，即没有进出栈所带来的开销，因此应比前面两算法效率高，可说是最佳算法。

定理 2.5 任何以比较为基础求n个元素中最大和最小元的算法在最坏情况下至少需要$\lceil 3n/2 \rceil - 2$次比较。

证明：我们以一个四元组(a, b, c, d)来表示算法执行过程中的状态，其中，a是尚未参加过比较的元素数，b是在以往的比较中从未输过的元素数，c是在以往的比较中从未赢过的元素数，d是既输过也赢过的元素数。算法以状态$(n, 0, 0, 0)$起始，而应以$(0, 1, 1, n-2)$结束。状态(a, b, c, d)下进行一次比较而使状态发生变化成为下面五种情形（比较坏的）之一：

(1) $(a-2, b+1, c+1, d)$：如果$a \geq 2$且a中两元素进行比较；

(2) $(a-1, b, c+1, d)$：$a \geq 1$，a中一元素与b中一元素进行比较，结果b中元素赢；

(3) $(a-1, b+1, c, d)$：$a \geq 1$，a中一元素与c中一元素进行比较，结果a中元素赢；

(4) $(a, b-1, c, d+1)$：$b \geq 2$，b中两元素进行比较；

(5) $(a, b, c-1, d+1)$：$c \geq 2$，c中两元素进行比较。

{ (6) (a, b, c, d)：$b \geq 1$，$c \geq 1$，b中一元素与c中一元素比较，结果b中元素赢。状态不发生变化}

为了达到状态$(0, 1, 1, n-2)$，a必须由n变为0，只有(1)，(2)或

(3)三种比较对此作贡献，这至少需要$\lceil n/2 \rceil$次比较；而d也必须由0变为$n-2$，只有(4)或(5)两种比较对此作贡献，这至少需要$n-2$次比较，所以总共至少需要

$$\lceil n/2 \rceil + n - 2 = \lceil 3n/2 \rceil - 2$$

次比较。

作业：

石子游戏：有$n\,(n \geq 2)$个石子，甲乙两个玩如下规则的取石子游戏：

(1) 甲乙轮流取石子，甲先取;

(2) 设甲各次取的石子数分别为

$m_{甲}^1$，$m_{甲}^2$，$m_{甲}^3$，…

乙各次取的石子数分别为

$m_{乙}^1$，$m_{乙}^2$，$m_{乙}^3$，…

则要求

$$1 \leq m_{甲}^1 < n, \qquad 1 \leq m_{乙}^1 \leq 2m_{甲}^1;$$

$$1 \leq m_{甲}^2 \leq 2m_{乙}^1,\ 1 \leq m_{乙}^2 \leq 2m_{甲}^2;$$

$$1 \leq m_{甲}^3 \leq 2m_{乙}^2,\ 1 \leq m_{乙}^3 \leq 2m_{甲}^3;$$

$$\cdots$$

直到取完石子为止，最后一次拿到石子的一方赢!

问：会不会发生这样的情形：当n是某些整数时，甲总有一种策略必赢，而当n取另外某些整数时，乙总有一种策略必赢?

4. 排序

所谓排序(sorting)就是将一些可比较大小的对象按递增或递减的顺序进行排列。这类问题在计算机的处理工作中经常遇到。据早先统计，包括所有计算机的用户在内，在它们的计算机上，运行时间的四分之一以上是花在排序上，所以排序问题引起众多计算机专家的关注，陆续产生了一系列排序算法。

4.1 排序算法概述

第一个排序的计算机程序是由 Von Neumann 写于 1946 年，并在第一台计算机上运行，但真正开始研究排序算法是本世界 50 年代的事。最简单直观的排序算法是选择排序，冒泡排序和插入排序，但是，它们的运行时间都是$\Theta(n^2)$. 为提高排序速度，人们不断改进上述算法。1954 年，D. L. Shell 提出了缩小增量法，这是对插入排序的有效改进。1962 年，C.A.R hoare 提出了快速排序(quicksort)方法，它的平均时间复杂度是$\Theta(n\log n)$，而最坏情况下运行时间仍是$\Theta(n^2)$.1964 年，由 J. Williams 首先提出而后经 Floyd 改进的堆排序算法是一个既在平均时间又在最坏情况下都是$\Theta(n\log n)$的有效算法。这期间人们还提出了一些别的排序算法，如归并排序，基数排序等成熟的算法。并且，在这期间，人们也认识到了比较排序的时间下界是$\Theta(n\log n)$.

到了 70 年代，排序方法的改进就更精细了，Singeton 在 1969 年，Frazer 和 Mckellar 在 1970 年提出了对快速排序的改进. Blum, Floyd, Pratt, Rivest 和 Tarjan 给出了$O(n)$时间的求顺序统计量(一

般选择问题）的算法。随后 Floyd 和 Rivest 又在 1973 年对此作了改进。

稳定的排序算法是 Horvath 在 1974 年提出的，大多数简单算法是稳定的，而多数知名算法是不稳定的。迄今所知的$\Theta(n\log n)$的原地且又稳定性的算法见朱洪有关文章。

1973 年，D. E. Knuth 在他的名著"the art of computer programming"第三卷（有中译本）中，对排序算法做了全面总结，使排序算法更系统，更完善。

最后，还要提及的是，1980 年, D. E. Knuth 的学生 Sedgewick 在他的博士论文"Quicksort"中，对排序算法作了全面，精细的分析，他的分析是对一个计算机算法进行数学分析的典范。

排序（sorting)是计算机程序中的一种重要运算，它的功能是将一个数据元素（或记录）的任意序列，重新排列成一个按关键字有序的序列。

注：为查找方便，通常希望计算机中的表是按关键字有序的。

为便于讨论，在此首先要对排序下一个确切的定义。

定义 2.1 假设含n个记录的序列为

$$\{R_1, R_2, \cdots, R_n\}$$

其相应的关键字序列为

$$\{k_1, k_2, \cdots, k_n\}$$

需要确定$1, 2, \cdots, n$的一种排列$p_1, p_2, \cdots, p_n$，使关键字序列

满足如下的非递减关系：

$$k_{p_1} \le k_{p_2} \le \cdots \le k_{p_n}$$

从而把记录序列重排为按关键字有序的序列

$$\{R_{p_1}, R_{p_2}, \cdots, R_{p_n}\}$$

这样的一种操作称为排序。

为称谓方便，在不引起混淆的情况下，我们将把关键字的有序性等同于记录或数据元素本身的有序性。

若n元序列$\{a_1, a_2, \cdots, a_n\}$中有相同的元素，则排序结果是不唯一的。当然，任何排序算法只给出一种结果，我们希望排序算法具有下面定义的稳定性。

定义 2.2 对任意$n>1$和任意n元序列$\{a_1, a_2, \cdots, a_n\}$,排序算法$A$总能给出关于$1, 2, \cdots, n$的排列$P$, 使其满足

(1) $a_{p_i} \le a_{p_{i+1}}$（$1 \le i \le n-1$）；

(2) 当$a_{p_i} = a_{p_j}, i < j$时，必有$p_i < p_j$，即序列中任意两个相同元素排序前后的顺序关系保持不变.

则称排序算法是稳定的。

注：稳定排序算法的提法背景，含多个关键字（具有优先顺序）的记录排序问题。

由于待排序的记录规模不同，使得排序过程中涉及的存储器不同，可将排序方法分为两大类：一类是内部排序，指的是待排序记录存放在计算机随机存储器中进行的排序过程；另一种是外部排序，指的是待排序记录的数量太大，以致内存不能一次容纳全部记录，在排

序过程中尚需对外存进行访问的排序过程。

本节集中讨论内部排序。

内部排序方法基本可分为以下几类：

一. 插入排序

1. 直接插入

2. 折半插入（二分插入）

3. 两路插入

4. 歇尔方法

二. 交换排序

1. 冒泡排序

2. 快速排序

三. 选择排序

1. 直接选择

2. 树选择

3. 堆选择

四. 归并排序

五. 分布排序

1. 基数排序

2. 映射排序

4.1 插入排序

最直接的排序方法是插入法，算法具体描述如下：

算法 2.6 插入排序

procedure INSERTISORT(A, n)

// 将$A(1:n)$中的元素按非降次序排列，$n \geq 1$.//

integer i, j

$A(0) \leftarrow$ "$-\infty$" //开始时生成一个虚拟值"$-\infty$"，它比中所有//

//元素都小.//

for $j \leftarrow 2$ to n do //$A(1: j-1)$已排好序//

$item \leftarrow A(j)$; $i \leftarrow j-1$

while $item < A(i)$ do //$0 \leq i < j$//

$A(i+1) \leftarrow A(i)$; $i \leftarrow i-1$

repeat

A(i+1) $\leftarrow$ item

repeat

end INSERTISORT

算法的基本操作：赋值与比较。while 体中的语句可能执行0次到$j-1$次，而j从2变到n，因此，这过程的最坏情况限界是

$$3n-2+\sum_{j=2}^{n} 2(j-1) = 3n-2+n(n-1);$$

如果输入数据本来就是非降次序排列的，则根本不会进入 while 的循环体，这是最好的情况，基本操作次数是：$3n-2$.

下面来分析其平均时间复杂度：仅考虑要排序的集合$S=\{a_1, a_2, \cdots, a_n\}$中的元素均不相同的情况，由于将$S$输入时

其元素是按某种顺序排列在数组$A(1:n)$中的，这种排列方式共有$n!$种，假定所有排列方式等概出现。对排列

$$A(1),\ A(2),\ \cdots,\ A(n)$$

定义如下反序概念：若$i<j$，但$A(i)>A(j)$，则称$\{(i,\ A(i)),\ (j,\ A(j))\}$是$A$的一个反序，设$A$的反序数是$I(A)$，则插入算法在输入$A$的情况下的基本操作数是：$3n-2+2I(A)$).

注意到A的反排列A^{-1}:

$$A(n),\ A(n-1),\ \cdots,\ A(1)$$

的反序数$I(A^{-1})$与$I(A)$的和为$n(n-1)/2$. 这是因为S中任二元素$A(i),\ A(j)$要么在排列A中构成反序$\{(i,\ A(i)),\ (j,\ A(j))\}$，要么在排列$A^{-1}$中构成反序$\{(n+1-i,\ A(i)),\ (n+1-j,\ A(j))\}$，且二者仅有其中之一成立。于是插入算法的平均时间复杂度

$$MT(n)=\frac{2}{n!}\sum_{A\in S_n} I(A)+3n-2, \qquad (2.30)$$

其中S_n是S中n个元素所有不同排列构成的集合。同时

$$MT(n)=\frac{2}{n!}\sum_{A\in S_n} I(A^{-1})+3n-2, \qquad (2.31)$$

(2.30)+(2.31)得到

$$MT(n)=\frac{1}{n!}\sum_{A\in S_n}(I(A)+I(A^{-1}))+3n-2$$

$$=\frac{1}{n!}\sum_{A\in S_n}n(n-1)/2+3n-2$$

$$=n(n-1)/2+3n-2$$

$$=n(n+5)/2-2$$

尽管直接插入法在最坏情况和平均情况下的时间复杂度都是$\Theta(n^2)$，由于其简单性，当n相当小或输入的元素列几乎有序的情况下，采用这种算法还是相当理想的。

二分检索树插入法：第一个元素$A(1)$作为单结点的树T_1，以下列递归方法生成$A(1:j)$($j\geq 2$)中元素的排序树T_j:

(1) 若$A(j)<T_{j-1}$的根结点，则(a) 若T_{j-1}无左子树，则将$A(j)$作为其左子树； (b) 否则，以递归方法将$A(j)$插入其左子树;

(2) 若$A(j)\geq T_{j-1}$的根结点，则(a) 若T_{j-1}无右子树，则将$A(j)$作为其右子树； (b) 否则，以递归方法将$A(j)$插入其右子树。

注：该排序算法是稳定的。

例：$(1,\ 2,\ 3)$，$(2,\ 1,\ 3)$，$(1,\ 3,\ 2)$，$(3,\ 1,\ 2)$，$(2,\ 3,\ 1)$，$(3,\ 2,\ 1)$分别对应的二分检索树如下图:

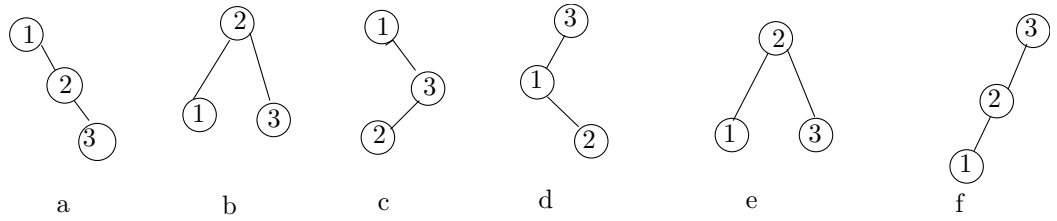


Fig 2.3

每次插入一个结点x的花费(比较次数)是该结点到根的路径长度。

算法的最坏时间复杂度:

$$WT(n)=0+1+2+\cdots+n-1=n(n-1)/2.$$

如果S中元素各不相同，在$n!$种输入中，总共有2^{n-1}种输入达到最坏时间复杂度。

下面来计算算法的平均时间复杂度$MT(n)$。同样假设S中元素各不相同，不妨假定：$a_1<a_2<\cdots<a_n$，并设这n个元素的每个排列$A(1:n)$（总共$n!$个）作为输入数组出现的概率相同。记这些排列组成的集合为S_n。则算法的平均时间复杂度可表示为

$$MT(n)=\frac{1}{n!}\sum_{A\in S_n} i(B(A)), \qquad (2.32)$$

其中$i(B(A))$表示由$A(1:n)$中元素生成的排序树的各结点到根的路径长度总和。

下面来设法得到关于$MT(n)$的递归关系式。为表示方便，约定

$$MT(0)=0, \qquad (2.33)$$

当$n>0$时，

$$i(B(A))=n-1+i(L(A))+i(R(A)),$$

因此，

$$MT(n)=\frac{1}{n!}\sum_{A\in S_n} i(B(A))$$

$$=\frac{1}{n!}\sum_{A\in S_n}(n-1+i(L(A))+i(R(A)))$$

$$=n-1+\frac{1}{n!}\sum_{A\in S_n} i(L(A))+\frac{1}{n!}\sum_{A\in S_n} i(R(A))$$

$$=n-1+\frac{2}{n!}\sum_{A\in S_n} i(L(A)) \qquad (2.34)$$

设S_n^j是使$B(A)$的根为S中第j小元素的那些排列$A(1:n)$组成的集合，即

$$S_n^j=\{A\in S_n : A(1)=a_j\}$$

则

$$S_n=\bigcup_{j=1}^{n} S_n^j \qquad (2.35)$$

并且，当$i\neq j$时

$$S_n^i\cap S_n^j=\varnothing \qquad (2.36)$$

当$A\in S_n^j$时，$L(A)$完全由前$j-1$个元素$a_1,\cdots,a_{j-1}$在A中相对排列方式决定，每个固定的这种相对排列方式对应着S_n^j中的

$$C_{n-1}^{j-1}\cdot(n-j)!=(n-1)!/(j-1)!$$

种排列方式。设S_{j-1}是$a_1,\cdots,a_{j-1}$的$(j-1)!$种排列组成的集合，对任意$\sigma\in S_{j-1}$，记

$$S_n^{j,\sigma}:=\{A\in S_n^j:\ a_1,\cdots,a_{j-1}\text{在}A\text{中的相对排列}=\sigma\}$$

则

(1) $S_n^j = \bigcup_{\sigma \in S_{j-1}} S_n^{j,\sigma}$,

(2) 当$\sigma_1, \sigma_2 \in S_{j-1}$，$\sigma_1 \neq \sigma_2$时，

$$S_n^{j,\sigma_1} \bigcap S_n^{j,\sigma_2} = \varnothing.$$

(3) 对于任意$\sigma \in S_{j-1}$，都有

$$| S_n^{j,\sigma} | = (n-1)!/(j-1)!.$$

因此

$$\sum_{A \in S_n^{j,\sigma}} i(L(A)) = \sum_{A \in S_n^{j,\sigma}} i(B(\sigma)) = (n-1)!/(j-1)! i(B(\sigma))$$

从而

$$\sum_{A \in S_n^j} i(L(A)) = \sum_{\sigma \in S_{j-1}} \sum_{A \in S_n^{j,\sigma}} i(L(A))$$

$$= \sum_{\sigma \in S_{j-1}} (n-1)!/(j-1)! i(B(\sigma))$$

$$= (n-1)!/(j-1)! \sum_{\sigma \in S_{j-1}} i(B(\sigma))$$

$$= (n-1)! MT(j-1) \qquad (2.37)$$

由(2.34—2.37)得到: 当$n > 0$时，

$$MT(n) = n - 1 + \frac{2}{n!} \sum_{j=1}^{n} \sum_{A \in S_n^j} i(L(A))$$

$$= n - 1 + \frac{2}{n!} \sum_{j=1}^{n} (n-1)! MT(j-1)$$

$$= n - 1 + \frac{2}{n} \sum_{j=1}^{n} MT(j-1) \qquad (2.38)$$

下面来解此递归式。将式(2.38)变形为

$$n \cdot MT(n) = n(n-1) + 2\sum_{j=1}^{n} MT(j-1)$$

分别以$n = k$，$n = k-1$代入上式得

$$k \cdot MT(k) = k(k-1) + 2\sum_{j=1}^{k} MT(j-1) \qquad (2.39)$$

$$(k-1) \cdot MT(k-1) = (k-1)(k-2) + 2\sum_{j=1}^{k-1} MT(j-1) \qquad (2.40)$$

(2.39)–(2.40)得

$$k \cdot MT(k) - (k-1)MT(k-1) = 2(k-1) + 2MT(k-1)$$

整理且令$k = n$得

$$n \cdot MT(n) = (n+1) \cdot MT(n-1) + 2(n-1) \qquad (2.41)$$

上式两边同除以$2n(n+1)$得

$$\frac{MT(n)}{2(n+1)} = \frac{MT(n-1)}{2n} + \frac{n-1}{n(n+1)}$$

$$= \frac{MT(n-1)}{2n} + \frac{2}{n+1} - \frac{1}{n} \qquad （2.42）$$

接连运用（2.42）得到

$$\frac{MT(n)}{2(n+1)} = \frac{MT(0)}{2} + \left(\frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{2}\right) - \left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1}\right)$$

$$= \frac{2}{n+1} + \left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1}\right) - 2 = H(n) - \frac{2n}{n+1}$$

$$= \ln\ n + E_n - \frac{2n}{n+1}$$

最后得到

$$MT(n) = 2(n+1)\ln\ n - 4n + (n+1)E_n \qquad (2.43)$$

用分治策略来设计排序算法，可得到两种排序算法：归并排序(Mergesort)，快速排序(Quicksort)。

这两种排序的主要特征分别是

归并排序(Mergesort): 分割易(easy division), 组合难(hard combination);

快速排序(Quicksort): 分割难(hard division), 组合易(easy combination).

4.2 归并排序

归并排序可使最坏时间复杂度变为$\Theta(n\log n)$。其基本思想是：将 $A(1), \cdots, A(n)$ 分成两个集合 $A(1), \cdots, A(\lfloor (n+1)/2 \rfloor)$ 和 $A(\lfloor (n+1)/2 \rfloor + 1), \cdots, A(n)$，对每个集合分别归并排序，然后将已排好序的两个序列归并成一个有序的序列。

算法 2.7 归并排序

procedure MERGESORT($A, low, high$)

// $A(low: high)$ 是一个全程数组，它含有 $high - low + 1 \geq 1$ 个//

//待排序的元素//

integer $low, high, mid$

if ($low < high$) then $mid \leftarrow \lfloor (low + high)/2 \rfloor$ //分割点 //

call MERGESORT(A, low, mid)

//子集合排序//

call MERGESORT($A, mid+1, high$)

//另一子集合排序//

call MERGE($A, low, mid, high$) //归并//

endif

end MERGESORT

算法 2.8 使用辅助数组归并两个已排序的集合

Procedure MERGE($A, low, mid, high$)

//$A(low, high)$是一个全程数组，它含有两个分别放在//

//$A(low, mid)$和$A(mid+1, high)$中已排好序的子集合. 目标//

//是将这两个已排好序的集合按顺序归并成序列并存放到//

//$A(low, high)$.使用辅助数组$B(low, high)$.//

integer $h, i, j, k, low, mid, high$

global $A(low: high)$; local $B(low, high)$

$h \leftarrow low;\ i \leftarrow low;\ j \leftarrow mid+1$

while $i \le mid$ and $j \le high$ do //当两个集合都没取尽//

if $A(i) \le A(j)$ then $B(h) \leftarrow A(i);\ i \leftarrow i+1$

else $B(h) \leftarrow A(j);\ j \leftarrow j+1$

endif

$h \leftarrow h+1$

repeat

if $i > mid$ then for $k \leftarrow j$ to $high$ do // 处理剩余元素//

$$B(h) \leftarrow A(k);\ h \leftarrow h+1$$

repeat

else for $k \leftarrow i$ to mid do

$$B(h) \leftarrow A(k);\ h \leftarrow h+1$$

repeat

endif

for $k \leftarrow low$ to $high$ do //把已归并的集合复制到A//

$$A(k) \leftarrow B(k)$$

repeat

end MERGE

注：n 个元素最初存放在 $A(1:\ n)$，调用 call MERGESORT$(A,\ 1,\ n)$ 将使这 n 个元素排好序且仍存于 $A(1:\ n)$ 中.

下面来分析其时间复杂度。从根本上说，关键在于分析归并所需的比较次数。

归并两个已排好序的序列 $A(1:\ n)$ 和 $B(1:\ m)$，当 m 远远小于 n 时，用二分插入法较优，仅考查 $m=1$ 时，所需比较的最大次数为 $\lfloor \log(n+1) \rfloor$； 但当 m 与 n 大小相差不多时，所需比较的最大次数差不多为 $m+n-1$, 有下面定理为证。

定理 2.6 任何以比较为基础的算法归并排好序的两个序列 $A(1:\ n)$

与$B(1:\ n)$在最坏情况下所需的比较次数不少于$2n-1$.

证明：在算法 2.8 中，归并两个已排好序的序列$A(1:\ n)$与$B(1:\ m)$最多需要的比较次数是$m+n-1$. 下面来说明当$m=n$时，这个上界是任何归并算法在最坏情况下都能达到的。我们来考查满足

$$B(1)<A(1)<B(2)<A(2)<\ \cdots\ <B(n)<A(n)$$

的两个待归并的序列，任何归并算法在归并此两个序列时必须执行下面$2n-1$个比较：

$B(1):A(1)$，　$A(1):B(2)$，$B(2):A(2)$，$\cdots$，

$A(n\text{-}1):B(n)$，$B(n):A(n)$

假如对某i，$B(i):A(i)$没有被执行，则该算法在输入这两个序列的情况下输出的结果与输入满足下式的两个序列时必然一样：

$$B(1)<A(1)<\cdots<B(i-1)<A(i-1)<A(i)<B(i)$$
$$<B(i+1)<A(i+1)<\cdots<B(n)<A(n),$$

显然必有一种输出是错误的，同样道理，若对某i，$A(i):B(i+1)$没被执行的话，则该算法在输入这两个序列的情况下输出的结果与输入满足下式的两个序列时必然一样：

$$B(1)<A(1)<\cdots<B(i-1)<A(i-1)<B(i)<B(i+1)$$
$$<A(i)<A(i+1)<\cdots<B(n)<A(n),$$

同样必有一种输出是错误的，得证。

注：从证明过程来看，应有：当$|m-n|=0,\ 1$时，任何以比较为基础的算法归并排好序的两个序列$A(1:\ n)$与$B(1:\ m)$在最坏情况下所需的比较次数不少于$m+n-1$. 定理说明归并算法 2.8 就比

较次数而言在最坏情况下是最优的。

归并排序算法 2.7 的最坏时间复杂度（基本操作是元素间的比较）满足下列递归式：

$$\begin{cases} WT(1)=0, \\ WT(n)=WT(\lfloor n/2 \rfloor)+WT(\lceil n/2 \rceil)+n-1,\ n\geq 2 \end{cases} \tag{2.44}$$

由 **master** 定理(情形(2))立即可知道, $WT(n)=\Theta(n\log n)$, 于是我们得到了一种最坏时间复杂度为$\Theta(n\log n)$的排序算法。关于归并排序算法平均时间复杂度的分析，我们宁愿稍微往后放一放，由一个更为一般的结论来涵盖。

事实上，对归并排序在最坏情况下的比较次数作更为精确的估计是一件有趣的事。

我们来考查归并排序形成的递归树（如 Fig 2.5 所示）。 注意到树的第d ($0\leq d\leq D-2$, 树的最深级$D=\lceil \log n \rceil$)级只含内点(这儿，内点指还需要继续分割的点)， 共有2^d个, 归并所花费的比较数总共是$n-2^d$; 树共有n个外点(指不需要分割的点，或者说，表示对 1 个元素进行排序的问题),外点处的花费是0. 外点只可能在第D级或第$D-1$级上，不管具体情况如何，第$D-1$级的结点(包括内结点和外结点)数是2^{D-1}，归并所花费的比较数仍是$n-2^{D-1}$，第D级只含外点，花费比较数是0。

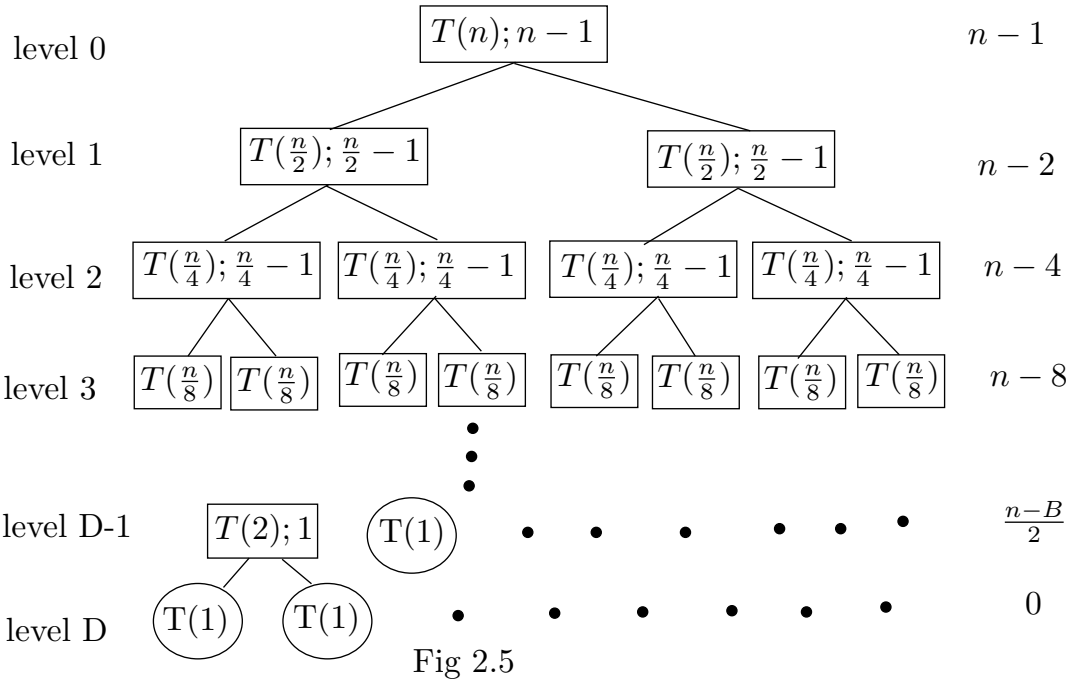


Fig 2.5

基于以上事实，我们有

$$WT(n)=\sum_{d=0}^{D-1}(n-2^d)$$

$$=nD-2^D+1 \qquad (2.45)$$

我们再作进一步估计，令$\alpha_n=2^D/n$，则

$$D=\log n+\log\alpha_n, \qquad (2.46)$$

将(2.46)代入(2.45)得

$$WT(n)=n\log\ n-(\alpha_n-\log\alpha_n)n+1.$$

注意到

$$1\le\alpha_n<2,$$

从而

$$(1+\ln(\ln 2))/\ln 2\le\alpha_n-\log\alpha_n\le 1$$

$$((1+\ln(\ln 2))/\ln 2\approx 0.914)$$

（注：计算函数 $f(x)=x-\log x$，$1\le x<2$ 的值域）

最后我们得到：

$$n\log\ n-n+1\le WT(n)\le n\log\ n-0.914n$$

4.3 快速排序(Quicksort)

快速排序是较早发现的分治排序算法，首先由霍尔(C.A.R.Hoare)发表于 1962 年。快速排序也是实际运算最快的排序算法。

快速排序的基本思路如下：取待排序的 n 个元素 $A(low:high)$ 中的某个元素 v（譬如取 $v=A(high)$），确定 v 在这 n 个元素升序排列应处的位置 i，且重排 $A(low:high)$ 中元素使得 $A(low,i-1)$ 中的元素都不大于 v，$A(i)=v$，$A(i+1:high)$ 中的元素都不小于 v。然后对 $A(low,i-1)$ 和 $A(i+1:high)$ 分别进行快速排序。

算法 2.9 快速排序

Procedure Quicksort(low, $high$, A)

// $A(low:\ high)$ 是一个全程数组，它含有 $high-low+1\ge 0$ 个//

//待排序的元素。//

Integer i, j; Local v

If ($high>low$) then

{

$v\leftarrow A(high)$; $i\leftarrow low-1$; $j\leftarrow high$

loop

loop $i \leftarrow i+1$ until $A(i) \geq v$

loop $j \leftarrow j-1$ until $A(j) \leq v$

$temp \leftarrow A(i);\ A(i) \leftarrow A(j);\ A(j) \leftarrow temp$

until $j \leq i$

$temp \leftarrow A(i);\ A(i) \leftarrow A(high);\ A(high) \leftarrow temp$

Quicksort($low,\ i-1,\ A$)

Quicksort($i+1,\ high,\ A$)

}

endif

end Quicksort

分析：同样以元素比较为基本操作，快速排序的最坏时间复杂度为$\Theta(n^2)$（这发生在$A(1:n)$中元素递增或递减排列时），平均时间复杂度为$\Theta(n\log n)$。事实上，设快速排序的平均时间复杂度（假定这些元素两两不同）为$T(n)$，则有下列递归式：

$$\begin{cases} T(1)=0 \\ T(n)=n-1+(2/n)\sum_{i=1}^{n-1}T(i) \end{cases}$$

与二元树插入排序算法的的平均时间复杂度$T(n)$满足的递推关系完全相同。

4.4 比较排序算法时间复杂度下界分析

任何以比较为基础的排序算法A对n个元素作排序的情况都可以用一个二元比较树(或二元决策树)来表示。该树的每个内结点表示一次比较操作，比较的结果总是将算法导向两个不同的状态之一；而该树的每个外结点表示一种可能的排序结果。例

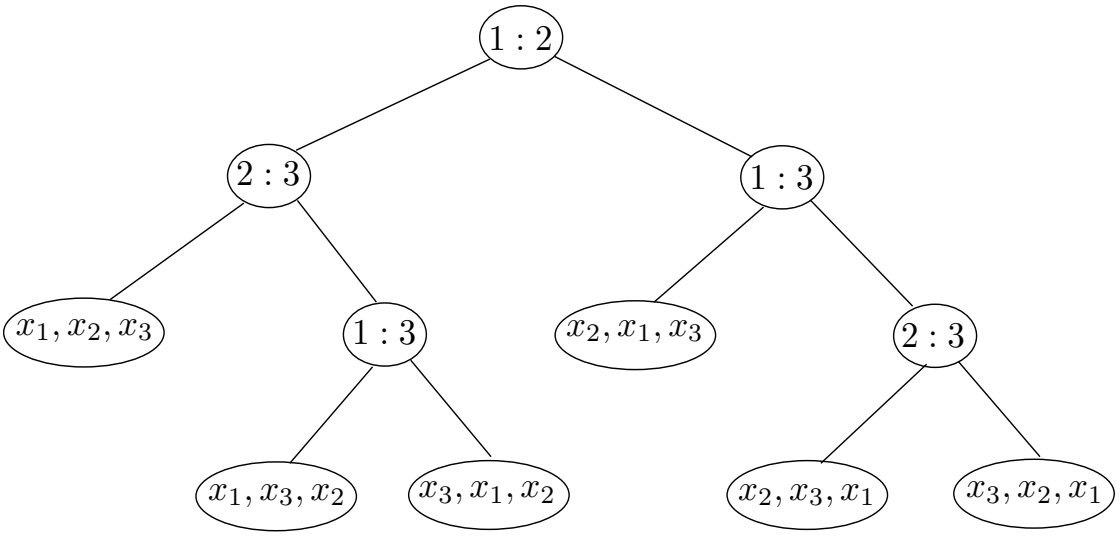


Fig 2.4 Decision tree for a sorting algorithm,n=3

所以该树至少有$n!$个外结点(假设n个元素两两不同)。一个具体的输入所需化费的比较数恰是表示所对应输出的外结点到根的距离。因此，算法A的最坏情况比较数是该树的高度，而平均比较数是该树所有外结点的级数的平均值。

任何二元树的高度h与外结点数目L有如下关系：$L \le 2^h$，等价地，$h \ge \lceil \log L \rceil$. 所以二元决策树的高度$h \ge \lceil \log(n!) \rceil$,

$$\log(n!) = \sum_{i=1}^{n} \log i = \sum_{i=2}^{n} \log i > \sum_{i=2}^{n} \int_{i-1}^{i} \log x \, dx$$

$$= \int_{1}^{n} \log x \, dx = \log e \int_{1}^{n} \ln x \, dx = \log e \cdot (x \ln x - x) \Big|_{1}^{n}$$

$$= \log e(n\ln n - n + 1) = n\log n - n\log e + \log e$$

$$> n\log n - 1.443n + 1$$

另外，在所有含相同个数外结点的二元树中，以均衡二元树的外结点的平均级数为最小。均衡二元树的外结点的级数是h或$h-1$.

综上所述，任何比较排序算法的最坏情况比较数不小于$\lceil n\log n - 1.443n \rceil + 1$, 平均情况比较数不小于$n\log n - 1.443n$.

作业

1．下列归并排序算法是数据无关的，请给出以比较为基本操作的时间复杂度。

```
procedure  MERGESORT(A, low, high)
// A(low: high) 是一个全程数组，它含有 high − low + 1 ≥ 0 个//
//待排序的元素, 使用辅助数组 B(1: ⌊(high − low)/2⌋ + 2) 和//
//C(1:⌈(high − low)/2⌉ + 1) 进行归并//
integer  low, high, mid
if (low < high) then
    {
     mid ← ⌊(low + high)/2⌋   //分割点 //
     call  MERGESORT(A, low, mid)
      //子集合排序//
     call  MERGESORT(A, mid + 1, high)
                    //另一子集合排序//
```

for $i \leftarrow 1$ to $mid - low + 1$ do

$B(i) \leftarrow A(i + low - 1)$

repeat

for $j \leftarrow 1$ to $high - mid$ do

$C(j) \leftarrow A(j + mid)$

repeat

$B(mid - low + 2) \leftarrow \max;\ C(high - mid + 1) \leftarrow \max$

$i \leftarrow 1;\ j \leftarrow 1$

for $k \leftarrow low$ to $high$ do

if $B(i) \le C(i)$ then $A(k) \leftarrow B(i);\ i \leftarrow i + 1$

else $A(k) \leftarrow C(j);\ j \leftarrow j + 1$

endif

repeat

}

endif

end MERGESORT

设$t(n)$是上述算法对n个元素排序所花费的比较次数，则有

$$\begin{cases} t(1) = 0 \\ t(n) = t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + n,\ n \ge 2 \end{cases}$$

请证明：$t(n) = n\lceil \log n \rceil + n - 2^{\lceil \log n \rceil}$.

2．请设计一个算法，计算n个元素（可比较大小）组成的排列的反序数，要求算法的时间复杂度为$O(n \log n)$。

3．考虑幂函数 $f(x)=x^n$ 的数值计算问题，其中 x 是正实数，n 是正整数。请编制时间复杂度为 $O(\log n)$ 的算法求解该问题。并证明以实数乘法为基本操作求解该问题的任何算法的时间复杂度 $t(n)$ 都满足：$t(n)=\Omega(\log n)$。

4. 给定实数数组 $A(1:n)$ 和实数 x，求解是否存在元素 $A(i)$，$A(j)$（$i\neq j$），使得 $x=A(i)-A(j)$。请给出最坏时间复杂度为 $O(n\log n)$ 的算法求解该问题。

5．矩阵 $A=(a_{i,j})_{m\times n}$ 中的元素取自某有序集，每行元素从左到右呈非递减排列。若将 A 中每列元素排序为自上而下非递减的，得到矩阵 $\overline{A}$，证明：$\overline{A}$ 中每行元素依然从左到右呈非递减排列。

5. 选择问题与魔鬼对策方法

本节研究几个可统称为选择的问题，找 n 个元素(可比较大小)的中间元，就是其中一个著名的例子。我们除了设计解决这些问题的高效算法外，还将探讨这些问题的下界估计。我们通过引入一种被广泛应用的称为魔鬼对策的方法来做这件事。

5.1 选择问题

假设 E 是一个含有 n 个可比较大小元素的集合，k 是一个正整数，$1\le k\le n$，选择问题就是寻找 E 中第 k 小元素的问题。我们约定可作的操作是：两元素的比较；元素拷贝或移动。而我们通常不太关心元素的移动。

除了找最大或最小元外，另一个常见的选择问题是找中间元，即

在E中找第$\lceil n/2 \rceil$小元素。中间元在很多以数据表示的事件中是有典型意义的，譬如说，要说明某个国家或某个行业人员的收入，房屋的价格，大学录取分数线，这些涉及处理数据的事情，往往归结为求平均数或中间数的问题。平均数可在$\Theta(n)$时间内容易求得，那么如何有效的来计算中间元呢？

当然，所有的选择问题实例可通过对E排序来解决：$E(k)$正是第k小元素。排序需要$\Theta(n\log n)$数量的比较操作，而我们已经注意到对某些k (如，$k=1,\ k=n$)，选择问题可在线性时间内解决。直观上，找中间元应是最困难的选择问题。那么，我们能在线性时间内找出中间元吗？或者，我们能确定找中间元的时间复杂度的下界是线性甚至是$\Theta(n\log n)$吗？我们将在这一节回答这些问题，并给出求解一般选择问题的算法框架。

5.2 下界估计

到目前为止，我们是利用决策树作为主要手段来对以比较为基本操作的问题进行下界估计的。在表示解决问题的任何算法的决策树中，每个内点表示算法可能执行的一次比较，而每个外结点表示一种可能的输出(在检索算法的情形下，每个内点也同时表示一种可能的输出)。算法在最坏情况下执行的比较次数正是决策树的高度(或深度)，该高度至少为$\lceil \log\ L \rceil$,其中，L是决策树的叶子（外结点）数目。

在检索问题中，决策树得到的下界估计是$\lceil \log(n+1) \rceil$, 能被二

分检索算法达到，这当然是最理想的；而在排序问题中，决策树得到的下界估计是$\lceil \log(n!) \rceil$，归并排序算法的复杂度几乎可以达到此界， 也相当令人满意。 但企图用决策树方法对选择问题作下界估计却不能奏效。

任何选择问题的决策树至少有n片叶子(作为输出)，因为n个元素的任何一个都可能是第k小元素，所以树的高度(即算法最坏情况下执行的比较数)至少是$\lceil \log n \rceil$，这不是一个好的下界估计。我们知道，最简单的选择问题（找最大元或最小元）在最坏情况下至少需要$n-1$次比较。问题出在哪儿呢？在找最大元的决策树中，可能由多片叶子来表示同一种输出。 但我们没有一种简易的方法一般性地确定任何一种输出究竟由多少片叶子来表示。

为此，我们采用一种称之为魔鬼对策(Adversary Argument)的方法，来取代决策树方法，对选择问题进行下界估计。

5.3 魔鬼对策

什么叫魔鬼对策呢？比方说，以警察捉小偷类比算法求解问题：把算法比作警察，输入比作小偷，算法对输入给出正确的输出比作警察捉住小偷。要对算法在最坏输入下的时间耗费作一个好的下界估计，必须把最坏的输入设想成一个狡猾的小偷，只要有机可乘，总是能逃脱警察的追捕，即让算法的每一步执行得到尽量差的结果。这就是魔鬼对策。再看一个更具体一些的例子，假定你与一个朋友做猜日期的游戏：你有权随意决定一天(365 天中)让朋友通过提出“是否问

题”来猜，随意的意思是你不必事先确定这个日子，而视朋友的实际发问而定，尽量不让他猜着，直到被逼出一个日子为止。譬如，朋友问：“这一天是在冬天吗？”你当然回答：“不是!”因为不是冬天的日子更多。再如，朋友问：“这一天所在的月份的第一个字母（英文）是在字母表的前一半吗？”你应回答：“是!”，在此例中，你扮演的正是魔鬼对手的角色。

现在假定我们有一个以比较为基本操作的有效算法，设想存在一个魔鬼对手在背后操纵着它的输入，致使算法执行每次比较时，由魔鬼来决定比较的结果，魔鬼选择的结果总是使算法的进展缓慢，而魔鬼必须遵循的唯一规则是相容性：必存在一种输入，使算法在该输入下的运行，与魔鬼选择的所有结果一致。如果魔鬼迫使算法必须至少执行 $f(n)$ 次比较，则 $f(n)$ 就是算法在最坏情况下时间复杂度的一个下界估计。魔鬼的魔法越高，得到的估计越好。

换一个角度，反过来看，“防御魔鬼对策”是有效求解基于比较操作的问题的好方法。就是说，设计算法时，尽量使得每次比较的结果相对均衡(两种结果的期望值一样)，以至魔鬼挑选哪一种，都区别不大。关于这一点，我们稍后再详细讨论，当下，我们的兴趣还是在用魔鬼对策方法估计下界。

我们想对一个问题的复杂度估计下界，而不是针对一个特殊的算法。所以，当我们采用魔鬼对策时，考虑的是某个算法类中的任意一个算法，就象我们应用决策树那样。为得到好的估计，我们必须请出大神通的魔鬼来挫折任何算法。

竞赛术语：本节我们要给出选择问题的算法，以及一些情形下估计下界的魔鬼对策。在很多时候，我们采用竞赛术语，来表示比较结果。两比较元中的较大者称为胜者(winner),而另外一个称为败者(loser).

