前言

本书是《算法艺术与信息学竞赛》的学习指导。所谓学习指导，是告诉读者学什么，如何学。算法包罗万象，很难在一本书中的篇幅覆盖很多内容的通知对每个知识点进行细致讲解，因此本书更多的是作为一本导引、工具书、手册或者学习大纲，给读者以宏观上的学习指导。和原书《算法艺术与信息学竞赛》相比，本书的特点有：

- 大量知识讲解。
- 更多循序渐进的习题。
- 重要算法的源代码。

从知识结构上看，本书的覆盖面比原书更广，补充了原书没有涉及到在知识点，包括计算理论中的NP完全理论和图灵机的基本概念、数据结构中的伸展树、Treap，左偏树、二项堆、Fibonacci堆、数论中的指数和原根、分解因数的快速算法、数值计算中的高斯消元法和FFT、组合游戏论初步、更多序列经典问题和线段树、后缀数组等数据结构的应用、树更多经典问题、多模式串匹配算法、后缀树构造的Ukkonen算法、后缀数组构造的Skew算法、更加详细的强连通分量/双连通分量算法、最大流和最小费用流算法、二分图和任意图的最大基数匹配算法和最大权匹配算法、稳定婚姻问题、线性规划在网络优化中的作用、向量代数基础、多边形剖分算法、平面剖分、半平面交、三维凸包、Voronoi图和直线排列的构造算法、几何对偶性的应用、Minkowski和与简单运动规划问题等。

从题目上看，本书的题目集中在习题部分，这样保证知识讲解部分相对完整和纯粹，也避免了用过多篇幅叙述和具体题目相关的内容。这些题目选择范围更广，难度搭配也更加合理，且包含了常见的小技巧，和原书那些巧妙但难以理解的题目比较起来更适合于初学者入门与提高，也为深入阅读原书打下了基础。

刘汝佳周源周戈林

2005年10月15日

目 录

插图目录

表格目录

第1章　概述

本章是原书的“前篇”，也是学习的起点。本章将介绍计算机、算法、问题求解步骤和程序设计竞赛，以及C++语言和数据结构概念，旨在给初学者一个整体的概念。建议有一定基础的读者也浏览一下这些内容，相信同样会受到启发。

1.1　认识计算机

在当今科技社会中，人们慢慢的接受了这样一个事实：电脑可以解决问题。但电脑究竟能解决什么样的问题，用什么方法解决的？在搞清楚这个问题之前，我们应当首先知道这样一个事实：通常意义下的计算机是没有思维的机器，它并不能自主解决问题，而只能机械地执行指令，而这些指令是人设计出来的。机器能解决什么样的问题，怎样解决，需要靠人。机器的优势是速度快、容量大、严格而精确。人应当充分发挥计算机的优势，避免劣势，让计算机用*正确的方法*做*正确的事*。哪些事情是“正确的事”？这取决于计算机的特点。怎样的方法是“正确的方法”？这正是《算法艺术与信息学竞赛》讨论的内容。直观的说，解决问题的方法称为**算法(algorithm)** 。

1.1.1　计算机的优势

计算的第一个优势：速度。很多时候人不是不知道怎么解决问题，而是嫌方法太麻烦。有这样一道题目：用1、2、3、4、5、6、7、8、9九个数字拼成一个九位数（每个数字恰好用一次），使得它的前三位、中间三位、最后三位的比值是$1:2:3$。例如192384576就是一个合法的解，因为$192:384:576=1:2:3$。

有一个办法是奏效的：列举出所有可能的九位数123456789，123456798，123456879，...，987654321，一个一个检查是否符合条件。理论上来说这是可行的，可几乎没有人愿意这么做，因为这样的九位数有362880个，即使每秒检查一个数（这已经是很快的速度了！），也需要100个小时。计算机就不一样了，它的运算速度很

快，同样是这个“笨方法”，人需要算100小时，计算机只需要不到0.1秒种。

下面是解决这个问题的 C++ 语言程序，有兴趣的读者可以运行一下试试。不会 C++ 语言也不要紧，至少可以试着从程序中体会思想。其中d中保留着九位数的各个数字（可以看出，第一个九位数是123456789）；x、y、z代表九位数的前三位、中间三位和最后三位；函数 next_permutation 的作用是计算出下一个九位数，而程序核心语句的意思是“如果y等于x的两倍，且z等于x的三倍，那么把x、y、z打印到屏幕”。

```
#include<iostream>
using namespace std;

int main()
{
    int i, x, y, z;
    int d[9]= {1, 2, 3, 4, 5, 6, 7, 8, 9};
    do{
        x = d[0] * 100 + d[1] * 10 + d[2];
        y = d[3] * 100 + d[4] * 10 + d[5];
        z = d[6] * 100 + d[7] * 10 + d[8];
        if(y == x * 2 && z == x * 3)
            cout << x << y << z << endl;
    }while(next_permutation(d, d + 9));
    return 0;
}
```

顺便说一句，答案有四个：192384576、219438657、273546819、327654981。这个例子告诉我们：计算机和人相比有速度优势。

计算机的第二个优势：记忆。在计算机未损坏的情况下，它所存储的内容不会消失、不会改变，永远得到精确的保存，而且能够轻易的读写和复制。这一点人是做不到的。人为什么要在合上书之前夹一枚书签？听课时为什么要做笔记？都是避免把曾经记住的东西忘掉。“记忆”分成两部分，用计算机术语来说，“记”是**写(write)**，“忆”是**读(read)** 。

计算机内部有很多可供读写的存储单元，而读和写就是存储单元的基本操作。存储单元越多，记忆容量就越大。人们常说的内存、硬盘、光盘、U盘等都是存储器，128M、80G等都是指的容量。容量越大，可以存的东西就越多。

计算机的每一个存储单元都有一个“编号”，即**地址(address)** ，只要指定了地址，就可以对此地址所对应的存储单元进行读写操作。“读”操作不会影响到该存储单元的数据，而“写”操作将会使该存储单元存入新的东西，以前的内容永远消失。需要特别注意的是：每一个内存单元的大小是固定的，你不能无限制的往里面存数据。正规的

说，一个**字节(byte)**（它是内存的常用单位）只能放0 255之间的整数，那么你就不能往里面放256。当然，你可以把256成两个128，放到两个字节里。所以在理论上，只要你的内存足够大，任意大（注意，不是无穷大）的整数都可以保存。

计算机的第三个优势：精确。神速计算加上永恒记忆，并不是计算机强大能力的全部。有一些任务，不需要神速计算，也不需要永恒记忆，可是如果不借助于工具，人就是做不好。“失之毫厘，谬之千里”说的就是这样的情况。在地上划一根长长的线段，然后闭上眼睛尽量沿着线走，你能笔直的走到线的尽头吗？也许你感觉你走得非常直，可是在走过相当长的距离后，几乎所有人都会走歪。看着你的手表，当秒针穿过0秒时闭上眼睛，默数5分钟。睁开眼睛后对照你的手表，真的是5分钟吗？人做事受感觉的影响，而感觉往往是不准确的。

图 1.1: Koch雪花曲线

图 1.1虽然好看，却很难手工画出来。这是著名的Koch雪花曲线的一部分，是**分形(fractal)** 的一种。这幅画的特点是部分和整体相似。不信你把它的局部展开，看到的是整体。即使你拥有Koch雪花曲线完整的数学定义，也无法用手画出来（它的细节是无穷的！）。哪怕只是画一个“看起来像”的曲线，也是很难的。而用计算机却很容易做到。以下就是画出Koch雪花的PostScript程序。

```
/kochR {
  2 copy ge { dup 0 rlineto }
  {
      3 div
      2 copy kochR    60 rotate
      2 copy kochR  -120 rotate
      2 copy kochR    60 rotate
      2 copy kochR
  } ifelse
  pop pop
} def
```

```
0   0 moveto   27 81 kochR
0  27 moveto    9 81 kochR
0  54 moveto    3 81 kochR
0  81 moveto    1 81 kochR
stroke
```

计算机可以严格按照指令工作，因此只要有了可操作的数学定义，这类事情用计算机完成很合适。这些例子共同说明了同一个问题：计算机是精确的。

1.1.2 限制和解决方案

计算机的速度快，并不是说可以给它无限大的工作量。同样的问题，解决方法不同，计算机的工作量也不同。为了计算$1+2+3+\ldots+n$，可以让计算机做$n-1$次加法，也可以让电脑直接计算$n(n+1)/2$，即1次加法，1次乘法和1次除法[1]。刚才两种算法（是的，“连加$n-1$次”和“计算$n(n+1)/2$”就是两个算法！）的差别告诉我们：同一个问题可以有多个算法，有的运算量大，有的运算量小。具体使用哪个算法由人说了算，计算机只会按部就班的执行。

计算机的容量大，也不是说可以要它保存无限多的东西。同样的问题，解决方法不同，需要保存的东西也不同。小学老师可以让我们背“九九乘法表”，然后用它算12×34或者56×78这样的两位数乘法，同样也可以让我们把所有两位数乘两位数的积全部背下来。这样的话，12×34 根本就不用算了，直接可以背出来。速度提高了，可是需要记更多的东西。小学教育的选择是：少记一点，宁愿可每次多花点时间算。计算机也一样，不同的算法可能使用不同大小的内存，不过速度也许会有差异，这需要权衡。

计算机是精确的，缺少人类所具有的模糊能力，没有专门的认知系统，也不会产生灵感或者闹情绪。精确可以带来好处，也可以引起麻烦。计算机能执行的指令，简单的来说就是四则运算、逻辑运算、存储器操作和输入、输出等。它很难解决如“这幅画漂亮吗？”这样模糊的问题。人可以回答，但却很难用精确的语言描述出他为什么这样回答。最后需要提醒读者注意的是：有很多算法对于人来讲是直观的，但是转化成计算机可以执行的运算指令却需要花一番功夫。例如，要求一个奇形怪状的容器能装多少水，计算机算起来比较麻烦（而且充满了数学味，不直观），而人却可以简单的做个实验得到结果。要把一个东西塞到瓶子里，人可以往各个方向使劲塞，而计算机

[1]除以2可以用移位实现，这样更快。

却很难使用这种“没头没脑”的“算法”。一句话：人和计算机是不同的，这个差别不只是速度、记忆容量和精确性的问题。请读者时刻注意这一点。

小测验

1. 计算机相对于人的三大优势是什么？举例说明
2. 不同的算法在速度和内存占用上可以有怎样的差异？举例说明
3. 计算机解决问题的方式和人有差别吗？举例说明

1.2 问题、算法及其分析

速度快、容量大和结果精确是计算机的硬件条件。为了解决问题，只有硬件条件远远不够。人们需要针对各种问题设计不同的算法，并把算法转化为计算机可以直接运行的**程序(program)** 。

1.2.1 问题实例

算法的好处在于通用性。当针对一个**问题(problem)** 设计出算法后，该问题的多个**实例(instance)** 都能被解决。换句话说，一个加法程序不仅能计算$1+1$，也能计算$100+100$和$1000+1000$。另一方面，如果一个算法只能求解问题的一个实例，它的能力是十分有限的。一般情况下只需要把这个实例和对应的答案记录下来，算法本身就没有意义了。

一个问题通常有多个输入参数，当各个输入参数都确定时，该问题的实例也确定下来了。问题求解结束以后，答案放在输出参数里。可以给一个问题下一个类似这样的定义：

问题：正整数加法
任务：计算正整数a和b的和c
输入：a, b。（a和b是不超过10000的正整数）
输出：c。（c是a和b的和）

有了问题定义，算法设计者很明确的知道了任务，而用户也知道如何使用该算法（即如何提供输入参数，输出参数代表什么含义）。一个不完整的例子是：

问题：找最大元素
任务：找出n个数中的最大元素

这个算法该如何设计、如何使用呢？不知道。如果算法的设计者认为输出应当是“第几个数最大”（输出序号），而使用者认为输出的是“最大数等于多少”（输出数值），那么就会出现问题。一个好的问题定义还应当包括一个或多个输入输出的例子，进一步澄清任务和输入、输出规定，例如：

问题：全等判定
任务：判断n个数是否全部相同。
输入：n个数
输出：如果全部相同，输出Yes，否则输出No
样例输入1：5 3 3 6
样例输出1：No
样例输入2：7 7 7 7 7 7 7
样例输出2：Yes

相信读者一定明白这个问题的任务和输入、输出格式了。在下一小节中，我们试着求解这一问题。

1.2.2 算法描述

刚才的“全等判定”问题显然不是一个困难的问题。如果是人来做，只需要“扫一眼”就可以了。是不是可以写这样的算法呢？

算法：扫一眼算法
把所有数“扫一眼”。如果刚才曾发现两个数不一样，输出No，否则输出Yes

这样的算法是无法被计算机执行的（但是被很多人“执行”！），因为这个“算法”里的话十分模糊，“扫一眼”、“发现”等词语都没有被精确定义（例如，有可能眼花了没发现），计算机自然无从执行。

这个“算法”是计算机无法执行的，那么哪些算法是计算机可以执行的呢？这个问题并不是很容易回答，需要学习计算模型或者一门通用程序设计语言（如 C++ 或者 Pascal），我们把这个问题留在本章后面和第二章中详细讨论。这里给出一个看起来正确的算法：

算法：相邻枚举算法
从前到后考察每一个输入参数，如果它和紧跟其后的数不相等，输出No
如果始终没有发现不相等的情况，输出Yes

算法的思路是：如果每个数都和“后边”的数相等，那么所有数相等，否则不全相等。思路没有问题，但细节存在不少问题：一、如果很多数和它后边的数都不相等，会输出很多“No”，不符合输出规定。二、最后一个数的“紧跟在它后边的数”是什么？没定义。

这些问题说明：把算法写成计算机可以准确执行的形式并不容易。下面是用修改后的枚举算法写成的 C++ 程序。虽然很抽象，但是它能清楚的说明所有细节。初学的时候可以像学习英语一样学习 C++，即先培养“语感”，在适当的时候再严格的学习语法和其他。

下面的程序中，vi 代表整数序列，$v[i]$ 代表序列 v 的第 i 个元素。第3行中的“for”语句对应于“从前到后”的循环。换句话说，语句4被执行了很多遍。它被称为“循环体”。“!=”运算符的意思是“不等于”。这个程序并不是完整可执行的程序，而只是一个函数。函数的输入参数就是整数序列 v，输出是 bool 值（即非真即假的值），true 表示“真”，false表示“假”。

```
#include<iostream>
#include<vector>
using namespace std;

typedef vector<int> vi;

bool all_same(vi v)
{
    for(int i = 0; i < v.size() - 1; i++)
      if(v[i] != v[i+1]) return false;
    return true;
}

int main()
{
    vi v;
    int i;
    while(cin >> i)
        v.push_back(i);
    cout << all_same(v);
    return 0;
}
```

没有完全明白也没关系，现在只需要培养“感觉”。

小测验

1. 算法为什么需要有通用性？
2. 一个好的问题定义应该包含哪几个方面？
3. 为什么算法描述的各个步骤需要精确定义？

1.2.3 算法分析

算法设计出来以后，应该先进行分析。一般来说，需要估计算法的时空开销，即：需要运行多长时间？需要多大内存？这是个很实际的问题，因为如果一个程序需要运行100年，或者需要很多很多内存，那么就算它能得出正确的答案也没有用，因为我们等不了这么久，或者买不起这么多内存。