5.4 找最大与最小元问题的再讨论

本小节中，我们用 max 和 min 分别表示n个元素中的最大与最小元。

我们已经知道，单找 max 或 min 用$n-1$次比较就可以了，这样找完 max 后再找 min 需$n-1=n-1+(n-1-1)=2n-3$次比较。但这样做不是最优的方法，尽管$n-1$次比较对单找 max 或 min 是最优的，因为同时找二者的话，能使某些比较结果为二者所共享，一种设想是先让n个元素捉对比较(总共$\lfloor n/2 \rfloor$次)，然后在胜者(总共$\lceil n/2 \rceil$个元素)中找 max，在败者(也是总共$\lceil n/2 \rceil$个元素)中找 min，由此设想得到的算法耗费的比较数为$\lfloor n/2 \rfloor + 2\left(\lceil n/2 \rceil - 1\right) = \lceil 3n/2 \rceil - 2$. 下面用魔鬼对策来证明该算法是最优的。

定理 2.7 任何以比较为基础的找n元中 max 和 min 的算法在最坏情况下至少需$\lceil 3n/2 \rceil - 2$次比较。

证明：为得此下界，我们仅考虑输入为n个两两不同元素的情形就够了。 为确定某元素x是 max 和某元素y是 min, 算法必须要知道异于x的元素输过比较，异于y的元素赢过比较. 如果我们把某元素能赢或会输都记为一个信息单位，则算法必须得到$2(n-1)$个信息单位

才能确保结果正确。

我们的魔鬼对策是让算法在每一次比较中得到尽量少的新信息单位。具体说来（如表 2.1 所示），就是如果比较是发生在从未参加过比较的两元之间，则一次比较可得到两个新信息单位，其他比较每发生一次至多允许得到一个新信息单位. 前一种比较的个数记为s，后一种比较的个数记为t，则总比较数为$s+t$，且有：

$$s \le \lfloor n/2 \rfloor, \quad 2s+t \ge 2(n-1),$$

由此得：

$$s+t = 2s+t-s \ \ge \ 2n-2-\lfloor n/2 \rfloor = \lceil 3n/2 \rceil - 2 。$$

所有比较结果列表

比较类型	结果	新增加信息单位
$N:N$	$N<N$	2
$N:W$	$N<W$	1
$N:L$	$N>L$	1
$N:WL$	$N>WL$	1
$W:W$	$W>W$	1
$W:L$	$W>L$	0
$W:WL$	$W>WL$	0
$L:L$	$L>L$	1
$L:WL$	$L<WL$	0
$WL:WL$	$WL<WL$	0

表 2.1

注：N表示没参加过比较的元素，W表示没输过的元素，LW表示没赢过的元素，WL表示赢过也输过的元素。

5.5 找第二大元素

我们当然可以找到n个元素中的 max 后, 再在剩余的$n-1$个元素中找 max 就得到 second-largest. 但会不会还有更有效的方法？进一步说，我们能证明某种方法是最优的吗？本小节回答这些问题。

因为要确定某元x是 second-largest, 必得知道x小于某元y，而又大于其它$n-2$个元素。显然y必是最大元 max。所以在算法中，第一步先找 max 是个合理的设想。只是在找 max 的步骤中，希望提供尽可能多的信息使接下来找 second-largest 元花费的比较数尽量少。我们注意到， 在找最大元 max 的过程中，max 以外的其他$n-1$个元素必输过，若元素z输过非最大元 max 的某个元素， 则z不可能是 second-largest。所以，我们仅在输过 max 的元素中找 second-largest 就够了。那么，如何在第一步找 max 的过程中，使与 max 比较过的元素数最少呢？我们采用锦标赛方法(Tournament Method)可做到这一点.

锦标赛方式：第一轮：所有元素分对比较；第二轮：第一轮中赢的元素分对比较；第三轮：第二轮中赢的元素分对比较；……，直到决出最大元 max. 锦标赛过程可用一个二元树来表示，这是一个从底(bottom)到顶(top)的方式：最底层每片叶子包含一个元素(n片叶子)，配对的元素的父亲是其中的胜者(winner), 而最顶部的根包含的

是 max. 在此锦标赛过程中，除 max 外，其它元素输过一次且仅一次，那么有多少元素输给 max 呢？如果$n=2^k$, 则恰有$\log n$个，一般地，该数目最坏为$\lceil \log n \rceil$ (注意到$\lceil \log \lceil n/2 \rceil \rceil + 1 = \lceil \log n \rceil$, 可用数学归纳法证明)，所以在第二步中，至多需$\lceil \log n \rceil - 1$次比较即可找到 second-largest, 总共花费$\lceil \log n \rceil - 1 + n - 1 = n + \lceil \log n \rceil - 2$次比较，下面我们来证明锦标赛方法是最优的。

定理 2.8 任何以比较为基础的找n个元素中 second-largest 元的算法在最坏情况下至少需$n + \lceil \log n \rceil - 2$次比较。

证明：我们用一个无向图来表示算法执行过程中所发生的比较：n个元素作为n个结点，若某两个元素作过比较，则在对应的两点之间连一线。如此得到的图必是连通图，而且删掉 max 点后，图仍是连通的。设与 max 相邻的点有t个，即 max 参加过t次比较，则算法至少执行了$n-2+t$次比较。下面来说明在最坏情况下总有

$$t \geq \lceil \log n \rceil.$$

我们仍考虑n个元素各不相同，魔鬼对手给每个元素x一个权$w(x)$，起始所有元素的权都为1，魔鬼要求算法将依据权来决定发生比较的二元x与y的大小：

$$w(x) \geq w(y) \Rightarrow x > y$$

相应地，

$$w(x) \leftarrow w(x) + w(y),\ w(y) \leftarrow 0$$

任何时候元素的权和都是n，算法执行完毕时，max 元的权必为n，

设 max 参加k次比较后的权为a_k，则

$$n = a_t \le 2a_{t-1} \le 2^2 a_{t-2} \le \cdots \le 2^t a_0 = 2^t$$

因此，

$$t \ge \log n$$

即

$$t \ge \lceil \log n \rceil$$

5.6 找中间元问题的一个下界估计

假设E是一个含n个元素的集合，n是奇数. 我们将针对任何以比较为基础找中间元(即第$(n+1)/2$小元素)的算法必须进行的比较数建立一个下界估计. 因为是考虑最坏情况的下界问题, 可不失一般性地假定这些元素两两不同.

我们断言, 要确定中间元median, 算法必须得到其它元与中间元的相对大小关系, 即, 对任意其他元x, 算法必须确知$x > \mathrm{median}$ 或 $x < \mathrm{median}$。具体说来, 算法必须确定：有$(n-1)/2$个元小于median, 有$(n-1)/2$个元素大于median. 算法是通过建立如图 2.5 所示的树状关系来确定中间元的。图中，每个结点表示一个元素，每条边表示一次比较的结果，边的上端点表示获胜的元素，下端点表示认输的元素。图 2.5 确定了 9 个元素的中间元。而图 2.6 所表示的

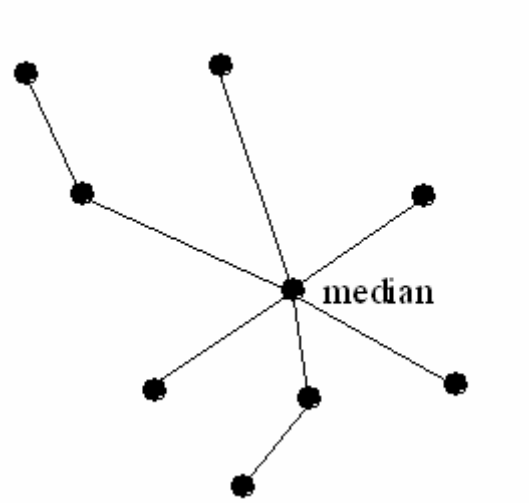


图 2.5

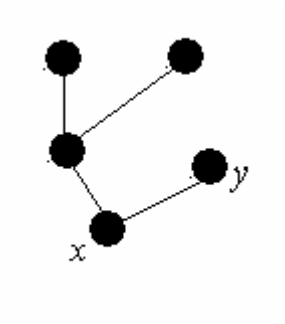


图 2.6

树状结构则不能确定中间元。图中，除结点 x 之外，其他结点均可能是中间元。

图 2.5 中的每个比较都是所谓的定性比较(crucial comparison).

定义 2.3 在找中间元的算法执行过程中，一个涉及 x（x 最终未被算法确定为 median）的比较称为关于 x 的定性比较，如果该比较是第一次如下情形的比较：$x \geq y$ 且（算法执行的过程最终指明）$y \geq \mathrm{median}$（此种情形下，x 被定性为 $x \geq \mathrm{median}$）；或者，$x \leq y$ 且（算法执行的过程最终指明）$y \leq \mathrm{median}$（此种情形下，x 被定性为 $x \leq \mathrm{median}$）。

注：定义中，当 x 与 y 发生比较的时候，并不要求 y 与 median 的大小关系已经确定。

下面的讨论限于作为输入的 n 个元素两两不同。

任何找中间元的算法，如果其执行过程中，没有发生关于某个非中间元的定性比较，则算法不可能正确。细致说来，这是因为，假如算法在某次执行过程中，有一个元素 x 没有执行过关于自身的定性比

较，换句话说，与x比较过的元素，要么没有定性；要么定性为大于median，同时在与x比较中获胜，或定性为小于median，同时在与x比较中认输。如此以来，则如下改变该次执行所对应的输入：(a). 若输入中，$x > \text{median}$，则可改为$x < \text{median}$，且能保持x与其他比较过的元素的相对大小；(b) 若输入中，$x < \text{median}$，则可改为$x > \text{median}$，且同样能保持x与其他比较过的元素的相对大小。由于算法的输出结果完全取决于所执行的比较，对输入作上述改变后，算法执行过程所发生的比较及其结果并无不同，所以算法给出的输出理应是一样的，显然必有一种情况下是错误的。所以为保证正确性，算法必然要执行$n-1$次定性比较。

图 2.6 中，x与y的比较是”无用的(useless)”比较，称之为非定性（noncrucial）比较。特别注意，在一般情形下，大于中间元的元素与小于中间元的元素之间的比较总是非定性比较。

我们下面将要说明，任何找中间元的算法除了必须执行$n-1$次定性比较外，可能执行的非定性比较数在最坏情况下至少为$(n-1)/2$。

定理 2.9 任何以比较为基础的找n（n为奇数）个元素中间元的算法在最坏情况下至少需要$3(n-1)/2$次比较。

证明：只需要说明任何算法执行的非定性比较数在最坏情况下至少为$(n-1)/2$。

我们确定魔鬼策略的原则是：迫使算法执行尽可能多的非定性比较。这儿给出一个较为简单的方案。此方案是通过控制这n个元素的

取值而实现的。具体做法是：在算法执行过程中，给这n个元素分别赋值。此处，给某元素x赋值的意思，不是给它一个具体的值，而是赋予x三种状态之一：大于中间元(记此状态为L)；小于中间元(记此状态为L)，是中间元(记此状态为E)。x未被赋值时的状态记为N。任何元素x第一次参加比较时被赋值。 魔鬼赋值的规则如表 2.2 所示。

比较类型	赋值结果
$N:N$	一个被赋值L，另一个被赋值S
$N:L$	被赋值S
$N:S$	被赋值L

表 2.2

要注意，赋值为L的元素数恰为$(n-1)/2$，赋值为S的元素数也恰为$(n-1)/2$。执行魔鬼赋值规则不能违反此限制。所以，当已经有$(n-1)/2$个L元或$(n-1)/2$个S元的情况下，其余赋值不必继续遵循魔鬼规则。而最后一个N元素被赋值为中间元。

表 2.2 涉及的比较都属于非定性比较。任何算法在此魔鬼策略下需要执行多少次这样的非定性比较呢？在魔鬼赋值规则失效之前，任何一次赋值比较至多产生一个S元，至多产生一个L元。因此当产生了$(n-1)/2$个L元或$(n-1)/2$个S元的时候，算法被迫执行了至少$(n-1)/2$非定性比较。

注 1：证明中描述的魔鬼策略不能保证会发生多于$(n-1)/2$次非定性比较，这是因为算法可以开始先进行$(n-1)/2$次的$N:N$型

比较。

注 2: 关于找中间元问题的下界, 目前得到的最好结果是比$2n$稍大一些, 与我们所知最好算法的比较数还有一点小差距。

5.7 防御魔鬼策略的设计方法

有一类操作譬如元素比较等，用来探求输入元素所含的信息（譬如顺序信息，相对大小信息等）。凭借这类基本操作所获得的信息来决定输出的算法（譬如排序算法， 检索算法，选择算法等），总是希望算法执行过程中发生的每次操作都得到尽可能多的信息。可惜并不能尽如人意，因为总有魔鬼策略在作对，而使操作往往呈现你不希望发生的结果。为了提高算法的质量，防御魔鬼策略的设计方法是一种很有效的技术。

防御魔鬼策略的设计方法，简而言之， 就是要使魔鬼策略无可乘之机。具体来说，要使算法涉及的每个基本操作，不管结果如何，都能得到几乎一样多的信息。

从前面关于一些典型算法的讨论中可以发现，一个好算法需要引入某些平衡的想法。譬如用决策树来表示检索算法或排序算法执行情况时，最坏情况下的比较次数是决策树的高度。在问题规模确定的情况下，为使决策树高度最小，就要尽量使树均衡化。决策树的每个内点表示一次比较，每个叶子表示算法的一种输出（有时，每个内点也表示一种输出，譬如在检索算法的决策树中，就是这样）。内点的左右子树分别表示比较的两种可能结果分别导向的算法输出数目。均衡

树要求这两个数目基本相同， 即相同或相差为 1。二分检索算法和归并排序算法对应的决策树正是均衡树。

在找最大最小元问题和找第二大元问题的算法设计中，使用的锦标赛方法，也是典型的防御魔鬼策略的设计方法。

作业：

利用魔鬼策略设计算法：

1. 请设计算法求5个元素的中间元，要求在最坏情况下只用6次比较.
2. 请设计算法给5个元素排序，要求在最坏情况下只用7次比较.

   （注： 注意到$\lceil \log(5!) \rceil = 7$，少于$7$次比较不能保证在任何情况下给$5$个元素排序）

5.8 一般选择问题算法

一般选择问题：找n个元素$E(1:n)$中的第k小元素。

对此问题， 下面介绍分治法给出的一种最坏时间复杂度为$O(n)$的算法。基本步骤：当$n \le 5$时，通过排序直接求解；当$n > 5$时，

1. 将$E(1:n)$中的元素分成$n/5$组，每组5个元素，然后分别找每组的中间元，然后再找这些中间元中的中间元m^*。如图 2.7 所示，以上操作将$E(1:n)$中元素分成四个区域。其中，C中元素比m^*小，B中元素比m^*大。

2. 接着A，D中的元素分别与m^*比较。到此，$E(1:n)$就分成比m^*小的元素集合S和比m^*大的元素集合L。

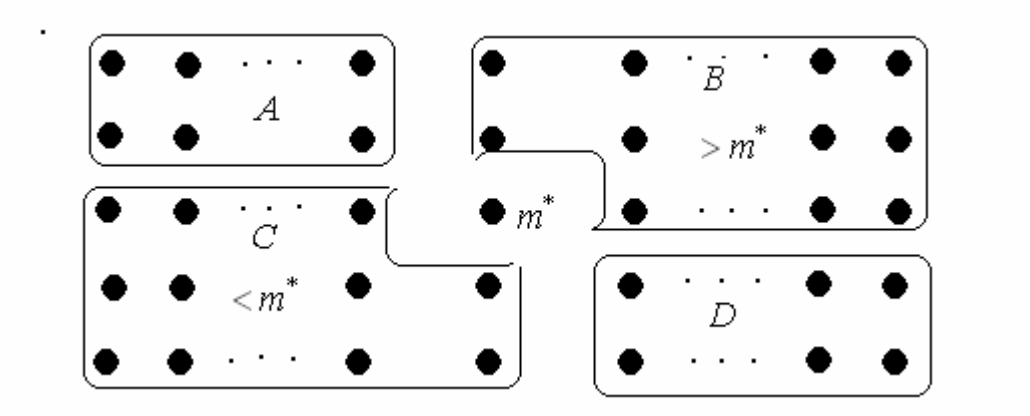


图 2.7

3. 若$k \le |S|$，则问题归结为找S中的第k小元素；若$k = |S|+1$，则m^*正为所求；若$k > |S|+1$，则问题归结为找L中的第$k-|S|-1$小元素。

以上算法的最坏时间复杂度$W(n)$满足下面递归式：

$$W(n) \le cn + W(n/5) + W(7n/10) \quad (c \ge 1) \qquad (2.47)$$

（更准确地，$W(n) \le cn + W(n/5) + \max\limits_{3n/10 \le m \le 7n/10} W(m)$）

递归式(2.47)中，cn包含了在所有 5 元小组找中间元和A，D中的元素分别与m^*比较所需要的比较数；$W(n/5)$是指在小组中间元中找中间元m^*所需的最大比较数；$W(7n/10)$表示找S中的第k小元素或找L中的第$k-|S|-1$小元素所花费的最大比较数。

定理 2.10 $W(n) \le 20cn$。

证明：采用数学归纳法可证得。当$n \le 5$时，通过排序直接求解，显然有$W(n) \le 20cn$；当$n > 5$时，由递归式（2.47）和归纳假设得

$W(n) \le cn + W(n/5) + W(7n/10) \le cn + 20c(n/5) + 20c(7n/10)$
$= cn + 4cn + 14cn = 19cn < 20cn$。

6. 平面上的最近点对和凸壳问题

本节考虑涉及平面上有限点集的两个著名问题. 这两个问题来自两个应用领域: 计算几何(computational geometry)和运筹学(operations research)。

6.1 最近对问题

最近对问题是要找平面上n个点中的两个相距最近的点. 当然, 这些点可以是更高维空间的点。我们以笛卡儿坐标(x, y)的形式表示点P, 点$P_i = (x_i, y_i)$与点$P_j = (x_j, y_j)$之间的距离是指他们的欧氏距离

$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

容易想到的直接求解方法是，分别算出这n个点两两之间的距离，来找出最小距离的两点。

算法 2.10 直接求解

Procedure BruteForceClosestPoints$(P, n, index1, index2)$

//输入: 数组$A(1:n)$ $n(n \ge 2)$个点$P_1 = (x_1, y_1)\cdots$, //

//$P_n = (x_n, y_n)$; 输出:最近两点的下标 $index1$, $index2$ //

$d\min \leftarrow \infty$

for $i \leftarrow 1$ to $n-1$ do

for $j \leftarrow i+1$ to n do

$d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$

If $d < d\min$

$d\min \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$

return($index1, index2$)

End BruteForceClosestPoints

直接求解算法的时间复杂度是数据无关的，为$\Theta(n^2)$。效率太低，下面利用分治法可得到最坏时间复杂度为$\Theta(n\log n)$的算法。

基本想法：将这n个点分成数目相同的两组，将原问题转化为两个子问题，在子问题求解的基础上，得到原问题的解。具体步骤如下：

1．将这n个点按y坐标排成升序，置放在数组$Y(1:n)$中；

2．将数组$Y(1:n)$按x坐标以稳定排序算法排成升序，置放在数组$X(1:n)$中；

（前面 2 个步骤作为下面递归算法的预处理。）

3．当$n \le 3$时，直接求解。否则，将数组$X(1:n)$分割成$X(1:\lfloor n/2 \rfloor)$和$X(\lfloor n/2 \rfloor + 1, n)$，记点$X(\lfloor n/2 \rfloor)$的横坐标为$x_m$，则数组$X(1:\lfloor n/2 \rfloor)$和$X(\lfloor n/2 \rfloor + 1, n)$的点分别处于垂直线$x = x_m$的左右两侧，如图 2.8 所示。相应地，将数组$Y(1:n)$分割成与这两个子数组包含相同元素的两个数组$Y(1:\lfloor n/2 \rfloor)$和$Y(\lfloor n/2 \rfloor + 1, n)$.

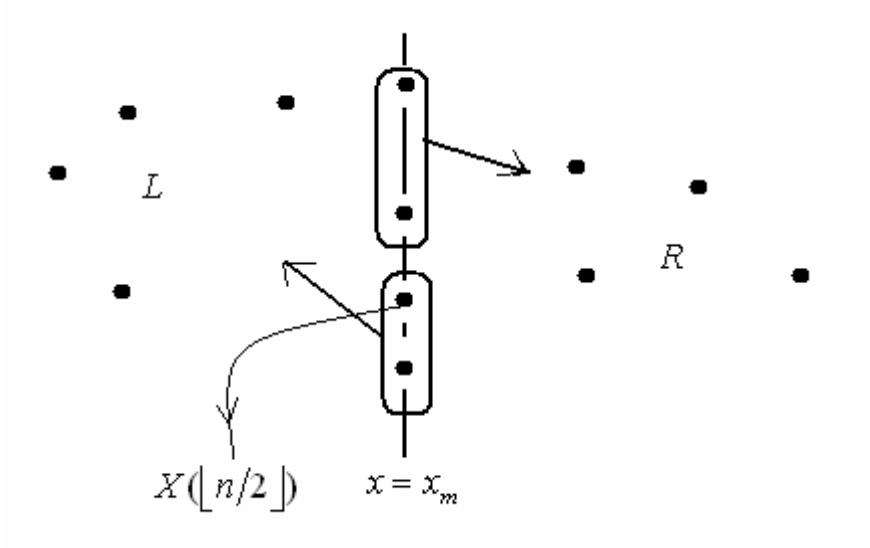


图 2.8

4. 以$X(1:\lfloor n/2 \rfloor)$和$Y(1:\lfloor n/2 \rfloor)$作为输入，求解这$\lfloor n/2 \rfloor$个点中距离最近的两点，并求得其距离d_1；以$X(\lfloor n/2 \rfloor+1, n)$和$Y(\lfloor n/2 \rfloor+1, n)$作为输入，求解这$\lceil n/2 \rceil$个点中距离最近的两点，并求得其距离$d_2$；

5. 记$d = \max(d_1, d_2)$。为求原问题的解，只剩下来考察区域L中点与区域R中点之间的最小距离$\bar{d}$。而只有当$\bar{d} < d$时，这个考察才有意义。因此只需要考察区域L_d(如图 2.9 所示)中点与区域R_d中点之间的最小距离。从数组$Y(1:n)$中删去区域L_d和R_d以外的点得到数组$Y(1:n_1)$（$n_1 \le n$）。时间复杂度$O(n)$。

6. 假设这个最小距离$\bar{d}$发生在如图 2.10 所示的P_l和P_r两点之间，即，$\bar{d} = d(P_l, P_r) < d$，且不妨假设$P_l$的纵坐标$y_l$不大于$P_r$的纵坐标。则$P_r$只可能处于两条水平线$y = y_l$和$y = y_l + d$之间，即$P_r$必在区域$R_{d,d}$中。不难看出区域$R_{d,d}$至多有$4$个点，同样区域$L_{d,d}$也至多有$4$个点。这说明要求得最小距离$\bar{d}$，只需对数组$Y(1:n_1)$中的

每个点P，求P与（在数组中排在）它后面的 7 个点（实际上 5 个点足够了）之间的最小距离。时间复杂度$O(n)$。

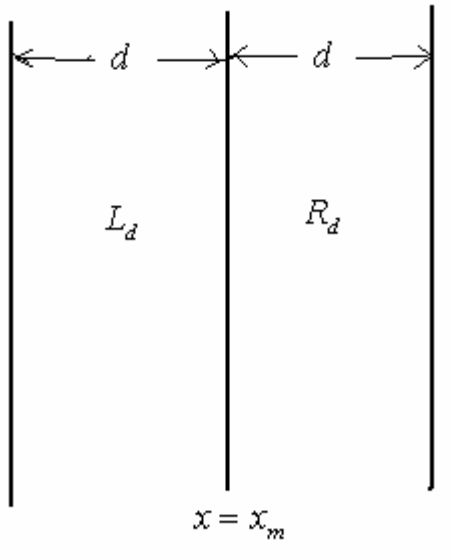


图 2.9

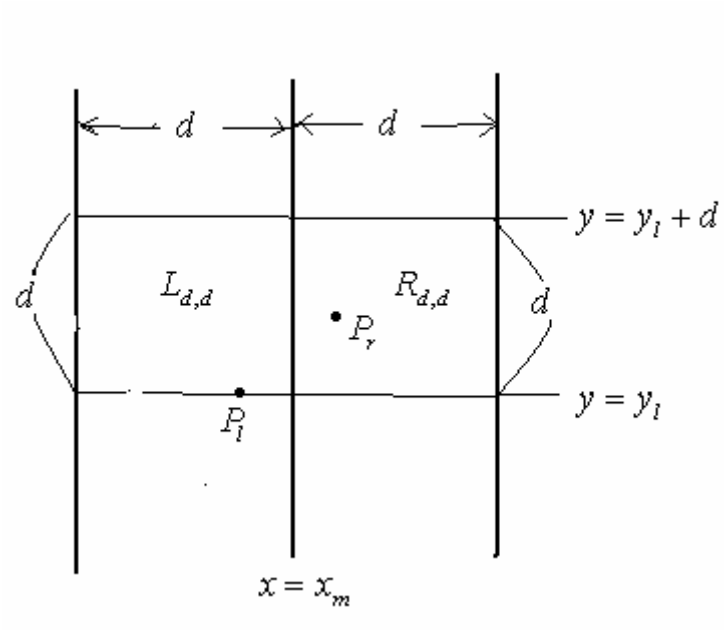


图 2.10

设该算法（不考虑预处理）的最坏时间复杂度为$W(n)$，则

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + \Theta(n)$$

依据 **master** 定理得知：$W(n)=\Theta(n\log n)$. 由于预处理是两个排序过程，所以总时间复杂度依然是$\Theta(n\log n)$。可证明任何最近点对算法在最坏情况下的复杂度为$\Omega(n\log n)$。

6.2 凸壳(convex hull)问题

先介绍一些基本概念和结论。

定义 2.4：平面上的点集S称为凸集，如果以S上的任何两点P和Q为端点的线段整个属于S。

图 2.11 是一些凸集，图 2.12 是一些非凸集。

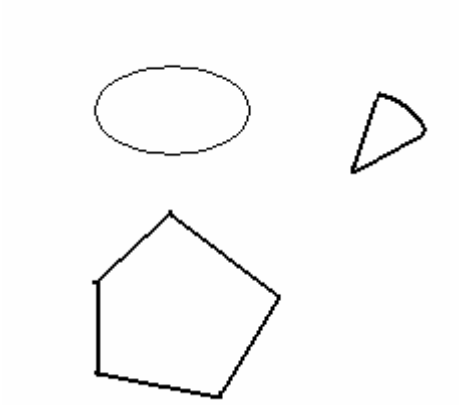


图 2.11

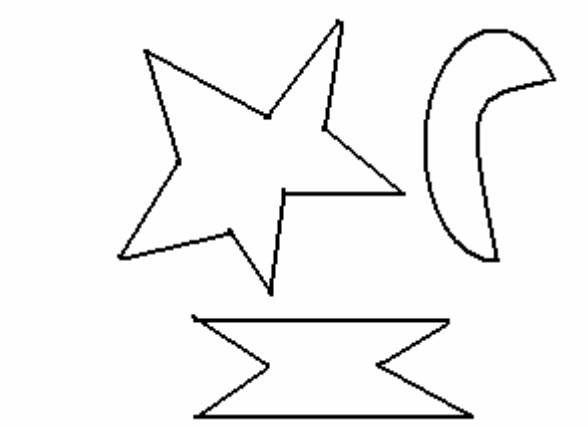


图 2.12

定义 2.5：平面点集S的凸壳（记作$CH(S)$）是指包含S的最小凸集。

注：这儿“最小”的意思是：$CH(S)$是任何包含S的凸集的子集。

凸集的凸壳当然是其本身，图 2.12 中非凸集的凸壳如图 2.13 所示。

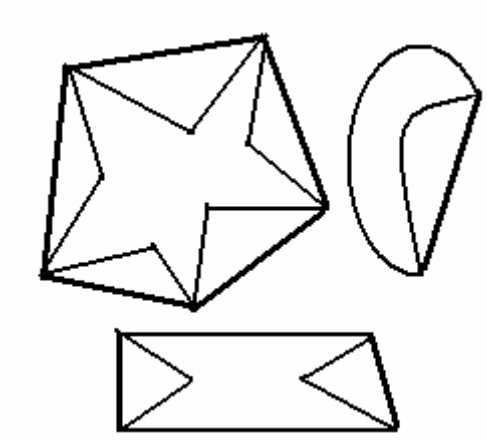


图 2.13

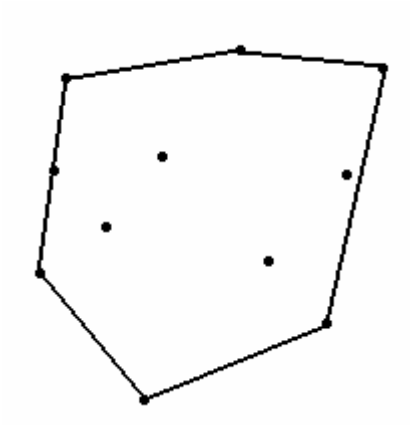


图 2.14

定理 2.11 设S是平面上不全在一条直线上的$n(n\geq 3)$个点组成的集合，则$CH(S)$是一个凸多边形，且凸多边形的顶点全是S中的点。如图 2.14 所示。

注：当S中的n个点全在一条直线上时，$CH(S)$是一条线段，且其两个端点是S中的点。

问题：给定平面上的n个点，求凸壳。

因为该问题的重要性，所以有多种算法，这儿介绍一种简单的分治算法。描述如下：

1. 分别找出横坐标最小和最大的点P_1，P_2，则P_1，P_2必是所求凸壳的顶点；
2. 将其他的点分割为直线P_1P_2上方的点和下方的点。分别在上方和下方的点中找与直线P_1P_2距离最远的点（如果存在的话）P_3，P_4（也必是所求凸壳的顶点）。
3. 继续分别找直线P_1P_3，P_3P_2上方的点，直至上方不再有点；
4. 继续分别找直线P_1P_4，P_4P_2下方的点，直至下方不再有点。

算法执行如图 2.15 所示。

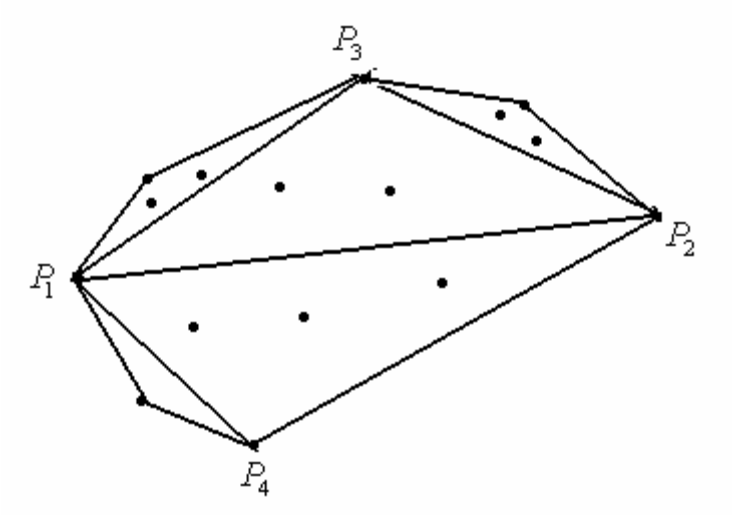


图 2.15

该算法最坏时间复杂度为$\Theta(n^2)$，平均时间复杂度为$\Theta(n\log n)$。

其他一些算法最坏时间复杂度可达到$\Theta(n\log n)$，但较为复杂。可证明任何凸壳算法在最坏情况下的复杂度为$\Omega(n\log n)$。

作业：

给定平面上的n条直线$L_i: y = a_i x + b_i (1 \le i \le n,\quad a_i \ne 0)$, 假定其中任三条直线不会相交于同一点。称某直线$L_j$在给定横坐标$x_0$处是最高的，若$L_j$上对应点的纵坐标大于其他直线上对应点的纵坐标，即$a_j x_0 + b_j > a_i x_0 + b_i$对$i \ne j$成立。称直线$L_j$是可见的，若$L_j$在某横坐标点处是最高的。请设计时间复杂度为$O(n\log n)$的算法，来给出这$n$条直线中所有的可见直线。

第三章 贪心法(Greedy)

1. 概述

贪心法是用来解决最优化问题的最为直接的设计技术。适用的最优化问题包括三个要素：由n个元素组成的输入集，约束条件，目标函数。满足约束条件的输入子集称为该问题的可行解(不唯一)；能够使目标函数取极值(最大或最小)的可行解称为最优解。目的要找到最优解。

贪心法是一种分级处理法：首先根据问题的特性，选取一种量度标准，然后将n个输入排成这种量度标准所要求的顺序，按这种顺序每次考查一个输入元，以决定该输入元是否应在最优解中。如果该输入元和当前已构成在这种量度意义下的部分最优解合在一起最终不能产生一个可行解，则不把此输入元加进这部分解之中。其中，量度标准的选取尤为关键，因为不适当的量度标准只能得到次优解(局部最优解)，而不是最优解。

我们先来回忆一个大家熟悉的例子。

例 3.1 求赋权连通图$G(V, E)$的最小生成树T.

解决该问题有很多算法，其中之一就是著名的 Kruskal 算法，该算法的设计正是采用了贪心策略。

输入：图G的边集E;

约束条件：作为解的边子集不可以构成回路;

目标函数：解集的边权和;

最优化：使目标函数最小。

量度标准：在任何部分解的情况下，加入的边使边权和的增加达到最小。按此量度标准，将边集依边权非递减的顺序排列。起初，部分解是空集，将最小边加入，当依顺序考虑加入一条边时，如果该边加入进去，会构成回路，则此边不放入部分解中；否则，加入得到新的部分解。继续考虑后面的边，直到部分解已包含了$n-1$条边为止。这样，最终得到一个最优解。

大致说来，贪心法设计的算法可抽象为如下过程：

procedure $GREEDY(A,\ n)$

$A(1:\ N)$ contains the n inputs

$solution \leftarrow \varnothing$ //initialize the solution to empty //

for $i \leftarrow 1$ to n do

$x \leftarrow SELECT(A)$

if $FEASIBLE(solution,\ x)$

then $solution \leftarrow UNION(solution,\ x)$

endif

repeat

return($solution$)

end $GREEDY$

函数$SELECT$的功能是按某种最优量度标准从A中选择一个输入，把它的值赋给x并从A中删掉它。$FEASIBLE$是一个布尔函数，它判定x是否可以包含在解向量中。$UNION$将x与解向量结合

成新的解向量并修改目标函数。过程$GREEDY$描述了用贪心策略设计算法的主要工作和基本控制路线。一旦给出一个特定的问题，就可将$SELECT$，$FEASIBLE$和$UNION$具体化并付诸实现。

另外一类可采用贪心策略的优化问题是排列问题，步骤是类似的，对于n个输入元素，根据适当的量度标准，从部分排列，逐个添加元素于其后，最终得到这n个输入元素的最优排列。

我们下面例举一些典型问题。

2. 磁带上的最优存储

将n个长度分别为l_i（$(1\le i\le n)$）的程序i按顺序：$I=i_1,\ \cdots,\ i_n$（I是$1,\ \cdots,\ n$的一个排列）存放在某一计算机磁带上，则检索程序i_r所花费的时间为

$$\sum_{1\le k\le r} l_{i_k} \qquad (3.1)$$

这是因为检索任一程序时，磁头总置于磁带起始端点。假如这n个程序经常平均地被检索，则平均检索时间(简记为 MRT: mean retrieval time)为

$$\frac{1}{n}\sum_{1\le r\le n}\sum_{1\le k\le r} l_{i_k} \qquad (3.2)$$

磁带最优存储问题要求对给定的n个程序确定一种排序I，使得对应的 MRT 最小，等价于使总检索时间：

$$D(I)=\sum_{1\le r\le n}\sum_{1\le k\le r} l_{i_k} \qquad (3.3)$$

最小。

借助贪心法可作如下设计：量度标准：使部分解的D值增加最小，部分解即为前r个排列：$i_1, \cdots, i_r$，当将第$r+1$个程序j加入时，D值的增加量为

$$\sum_{1\le k\le r} l_{i_k} + l_j \qquad (3.4)$$

因为$\sum_{1\le k\le r} l_{i_k}$已确定，要使式(3.4)最小，只须使$l_j$最小。由此分析可知，按程序长度的非降次序来存放程序，就可得到该量度标准下的最优解。这种排列次序只要使用一个有效的分类算法就可在$\Theta(n\log n)$时间内完成.下面定理 3.1 说明这种排列正是n个程序在磁带上的最优存储方式。

定理 3.1　在所有可能的排列$I = i_1, \cdots, i_n$中，满足

$$l_{i_1} \le l_{i_2} \le \cdots \le l_{i_n} \qquad (3.5)$$

的排列使

$$D(I) = \sum_{1\le r\le n}\sum_{1\le k\le r} l_{i_k}$$

取最小。

证明：设$I = i_1, \cdots, i_n$是下标集合$\{1, \cdots, n\}$的任一排列，则

$$D(I) = \sum_{1\le r\le n}\sum_{1\le k\le r} l_{i_k} = \sum_{1\le k\le n}\sum_{k\le r\le n} l_{i_k} = \sum_{1\le k\le n}(n-k+1)l_{i_k} \qquad (3.6)$$

$n-k+1$即为磁带上排列在程序i_k后面的程序数目（包括程序i_k在内）。 如果排列I不满足式(3.5)，即存在h，$1\le h<n$，使得$l_{i_h} > l_{i_{(h+1)}}$，则交换i_h和$i_{(h+1)}$得到排列I',有

$$D(I') = \sum_{k \neq h, h+1} (n-k+1) l_{i_k} + (n-h+1) l_{i_{(h+1)}} + (n-h) l_{i_h} \qquad (3.7)$$

以$D(I)$减$D(I')$得

$$D(I) - D(I') = l_{i_h} - l_{i_{(h+1)}} > 0 \qquad (3.8)$$

这说明，那些不按l_i非降次序存放的排列不可能取得最小的D值。而由式(3.6)容易看出，按l_i非降次序存放的所有排列都有相同的D值，故满足式(3.5)的排列次序使D取最小值。

可以将一盘带的存储问题扩展到多盘带的存储问题。如果有m $(m>1)$盘带：T_0，…，T_{m-1}，则要把n个程序分布到这m盘带上，每盘带将单独构成一种存储排列。记I_j $(0 \le j \le m-1)$是存储在磁带j上的程序的排列方式，$D(I_j)$定义如前，再设TD是总检索时间，则

$$TD = \sum_{0 \le j \le m-1} D(I_j) \qquad (3.9)$$

一盘带情形下的量度标准同样适用于多盘带：即当前正在考虑的程序大小及其把它存储在哪个磁带上应使TD产生的增量最小。满足这样要求的做法自然是：

1.选择迄今用带量最少的磁带来存放下一个程序；

2.将要存放的程序是尚未存储的程序中长度最短的。

具体操作如下：首先把这n个程序按其长度的非降次序(即$l_1 \le l_2 \le \cdots \le l_n$)来排列，然后把它们依次轮流分配给带$T_0$，…，$T_{m-1}$，即程序$i$存储在$T_j$ ($j = i-1 \pmod m$)磁带上，顺序数是：$\lceil i/m \rceil$.