但问题在于编程之前我们不可能精确的知道一个算法要运行多长时间，因为它还没有被转化成程序。用不同的方式写成程序，运行时间显然不一样；同一个程序在不同的计算机上运行，时间也不一样。同一个程序在同一个计算机上运行，时间也不总是一样（比如，有其他程序同时运行）。我们的原则是：一、忽略机器和程序实现，只分析算法本身；二、忽略次要因素，保留主要因素，以得到简洁的结果。在多数情况下，我们把程序运行时间写成像$O(n)$，$O(n^2)$这样简洁的表达式[2]，其中n是输入规模，即影响程序运行时间的最主要参数。这样的表达式只跟输入规模有关，不反映具体运行时间（也无法反映，因为它和太多因素有关），因此称为**渐进时间复杂度(asympotic time complexity)**。

> 输入规模：主要影响算法时间的输入参数。比如当程序需要输入n个数时，n越大运行时间往往越大。这时我们一般把n作为输入规模。
> 渐进时间复杂度：输入规模的一个简单表达式，它往往被化成非常简单的形式，目的在于反映算法的时间效率。不同渐进时间复杂度的算法往往时间效率相差非常大。

下面我们分析全等判定问题的枚举算法。假设一共有n个数，如果把“判断一个数和紧跟它后面的数是否相等”看成一个基本操作，那么枚举算法最多执行$n-1$次操作。

[2]也许读者会觉得$O(n)$这个符号看起来很奇怪，但这不要紧，我们将在后面详细介绍它的含义，并引出另几个类似的符号。

也许有的机器判断得快，有的机器判断得慢，但是“最多执行$n-1$次操作”是确定的，不随着基本操作的时间而改变。假设一台计算机执行此算法，$n=100,000,000$的时候用了约1秒钟，那么当$n=200,000,000$的时候需要多长时间呢？答案是：约2秒钟。我们不需要计算每次操作需要多长时间，反正操作的数目增加了约一倍，因此时间花费也增加约一倍。我们称这样“n增加一倍时操作数目也增加约一倍”的算法为**线性时间算法(linear-time algorithm)**，或者$O(n)$的算法。

> 线性时间算法：输入规模增加一倍，则运行时间约扩大一倍。也称$O(n)$时间算法。

顺便说一句，由于n越大，需要的时间越长，我们称n为输入规模，它主要影响了算法的时间（因为n增大一倍时时间也约增加一倍）。

下面我们把问题变一下：判断n个数是否两两不等，即：

> **问题**：全异判定
> **任务**：判断n个数是否两两不相同。
> **输入**：n个数
> **输出**：如果两两不同，输出Yes，否则输出No
> **样例输入1**：5 3 3 6
> **样例输出1**：No
> **样例输入2**：1 3 5 4 2 6 7
> **样例输出2**：Yes

问题只把“相同”改为了“不同”，那么算法是否也只需要把“相同”改为“不同”呢？这是不对的，因为在序列“1 2 1”中，每个数和紧随其后的数都不相同，但是却出现“隔了一个数”的两个1是相同的。怎么办呢？只好把算法变一下：

> **算法**：完全枚举算法
> 从前到后考察每一个输入参数，如果和后边的某个数相等，输出No并退出
> 如果始终没有输出No并退出，输出Yes

这样一来，算法倒是对的，可是时间如何呢？比较一个数和紧随其后的数只需要一次比较，而比较一个数和其后的所有数是否相同，最多需要$n-1$次。全部加起来，

你会发现最多需要约$\frac{1}{2}n^2$次比较。这意味着：n增加一倍时操作数目将变为原来的4倍！前面所说的那台计算机在1秒钟之内可以解决10^8个数的“全等判定”问题，却只能解决约20000个数的“全异判定”问题。我们称这样“n增加一倍时操作数目变为原来的4倍”的算法为**平方时间算法(quadratic-time algorithm)**，或者$O(n^2)$的算法。

> 平方时间算法：输入规模增加一倍，则运行时间约变为原来的4倍。也称$O(n^2)$时间算法。

顺便说一句，以后我们将学到一个$O(n\log n)$的算法，它的速度比线性算法略慢，可在1秒内解决至少1,000,000个数的“全异判定”问题，比刚才的$O(n^2)$完全枚举算法快得多。

和刚才介绍的的$O(n)$, $O(n\log n)$, $O(n^2)$算法相比，有一类算法要差得多，例如$O(2^n)$的算法，它只能解决n不超过20的问题[3]。怎么办呢？如果买一台更快的计算机，例如比刚才那台快十亿倍（如果买得到的话）的计算机，也只能解决n不超过40的问题。由于在渐进时间复杂度表达式里n在指数位置，我们称这样的算法为**指数时间算法(exponential-time algorithm)**。和指数时间算法对应，如果渐进时间复杂度表达式为输入规模的多项式函数，我们说该算法是**多项式时间算法(polynomial-time algorithm)**。由于非多项式函数也不一定是把输入规模放在指数位置，指数时间算法的范围还包括其他比多项式“增长快”的函数（如阶乘$n!$）。这类算法往往时间效率低下[4]。

> 多项式时间算法：渐进时间复杂度表达式为输入规模的多项式的算法。这类算法往往效率较高，但有例外。
>
> 指数时间算法：在渐进时间复杂度表达式中输入规模在指数位置的算法，还包括其他增长速度严格比多项式快的算法。这类算法的时间效率往往较低。

另一种方法是重新设计算法。如果能找到一个$O(n^2)$算法，不用买新计算机也可以解决n上万的问题，如果还能找到线性时间算法，还可以做得更好[5]。这个例子再次告诉我们：同一个问题可以有多种算法，它们的时间消耗可以不一样。如何设计出好的算法？这就是《算法艺术与信息学竞赛》所讨论的主题。

[3]细心的读者一定看出了：这里的20并不是绝对的，它取决于机器运算速度和隐藏在$O(2^n)$里的系数，这里不做深究。

[4]也有例外，这里不深究。

[5]不要觉得这样的情况很少见。如此明显的算法改进在本书中常常出现。

本节的概念性比较强，但目的只是让读者有个直观概念，因此并没有给出每个复杂度类的严格数学定义。读者可以在本书后面章节找到更为完整的讨论。

小测验

1. 用自己的语言描述“全等判定”问题和“全异判定”问题及它们的算法（各一种）
2. 什么是时空开销？为什么要估计算法的时空开销？
3. 用什么方式衡量算法的时空开销？为什么不用具体的运行时间？

1.2.4 难解问题

通过上一节的学习，我们感觉到：全等判定和全异判定看起来是两个很接近的问题，但实际上难度是不一样的。前者可以很轻松的设计出线性时间算法，而全异判定问题只能设计出略差一点的$O(n \log n)$算法，这个算法还不是那么容易解释清楚。

在生活中，我们通常是主观的比较两个问题的难易程度。由于知识、能力不同，可能对于同样两个问题，有人觉得第一题较难，有人觉得第二题较难。而对于计算机来说，我们可以（并非必须）用目前已知的最好算法来比较。有些好算法设计起来非常困难，描述起来也复杂，但是一旦完成，计算机可以永远记忆，并且不断的被用来解决新的问题实例（一次写成，终身受益！）。当我们看到计算机在解决问题中表现出来的良好性能时，大多数人都会忘记设计算法时的困难。对于计算机来说，有的问题是容易的，有的问题是难的，有的问题根本不可解！不严格地讲，难解的问题是“目前还没找到有效算法，甚至很可能根本不存在有效算法”的问题。这里的有效算法在数学上有严格的定义：它在速度、空间、结果等方面都有要求。

有一个商人，需要在N个城市里去推销他的商品，每两个城市都可以相互直达（例如坐火车、飞机、船等）。商人拿到了每两个城市间旅行的价格表，如何才能花最少的钱，从商人自己家乡所在的城市出发，经过所有城市以后再回到自己的家乡？在数学上，这是个NP完全问题（一种难解问题，目前您不需要了解它的定义），可是商人总不能因为数学上的难解性就放弃了经商的念头吧。商人总是会采取某种方案的，因为：

理由一：我看了价格表，所有价格都一样，所以怎么走都没区别嘛。

理由二：不一定总需要花费最小的方案。只要相差不太多，例如几百元钱之

内，都可以。

理由三：别想那么多了，随便走就是了，因为钱是可以报销的。

不管怎样，商人的理由提示我们可以从几个方面放宽“好算法”的标准：首先，也许只需要解决问题的一个实例就可以了，不需要设计通用算法；第二，也许结果不需要绝对最好，只需要比较好就可以了；最后，也许在某些时候加入随机因素会让算法有不错的表现。这三个方案在原书中有所体现，由于篇幅关系，这里不再赘述。

小测验

1. 有哪些方法比较两个问题的难易程度？
2. 什么是难解问题？它和已知算法本身的复杂程度有关吗？为什么？
3. 为了求解数学上的难解问题，通常有哪些方法可以尝试？

1.3 问题求解与程序设计竞赛

通过前面两节介绍，相信读者已经对计算机和算法有了一定的认识。这一节介绍问题求解的一般步骤和程序设计竞赛，希望能引起更多人对它们的关注。

1.3.1 问题求解周期

从一个实际问题的提出，到计算机可以执行的程序代码的完成，可以看作“问题求解”的一个周期。

问题简化 大多数实际问题涉及到的因素很多，在求解之前必须经过简化，得到问题的**原型(prototype)** 。这个原型应当是没有歧义的，可以用前面介绍的“任务-输入-输出-样例”标准方法加以定义。事实上，这就是程序设计竞赛所采用的标准定义。本书所讨论的问题都是以原型的形式出现。问题简化的例子是“给出全国铁路网中任意一班车的价格和两个城市A和B，求从A到B花费最小的旅行路线”。

建模与分析 问题的原型简洁的叙述了问题的条件、限制和求解目标，但是没有表明问题的本质。很多表面上看起来完全不同的问题具有相同的本质。数学建模本身就是一门科学，有很多这方面的著作，但是它们的侧重点和本书有差异。刚才的例子数学建模的结果是“给正权有向图G和两点A和B，求A到B的最短路”。同一个问题可以有多种方法建立模型，模型求解的难度有差异，就好比同一个问题有多种算法可以求解，它们的时空花费与解的精确程度有差异。有的模型非常巧妙（艺术！），原书中

就有不少这样的例子。得到数学模型以后，只要不是简单到可以直接求解或者套用经典模型的程度，一般需要进行模型分析，得到初步结论。很多领域的很多经典模型前人已经做过详细而透彻的分析了，学习“问题求解”应当尽量的积累这样的经典模型的经典算法。刚才的“正权图上的单源最短路问题”是一个经典模型，可以用dijkstra算法求解。这个算法在原书的“图论算法与模型”一节有介绍。另一方面，如果模型是新的，需要继续进行算法设计，这一步往往比建模更有挑战性（同样是艺术！）。

算法设计 由于最终需要由计算机执行，模型分析的最后要变成算法。从数学角度不用进一步分析的问题往往还需要进行算法设计，用最适合计算机的方式进行求解。算法设计、数据结构、计算数学都能派上用场，目标只有一个：设计出最适合计算机求解的算法。原书在分析问题时把终点定在了“算法分析”，而假定读者有足够的能力把算法转化成程序。为了方便更多的初学者，本书把周期继续延伸，给出几乎所有算法的核心程序。这样的好处在于：一、通过实例介绍把算法转化成程序的方法和技巧；二、方便读者做参考甚至直接使用（不推荐！）；三、允许读者通过代码反过来对算法进行更细节的学习。因此，本书中包含了问题求解周期的最后一步骤：

程序设计、测试与微调 如果算法设计充分，大多数算法在程序设计时都主要是“体力活”。有一些算法描述并未包含很多实现细节，而这些细节可能极大的影响程序的效率和空间消耗，需要在程序设计时仔细考虑（还是艺术！）。算法和代码的效率有时候并不能只靠理论分析，而需要用实现测定，一些待定参数或者某些代码也需要根据实验结果微调，这些内容，广义的来说也属于程序设计步骤应该完成的工作。这些内容，原书很少有提及，而这本《学习指南》将为它们花不少篇幅。这样的改变大大的提高了本书的实用性，而科学性不会受到影响。

1.3.2 程序设计竞赛：问题求解实践

实际的问题往往复杂且涉及到很多非计算机科学的内容，因此本书只考虑简化的问题求解，即从有精确定义的、可客观评价（不会出现“用计算机随机生成美妙的音乐”这样难以用客观准则界定结果的优劣程度的问题）的原型开始，以实际程序（或数据）为成品，用事先设定的测试数据作为程序的标准。恰好，两大著名程序设计竞赛**国际中学生信息学奥林匹克竞赛IOI(International Olympiad in Informatics)** 和**国际大学生程序设计竞赛ACM/ICPC(ACM International Collegiate Programming Contest)** 的比赛内容就是用计算机进行问题求解。TopCoder算法组比赛内容也

是问题求解。如果您是某项比赛的选手或者教练，那么本书是您很好的参考；如果不是也没有关系，因为：

- 如果您希望进行学术研究，程序设计竞赛是很好的试金石。算法设计能力本身就是很多学术研究的基本功，而且很多学术研究的雏形或者核心曾经出现在竞赛题中，解决这些题目可以看成是简单的学术实践。

- 如果您希望学习程序设计或者算法，程序设计竞赛是最好的教材，它融合了数学建模、算法设计和程序设计的各个方面，其综合性和实用性远非传统教科书中的“课后习题”甚至所谓的“项目设计”所能及。

- 如果您只是为了提高自己的科学素养，那程序设计竞赛将是最神奇的催化剂。算法和题目本身是有趣的，而其中蕴涵的哲理和通用性广的方法与技巧将让您受益终身。不管你是专业是什么，计算机能给你现代化的包装。

如果您想进一步了解竞赛，这里是一些一般性的信息：

- 竞赛形式是限时解决若干题目。例如IOI比赛分两天进行，每天5小时，各解决3道题目；ACM/ICPC比赛只有一场，5小时解决8 10道题目。

- 通常的竞赛题目以“任务-输入-输出-样例”的标准形式给出，要求提供源程序。从原型得到程序的中间步骤（即上小节所说建模、模型分析、算法设计等）不需要提交。分数完全由程序决定。

- 评分方式为完全客观的黑箱测试法。裁判事先准备好（但不公布）多组符合题目的输入输出要求的测试数据，把输入提供个选手程序，测试输出是否正确。由于问题的任务和输入输出格式被无歧义的精确定义好，因此能客观确定任何一份输出是否正确。

- 考查算法的时空承受能力。前面说过，同一个问题有不同的算法，不是每个算法都能得到同样的分数。每道题目有时间限制和空间限制，如果在规定的时间内程序没有运行结束，或者使用了超过规定范围之内的内存，你的程序将不能得分（即使答案是正确的！）。

以上是共同点，但不同的竞赛也有不同的特点。

ACM/ICPC 组队赛，三个队员使用同一台电脑解题。每道题目只有两种结果：正确或者错误，“半对半错”或者“差点正确”都被认为是不正确的解答（要求严格！）。比赛结果是实时(real-time)的，即任何时候都可以把某题的程序提交给裁判，裁判告诉你正确还是错误。如果错误，可以继续修改后提交。比赛结束后，按照解决问题的多少排名；如果解决问题的数目相同，按照解决问题的总时间从小到大排序。总之，在正确的情况下，题目做得越快越好。这一规则也适用各大在线题库，如UVa，Ural，SGU，ZOJ，POJ等。ACM/ICPC比赛相对于IOI来说激烈很多，比赛场上往往会发生很多戏剧性的变化，这对选手的决策能力和心理素质都是极大考验。

IOI 单人赛。每道题目分为多组数据，每组数据单独评分。换句话说，不完美的解答一道题目也可以得到50%、80%甚至95%的分数，而在ACM/ICPC中，任何一点错误都是不允许的。IOI的迷人之处在于它可以根据优劣程度进行相对评分（符合商人的理论），可以与另一个程序进行交互（下棋、玩游戏），还有一种题目类型只需要解决一个问题的若干实例，可以写多个程序，还可以人机合作解题，大大提高了题目的灵活性。这一规则也适用于NOI（全国青少年信息学奥林匹克竞赛）、联赛NOIP和其他国家地区的比赛如美国的USACO，波兰的POI，中欧的CEOI等。

本书同时讨论ACM/ICPC和IOI竞赛的题目，读者应能区分它们的区别。但不管参加哪个比赛，其他比赛题目都是很有借鉴价值的，其中的理论、方法和技巧是共通的。

小测验

1. 描述问题求解周期的几个步骤？原书关注哪些步骤？本书有何不同之处？
2. 结合自己的实际情况，解释学习本书有哪些好处。
3. 在Internet上寻找ACM/ICPC、IOI和TopCoder的信息和比赛题目，它们和本书的关系如何？

1.4 C++语言介绍

本节介绍C++语言和数据结构。可以有两种方法阅读本章。第一种方法是先经过一段时间的学习，了解C++语言和数据结构的基本内容并能独立完成一些比较简单的

程序（会编码、调试、测试），然后阅读本章。这样做的好处是有一定实践经验的积累后，能更好的理解本章。第二种方法是先浏览本章，领会主要思想，然后再专门学习本章没有提到的细节内容，在实践中不断体会本章。这样做的好处是进行正规学习时比较有方向感，不会迷失在无谓的细节中。两种方法各有优缺点，读者可以根据情况选用。即使对于语言和数据结构非常熟悉的读者，也建议浏览本章，很可能会有意外的收获。

C++是一种支持面向对象的程序设计语言，但在进行问题求解时我们可以不使用它的面向对象特性，因为我们的重点是实现算法。本书中代码的主要意图是用清晰的方式向读者展示算法细节和实现技巧，因此工程上的要求退居其次。但读者应该时刻注意：在实际软件编写的过程中，应充分重视可维护性、重用性、移植性等的要求。

学习一门语言有三个层次：概念、使用和原理。本章的语言回顾着眼于概念和使用，而对原理很少加以叙述，读者应能避免编译器相关的细节问题（下文中将提到一些这样的问题并说明如何避免）。

1.4.1 第一个C++程序

语言入门的一个不错的选择是从实例开始学习。下面给出了一个简单的例子，它的作用是输入两个数，输出它们的和。

```
#include<iostream>
using namespace std;
int main()
{
    int a, b;
    cout << "Please␣input␣two␣numbers:␣";
    cin >> a >> b;
    cout << a << "␣+␣" << b << "␣=␣" << a + b << endl;
    return 0;
}
```

一个典型的执行过程是：

第一步：程序打印出提示信息：“Please input two numbers”

第二步：用户用键盘输入“3 4”，并按回车键。

第三步：程序打印出结果：“3 + 4 = 7”

这样使用程序的方法有一个重要的缺陷：不适合输入海量数据。如果需要计算一万个数的和，就需要用键盘敲入一万个数。也许读者会问：“如果不用键盘一个个敲，

程序还能从哪里弄来这一万个数？”答案是：文件。事先在文件里保存好程序的输入数据，然后运行程序得到输出结果。

文件中的数据可以是其他程序的计算结果，可以是通过传感器搜集到的，可以是从Internet上获得的，当然也可以是用户用键盘敲进去的。即使是最后一种情况，事先把数敲进文件也比运行时敲给程序输入好得多（想一想，为什么）。

既然是这样，输入提示信息就失去作用了，而输出也只需要说明结果是几。输入从文件 sum.in 读取，而输出写到文件 sum.out 中，像这样：

```
#include<iostream>
#include<fstream>
using namespace std;

ifstream fin("sum.in");
ofstream fout("sum.out");

int main()
{
    int a, b;
    fin >> a >> b;
    fout << a + b << endl;
    return 0;
}
```

未加说明的情况，本书的程序都是这样的结构：文件输入，文件输出。为了测试程序是否正确，我们一般提供一个或多个**标准输入文件(standard input file)**，运行程序并判断输出是否正确。

这样的“给输入-运行程序-判输出”的测试方法称为黑盒测试法。黑盒测试法的特点是：不关心程序结构，只要能得到正确的结果就可以了。显然，对程序的评价很大程度上依赖于标准输入文件。如果对于上面的题目，只提供了一个“0 0”的测试数据，那么即使写一个求两个数差或者乘积的程序也会被认为是“正确”的！解决的方法是：使用多组测试数据。比如除了那个“0 0”的测试数据，再提供一个“30 40”的测试数据。

黑盒测试法：用事先设计的测试数据，用给输入-运行程序-判输出的方法，
在不检查程序内部结构的情况下评价程序的正确性和时空效率。

请读者牢牢记住这种多测试数据的“给输入-运行程序-判输出”的黑盒测试法，它是理解本书的基础。

小测验

1. 学习语言有哪三个层次？本章的侧重点是哪些层次？
2. 文件输入输出和键盘输入相比有什么好处？
3. 什么是黑盒测试法？为什么需要用多组数据测试一个程序？

1.4.2 静态分析

刚才的程序对于初学者来说需要解释一下。请注意：现在不需要理解所有的内容，而只需要知道核心部分就可以了。

怎样从文件里读数据 在本书中，文件被看作**流(stream)**，具有顺序性，好比瀑布中的水流一样，一旦流走，就再也回不来了。程序每次读取流的“下一个元素”，直到文件中没有更多的内容，或者剩下的数据已经不需要读取了。

数据读到哪里 由于流的特性是“一去不复返”，如果读取一个元素以后不加以处理或者保存，以后无法再从流中重新获得这个数据。数据以保存在**变量(variable)** 中，你可以把它理解为一个小盒子。正如可以在盒子的标签上写“装苹果的盒子”或者“装小说的盒子”，每个变量有它特定的**类型(type)**，表示这个小盒子里面可以放什么东西。最常见的类型是“int”，它表示存放整数的变量。正如再大的盒子也不能装下无数东西，int类型的变量只能表示约-2,000,000,000到2,000,000,000的整数[6]。理解数学上的整数和程序设计语言中的“整数”的差别相当重要，请读者牢记。

怎样进行运算 和数学一样，程序设计语言里也有**表达式(expression)** 的概念，它必须是可以计算的[7]。在代码中，“a+b”就是一个表达式，它的含义相当直观。可以预见，把它换成“a*b”后（用符号“*”表示乘号，因为“×”在标准键盘中找不到），程序将输出a和b的乘积。表达式可以相当复杂，比如“4-(a*(b+a)-(a+a*2)*(b-a))”，它很长，但仍然容易理解它的意思。

怎样把运算结果写到文件中 和输入类似，输出文件也被看成流，必须先输出前面的结果，再输出后面的。

以上的解释是概念层次的（还记得学习语言的三个层次吗？），程序流程可以概括为：

第一步 从输入流读入数据放在变量中

[6]由此可见，类型必须指明取值范围。
[7]后面将会看到，表达式可以有副作用，这是应当引起读者足够注意的。

第二步 使用变量和表达式进行计算

第三步 往输出流写入计算结果

以下在使用层次上进一步解释。

核心代码 有一些东西是大多数程序公共的，第1 3行表明这是一个使用标准C++流的程序，第5 6行表明输入流是文件sum.in，输出流是文件sum.out。第8、9、13、14行定义了主程序main，而只有10 12行才是核心程序。换句话说，在初学阶段，你可以忽略除了10 12行的其他代码，假设它们必须这么写（除了可以把sum.in和sum.out替换成你们自己喜欢的输入输出文件名）。换句话说，你可以每次在下面这段程序中添加自己的核心代码。需要说明的是下面代码的第10行是“注释”，因为它是以“//”开头的，这些文字对程序本身没有影响，相当于给给人看的说明性文字。给程序加注释是良好的习惯，它可以让别人或者三个月后的自己理解编写程序时的思路和细节。

```
#include<iostream>
#include<fstream>
using namespace std;

ifstream fin("filename.in");
ofstream fout("filename.out");

int main()
{
    // enter your code here
    return 0;
}
```

变量的声明 使用变量之前必须声明[8]，如程序2-2的第10行。这行告诉计算机[9]有两个名叫a和b 的变量，它们是int类型的。换句话说，a和b中可以保存约-2,000,000,000到2,000,000,000的整数。

变量的使用 变量声明以后可以使用。程序2-2的第11 12两行就是对变量的使用：首先用“fin >> a >> b;”语句从输入流中读入a和b，然后用“fout << a + b << endl;”语句往输出流中写入a+b并换行。

小测验

1. 什么是流？如何理解输入流和输出流的顺序性？举出生活中类似的例子

[8]由于这是”使用”层次，这里不讨论为什么要先声明，读者只需记住这个规则：使用变量前先声明。
[9]严格地说，是告诉编译器，见后文。

2. 什么是变量？如何从概念上理解变量的类型？

3. 什么是注释？为什么需要给程序加注释？

1.4.3 动态分析

通过对程序2-2的分析，读者应该对这个程序有了一定的了解。下面我们从另一个侧面理解这个程序。程序是一个指令序列，计算机一条一条的执行这些指令，最终得到程序输出。因此，程序的执行是一个动态的过程，它的具体流程由程序本身决定，而不像书一样可以用任意自己喜欢的顺序阅读。请注意，程序2-2的第9行是程序入口[10]，而第10行不需要执行（它只是通知计算机），默认情况下，程序将一条条依次执行，因此执行顺序为：$9 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 14$。

计算机在任何时候只有一个"正在执行的行"，称为当前行(current line)[11]，计算机每次执行一行的同时要计算下一个需要执行的行(next line)，然后去执行该行。一般情况下，下一行就是紧随其后的行，但是也两个特例，这将在基础语法部分详细介绍。

前面提到，数据存放在变量里。随着程序的执行，变量中的值会发生变化。每个变量在任意时刻有且仅有一个值。执行过程中变量的变化过程是动态分析的核心内容。只要你的程序写得没有二义性[12]，那么完全可以把你自己当作计算机去"执行"这个程序，算出运行结果。这再一次印证了第一章中的说法：计算机只是机械执行指令的工具。人也可以按照同样的规则计算，只是在速度、容量、精确度上远不及计算机。

细心的读者会问：那刚执行完第9行，还没有执行第11行时，a和b的值是多少？答案是：不一定。从理论上说，任何值都是有可能的，但是有且只有一个值，但是我们不关心，因为它不会影响到程序的结果。但是如果硬要在这样的"未知情况"下输出a+b的值，输出结果将是不确定的，我们应避免这种情况。换句话说，在使用一个变量之前，先要让它拥有确定的值。简单的说：先赋值，再使用。

程序的动态执行过程相当细节，不好理解，但是非常重要。读者应反复体会，并在实践中多利用IDE的调试功能，看看程序的动态执行过程是怎样的。

小测验

1. 程序的执行顺序和书籍的阅读顺序有何不同？

2. 什么是"当前行"？程序执行的过程中，当前行将如何变化？

[10]严格说来不是，读者目前可以忽略这个问题。
[11]严格说来应该是当前语句，不过本书一般只在一行写一条语句。
[12]不要写编译器相关的代码！它们是有二义的。

3. 动态分析的核心内容是什么？理论上是否可以不执行程序而完成动态分析？这说明了什么问题？

1.4.4 编译器和IDE

到现在为止，读者从感性上已经对C++语言有所认识了。接下来的问题是：我在什么地方输入C++ 语言的程序，又怎么样执行这些程序，得到想要的结果？为了回答这两个问题，本节介绍编译器和IDE。

编译器 计算机可以直接执行的指令非常简单。比如加法指令每次只能加两个数。对于a+b+c+d+e这样一条简单的C++语句，机器内部是分为四条机器指令方能完成的；机器还没有“变量”和“类型”的概念，所有的盒子大小都一样，只能装整数且必须通过地址来访问。这意味着我们不能说“往名叫NameOfBox的盒子中放字符A”，而只能说“往编号为12345的盒子中放一个整数65”。这对于程序员来说显然很不方便，可机器很好处理，因此把C++语言划分为“高级语言”，表明它接近人的思维而远离计算机的工作方式。

这样，C++语言必须翻译成机器语言才能被计算机执行[13]，这个翻译过程称为**编译(compile)** 。比如把表达式a+b+c+d+e翻译成四条“机器加法”，把“变量NameOfBox”翻译为“盒子12345”，把“字符A”翻译成“整数65”。这个翻译过程是麻烦而乏味的，幸好有编译器的存在，我们无需考虑这些问题。

IDE IDE是**集成开发环境(Integrated Development Environment)** 的缩写，它的作用是方便程序员进行开发工作。它的主要工作一般是让程序员：方便的编写代码；方便的编译程序（通常只需要一个键就可以调用编译器编译你的程序，并返回编译结果）；方便的执行程序；方便的调试程序（即通过考察程序动态执行的过程发现程序中的错误）。

理论上，只需要一个最简单的文本编辑器和一个编译器也可以写程序，但是IDE会大大的提高编程效率，所以几乎所有人使用IDE编程。IDE的灵活使用已经超出本书的范围，建议读者阅读相关书籍并做大量的实践。C++语言有很多IDE，著名的有商业软件Microsoft Visual C++，免费软件Bloodshed Dev C++和RHIDE for windows等，更多介绍见附录。

小测验

[13]理论上也可以设计直接执行C++ 语言的机器或者软件。事实上，成熟的C语言解释器已经存在

1. 什么是编译器？什么是机器语言？为什么我们不通常直接写机器语言程序？
2. 什么是IDE？它的作用是什么？是否必须有IDE才能编程？
3. 在Internet上调查有哪些C++编译器和IDE，它们的特点和比较。

1.4.5 C++词法

下面介绍C++基本语法。和英语一样，以词为单位，用语法来保证句子的“正确性”，而用一些常识来限制语义。比如这一句话：“Guten Tag!”[14] 一看就不像英语，因为它的单词就不像英语。而这句话：“I likes algorithm.” 这句话也不对，因为虽然每个词都是合法的英文单词，但是不符合语法规定。如果是这样的对话：“–我弟弟生病了。–她生啥病了？”每一句话都是语法正确的，但是语义却不对，因为“她”只能指代女性，而弟弟是男性。