算法 3.2 以一个过程的形式描述了这一规则。在调用这过程之前，假定要存放的程序已按长度的非降次序排序。这个算法的时间复杂度为$\Theta(n)$. 定理 3.2 证明了算法 3.2 产生最优的存储模式。

算法 3.2 把程序分配给磁带(Assigning programs to tapes)

procedure $STORE(n, m)$

//n is the number of programs and m is the number of tapes.//

integer n, m, j

$j \leftarrow 0$ //next tape to store on //

for $j \leftarrow 1$ to n do

print('append program', i, 'to permutation for tape', j)

$j \leftarrow j+1 \pmod m$

repeat

end $STORE$

定理 3.2 若$l_1 \leq l_2 \leq \cdots \leq l_n$, 则算法 3.2 生成一个对$m$盘磁带最优的存储模式。

证明：在对m盘磁带的任何存储模式中，设r_i是程序i所在带上位于程序i后面的程序个数(包括程序i), 则总检索时间TD由下式给出：

$$TD = \sum_{1 \leq i \leq n} r_i l_i \qquad (3.10)$$

首先，由定理 3.1 的证明过程可知，只有当$r_1 \geq r_2 \geq \cdots \geq r_n$的情况下，$TD$才有可能取最小值。其次，$r_1, \cdots, r_n$的最优取值为：

$$r_n = r_{n-1} = \cdots = r_{n-m+1} = 1,\ r_{n-m} = \cdots = r_{n-2m+1} = 2,\ \cdots\cdots$$

即让r_i尽量小，但取同一值的r_i数目不超过m，算法 3.2 得到的

$r_i = \lceil (n-i+1)/m \rceil$即是这样的。

由上面证明可以看出，使TD取最小值的存储模式不是唯一的，而且有很多个。简单说来，如果将具有相同r_i的程序互易其位，所得新存储模式的TD值与原模式相同。若n是m的倍数，则至少有$(m!)^{n/m}$种使TD取最小值的存储模式，一般来说，总共有

$$(m!)^{\lfloor n/m \rfloor} \cdot P_m^k,$$

种取最小值的存储模式，其中$k = n \bmod m$。

作业

1. 假定要将长为$l_1, l_2, \cdots, l_n$的n个程序放入一盘磁带，程序i被检索的频率是f_i。如果程序按排列$I = i_1, i_2, \cdots, i_n$的次序存放，则期望检索时间是

$$ERT(I) = [\sum_j (f_{i_j} \sum_{k=1}^{j} l_{i_k})] \Big/ \sum f_i .$$

证明：

(1). 按l_i的非降次序存放程序不一定得到最小的ERT；

(2). 按l_i的非增次序存放程序不一定得到最小的ERT；

(3). 按f_i/l_i的非增次序存放程序时得到最小的ERT。

2. 设某种货币有如下几种面值的钞票：10 元，5 元，2 元，1 元，出纳员要支付$n\,(n \geq 1)$元现金。请你给出一种策略，使支付的钞票个数最少，并加以证明。

3. 背包问题(KNAPSACK PROBLEM)}

问题：给定一个容量(重量)为M的背包和其重量分别为w_i的n种物品i $(1 \le i \le n)$, 假定将物品i的一部分x_i $(0 \le x_i \le 1)$放入背包就会得到大小为$p_i x_i$ $(p_i > 0)$的效益，试问采用怎样的装包方法才会使装入背包的物品总效益最大？可将这个问题形式描述如下：

极大化：$$\sum_{1 \le i \le n} p_i x_i \qquad (3.11)$$

约束条件：$$\sum_{1 \le i \le n} w_i x_i \le M，\ 0 \le x_i \le 1 \qquad (3.12)$$

其中式(3.11)是目标函数，满足约束条件的任一向量$(x_1, \cdots, x_n)$是一可行解，使目标函数取最大值的可行解是最优解。我们正是要求这样的最优解。

分析：如果这n件物品的总重量不超过M，把所有的物品装入背包(即$x_i = 1,\ 1 \le i \le n$)自然获得最大效益，否则，要使效益最大，必须选取x_i使得

$$\sum_{1 \le i \le n} w_i x_i = M$$

以贪心法来求解。量度标准：既然受限的背包的容量，自然优先考虑单位重量效益较大的物品，即将物品按比值p_i / w_i的非增次序排列，依此顺序考虑将物品逐个放入背包。首先试图将物品整个放入背包，若成功，则继续考虑放入后面的物品，否则只能放入该物品的一部分背包容量饱和，即得最大效益。

如果将物品事先按 p_i/w_i 的非增次序排列，则过程 $GREEDY\_KNAPSACK$ 就得出这一策略下背包问题的解。如果将物品排序的时间不算在内，此算法所用时间为 $\Theta(n)$.

算法 3.3 背包问题的贪心算法(Algorithm for greedy strategies for knapsack problem)

procedure $GREEDY\_KNAPSACK(p, w, M, X, n)$

// $p(1:n)$ and $w(1:n)$ contain the profits and weights// //respectively of n objects ordered so that// // $\frac{p(i)}{w(i)} \geq \frac{p(i+1)}{w(i+1)} (1 \leq i \leq n-1)$, M is the knapsack size and// // $X(1:n)$ is the solution vector.//

real $p(1:n)$, $w(1:n)$, $X(1:n)$, M, c;

$X \leftarrow 0$ // initialize solution to zero //

$c \leftarrow M$ //c: remaining knapsack capacity //

for $i \leftarrow 1$ to n do

if $w(i) > c$ then exit endif

//or if $w(i) > c$ then { $X(i) \leftarrow \frac{c}{w(i)}$; return } endif //

$X(i) \leftarrow 1$

$c \leftarrow c - w(i)$

repeat

if $i \le n$ then $X(i) \leftarrow \dfrac{c}{w(i)}$ endif

end $GREEDY-KNAPSACK$

下面来证明该算法确实得到了最优解。

定 理 3.3 如果 $\dfrac{p_1}{w_1} \le \dfrac{p_2}{w_2} \le \cdots \le \dfrac{p_n}{w_n}$，则算法 $GREEDY\_KNAPSACK$ 对于任何给定的背包问题实例生成一个最优解。

证明：设 $X=(x_1, x_2, \cdots, x_n)$ 是 $GREEDY\_KNAPSACK$ 所生成的解，如果所有的 x_i 等于 1, 显然这个解就是最优解。于是设 j 是使 $x_j \ne 1$ 的最小下标，由算法的执行过程可知，当 $1 \le i < j$ 时，$x_i = 1$; 当 $j < i \le n$ 时，$x_i = 0$; 而 $0 < x_j < 1$. 设 $(y_1, y_2, \cdots, y_n)$ 是任一可行解，即有：

$$\sum_{1 \le i \le n} w_i y_i \le M, \quad (0 \le y_k \le 1,\ 1 \le k \le n)$$

因为

$$\sum_{1 \le i \le n} w_i x_i = M,$$

所以，

$$\sum_{1 \le i \le n} w_i y_i \le \sum_{1 \le i \le n} w_i x_i,$$

即

$$\sum_{1\le i<j}(x_i-y_i)w_i+(x_j-y_j)w_j-\sum_{j<i\le n}y_iw_i\ge 0。$$

终于

$$\sum_{1\le i\le n}x_ip_i-\sum_{1\le i\le n}y_ip_i=\sum_{1\le i<j}(x_i-y_i)w_i\frac{p_i}{w_i}+(x_j-y_j)w_j\frac{p_j}{w_j}-\sum_{j<i\le n}y_iw_i\frac{p_i}{w_i}$$

$$\ge\sum_{1\le i<j}(x_i-y_i)w_i\frac{p_j}{w_j}+(x_j-y_j)w_j\frac{p_j}{w_j}-\sum_{j<i\le n}y_iw_i\frac{p_j}{w_j}$$

$$=\frac{p_j}{w_j}\{\sum_{1\le i<j}(x_i-y_i)w_i+(x_j-y_j)w_j-\sum_{j<i\le n}y_iw_i\}\ge 0,$$

即

$$\sum_{1\le i\le n}x_ip_i\ge\sum_{1\le i\le n}y_ip_i。$$

这说明$X=(x_1,\ x_2,\ \cdots,\ x_n)$是最优解，证毕。

4. 带有限期的作业排序(JOB SEQUENCING WITH DEADLINES)

问题：假定在一台机器上处理n个作业，每个作业均需一个单位时间才能完成，且每个作业i都有一个截止期限d_i $(1\le d_i\le n)$(整数)，任何作业i必须在它的截止期限d_i以前完成时，才可获得一定的效益p_i，问：如何在机器上安排这n个作业，以使获得的效益最大。这个最优化问题的可行解是这n个作业的一个子集合J，J中的每个作业都能被安排在各自的截止期限以前完成。可行解的效益值是J中这些作业的效益之和，即$\sum_{i\in J}p_i$．具有最大效益值的可行解就是最

优解。

例 3.2 $n=4,\ (p_1,\ p_2,\ p_3,\ p_4)=(100,\ 10,\ 15,\ 20),$

$(d_1,\ d_2,\ d_3,\ d_4)=(2,\ 1,\ 2,\ 1)$. 该实例的可行解罗列如表 3.1：

表 3.1

可行解	处理顺序	效益值
{1}	1	100
{2}	2	10
{3}	3	15
{4}	4	20
{1,2}	2,1	110
{1,3}	1,3(或 3,1)	115
{1,4}	4,1	120
{2,3}	2,3	25
{3,4}	4,3	35

其中解$\{1,4\}$是最优解。

为借助贪心法拟制求最优解的算法，将目标函数$\sum_{i\in J} p_i$作为量度，优先考虑效益p_i大的作业，即按p_i的的非增次序来考虑这n个作业。首先将效益最大的第一个作业放入解集J中，再依次考虑后面每一个作业，当考虑作业i时，若$J\cup\{i\}$仍是可行解，则将作业i加

入J，否则i不加入J中。

上面描述的贪心法以算法 3.4 的形式给出其粗略的描述。

算法 3.4 作业排序算法的粗略描述(High level description of job sequencing algorithm)

procedure $GREEDY\_JOB(P, D, J, n)$

//The jobs are ordered such that $p_1 \geq p_2 \geq \cdots \geq p_n$. J is an //output variable, it is the set of jobs to be completed by their //deadlines.//

$J \leftarrow \{1\}$

for $i \leftarrow 2$ to n do

  if {all jobs in $J \cup \{i\}$ can be completed by their deadlines}

  then $J \cup \{i\}$

  endif

repeat

end $GREEDY\_JOB$

定理 3.4 算法 3.4 所描述的贪心算法对于作业排序问题总是得到一个最优解。

证明：设(p_i, d_i)，$1 \leq i \leq n$，是作业排序问题的任一实例，$J = \{j_1, j_2, \cdots, j_m\}$是由贪心方法所选择的作业集。若$J$不是最优解，因可行解数目有限，最优解总是存在的。对任何一个最优解I来说，显然$I \subsetneqq J$，同时由J的生成过程知$J \subsetneqq I$，选取最优解I，使$|I \cap J|$达到最大。设j_k是属于J但不属于I的具有最高效益值的

作业，即 $j_1, j_2, \cdots, j_{k-1} \in I$，但 $j_k \notin I$.对于在 I 中但不在 J 中的任意作业 b，都有 $p_b \le p_{j_k}$，这是因为若 $p_b > p_{j_k}$，则 $\{j_1, \cdots, j_{k-1}, b\}$ 是可行解，这样的话，在贪心算法的执行过程中，作业 b 将先于作业 j_k 加入 J 中，与 b 不在 J 中的假定矛盾。

设 S_I 和 S_J 分别是 I 和 J 的可行调度表，对既属于 I 又属于 J 的每一作业 i，设 i 在 S_I 中在 t 到 $t+1$ 单位时间段被调度，而在 S_J 中在 t' 到 $t'+1$ 单位时间段被调度，不妨设 $t < t'$，则在 S_I 中，将 $[t', t'+1]$ 时刻所调度的作业(如果有的话)与 i 相交换。于是得到调度表 S_I' 和 S_J'，使 $I \cap J$ 中的作业在相同的时间段被调度。设在 S_J' 中作业 j_k 在时间段 $[t_k, t_k+1]$ 被调度，假如在 S_I' 中的这一时间段里作业 b (如果有的话)被调度，则在 S_I' 中去掉作业 b 而替之以作业 j_k，则给出一张关于作业集 $I' = (I - \{b\}) \cup \{j_k\}$ 的可行调度表，由于 $p_b \le p_{j_k}$，所以 I' 的效益不小于 I 的效益，即 I' 也是最优解．但 $|I' \cap J| = |I \cap J| + 1$，这与 I 的选取矛盾。说明假设错误，J 是最优解。

下面定理给出判定一个作业集 J 是不是可行解的依据。

定理 3.5 设作业集 J 包含 m 个作业，这 m 个作业按期限非减顺序排列为：$\sigma = i_1 i_2 \cdots i_m$，即有，$d_{i_1} \le d_{i_2} \le \cdots \le d_{i_m}$。则 J 是可行解，当且仅当若 J 中的作业按照排列 σ 的顺序执行的话，则 J 中的每个作业都可在期限以前完成， 即 $d_{i_j} \ge j$， $1 \le j \le m$。

证明：若 J 是可行解，即存在排列 $\tau = s_1 s_2 \cdots s_m$，使得 J 中的作业

按照排列τ的顺序执行的话，则J中的每个作业都可在期限以前完成， 即$d_{s_j} \geq j$，$1 \leq j \leq m$。如果$\tau \neq \sigma$，设k是使得$s_j \neq i_j$的最小下标。即，当$j < k$时，$s_j = i_j$，但$s_k \neq i_k$。应有$d_{i_k} \leq d_{s_k}$。设$s_h = i_k$，显然，$h > k$。在排列τ中，交换s_k和s_h的位置得到排列$\tau_1 = s_1 \cdots s_{k-1} s_h s_{k+1} \cdots s_{h-1} s_k s_{h+1} \cdots s_m$。因为$d_{s_k} \geq d_{s_h} \geq h$，所以排列$\tau_1$也可使得$J$中的每个作业都可在期限以前完成。对排列$\tau_1$进行对排列$\tau$同样的操作得到排列$\tau_2$，继续进行这种步骤，得到$\tau_3$，$\cdots$，直至某$\tau_t = \sigma$。说明排列$\sigma$可使得$J$中的每个作业都可在期限以前完成。

4. 拟阵与贪心算法

有许多最优化问题具有如下特征：E是输入集，当$X \subseteq E$是该问题的可行解时，任意$\mathrm{Y} \subseteq \mathrm{X}$也是该问题的可行解; 且其目标函数为如下形式：

$$f(X) = \sum_{x \in X} w(x),$$

其中w是定义在输入集E上的取正值的赋权函数。使$f(X)$取最大值的可行解X即是所求最优解。

最小生成树问题和带有限期的作业排序问题都属于这类最优化问题。 那么这类最优化问题是不是都可以采用贪心法求解呢? 本节试图系统地回答这个问题。

为此，我们引进一个新概念-----拟阵。

定义 3.1 设E是非空有限集合，I是E的非空子集族，如果满足下

列条件：

(1).若 $X \in I$ 且 $Y \subseteq X$，则 $Y \in I$；（遗传性）

(2).若 $X, Y \in I$ 且 $|Y| > |X|$，则存在 $y \in Y - X$，使 $X \cup \{y\} \in I$.

(交换性)

则称 (E, I) 为一拟阵(matroid)，记为 $M = M(E, I)$.

注：$\varnothing \in I$.

拟阵这个概念的提出是在 20 世纪 30 年代。它是 1935 年 Whiteney 在"关于线性相关的抽象性质"一文中提出的，将拟阵作为向量线性相关关系的抽象推广。

我们先作如下考察：设 V 是一线性空间。若其子集 $X = \{x_1, \cdots, x_n\}$ 的元素是线性无关的，则对任意 $Y \subseteq X$，Y 中的元素必是线性无关的，并且若 $X = \{x_1, \cdots, x_p\}$ 和 $Y = \{y_1, \cdots, y_q\}$ 均是 V 的线性无关的子集，且 $q > p$，则必存在 $y_i \in Y - X$，使 $X \cup \{y_i\}$ 是线性无关集，由此我们得到如下拟阵的第一个例子。

例 3.5 设 V 是一线性空间，$E \subseteq V$ 是一非空有限集，定义

$I = \{X \subseteq E: X$ 是线性无关集$\}$,

则显然 (E, I) 是一拟阵。

所以说，拟阵是关于线性相关概念的推广。另外，拟阵概念也由其它的研究者从图论的角度提出。

例 3.6 设 $G = (V, E)$ 是一无向图，定义

$I = \{X \subseteq E: X$ 中的边不能形成圈$\}$,

则$M=(E,\ I)$是一拟阵，称为图G的圈拟阵。

注：需要验证其满足交换性。

下面关于拟阵的其它概念得名于线性空间中元素的相关性。

定义 3.2 设$M=(E,\ I)$是一拟阵，任意$X\in I$称为M的独立集，不是独立集的E的子集称为M的相关集。若$B\in I$，但不存在$B'\in I$，使$B\subset B'$，则称B是M的基。由拟阵的交换性可知，所有的基有相同的元素数，该数称为拟阵的秩。

拟阵理论有很丰富的内容，我们在此不可能也没必要作详细介绍。有兴趣的同学可参看有关书籍，譬如，“拟阵”,刘桂真，陈庆华著，国防科技大学出版社，1994.

作了上面的介绍，我们就可以讨论拟阵与贪心法的关系。

定义 3.3 设$M=(E,\ I)$是一拟阵，w是定义在E上的正的赋权函数。对于$X\in I$，定义$f(X)=\sum_{x\in X}w(x)$，则使$f(X)$取最大值的独立集必是M的基，称为赋权拟阵$(M,\ w)$的最优基。

我们可采用如下算法来求最优基。

算法 3.5 赋权拟阵的贪心算法

procedure $GREEDY(E,\ w)$

//Finding an optimal base for weighted matroid$(E,\ I,\ w)$.//

//$E(1:\ n)$ are the elements of E sorted into nonincreasing //

//order by weight w.//

$B\leftarrow\varnothing$

For $i \leftarrow 1$ to n do

if $B \cup \{E(i)\} \in I$ then $B \leftarrow B \cup \{E(i)\}$endif

repeat

end *GREEDY*

定理 3.5 上述贪心算法给出了赋权拟阵(E, I, w)的最优基。

证明：假设算法产生的基$B = \{b_1, b_2, \cdots, b_r\}$不是最优基，则取$(E, I, w)$的最优基$\overline{B}$，因$B \subsetneq \overline{B}$，必存在$j$，$1 \le j \le r$,使$b_1, \cdots, b_{j-1} \in \overline{B}$, 但$b_j \notin \overline{B}$. 总可以选取$\overline{B}$, 使$j$取最大值。由$B$的生成过程知，对任意$x \in \overline{B} - \{b_1, \cdots, b_{j-1}\}$，总有$w(x) \le w(b_j)$, 这是因为作为$\overline{B}$的子集，$\{b_1, \cdots, b_{j-1}, x\}$是独立集，若$w(x) > w(b_j)$, 则在上述贪心算法的执行过程中，$x$必先于$b_j$进入$B$中，这与$x \notin \{b_1, \cdots, b_{j-1}\}$矛盾. 设$A = \{b_1, \cdots, b_{j-1}, b_j\}$，由$M$的交换性，可选取$\overline{B} - \{b_1, \cdots, b_j\}$中的$r - j$个元素加入$A$以致生成$M$的基$\tilde{B}$, 注意到$\tilde{B}$与$\overline{B}$仅有一元素不同，$\tilde{B} - \overline{B} = b_j$，记$z = \overline{B} - \tilde{B}$，则$z \in \overline{B} - \{b_1, \cdots, b_{j-1}\}$，因$w(z) \le w(b_j)$, 所以$f(\tilde{B}) \ge f(\overline{B})$, $\tilde{B}$也是最优基，但$\tilde{B}$对应的$j(\tilde{B}) \ge j(\overline{B}) + 1$, 与$\overline{B}$的选取矛盾。

例 3.7 带有限期的作业调度可归结为求一个赋权拟阵的最优基。

说明：E是n个作业构成的集合，令$I = \{X \subseteq E : X$ 中的作业都可以安排使其在限期以前完成$\}$. I显然满足遗传性，下面来证I满足交换性，从而(E, I)构成一拟阵。该证明要用到下面引理：

引理 3.1 $X \subseteq E$, 记$N_t(X)$($t \ge 0$为整数)为X中限期不超过t的

作业个数， 则下列三论断等价：

1. 作业集 X 是独立集；

2. 对于 $1 \le t \le n$, 有 $N_t(X) \le t$.

3. 如果 X 中的作业按限期的非递减顺序来调度，则 X 中所有作业都不会误期。

证明：$1 \Rightarrow 2$：如果有某 t, 使 $N_t(X) > t$, 则不存在一个调度使 X 中的所有作业都不误期，因为在 t 个单位时间段内完成 $N_t(X)$ 个作业是不可能的。

$2 \Rightarrow 3$：因 X 中作业按限期的非降顺序来调度的话，则第 k 个调度的作业的限期 d_k 必然不小于 k .若不然， $d_k < k$ ， 则 $d_1 \le d_2 \le \cdots \le d_k \le k-1$, 导致 $N_{k-1}(X) \ge k > k-1$, 矛盾。

设 X, $Y \in I$, $|X| < |Y|$, 记 k 是使 $N_t(Y) \le N_t(X)$ 的最大 t 值，因 $N_0(Y) = N_0(X) = 0$, $N_n(Y) = |Y| > |X| = N_n(X)$,这样的 k $(0 \le k < n)$ 总可以取到. 当 $j > k$ 时， $N_j(Y) > N_j(X)$, 取 $y \in Y - X$, $d_y = k+1$, 令 $\overline{X} = X \cup \{y\}$, 则 $\overline{X} \in I$, 因

$$N_t(\overline{X}) = \begin{cases} N_t(X), & 1 \le t \le k \\ N_t(X)+1, & k+1 \le t \le n \end{cases},$$

所以 $N_t(\overline{X}) \le \max(N_t(X), N_t(Y)) \le t$. 这就证明了 I 满足交换性。

n 个作业的效益作为该拟阵的赋权函数，此赋权拟阵的最优基正是带有限期的作业调度问题的最优解。

例 3.8　求赋权连通图$G=(V,\ E,\ w)$的最小生成树。

解：令$w_0=\max\limits_{x\in E}\{w(x)+1\}$，定义$\overline{w}(x)=w_0-w(x)$，$x\in E$．把$\overline{w}$作为图$G$圈拟阵的赋权函数，则生成的赋权拟阵的最优基正是我们要求的连通图$G=(V,\ E,\ w)$的最小生成树。此种情形下的贪心算法 3.3 可翻译成 Kruskal 算法。

最后，我们要指出，可用该贪心算法求解的本类最优化问题的可行解集合必构成一拟阵的独立集族。

定理 3.6 设I是E的非空子集族，若只要$X\in I$且$Y\subseteq X$就有$Y\in I$(即I满足遗传性)，则对E的元素取任意正实数权贪心算法得到的解都是最优的当且仅当(E,I)是一个拟阵。

证明：若(E,I)不是拟阵，即I不满足交换性，则存在$X\in I$，$Y\in I$，使$|X|<|Y|$，但对任意的$y\in Y-X$，恒有$X\cup\{y\}\notin I$．设

$$X\cap Y=\{e_1,\ \cdots,\ e_j\},$$

$$X=\{e_1,\ \cdots,\ e_j,\ x_1,\ \cdots,\ x_n\},$$

$$Y=\{e_1,\ \cdots,\ e_j,\ \mathrm{y}_1,\ \cdots,\ \mathrm{y}_m\},$$

构作

$$w(x)=\begin{cases}3, & x\in X\cap Y\\ 1+1/n, & x\in X-X\cap Y\\ 1+1/m, & x\in Y-X\cap Y\\ 1/|E|, & x\in E-X\cup Y\end{cases},$$

在此情况下，由贪心算法生成的解B必包含X，但因对于所有$y\in Y-X$ ($Y-X=Y-Y\cap X$ },恒有$X\cup\{y\}\notin I$，所以B中不含有$Y-Y\cap X$中的元素，这样就有

$$f(B) \le \sum_{x \in X} w(x) + \sum_{x \in E-X \cup Y} w(x) < 3j + n + 1 + 1 = 3j + n + 2,$$

$$f(Y) = \sum_{x \in Y} w(x) = 3j + m + 1 \ge 3j + n + 2 \ > \ f(B)$$

这说明贪心算法得到的解不是最优解。

5. 最优归并模式

问题：要将n个已排好序的外部文件X_1，…，X_n归并成一个排好序的文件。限定是每次只能将两个已排好序的文件进行归并，求花费最小的归并方案。

外部归并主要的花费是元素在内外存之间的移动，归并文件X_1，X_2的花费可用$|X_1|+|X_2|$来度量。

归并n个文件总共有多少种不同的方式呢？设此数为$g(n)$，则应有：$g(n) = C_n^2 g(n-1)$，$g(n) = n!(n-1)!/2^{n-1}$.

用贪心法来设计归并方案，我们选取的量度标准是，每次归并都选取待归并的文件中长度最小的两个文件。

任何归并方案都可以用一个二元归并树来表示，在该二元归并树中，叶子表示原始文件，内部结点对应着由归并生成的中间文件，于是归并的花费是该二元归并树的带权外部路径长度：$\sum_{i=1}^{n} d_i |X_i|$，其中d_i是文件X_i对应的叶子到根的路径长度。

使该带权外部路径长度最小的二元归并树正是一棵 Huffmann

树，上述的贪心算法实际上正是 Huffmann 树的生成算法，由 Huffmann 树的最优性， 可知给出的方案即为所求。

6. 最小根树（Minimum-Cost Arborescences）：多步骤贪心算法（A Multi-Phase Greedy Algorithm）

问题：设$G=(V,E)$是一个有向图，把某结点$r\in V$作为G的一个根来考虑。称G的生成子图$T=(V,F)$是（G的）以r为根的根树，如果T的基图是无向树，且在T中r到其它结点$v\in V-r$都存在有向道路。

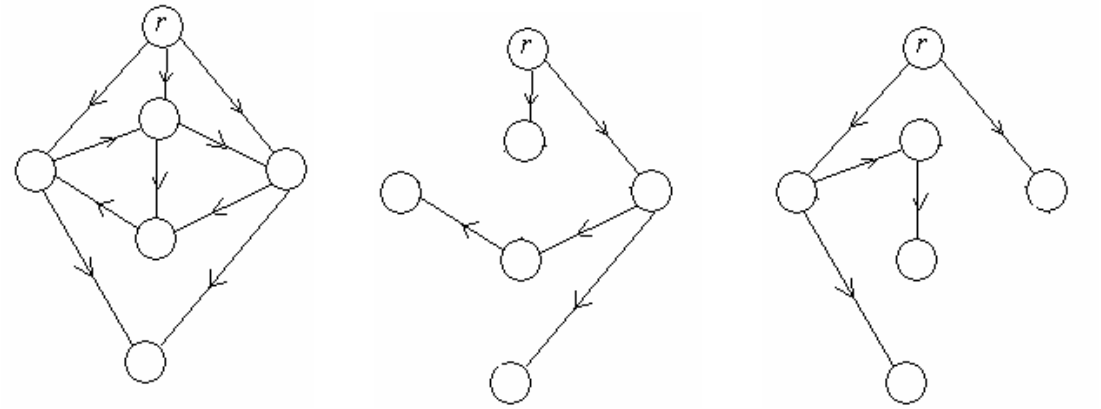


图 3.1

罗列如下结论以便了解根树的性质：

1. $T=(V,F)$是G的以r为根的根树当且仅当T中不含圈且对每个结点$v\neq r$，F中恰有一条边进入v。

2．有向图G存在以r为根的根树当且仅当存在从r到其它每个结点的有向道路。(可通过宽度优先检索法进行判定)

如果根树存在的话，可能有多个。现在给G的每条边一个正权

值，要求以r为根的最小根树（边权和最小的根树，也称为最优根树）。

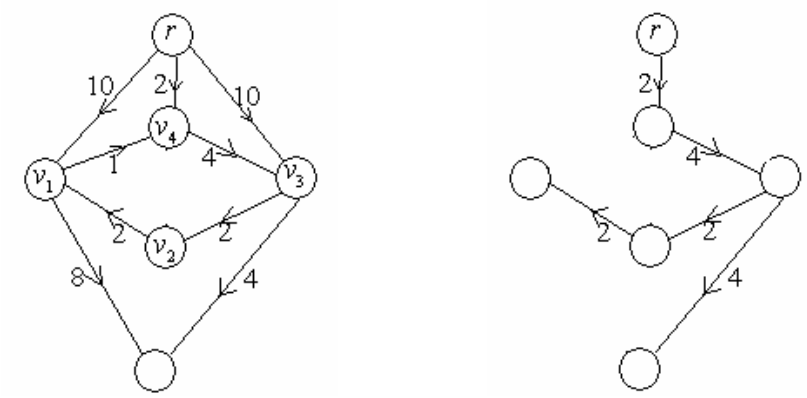


图 3.2

这个问题表面上看，类似于求赋权无向连通图的最小生成树。其实不然。如图 3.2 所示，右图是左图唯一的（以r为根的）最小根树。考察左图中的回路（圈）$v_1v_2v_3v_4v_1$，其中最小权的边v_1v_4不在最小根树中，反而其中最大权的边v_2v_3在最小根树中。

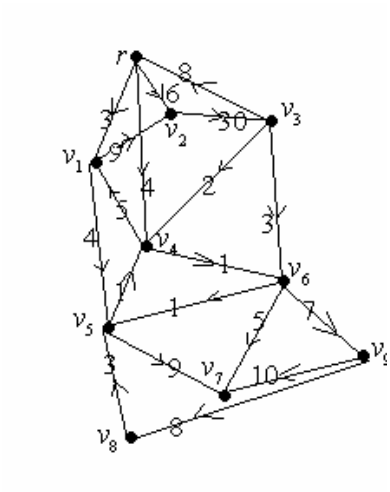
细究起来，我们会发现，每棵根树有$n-1$（n是G的结点个数）条边，这些边进入不同的结点，恰好一一对应于所进入的$n-1$个结点（除r之外的其它结点）。鉴于此，对每个结点$v \neq r$，找一条进入它的最小边(边权最小，此边权记为y_v)，设F^*是这$n-1$条边的集合。如果F^*中的边不会形成圈，则F^*中的这些边就形成一颗最小根树，这当然是非常理想的情况。但是如果F^*中的边形成某回路C（注意，显然根结点r不在C上），怎么办呢？为从长计议，考虑将G的权作如下改变：对G的每条进入某$v \neq r$的边e，边权减少y_v，即$c'_e = c_e - y_v$。

断言：T在边权$\{c_e\}$下是G的最小根树当且仅当T在边权$\{c'_e\}$下是G的最小根树。这是因为任何以r为根的根树T分别在边权

$\{c_e\}$和边权$\{c'_e\}$下的边权和相差一个常数（与根树T无关）：

$$\sum_{e\in T} c_e - \sum_{e\in T} c'_e = \sum_{v\neq r} y_v$$

这样问题就转换为求G在边权$\{c'_e\}$下的最小根树。此时，F^*中的边的权都是0。我们当然想尽量多用F^*中的边来形成根树。对F^*中的边形成的回路C来说，显然至少得舍弃其中一条边。如何舍弃呢？我们考虑将回路C收缩为一个超结点（supernode），



得到一个新的有向图$G'=(V',E')$，G中原来与C上的结点关联的边，结果在G'中与超结点关联，注意要删去超结点处形成的所有自环。当然G'的结点数变少了。后面将会说明G'的以r为根结点的最小根树T'将会在超结点处展开成G的以r为根结点的最小根树。这样就得到求最小根树的递归算法（对G的结点数递归）。

算法 3.6 递归算法描述如下：

If number of nodes of G is equals to 2, we can solve it directly.

Otherwise

{For each node $v\neq r$

Let y_v be the minimum cost of any edge entering node v.

Modify the costs of all edges e entering v to $c'_e = c_e - y_v$.

Choose one 0-cost edge entering each $v\neq r$,obtaining a set F^*

If F^* forms an arborescence, then return it

Else there is a directed cycle $C \subseteq F^*$

Contract C to a single supernode, yielding a graph $G' = (V', E')$.

Recursively find an optimal arborescence (V', F') in G' with cost $\{c'_e\}$.

Extend (V', F') to an arborescence (V, F) in G by adding all but one edge of C.}

定理 3.7 G'的以r为根结点的最小根树T'在超结点处能展开成G的以r为根结点的最小根树。

证明：论证由两个事实构成：

其一 G'的以r为根结点的每个根树T'能一一对应地展开为G的以r为根结点的这样的根树T：T中进入C的边（称边uv进入C，如果$u \notin C$而$v \in C$）只有一条。且在如此一一对应下，T'的边权和等于T的边权和。

其二 存在G的以r为根结点的这样的最小根树T：T中进入C的边只有一条。

其一之证明：设$T' = (V', F')$是G'的以r为根结点的任意一棵根树，设T'中进入超结点的边对应于G中的边wv，如图3.3所示，则在T'中将超结点展开成回路C，然后删去C上的边uv，就得到G的以r为根结点的这样的根树T：T中进入C的边只有一条。 并且容易验证不同的T'展开出来的T不同。反过来，设T是G的以r为根结点的

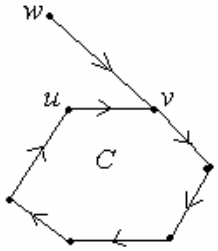


图 3.3

任意一棵这样的根树：T 中进入 C 的边只有一条，则在 T 中将 C 上的所有结点收缩成超结点，就得到 G' 的以 r 为根结点的一棵根树 T'。容易看出，T' 通过上述展开步骤恰好得到 T。最后，显然 T' 的边权和等于 T 的边权和，因为 C 上所有边的权都是 0。

其二之证明：取定以 r 为根结点的一棵最小根树 $\tilde{T}$，不妨设在 $\tilde{T}$ 中，进入 C 的边不止一条。设 wv 是进入 C 的边，满足：在 $\tilde{T}$ 中，根结点 r 到结点 v 的路径是 r 到 C 上结点的最短路径，这样，该路径上，除 wv 以外，不再有进入 C 的边。对 $\tilde{T}$ 中的边作如下替换：除 wv 以外，其它进入 C 的边用 C 上的边来替代，如图 3.4 所示，u_1v_1 替代 w_1v_1，u_2v_2 替代 w_2v_2。结果得到另外一棵以 r 为根结点的根树 T（C 中的边除 uv 之外全在 T 中），T 的边权和不会超过 $\tilde{T}$ 的边权和，这是因为替换进来的新边的权都是 0。所以 T 也是最小根树。

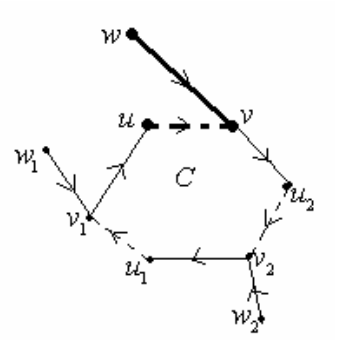
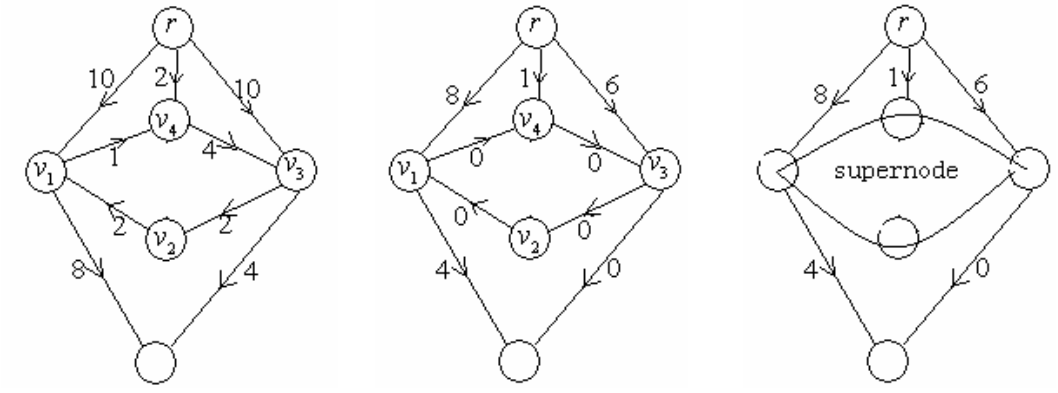
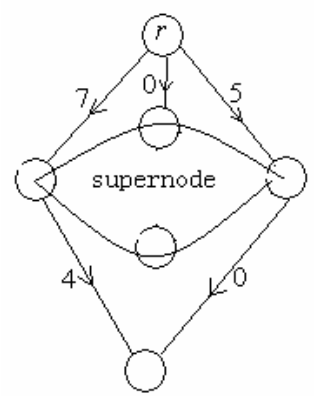


图 3.4

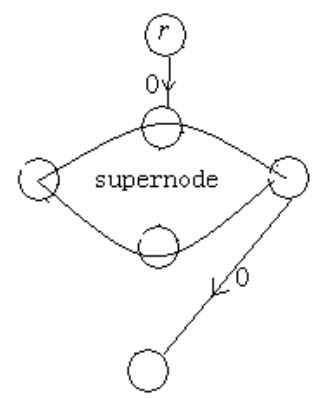
例 3.9 下图表示一个用上述算法求最小根树的例子



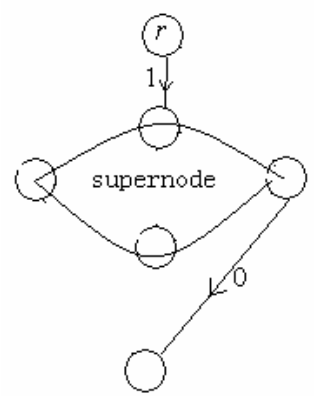
原赋权有向图 G　　改变 G 权　　收缩回路得 G'



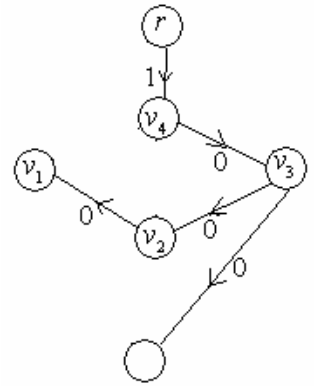
改变G'权



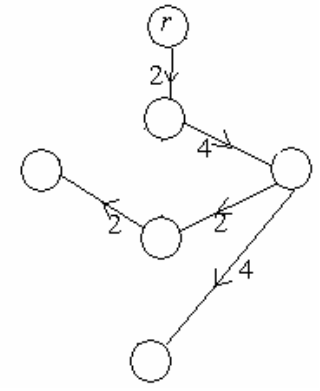
G'的最小根树



G'原权下的最小根树



展开成G的最小根树



恢复成原权的最小根树

7. 最小延误调度问题（Scheduling to Minimize Lateness）：轮换论证方法（An Exchange Argument）

问题：有一个独占性设备和n个使用该设备的任务，每个任务i（$1 \le i \le n$）使用设备的时间为t_i，要求在d_i时刻前完成。问：如何在设备上安排这n个任务的使用顺序，能使得延误最小？

解释：设在0时刻开始使用设备，在排列σ下（排列σ的第i个位置摆放的任务记为$\sigma(i)$），任务i在时刻$s_i(\sigma)$完成， 则任务i的延误

时间为：

$$L_i(\sigma)=\max(0,s_i(\sigma)-d_i)$$

排列σ的延误定义为

$$L(\sigma)=\max_{1\le i\le n}L_i(\sigma)。$$

定理 3.8 设排列σ满足：$d_{\sigma(1)}\le d_{\sigma(2)}\le\cdots\le d_{\sigma(n)}$，则排列$\sigma$的延误$L(\sigma)$最小。

证明：证明由下面两个断言构成

断言 1：若排列π满足：存在j，$1\le j<n$，使得$d_{\pi(j)}\ge d_{\pi(j+1)}$，则交换任务$\pi(j)$与$\pi(j+1)$的位置得到新排列$\overline{\pi}$，我们有$L(\overline{\pi})\le L(\pi)$。

断言 2：所有排列都可通过有限次断言 1 中的交换变成排列σ。

断言 1 的证明：注意到

（1） $s_{\pi(j)}(\pi)=s_{\pi(j-1)}(\pi)+t_{\pi(j)}$，

$s_{\pi(j+1)}(\pi)=s_{\pi(j-1)}(\pi)+t_{\pi(j)}+t_{\pi(j+1)}$，

$s_{\pi(j+1)}(\overline{\pi})=s_{\overline{\pi}(j)}(\overline{\pi})=s_{\pi(j-1)}(\pi)+t_{\pi(j+1)}$，

$s_{\pi(j)}(\overline{\pi})=s_{\overline{\pi}(j+1)}(\overline{\pi})=s_{\pi(j-1)}(\pi)+t_{\pi(j+1)}+t_{\pi(j)}$，

（2）$L_{\pi(j)}(\pi)=\max(0,s_{\pi(j-1)}(\pi)+t_{\pi(j)}-d_{\pi(j)})$，

$L_{\pi(j+1)}(\pi)=\max(0,s_{\pi(j-1)}(\pi)+t_{\pi(j)}+t_{\pi(j+1)}-d_{\pi(j+1)})$，

$L_{\pi(j+1)}(\overline{\pi})=\max(0,s_{\pi(j-1)}(\pi)+t_{\pi(j+1)}-d_{\pi(j+1)})$，

$L_{\pi(j)}(\overline{\pi})=\max(0,s_{\pi(j-1)}(\pi)+t_{\pi(j)}+t_{\pi(j+1)}-d_{\pi(j)})$，

（3）$s_{\pi(j-1)}(\pi)+t_{\pi(j)}+t_{\pi(j+1)}-d_{\pi(j+1)}$

$\ge s_{\pi(j-1)}(\pi)+t_{\pi(j)}+t_{\pi(j+1)}-d_{\pi(j)}$，

$$s_{\pi(j-1)}(\pi)+t_{\pi(j)}+t_{\pi(j+1)}-d_{\pi(j+1)}$$
$$\geq s_{\pi(j-1)}(\pi)+t_{\pi(j)}-d_{\pi(j)},$$
$$s_{\pi(j-1)}(\pi)+t_{\pi(j)}+t_{\pi(j+1)}-d_{\pi(j+1)}$$
$$\geq s_{\pi(j-1)}(\pi)+t_{\pi(j+1)}-d_{\pi(j+1)},$$

我们有

$$L_{\pi(j+1)}(\pi)\geq L_{\pi(j)}(\bar{\pi}),$$
$$L_{\pi(j+1)}(\pi)\geq L_{\pi(j)}(\pi)$$
$$L_{\pi(j+1)}(\pi)\geq L_{\pi(j+1)}(\bar{\pi}),$$

从而

$$\max(L_{\pi(j+1)}(\pi),L_{\pi(j)}(\pi))\geq\max(L_{\pi(j+1)}(\bar{\pi}),L_{\pi(j)}(\bar{\pi}))$$

又注意到

$$L_{\pi(i)}(\pi)=L_{\pi(i)}(\bar{\pi}),\quad i\neq j,\ j+1,$$

我们有

$$\max_{1\leq i\leq n}L_i(\pi)\geq\max_{1\leq i\leq n}L_i(\bar{\pi}),$$

即

$L(\pi)\geq L(\bar{\pi})$。

断言 2 是显然的。

作业：

1. n个活动请求安排在某个多功能厅举行，第i $(1\leq i\leq n)$个活动进行的时间段是$[s_i,\ f_i]$. 该多功能厅任何时间都不能同时举办两个活动，所以这n个活动可能会发生时间冲突，不能都一一安排在该多功

能厅，那么，自然面临这样的一个问题：最多可以安排多少个活动，使之不发生冲突？请给出你的设计策略求解该问题。

2．这是一道推理题。5 个海盗抢到了 100 颗宝石，每一颗都一样的大小和价值连城。他们决定这么分：

(1) 抽签决定自己的号码（1，2，3，4，5）

(2) 首先，由 1 号提出分配方案，然后大家 5 人进行表决，当且仅当半数和超过半数的人同意时，按照他的提案进行分配，否则将被扔入大海喂鲨鱼。

(3) 如果 1 号死后，再由 2 号提出分配方案，然后大家 4 人进行表决，当且仅当半数或超过半数的人同意时，按照他的提案进行分配，否则将被扔入大海喂鲨鱼。

(4)依此类推.....