程序设计语言同样有词法、语法和语义上的要求。后面我们将会看到，078是词法错误，给常量赋值是语法错误，变量没声明就引用是语义错误。

C++语言有四大类词：**标识符(identifiers)**，**关键字(keywords)**，**文字(literals)**，**符号(symbols)**，每一类词都有其构成规则。容易看出，任何一个词至多属于其中的一类。

标识符 可以拿来当名字，比如变量名，函数名等。标识符可以由字母、数字或者下划线_组成，但不能以数字开头（否则将引起歧义，见后）。

关键字 它们是被系统保留的标识符，它们有特别的含义，不能拿来给变量或者函数取名字，比如int代表整数类型，因此你不能给一个变量取名叫int。

文字 文字用来表示常量，比如整数常量12345，实数常量8212.02，布尔常量true和false，字符常量’A’ 和字符串常量“I love algorithm”。字符串是多个字符拼起来的序列，本书的后面部分会专门讨论字符串处理技巧。

符号 这些符号都是具有特殊含义的，如“+”是加号，“-”是减号和负号，“::”是作用域运算符。

刚才的解释比较抽象，初学者可以先只弄清楚一个程序的每个词各是什么类型的。大多数词都是很直观的。例如语句“int happy = 12345;”包含5个词，如表 1.1

词与词之间用空格或者换行符分隔。因为程序被看作是词的序列，因此多个空格

[14]德语的“你好！”

词	int	happy	=	12345	;
类型	关键字	标识符	符号	文字	符号
含义	整数类型	变量名	赋值运算符	整数常量	语句结束

表 1.1: 词法分析结果示例

十进制	1	2	3	4	5	6	7	8
二进制	1	10	11	100	101	110	111	1000
中文	一	二	三	四	五	六	七	八
英语	One	Two	Three	Four	Five	Six	Seven	Eight
罗马数字	I	II	III	IV	V	VI	VII	VIII

表 1.2: 前8个自然数的各种表示法

和一个空格没区别，只要分隔出来的词是一样的。

小测验

1. 词法、语法和语义是什么意思？举例说明
2. C++有哪四类词？一个词是否可能同时属于好几类？
3. 在程序的任何一个地方插入一个空格，都不会改变程序的本质吗？为什么？

1.4.6 二进制和十六进制

这一节比较抽象，但对于理解C++语言是至关重要的。

也许你听说过，在计算机内部，数以**二进制(binary system)** 表示。那么什么是二进制呢？二进制是数的一种表示法，就像我们用的**十进制(decimal system)** 一样。每个数可以有多个表示法，如表 1.2。同一列中的数是相同的，虽然表示方法不一样。

既然只是表示方法不同，因此“在十进制中2+3=5，因此二进制中10+11=101”这样的类比是正确的。十进制是“逢十进一”，二进制是“逢二进一”，因此10+11应该等于101，和十进制非常类似。二进制的每一位称为一个**比特(bit)** ，它要么是0，要么是1。

二进制是一种相当紧凑的表示方法。在很多情况下0表示“否”，1表示“是”，或者0表示“无”，1表示“有”。假如你问我10个问题：“你喜欢吃饭吗？你喜欢睡觉吗？你喜欢打篮球吗？你喜欢唱歌吗？你喜欢玩吗？你喜欢画画吗？你喜欢打游戏吗？你喜欢做作业吗？你喜欢吃零食吗？你喜欢这本书吗？”我可以回答：“是；是；否；是；

十六进制	0	1	2	3	4	5	6	7
二进制	0000	0001	0010	0011	0100	0101	0110	0111
十六进制	8	9	A	B	C	D	E	F
二进制	1000	1001	1010	1011	1100	1101	1110	1111

表 1.3: 十六进制和二进制对应表

是；是；是；否；否；是”，不过这个太罗嗦。我会说：“889。”想象一下，你是愿意记住那一堆“是”和“否”，还是愿意记住“889”？当然是后者了。

你可能不明白889是什么意思。这没有关系，打开Windows附件中的“计算器”，在菜单里选“查看”,“科学型”，键入“889”，然后点“二进制”。发现了吗？889变成了1101111001，它就是889的二进制表示。1代表“是”，0代表”否”，则二进制表示的10个数字就代表了我对十个问题的回答。

需要说明的是，本章不介绍二进制和十进制的相互转换，因为这里还没有必要使用。读者在做实验时可以像刚才一样借助于计算器或者其他辅助工具（将在本章后面介绍）。

另一方面，**十六进制(hexadecimal system)** 和二进制的转换就十分容易了。十六进制的每一位可以是0，1，2，3，4，5，6，7，8，9或者A，B，C，D，E，F，其中A F分别对应十进制的10 15。表 1.3是这16 个**数字(digit)** 的二进制对应关系：

一个十六进制数字恰好对应4个比特，因此刚才的1101111001转换成十六进制的方法就是：0011 → 3，0111 → 7，1001 → 9，即379。转换应该从后往前，前面不够4位的用0补充。如果计算器没有关的话，点击”十六进制”即可看到这一结果。

十六进制有什么好处？如果你做了刚才的两个练习应该可以体会到：十六进制表示比较短且容易和二进制相互转换，可以看作是二进制的“直观压缩表示”。本书将很多次的使用十六进制，建议读者熟记上表。本书表示十六进制时，在前面加上前缀0x(第一字符是数字零，而不是字母欧)，如 0xAABBCC 。类似的，**八进制(octal system)** 以0开头，如035是八进制形式，对应的十进制数是29。

在结束本小节之前，留一个思考题：数一数，4个比特的二进制数有多少个？它们对应的十进制数、十六进制数是哪些？答案是：$2^4 = 16$个，对应的十进制数是0，1，2 . . . 14，15，十六进制数是0x0 0xF。

小测验

1. 用表2-2把二进制数101010转换成十六进制，0xAABBCC转换成二进制。

再用Windows的"计算器"重做这题，检查你的结果是否正确

2. 十六进制和二进制相比有什么好处？

3. 八个比特的数有多少个？它们对应的十六进制数有哪些？

1.4.7 内存和变量

理解了二进制，下一个需要知道的事情是"数被放在计算机的存储器里"，就像我们把数记在脑子里或者写在纸上里，哪怕只是暂时性的。存储器有很多种，在本书里，除了作为输入输出载体的文件外，我们只用到**内部存储器(memory)** 的操作。在计算机里，为了方便访问，每8个位"打包"起来放在一个内存单元里，称为一个**字节(byte)** ，单位B。

如果你还记得上小节的思考题，应该能计算出一个字节有$2^8 = 256$种可能的内容，即二进制的00000000 11111111或者十进制的0 255或者十六进制的0x00 0xFF。

一个字节能保存的内容太少，好在计算机里的内存通常比这大得多。目前，典型的个人计算机里有1GB内存，1GB=1024MB，1MB=1024KB，1KB=1024B。对于本书中涉及到的大部分应用已经足够了。

一个内存单元里在任意时刻都有一个字节的内容，我们可以往里**写(write)** 或者**读(read)** 数据，称为对此内存单元的**访问(access)** 。

写数据 写操作需要一个操作数x（$0 \le x \le$ 0xFF），即写入的数据。写完后该内存单元里的数据变为x，和操作前的原始数据无关。

读数据 设该内存单元里的数据为x（$0 \le x \le$ 0xFF），则读操作将返回x，内存单元里的数据保持不变。

前面说过，在程序中，数据放在变量中；而本节告诉我们：在计算机内部，数据放在内存里。这二者有什么联系呢？

这个联系就是**地址(address)** [15]。整个内存被分为了相同大小的若干单元，每个单元是一个字节。每个字节都有一个编号，称为地址。一种可能的内存情况如图 1.2所示：

图 1.2: 一种可能的内存情况

[15]地址空间是线性的并不代表内存物的理结构也是线性的，有兴趣的读者可以阅读存储器方面的资料

其中黑色代表int型的变量a，灰色部分为char型变量b，白色为空闲内存。事实上，“空闲内存” 也是有值的，只是它不是程序中的任何变量。在这个例子中，变量a的地址为1031，变量b的地址为1037。正如图里看到的，int类型占四个字节，char类型只占一个字节。

等一等，一个整数要占四个字节？是的。前面提到过，一个字节只能保存256种可能的内容，这显然不能满足通常的要求。四个字节可以保存$256^4 = 4294967296$[16]种可能的数

这样的话，程序里的“给变量a赋值54321”在计算机内部执行的操作就是“分别往内存1031, 1032, 1033, 1034里写入0x31, 0xD4, 0x00, 0x00”。看上去很奇怪，好在大多数情况下，你并不需要知道这些细节。你需要牢牢记住的是：

- 字节包含8个二进制位，可以表示256种不同的数；
- 地址是内存单元的编号，内存中每个字节都拥有一个唯一的地址；
- 变量可能占用多个字节以表示更大范围的数，其中第一个字节的地址称为变量的地址

小测验

1. 一个字节有多少个比特？为什么有的变量需要多个字节？
2. 对内存单元的访问有哪两种类型？操作完后此内存单元会有什么变化？
3. 什么是地址？它和内存中的字节有什么对应关系？它和变量有什么关系？

1.4.8 变量的类型

从本质上说，类型就是一个值集合和在此集合上的操作集合。不管是什么样的数据，但了计算机内部都将是一个个字节，关键在于如何解释这些字节。换句话说，需要对各种各样的数据进行编码，用整数表示所有的东西，进一步用一个或多个字节表示这些整数。

整数 前面已经讲过，int的范围是约-2,000,000,000 2,000,000,000 ，有了二进制的知识，我们可以进一步说这个范围是-231 231-1，即-2147483648 2147483647。由于取值范

[16]建议读者把这个数记下来。它等于2^32，比4×10^9略大

围有232个元素，所以需要32/8=4字节储存。如果保证整数是非负的，可以用unsigned int，即无符号整数，它的范围是0 232-1，即0 4294967295。有的编译器支持更大范围的整数

实数 实数是一个很麻烦的数据类型，从初学者到熟悉语言C++的高手，几乎每个人都让实数弄得非常头疼。麻烦的根源在于实数的储存方式。详细的讨论需要很长的篇幅，这里只提两点：一、实数的表示本身是有精度限制的，double型一般保留15位有效数字。二、运算也会有误差。所以理论上相同的两个数可能会因为运算的误差变得有细微差别。建议读者不要直接比较两个实数是否完全相等，而是判断它们是否足够接近。

字符 char类型保存字符，包括可见字符如'a', 'W'，'$'等，也包括不可见字符，好比通常所说的"乱码"。char类型内部也是用整数来表示的字符，比如0x41代表'A'，0x61表示'a'，但是建议读者不要直接使用字符的数字表示。这一部分涉及到编码方面的知识，有兴趣的读者可以阅读相关书籍。字符有256种[17]，因此只占一个字节。

布尔型 bool类型保存"真假"，它只有两种取值，即true（真），false（假）。这是一种相当重要的数据类型。标志变量经常是bool类型，而所有关系运算和逻辑运算的结果都是bool类型的，它们常直接用于分支判断（见后）中。

数组 数组是多个相同变量的序列。设想你要保存全班50名同学的成绩，如果用变量score1, score2, score3, ...score50保存的话，不仅看起来很傻，而且如果全班人数变成51，需要修改程序，而且使用也不方便。比较好的方法是声明一个大小为50的数组，用score[i]表示第i个学生的成绩。这样只需要用到一个变量即可。声明一个大小为1000的数组，就可以应付全班人数不超过1000的各种情况–后面的元素不使用即可。

字符数组可以用来表示字符串。例如串"I love algorithm."的存储方式如图 1.3下：

'I'	' '	'l'	'o'	'v'	'e'	' '	'a'	'l'	'g'	'o'	'r'	'i'	't'	'h'	'm'	'.'	'\0'

图 1.3: 串的存储方式

其中数组的每个元素都是一个字符变量，包括空格' '和字符串结束标志'0'。

[17]严格地说这只是ASCII码，有的程序语言支持其他编码方式，字符可以占两个、三个字节甚至不定长。

这个程序是一个很好的例子：

```
#include<stdio.h>

union{
        int a[2];
        double b;
        char c[8];
}x;

int main()
{
        x.c[0] = 'G';
        x.c[1] = 'O';
        x.c[2] = 'O';
        x.c[3] = 'D';
        x.c[4] = 'B';
        x.c[5] = 'A';
        x.c[6] = 'B';
        x.c[7] = 'Y';

        printf("int:␣%d␣byte(s)\n", sizeof(int));
        printf("double:␣%d␣byte(s)\n", sizeof(double));
        printf("char:␣%d␣byte(s)\n", sizeof(char));
        printf("bool:␣%d␣bytes(s)\n", sizeof(bool));
        printf("\n");

        printf("integer:␣%d␣%d\n", x.a[0], x.a[1]);
        printf("real␣number:␣%e\n", x.b);
        printf("characters:␣%c%c%c%c%c%c%c%c\n", x.c[0], x.c[1], x.c[2],
                x.c[3], x.c[4], x.c[5], x.c[6], x.c[7]);
        printf("HEX:␣%X%X%X%X%X%X%X%X\n", x.c[0], x.c[1], x.c[2], x.c[3],
                x.c[4], x.c[5], x.c[6], x.c[7]);
    return 0;
}
```

该程序输出各种数据类型占用的字节数，然后把8个字符“GOODBABY”分别当作两个int、一个double、八个字符和八个十六进制数打印出来。运行结果如下：

int: 4 byte(s)

double: 8 byte(s)

char: 1 byte(s)

bool: 1 byte(s)

integer: 1146048327 1497514306

real number: 9.427752e+121
characters: GOODBABY

从图上可以看出，不同的数据类型并没有物理上的差异，只是解释不同罢了。程序细节暂时不用追究，也不提倡读者用union类型做这样的操作。

小测验

1. 什么是数据类型？有哪四种基本数据类型？不同类型的数据在物理形态上有差别吗？
2. 如何表示人的血型？自己设计一个数据类型，并说明如何把血型和整数对应起来
3. 什么是数组？数组有什么方便之处？字符数组是如何表示字符串的？

1.4.9 变量的声明和使用

前面提到过，变量使用前要先声明，指明变量的名字和类型。从词法上讲，变量名是标识符，在遵守规定的情况下你可以任意取名字。你可以声明一个叫numberOfPeople的变量，保存人数。为了避免出现"使用前拥有不确定值"的情况，可以在声明一个变量的同时给它**初始化(initialize)**，即赋初值。

```
#include<iostream>
#include<fstream>
using namespace std;

ifstream fin("month.in");
ofstream fout("month.out");

int main()
{
    int num[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    int month, day;
    double week;

    fin >> month;

    day = num[month - 1];
    week = day / 7.0;

    fout << "There␣are␣" << day << "␣days,␣i.e␣" << week <<
            "␣weeks␣in␣month␣" << month << "." << endl;
```

```
    return 0;
}
```

这是一个输出某个月份有多少天和多少个星期的程序。比如输入4，输出“There are 30 days, i.e 4.28571 weeks in month 4”。请注意实际上有30/7周，变成小数以后只输出前5位就成了4.28571。

第10 12三行是变量*声明语句*。month是一个int型的变量，num是一个int型的数组，它为初始化为31, 28, 31, ...。需要注意的是数组的第1，2，3 个数分别用num[0], num[1], num[2]...表示，第一个元素在num[0]而不是num[1]。这个可以理解为“第一个元素离数组头的偏移是0个元素长度”。因此第month月的天数保存在num[month - 1]中。

第17行是一个典型的*赋值语句*，左边的week代表“week这个变量”，而右边的day代表“变量day的值”。赋值a = x应该被理解为“把数值x放到变量a中”，是一个动作而不是事实或者关系。右边可以是任意一个表达式，只要它计算出一个值，但左边必须是一个变量，或者“代表变量”的东西[18] ，这一点相当重要。

不能写”a + 2 = b”，因为”a + 2”不是一个”东西”，无法把b放在里面。

可以写”a = a + 1”，因为右边可以计算出一个值，而左边是一个变量。这个式子有点奇怪，因为它在数学上是不成立的。再次强调：赋值代表一个动作而不是在陈述事实。”a = a + 1”的结果是让a的数值增加了1。

如果依次执行”a = b; b = c”，是否意味着a、b、c都相等呢？不是的！赋值不是在陈述事实。程序是动态执行的，曾经成立的关系也许以后不再成立。假设程序执行前a = 1, b = 2, c = 3，那么执行完第一条语句后a = 2, b = 2, c = 3，而执行完第二条语句后a = 2, b = 3, c = 3，a并不等于c。这里使用了动态分析，需要读者熟练掌握。本书将多次使用动态分析技巧，逐步列出每个步骤后重要变量的值。

小测验

1. 如何声明一个有100个元素的实数数组abc？如何表示它的第k个数？
2. 什么是赋值语句？是否可以用语句”x * x = 1”解方程x2=1？
3. 设三个int型变量x, a, b，执行”x=a; a=b; b=x”后a和b的值将发生什么样的变化？如果再执行”a=a+b; b=a-b; a=a-b;”呢？

[18]严格来说，左边必须是Lvalue，包括变量或者其他指向可变内容的地址的东西

1.4.10 运算符和表达式

常用的运算分以下几类：算术运算、关系运算、逻辑运算、位运算、赋值运算。

算术运算 就是加减乘除，分别用+, -, *, /表示。需要特别注意的是运算符和变量类型有关。例如整数除以整数等于整数，实数除以实数等于实数。因此5/2=2，而5.0/2.0=2.5。对于整数，还有一类特殊的运算"取模"，即取余数，运算符为%。例如7除以4商1余3，即7/4=1, 7%4=3。注意负整数做除法和取模时，余数是负的。字符也可以做运算，比如'A'+2='C', '3'+6='9'。请只对字符使用两种运算："字符1+整数=字符2"以及"字符2-字符1=整数"。'A'+'B'虽然也可以得到一个值，但是意义不明确。需要特别注意的是，某些运算将会引起错误，如计算1/0（除零错），或者两个等于十万的int型整数做乘法（运算结果超过int表达范围，或称溢出overflow）。在所有运算中，算术运算是最危险的。

```
#include<iostream>
using namespace std;

void test_int()
{
    int a = 100000;
    cout << "100000␣*␣100000␣=␣" << a * a << endl;
        cout << "14␣/␣5␣=␣" << 14 / 5 << endl;
        cout << "14␣%␣5␣=␣" << 14 % 5 << endl;
        cout << "-14␣/␣5␣=␣" << -14 / 5 << endl;
        cout << "-14␣%␣5␣=␣" << -14 % 5 << endl;
        cout << "14␣/␣-5␣=␣" << 14 / -5 << endl;
        cout << "14␣%␣-5␣=␣" << 14 % -5 << endl;
        cout << "-14␣/␣-5␣=␣" << -14 / -5 << endl;
        cout << "-14␣%␣-5␣=␣" << -14 % -5 << endl;
}

void test_double()
{
        double c, d, z;

        c = 1.0; d = 0.0;
        cout << "1.0␣/␣0.0␣=␣" << c/d << endl;
        c = -1.0; d = 0.0;
        cout << "-1.0␣/␣0.0␣=␣" << c/d << endl;
        c = 0.0; d = 0.0;
        cout << "0.0␣/␣0.0␣=␣" << c/d << endl;

        z = 1.0;
    cout << "z␣=␣1.0;␣z␣=␣" << z << endl;
        z /= 12345678912345.0;
```

```
        cout << "z␣/=␣12345678912345.0;␣z␣=␣" << z << endl;
        z += 1.0;
        cout << "z␣+=␣1.0;␣z␣=␣" << z << endl;
        z -= 1.0;
        cout << "z␣-=␣1.0;␣z␣=␣" << z << endl;
        z *= 12345678912345.0;
        cout << "z␣*=␣12345678912345.0;␣z␣==␣" << z << endl;
}

void test_bool()
{
        cout << "(1␣<=␣2)␣=␣" << (1 <= 2) << endl;
        cout << "(3␣==␣4)␣=␣" << (3 == 4) << endl;
        cout << "(5␣!=␣6␣&&␣-7␣<=␣10)␣=␣" << (5 != 6 && -7 <= 10) << endl;
        cout << "(1234␣&␣5678)␣=␣" << (1234 & 5678) << endl;
        cout << "(1␣<<␣16)␣=␣" << (1 << 16) << endl;
}

void test_order()
{
        int a = 2, b = 3;
        cout << "(a++)␣*␣(a+2*b)␣*␣(b++)␣=␣" << (a++) * (a+2*b) * (b++) << endl;
}

void test_char()
{
        cout << "'A'␣+␣2␣=␣" << 'A' + 2 << endl;
        cout << "(char)('A'␣+␣2)␣=␣" << (char)('A' + 2) << endl;
        cout << "'z'␣-␣'f'␣=␣" << 'z' - 'f' << endl;
}

int main()
{
        printf("\ntest␣int:\n");
        test_int();

        printf("\ntest␣double:\n");
        test_double();

        printf("\ntest␣bool:\n");
        test_bool();

        printf("\ntest␣char:\n");
        test_char();

        printf("\ntest␣order:\n");
        test_order();

    return 0;
}
```

函数test_int()测试了整数运算。运行结果如下：

100000 * 100000 = 1410065408
14 / 5 = 2
14 % 5 = 4
-14 / 5 = -2
-14 % 5 = -4
14 / -5 = -2
14 % -5 = 4
-14 / -5 = 2
-14 % -5 = -4

函数test_double()测试了实数运算。除0操作会得到一些异常的结果，而即使是正常运算，误差也会让答案完全不一样。运行结果如下：

1.0 / 0.0 = 1.#INF
-1.0 / 0.0 = -1.#INF
0.0 / 0.0 = -1.#IND
z = 1.0; z = 1
z /= 12345678912345.0; z = 8.1e-014
z += 1.0; z = 1
z -= 1.0; z = 8.10463e-014
z *= 12345678912345.0; z == 1.00057

为什么这样？z的值应该等于1啊！原因在于，当z等于1/12345678998765432 1时，z里保存着数$8.1000000656104*10^{-18}$。加1后应该等于1.00000000000000000810000000089100，可是double型无法保存这么多有效数字，所以变成了1.0000000000001。这一步不太好解释（涉及到实数的格式），但是丢失有效数字是肯定的。如果把z += 1.0和z-=1.0中的1.0改成10000.0，那么加法的结果将会变成10000，使得最后答案变成0，而不是正确答案1！这个例子提醒我们：实数的误差不可忽略。

关系运算 关于运算比较两个数，返回一个布尔值。常用的关系运算有：<,>,<=（小于或等于）,>=,== （等于）和! =（不等于）例如“3 < 5”的值为true，“6==7”的值

为false。字符也可以做比较，比如'A'<'B'，'3'<'5'。大写字母小于小写字母。前面提过，由于运算误差的存在，一般不推荐直接比较两个实数。

逻辑运算 逻辑运算是布尔值的运算，主要有与、或、非三种，用&&、||和!表示。比如"$a < b||c < d$"表示"a小于b或者c小于d"，而"$1 <= x\&\&x <= 10$"表示$1 \le x \le 10$。注意不能把两个"<="号连在一起，而必须写成两个部分，用逻辑与连接。"$(!a\&b)||(a\&\&!b)$"表示a为假且b为真，或者a为真且b为假，即a和b恰好有一个成立。这叫"异或"操作。

位运算 理解了二进制，位运算都是很容易理解的。位运算<<代表左移，$a << b$等价于$a * 2^b$。因此经常用$1 << x$来表示$2x$。>>代表右移，经常用$x >> 1$来代替$x/2$。按位的逻辑运算是把两个数的每一位都看成是布尔值（1表示真，0表示假），然后每位分别做运算。按位的逻辑与、或、非分别记为&, —和 。例如1234 & 5678=1026，因为1234写成二进制为10011010010，5678写成二进制为1011000101110，每位取二者的"与"（从最后一位开始往左，不够的补0），得10000000010，即1026。C++语言有按位异或运算符∧ ，但没有专门的逻辑异或运算符。如果用整数1和0来表示真假（而不用bool型），可以用∧ 来进行逻辑异或运算。需要说明的是：<<运算符也是可能溢出的。

函数test_bool()运行结果如下：

```
(1 <= 2) = 1
(3 == 4) = 0
(5 != 6 && -7 <= 10) = 1
(1 << 16) = 65536
```

当使用多个运算符时，需要了解它们的优先级和结合性。优先级指明不同运算符的相对计算顺序。1+2*3被解释为1+(2*3)，因为乘法的优先级高于加法。结合性指明同一运算符连续使用多次时的计算顺序。例如1+2+3被解释为(1+2)+3还是1+(2+3)。不要觉得这二者显然是相等的。下一节介绍函数，如果表达式是"a()+b()+c()"，而三个函数都有副作用时，先算哪个后算哪个就不再是无足轻重了。

表达式都是有值的，可以是左值也可以是右值[19]。比如"a[i] = c + d"，它的左边是一个表达式，计算出"该把值放哪里"，右边也是一个表达式，计算出"往那个地方放一个什么东西"。写成一般形式，就是"左值= 右值"。

[19]这里不给出左值和右值的严格定义，读者可以在上下文中体会其含义。

有一种特殊的赋值需要特别注意。a = a + b可以写为a+=b。例如a+=2，$c <<= 1$或者$f\&\& = (1 << 2)$都是合法的。a+=1可以进一步写成a++ 或者++a，它们不仅是有副作用的赋值语句（自身加1），本身还产生值。a++的值为原来的值，++a为改变后的值。建议初学者不要使用增量运算的值，只把它作为赋值语句使用。减量操作符"–"类似。如果你觉得你对C++的表达式求值规则很清楚，不妨解释一下test_order()函数中表达式的计算顺序是什么，结果为何是48？如果你无法解释，那么请只把"++"和"–"作为赋值语句使用，而不用出现在表达式的其他地方。

小测验

1. 如何理解'A'+2等于'C'，'0'+8等于'8'？计算3%4，19%5和4/6，并举出两个引起出错的算术运算。
2. a+=3和–c的值是什么？副作用是什么？应如何使用这些有副作用的表达式？
3. 什么是优先级和结合性？什么时候需要用到优先级和结合性？

1.4.11 函数

前面介绍的运算符虽然丰富，可是很多计算仍难于处理。比如开平方、三角函数。正如数学函数一样，C++语言也提供函数以供使用，sqrt(a)就是a的平方根，sin(x)就是x的正弦函数值。*你可以自己设计函数，也可以使用现成的函数。*在代码实现时，本书中的算法大都以函数的形式提供。

简单的说，**函数(function)** 是一段程序。2.1节介绍的main()就是一个函数。你还可以写其他函数，然后像调用sqrt, sin函数一样的调用它们。函数最大的好处是重用性[20] 。比如写出了求两个数的最小公倍数的程序以后，可以在其他程序里直接调用这个函数而不需要再写一遍。本书将给出很多这样的函数。

和声明变量类似，声明函数也要指明函数的名称和类型。不同的是函数是输入到输出的映射，因此要分别指明输入的类型和输出的类型，例如：double max(double, double)表示一个名为max的函数，它的输入参数有两个，都是double 类型，输出也是double类型。读者可能有疑问：输出参数有多个怎么办？通常的解决方法有两种，一是使用"引用"，把输出参数当作可变的输入参数传进来，二是把所有输出参数打包成一个结构体，返回整个结构。两种方法各有千秋，本书都有使用。

[20]重用性有很多种表现方式，函数只是其中一种

下面就是这样两个例子。a除以b的结果由两部分组成：**商(quotient)** 和**余数(remainder)**，第一个函数有两个输入参数a和b，返回一个result类型的结果。result是一个结构体，它有两个成员变量quotient和remainder组成。第二个函数有两个输入参数a和b，以及输出参数quotient和remainder。

下面是一个测试函数的程序：

```
#include<iostream>
using namespace std;

struct result{
        int quotient;
        int remainder;
};

result divide(int a, int b)
{
        result t;
        t.quotient = a / b;
        t.remainder = a % b;
        return t;
}

void divide(int a, int b, int& quotient, int& remainder)
{
        quotient = a / b;
        remainder = a % b;
}

void swap(int&a, int& b)
{
        int x = a;
        a = b;
        b = x;
}

void bad_swap(int a, int b)
{
        int x = a;
        a = b;
        b = x;
}

int a;
void greetings()
{
        a = 0;
        cout << "Hello!" << endl;
}
```

```
int bad_one_plus_one()
{
        a = 1 + 1;
        greetings();
        cout << "1␣+␣1␣=␣" << a << endl;
        return 0;
}

int one_plus_one()
{
        int a;
        a = 1 + 1;
        greetings();
        cout << "1␣+␣1␣=␣" << a << endl;
        return 0;
}

int main()
{
        bad_one_plus_one();
        one_plus_one();
        return 0;
}
```

从程序可以看出：不同的函数可以有相同的名字，只要参数类型不同就行。参数有两种类型，一种是**传值(by value)**，一种是**传参(by reference)**。在类型int后跟符号"&"表示这是参数是传参方式，否则为传值方式。二者的区别好比赋值时的左值右值：传值型的参数只使用值，因此如果a=3，那么调用divide(7,3,x,y)和divide(2*a+1,a,x,y)效果是一样的。它只用到3和a+7的值。而传参型的参数必须使用x、a[i] 这样的左值。例如调用divide(10, b, 1, 4*x)是不行的，因为1和4*x都不是左值。这样的调用就好比是"计算10除以b，把商放到1里，余数放到4*x里"，是不可能实现的。

需要特别强调的是，C++语言的函数是一段代码，而不是数学上的函数。数学上的函数是按一定规则定义出来的，而代码除了计算函数值外，完全可以做更多的事情，当然，也包括破坏。换句话说，C++语言的函数和数学函数的最大区别在于：它是有**副作用(side effect)** 的。副作用的存在是过程式程序设计语言的通病。

函数bad_one_plus_one调用"greetings()"函数打印欢迎信息，然后输出"1 + 1 = 2"。可读者不难看出，greetings()函数"顺便"把a改变了！这个改变使得bad_one_plus_one()函数出人意料的打印出了"1 + 1 = 0"这样的荒唐结果。