条件：每个海盗都是很贪婪的人，同时都能很理智的判断得失，从而做出选择。

问题：第一个海盗提出怎样的分配方案才能够使自己免于被扔入大海喂鲨鱼同时收益最大？

3. 某机器要处理n（$n \geq 2$）个工件，工件i（$1 \leq i \leq n$）需占用机器的时长为t_i，限定在d_i时刻前完成，当然，$d_i \geq t_i$。请给出算法（要求时间复杂度为$O(n^2)$）来确定这n个工件在机器上的一个处理顺序，使得能够在期限前完成的作业数目最多。并证明你的算法的正确性。

第四章 动态规划(Dynamic Programming)

1. 一般方法概述(The general Method)

动态规划是用于解决多阶段决策最优化问题的算法设计方法。该术语是 1957 年 Richard Bellman 在描述一类最优控制问题时提出的, 事实上该术语描摩的是问题特征, 而非方法特征.该类问题的特征是，它的活动过程可以分为有序的若干阶段，而且在任一阶段i，过程在阶段i以后的行为仅依赖于阶段i的过程状态，与阶段i以前过程如何达到这种状态的方式无关.它满足所谓的最优性原理(Principle of Optimality), 该原理指出，过程的最优决策序列具有如下性质：无论过程的初始状态和初始决策是什么，其余的决策必须相对于初始决策产生的状态构成一个最优决策序列。概略地说，动态规划是建立在最优性原理基础上的枚举法，该问题的解可被看作k个阶段决策的结果，其子问题解相应地可被看作$j(1\le j\le k)$个阶段决策的结果，其中$j+1$个阶段子问题的最优解是由j个阶段子问题的最优解生成的，由此就可得到子问题间的递推关系式，从而解出整个问题的最优解. 所以解决问题的关键在于获取各阶段间的递推关系式。

描述上述机理的一个好的模型是多段图问题。

例 4.1 {多段图问题} 如图 4.1 所示，多段图$G=(V, E)$是一个有向图，它具有如下特性：图中的结点被划分成$k(k\ge 2)$个不相交的集合V_i $(1\le i\le k)$, 其中V_1和V_k分别只包含一个结点s (称为源点)和t (汇点)；图中所有的边(u, v)均具有如下性质：若$u\in V_i$，则

$v \in V_{i+1}$，$1 \le i < k$,且每条边(u, v)均附有成本$c(u, v)$．从s到t的一条路径成本是这条路径上边的成本和。多段图问题(Multistage Group Problem)要求的是s到t的最小成本路径（最短路径）。每个集合V_i定义图中的一个阶段，由于E的约束，每条从s到t的路径都是从第 1 阶段开始，到达第 2 阶段，再到第 3 阶段，…,最后终止在第k段(即t点)。

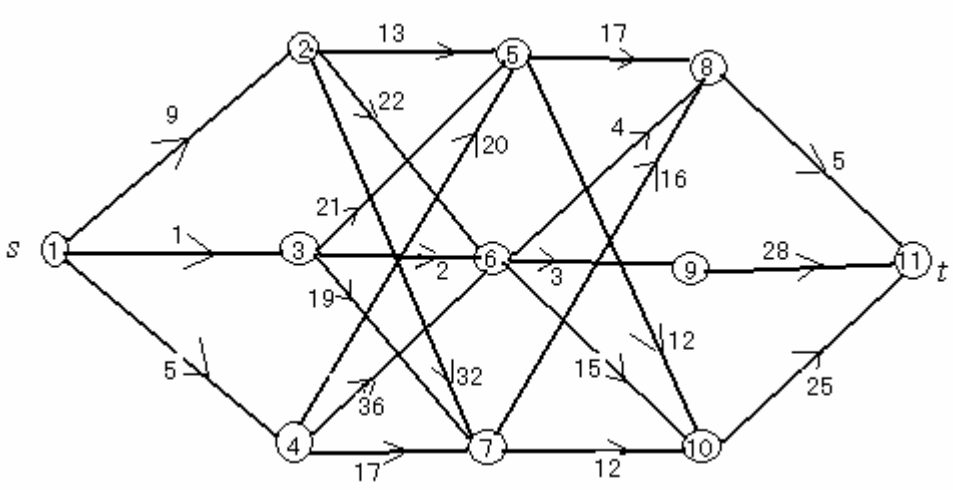


图 4.1

对于每一条由s到t的路径$v_{i_1} v_{i_2} \cdots v_{i_{k-1}} v_{i_k}$（$v_{i_1} = s$，$v_{i_k} = t$），可以把它看成在$k-1$个阶段中作出的某个决策序列的结果。在阶段$j$（$1 \le j \le k-1$），过程所处的状态是停留的结点$v_{i_j}$；第$j$次决策就是确定从结点$v_{i_j}$沿边$v_{i_j} v_{i_{j+1}}$到达阶段$j+1$，它只与结点$v_{i_j}$有关，而与如何从$s$到达结点$v_{i_j}$的路径无关。如果$v_{i_1} v_{i_2} \cdots v_{i_{k-1}} v_{i_k}$是一条从$s$到$t$的最短路径，则对任何$1 \le j < l \le k-1$来说，$v_{i_j} v_{i_2} \cdots v_{i_l}$是从$v_{i_j}$到$v_{i_l}$的最短路径。所以，多段图的最短路径满足最优性原理，可以用动态规划方法求解。

从上面的分析可以发现，从阶段1到阶段k的最短路径，是从阶段1到阶段$k-1$的最短路径延伸生成的；而从阶段1到阶段$k-1$的最短路径，又是从阶段1到阶段$k-2$的最短路径延伸生成的；…；一般来说，从阶段1到阶段$j\ (2\le j\le k)$的最短路径，是从阶段1到阶段$j-1$的最短路径延伸生成的。所以多段图的最短路径问题$l(s,t)$，可归结为求解下面的递归式：

$$\begin{cases} l(s,s)=0, \\ l(s,v_{i_j})=\min\limits_{v_{i_{j-1}}}\{l(s,v_{i_{j-1}})+c(v_{i_{j-1}},v_{i_j})\},\ 2\le j\le k \end{cases}$$

对于许多貌似指数复杂度的问题，采用动态规划，往往能给出多项式复杂度的算法。后面各节中就介绍了一些这样的例子。

2. 矩阵连乘问题(Multiplying a Sequence of Matrices)}

考虑n个矩阵M^1,M^2，…，M^n(上标表示矩阵的序号，而不是指数)的连乘积问题：

$$M^1\bullet M^2\bullet\cdots\bullet M^n=?$$

由矩阵乘法的定义，这n个矩阵中两相邻的矩阵有这样的性质：前者的列数与后者的行数相等. 因此这n个矩阵的维数的大小只要$n+1$个正整数即可确定(矩阵M^i有r_{i-1}行r_i列).

易知，任意$p\times q$阶的矩阵A与任意$q\times r$阶的矩阵B相乘需花费pqr次乘法，pqr次加法，及pqr次赋值，因此可用pqr来表征这两个矩阵相乘的运算量。

矩阵乘法适合结合律：

$$(M^1 \bullet M^2) \bullet M^3 = M^1 \bullet (M^2 \bullet M^3),$$

因此，$M^1 \bullet M^2 \bullet \cdots \bullet M^n$ 有多种不同的结合方式，即有不同的乘法运算顺序。不同的结合方式，当然能得到同样的运算结果，但是会导致不同的运算过程。

我们用 $M^{i,j}$ 表示矩阵连乘积: $M^i \bullet \cdots \bullet M^j$,考虑如下四个矩阵的乘积($r_0 = 10,\ r_1 = 30,\ r_2 = 70,\ r_3 = 1,\ r_4 = 200$):

$$M^1_{10\times 30} \bullet M^2_{30\times 70} \bullet M^3_{70\times 1} \bullet M^4_{1\times 200}.$$

该积按我们的约定可记作：$M^{1,4}_{10\times 200}$.

为比较不同结合方式运算量的差异，考察上式的 5 种不同的结合方式：

(1) $((M^1_{10\times 30} \bullet M^2_{30\times 70}) \bullet M^3_{70\times 1}) \bullet M^4_{1\times 200}$

$= (M^{1,2}_{10\times 70} \bullet M^3_{70\times 1}) \bullet M^4_{1\times 200}$

$= M^{1,3}_{10\times 1} \bullet M^4_{1\times 200} = M^{1,4}_{10\times 200},$

其中包含了三个矩阵乘法，其运算量分别为: $10\times 30\times 70 = 21000,$ $10\times 70\times 1 = 700,\ 10\times 1\times 200 = 2000,$ 合起来为 $23700.$

(2) $(M^1_{10\times 30} \bullet (M^2_{30\times 70} \bullet M^3_{70\times 1})) \bullet M^4_{1\times 200}$

$= (M^1_{10\times 30} \bullet M^{2,3}_{30\times 1}) \bullet M^4_{1\times 200}$

$= M^{1,3}_{10\times 1} \bullet M^4_{1\times 200} = M^{1,4}_{10\times 200},$

总运算量为

$$30\times 70\times 1 + 10\times 30\times 1 + 10\times 1\times 200 = 4400.$$

(3) $(M^1_{10\times 30} \bullet M^2_{30\times 70}) \bullet (M^3_{70\times 1} \bullet M^4_{1\times 200})$

$= M^{1,2}_{10\times 70} \bullet (M^3_{70\times 1} \bullet M^4_{1\times 200})$

$= M_{10\times70}^{1,2} \bullet M_{70\times200}^{3,4} = M_{10\times200}^{1,4}$,

总运算量为

$$10\times30\times70+70\times1\times200+10\times70\times200=175000.$$

(4) $M_{10\times30}^{1}\bullet(M_{30\times70}^{2}\bullet(M_{70\times1}^{3}\bullet M_{1\times200}^{4}))$

$= M_{10\times30}^{1}\bullet(M_{30\times70}^{2}\bullet M_{70\times200}^{3,4})$

$= M_{10\times30}^{1}\bullet M_{30\times200}^{2,4} = M_{10\times200}^{1,4}$,

总运算量为

$$70\times1\times200+30\times70\times200+10\times30\times200=494000.$$

(5) $M_{10\times30}^{1}\bullet((M_{30\times70}^{2}\bullet M_{70\times1}^{3})\bullet M_{1\times200}^{4})$

$= M_{10\times30}^{1}\bullet(M_{30\times1}^{2,3}\bullet M_{1\times200}^{4})$

$= M_{10\times30}^{1}\bullet M_{30\times200}^{2,4} = M_{10\times200}^{1,4}$,

总运算量为

$$30\times70\times1+30\times1\times200+10\times30\times200=68100.$$

由这个实例可以看出，结合方式不同，运算量可能相差很大，这促使人们寻找一种运算量最小的结合方式。

若采用枚举法来找运算量最小的结合方法，先来估算结合方式数目。设 $p(n)$ $(n\geq1)$是n个矩阵连乘可能的结合方式数目，则有

$$p(n)=\begin{cases}1, & n=1\\ p(1)p(n-1)+p(2)p(n-2)+\cdots+p(n-1)p(1), & n\geq2\end{cases}$$

(4.1)

它是著名的 Catalan 数：$p(n)=\frac{1}{n}C_{2(n-1)}^{n-1}$. 易知 $p(n)\geq2^{n-2}$，（事

实上，更好的估计是：$p(n)=\Omega(4^n/n^{3/2})$，这可以利用 Strling 公式：$n!=\Theta(\sqrt{n}(n/e)^n)$，或更准确的 Strling 不等式：$\sqrt{2\pi n}(n/e)^n e^{1/[12(n+1)]}\le n!\le\sqrt{2\pi n}(n/e)^n e^{1/[12n]}$得到）从而枚举法的时间复杂度为$n$的指数函数。

注：$p(n)$可用母函数法求解。令

$$g(x)=\sum_{n=1}^{\infty}p(n)x^n \qquad (4.2)$$

则有

$$g^2(x)=\sum_{n=2}^{\infty}\left(\sum_{k=1}^{n-1}p(k)\,p(n-k)\right)x^n=\sum_{n=2}^{\infty}p(n)x^n$$

$$=\sum_{n=1}^{\infty}p(n)x^n-x=g(x)-x \qquad (4.3)$$

即

$$g^2(x)-g(x)+x=0 \qquad (4.4)$$

解之得

$$g(x)=\frac{1\pm\sqrt{1-4x}}{2} \qquad (4.5)$$

注意到$g(0)=0$，在式(4.5)中舍去"+"号得

$$g(x)=\frac{1-\sqrt{1-4x}}{2} \qquad (4.6)$$

写出$\sqrt{1-4x}$的泰勒展式：

$$\sqrt{1-4x}=1-2x+\cdots+\frac{\frac{1}{2}(\frac{1}{2}-1)\cdots(\frac{1}{2}-n+1)}{n!}(-4x)^n+\cdots$$

(4.7)

由$g(x)$泰勒展式的唯一性知

$$p(n)=-\frac{\frac{1}{2}(\frac{1}{2}-1)\cdots(\frac{1}{2}-n+1)}{2n!}(-4)^n=\frac{1\cdot3\cdot5\cdots(2n-3)2^{n-1}}{n!}$$

$$=\frac{1\cdot3\cdot5\cdots(2n-3)2\cdot4\cdots(2n-2)}{n!(n-1)!}=\frac{(2n-2)!}{n!(n-1)!}=\frac{1}{n}C_{2n-2}^{n-1}$$

下面考虑用动态规划法求解该问题，假设最优结合法的最后一次乘法是$M^{1,i}$和$M^{i+1,n}$相乘。即

$$M^{1,n}=M^{1,i}\bullet M^{i+1,n},\ 1\le i<n$$

则$M^{1,i}$和$M^{i+1,n}$各自的结合法也应是最优的，满足最优性原理. 一般地，设$m_{i,j}$表示$M^{i,j}$所需的最小运算量，则

$$\begin{cases} m_{i,i}=0, & 1\le i\le n \\ m_{i,j}=\min\limits_{i\le k<j}\{m_{i,k}+m_{k+1,j}+r_{i-1}r_kr_j\}, & i<j \end{cases}.$$

可通过一个n阶方阵来表示出各个最小运算量$m_{i,j}$及分割$M^{i,j}=M^{i,k}\bullet M^{k+1,j}$的最佳分割点$k$.

当$j>i$时，$1\le j-i\le n-1$, 因此，先求$j-i=1$时的$m_{i,j}$，其次求$j-i=2$时的$m_{i,j}$，继续下去，直到求得$j-i=n-1$时的$m_{i,j}$即$m_{1,n}$. 求方阵D可表示为下面的算法。

procedure $MATD(N,\ R,\ D)$

//N是矩阵(连乘的)个数，$R(N+1)$存放N个矩阵的行列数，D是

//N阶方阵,$D(i,\ j)=m_{i,\ j}(i<j)$,//

//$D(j,\ i)$=($M^{i,j}=M^{i,k}\bullet M^{k+1,j}$)的最优分割点$k$.//

integer N, I, J, K, T, $R(N+1)$, $D(N,\ N)$

for $I\leftarrow 1$ to N do

$D(I,\ I)=0$

repeat

for $I\leftarrow 1$ to $N-1$ do

for $J\leftarrow 1$ to $N-I$ do

$D(J,\ J+I)=MAXINT$

for $K\leftarrow 0$ to $I-1$

$T=D(J,\ J+K)+D(J+K+1,\ J+I)$

$+R(J\text{-}1)\cdot R(J+K)\cdot R(J+I)$

if $T<D(J,\ J+I)$

then $D(J,\ J+I)\leftarrow T$; $D(J+I,\ J)\leftarrow J+K$

endif

repeat

repeat

repeat

end $MATD$

由$D(j,\ i)(i<j)$确定N个矩阵的最佳结合方式，这由一个递

归过程来完成。

procedure $P(I, J)$

//过程P给出矩阵M^I，M^{I+1}，$\cdots$，M^J连乘的最佳结合方式，//

//也就是在M^I，M^{I+1}，$\cdots$，M^J中添加表示结合顺序的括弧.//

case

: $I = J$: $write(\ '(M^I)'\ )$

: $I+1 = J$: $write(\ '(M^I M^J)'\ )$

: else:

{

$K \leftarrow D(J, I)$

$write(\ '('\ )$

call $P(I, K)$

call $P(K+1, J)$

$write(\ ')'\ )$

}

endcase

end P

主程序中可这样调用过程P: call $P(1, N)$

过程P的工作是加上一些括弧，计算量为$\Theta(n)$，无足轻重，因此，整个算法的时间复杂度取决于过程$MATD$.

过程$MATD$最内层 for 循环体的执行总数是

$$\sum_{i=1}^{n-1}(n-i)i=\sum_{i=1}^{n-1}(ni-i^2)=\frac{n^2(n-1)}{2}-\frac{n(n-1)(2n-1)}{6}=\Theta(n^3)$$

即该算法的时间复杂度为$\Theta(n^3)$, 是多项式函数。

3. 最优二分检索树(Constructing Optimal Binary Search Trees)

在第二章关于二分检索那一节里，我们细致地讨论了二元比较树，二元比较树就是一种二分检索树，二分检索树的一般形式如下定义。

定义 4.1 二分检索树(Binary Search Tree) T 是一棵二元树，它或者为空，或者其每个结点含有一个可比较大小的数据元素，并且

(1) T 的左子树的所有元素比根结点T 中的元素小；

(2) T 的右子树的所有元素比根结点T 中的元素大；

(3) T 的左子树和右子树也是二分检索树。

对于一个给定的标识符集合，可以用若干棵不同的二分检索树来表示。设包含n个标识符的二分检索树的数目为$q(n)$, 则

$$\begin{cases} q(0)=1, \\ q(n)=q(0)q(n-1)+q(1)q(n-2)+\cdots+q(n-1)q(0),\ n\geq 1 \end{cases}$$

易知，$q(n)=p(n+1)=\frac{1}{n+1}C_{2n}^{n}$.（$p(n)$见上一节）

二分检索树是为了确定一个给定的标识符 X 是否在它所表示的标识符集合中出现，将 X 先与根比较，如果 X 等于根中标识符，则检索成功地终止；否则，视 X 比根中元素大或小来决定在左子树还

是右子树中继续检索下去。

问题：给定一个标识符集合$\{a_1, \cdots, a_n\}$，希望产生一个对应的二分检索树，使得检索花费的平均次数最少。

设待检索元X取a_i($1 \le i \le n$)的概率为$P(i)$，X满足$a_i < X < a_{i+1}$($1 \le i \le n$，a_0=最小元，a_{n+1}=最大元)的概率为$Q(i)$，则

$$\sum_{i=1}^{n} P(i) + \sum_{i=0}^{n} Q(i) = 1.$$

对于一个关于该标识符集合的二分检索树T来说，其每个内结点对应着某一个a_i，每个外结点对应着某一个使$a_i < X < a_{i+1}$的状况E_i.

平均检索时间为

$$COST(T) = \sum_{1 \le i \le n} P(i)\ level(a_i) + \sum_{0 \le i \le n} Q(i)\ (level(E_i) - 1)$$

我们知道当$P(i)$，$Q(i)$取值相同时，当且仅当二分检索树是一棵均衡树时，平均检索时间为最小。对于一般情况而言，可用动态规划法求解最优二分检索树。

设二分检索树T的根结点为a_k，则其左子树L是关于标识符集合$\{a_1, \cdots, a_{k-1}\}$的二分检索树，其右子树R是关于标识符集合$\{a_{k+1}, \cdots, a_n\}$的二分检索树，记

$$COST(L) = \sum_{1 \le i < k} p(i) \cdot level(a_i) + \sum_{0 \le i < k} Q(i) \cdot (level(E_i) - 1)$$

$$COST(R) = \sum_{k < i \le n} p(i) \cdot level(a_i) + \sum_{k \le i \le n} Q(i) \cdot (level(E_i) - 1)$$

其中级数$level$是分别关于L和R而言的。记

$$w(i,\ j)=Q(i)+\sum_{i+1\le l\le j}(P(l)+Q(l))$$

则

$$COST(T)=p(k)+COST(L)+w(0,\ k-1)+COST(R)+w(k,\ n)$$
$$=COST(L)+COST(R)+w(0,\ n)$$

由上式可知，若T是最优二分检索树，则其左子树L和右子树R也必是最优二分检索树，满足最优性原理。一般地，记$C(i,\ j)$是对应于标识符集合$\{a_{i+1},\ \cdots,\ a_j\}$的最小花费，则

$$\begin{cases}C(i,\ i)=0,\\ C(i,\ j)=\min\limits_{i<k\le j}\{C(i,\ k-1)+C(k,\ j)\}+w(i,\ j),\ i<j\end{cases}$$

(4.7)

与矩阵连乘问题的解法完全类似的，可用下列步骤解上式，从而最终解得$C(0,\ n)$，且得到最优二分检索树。首先计算所有使得$j-i=1$的$C(i,\ j)$，接着计算所有使得$j-i=2$的$C(i,\ j)$，$\cdots$，最后计算使得$j-i=n$的$C(i,\ j)$即$C(0,\ n)$.

记$T_{i,\ j}$是最小花费$C(i,\ j)$对应的二分检索树，其根为$R(i,\ j)$，$R(i,\ j)$正是使(4.7)式取最小值的k.

算法 4.5 找最小成本二分检索树

procedure $OBST(P,\ Q,\ n)$

//Given n distinct identifiers $a_1<a_2<\cdots<a_n$ and probabilities//

//$P(1:\ n)$ and $Q(0:\ n)$. This algorithm computes the cost//

// $C(i,\ j)$ of optimal binary search trees $T_{i,\ j}$ for identifiers//

// $\{a_{i+1}, \cdots, a_j\}$. It also computes $R(i, j)$, the root of $T_{i,j}$.// // $w(i, j)$ is the weight of $T_{i,j}$.//

integer $R(0:n, 0:n)$

for $i \leftarrow 0$ to $n-1$do

$(w(i,i), R(i,i), C(i,i)) \leftarrow (Q(i), 0, 0)$ //initialization//

$(w(i,i+1), R(i,i+1), C(i,i+1)) \leftarrow$

$(Q(i)+Q(i+1)+P(i+1), i+1, Q(i)+Q(i+1)+P(i+1))$

//Optimal trees with one node //

repeat

$(w(n,n), R(n,n), C(n,n)) \leftarrow (Q(n), 0, 0)$

for $\mathrm{m} \leftarrow 2$ to n do // find optimal trees with m nodes//

for $i \leftarrow 0$ to $n-m$ do

$j \leftarrow i+m$

$w(i, j) \leftarrow w(i, j-1) + p(j) + Q(j)$

$k \leftarrow$ a value of $l \in [R(i, j-1), R(i+1, j)]$ that minimizes $C(i, l-1)+C(l, j)$

//solve (4.7) using knuth's result (见下面注) //

$C(i, j) \leftarrow w(i, j) + C(i, k-1) + C(k, j)$

$R(i, j) \leftarrow k$

repeat

repeat

end $OBST$

注：Knuth 指出，在(4.7)式中，最优的 k 可在区间 $R(i, j-1) \le k \le R(i+1, j)$ 求得。

这个结果的完整证明由下面几个步骤组成：(Knuth, second edition, Volume 3, 456-457)

Step1: Given that $p(n) = q(n) = 0$ and that the other weights are nonnegative, prove that an optimum tree for $(p(1), \cdots, p(n);\ q(0), \cdots, q(n))$ may be obtained by replacing

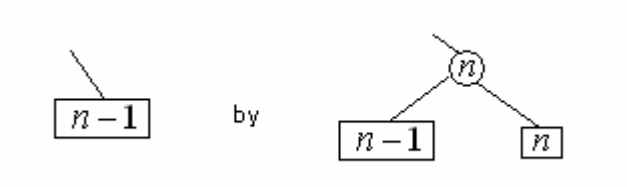


Figure 4.1

In any optimum tree for $(p(1), \cdots, p(n-1);\ q(0), \cdots, q(n-1))$.

Step 2: Let A and B be nonempty sets of real numbers, and define $A \le B$ if the following property holds:

$(a \in A, b \in B, \text{ and } b < a)$ implies $(a \in B \text{ and } b \in A)$

a) Prove that this relation is transitive on nonempty sets.

b) Prove or disprove: $A \le B$ if and only if $A \le A \cup B \le B$.

Step 3: Let $(p(1), \cdots, p(n);\ q(0), \cdots, q(n))$ be nonnegative weights, where $p(n) + q(n) = x$. Prove that as x varies from 0 to ∞, while $(p(1), \cdots, p(n-1);\ q(0), \cdots, q(n-1))$ are held constant, the cost $C(0, n)$ of an optimum binary search tree is a concave, continuous, piecewise linear function of x with integer slopes.

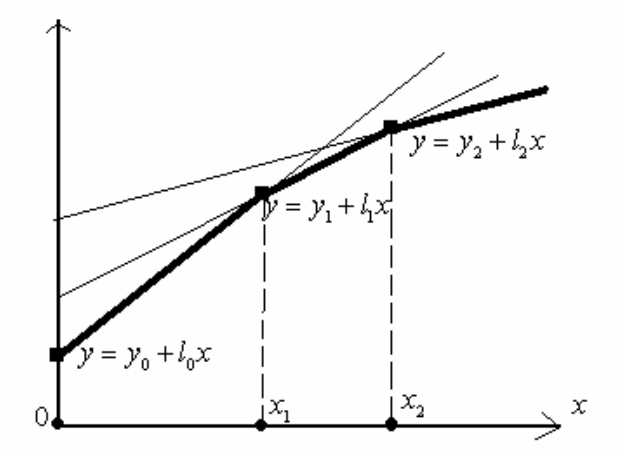


Figure 4.2

In other words, prove that there exist positive integers $l_0 > l_1 > \cdots > l_m$ and real constants $0 = x_0 < x_1 < \cdots < x_m < x_{m+1} = \infty$ and $y_0 < y_1 < \cdots < y_m$ such that $C(0,n) = y_h + l_h x$ when $x_h \le x \le x_{h+1}$, for $0 \le h \le m$.

Step 4: The object of this step is to prove that the sets of roots $R(i, j)$ of optimum binary search trees satisfy

$$R(i, j-1) \le R(i, j) \le R(i+1, j), \quad \text{for } j - i \ge 2,$$

in term of the relation defined in above, when the weights $(p(1), \cdots, p(n);\ q(0), \cdots, q(n))$ are nonnegative. The proof is by induction on $j - i$; our task is to prove that $R(0, n-1) \le R(0, n)$, assuming that $n \ge 2$ and that the stated relation holds for $j - i \le n$. (By left-right symmetry it follows that $R(0, n) \le R(1, n)$.)

a) Prove that $R(0, n-1) \le R(0, n)$ if $p(n) = q(n) = 0$.(See step 1)

b) Let $p(n) + q(n) = x$. In the above notation, Let R_h be the set $R(0, n)$ of optimum roots when $x_h < x < x_{h+1}$, and let R'_h be

the set of optimum roots when $x = x_h$. Prove that

$$R'_0 \le R_0 \le R'_1 \le R_1 \le \cdots \le R'_m \le R_m .$$

Hence by part (a) and step 2 we have $R(0, n-1) \le R(0, n)$ for all x. (Hint: Consider the case $x = x_h$, and assume that both the trees are optimum, with $r > s$ and $l > l'$ (Shown as Figure 4.3). Use the induction hypothesis to prove that there is an optimum tree with root r such that n is at level l', and an optimum tree with root s such that n is at level l.)

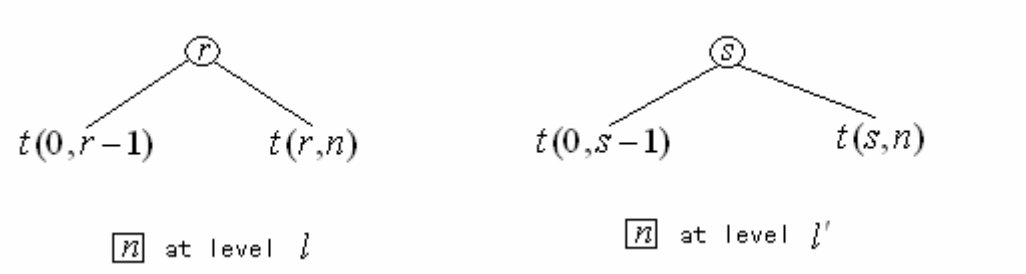


Figure 4.3

Proof:

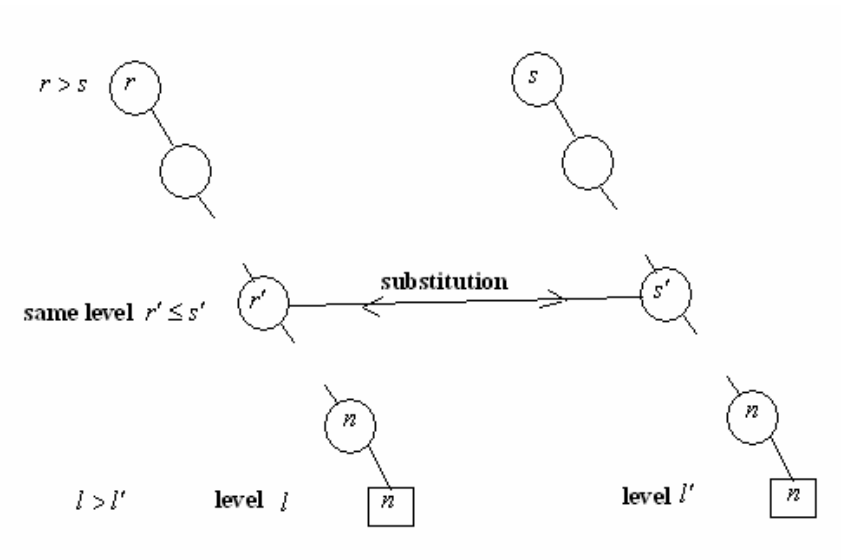


Figure 4.4

计算复杂度主要在两重循环中，关于每个m的内层循环，所花费的操作数为

$$\sum_{i=0}^{n-m}(R(i+1,i+m)-R(i,i+m-1)+1)$$

$$=n-m+1+R(n-m+1,n)-R(0,m-1)\le 2n$$

不超过$2(R(n-m+1,\ n)-R(0,\ m-1))\le 2n$，故总复杂度为$O(n^2)$.

4. $0/1$背包问题

问题：极大化：$\sum_{1\le i\le n} p_i x_i$

约束条件：$\sum_{1\le i\le n} w_i x_i \le M$，

$$x_i=0,\ 1,\ 1\le i\le n.$$

如果用$KNAP(1,\ j,\ X)$来表示背包问题的一般化情形：

极大化：$\sum_{1\le i\le j} p_i x_i$

约束条件：$\sum_{1\le i\le j} w_i x_i \le X$，

$$x_i=0,\ 1,\ 1\le i\le j.$$

用$f_j(X)$表示$KNAP(1,\ j,\ X)$问题的最优值，则可得到

$$f_0(X)=\begin{cases}0, & X\ge 0\\ -\infty, & X<0\end{cases}, \qquad (4.8)$$

$$f_j(X)=\max(f_{j-1}(X),\ f_{j-1}(X-w_j)+p_j),\ \ j\ge 1. \quad (4.9)$$

最终解：

$$f_n(M)=\max\{f_{n-1}(M),\ f_{n-1}(M-w_n)+p_n\}$$

从递归式可知：$f_j(X)$单调非减的阶梯函数，它的值完全由一些数对$(w^{(i)},\ p^{(i)}),\ 1\le i\le t_j$所决定. 其中,

$$w^{(i)}<w^{(i+1)},\ \ p^{(i)}<p^{(i+1)}.$$

即 $f_j(X)$可表示为

$$f_j(X)=\begin{cases}-\infty,\ X<0\\ p^{(i)},\ w^{(i)}\le X<w^{(i+1)},\ 1\le i\le t_j\end{cases}\qquad(4.10)$$

其中

$$(w^{(1)},\ p^{(1)})=(0,\ 0),$$
$$(w^{(t_j)},\ p^{(t_j)})=(w_1+\cdots+w_j,\ p_1+\cdots+p_j),$$
$$w^{(t_j+1)}=+\infty.$$

将这些数对的集合记为

$$S^j=\{(w^{(i)},\ p^{(i)}):\ 1\le i\le t_j\}.$$

所以，要求 $f_j(X)$归结为求S^j，但我们的最终目的是求 $f_n(M)$，因此，只需求出使$w^{(i)}\le M$ 的那些数对$(w^{(i)},\ p^{(i)})$，仍以S^j表示这些数对的集合.

那么如何求S^j呢？

当 $j=0$时，$S^0=\{(0,0)\}$；当S^{j-1}确定后，求S^j的过程如下：由递归式知，S^j产生于S^{j-1}和对应于阶梯函数 $f_{j-1}(X-w_j)+p_j$的数对集

$$S_1^{j-1}=\{(w^{(i)}+w_j,\ p^{(i)}+p_j):\ (w^{(i)},\ p^{(i)})\in S^{j-1},\ w^{(i)}+w_j\le M\}$$

的归并. 归并就是在中舍弃掉一些数对，舍弃的规则如下：若 $S^{j-1} \cup S_1^{j-1}$ 中有两数对 $(w^{(k)}, p^{(k)})$，$(w^{(l)}, p^{(l)})$ 使得 $w^{(k)} > w^{(l)}$ 但 $p^{(k)} \le p^{(l)}$, 则弃掉 $(w^{(k)}, p^{(k)})$.直到不再有这样的两数对为止，就得到了 S^j.

得到 S^{n-1} 之后，只需在 S^{n-1} 和 S_1^{n-1} 中分别挑取数对 $(w^{(r)}, p^{(r)})$ 和 $(w_1^{(s)}, p_1^{(s)})$，使 r 和 s 分别为使 $w^{(r)} \le M$，$w_1^{(s)} \le M$ 的最大指标。若 $p^{(r)} > p_1^{(s)}$, 则 $f_n(M) = p^{(r)}$, $x_n = 0$; 否则 $f_n(M) = p_1^{(s)}$, $x_n = 1$. 再经过一回溯过程，可得到最优解的其它决策 $x_{n-1}, x_{n-2}, \cdots, x_1$. 回溯过程可简述如下：设 $(W_{(n)}, P_{(n)}) \in S^n$ 是满足 $M \ge W_{(n)}$，$f_n(M) = P_{(n)}$ 的数对. 若 $(W_{(n)}, P_{(n)}) \in S^{n-1}$, 则 $x_n = 0$, 令 $(W_{(n-1)}, P_{(n-1)}) = (W_{(n)}, P_{(n)})$; 若 $(W_{(n)}, P_{(n)}) \in S_1^{n-1}$，则 $x_n = 1$，令 $(W_{(n-1)}, P_{(n-1)}) = (W_{(n)} - w_n, P_{(n)} - p_n)$. 一般地，若 $(W_{(j)}, P_{(j)}) \in S^{j-1}$, 则 $x_j = 0$, 令 $(W_{(j-1)}, P_{(j-1)}) = (W_{(j)}, P_{(j)})$; 若 $(W_{(j)}, P_{(j)}) \in S_1^{j-1}$，则 $x_j = 1$，令 $(W_{(j-1)}, P_{(j-1)}) = (W_{(j)} - w_j, P_{(j)} - p_j)$ 将此过程继续下去，直到得到 x_1 的值为止.

例 $n = 4$，(w_i, p_i)（$1 \le i \le 4$）分别是 $(2,10)$，$(5,7)$，$(3,4)$，$(6,12)$，$M = 12$。

解：S^0：$(0,0)$；

S_1^0：$(2,10)$；

S^1：$(0,0)$，$(2,10)$；

S_1^1：$(5,7)$，$(7,17)$；

S^2：$(0,0)$，$(2,10)$，$(7,17)$；

S_1^2：$(3,4)$，$(5,14)$，$(10,21)$；

S^3：$(0,0)$，$(2,10)$，$(5,14)$，$(7,17)$，$(10,21)$；

S_1^3：$(6,12)$，$(8,22)$，$(11,26)$,$(13,29)$,$(16,33)$;

S^4: $(0,0)$,$(2,10)$,$(5,14)$,$(7,17)$,$(8,22)$,$(11,26)$,
$(13,29)$,$(16,33)$.