为了避免每个变量不受到意外的改变，应尽量把它声明成**局部变量(local vari-**

able) 。在函数体内部声明的变量称为局部变量。局部变量只能被该函数修改，而不能被其他函数修改。声明在所有函数之外的变量称为**全局变量(global variable)**，它们可以被所有函数访问，因此也可能被不经意的改变，不推荐使用。为了更清晰的展示算法实现，本书使用全局变量表示核心数据结构，但建议读者在使用时改外局部变量。one_plus_one()函数是一个比较好的实现方式，它只使用了局部变量，即使greetings()函数有副作用[21]，也不会影响到它。

函数有一种特别的调用方式：自己调用自己，即**递归调用(recursive call)**，将在后面详细介绍。

小测验

1. 什么是函数？为什么要使用函数？C++语言的函数和数学函数相比有何不同？
2. 为什么要尽量使用局部变量？全局变量有什么危险之处？
3. 如何声明一个函数？如何定义一个函数？如何调用一个函数？什么是递归调用？

1.4.12 控制流和程序结构

过程式的C++程序由若干个函数组成，整个程序的入口在main函数。

每个函数由若干条语句组成。语句是C++程序的基本单位。语句的类型有：声明语句、表达式、复合语句、分支语句和循环语句。还有异常处理语句，本书不作讨论。

声明语句和表达式语句前面已经讨论过。复合语句只是简单的用""和""把多条语句合成一条语句，并且可以在复合语句中定义自己的局部变量，好比函数一样。分支语句和循环语句是本节讨论的重点。前面说过，默认情况下，每执行完一条语句，将执行紧随其后的那条语句。但分支和循环可以改变语句执行的顺序。

简单的说，分支语句"if(x) y; else z;"的意思是"如果满足条件x，那么执行语句y，否则执行语句z"；循环语句while(x) y的意思是"当满足条件x时一遍又一遍的执行y，直到条件x不满足就立刻退出"。由于语句y和z可以是复合语句，所以循环体可以很长。

下面是一个很好的例子：

"如果x是奇数，那么乘以3加1；如果x是偶数，则除以2，直到变成1。"

[21] 只要不访问非法内存区域

例如，从9开始，依次变为28，14，7，22，11，34，17，52，26，13，40，20，10，5，16，8，4，2，1。程序在函数sequence()中。

for循环是另一个常用的循环语句，它书写紧凑，经常被用来编写规则循环，例如函数sum()，它一开始给i赋值为0，每次把x[i]添加到和s里，输出i、x[i]、s的值，然后执行i++，直到不满足条件“$i < x.size()$”。采用动态分析的方法很容易分析出程序的作用是：把数组x的所有元素加起来，放到变量s里并输出。

continue语句和break语句 在特殊的情况下，循环需要中途退出或者进入下一个循环。例如我们可以把刚才求和的函数修改一下，让它忽略所有负数，并且当和大于10000时立刻退出，见函数sum2()。

其他语句 分支语句还包含switch语句，循环语句还包含do-while语句，它们的含义可以用menu() 函数加以说明。这是一个菜单的例子，用户有三种选择。如果输入错误（不是1，2，3或者0），程序将继续询问，直到用户给出一个正确的输入。不同输入的处理使用switch语句实现，注意每个case语句的最后应该有break语句。

下面是完整程序：

```
#include<iostream>
#include<vector>
using namespace std;

typedef vector<int> vi;

int sum(vi x)
{
        int s = 0;
        for(int i = 0; i < x.size() - 1; i++){
                s += x[i];
                cout << "i␣=␣" << i << "␣,␣x[i]␣=␣" << x[i] <<
                        "␣,␣s␣=␣" << s << endl;
        }
        return s;
}

int sum2(vi x)
{
        int s = 0;
        for(int i = 0; i < x.size() - 1; i++){
                if(x[i] < 0) continue;
                s += x[i];
                cout << "i␣=␣" << i << "␣,␣x[i]␣=␣" << x[i] <<
                        "␣,␣s␣=␣" << s << endl;
                if(s > 10000) break;
        }
```

```
        return s;
}

void sequence(int x)
{
        cout << x;
        while(x > 1){
                if(x % 2 == 1) x = 3 * x + 1;
                else x >>= 1;
                cout << " " << x;
        }
        cout << endl;
}

void menu()
{
        int choice;
        do{
                cout << "What do you want? (1-Eat, 2-Drink, 3-Game, 0-exit)";
                cin >> choice;
                switch(choice){
                        case 1:
                                cout << "Eating..." << endl;
                                break;
                        case 2:
                                cout << "Drinking..." << endl;
                                break;
                        case 3:
                                cout << "Gaming..." << endl;
                                break;
                        case 0:
                                cout << "Bye!" << endl;
                                break;
                        default:
                                cout << "Input error!" << endl;
                                break;
                }
        }while(choice < 0 || choice > 3);
}

int main()
{
        vi x;
        x.push_back(1);
        x.push_back(2);
        x.push_back(3);
        x.push_back(4);
        x.push_back(5);
        cout << "sum = " << sum(x) << endl;

        sequence(9);
```

```
        menu();
        return 0;
}
```

小测验

1. 过程式的C++程序由什么组成？基本单位是什么？语句有哪几种类型？程序的组成规则属于词法、语法还是语义规则？

2. 分支语句的作用是什么？如何用自然语言描述“$if(a > b)max = a; elsemax = b$”的含义？

3. 循环语句的作用是什么？如何用自然语言描述“$for(i = 1; i <= 100; i++)sum+ = i;$”的含义？

1.5 数据结构基础

本节介绍数据结构基础。主要目的有两个：一是从概念层次上介绍数据结构，二是让读者建立算法分析的基本概念。

1.5.1 逻辑结构：线性表、树和图

本节介绍三个最基本的逻辑结构：线性表、树和图。

线性结构 线性结构的特点是：有唯一的“第一元素”。除了最后一个元素之外，每一个元素都有唯一的“上一个元素”和“下一个元素”。这是最简单的逻辑结构，不用多说。

树型结构 树型结构具有层次性，看起来像是一棵倒立的“树”，如图 3.7所示。树有许多有趣的术语，非常形象。每棵树由一些点组成，称为**结点(node)**，其中最特殊的结点是**树根(root)** 和**叶子(leaf)**。树根处于最顶层，是所有其他结点的**祖先(ancestor)**，叶子是树的末端。每个结点有唯一的**父亲(father)**，或称**双亲(parent)**。每个非叶结点有一个或多个**儿子(son)**，或称**孩子(child)**。以儿子为根的树称为**子树(subtree)**。父亲相同的结点互为**兄弟(brother或sibling)**。叶子没有儿子，根没有父亲。需要特别注意的是：树型结构强调的是结点之间的关系，而不是“如何把关系形象的画出来[22]”。通常可以给树下这样的定义：一棵**树(tree)** 要么为空，要

[22]另外，同一棵树可以有很多种画法。把每个结点作为根，都可以画出一棵树。

么由根结点和根的$n(n >= 0)$ 棵子树组成。这个定义是递归的，因为子树又可以有子树，子树的子树还可以有子树

本书的结构就是树状的。书里分若干章（章是书的儿子），每章分为若干节（节是章的儿子），每节又分若干小节（小节是节的儿子）。

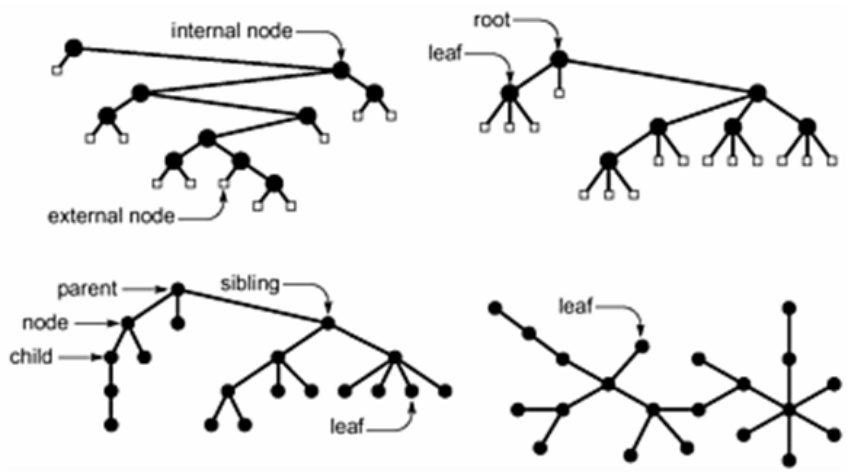


图 1.4: 树的相关术语

网状结构 更复杂的结构可以是网状的，任意两个结点都可能有联系。图 1.5是一个典型的图。和树类似，我们并不关心如何把图画出来，关键是哪些结点之间有关系。下图中，结点7和结点0、1、2、4有关系。包含结点及其相互关系的结构称为**图(graph)** ，结点之间的关系称为边，那么图可以表示成$G = (V, E)$，其中G为图，V为结点集，E为边集。如果把Internet上的计算机想象成点，直接连接起来的计算机想象成边，那么整个Internet的拓扑结构就是一个图。

图可以是**有向(directed)** 的，a和b具有某种关系并不代表b和a具有这种关系。如果a和b都是人，a知道b并不意味着b也知道a（比如b是个名人）。

边上还可以有**权(weight)** ，表示边的某个属性。如果把北京的岔路口想象成结点，街道想象成边，边上的权代表骑车从街道一端到另一端需要的时间，那么得到的就是一个**带权图(weighted graph)** ，或称**网络(network)** 。一个很自然的问题是：从清华骑车到北京站，走怎样的线路，时间最短？这是图论中的标准问题，《算法艺术与信息学竞赛》中已经详细讨论，本书后面将给出详细的程序。

线性结构中的数据有前驱-后继关系；树中的数据有父子关系和由此产生的层次关系；图中的任何两个元素都可以有关系，它在三者中是最广泛的。线性结构可以看作

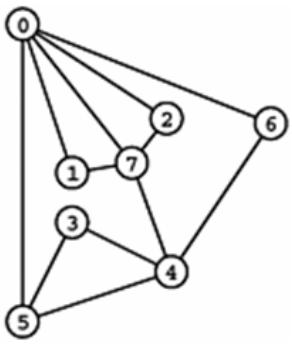


图 1.5: 图的直观表示

树的特殊情况，树又可以看成图的特殊情况。

本节讨论的三种结构：线性结构、树状结构和网状结构只是比较典型的结构，还可以有其他变形或者组合，如环型结构、矩阵等。这里只讨论逻辑结构：即数据间的关系，至于具体怎么用C++语言实现则是下一小节所关心的内容。

小测验

1. 有哪三种常见的逻辑结构？它们分别表示了数据间的什么关系？
2. 在逻辑结构中，哪一样是本质？关系还是图形表示？为什么说线性结构是树的特例，树是图的特例？
3. 举出一个实际的例子，可以用有向加权图来建模。

1.5.2 逻辑结构的物理实现

最简单的逻辑结构是线性结构，它有两种物理结构：数组和链表，前者是连续的，后者是不连续的。前者通过绝对位置定位，地址是下标的函数；后者通过相对位置定位，地址依赖于其他元素。

数组(array) 数组采取绝对定位，所有元素的物理位置是连续的。数组的不方便之处在于：如果需要删除一个元素，需要把它后面的元素全部一个一个往前移动（不能同时移动！），元素越多，越花时间。如图 1.6，把第三个元素删除后引起了4次元素移动。

链表(linked list) 链表采用相对定位，每个元素保存下一个元素的位置。下图是一个删除的例子，所有元素的位置都不需要发生改变，只需要让一个元素的“下一元素指针”发生变化即可。在这一点上，链表比数组优秀。但是链表随机访问比较慢，即如

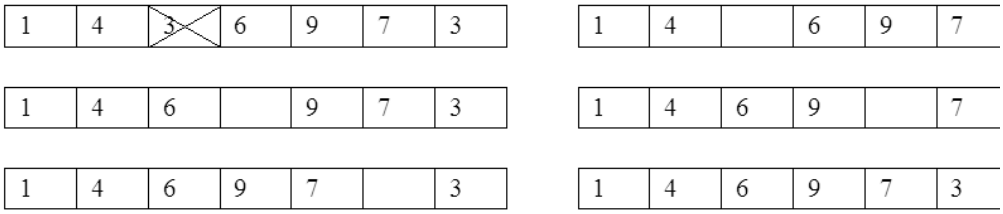


图 1.6: 数组中的删除操作

果要读取“第100个元素”，只能先读第一个元素，顺着它的指向去找第二个元素，再找第三个...而不像数组，可以直接计算出第100个元素在什么地方，如图 1.7

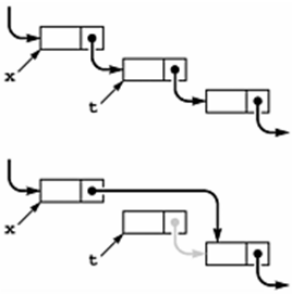


图 1.7: 链表

为了更加形象的分析线性表，读者回忆军训时的情形[23]。一排同学站在一起，教官可以很方便的找到左数第3名同学或者第7名同学（随机访问），叫他出列并让其他同学向右看齐，把他的空位补上（删除）。这个例子说明：顺序表适合随机访问但不适合删除（向右看齐需要多名同学移动）。

为了考察链表的特性，想象自己正在在一个游园会中玩一个闯关游戏。你需要从第一关开始玩，每过一关将得到一个小盒子，记录下一关的位置。一开始的时候，谁也无法知道第6关在什么地方，只能老老实实的玩完前5关，得到第5个小盒子，才知道第6关在什么地方。现在假设游戏的设置发生了一点改变：原来的第9 关被删除了，原来的第10关需要紧接在第8关后面。由于关卡的设置是链式的（看出来了吗？），只需要用原来第9关的盒子把第8关的盒子替换掉就可以了，其他盒子都不需要有任何变化。这就是链式结构的特点：随机访问困难（否则就不用一关一关闯了，想玩哪关就玩哪关），而删除简单（只修改一个盒子即可）。

[23]如果没有参加过军训，可以考虑上体育课

从逻辑上看，数组和链表都表示了线性结构，因为有唯一的开始元素，除了最后一个元素外其他元素都有唯一的前驱和后继。可以这个例子也可以看出：同一个逻辑结构可以有不同的物理实现。

树的情形有些复杂，我们不妨先考虑二叉树，即每个结点恰好有两棵子树：左子树和右子树[24]。下面左图所示就是一棵二叉树，其中空心方块表示空树，它们不是结点。

二叉树可以用链式存储，如图 1.8。我们无法知道第3层第一个结点在哪里，而必须从根结点开始自上而下的**遍历(traversal)** 。

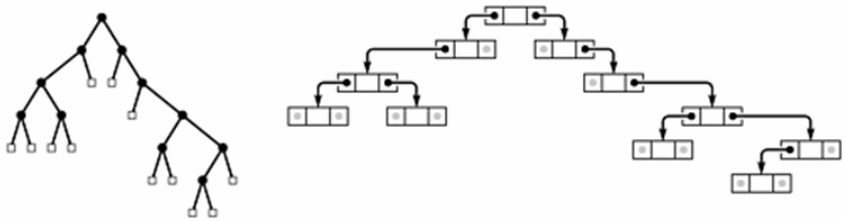


图 1.8: 二叉树的链式存储法

二叉树也可以用数组存储，从根开始，每层自左向右编号为1，2，3 ，如图 1.9。除了结点1外，每个结点i的父亲是i/2，左儿子是2i，右儿子是2i+1。同样是以数组作为物理结构，这里我们把它理解为了一棵二叉树。