最优值：26，$x_4=1$，$x_3=1$，$x_2=0$，$x_1=1$。

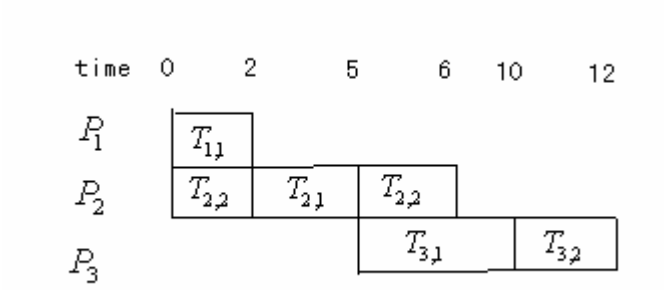
5.流水线调度问题(Flow Shop Scheduling)

处理一个作业通常需要执行若干个不同的任务。对于在多道程序环境中运行的一些计算机程序，也需要执行输入，运算，然后排队打印输出等任务。假设在一条流水线上有n个作业，每个作业i要求执行m个任务：$T_{1,i}$，$T_{2,i}$,…，$T_{m,i}$，$1\le i\le n$。并且任务$T_{j,i}$只能在设备P_j上执行，$1\le j\le m$。除此之外，还假定对于任一作业i，在任务$T_{j-1,\,i}$没完成以前，不能对任务$T_{j,\,i}$开始处理，而且同一台设备在任何时刻不能同时处理一个以上的任务。如果假设完成任务$T_{j,\,i}$所要求的时间是$t_{j,\,i}$，$1\le j\le m$，$1\le i\le n$，那么如何将这$n\times m$个任务分配给这m台设备，才能使这n个作业在以上要求下顺利完成呢？这就是流水线作业调度问题。

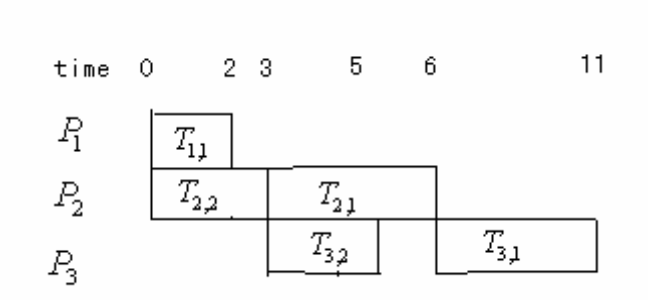
例 4.3 考虑在三台设备上调度两个作业，每个作业包含三项任务，完成这些任务要求的时间由矩阵$J=(t_{j,i})_{3\times 2}$给出，具体写出为

$$J=\begin{pmatrix}2 & 0\\ 3 & 3\\ 5 & 2\end{pmatrix}$$

这两个作业的两种可能调度如图 4.2 所示



(a)



(b)

图 4.2

流水线上的作业调度，有两种基本方式：一种是非抢先调度，它要求在任何一台设备上处理一个任务一直到完成才能处理另一任务。另一种是抢先调度，它没有以上要求，图 4.2(a)表示一种抢先调度，而图 4.2(b)表示一种非抢先点调度。作业i的完成时间$f_i(S)$是在S调度方案下作业i的所有任务得以完成的时间。在图 4.2(a)中$f_1(S)=10$，$f_2(S)=12$。在图 4.2(b)中，$f_1(S)=11$，$f_2(S)=5$。调度S的完成时间$F(S)$由下式给出：

$$F(S)=\max_{1\le i\le n}\{f_i(S)\} \qquad (4.11)$$

平均完成时间$MFT(S)$定义为

$$MFT(S)=\frac{1}{n}\sum_{1\le i\le n} f_i(S) \qquad (4.12)$$

一组给定作业的最优完成时间OFT调度是指一种非抢先调度S，在所有非抢先调度中，它对应的$F(S)$值取最小。可以很容易地给出抢先最优完成时间$POFT$，最优平均完成时间$OMFT$和抢先最优平均完成时间$POMFT$等调度的相应定义。

当$m>2$时，得到OFT和$POFT$调度的一般问题是难于计算的问题，得到$OMFT$调度的一般问题也是难于计算的问题。本节只讨论当$m=2$时获取OFT调度这种特殊情况的算法。

为方便起见，用a_i表示$t_{1,i}$，b_i表示$t_{2,i}$。在只有两台设备的情况下容易证明，在两台设备上处理的任务若不按相同的作业排列次序处理，则在调度完成时间上不比按相同次序处理花费时间少（注意：若$m>2$则不然）。具体说来，设n个作业在设备P_1上按排列$I:i_1i_2\cdots i_n$执行，在设备P_2上按排列$J:j_1j_2\cdots j_n$执行，完成所有任务花费的时间记为$f_{I,J}$，则有：

$$f_{I,I}\le f_{I,J} \qquad (4.13)$$

因此，设计调度方案时，只需考虑这些作业的某个排列次序，使得这些作业在每台设备上都按该次序执行。调度方案的好坏完全取决于这些作业的排列次序。

对于这n个作业的调度排列可看作n个阶段的决策过程。我们下面将会看到，对每个阶段可以定义合适的状态表示，使得在任一阶段来看，过程以后的行为仅依赖于该阶段的状态。假设对作业$1,2,\cdots,n$的一种调度排列为I：$i_1 i_2 \cdots i_n$。对于这个调度排列，记$f_{I,k}^1$和$f_{I,k}^2$分别是在设备P_1和P_2上完成任务$(T_{1,i_1},T_{1,i_2},\cdots,T_{1,i_k})$和$(T_{2,i_1},T_{2,i_2},\cdots,T_{2,i_k})(1\le k\le n)$的时间；又设$t_{I,k}=f_{I,k}^2-f_{I,k}^1$，则这两台设备在调度排列$I$下完成这$n$个作业所花费的时间为$f_{I,n}^2$。注意到

(1) $f_{I,n}^2=f_{I,n}^1+t_{I,n}$

和

(2) $f_{I,n}^1=\sum_{i=1}^{n}a_i$

是常数，调度排列I下完成这n个作业所花费的时间完全由$t_{I,n}$所确定。而这两台设备在执行完前k个作业后对$t_{I,n}$的影响完全可以用$t_{I,k}$来表示。

状态为$t_{I,k}$表示当设备P_1处理完前k个作业的任务之后，设备P_2还需花费$t_{I,k}$的时间才能完成前k个作业的任务。可等价地认为，当设备P_1可用来处理后面的任务时，设备P_2还需等待$t_{I,k}$的时间才能处理后面的任务。设在状态为t的情况下，处理完某作业i的状态为$t_{i\rightarrow}$，则有

$$t_{i\rightarrow}=b_i+\max(t-a_i,0). \qquad (4.14)$$

这是因为，以设备P_1可用来处理作业i之时刻算起，处理完作业i的

第一个任务的时间为a_i，处理完作业i的第二个任务的时间为$\max(a_i,t)+b_i$，两者相减即得(4.14). 显然，t增大，$t_{i\rightarrow}$亦随之增大。这说明在任何阶段，t越小意味着状态越优。

为了找出最优排列，我们来作如下考察。在任何调度排列I：$i_1\cdots i_k iji_{k+3}\cdots i_n$中，交换相邻的两个作业$i$，$j$，得到另外一个调度排列$\tilde{I}$：$i_1\cdots i_k jii_{k+3}\cdots i_n$。为了使$t_{I,n}\le t_{\tilde{I},n}$，即要使$t_{I,k+2}\le t_{\tilde{I},k+2}$，来看需满足什么条件。注意到$t_{I,k}=t_{\tilde{I},k}$，记这个值为$t$，则根据(4.14)得

$$t_{I,k+1}=b_i+\max(t-a_i,0),$$

$$\begin{aligned}t_{I,k+2}&=b_j+\max(t_{I,k+1}-a_j,0)\\&=b_j+\max(b_i+\max(t-a_i,0)-a_j,0)\\&=b_j+b_i-a_j+\max(\max(t-a_i,0),a_j-b_i)\\&=b_j+b_i-a_j+\max(t-a_i,0,a_j-b_i)\\&=b_j+b_i-a_j-a_i+\max(t,a_i,a_i+a_j-b_i)\end{aligned}$$

同理，

$$t_{\tilde{I},k+2}=b_j+b_i-a_j-a_i+\max(t,a_j,a_j+a_i-b_j)$$

要使$t_{I,k+2}\le t_{\tilde{I},k+2}$，即要使

$$\max(t,a_i,a_i+a_j-b_i)\le\max(t,a_j,a_j+a_i-b_j)\qquad(4.15)$$

为使(4.15)不依赖于t而成立，只需使

$$\max(a_i,a_i+a_j-b_i)\le\max(a_j,a_j+a_i-b_j)$$

即

$$a_i+a_j+\max(-a_j,-b_i)\le a_i+a_j+\max(-a_i,-b_j)$$

即

$$\min(a_j, b_i) \geq \min(a_i, b_j) \qquad (4.16)$$

等价地

$$a_i \text{或} b_j = \min(a_i, a_j, b_i, b_j) \qquad (4.17)$$

由式(4.16)知，在最优排列中，对每个相邻的作业对(i, j)，都有$\min(a_j, b_i) \geq \min(a_i, b_j)$。

但反过来，满足这一性质的排列未必是最优排列。例如，$n=5$, 作业$1,2,3,4,5$对应的参数分别为 $(7,4)$，$(2,2)$，$(9,5)$，$(3,3)$，$(13,6)$。排列I_1：$1,2,3,4,5$满足(4.16), 此排列下的完成时间是40；排列I_2：$2,4,5,3,1$下的完成时间是38。这说明排列I_1不是最优排列。不过我们可以依据下面的事实得到最优排列。如果$\min\{a_1, a_2, \cdots, a_n, b_1, b_2, \cdots, b_n\}$是$a_i$，那么将任何排列中的作业$i$移到第一个位置，完成时间将不会增加，说明最优排列可以在第一个作业为i的排列中产生；如果$\min\{a_1, a_2, \cdots, a_n, b_1, b_2, \cdots, b_n\}$是$b_j$，那么将任何排列中的作业$j$移到最后一个位置，完成时间将不会增加，说明最优排列可以在最后一个作业为i的排列中产生；接着在限定的排列类中对其他$n-1$个作业作类似操作，直到最后得到一个排列，必是最优排列。基于此，下面算法将得到一个最优排列：

step1: 把集合$\{a_1, a_2, \cdots, a_n, b_1, b_2, \cdots, b_n\}$排成非降序列；

step2: 以正反两个指针的移动来形成这n个作业的一个排列。起初，正指针指向第一个位置，反指针指向第n个位置(最后一个位置)。来考察已得到的非降序列：如果序列中下一个数是a_i，且作业i还没被

调度，则将作业i放入正指针当前指向的位置，正指针移向下一个位置；如果序列中下一个数是b_j，且作业j还没被调度，则将作业j放入反指针当前指向的位置，反指针移向前一个位置；如果作业i或作业j已被调度，则转到序列的下一个数。

上例中的排列I_2即是该算法得到的排列，是最优排列，另外须指出，不同的非降序列可能导致不同的最优序列。下列排列即是算法针对上例得到的不同最优序列：

I_3: 4, 5, 3, 1, 2; I_4: 2, 5, 3, 1, 4; I_5: 5, 3, 1, 4, 2;

6. 最长非递减子序列

问题： 给定某有序集上的序列$\{x_i\}_{i=1}^n$，求其最长非递减子序列？

可用如下动态规划法求解。设以x_i（$1 \le i \le n$）为最后一项的最长非递减子序列的长度为l_i，则l_i满足如下递归式：

$$l_i = \max_{0 \le k < i, x_k \le x_i} l_k + 1, \quad i \ge 1, \tag{4.18}$$

其中， 设置$l_0 = 0$， $x_0 = \min\_\text{element}$。最长非递减子序列的长度$l$满足$l = \max_{1 \le i \le n} l_i$。

算法 4.6 动态规划求解最长非递减子序列

Procedure $\text{LSS}(X(1:n), n, m, Y(1:n))$

//序列$X(1:n)$最长非递减子序列长度为m，存放在$Y(1:m)$中。//

Integer $L(1:n)$; $I(1:n)$; i; k; $last$

for $i \leftarrow 1$ to n do

$I(i) \leftarrow 0$

repeat

//$I(i)$表示以$X(i)$为最后一项的最长非递减子序列的倒数第二项在数组$X(1:n)$中的下标//

$L(1) \leftarrow 1;$

for $i \leftarrow 2$ to n do

$L(i) \leftarrow 0$

* for $k \leftarrow 1$ to $i-1$ do

* if ($X(k) \le X(i) \& L(k) > L(i)$)

* then $L(i) \leftarrow L(k);\ I(i) \leftarrow k$

* endif

* repeat

$L(i) \leftarrow L(i)+1$

repeat

$m \leftarrow 1;\ last \leftarrow 1$

for $i \leftarrow 2$ to n do

if ($L(i) > m$) then $m \leftarrow L(i);\ last \leftarrow i$

endif

repeat

$Y(m) \leftarrow X(last);\ i \leftarrow m-1$

While $i \ge 1$ do

$Y(i) \leftarrow X(I(last))$

$i \leftarrow i-1;\ last \leftarrow I(last)$

repeat

Print(“m”; “$Y(1:m)$”)

end LSS

算法的最坏时间复杂度为$\Theta(n^2)$。

事实上，可以改进算法，使最坏时间复杂度降低为$\Theta(n\log n)$。思路是：在上述算法中，特别考虑长度为l（$1\le l\le m$，m是最长非递减子序列的长度）的非递减子序列的最小最后项$Z(l)$，显然有$Z(1)\le Z(2)\le\cdots\le Z(m)$。这样做的结果可使以”*”标记的循环的时间复杂度从$\Theta(i)$降低为$\Theta(\log i)$。

算法 4.7 改进的动态规划求解最长非递减子序列

Procedure ILSS($X(1:n),n,m,Y(1:n)$)

//序列$X(1:n)$最长非递减子序列长度为m，存放在$Y(1:m)$中。//

Integer $L(1:n)$; $I(1:n)$;$\min(1:n)$; i; k;$last$

for $i\leftarrow 1$ to n do

$I(i)\leftarrow 0$

repeat

//$I(i)$表示以$X(i)$为最后一项的最长非递减子序列的倒数第二项在数组$X(1:n)$中的下标//

$L(1)\leftarrow 1$; $m\leftarrow 1$; $\min(1)\leftarrow 1$

//$\min(l)$表示长度为l的非递减子序列的最小最后项在数组$X(1:n)$中的下标//

for $i\leftarrow 2$ to n do

Case

$: X(i) < X(\min(1)):\ L(i) \leftarrow 1;\ \min(1) \leftarrow i$

$: X(i) \geq X(\min(m)):$

$L(i) \leftarrow m+1; I(i) \leftarrow \min(m)$

$m \leftarrow m+1;\ \min(m) \leftarrow i$

:else: $low \leftarrow 1;\ high \leftarrow m$

While($high - low \geq 2$) do

$$mid = \left\lfloor \frac{high + low}{2} \right\rfloor$$

if $X(i) \geq X(\min(mid))$ then $low \leftarrow mid$

else $high \leftarrow mid$

endif

repeat

$L(i) \leftarrow high;\ I(i) \leftarrow \min(low);\ \min(high) \leftarrow i$

endcase

repeat

$last \leftarrow \min(m)$

$Y(m) \leftarrow X(last);\ i \leftarrow m-1$

While $i \geq 1$ do

$Y(i) \leftarrow X(I(last))$

$i \leftarrow i-1;\ last \leftarrow I(last)$

repeat

Print(“m”; “$Y(1:m)$”)

end ILSS

算法 4.7 的最坏时间复杂度为$\Theta(n\log n)$。

作业

1. 假定两个仓库W_1和W_2都存有同一种货物，其库存量分别为r_1和r_2，现要将其全部发往n个目的地$D_1, \cdots, D_n$，设发往D_j的货物量为d_j，则$r_1+r_2=\sum_{j=1}^{n} d_j$. 如果由仓库$W_i$发送量为$x_{ij}$的货物到目的地$D_j$的花费为$c_{ij}(x_{ij})$，那么仓库问题就是求两个仓库应给每个目的地分别发多少货量才会使总的花费最小，即要求出这些非负整数x_{ij}，$1\le i\le 2$，$1\le j\le n$,使得$x_{1j}+x_{2j}=d_j$，$1\le j\le n$ ，并且使$\sum_{\substack{i=1,2\\ 1\le j\le n}} c_{ij}(x_{ij})$取最小值. 假设当$w_1$有$x$的库存量且按最优方式全部发往目的地$D_1, \cdots, D_i$时所需的化费为$g_i(x)$ (此时W_2的库存为$\sum_{1\le j\le i} d_j - x$), 于是此仓库问题的最优总花费是$g_n(r_1)$.

(1).求$g_i(x)$的递推关系式；

(2).写一个算法求解这个递推关系式并要能得到x_{ij} ($1\le i\le 2$, $1\le j\le n$)的决策值的最优序列.

2. (最长公共子序列) 求序列$\{x_i\}_{i=1}^{n}$与序列$\{y_i\}_{i=1}^{m}$的最长公共子序列。

3. 有两台机器A，B和n个要处理的作业。每台机器都能单独处理

这n个作业，但不能把一个作业分由两台机器处理，而且每台机器都不能同时处理两个作业。另外，由于作业和机器特性的缘故，用机器A处理作业i（$1 \le i \le n$）与用机器B需要的时间未必相同，分别记作a_i，b_i。请写出能确定对这个作业所需最少处理时间的动态规划递推关系式，并举例说明如何求解得到的递推式。

4.（装配线调度问题）汽车装配厂负责把半成品汽车加工成品，共需n道工序. 该厂有三条独立的流水线，每条流水线上相应地设置n个独立的机床，第i（$1 \le i \le n$）个机床对应执行第i道工序.三条流水线上同样编号的机床进行的操作完全一样，但由于机床的制造商和年份不同，所花费的操作时间不尽相同,第l $(1 \le l \le 3)$条流水线上第i道工序花费的时间为$e_{l,i}$.通常每辆汽车的装配总是在某条流水线l上完成,花费的总时间是

$$\sum_{i=}^{n} e_{l,i} + x_l + y_l$$

其中x_l， y_l分别是半成品汽车输入到流水线l和成品汽车从流水线l输出花费的时间。但有时急需装配某辆汽车，要求越快越好，每道工序都允许从三个不同流水线的机床中选择一个来执行。如果第i道工序在流水线l_1上执行，而第$i+1$道工序在流水线l_2上执行，则汽车从流水线l_1移动到流水线l_2花费的时间为$m_{l_1,l_2}^{(i)}$. 在此情况下，如何调度来装配一辆汽车花费的总时间最少呢？请编制算法求解该问题，限定算法时间复杂度为$O(n)$.

5. 给定实数列$\{x_i\}_{i=1}^n$，要确定$1\le i\le j\le n$，使得$\sum_{k=i}^{j}x_k$最小。请编制时间复杂度为$O(n)$的算法求解该问题。

6. n个活动请求安排在某个多功能厅举行，第i $(1\le i\le n)$个活动进行的时间段是$[s_i, f_i]$，如果得到安排，多功能厅可获益p_i元. 该多功能厅任何时间都不能同时举办两个活动，所以这n个活动可能会发生时间冲突，不能都一一安排在该多功能厅。现在自然面临这样一个问题：多功能厅能够安排其中哪些活动，使之不发生冲突而且获益最多？请给出求解该问题的算法。

7.（套汇问题）套汇是指利用货币兑换率的差异把一个单位的某种货币转换为大于一个单位的同种货币的操作。例如，假定 1 美元可以买 8 个人民币，1 人民币可以买 15 日元，1 日元可以买 0.009 美元， 一个商人可以通过下列方式兑换货币：美元换人民币， 人民币换日元，日元换美元，从 1 美元得到 1.08 美元。 假定有n种货币以及相应的兑换表$R(1:n,1:n)$， 使得 1 单位的货币i可以买$R(i,j)$个单位的货币j。试设计一有效算法，确定是否存在货币序列$i_1,\cdots,i_s$，满足$R(i_1,i_2)\bullet R(i_2,i_3)\bullet\cdots\bullet R(i_{s-1},i_s)>1$，如果存在的话，要求输出该序列。并请分析你所设计的算法的运行时间。

第五章 图问题与基本周游方法

主要参考：

14. Sara Baase, Allen Van Gelder, “Computer Algorithms: Introduction to Design and Analysis (Third Edition)”, Higher Education Press & Pearson Education Asia Limited., 2001. ***Chapter 7***

1．引言

很多问题可归结为关于图的问题。这些问题不只是根源于计算机领域，也贯穿了各个学科与工商业界。针对许多图问题已发展出高效算法，这极大地提高了人们解决实际问题的能力。然而，还有许多重要的图问题，迄今仍无有效方法求解。对于另外一些问题，则拿不准现存的求解方法已经是最有效的还是可有进一步改进的余地。

在本章，我们首先回顾图的定义和一些基本性质，然后详述有效周游图的基本方法，即深度优先检索法和宽度优先检索法。以这两种检索法的算法为基础，可扩展出关于许多典型图问题的高效算法，所谓高效即在线性时间内可求解。粗略地说，这样的图问题称为“易解”的图问题，易解的意思不是说容易编制算法或程序，而是指编制的算法

可以非常高效地求解问题，在实践中可解决很大规模的实例，比方说，可处理有数百万结点的图。

依此粗略观点来看，中等难度图问题包括那些可在多项式时间内求解的图问题，比如最优检索树问题等，这些问题虽然不能在线性时间内求解，但它们的时间复杂度往往受限于n^2或n^3这样的渐近增长率，在实际中可解决成千上万之规模的实例，也可令人接受。“困难”的图问题，是没找到多项式时间算法的问题，这样的问题就实际应用来说几乎是不可求解的，因为很小规模的实例就可能耗费几年时间。就目前所知，看不到发现有效算法的希望，但还无人证明确实不存在多项式算法。它们被称为 NP 难度的问题。

图问题的一大诱人之处在于稍微改变问题的提法就会导致难度类型的变化。因此熟悉现存问题的难度及其原因将有助于处理碰到的新问题。

2．图的概念和数据结构表示

2.1 图的基本概念和术语

大凡牵涉到研究离散对象间某种结构的问题总能用图模型来表示。譬如，交通运输线路图，程序流程图，计算机网络，电子线路等等，对

应于这些结构的问题就转化为典型的图问题。

抽象地说，图为离散对象集合上的二元关系提供了一种描述方法。

图(graph)分为无向图(undirected graph)和有向图(directed graph, digraph)。任何图由二要素组成：结点集V和边集E。$G=(V,\ E)$。无向图与有向图的区别是，前者的边是无向边，后者的边是有向边。无向图表示了一种**对称的**二元关系，在大多数情况下可看成是一种特殊的有向图：**对称有向图**，它是将无向图的每条边换成两条方向相反的有向边得到的。

有限简单图：包含有限个结点，不含平行边和自环的图。结点集

$$V=\{v_1,v_2,\cdots,v_n\}$$

边集

$$E=\{e_1,e_2,\cdots,e_m\}$$

任何边可表示为$e_k=v_iv_j$，其中v_i与v_j是边e_k的端点，说v_i与v_j相邻，边e_k与v_i和v_j相关联；在有向图中，v_i是起点（tail），v_j是终点（head），说v_i是v_j的内邻点，v_j是v_i的外邻点。

图的其他重要概念：子图，路径，道路，回路，连通图(无向图)与**连通分支**，强连通图(有向图)与**强连通分支**，**反圈图**，**赋权图**。

连通关系：无向图G的结点集V上的一个二元关系，称结点u与v是连通的，如果它们之间存在道路。连通关系是等价关系。若无向图G的任何两个结点都是连通的，则称G是连通图。若G的子图G_1是连通图，则称G_1是G的连通子图。

连通分支：无向图的结点集上连通关系的等价类；也是无向图的极大连通子图。

强连通关系：有向图G的结点集V上的一个二元关系。称结点u与v是强连通的，如果G中既存在u到v的道路，又存在v到u的道路。强连通关系是等价关系。若有向图G的任何两个结点都是连通的，则称G是强连通图。 若G的子图G_1是强连通图，则称G_1是G的强连通子图。

强连通分支：有向图的结点集上强连通关系的等价类，是有向图的极大强连通子图。

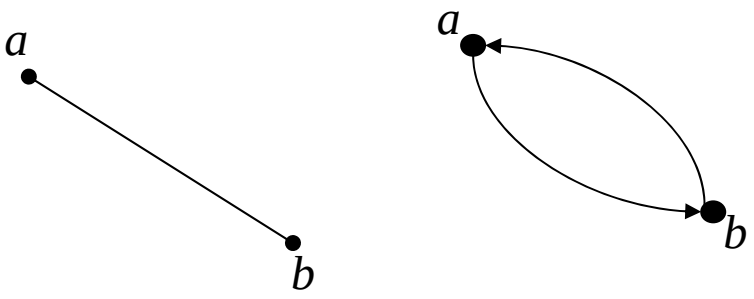
反圈图（Acyclic Graph）：不包含回路的图。

无向反圈图（Undirected Acyclic Graph）：称为无向森林（Undirected forest），如果还是连通的，称为无向树（Undirected tree）。

有向反圈图（Directed Acyclic Graph, 简记为 DAG）：不含回路的有向图。

转置图：设G是一个有向图，将它的每条边的方向改变为相反方向，就得到它的转置图G^t。

注：在涉及无向图的问题中，如果回路起重要作用，则不可将其看作对称有向图。譬如无向图ab没回路，但它对应的对称有向图有两条有向边：ab和ba，形成一个回路。



赋权图：三元组(V, E, W)，其中，W是定义在边集E上的实值（有些情况下，限制为整数、有理数是适当的）函数。权函数可有多种实际含义。

2.1 图的数据结构表示：

邻接矩阵

n阶图G可被如下定义的n阶方阵$A=(a_{ij})$所表示：

$$a_{ij}=\begin{cases}1 & \text{if } v_iv_j\in E\\ 0 & \text{else}\end{cases}\qquad 1\le i,j\le n$$

对无向图来说，邻接矩阵是对称的，只有上三角（或下三角）是必须存储的。如果$G=(V,E,W)$是赋权图，可以修改邻接矩阵使之表示出权来。

$$a_{ij}=\begin{cases}w(v_iv_j) & \text{if } v_iv_j\in E\\ c & \text{else}\end{cases}\qquad 1\le i,j\le n$$

其中c是常数，依赖于权的实际含义和待求问题的提法。比如说，如果权是路程花费，则可取$c=\infty$；如果权是网络线路的容量，则可取$c=0$.

有些问题需要处理每一条边，如果用邻接矩阵来表示图，需验证每一可能的边，可能的边数为$n(n-1)$（有向图）或$n(n-1)/2$（无向图），所以算法复杂度的下界是n^2。

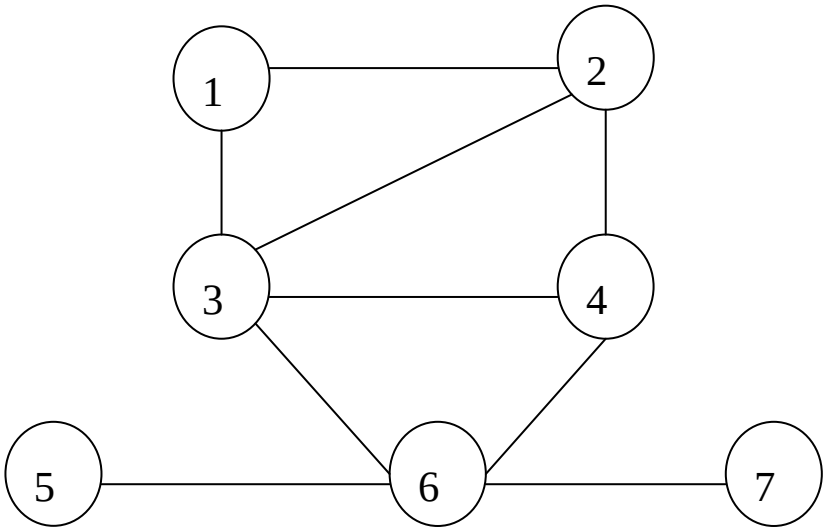
邻接表

n个链表，每个结点v_i对应一个链表i，表示结点v_i的所有外邻点与关联的边，每个链表有个头指针$H(i)$，它指向表示v_i的第一个外邻点的链结点，每个链结点包含若干个域，其中有两个域分别表示外邻

点和指向下一个外邻点的指针， 其他域可表示对应边的信息， 例如边的权值等。

邻接表是邻接矩阵的压缩表示，链结点的个数为图的边数m（有向图）或$2m$（无向图）。对很多问题的求解来说，可提高运算效率，有可能使算法时间复杂度不超过$O(n+m)$。

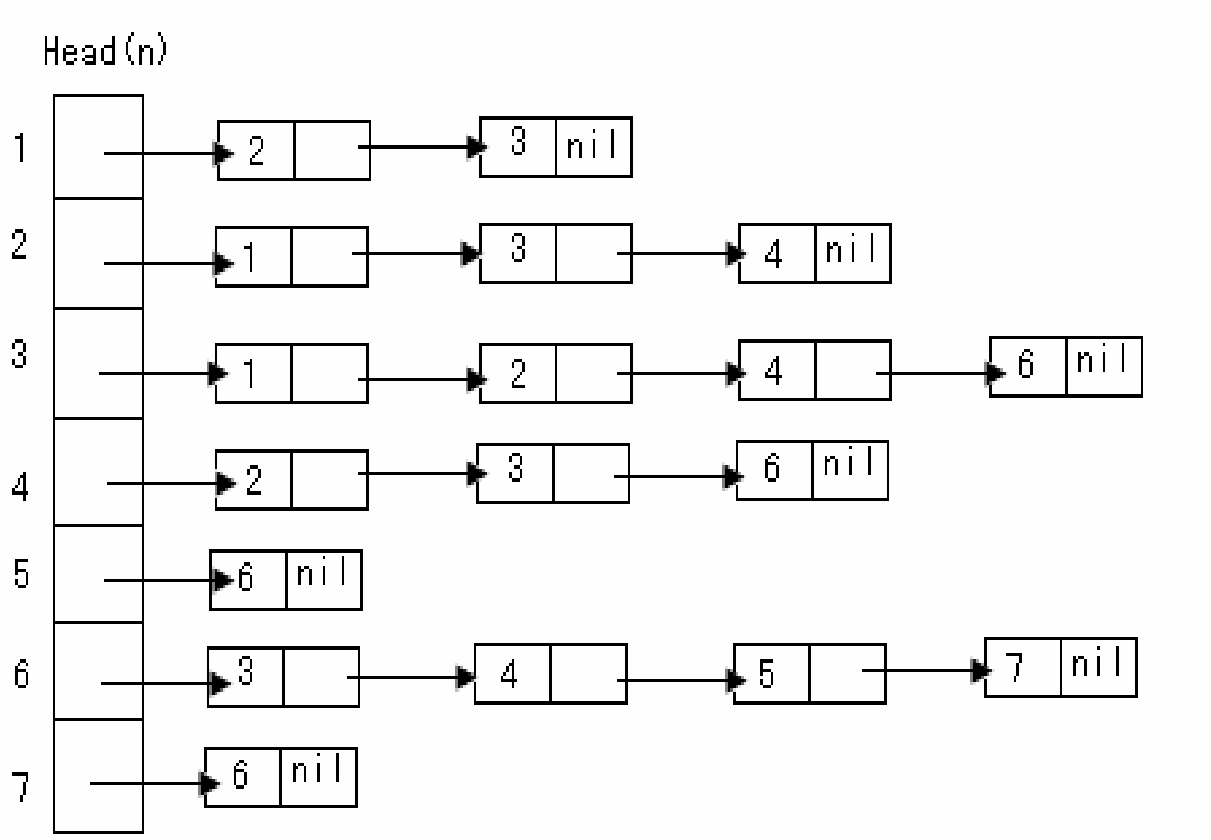
举例：



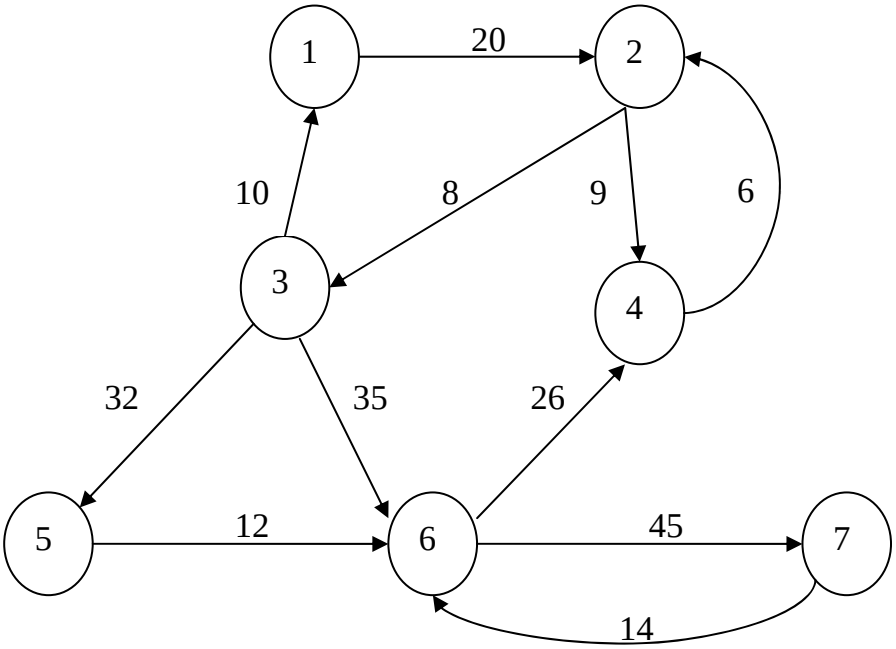
无向图示例

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

邻接矩阵



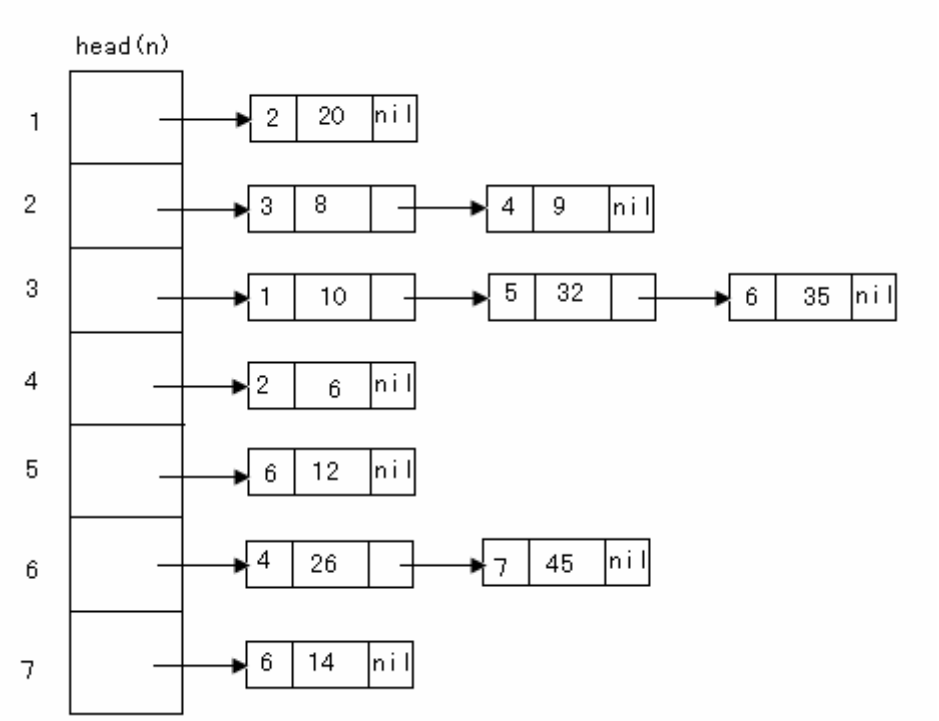
邻接表



赋权有向图示例

$$\begin{pmatrix} 0 & 20 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 8 & 9 & \infty & \infty & \infty \\ 10 & \infty & 0 & \infty & 32 & 35 & \infty \\ 6 & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & 12 & \infty \\ \infty & \infty & \infty & 26 & \infty & 0 & 45 \\ \infty & \infty & \infty & \infty & \infty & 14 & 0 \end{pmatrix}$$

赋权矩阵



赋权图邻接表

作业

请编制算法，从图G的邻接表得到它的转置图G^t的邻接表，要求时间复杂度为$O(n+m)$。假设结点用n个数字编号，每个链表中，链结点的顺序由所表示的外邻点的编号决定（从小到大）。

3．图的基本周游方法

许多图问题的求解需要检查或处理（访问）图的每个结点和每条边。宽度优先检索（breadth-first search, breadth-first traveral）和深度优先检索（depth-first search, depth-first traveral）是两种基本检索方法，可有效地访问图的每个结点和每条边(不重复)。不同的问题需

选择采用不同的检索方法求解，能得到线性复杂度($\Theta(n+m)$)的算法。

我们首先以有向图为例，介绍这两种检索方法。

3.1 深度优先检索概述

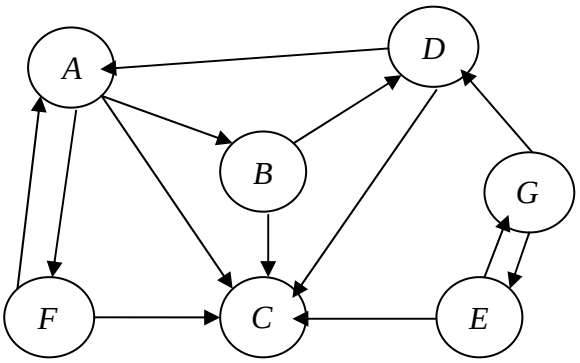
深度优先检索是树周游方法（先根周游，中根周游，后根周游）的推广，契合了递归算法的结构，所以它比宽度优先检索的应用范围更广，借以生成许多重要算法，后面几节会看到一些例子。

检索起点由问题本身决定或随意选取。深度优先检索可看作“旅游”某景点区。

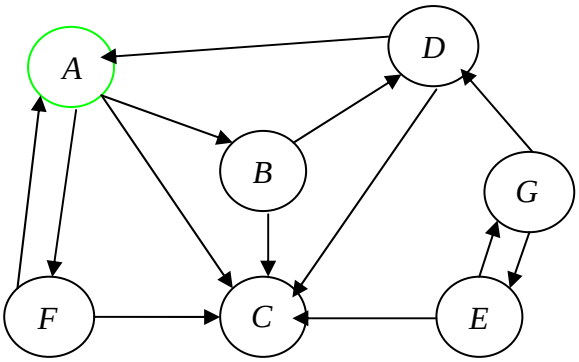
把有向图想象成以桥连接的丛岛。桥是单向交通道，但步行者不受此限，可来回走。让我们来周游此丛岛。我们采取的策略是，从某个岛出发，我们首次穿过某桥时，总是沿着交通方向走，这称为“探索（exploring）”一条边。所以当我们逆着交通方向通过某桥时，必然回到以前到过的地方， 这称为“回溯（backtracking）”。深度优先检索的要旨是尽可能地“探索”，“探索”不成才作“回溯”，以寻找其他“探索”的机会。直到“回溯”到起点，无法再探索为止。

术语:“探索(exploring)”,“回溯(backtracking)”,“发现（discover）”,“检查(checking)”,“未发现点（undiscovered）”,“活动点（active）”,“死点(dead end)”,“完成点(finished)”。

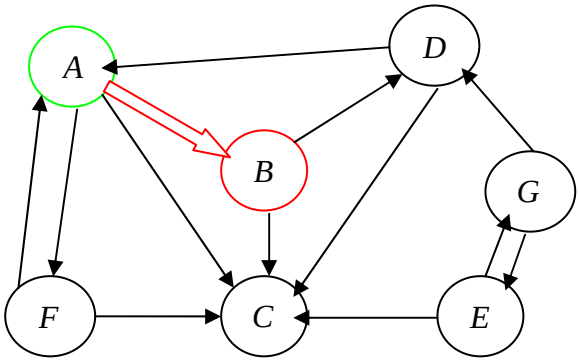
例.