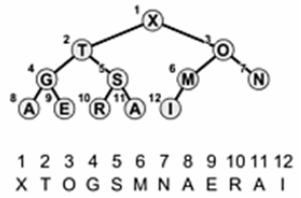


图 1.9: 二叉树的数组存储法

至于一般意义下的树（每个结点可以有多个儿子，而且各个儿子之间也没有顺序关系），可以转化成二叉树存储。在新的二叉树中，每个结点的左儿子为原树中的第一个儿子，右儿子为原树中的下一个兄弟。这样的表示法称为**左儿子-右兄弟表示法(left child-right sibling representation)** ，它相当于把所有儿子串成了一个链表，如图 1.10。

[24]二叉树不是树的一种，因为树的各个儿子直接没有顺序关系

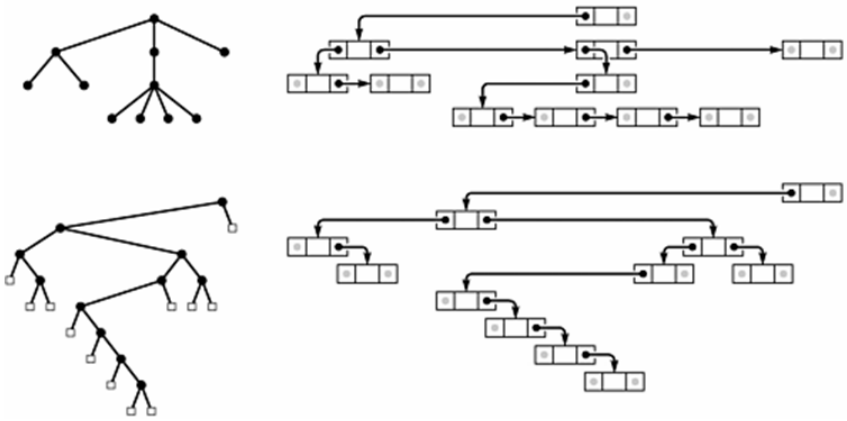
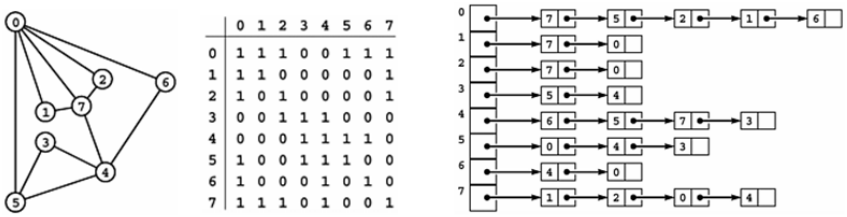


图 1.10: 树的左儿子右兄弟存储法

图有两种常用的表示法：**邻接矩阵(adjacency matrix)** 和**邻接表(adjacency list)**，如图 1.1所示邻接矩阵直接记录任何两个结点之间是否有边，而邻接表实际上是为每个结点做了一个链表，记录与它相邻的所有结点。和数组与链表的比较类似，数组直接判断两个点是否有边相连比较方便，而链表不方便。然而，本书中的图结构一般很少增加或者删除点，邻接矩阵在这方面并没有劣势。邻接矩阵的主要缺点是空间占用大。一个1万个点、10万条边的图，邻接矩阵需要1亿个表项，而邻接表只需要10万个。边比较少的图称为稀疏图，经常用邻接表表示；因为空间占用小而且很多算法的时间效率也较高，而稠密图或者结点数较少的图经常用邻接矩阵表示，基于邻接矩阵的算法常常简单、容易记忆，且对这类图特别有效。



	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	1	1	1	0	0	0
4	0	0	0	1	1	1	1	0
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

图 1.11: 图的邻接矩阵和邻接表

小测验

1. 物理结构和逻辑结构有什么区别？线性的逻辑结构可以用哪两种经典的物理结构实现？它们各自有什么特点？

2. 什么是二叉树？树如何转化成二叉树？二叉树有哪些表示法？
3. 比较图的邻接矩阵和邻接表表示的优缺点。

1.5.3 外部特性和内部结构

不管是物理结构还是逻辑结构，数据结构的某些属性可以被外界所感知，而另外一些属性却是不可见的。换句话说，数据结构包含外部特性和内部结构两个方面。外部特性决定了它将如何被使用，而内部结构决定了它的原理和具体实现。初学数据结构时，可以更多的关注它们的外部特性，在熟练使用并体会到数据结构的好处之后再学习它们的内部结构，并根据其中的设计思想创造出新的数据结构。

有一只自动储钱罐，它有一个孔和一个按钮。存钱的时候，你可以从小孔往里面投一枚硬币；取钱的时候，只要按一下按钮，面值最大的硬币就会从孔里掉出来。

自动储钱罐的设计者告诉你：钱罐里有一个很小的机器人，每次按下按钮的时候，它就从钱罐里找出最值钱的一枚，从孔里扔出来。如果硬币有很多的话，从一大堆钱里找到最值钱的硬币是需要花时间的，所以可能你按下按钮以后需要等待几分钟，让小机器人慢慢找。

小机器人是这样工作的：当你扔一枚硬币进来的时候，它什么都不做，自己睡大觉；当你按按钮的时候，它慌了，赶紧找钱。它先随便挑出一个硬币拿在手里，然后把其他所有硬币的看一遍，如果发现更值钱的，就用把手里的硬币换掉，最后手里拿着的就是最值钱的硬币，然后从孔里扔出去。

下面是小机器人的C++语言程序（仅作为示例）：

```
void new_coin()
{
        zzzZZZ();
}

void delete_max()
{
        int i;
        int best = 0;
        for(i = 1; i < coin_count; i++)
                if(coin[i] > coin[best]) best = i;
        throw_away(best);
}
```

其中coin[i]表示第i枚硬币，throw_away(i)表示把第i枚硬币扔出去且终止程序。best代表当前手中的硬币，它在小机器人已经检查过的硬币中是面值最大的。由于coin是无序

数组，我们把这种方法称为*无序数组实现法*。

可以预料，这个钱罐“添加硬币”很快（小机器人啥都不做），找最大面值却很慢。如果小机器人检查一枚硬币的时间是0.01秒，那么有100个硬币时需要1秒，有10,000个硬币时需要100秒（约两分钟），而1,000,000个硬币时就需要10,000秒（约2.8小时）！你可以忍受这样的速度吗？

如果改改小机器人的程序，情况就完全不一样了！如果你曾留意过超市里的售货员是怎样找零钱的，你也立刻明白这个方案：拿几个不同的小桶，每个桶装一种面值的硬币。假设一共有1元、5角、1角、5分、2分和1分共6种面值的硬币，则只需要六个桶。当来了一枚新硬币时，小机器人把它放到相应的盒子中；需要找钱时，小机器人只需要看1元的盒子里有没有硬币，有的话随便拿一个扔出去；如果没有的话再看5角的盒子有没有硬币，有的话随便拿一个扔出去　不管有多少硬币，只要盒子装得下，总是最多只需要开6次桶即可，即使每开一个桶需要5秒钟，有1,000,000个硬币时最多也只需要半分钟，比刚才的2.8小时快多了。程序如下：

```
void new_coin(int value)
{
    count[value]++;
}

void delete_max()
{
        int i;
        for(i = 1; i <= 6; i++)
                if(count[i] > 0) throw_away(i);
}
```

其中count[i]表示第i种面值的桶里有多少硬币，throw_away(i)表示把一枚面值为i的硬币扔出去。每个count[i]保存了捅里的硬币数目，我们把这种方法称为*桶实现法*。

这个方法虽然好，但是前提是只有6种面值。如果1分、2分、3分、4分、...99元9角8分、99元9角9分、100元整这1万种面值的硬币都有，那就需要1万个盒子。如果扔的1,000,000个硬币的面值全部不同，那么这个新方法就没有任何优势了。

讲了这么多，只是想说明一个问题：相同的外特性（自动储钱罐）可以用多种结构（小机器人的程序）实现，它们的时间效率可能不同，空间开销也可能不同（新程序需要用附加盒子）。事实上，存在一个更好的结构来实现这个储钱罐，它就是原书中介绍的*堆实现法*。不管面值有多少种，在有1,000,000个硬币的情况下也最多只需要不到一秒钟。

小测验

1. 解释数据结构的外部特性和内部结构。
2. 描述自动储钱罐的外部特性和两种内部结构：无序数组实现和桶实现。
3. 如何评价同一种外部特性的不同内部结构之间的优劣？

1.5.4 抽象数据类型

抽象数据类型(Abstract Data Types, ADT) 是只通过接口访问的数据类型（数据类型是指值集合和值上的操作集合）。使用ADT的程序叫**客户端(client)** , 指定数据类型细节的程序叫**实现(implementation)** 。接口不是透明(opaque)的, 即客户端无法看到（往往也不关心）实现, 更无法直接修改ADT的内部结构。换句话说，可以用接口完全描述一个ADT，即定义ADT所支持的操作。例如刚才的储钱罐就是一个**优先队列(priority queue)** ADT，每个硬币的优先级就是它的面值。每次优先级最大的出对。基本优先队列的操作如下：

bool empty()：判断队列是否为空
void insert(i, p)：往队列里加入一个元素i，优先级为p
int getmax()：取得队列里最大元素并把它从队列里删除

每次投入一枚面值相当于执行操作insert，按一次按钮相当于先执行empty，如果队列不空，则执行getmax。

队列和栈可以看作优先队列的特殊情况。

队列(queue) 的特点是先进队的元素先出队，好比在食堂排队买饭。用吸管喝饮料的情况也是这样，最先被吸到吸管中的饮料最先离开吸管。队列的**先进先出(First In First Out, FIFO)** 特性体现了元素的公平性，在算法中往往被用来放置还没来得及处理的，应维持某种顺序的元素。这个特点是广度优先遍历算法的基础，因为所有待扩展结点需要按照深度从小到大依次处理。

栈(stack) 的特点是后进栈的元素先出栈。只有一端允许操作的结构往往都是这样。例如，往抽屉里放东西，先放的东西被压在最底下，只能后出栈；后放的东西在最上面，先出栈。栈的**后进先出(Last In First Out, LIFO)** 特性体现了元素的局部性，比如你本来要往抽屉里放10本书，放了8本以后我可以借你的抽屉暂时放两本书，两分钟以后再拿走。除了在时间上耽误了两分钟外，我的动作对你没有一点影响。而

这在队列里是行不通的，因为一旦我把东西放在了队列中，必须由你把前面的元素全部移出队列才能拿回我的东西，而不能“自给自足”的“借用”。这个特点是递归、表达式处理和栈统计算法的基础。

如何把队列和栈看作优先队列的特殊情况？在队列中，把优先级设为入队时间的减函数；在栈中，把优先级设置为入队时间的增函数即可（想一想，为什么）。不过这只是一种可行的方案，并不是最好的。事实上，由于队列和栈的特殊性，通常用数组来实现，插入和删除都是常数级的。

小测验

1. 什么是ADT？ADT描述的是数据结构的外特性还是结构？
2. 描述基本优先队列的ADT，并为它设计几个新的操作成为扩展优先队列。
3. 什么是栈？什么是队列？如何用优先队列来实现栈和队列？一定要用优先队列来实现它们吗？

1.5.5 时间复杂度

在第一章中，我们提到过算法分析，也许你还记得$O(n)$这样的记号。本节将对时间复杂度记号做进一步介绍。对于单一操作（如加法）的算法, 我们有以下公式：

运行时间= 操作时间×操作次数[25]

在这个式子里，操作时间取决于计算机，而操作次数取决于算法。在刚才的钱罐的例子中，小机器人拿硬币、打开盒子的时间取决于机器人的硬件，而需要拿多少个硬币、打开多少个盒子却只和算法有关。

我们的目标是分析算法特性, 估算出操作次数。这样，对于典型的计算机速度，我们也可以估算它执行某个程序的时间；对于同一台计算机，改变输入时估算它运行时间的变化。

对于第一个钱罐，每次getmax操作时，小机器人一共会检查多少个硬币？答案是：这取决于钱罐里有多少个硬币。设里面有n个硬币，则小机器人会检查n个硬币。换句话说，算法执行的操作数往往和一些参数有关，我们把这些参数称为规模。

[25]不考虑cache和其他与体系结构有关的问题。

$\log n$	$\sqrt{n}$	n	$n\log n$	$n(\log n)^2$	$n^{\frac{3}{2}}$	n^2
3	3	10	33	110	32	100
7	10	100	664	4414	1000	10000
10	32	1000	9966	10^5	31623	10^6
13	100	10000	10^5	$2*10^6$	10^6	10^8
17	316	10^5	$2*10^6$	$3*10^7$	$3*10^7$	10^10
20	1000	10^6	$2*10^7$	$4*10^8$	10^9	10^12

表 1.4: 典型的化简结果和各规模下操作数的值

秒数	10^2	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}	10^{11}
时间	2分钟	3小时	1天	10天	4个月	三年	三十年	三百年	Never

表 1.5: 秒数的含义

每秒的运算次数	规模N=1,000,000			规模N=1,000,000,000		
	n或$n\log n$	n^2	n^3	n或$n\log n$	n^2	n^3
10^6	几秒钟	几星期	Never	几小时	Never	Never
10^9	瞬间	几小时	几十年	几秒钟	几十年	Never
10^{12}	瞬间	几秒钟	几星期	瞬间	几星期	Never

表 1.6: 各种计算机速度下各种算法解决各种规模问题的时间

如果规模为n，程序一的操作数为$3n+7$，而程序二的操作数为$10n^2+5n+23$，显然程序一比程序二好。对于一般情况，如何判断两个程序哪个更好呢？显然，可以先数出两个程序的操作数目，再加以比较，但是对于复杂的程序，很难写出完整的表达式，更别说比较了。在算法分析理论中，我们使用渐进的方法，先把操作数化成简单的形式，然后比较它们的阶。

简化的方法是：只保留最大项，忽略系数。比如程序一的3n+7化简后的结果为n，程序二为n^2，而$2^{n+7}+10n^{50}+987\log n$ 化简为2^n。显然这样化简忽略了很多因素，但是它保留最主要的项，方便比较。典型的化简结果和各种规模下操作数的值如下表：

从上表可以看出，$\log n$是一个非常小的函数，因此n和$n\log n$差别不太大，而n和n^2有比较大的差距。

下面这个表把“秒数”转换成了更容易理解的时间：

下面是各种计算机速度下，用各种算法解决各种规模问题的时间表

实现方法	Empty	Insert	Getmax
无序数组	$O(1)$	$O(1)$	$O(n)$
桶	$O(1)$	$O(1)$	$O(m)$
堆	$O(1)$	$O(\log n)$	$O(\log n)$

表 1.7: 存钱罐的三种实现方法比较

第二行的计算速度比较接近于目前常用的PC机。事实上，表里的几个算法（n，$n\log n$，n^2和n^3）在很多问题里是非常好的算法了。如果不幸设计出2^n的算法，那么规模$n = 80$时已经足以让第三行那台比普通PC机快一千倍超级计算机运行到永远了。而如果是$n!$甚至n^n的算法，那么即使n只有二十出头，也永远运行不完。