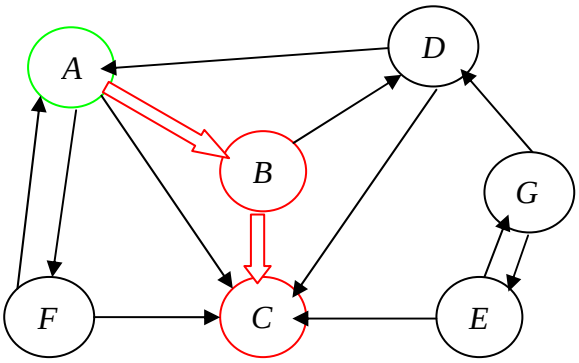
有向图示例



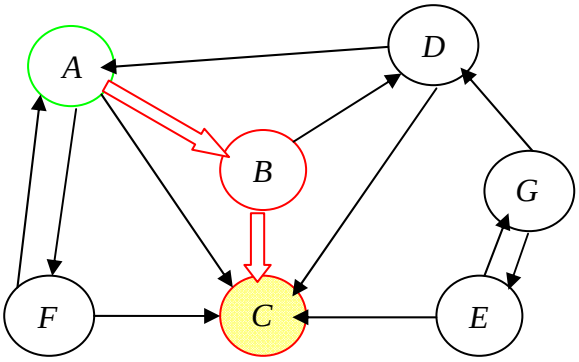
深度优先检索从结点 A 开始



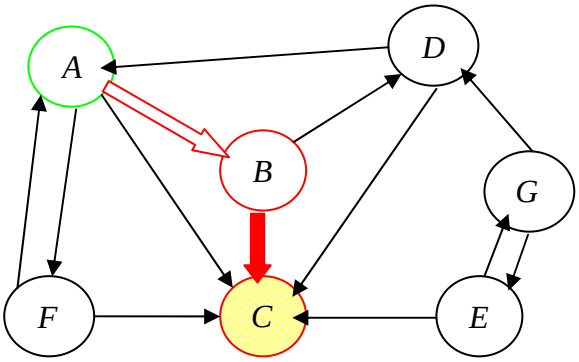
探索边 AB，发现结点 B



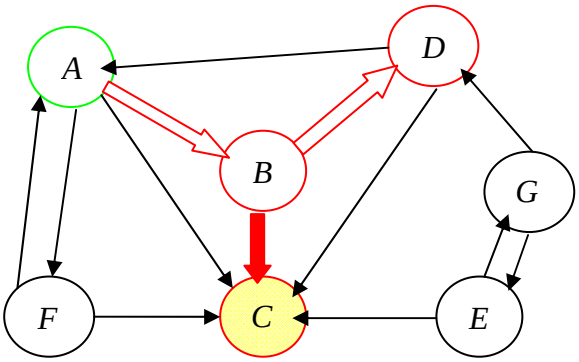
探索边 BC，发现结点 C



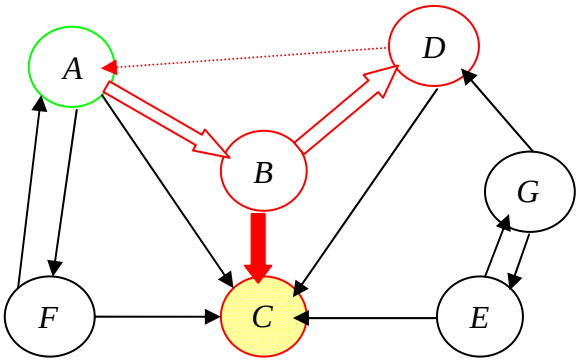
从结点 C 无路可走，称为“死角（dead end）”



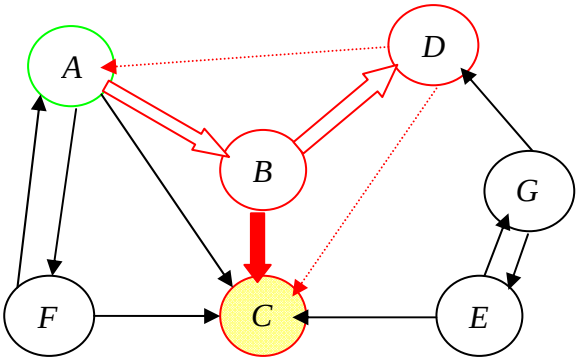
C 称为完成点（finished），从 C 回溯到 B。



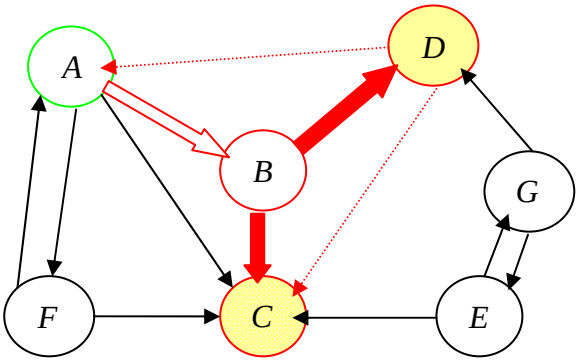
探索边 BD,　发现结点 D



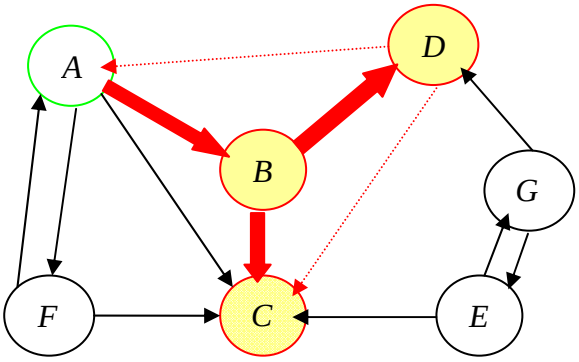
检查边 DA（A 是已发现的结点，　无需探索）



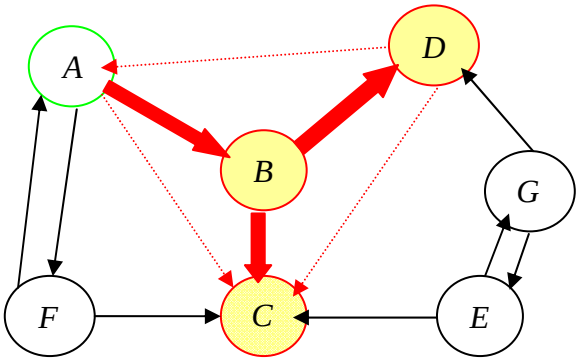
检查边 DC



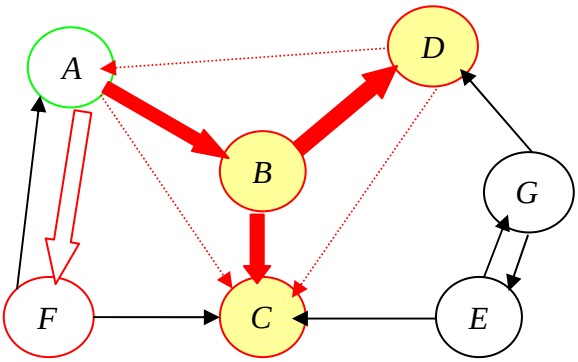
D 变成完成点，从 D 回溯到 B。



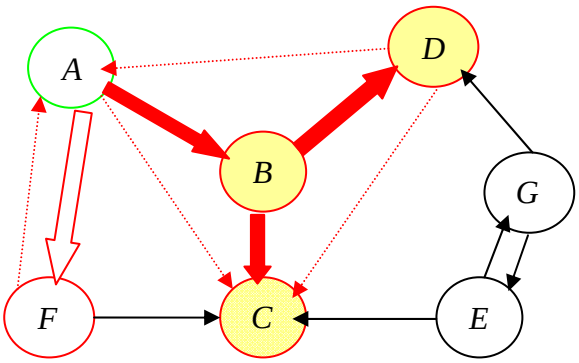
B 变成完成点，从 B 回溯到 A。



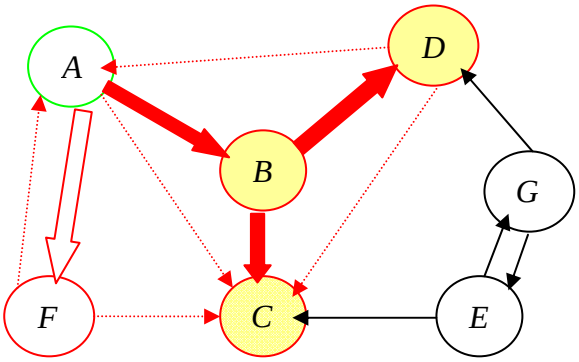
检查边 AC



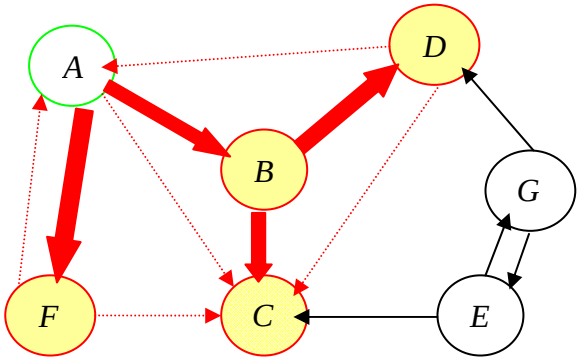
沿索边 AF，发现结点 F



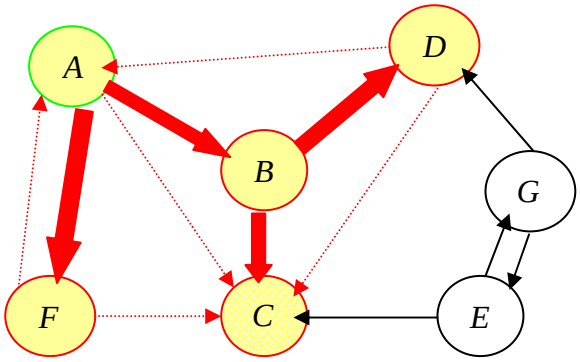
检查边 FA



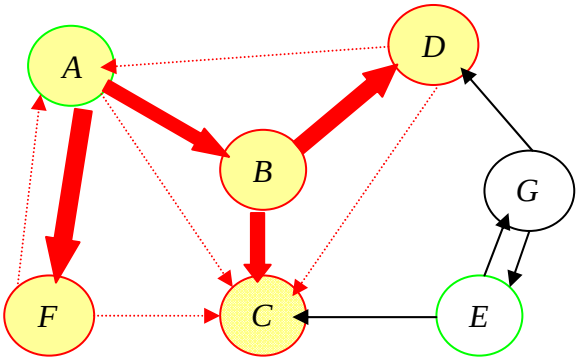
检查边 FC



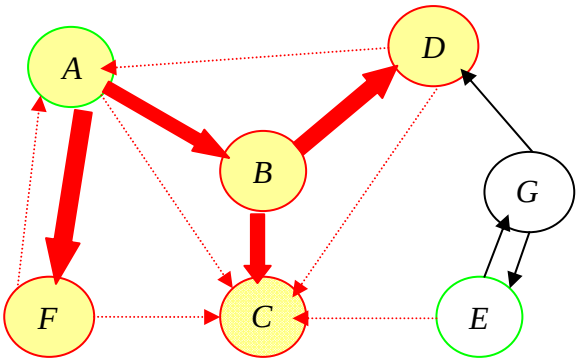
F 变成完成点，从 F 回溯到 A。



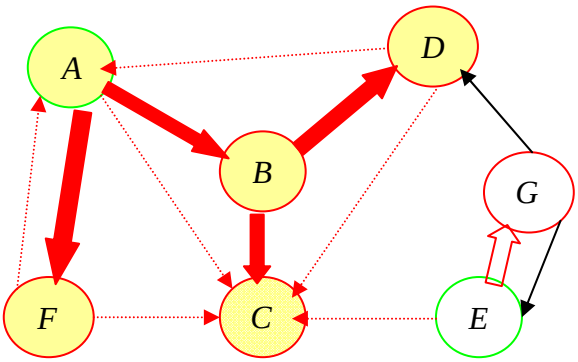
以 A 为起点的深度优先检索至此完成



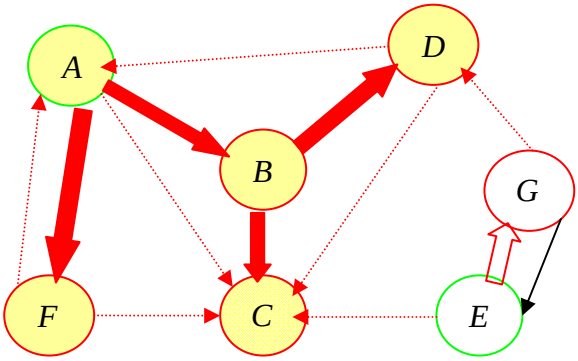
接着以 E 为起点进行深度优先检索



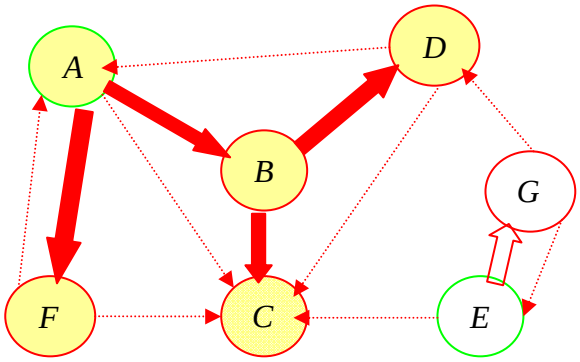
检查边 EC



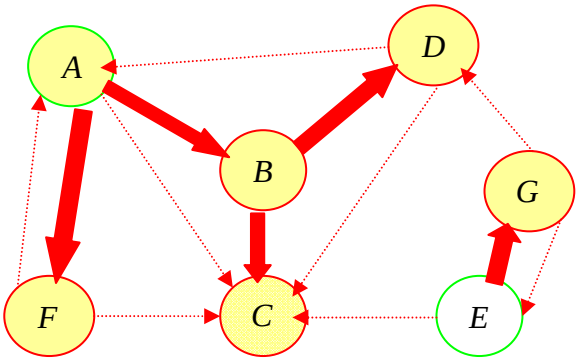
探索边 EG，发现结点 G



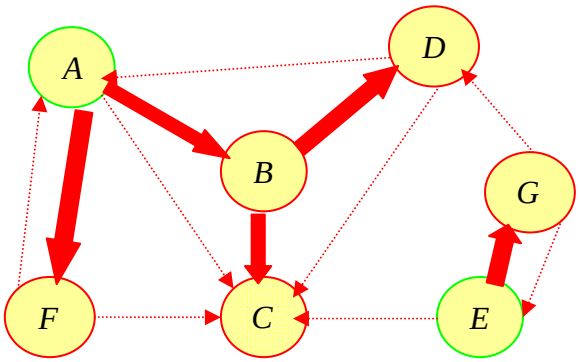
检查边 GD



检查边 GE



G 变成完成点，从 G 回溯到 E。



E 变成完成点。至此图的深度优先周游完成

深度优先检索树(depth-first spanning tree**)**：由起始点为树根，由探索边作为树枝形成。非树枝边有特殊的意义，往后有细致讨论。

深度优先检索和周游的概略描述

1．以v为起点的深度优先检索：

dfs(G, v)

Mark v as “discovered”.

For (each vertex w such that edge vw is in G) do

if (w is undiscovered) then

{dfs(G, w); that is, explore vw, visit w, explore from there as much as possible, and backtrack from w to v.}

Otherwise:

{ “Checking” vw without visiting w.}

endif

Mark v as “finished”.

repeat

end dfs(G, v)

2. 对图G的深度优先周游：

Procedure dfsSweep(G)

Initialize all vertices of G to “undiscovered ”.

For each vertex $v \in G$, in some order do

if (v is undiscovered) then

{ dfs(G, v) ; that is, perform a depth-first search beginning (and ending) at v; any vertices discovered during an earlier depth-first search visit are not revisited; all vertices visited during this dfs are now classified as “discovered”.}

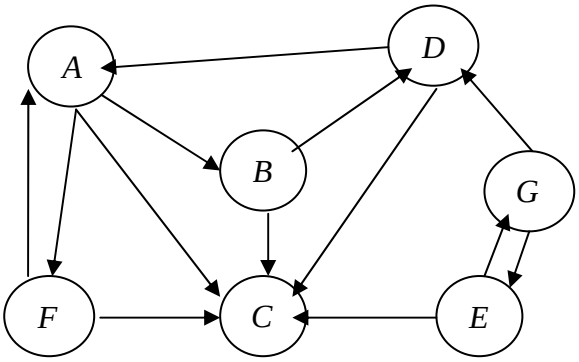
endif

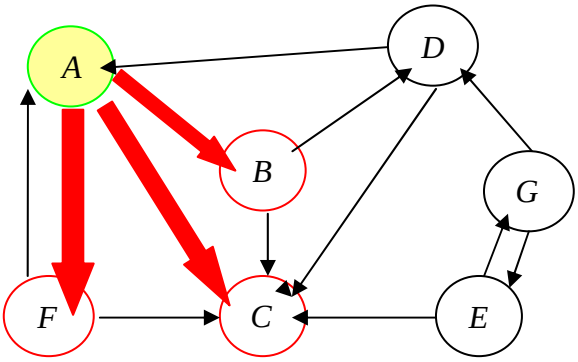
end dfsSweep

3.2 宽度优先检索概述

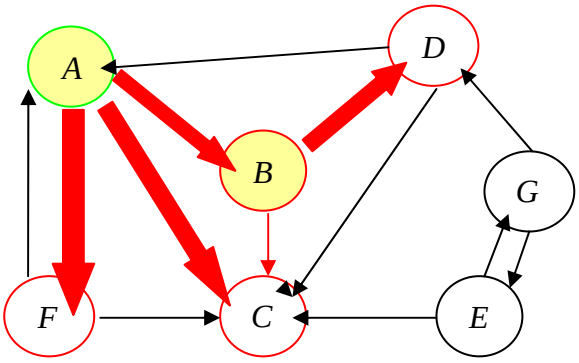
宽度优先检索可看作传染病的传播。所以从图的某个结点开始，首先传播到与它相邻的所有结点，传播途径是边。然后再从这些被传染的

结点出发，以同样的方式继续传播。

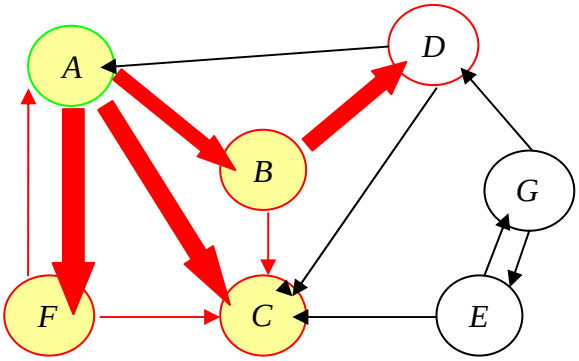




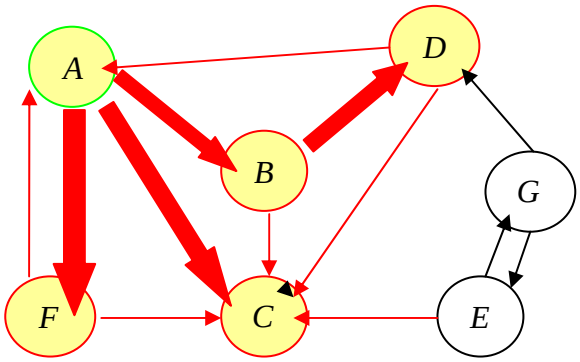
从结点 A 为起点分别沿边 AB, AC, AF 传染到结点 B，C，F。



从结点 B 检查边 BC（C 已被传染），沿边 BD 传染到结点 D。

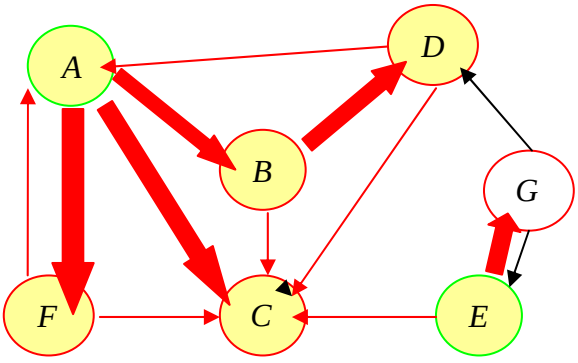


在结点 C 无传播路径。

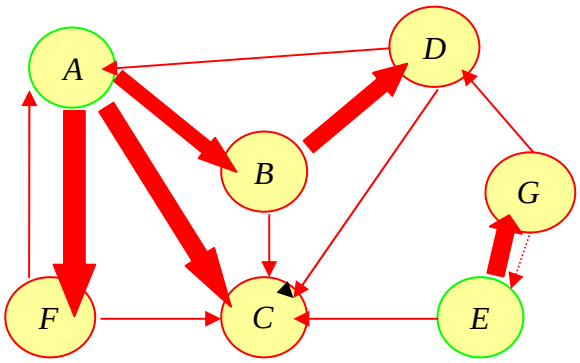


在结点 D 处，检查 DA，DC，以 A 为起点的宽度

优先检索至此完成。



从结点 E 处，检查边 EC， 沿边 EG 传染到结点 G



在结点 G 处检查边 GD，GE。至此整个图的宽度优先周游完成

宽度优先生成树(breadth-first spanning tree): 由起始点S为树根，传染边作为树枝形成。特征：在该树中根S到任何其他点V的路径是图G中S到V的最短路径。

算法 7.1 宽度优先检索

数据结构：

链表数组：$adj(n)$，

点状态数组：$color(n)$，

生成树结点的父亲数组：$parent(n)$

$\text{bfs}(n, adj, color, s, parent)$

Queue Q

$parent(s) \leftarrow -1$

$color(s) \leftarrow gray$

$\text{enqueue}(Q, s)$

while(Q is nonempty) do

$v \leftarrow \text{front}(Q)$

dequeue(Q) /delete v ///

for (each vertex w in the list $adj(v)$) do

if($color(w) == white$) then

{$color(w) \leftarrow gray$

enqueue(Q, w) //w enters in Q//

$parent(w) \leftarrow v$ //process tree edge vw//}

endif

repeat

$color(v) \leftarrow black$

repeat

end bfs

bfsSweep$(n, adj, color, parent)$

Initialize $color(1), color(2), \cdots, color(n)$ to $white$

for $v \leftarrow 1$ to n do

if ($color(v) == white$) then

call bfs$(n, adj, color, v, parent)$

endif

repeat

end bfsSweep

3.3 深度优先检索与宽度优先检索的比较

数据结构：深度优先检索：栈；宽度优先检索：队列。

生成树：宽度优先检索树的树枝有特征意义，非树枝无任何意义；而深度优先检索树的非树枝比树枝更有意义，这是后话，此处先指出这一点。

深度优先检索存在“后处理”决定了它的广泛应用，后面各节有较细致的介绍；而宽度优先检索的简单“一次性”处理过程相对来说只有较少的应用范围，典型的应用如找**无向图的连通分支**，**求解所有边权为单位值的赋权图的单源最短路径问题**。事实上，以对称有向图来表示无向图，算法 7.1 中的数组 $parent(n)$ 能够表示结点 s 到其他结点的最短路径。

在无向图中，任何一次宽度优先检索过程中检索到的所有结点构成一个连通分支，以起始点 v 作为这些结点的标记，就得到了无向图的所有连通分支

算法 7.2 利用宽度优先检索求无向图的连通分支（同样以对称有向图来表示无向图）

数据结构：

链表数组：$adj(n)$，

点状态数组：$color(n)$，

结点所在连通分支标记：$ConComp(n)$

bfs_con($n, adj, color, s, ConComp$)

Queue Q

$ConComp(s) \leftarrow s$

$color(s) \leftarrow gray$

enqueue(Q, s)

while(Q is nonempty) do

$v \leftarrow$ front(Q)

dequeue(Q) /delete v ///

for (each vertex w in the list $adj(v)$) do

if($color(w) == white$) then

{$color(w) \leftarrow gray$

enqueue(Q, w)

$ConComp(w) \leftarrow s$}

endif

repeat

repeat

end bfs_con

```
bfsSweep_con(n,adj,color,ConComp)
Initialize color(1),color(2),⋯,color(n) to white
for v←1 to n do
 if (color(v) == white) then
  call bfs_con(n,adj,color,v,ConComp)
 endif
repeat
end bfsSweep_con
```

4. 有向图上的深度优先检索

我们开始细致处理有向图上的深度优先检索过程。 我们首先描述一种一般化的深度优先检索框架，它可用来求解许多问题，我们将以几个典型问题来说明这一点。

探究无向图上的深度优先检索过程较之有向图更为复杂，这是因为在无向图的表示中，每条边的表示在所用数据结构中出现两次，我们必须对边的这两次出现加以区别对待。我们将在第 6 节单独论述。当然在有些情况下，可以将无向图当作对称有向图来处理，此时，可应用关于有向图的深度优先检索过程。从原理上说，**只要无向图上的深度优先检索可忽略非树枝边**，就可这样做。

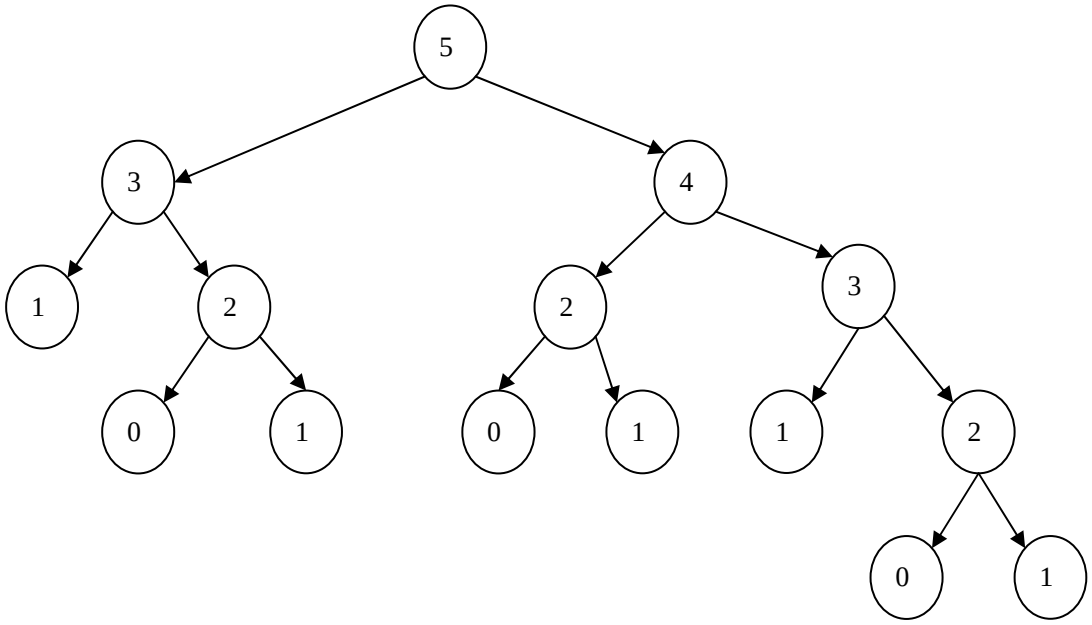
关于**转置图**：很多以有向图作为数学模型表示的问题中，采用边的两种方向都是可行的，比如说，有向边 “vw”表示父子关系的话，我们采用相反的方向就表示子父关系。哪种方向作表示更方便就采用哪种。这两个边方向相反的有向图互称为转置图。

深度优先检索与递归调用的关系：深度优先检索算法可简单地写成一个递归算法；而任何递归算法都可以表示成对一棵树的深度优先检索。以计算斐比那契(Fibonacci)数的递归算法：$F_0 = F_1 = 1$，$F_n = F_{n-2} + F_{n-1}$为例，取$n = 5$。

斐比那契数的直接递归算法：

```
Procedure Fibonacci(n, x)
// x表示待求的第n个斐比那契数//
Integer y, z
If(n = 0 or n = 1)then x ← 1
Else
  {
    Call Fibonacci(n − 2, y)
    Call Fibonacci(n − 1, z)
    x ← y + z
  }
Endif
```

End Fibonacci



(a) 第 5 个斐比那契数的递归计算调用结构图。标记数字i的结点表示求解第i个斐比那契数。

上述算法的时间复杂度$t(n)$满足下列递归式

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

容易算出$t(n)$是指数复杂度的。

当然这里出现了许多重复运算，把算法改写成关于一个有向图（这样的图总是有向反圈）的深度优先检索将更为有效。

斐比那契数的优化递归算法

Procedure OpFibonacci(n, x)

Integer $F(0:n)$// 全局数组//

Boolean $Q(0:n)$// 全局数组//

For $i \leftarrow 1$ to n do

$Q(i) \leftarrow false$

repeat

Call Fibonacci(n)

$x \leftarrow F(n)$

End OpFibonacci(n, x)

Procedure Fib(k)

If($k = 0$ or $k = 1$) then $F(k) \leftarrow 1$; $Q(k) \leftarrow true$

Else

{

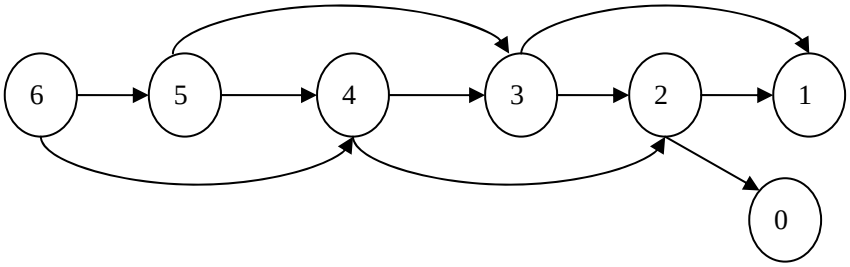
If ($Q(k-2) = false$) then call Fib($k-2$) endif

If ($Q(k-1) = false$) then call Fib($k-1$) endif

$F(k) \leftarrow F(k-2) + F(k-1)$; $Q(k) \leftarrow true$

}

Endif Fib



(b) 深度优先检索实际上是将中间运算结果存储起来，避免了重复运算。

去掉重复运算后，算法复杂度降低为线性的。再应用我们后面介绍的有向反圈的反拓扑序，算法可进一步改进为非递归算法。

Procedure SimFib(n, x)

Integer $F(0:n)$// 全局数组//

If($n=0$ or $n=1$)then $x \leftarrow 1$; return endif

$F(0) \leftarrow F(1) \leftarrow 1$

For $i \leftarrow 2$ to n do

$F(i) \leftarrow F(i-1) + F(i-2)$

Repeat

$x \leftarrow F(n)$

end SimFib

深度优先检索的简单应用之一： 找出无向图的连通分支。 还是以对

称有向图来表示无向图。

以深度优先检索算法为基本框架，从某结点出发，能探索到的所有结点刚好构成一个连通分支，以出发点的编号作为该连通分支每个结点的标记，

算法 7.3 深度优先检索法找出无向图的连通分支

```
procedure  ConnectedComponents(adj,n,comp)
  int  color(n)
  int  v
  Initialize  color array to  white  for all vertices
  For  v ← 1  to  n  do
    If (color(v) = white) then
      call  Compdfs(adj,color,v,v,comp)
    endif
  repeat
end  ConnectedComponents

Procedure  Compdfs(adj,color,v,compnum,comp)
Int  w
Int  temadj
```

$color(v) \leftarrow gray$

$comp(v) \leftarrow compnum$

$temadj \leftarrow adj(v)$

while($temadj \neq nil$) do

$w \leftarrow first(temadj)$

if ($color(w) = white$) then

call Compdfs($adj, color, w, compnum, comp$)

endif

$temadj \leftarrow rest(temadj)$

repeat

//$color(v) \leftarrow black$//

end Compdfs

注：不需要 Postorder vertex processing 和 inorder vertex processing。

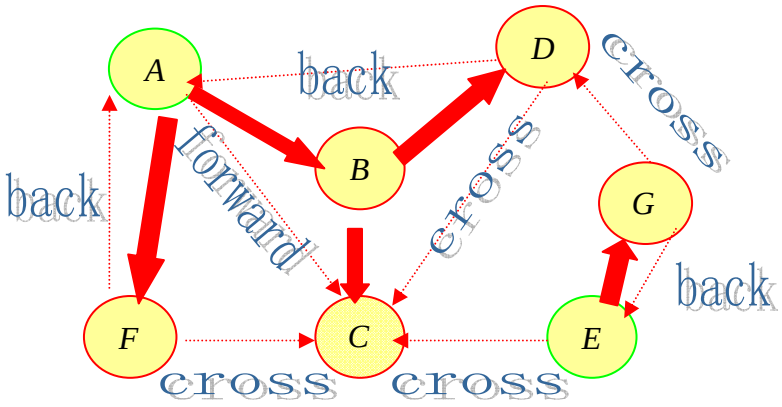
注：时间复杂度$\Theta(n+m)$。

注：可用链表表示这些连通分支的结点集和边集。

深度优先检索树对于深度优先检索过程提供了非常重要的视角，很多问题虽然不需要明显地建构深度优先检索树，但它作为一个分析方法

起作用。

深度优先检索树有关的术语：祖先，父亲，树枝(tree edge)，向后边(back edge)，向前边(forward edge)，交叉边(cross edge)。



非树枝边的分类

算法 7.4 完整细化的深度优先周游过程：

procedure DfsSweep($n, adj, color, ans \cdots$)

int ans // answer//

Allocate $color$ array and initialize to $white$.

For each vertex v of G, in some order:

If($color(v) = white$) then

{

call Dfs($adj, color, v, vAns, \cdots$)

Process $vAns$

}

endif

repeat

process ans

end DfsSweep

procedure Dfs($adj, color, v, vAns, \cdots$)

int w

int $temadj$

int $temans$

$color(v) \leftarrow gray$

Preorder processing of vertex v

$temadj \leftarrow adj(v)$

while($temadj \neq nil$) do

$w \leftarrow first(temadj)$

if($color(w) = white$) then

{

Exploratory processing for tree edge vw

call Dfs($adj, color, w, wAns, \cdots$)

Backtrack processing for tree edge vw, using $wAns$

(like inorder processing of vertex v)

}

else

{ Checking (i.e. processing) for nontree edge vw}

endif

$temadj \leftarrow rest(temadj)$

repeat

Postorder processing of vertex v, including final computation of $vAns$

$color(v) \leftarrow black$

end Dfs

作业

请编制算法来判定有向图G是否反圈，时间复杂度限定为$\Theta(n+m)$，假定G表示为邻接表。

深度优先检索的性质

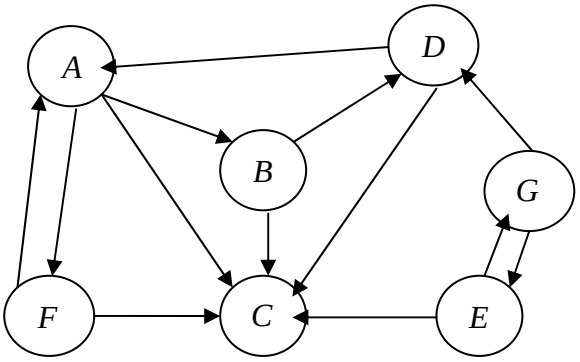
对深度优先周游过程中的两个重要操作：探索到新结点和从完成点回溯，进行记时，即进行一次这样的操作，时钟走动一次。这样就可以得到每个结点的活动时间段。活动时间段对于分析深度优先检索的性质和特征来说，是非常方便好用的概念。

结点v的活动时间段：

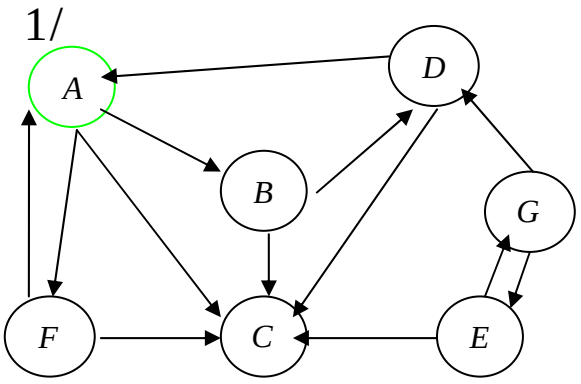
$$active(v) = discovertime(v), \cdots, finishTime(v),$$

其中$discovertime(v)$是结点v被发现的时间 $finishTime(v)$是结点v完成了检索的时间（即回溯时刻）

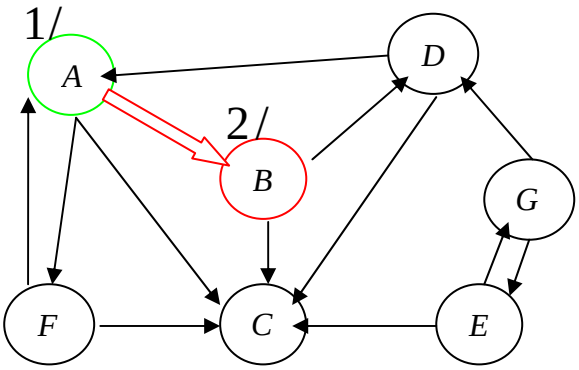
例子说明



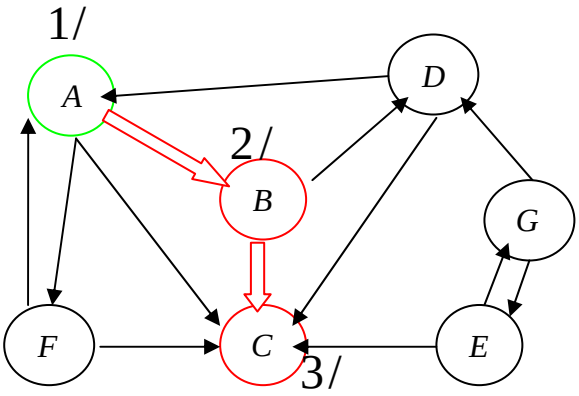
有向图示例



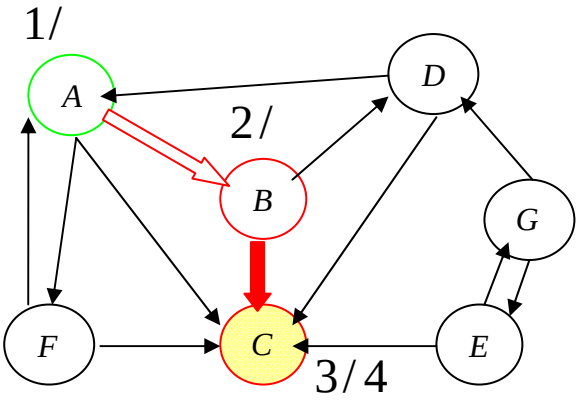
深度优先检索从结点 A 开始



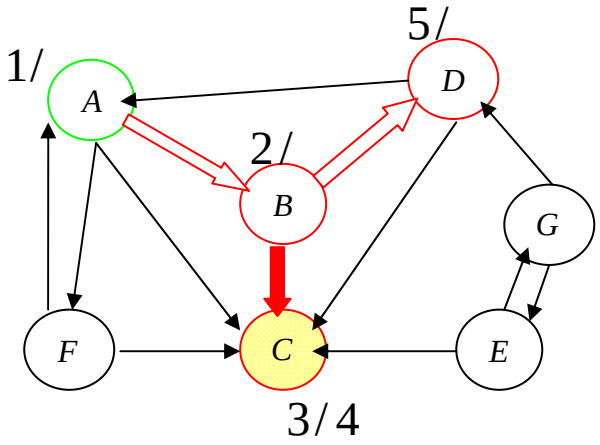
探索到结点 B



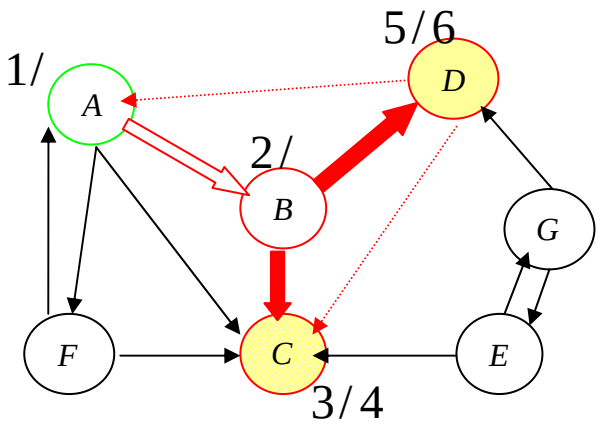
探索到结点 C



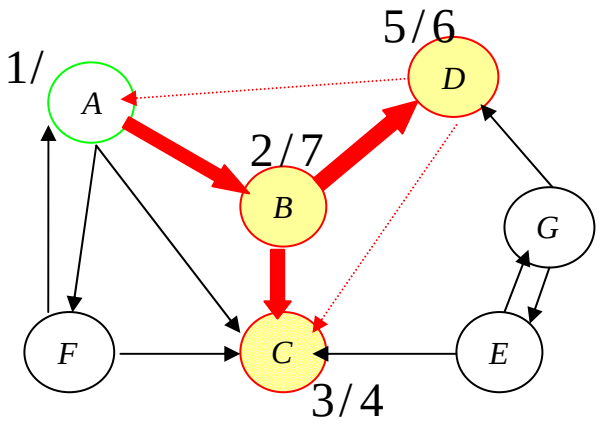
C 变成完成点，从 C 回溯到 B



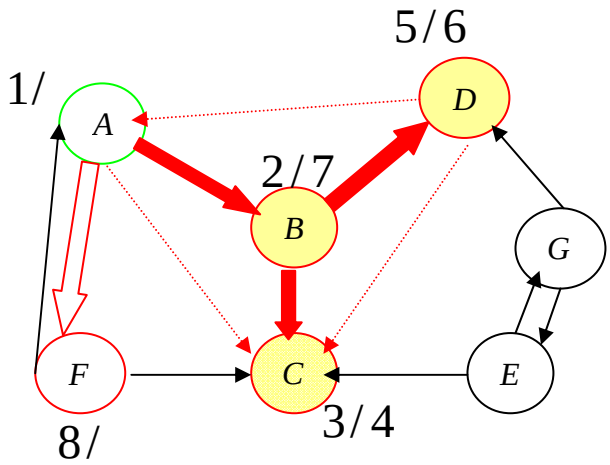
从 B 探索到 D



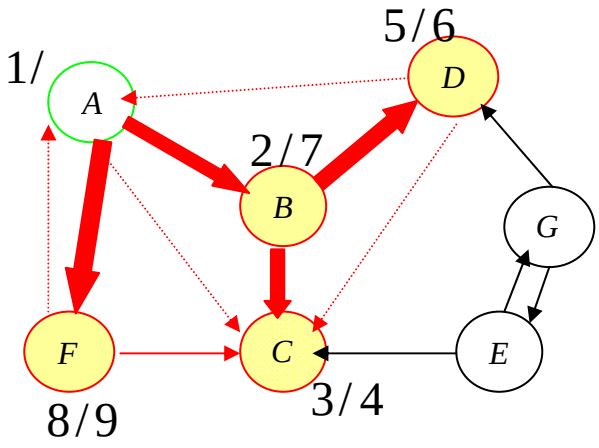
(检查边 DA ， DC)，D 变成完成点，从 D 回溯到 B。



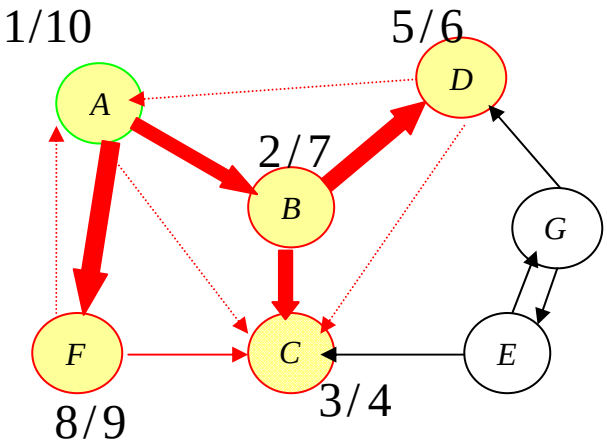
B 变成完成点，从 B 回溯到 A。



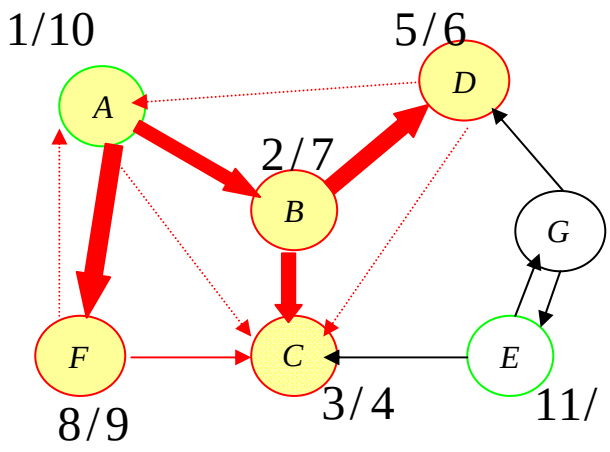
（检查边 AC），探索到结点 F。



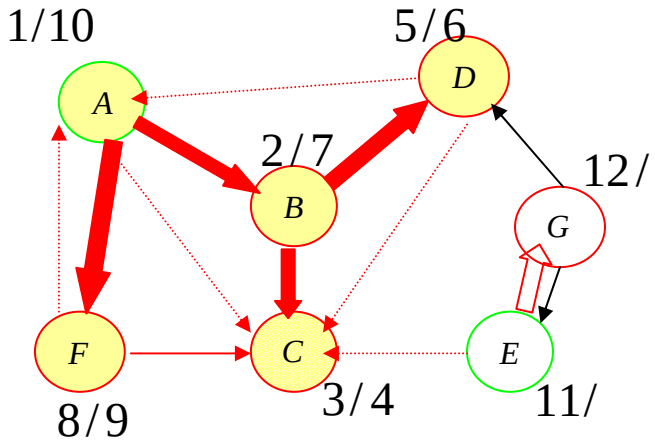
（检查边 FA，FC），F 变成完成点，从 F 回溯到 A。



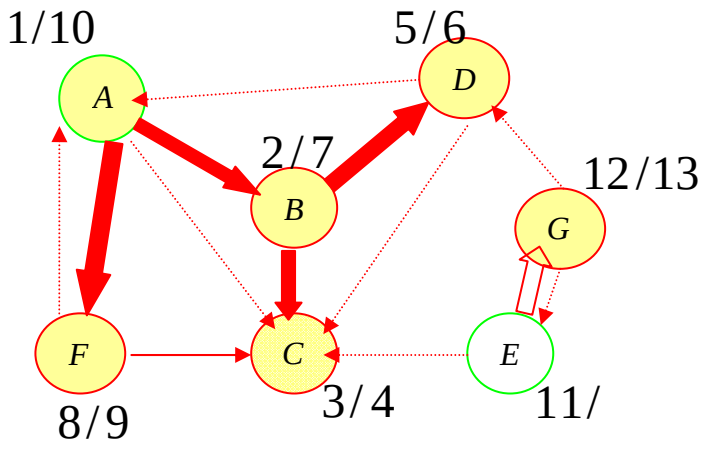
以 A 为起点的深度优先检索至此完成



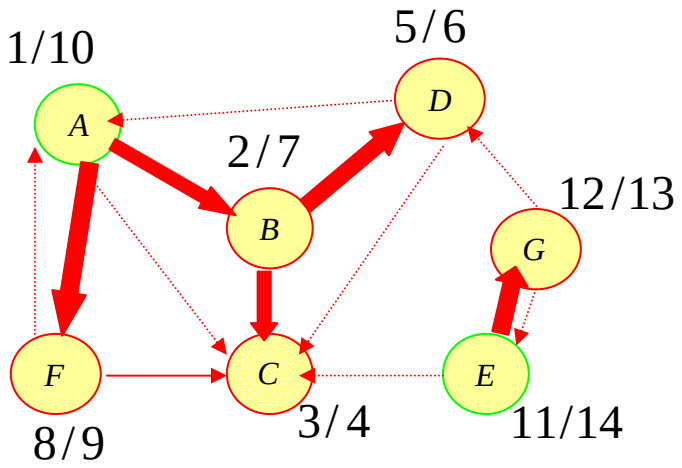
接着以 E 为起点进行深度优先检索



（检查边 EC），探索到结点 G。



（检查边 GD，GE），G 变成完成点，从 G 回溯到 E。



E 变成完成点。至此图的深度优先周游完成。

算法 7.5　算法 7.4 的细化(加入树和活动时间)

procedure　DfsSweep($n, adj, color, ans,$

$parent, discoverTime, finishTime \cdots$)

int ans // answer//

int $time$

Allocate $color$ array and initialize to $white$.

$time \leftarrow 0$

For each vertex v of G, in some order:

If($color(v) = white$) then

{

$parent(v) \leftarrow -1$

call Dfs($adj, color, v, vAns, \cdots$)

Process $vAns$

}

endif

repeat

process　ans

end DfsSweep

procedure Dfs($adj, color, v, vAns, \cdots$)

int w

int $temadj$

int $temans$

$color(v) \leftarrow gray$

$time++;\ discoverTime(v) \leftarrow time$

Preorder processing of vertex v

$temadj \leftarrow adj(v)$

while($temadj \neq nil$) do

$w \leftarrow first(temadj)$

if($color(w) = white$) then

{

$parent(w) \leftarrow v$

Exploratory processing for tree edge vw

call Dfs($adj, color, w, wAns, \cdots$)

Backtrack processing for tree edge vw, using $wAns$

(like inorder processing of vertex v)

}

else

{ Checking (i.e. processing) for nontree edge vw }

endif

$temadj \leftarrow rest(temadj)$

repeat

$time++;\ finishTime(v) \leftarrow time$

Postorder processing of vertex v, including final computation of $vAns$

$color(v) \leftarrow black$

end Dfs

定理 7.1 假定$active(v)$如上定义，则

性质 1. w是v的子孙$\Leftrightarrow active(w) \subseteq active(v)$。此时，如果$w \neq v$，则$active(w) \subset active(v)$。

性质 2. 如果v与w没有祖孙关系，则

$$active(v) \cap active(w) = \varnothing$$。

性质 3. 如果$vw \in E$， 则

a. vw是交叉边$\Leftrightarrow active(w)$完全在$active(v)$之先；

b. vw是向前边$\Leftrightarrow$存在另一个结点x，使得

$$active(w) \subset active(x) \subset active(v)$$；

c. vw是树枝$\Leftrightarrow active(w) \subset active(v)$，但不存在另一个结点$x$，使得$active(w) \subset active(x) \subset active(v)$；

d. vw是向后边$\Leftrightarrow active(v) \subset active(w)$。

推论 7.2 在结点v的活动时间段里发现的所有结点恰是它的所有子孙。

另一个容易看到的事实：w是v的子孙当且仅当v没被发现前存在

一条v到w的道路，其上所有点都尚未被发现。严格的证明形成下面的“白道定理”。

定理 7.3（**白道定理** White Path Theorem）在对一个图G的任何深度优先检索中，结点w是结点v在相应的深度优先检索树中的子孙当且仅当，在结点v被发现的时刻（恰在其被染成灰色之前），G中有一条从v到w的道路，其上的结点全是白结点。

证明：（仅当，必要性）如果w是v的子孙，由深度优先检索树的生成知道，树中从v到w的道路在v被发现的时刻是一条白道路。

（当，充分性）对从v到w的白道路的长度k施行归纳，基础情形是$k=0$，此时$v=w$，定理成立。对于$k\geq 1$，设$P=(v,x_1,\cdots,x_k)$，其中$x_k=w$，是从v到w的长度为k的白道路。设x_i是白道上在v活动时间段内最早被发现的结点。这样的假定是合理的，因为至少x_1能在此期间被发现。这样，x_i是v的子孙。把道路P分成从v到x_i的一段P_1和从x_i到w的一段P_2，如此看来，当x_i被发现时，P_2是从x_i到w的白道路，由归纳假设，w是x_i的子孙，从而w是v的子孙。

作业：

一个有向反圈G称为一个格，如果有一个结点s可以到达所有其他结

点，另有一个结点t能被其他结点所到达。假设有向图G用邻接表表示，请编制算法来判定图G是否格，要求时间复杂度为$\Theta(n+m)$。

关于有向反圈的讨论：

有向反圈之重要性在于

1．工程调度问题由有向反圈模型来表征：某工程由若干任务所构成，这些任务之间存在依赖关系，即每个任务的执行必须在若干任务完成之后方可进行。

2．许多有向图上的问题（指数复杂度）在有向反圈（线性复杂度）上更易解决。

3．任何有向图都唯一对应于一个有向反圈（**收缩图**，condensation graph）

有向反圈在数学上对应于结点集上的一个偏序。 有向反圈G的边vw可解释为$v \prec w$，当G中有从v到w的道路时，也认为$v \prec w$（为使其满足传递性）。

拓扑序（Topological Order）：当我们考虑某些有向图上的问题时，可能出现这样的念头：“如果有向图存在一种画法能使边上的方向都是从左到右（或从上到下，象根树一样），这将有助于解决问题吗？”当然，如果有向图包含回路，这种画法显然不可能。但是，如果有向

图不包含回路，也就是说，是反圈，这种画法能够做到，该画法中结点形成一个顺序，称为图的拓扑序。

拓扑序的定义：设$G=(V,E)$是n阶有向图。G的一个拓扑序是指用整数$1,\cdots,n$给G的n个不同结点进行编号（称为结点的拓扑数，topological numbers）使得对G的任何边vw来说，v的拓扑数总小于w的拓扑数。G^T的拓扑序称为G的反拓扑序（reverse topological order）。

引理 7.4. 如果图G有圈，则G没有拓扑序。

注：在某种意义下，拓扑序是反圈的基本问题，有了拓扑序，许多问题就会变得简单明了。在有些情况下，仅仅意识到拓扑序的概念就可能引导我们得到问题的有效解。

注：以深度优先检索算法为基础，就可得到求解拓扑序的算法，我们给出求反拓扑序的算法，因为在应用问题中经常用到的是反拓扑序。

算法 7.6 求反拓扑序数 Reverse Topological Ordering

在$DfsSweep$框架里另外需要的数据结构：$topo(n)$，$toponum$

输出：全局数组$topo(n)$被赋以对应结点的反拓扑序

Strategy: Modify the $DfsSweep$ and Dfs skeleton as follows :

1. In $DfsSweep$, initialize $toponum$ to 0;

2. In Dfs, at postorder processing, insert

$$toponum++;\ topo(v) \leftarrow toponum$$

注：如果是求解拓扑序，相应地，

1. In $DfsSweep$, initialize $toponum$ to $n+1$;

2. In Dfs, at postorder processing, insert

$$toponum--;\ topo(v) \leftarrow toponum$$

以求反拓扑序数为例

procedure DfsSweep_topOrder($n, adj, color, topo$)

int $toponum$

Allocate $color$ array and initialize to $white$.

$toponum \leftarrow 0$

For each vertex v of G, in some order:

  If($color(v) = white$) then

    call Dfs_topOrder($n, adj, color, v, topo$)

  endif

repeat

end DfsSweep_topOrder

procedure Dfs_topOrder($n, adj, color, v, topo$)

int w

int $temadj$

$color(v) \leftarrow gray$

$temadj \leftarrow adj(v)$

while($temadj \neq nil$) do

  $w \leftarrow first(temadj)$

  if($color(w) = white$) then

    call Dfs_topOrder($n, adj, color, w, topo$)

  endif

  $temadj \leftarrow rest(temadj)$

repeat

$toponum++;\ topo(v) \leftarrow toponum$

$color(v) \leftarrow black$

end Dfs_topOrder

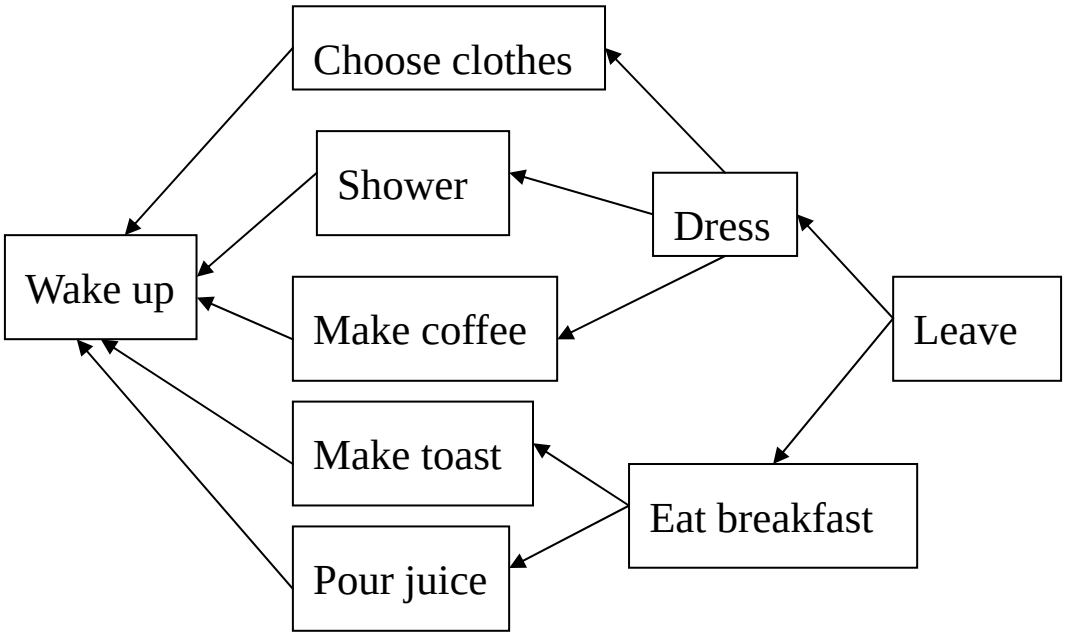
定理 7.5. 上述算法给出了反圈G的一个反拓扑序。

证明：因为只是在结点刚处于完成状态（即回溯）时，被赋予拓扑数。而结点一旦成为完成状态，将不会被再次访问。所以结点被赋拓扑数这种操作对每个结点只会发生一次，n个结点的拓扑数都不同，且取值范围是$1, \cdots, n$这n个数。下面只需要说明，对G的每条边vw来说，都有$topo(v) > topo(w)$。以深度优先检索对边的分类来看，反圈没有“向后边”，而“树枝”，“向前边”，“交叉边”，都满足上面的不等式。

依赖性任务构成的工程调度问题：描述各任务之间依赖关系的最自然方式是列表，表中的每一行列出某个任务直接依赖的所有任务。这儿有一个简单例子：一个人早上起床到离开家所干事项之间的依赖关系，把这些事项作为任务。这些任务的编号按字母顺序来做。

Task and Number	Depends on
Choose clothes 1	9
Dress 2	1,8
Eat breakfast 3	5,6,7
Leave 4	2,3
Make coffee 5	9
Make toast 6	9
Pour juice 7	9
Shower 8	9
Wake up 9	-

用有向图G来表示这些任务之间的依赖关系：每个结点表示一项任务，边vw表示任务v直接依赖任务w。则上面的表给出了图G的外邻点表。图G是反圈，成为依赖图。依赖图的反拓扑序给出工程中任务进行的先后顺序。



工程各任务的依赖关系示例

图G的一个反拓扑序：

深度优先周游

深度优先检索的起点：Choose clothes, 得序：Wake up: 1, Choose clothes: 2 ;

深度优先检索的起点: Dress, 得序：Shower: 3 , Dress: 4;

深度优先检索的起点: Eat breakfast, 得序：Make coffee: 5, Make toast: 6, Pour juice: 7, Eat breakfast:8;

深度优先检索的起点: Leave, 得序; Leave: 9.

按照这个序来进行这些任务，可使得在进行任何任务时，它依赖的任

务已经完成。

关键路径（Critical Path）分析

关键路径分析与求拓扑序有关，但它本身是一个最优化问题，即要求反圈中的最长道路。以上述调度问题为例，一个工程包括一组任务，这些任务之间有依赖关系。现在，假定完成每个任务有一个时间量，此外，假定每个任务都可在它所依赖的任务完成的同时开始进行，也就是说有足够的工人可供支配。当然后一个假定在许多实际情况中是有问题的，但是初步考虑问题时我们先作此简化。

假定工程在0时刻开始。我们可以定义每个任务的最早完成时间。

最早开始时间，最早完成时间，关键路径

假定一个工程包括一组任务，编号为$1, \cdots, n$，对于每个任务v，有一个它所直接依赖任务的列表，一个非负实数表示完成该任务所花费的时间（duration）$d(v)$。任务v的**最早开始时间**$est(v)$定义为

1. 如果任务v不依赖任何任务，则$est(v) = 0$；
2. 否则，$est(v)$是所依赖任务最早完成时间的最大值。

任务v的**最早完成时间**$eft(v)$则是$est(v)$加上$d(v)$。

工程的一个**关键路径**是指一个任务序列$v_0, v_1, \cdots, v_k$使得

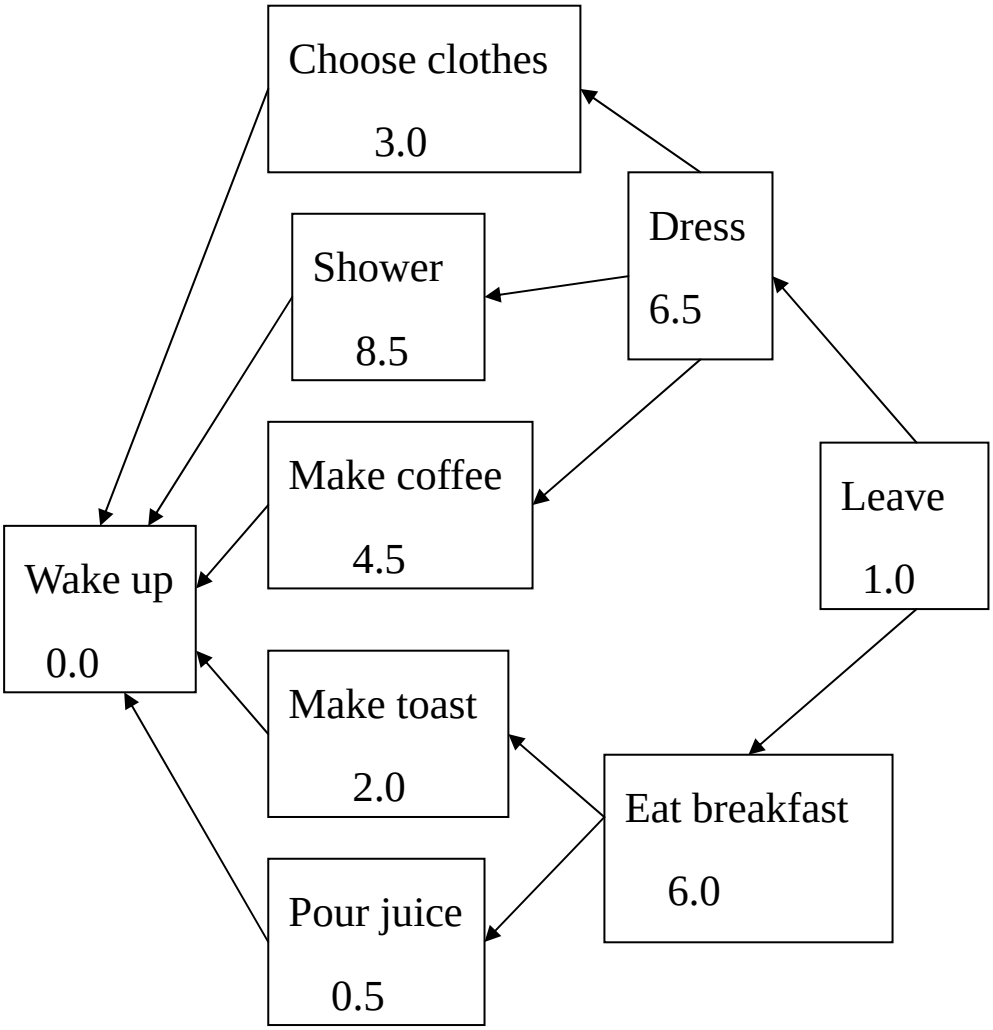
1．v_0无依赖的任务；

2．对于任意$1 \le i \le k$，v_{i-1}是v_i的一个直接依赖任务，且$est(v_i) = eft(v_{i-1})$；

3．$eft(v_k) = \max_{1 \le v \le n} eft(v)$。

关键路径没有“松弛部分（slack）”，即路径上，一个任务的完成和下一个任务的开始之间没有停顿。换句话说，如果在关键路径上，v_i紧跟着v_{i-1}，则v_{i-1}的最早完成时间是v_i的所有直接依赖的任务中最大的。所以v_{i-1}是v_i的关键依赖，意味着v_{i-1}的任何延误将造成v_i的延误。以另外一个观点看，假如我们要通过加快某个任务的完成时间来加快所有任务(整个工程)的完成时间，如果该任务不在所有关键路径上，则不可能做到这一点。研究关键路径的实际意义正在于此。

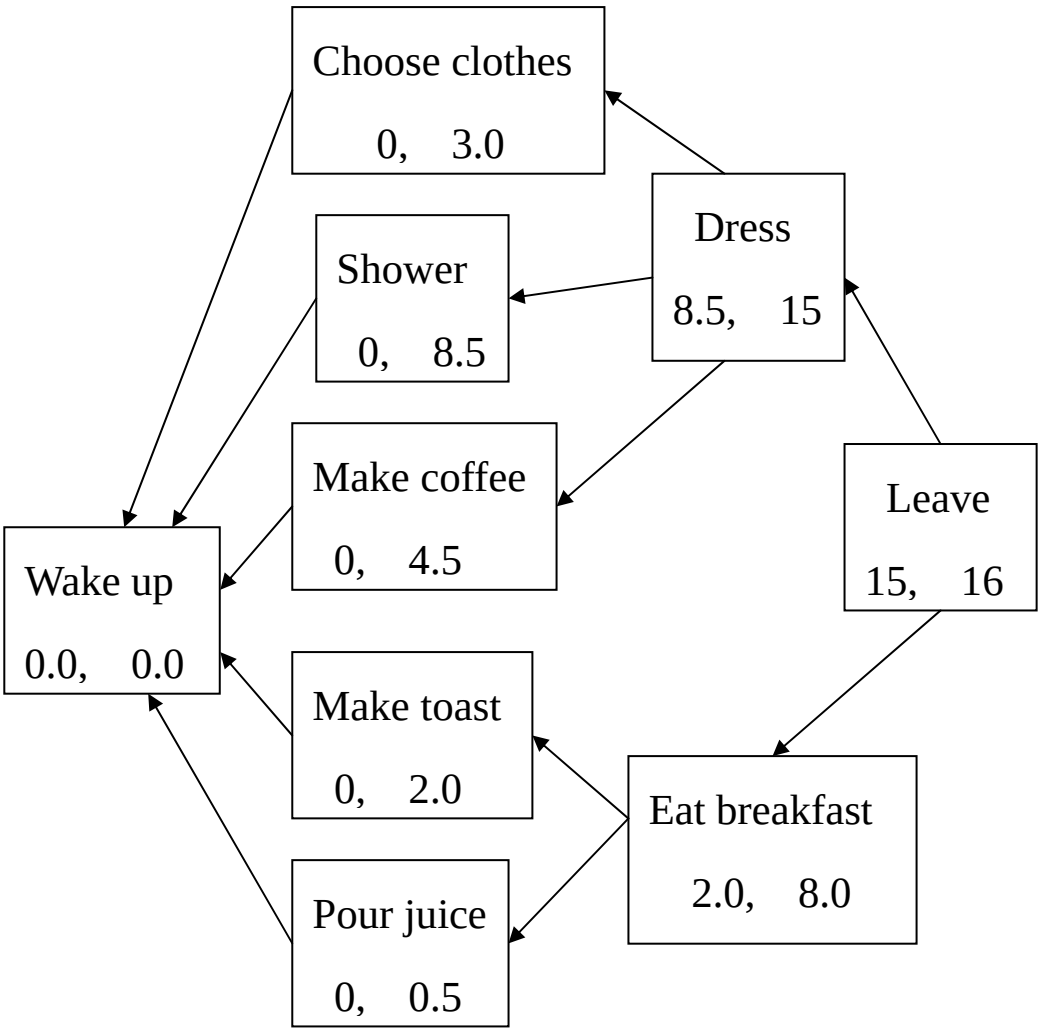
为简单起见，我们假定每个任务有固定的花费时间。在很多实际情形下，可以分配更多的资源给某任务以减少其花费时间，这或许要减少不在关键路径上的某任务的资源。



关于各任务花费时间的描述

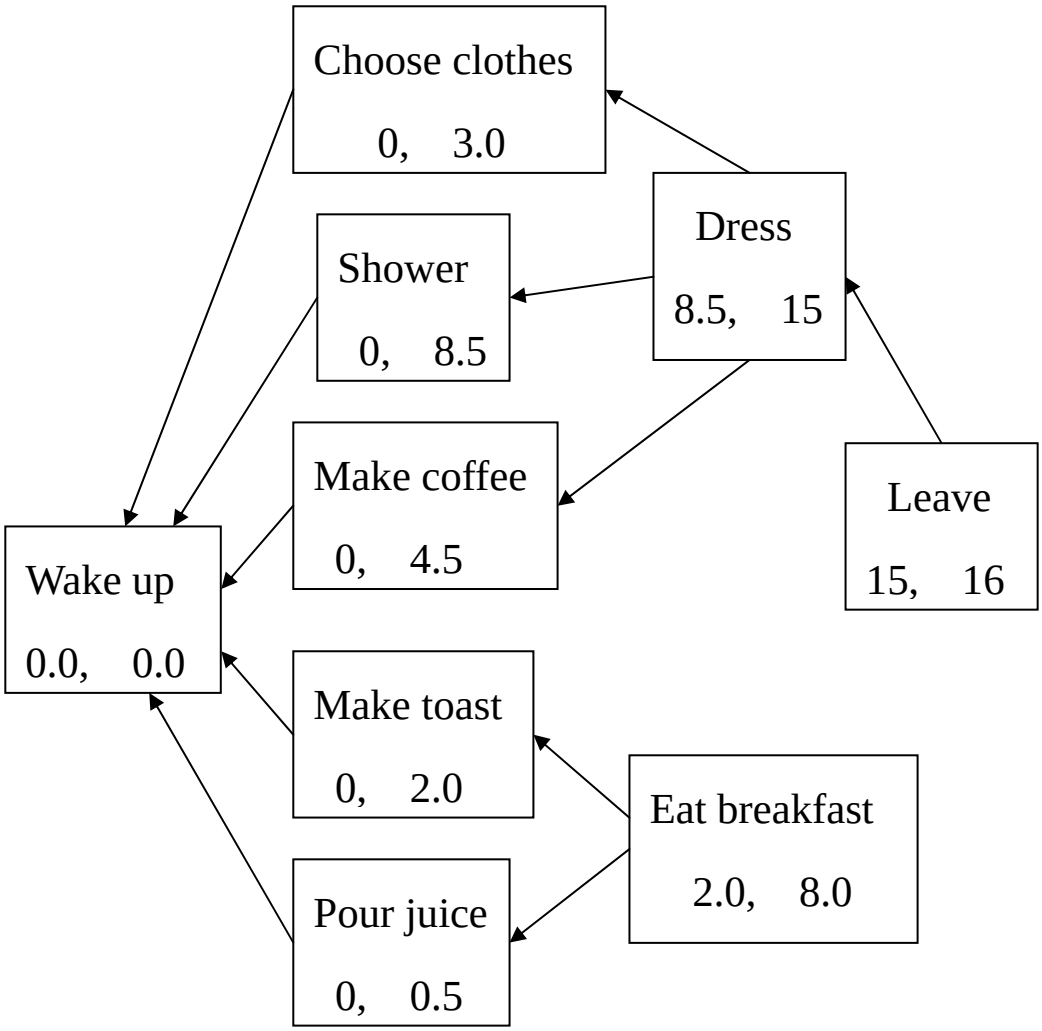
来计算上面例子的关键路径。假定任意多的任务可以同时进行，只要它们所依赖的任务已经完成。如下图所示，各任务的最早开始时间和最早完成时间列在表示结点的方框中。

关键路径：wake up→shower→Dress→leave.



各任务的最早开始时间和最早完成时间

关键依赖：在每个任务v所依赖的任务中，其中有个任务w的最早完成时间最大，即有：$est(v)=eft(w)$。w称为v的关键依赖任务。



各任务的关键依赖任务

关键路径是点加权反圈上的最优化问题。为运算方便起见，对图做点加工：即增添一个结点$done$，它直接依赖的任务是那些不被任何任务直接依赖的任务，它花费的时间量当然为0，其反拓扑数为$n+1$。如此以来，关键路径序列的末点总是结点$done$，即总有$eft(n+1)$最大。可以通过深度优先检索框架来计算$eft(v)$和关键路径。

算法需增添的数据结构：

数组$d(n+1)$：每个任务花费的时间量。

数组$eft(n+1)$：存放每个任务的最早完成时间。

数组$critDep(n+1)$：存放每个任务的关键依赖任务。

结点v_{end}：表示关键路径上的最后一个任务。

算法 7.7 求解关键路径

Procedure

DfsSweep_CritP($n, adj, color, d, eft, critDep, v_{end}$)

Allocate $color$ array and initialize to $white$.

For each vertex v of G, in some order:

If($color(v) = white$) then

call Dfs_CritP($adj, color, v, d, eft, critDep$)

endif

repeat

Find out the vertex v_{end} such that $eft(v_{end})$ is maximum .

end DfsSweepCritP

procedure Dfs_CritP($adj, color, v, d, eft, critDep$)

int w

int $temadj$

$color(v) \leftarrow gray$

$est \leftarrow 0$； $critDep(v) \leftarrow -1$

//Preorder processing of vertex v//

$temadj \leftarrow adj(v)$

while($temadj \neq nil$) do

$w \leftarrow first(temadj)$

if($color(w) = white$) then

call Dfs_CritP($adj, color, w, d, eft, critDep$)

endif

if($eft(w) \geq est$) then

{ $est \leftarrow eft(w)$, $critDep(v) \leftarrow w$ }

endif

$temadj \leftarrow rest(temadj)$

repeat

$eft(v) \leftarrow est + d(v)$

$color(v) \leftarrow black$

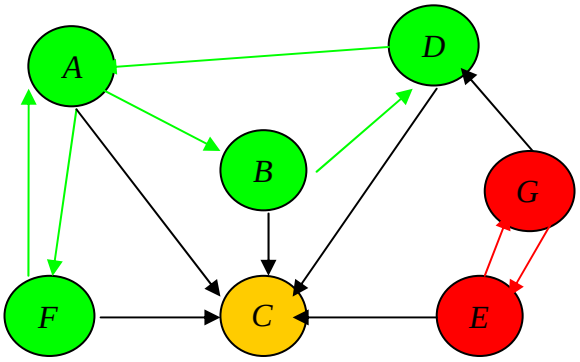
end Dfs_CritP

我们只是论述了有向反圈的一些基本的东西，有向反圈还有许多应用。下一节，我们将了解到，任何有向图G都相伴着一个反圈：G的收缩图（condensation graph），所以反圈的应用问题可延伸到含圈的有向图。

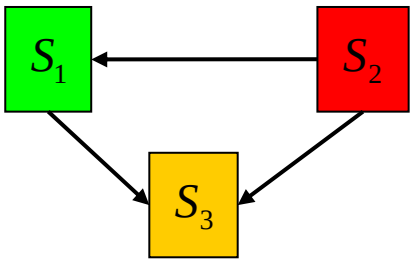
5．有向图的强连通分支

收缩图：设$S_1,S_2,\cdots,S_p$是有向图$G=(V,E)$的强连通分支。G的收缩图$G\downarrow$定义为有向图$G\downarrow=(V\downarrow,E\downarrow)$，其中$V\downarrow$含有 p 个元素$s_1,s_2,\cdots,s_p$，$s_is_j\in E\downarrow$当且仅当E中有一条从S_i到S_j的边。$G\downarrow$相当于对G作如下操作得到的图：G的每个强连通分支收缩为一点，每两个强连通分支之间的所有边化为所对应两个收缩点之间的一条边。如果不考虑边的方向，有向图与其转置图具有相同的强连通分支，和相同的转置图，即有：$(G^T)\downarrow=(G\downarrow)^T$。

举例



三个强连通分支



收缩图

如果我们能够确定有向图G的所有强连通分支，G的收缩图就很容易得到。我们将要用深度优先检索方法来解决这个问题。

为此，我们来看深度优先检索树与强连通分支之间的关系。

强连通分支的 leader：给定有p个强连通分支$S_1, S_2, \cdots, S_p$的有向图G，在对G所作的深度优先周游过程中，S_i（$1 \le i \le p$）中首先被发现的结点称为S_i的 leader，记作v_i。

假如深度优先周游从v_1（强连通分支S_1的 leader）开始，即v_1是深度优先检索树的根。则由白道定理，S_1中所有结点在该深度优先检索树中都是v_1的子孙。而且，如果能到达任何强连通分支S_j（首先被发现的是v_j），则在v_j被发现时应用白道定理，我们看到S_j中所有结点都在该树上。随后其他的深度优先检索树也同此理。这证明了下面引理。

引理 7.6 有向图G的任何深度优先检索森林的每一棵树由一个或多个强连通分支所组成，（即任何强连通分支的结点都同属于某棵树）。

性质 7.7 leader v_i 是S_i中最后完成的结点。（leader 冲锋在前，撤退在后）

那么是否存在一种（结点的）被检索顺序，使得每一棵树恰好组成一个强连通分支？这是我们求解强连通分支的基本思路。从收缩图可以清楚地看到，如果一个强连通分支没有发射出来的边(arrows coming out from it)，则以它的任何结点为起始点的深度优先检索恰好遍历该分支。但是这个事实又有什么实际意义呢？。下面通过考察 leader 的进一步性质，我们就可以利用这个事实得到求解算法。

尽管我们不知道强连通分支或 leaders，我们还是能够得到一些结论。假如从强连通分支S_i到S_j存在一条边，意味着G中有一条从v_i到v_j的道路。如果v_i比v_j被发现得早，则在包含v_i的深度优先检索树中，v_j是v_i的子孙，此时，$active(v_j) \subset active(v_i)$。反之，如果$v_j$比$v_i$被发现得早，显然，$v_j$不可能是$v_i$的祖先，否则它们将处于同一个强连通分支。

引理 7.8 在深度优先检索中，当 leader v_i被发现时，不存在从v_i到任何灰（gray）结点x的道路

证明：此时，任何灰结点都是v_i的真祖先，且被发现在v_i之前，所以必定在其它的强连通分支中。既然已有从x到v_i的道路，就不能再有从v_i到x的道路。

引理 7.9 如果v是某强连通分支S的 leader，x是另一强连通分支中的结点，且G中存在从v到x的道路，则在v被发现之时，要么x是

黑结点，要么存在从V到X的白路（X是白结点），在两种情况下，X都比V完成得早。

证明：V被发现之时，考察任何从V到X的道路，如果该道是白道，所证成立。否则，设Z是该道上最后一个非白结点，由引理 7.8, Z必是黑结点。如果$Z \neq X$，则回顾Z被发现时，Z到X这段是白道，由白道定理，X是Z的子孙，X必在Z变黑之前变成黑结点, 所以X也是黑结点，这与Z是该道上最后一个非白结点相矛盾， 所以$Z = X$。总之，X是黑结点。

所以，如果从强连通分支S_i到S_j存在一条边，我们排除了v_j的活动时间段整个在v_i的活动时间段后面的可能性。如果不要求v_i是 leader 的话，可找到引理的反例。

求强连通分支的算法

我们将利用上述性质得到求强连通分支的算法。第一个线性复杂度的算法归功于 R.E.Tarjan, 基于深度优先检索。我们要叙述的算法是属于 M. Sharir 的，也基于深度优先检索。它简单精微而颇显雅致。

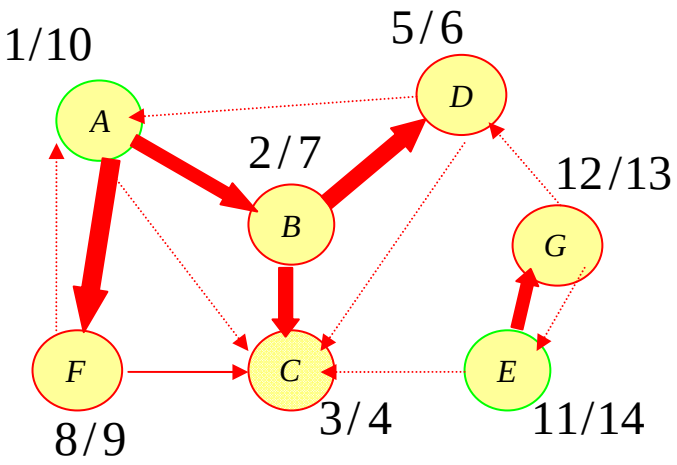
算法包括两大步：

1. 对G作深度优先周游，G的所有结点按其完成时间被堆放在一个栈中；

2．对G^T作深度优先周游，只是需要按结点出栈的顺序确定每一次新的检索的起点。每一次检索恰好得到一个G^T的强连通分支。

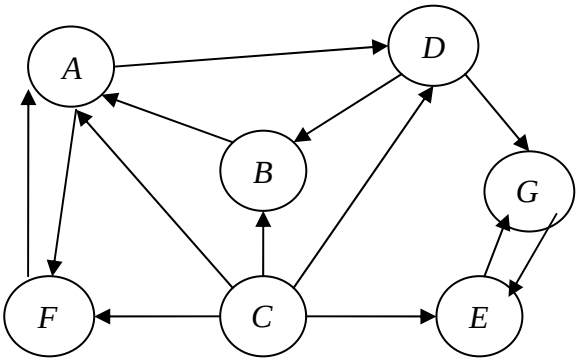
注：算法将启用一个数组$scc(n)$以存放每个结点所在强连通分支的leader。

举例

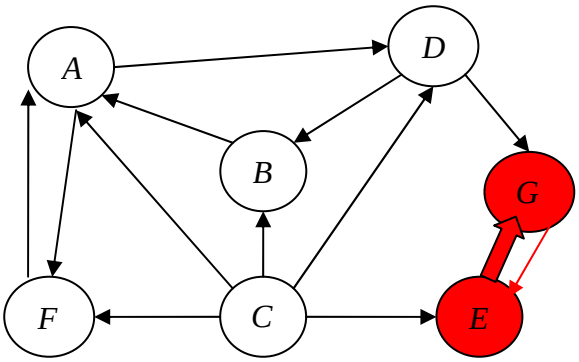


E
G
A
F
B
D
C

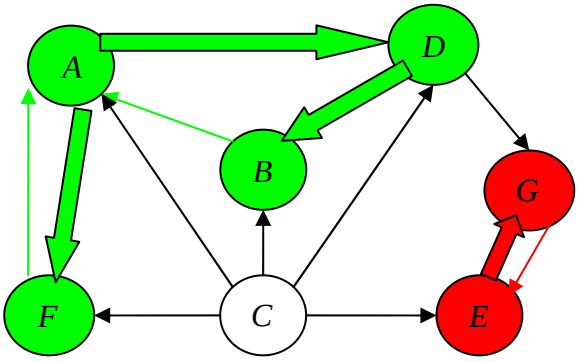
完成对G的深度优先周游后，栈的情形



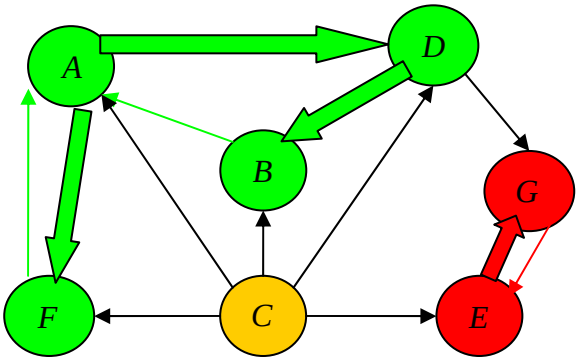
转置图



E 为起点深度优先树



A 为起点深度优先树



C 为起点深度优先树

算法 7.8 求强连通分支

procedure StrongComponents(adj, n, scc)

1. create $finishstack$;
2. perform a depth-first search on G, using the DFS skeleton. At postorder processing for vertex v, insert the statement: $push(finishstack, v)$;
3. Compute G^T, the transpose graph, represented as the array $Transadj$ of adjacency lists.
4. DfsTSweep($transadj, n, finishstack, scc$)

end StrongComponents

procedure DfsTSweep($transadj, n, finishstack, scc$)

//DfsSweep on transpose graph//

Allocate $color$ array and initialize to $white$.

While($finishstack$ is not empty) do

$v \leftarrow top(finishstack)$

$pop(finishstack)$

if($color(v) = white$)then

call DfsT($transadj, color, v, v, scc$) endif

repeat

end DfsTSweep

procedure DfsT($transadj, color, v, leader, scc$)

Use the standard depth-first search skeleton. At preorder processing for Vertex v, insert the statement:

$$scc(v) \leftarrow leader$$

pass *leader* and *scc* into recursive calls

end DfsT

引理 7.10 算法的第二大步中，每当从栈中弹出一个白结点时，该结点必是第一大步中某强连通分支的 leader。

证明：弹出结点的顺序是逆着第一步深度优先周游中结点的完成顺序，即后完成的结点先弹出。而 leader 总是其所在强连通分支中最后完成的结点。所以当非 leader 结点x被弹出时，因为其 leader 结点先于它被弹出，故x不可能是白结点。

定理 7.11 算法的第二大步中, 每个深度优先检索树恰好构成一个强连通分支。

证明：一般说来，每个深度优先检索树由若干个强连通分支构成。我们需要说明的是，在这种情况下，正好是一个。设v_i是第一步中某S_i的 leader, 假设v_i从栈中被弹出时是白结点，则v_i是一棵深度优先检索树的根，如果v_i在G^T不能到达其它强连通分支，则结论成立毫无

问题。假如v_i在G^T能到达某强连通分支S_j(其 leader 是v_j)，则在G中存在从v_j到v_i的路径，由引理 7.9, 在第一步的周游中v_j完成在v_i之后，所以在第二大步的周游中当v_i从栈中被弹出时，v_j已被弹出且S_j中所有结点都已成为黑结点。故从v_i出发的检索不会跑出S_i。

定理 7.12 算法*StrongComponents*正确地求出了G^T的所有强连通分支。

注：算法*StrongComponents*的时间复杂度与空间复杂度都是$\Theta(n+m)$。

6. 无向图上的深度优先检索

无向图上的深度优先检索的基本思路与有向图的情况一样：尽可能“探索”，必要时回溯。所以可沿用有向图上深度优先检索算法的框架，及其一些主要术语：vertex color, discovery times, finishing times, DFS trees. 但无向图上的深度优先检索更为复杂，其原因在于每条边应该只被“探索”或“检查”一次，但在数据结构中每条边有两种表示。

有些无向图的问题每条边被处理两次并无影响，如求连通分支，所以

可把无向图当作对称有向图。这节涉及的问题作这种简化行不通。原则上讲，无向图牵扯到回路的问题就要求每条边只能被处理一次。

对于一个无向图来说，深度优先检索给它的每条边标定了一个方向，这取决于边在哪个端点处首次被遇到，则以该端点为起点定义边的方向。同有向图一样，被发现的边称为树枝，被检查的边称为非树枝边。

当一个结点在其数据结构(邻接表，或邻接矩阵)中遇到已标定方向的一条边（其定向指向该结点）时，不再作任何处理，因为该边已被处理过。DFS 框架可以稍加修改，使其能够辨认这种情形。对于无向图，非树枝边有下列事实：

1. 不会出现交叉边。
2. 如果vw是向后边，w必是灰结点，且w不是v之父亲。
3. 向前边是无向边的第二次出现，故不必处理。因为vw作为向前边被发现时，vw已经作为结点w处的向后边被处理过了。实质上说，没有向前边。

所以，无向图只有树枝和向后边。

上述事实促使我们对深度优先检索框架作下面修改。首先，Dfs 将增添检索起点v的父结点 p 为参数，这样使得在处理边vw时，如果w是非白点，则来作进一步的检测。如果w是灰结点且其不同于v的

父结点 p，则边 vw 第一次被遇到，是向后边；否则，vw 已作为向后边或树枝（wv）被处理过了。

算法 7.9 无向图的深度优先检索

procedure Dfs$(n, adj, color, v, p, ans\cdots)$

int w

int $temadj$

int $wAns$

1. $color(v) \leftarrow gray$

2. Preorder processing of vertex v

3. $temadj \leftarrow adj(v)$

4. while($temadj \neq nil$) do

5. $w \leftarrow first(temadj)$

6. if($color(w) = white$) then

7. { Exploratory processing for tree edge vw

8. call Dfs$(n, adj, color, w, v, wAns, \cdots)$

9. Backtrack processing for tree edge vw, using $wAns$ (like inorder)}

10. else if ($color(w) = gray \&\& w \neq p$) then

11. { Checking (i.e. processing) back edge vw}

//else wv was traversed , so ignore vw. //

endif

endif

12. $temadj \leftarrow rest(temadj)$

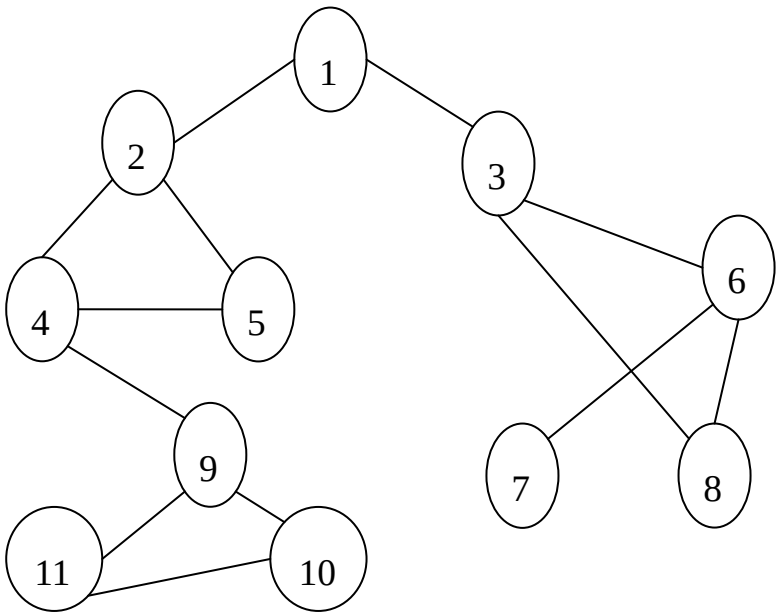
repeat

13. Postorder processing of vertex v, including final computation of ans

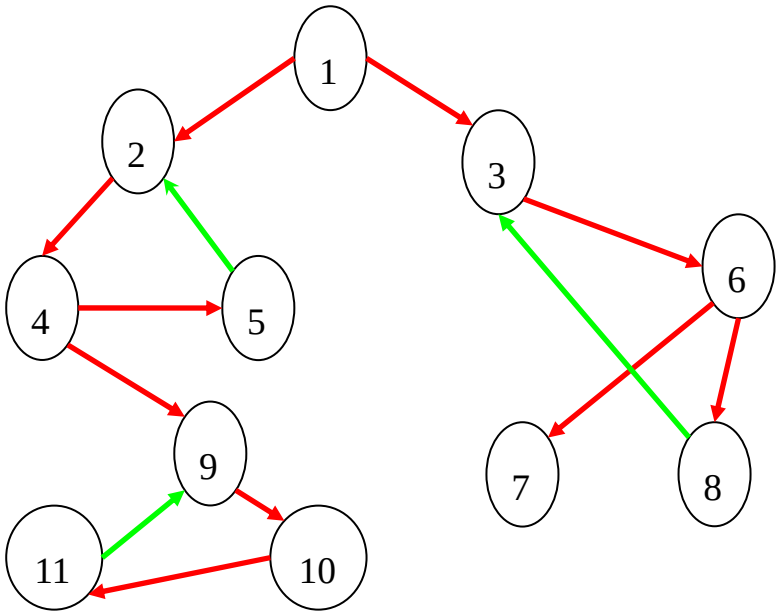
14. $color(v) \leftarrow black$

end Dfs

时间复杂度$\Theta(n+m)$，额外的空间开销$\Theta(n)$。



无向图示例



深度优先检索：红色边是树枝，绿色边是向后边

无向图宽度优先检索

问题可简化为：将无向图表示成对称有向图，采用有向图的深度优先检索。

7．无向图的双连通分支

背景问题举例：

1．在交通图中，一个站点被毁坏，其它站点能否仍然连通？

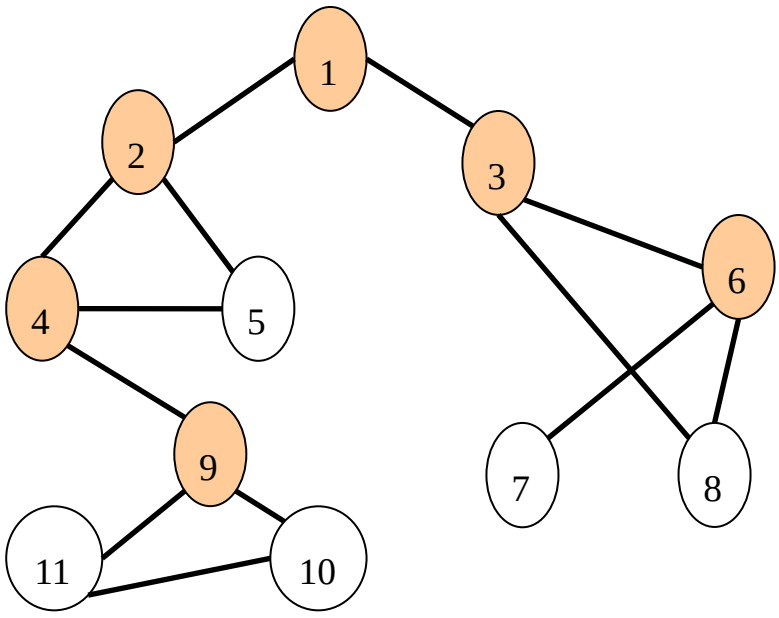
2．计算机网络中，一台计算机除了故障，其他计算机能否继续保持通讯？

问题：删去连通无向图的一个结点（及其与它关联的边），所留下来的图还连通吗？

这个问题在表示通讯或运输网络的图中，是一个很重要的问题。找出能造成不连通的结点（如果删去它）也是很重要的。本节的目的是介绍回答这些问题的一个有效算法。这个算法是由 R.E.Tarjan 发现的，是佐证深度优先检索强大威力的早期算法之一。

关节点（断点，articulation point，cut point）与双连通分支

无向图G的结点u称为**关节点**（或**断点**），如果存在两个不同结点v和w，使得u在所有v和w之间的道路上。



茶色结点是关节点

显然，删去一个关节点就会导致一个不连通图。所以一个连通图称为是双连通的，如果它不包含任何关节点。双连通分支给出图边集的一个划分。可以在G的边集E上定义一个关系二元～：$e_1 \sim e_2$当且仅当$e_1 = e_2$或e_1和e_2都在某个回路上。 可验证～是等价关系，其每个等价类对应着G的一个双连通分支。

我们将给出的求解双连通分支的算法利用无向深度优先检索算法框架，主要基于深度优先检索树的**树枝必是向后边**的这一特征。在检索进行的过程中，所需要的信息被计算，图的边集及其关联的结点按其所在双连通分支而被划分存储。 那么需要什么样的信息？如何确定一个双连通分支？因为每个回路至少含有一条向后边，所以需要研究向后边与双连通分支之间的关系

双连通分支算法

在深度优先检索过程中，结点可以在三种时刻进行处理：先序（它被发现时），中序（每个子结点回溯到它时），后序（它将要完成时）。双连通分支算法对深度优先检索树中的每个结点在它回溯时进行检测，以断定它是不是关节点。再重申一下，对深度优先检索树（本节简称为“树“）来讲，所有边都是树枝或向后边。

假设检索从结点W回溯到V时，不存在从W为根的子树中的某点到V的某个真祖先的向后边，则V一定在从树的根到W的每条道路上，所以是关节点。W为根的子树及其中的向后边再加上边VW将与图的剩余部分在关节点V分离开来，但它未必是一个双连通分支，它可能由几个连通分支所组成。我们将确保每个双连通分支被恰当地分离，这可以通过每发现一个双连通分支就移走来做到。当W回溯到V时，W为根的子树中含有的其它双连通分支已被剥离，剩下的都在同一个分支中。

定理 7.13 在深度优先检索树中，对任何结点V，

1. 如果V是根结点，则V是关节点当且仅当V有不少于两棵子树；
2. 如果V不是根结点，则V是关节点当且仅当，V不是外结点且V的某棵子树没有与V的真祖先关联的向后边。

对每个结点V，它有被发现的时间$discoverTime(v)$，我们知道，当V是W的真祖先时，$discoverTime(v) < discoverTime(w)$，以$back(v)$定义以$V$为根的子树通过向后边可到达的最早的祖先的被发现时间，则

1 $$back(v) = \min\{discoverTime(v), \min_{vw\text{是向后边}} discoverTime(w), \min_{w\text{是}v\text{的儿子}} back(w)\}$$

2 对对任何内结点v来说，v是关节点当且仅当存在v的某儿子w，使得$discoverTime(v) \le back(w)$。

算法 7.10 求解无向图的双连通分支

procedure bicomponents(adj, n)

int v, $color(n)$

Initialize $color$ array to $white$ for all vertices

$time \leftarrow 0$

$edgestack \leftarrow create()$

for $v \leftarrow 1$ to n do

if($color(v) = white$) then

call bicompDFS$(adj, color, v, -1, back)$ endif

repeat

end bicomponents

procedure bicompDFS$(adj, color, v, p, back)$

int w

IntList $remAdj$

1. $color(v) \leftarrow gray$

2a. $time++$; $discoverTime(v) \leftarrow time$

2b. $back \leftarrow discoverTime(v)$

3. $remAdj \leftarrow adj(v)$

4. while($remAdj \neq nil$) do

5. $w \leftarrow first(remAdj)$

6. if($color(w) = white$) then

{

$push(edgestack, vw)$

call bicompDFS($adj, color, w, v, wback$)

//backtrack processing of tree edge vw//

if($wback \geq dis\,cov\,erTime(v)$) then

{

Initialize for new bicomponent

Pop and output $edgestack$ down through vw

}

else $back \leftarrow \min(back, wback)$

endif

}

else

{

if($color(w) = gray \,\&\&\, w \neq p$) then

//process back edge vw//

{

$push(edgestack, vw)$

$back \leftarrow \min\{back, discoverTime(w)\}$

}

//else wv was traversed , so ignore vw//

endif

}

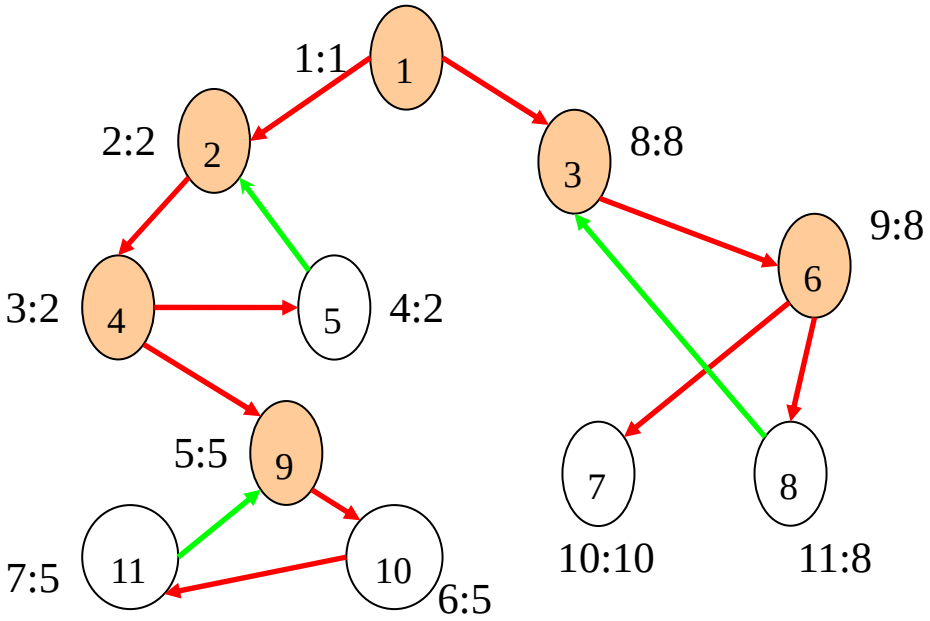
endif

$remAdj \leftarrow rest(remAdj)$

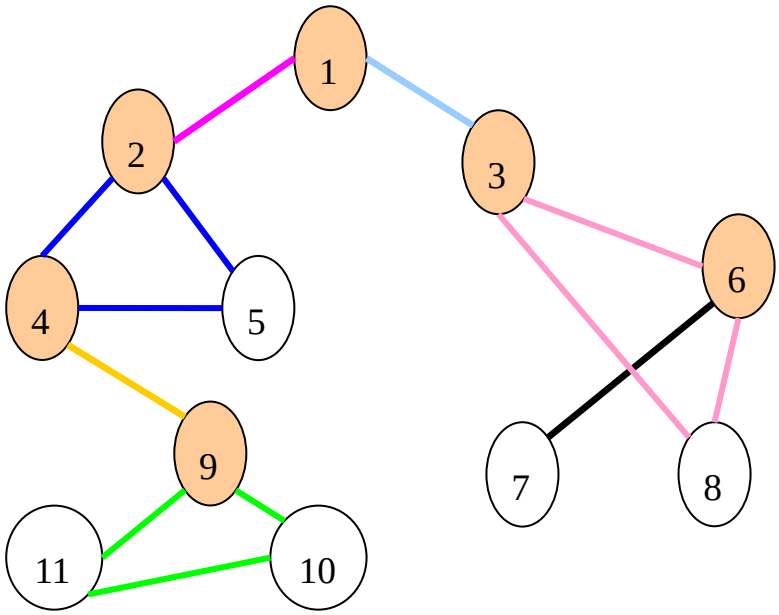
repeat

$color(v) \leftarrow black$

end bicompDFS



discoverTime : back 示意图



双连通分支

作业：

1. 请编制算法判定无向图G是否二部图。图G用邻接表表示， 要求算法复杂度为$O(n+m)$。
2. 称s是有向图G的超汇点，如果对G的任何其他结点v来说，都有vs是G的边，而sv不是G的边。请给出一个算法判定有向图G是否有超汇点，在有向图G用邻接矩阵表示的情况下，要求算法最坏时间复杂度为$O(n)$。
3. 有n台相互联网的计算机，分别标记为$c_1,c_2,\cdots,c_n$，某种病毒在时刻x感染了其中的一台计算机c_a，问在时刻y（$y>x$）计算机c_b有没有感染这种病毒。已知数据为m个三元组(c_i,c_j,t_k)，(c_i,c_j,t_k)表示计算机c_i与计算机c_j在时刻t_k进行过数据交换；

如果计算机c_i（计算机c_j）在时刻t_k前（包括在时刻t_k）感染这种病毒，那么在时刻t_k计算机c_j（计算机c_i）也会感染这种病毒。假设任两个计算机至多进行过一次数据交换，m个三元组(c_i,c_j,t_k)分别在数据交换发生的时刻t_k报告给你。请编制算法解决该问题。要求算法时间复杂度为$O(n+m)$。

第六章 回溯法(BACKTRACKING)

1. 概述

回溯法是一种穷举方法。对于寻找一组解的问题或者求满足某些约束条件的最优解问题都可采用回溯法求解。其基本思想是：假定一个问题的解能够表示成n元组$(x_1,\cdots,x_n)$，其中x_i取值于有限集合$S_i(1\le i\le n)$。n元组的子组$(x_1,\cdots,x_i)(0\le i\le n)$应满足一定的约束条件(这种约束条件一般可表示成谓词，记作$f_i(x_1,\cdots,x_i)$，要使$f_i(x_1,\cdots,x_i)$取真值)，称其为部分解。我们采用两种操作过程来求解。其一是添加操作，在已有部分解$(x_1,\cdots,x_i)$ (空数组()是初始化部分解)的情况下，试图随尾添加某$x_{i+1}\in S_{i+1}$，形成更大的部分解$(x_1,\cdots,x_i,x_{i+1})$。 这需要验证$(x_1,\cdots,x_i,x_{i+1})$是否仍满足约束条件, 满足的话，则添加成功，继续后一步的添加操作；否则，重新选取S_{i+1}中的其他元素进行验证。如果遍历S_{i+1}中的所有元素，都不能使添加成功，这表明部分解$(x_1,\cdots,x_i)$不可能扩张为一个全解, 此时，我们施行第二种操作：回溯（“返工”），即删掉该部分解的尾巴后退到部分解$(x_1,\cdots,x_{i-1})$，转入前一步的添加操作过程，尝试添加此前未曾被添加过的S_i中的元素。添加回溯反复进行，直到

(1) 当得到一解时，不需要再进行下去（问题只要求一个解）；

或

（2）当回溯到空数组，且无新元素可添加，无法再进行下去，即

已遍历所有n元组，找到了所有的解（无解是其中一特殊情形）。

回溯法可有一比，好比从山脚下找一条爬上山顶的路，起初有好几条道可走，当选择一条道走到某处时，又有几条岔道可供选择，只能选择其中一条道往前走，若能这样子顺利爬上山顶则罢了，否则走到一绝路上时，只好返回到最近的一个路口，重新选择另一条没走过的道往前走。如果该路口的所有路都走不通，只得从该路口继续回返。照此规则走下去，要么找到一条到达山顶的路，要么最终试过所有可能的道，无法到达山顶。

例 6.1 令x_1，x_2和x_3是三角形的三边长，求边长为整数且满足$x_1+x_2+x_3=14$的三角形种类。

解 1：不妨设$x_1 \geq x_2 \geq x_3$。x_1，x_2和x_3是三角形的三边长$\Leftrightarrow x_2+x_3 \geq x_1$，再由$x_1+x_2+x_3=14$知，$14/3 \leq x_1 \leq 14/2$（即$5 \leq x_1 \leq 6$），$x_2 \leq x_1$，$x_3 \leq x_2$。利用回溯法求解该问题如下：

解的表达形式(x_1,x_2,x_3)，其实应为(x_1,x_2)，因为$x_3=14-x_1-x_2$。

$() \rightarrow (5) \rightarrow (5,5)$

$() \rightarrow (5) \rightarrow (5,4)$

$() \rightarrow (6) \rightarrow (6,6)$

$() \to (6) \to (6,5)$

$() \to (6) \to (6,4)$

$() \to (6) \to (6,3)$

解 2：不妨设 $x_1 \geq x_2 \geq x_3$。x_1，x_2 和 x_3 是三角形的三边长 $\Leftrightarrow x_2 + x_3 \geq x_1$。再由 $x_1 + x_2 + x_3 = 14$ 知，$14/3 \leq x_1 \leq 14/2$（即 $5 \leq x_1 \leq 6$），$14 - x_1 - x_2 \leq x_2 \leq x_1$（即 $(14 - x_1)/2 \leq x_2 \leq x_1$，）。利用回溯法求解该问题如下：

解的表达形式 (x_1, x_2, x_3)，其实应为 (x_1, x_2)，因为 $x_3 = 14 - x_1 - x_2$。

$() \to (5) \to (5,5)$，

$() \to (6) \to (6,4)$，

$() \to (6) \to (6,5)$，

$() \to (6) \to (6,6)$。

最后解得 4 种三角形：$(5,5,4)$，$(6,4,4)$，$(6,5,3)$，$(6,6,2)$。

从上面的讨论可以看出，用回溯法求解一个问题的过程，总可归结为深度优先周游一棵树的过程，这棵树的结点越少，高度越小，则算法的时间复杂度越小。回溯法仅提供了一种规律性穷举的方式，一般说来，时间复杂度比较高。

例 6.2 稳定婚姻问题：依照男女间相互喜爱程度将 n 个男子与 n 个女

子进行婚配。如果就已婚配成的n对夫妇来说，有一男一女不是夫妇，但相互喜爱的程度超过了对各自配偶的喜爱，则这样的n对夫妇称为不稳定婚姻。不存在上述不稳定因素的n对夫妇称为稳定婚姻。我们的目标是要制定一个稳定婚配。当然，我们事先知道这n个男子与n个女子对异性的喜爱情况。每人都填写好一个“志愿表”，即每个男子按自己的喜爱倾向给n个女子排序，同样每个女子按自己的喜爱倾向给n个男子排序。

注：稳定婚姻问题初看起来有点无聊，但将它应用到其他情况就有意义了。例如某公司的 10 个部门各空出一个位置，公司人力资源部招募了 10 名员工，要将他们安置到这些空缺的岗位上。为了避免日后出现员工不安心本职工作，要求调动的麻烦。分配工作前，对员工证求意见，填写对工作的志愿表，同时也召集各部门经理，征求意见，填写对员工的选择表。据此分配工作以保障安定团结的局面。该问题可归结为稳定婚姻问题模型。

解：将n个男子分别编号为$1, \cdots, n$；n个女子也分别编号为$1, \cdots, n$。引入四个n阶方阵MR，MW，WR，WM来表示这些男女间的喜爱状况，这四个n阶方阵如下定义：

（1）若第i个男子第r喜欢的是第j个女子，则定义：

$$MR(i, r) = j，\ MW(i, j) = r；$$

（2） 类似定义WR，WM。

方阵MR和MW表示男子对女子的喜爱情况，它们蕴含同样多的信息，可通过下面等式互相求得：

$$MW(i,\ MR(i,\ r)) = r\ ,\quad MR(i,\ MW(i,\ j)) = j\ 。$$

只是为了使用方便起见，同时采用这两个方阵。WR和WM 女子对男子的喜爱情况，有类似关系。可认为这些矩阵是算法的输入，算法的输出是n个有序对$(m_i, w_i)(1 \le i \le n)$, 表示$n$对夫妇，其中$m_i$是丈夫，$w_i$是妻子。不稳定婚姻可如下描述：若输出中存在两个不同的有序对$(m_i,\ w_i)$，$(m_j,\ w_j)$，使得下面两个不等式都成立：

$$MW(m_i,\ w_j) < MW(m_i,\ w_i),\ WM(w_j, m_i) < WM(w_j, m_j)$$

则该输出为不稳定婚姻，m_i与w_j是不稳定因素。反之，若输出中不存在这样的两个有序对，则输出为稳定婚姻。

事实上，该问题的解可表为n元组$(w_1,\ \cdots,\ w_n)$，它是n个女子的一种排列，其中w_i（$1 \le i \le n$）表示第i个男子的配偶。

用纯粹的穷举法来解决这个问题，需考虑$n!$种可能的婚配方式，检验其是否为不稳定婚姻。用回溯法求解该问题的基本思路是：当部分解为$(w_1,\ \cdots,\ w_{m-1})$，即前$m-1$位男子都确定了配偶并且这$m-1$对夫妇间没有不稳定因素的情况下，给第m位男子确定配偶，试图从其余女子（尚未确定配偶）集$\{1,\cdots,n\}-\{w_1,\cdots,w_{m-1}\}$中选择一位女子$w$匹配给$m$，形成有序对$(m,\ w)$，使得新的部分解满足稳定性，即是说，使得这$m$对夫妇间没有不稳定因素。

稳定性的检验：

（1）男子m是否会与其他已婚女子引起不稳定情况；

（2）女子w是否会与其他已婚男子引起不稳定情况。

为表征算法，再引入三个数组

integer $X(n)$, $Y(n)$

Boolean $Single(n)$

其中，$X(i)$表示第i位男子的配偶；$Y(i)$表示第i位女子的配偶；$Single(i)=true$ 表示第i位女子仍单身，$Single(i)=false$表示第i位女子已婚。

对于（1）设$MW(m,\ w)=r$只需检测m是否与他更喜欢的已婚女子

$$MR(m,i),\quad 1\le i<r$$

引起不稳定，记$pw=MR(m,i)$，即检验当$Single(pw)=false$时，是否有

$$WM(pw,m)<WM(pw,Y(pw))；（3）$$

同样对于（2），记$R=WM(w,m)$，检测w是否与她更喜欢的已婚男子

$$WR(w,i),\quad 1\le i<R$$

引起不稳定。记$pm=WR(w,i)$，即检验当$pm<m$时，是否有

$$MW(pm,w)<MW(pm,X(pm))\qquad（4）$$

如果（3），（4）均不满足，则说明可取$w_m = w$，否则，另觅人选。

稳定性的检测可写成如下过程

procedure $STABLE(r,\ m,\ w,S)$

//w是m的第r选择，w尚未婚配。若w可匹配给m，则S取$true$值，否则S取 $false$值.//

integer $r,\ m,\ w,\ i,\ R,\ pw,\ pm$

boolean S

$S \leftarrow true$

$i \leftarrow 1$

While ($i < r$) and S do

$pw \leftarrow MR(m,\ i)$

if not $Single(pw)$ then

$S \leftarrow WM(pw,m) > WM(pw,Y(pw))$

endif

$i \leftarrow i+1$

repeat

$i \leftarrow 1$

$R \leftarrow WM(w,\ m)$

While ($i < R$) and S do

$pm \leftarrow WR(w,\ i)$

if $pm < m$ then

$S \leftarrow MW(pm, w) > MW(pm, X(pm))$

endif

$i \leftarrow i+1$

repeat

end $STABLE$

在部分解$(w_1, \cdots, w_{m-1})$的情况下，能否确定w_m使部分解$(w_1, \cdots, w_m)$最终生成解由下述过程完成。

Procedure $TRY(m,\ q)$

boolean $q,\ q_1$

integer $m,\ r,\ w$

$r \leftarrow 0;\ q_1 \leftarrow false$

loop

$r \leftarrow r+1$

$w \leftarrow MR(m,\ r)$

if $not\ Single(w)$ then cycle endif

call $STABLE(r,\ m,\ w, test)$

if $test$ then

{

$X(m) \leftarrow w,$

$Y(w) \leftarrow m$

$Single(w) \leftarrow false$

if $m < n$ then

{

call $TRY(m+1,\ q_1)$

if not q_1 then $Single(w) \leftarrow true$ endif

}

else $q_1 \leftarrow true$

endif

}

endif

until q_1 or $(r = n)$ repeat

$q \leftarrow q_1$

end TRY

主程序初始调用$TRY(1,\ q)$，当$q = true$时说明有解：$X(n)$；当$q = false$时说明无解。可写成如下过程。

procedure $MRG(n)$

for $i \leftarrow 1$ to n do

for $r \leftarrow 1$ to n do

read$(MR(i,\ r))$

$MW(i,\ MR(i,r)) \leftarrow r$

repeat

repeat

for $i \leftarrow 1$ to n do

for $r \leftarrow 1$ to n do

read($WR(i, r)$)

$WM(i, WR(i,r)) \leftarrow r$

repeat

repeat

for $i \leftarrow 1$ to n do

$Single(i) \leftarrow true$

repeat

call $TRY(1, q)$

if q then write($X(i)$)

else print(“No solution”)

endif

end MRG

非递归算法如下：

Procedure StableMarriage(n)

for $i \leftarrow 1$ to n do

for $r \leftarrow 1$ to n do

read$(MR(i,\ r))$

$MW(i,\ MR(i,r)) \leftarrow r$

repeat

repeat

for $i \leftarrow 1$ to n do

for $r \leftarrow 1$ to n do

read$(WR(i,\ r))$

$WM(i,\ WR(i,r)) \leftarrow r$

repeat

repeat

for $i \leftarrow 1$ to n do

$Single(i) \leftarrow true$

repeat

$m \leftarrow 1$

$r(m) \leftarrow 0$

While $(m > 0)$ do

if $r(m) = n$ then

$\{m \leftarrow m - 1;\ Single(X(m)) \leftarrow true;$ Cycle$\}$

endif

$r(m) \leftarrow r(m) + 1$

$w \leftarrow MR(m, r(m))$

If not $Single(w)$ then Cycle endif

Call $Stable(r(m), m, w, test)$

If $test$ then

{

$X(m) \leftarrow w;\ Y(w) \leftarrow m;\ Single(w) \leftarrow false$

if $m = n$ then

{

Print ($X(1:n)$;

($Single(X(m)) \leftarrow true; m \leftarrow m-1$) //all solution//

(return) // only one solution//

}

else

{

$m \leftarrow m+1$

$r(m) \leftarrow 0$

}

endif

}

endif

Repeat

Print("No more solution!")

End StableMarriage

例 6.3 骑士巡游问题（也称为马步问题）$n \times n$的方棋盘有n^2个格子，马在棋盘上初始位置的坐标为(x_0, y_0)。按马步走的规则移动马。试图让马访问所有格子，且每个格子只访问一次。目标是编制算法给出成功的移动路线。当这样的移动路线不存在时，则要求算法给出“不存在移动路线”的结论。当$n=5$或$n=6$时的解如下（解不唯一）：

23	10	15	4	25
16	5	24	9	14
11	22	1	18	3
6	17	20	13	8
21	12	7	2	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29

该问题相当于在一类特殊图上找哈密顿路。

解：马步问题用回溯法来解决显然是可以的。用二维数组$H(n, n)$来

表示棋盘中格子的状态，$H(x, y)=0$表示格子(x, y)尚未被访问；$H(x, y)=i$（$1\le i\le n^2$）表示格子(x, y)在第i步被访问。现在来考察解决该问题的几个重要步骤：

1． 列出所有可能的动作

当马在格子(x, y)时，它下一步可能走的位置有 8 个，这 8 个

	×		×	
×				×
		马		
×				×
	×		×	

位置与(x, y)的横，纵坐标差是固定的，可用两个数组$A(8)$，$B(8)$表示

A	2	1	−1	2	−1	−2	−2	1
B	1	2	2	−1	−2	−1	1	−2

即下一步可走的位置为$(x+A(i), y+B(i))$（$1\le i\le 8$）

2． 检查这个动作可接受否

选定下一步要走的位置$(x+A(i), y+B(i))$后，检查该格子是否在棋盘上且尚未被访问，即检查是否满足：

$$1\le x+A(i), y+B(i)\le n,$$

$$H(x+A(i), y+B(i))=0。$$

3． 记录这个动作

当格子$(x+A(i), y+B(i))$满足条件时，给

$H(x+A(i), y+B(i))$赋值i，表示第i步访问了该格子。

4. 解完整否？

若$i=n^2$，则这是最后一步，棋盘上的格子都访问遍了；否则$i<n^2$，转向 1。

下面给出马步问题的主要过程 TRY

procedure $TRY(i,\ x,\ y,\ q)$

//第$i-1$步访问格子$(x,\ y)$的情况下，若能找到一空格作第i步访//

//问且最终能得到一解时，置$q=true$，否则置$q=false$.//

integer $u,\ v,\ k$

boolean q_1

$k \leftarrow 0;\ q_1 \leftarrow false$

loop

$k \leftarrow k+1$

$u \leftarrow x+A(k);\ v \leftarrow y+B(k)$

if $(1 \le u \le n)$ and $(1 \le v \le n)$ and $(H(u,\ v)=0)$ then

{

$H(u,\ v) \leftarrow i$

if $i<n^2$ then

{

call $TRY(i+1,\ u,\ v,\ q_1)$

if not q_1 then $H(u, v) \leftarrow 0$ endif

}

else $q_1 \leftarrow true$

endif

}

endif

until q_1 or $(k = 8)$ repeat

$q \leftarrow q_1$

end TRY

主程序要做的事情是

（1）给数组A及数组B置数；

（2）矩阵H清零；

（3）确定马的初始位置：$H(x_0, y_0) \leftarrow 1$；

（4）调用过程$TRY(2, x_0, y_0, Q)$；

（5）若Q返回$true$值则打印结果，不然打印失败信息。

总结

(1)回溯法的实质是检测所有可能的解，也就是穷尽所有的可能情况，从中寻求问题的答案，因此其时间复杂度与穷尽法一样。人工智能学应用回溯法求解时，要用启发信息, 启发函数来判

断每一个可能的动作，按其函数值的大小将所有可能的动作排成一列。换言之，把导致成功可能性大的动作排在前面，这样很可能很快就得到了问题的答案。

(2) 回溯法可以用来求一个解，也可用来求出所有的解。得出所有解的回溯法的一般算法如下：

```
procedure  TRY(i)
将所有可能的动作排成一列
loop
 从列中选择下一个动作，并将之从列中删除
 if 这个动作可以接受
  then
   {
    记录这个动作
    if 解不完整 then call TRY(i+1)
    else
     {
      打印出一解
      删去记录
     }
   endif
 }
endif
```

until 列空　repeat

end TRY

求所有解的算法看起来比求一解的算法还简单。回溯法求一解的一般算法如下：

procedure $TRY(i)$

将所有可能的动作排成一列

loop

从列中选择下一个动作，并将之从列中删除

if 这个动作可以接受

then

{

记录这个动作

if 解不完整 then

{

call $TRY(i+1)$

if not 成功 then 删去记录 endif

}

else 做成功标记

endif

}

endif

until 成功 or 列空 repeat

报告成功或失败

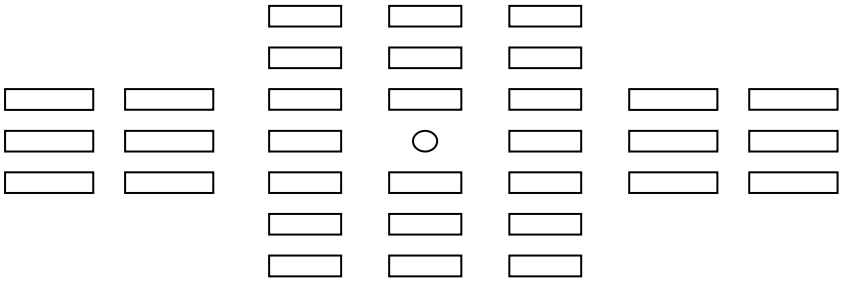
end *TRY*

作业（任选一题）

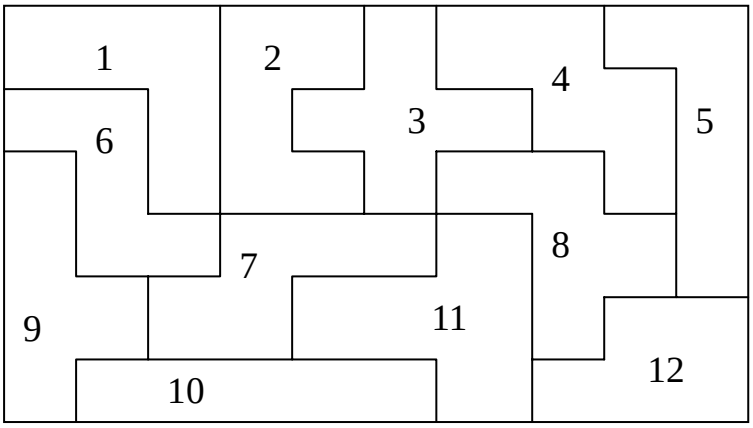
1. 分派问题一般陈述如下：给n个人分派n件工作，把工作j分配给第i个人的成本为$COST(i, j)$。设计一个回溯算法，在给每个人分派一件不同工作的情况下使得总成本最小。

2.（邮票问题）设想一个国家发行n种不同面值的邮票，并假定每封信上至多只允许贴m张邮票。对于给定的m和n值，写一个算法求出从邮资 1 开始在增量为 1 的情况下可能获得的邮资值的最大连续区域以及获得此区域的一种邮票面值方案。例如，对于$n=4, m=5$，邮资最大连续区域为 1 到 71，获得此连续区域的一种邮票面值方案为$\{1, 4, 12, 21\}$。

3. 假定有 32 张牌象下图那样摆在有 33 格的盘上，只有中心格空着。现规定当一张牌跳过邻近的一张牌到空格时，就将这张邻近的牌从盘上拿掉。写一个算法来找出一系列跳步，使除了最后留在中心格一张牌外其余的牌均被拿掉。



4. 设想有 12 个平面图形，每个图形由 5 个大小相同的正方形组成，每个图形的形状均与别的图形不同。下图中用上述 12 个图形拼成了一个 6×10 的长方形。写一个算法找出将这些图形拼成 6×10 长方形的全部摆法。



5. 假定要将一组电子元件安装在线路板上，给定一个连线矩阵 $CONN$ 和一个位置距离矩阵 $DIST$，其中，$CONN(i,j)$ 等于元件 i 和元件 j 间的连线数目，$DIST(r,s)$ 是此线路板上位置 r 和位置 s 间的距离。将这 n 个元件各自放在线路板的某位置上就构成一种布

线方案，布线成本是$CONN(i,j)\times DIST(r,s)$乘积之和，其中元件$i$放在位置$r$，元件$j$放在位置$s$。设计一个算法找出使布线总成本取最小值的一种布线方案。

6. 假设有n个要执行的作业但只有k个可以并行工作的处理器，作业i用t_i时间即可完成。写一个算法确定哪些作业按什么次序在哪些处理器上运行使完成全部作业的最后时间取最小值。