在一些时候，算法的操作数不仅和规模有关，还和具体的输入数据有关。同样是规模n的数据，可能有的数据运行时间为n，有的数据为2^n。这时需要分别说最坏情况和平均情况的时间复杂度。

很多时候算法分析不能做到十分精确，我们往往不说操作数等于多少，而只说操作数的上限是多少，我们用记号$g(n) = O(f(n))$来表示当n充分大时，$g(n)$不超过$f(n)$的常数倍。未加说明的情况下，本书中提到的"渐进时间复杂度"一律采用这样的**大欧记号(Big-oh notation)**。如果$f(n)$为多项式（如n，$n\log n$，n^2），则称此算法为**多项式时间算法(polynomial-time algorithm)**，而像2^n，$n!$这样的算法称为**指数时间算法(exponential-time algorithm)**。从指数算法到多项式算法是一种巨大的改进。可惜有很多问题至今没有发现多项式算法，也就是第一章中所提到的"难解问题"。NP完全问题就是这样一类问题，它的严格定义可以在原书中找到[26]。

有了这些知识，我们可以把前面"存钱罐"的三种实现比较如下：

其中，n为硬币个数，m为不同的面值数。如果你觉得这张表没有前面所说的"有1,000,000个硬币，面值有6种，检查一枚硬币需要0.01秒，打开一个桶需要5秒，则当用户按下按钮时，三种实现方法分别需要让用户等2.8小时、半分钟和不到一秒"直观和清晰的话，请重新阅读本节。以后对于算法的分析都会采取上表的形式。

在网站中可以找到三种实现的完整代码robot.cpp，读者自己设置的初始的硬币个数n、面值种类m和Insert 和Getmax的次数I、G，在不同情况下比较三者的运算速度。

[26]需要特别说明的是：这些分析都是针对图灵机模型的，适用于现在的通用计算机。生物计算机、量子计算机等有它们自己的计算模型，算法设计和分析都有很大不同，本书不作讨论。

本节比较理论化，如果读者理解有困难的，可以参考本书的配套ppt，但至少需要记住

- 算法的时间效率可以用渐进时间复杂度表示，如：1（常数，constant），$\log n$（对数，logarithm），n（线性，linear），$n\log n$，n^2（平方，quadratic）等
- 时间复杂度通常使用大欧记号，表示上限(upper bound)。如$O(n)$，$O(n\log n)$等
- 不同时间复杂度的算法在效率上可以相差非常远，这样的差距不是简单的提高计算机硬件速度所能弥补的。
- 时间复杂度的阶分两大类：多项式时间和指数时间。从指数时间到多项式时间是一个巨大的改进。
- 有很多问题至今没有发现多项式算法，例如NP完全问题就是这样一大类问题

时间复杂度分析本身不是一个简单的话题，本小节不作深入讨论。只要能读懂别人的算法分析结果，区分指数算法和多项式算法，本小节的主要目的就达到了。

小测验

1. 忽略体系结构方面的因素，程序的运行时间主要由哪两个因素决定？其中哪个由计算机决定，哪个由算法决定？我们主要关心哪个？
2. 什么是规模？除了规模外，还有什么因素将会影响到操作次数？
3. 什么是大欧表示法？如何用大欧表示法描述算法的时间复杂度？时间复杂度的阶分哪两大类？每大类有哪些常见的表达式？

1.6 语言和数据结构小结

本章的后半部分介绍语言和数据结构。这些内容可以分为三部分，分别介绍C++基本概念、C++基础语法和数据结构概念。其中第一小节从概念和使用两方面介绍了C++语言最基本的特性，包括：

文件 为了避免“用键盘敲10000个数”这样的情况所采用的数据载体。输入文件保存程序所需要的输入数据，程序执行完毕后，输出结果保存在输出文件中。

黑盒测试 给程序多组标准输入数据，运行程序后检查输出数据是否正确。这种方式不关心程序的内部结构，无法完美的测试出程序的错误，但由于客观性和可自动完成的特性而被广泛的应用。

输入/输出流 像水流一样的数据流，数据必须按顺序读出/写入。

变量 存放数据的小盒子。每个变量都有自己的类型和名字，比如int型的变量a可以保存-2,000,000,000到2,000,000,000的整数。

表达式 把变量和运算符连接起来的式子，类似于数学式，但必须是可计算的。

变量的使用 变量必须先声明，再使用。使用前要赋值，否则变量将会拥有不确定的值。

程序的动态执行 程序的执行过程是动态的。在任意时刻都有一个当前行，执行完当前行后转移到下一行，默认是紧跟其后的那一行，后面将学到让程序跳转执行的方式。每个变量在任何时期有且只有一个确定的值，这个值将随程序的执行发生变化，逐步得到我们想要的结果。分析程序数据的变化是一种重要的分析手段。

第二小节用约15页覆盖了C++语言基本语法的主要内容，涉及到词法、进位制、内存、数据类型、变量的声明与使用、运算符、表达式、函数、控制流、程序结构的主要内容。

词法 C++有四大类没有交集的词：标识符，关键字，文字和符号。程序被看作是单词流，单词间的空格无关紧要。

进位制 在计算机内部，数用二进制表示。数的各种进位制具有相同的本质，只是表现形式不同，运算规则也不同。二进制和十六进制可以方便的进行换算，n个比特的数一共有2n个。

内存 计算机里用内部存储器（简称内存）保存数据，内存的单位是字节，每个字节包含8个比特，因此有$2^8 = 256$ 种不同的字节。内存有两种访问方式：读和写。读操作不影响内存单元里的内容，而写操作会修改内存并使以前的内容永远丢失。为了访问内存，必须（也只需）知道内存单元的地址。每个字节都有一个地址。

数据类型 数据类型是值集合和在此集合上的操作集合。每个变量都有它的数据类型。一个变量可以占多个字节。所有数据归根结底都是整数，不同的数据类型只是对整数的解释和运算规则不同，没有物理上的本质差别。常用的数据类型有：整数、实数、字符、布尔。相同类型的数据可以构成数组。

变量的声明与使用 变量必须先声明，后使用。声明时需要指定变量名和数据类型，还可以进行初始化。变量的最基本操作是赋值，它应该被理解为把一个值放到一

个盒子里占个动作，而不是陈述两个式子相等这一事实。

运算符和表达式 常见的运算有算术运算、关系运算、逻辑运算和位运算。算术运算由于除0错误、溢出和浮点误差（即实数误差）的存在，需要小心对待。运算符有优先级和结合性的区别，它们影响到表达式的计算顺序。带有副作用的运算符最好不要在复杂的表达式里使用，以绕开复杂的表达式求值规则。

函数 函数最好的地方在于重用性。知道了函数的输入和输出就可以使用它了。在这个意义下，算法可以封装到一个函数里，而不需要每次都写完整的程序。多输出的函数可以用结构体包装所有输出，也可以用传参方式的参数。调用函数虽然省事，但是一定要注意函数的副作用。一个比较好的方法是尽量只使用局部变量。

控制流 除了默认的顺序执行外，分支和循环是另两类常用的控制流。分支语句有if-else和switch两种，它的意思是“如果xx，那么yy，否则zz”，循环有while、for和do-while三种，它们的意思都类似于“如果xx，那么一遍遍的执行yy，直到xx不再满足”。

程序结构 过程式的C++程序由若干函数构成，每个函数由若干语句组成，语句有声明语句、表达式、复合语句、分支语句和循环语句等几种。复合语句广泛的用在分支和循环语句当中。程序的入口在main()函数中。

第三节介绍了数据结构。本节的目的有两个：一是从概念层次上介绍数据结构，二是让读者建立算法分析的基本概念。

逻辑结构 逻辑结构体现了数据间的关系。常见的逻辑结构有线性表、树和图。线性表里的元素只有前驱后继关系，树里的元素有父子和层次关系，而图里的元素可以有更为广泛的关系。

物理结构 物理结构体现了逻辑关系的实现方式，通常有顺序结构和链结构两种。顺序结构往往容易随机访问但修改的代价大；链结构往往容易修改结构但随机访问困难。

外特性和内部结构 不管是逻辑结构还是物理结构，有一些对于外界来说是可见的，称为外特性，另一些是不可见的称为内部结构。相同的外特性可以对应不同的内部结构，它们的性能可以有很大差异。

抽象数据类型 只考虑外特性，用一组接口函数所描述的类型称为抽象数据类型，即ADT。从ADT的定义可以看出它所支持的操作种类，但是无法知道各个操作具体性能。通常用各个操作的时间复杂度表来评价ADT的实现。

优先队列 基本的优先队列是一个元素集合，它所支持的操作有：empty，insert和getmax。每个元素都有一个优先级，每次只能取优先级最大的元素并从集合中删除。

栈和队列 栈是后进先出表，而队列是先进先出表，虽然二者都是优先队列的特殊情况，但是往往不用优先队列实现。

时间复杂度 通常用时间复杂度来衡量算法的时间效率，一般采用大欧表示法，它指出了算法的时间复杂度上限。从阶来看，复杂度分为多项式算法和指数算法两大类，其中前者比后者优秀很多。

这三节的内容很多，但并不是速成教材——你应当花足够的时间来实践这些内容。本节不能让一个初学者学会一门语言，却是一个学习方法的指导。即使对于已经接触过并有一定经验的程序员，理清本章的脉络也是很有好处的。

第2章　问题的复杂性

是不是对于任何一个问题，都能找到合适的方法求解？问题本身有没有固有的复杂性的差别？本章试图从多个角度分析这个问题，重点讨论问题本身而不是具体的算法。

2.1　时间下界

到现在为止，我们考虑的是如何设计算法来解决问题。对于一个确定的问题，设$T_A(X)$表示在输入X下算法A的运行时间，那么对于规模n的输入X，算法A的最坏情况运行时间为$T_A(n) = \max\{T_A(X)\}$，其中|X|=n。前面我们的任务都是：设计A使得$T_A(n)$在渐进意义下尽量小。现在我们把视野放宽，考虑问题p本身的复杂性，即在所有能解决问题的算法A中取TA(n)最小的，即$T_p(n)=\min\{T_A(n)\}$，其中A取遍所有能解决p的算法。

每设计出一个算法[1]A，假设它的最坏情况时间复杂度为O(f(n)，则可以得到

$$T_p(n) <= T_A(n) = O(f(n))$$

它是p的复杂度上界。算法越好，这个上界越紧。这相当于是在回答：这个问题到底有多容易。本节从反面考虑：这个问题至少有多难？即需要给问题建立下界f(n)，使得每个解决p的算法的最坏情况时间复杂度均为Ω(f(n))，而没有时间为o(f(n))的算法。

2.1.1　计算模型

在下界中，我们需要考虑“所有算法”，这通常是困难的，因为没有对算法的通用的、可操作的定义。我们必须首先正规定义算法是什么、如何度量它的运行时间。这样的定义称为**计算模型(computation model)** 。

[1] 这里只是说一般意义下的算法，不包括概率算法、近似算法等

决策树(decision tree) 是一个常用而强大的计算模型。决策树是一棵树，每个内部结点对应一个询问(query)，是关于输入的一个问题，而每条出边对应于一个可能的回答。使用决策树的方法很简单，从根开始问问题，根据回答走到相应的儿子结点，直到到达叶子。下面是一个典型的决策树。

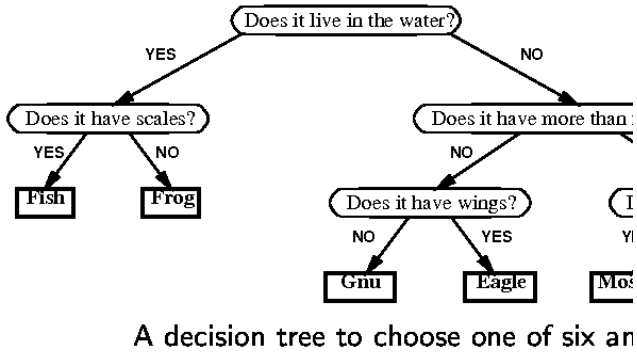


图 2.1: 决策树

字典问题 给一个数组A和一个元素x，判断x是否在A中出现。如果出现，给出位置。一种方法是先给A排序（附加记录每个元素的原始位置），然后二分查找x。二分查找过程可以看成一棵隐式排序二叉树，如图(a)，对应的判定树如图(b)。

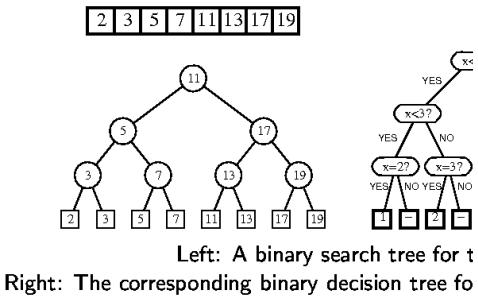


图 2.2: 字典问题的隐式排序二叉树和判定树

每种输入需要的询问数为从根到相应叶子的询问次数，而最坏情况的询问次数是树的高度。由于在这里完全忽略了其他计算，假设判定输入所属等等价类后可以立即得到结果，因此它是一个正确的下界。除了刚才的二元判定树外，还可以是3元、4元、...直到k元。在后面的讨论中，我们假设k是常数。

对于一棵k元判定树，假设它有k种不同的输出，那么至少有k个叶子（一个叶子只能有一种输出），因此树的高度至少为$[\log_k N]=\Omega(\log N)$。对于字典问题，一共有n种可

能的输出，因此问题的复杂度下界为Ω(logN)。由于二分查找是O(logN)，恰好和下界吻合，因此在渐进意义下它是最优的。

矛盾？ 前面介绍过哈希表，它可以在期望O(1)的时间复杂度里找到解。这和刚才的下界矛盾吗？不矛盾，因为在这里每个询问的回答数k并不是常数。当回答数k不是常数时，你甚至可以询问“哪个i满足A[i]=x？”这样直接得到答案。这不是不可能的，但它取决于计算模型。这个例子告诉我们：计算模型的选择非常重要。

基于比较的排序 由于在决策树模型中，我们无法描述如何“移动元素”，所以我们把排序问题修改为：给出每个元素i在排序后数组的序号p_i。由刚才的结论，如果每次只能比较两个输入元素，则结果只有小于、大于和等于三种，k=3。由于输出有n!种，因此下界时Ω(log(N!))的。由Stirling近似公式，

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right) > \left(\frac{n}{e}\right)^n \tag{2.1}$$

因此log(n!)¿[nlogn - nloge]=Ω(nlogn)。可以得出结论：在基于比较的排序中，归并排序在渐进意义下是最优的。但在后面的章节中我们会学习其他排序算法，如基数排序和桶排序，它们并不是基于排序的，因此不满足这个下界。

代数决策树 在排序问题中，我们考虑了的判定树是非常弱的：它只允许直接比较两个数的大小。下面的代数决策树要强得多，而且返回值是布尔的，非真即假。形式定义如下：一个在有n个变量$x_1, x_2, ..., x_n$集合上的代数决策树是一棵具有这样性质的二叉树，它的每个顶点用如下方法标记一个语句，与每个内结点联结的实质上是一个测试形式的语句。如果$f(x_1,x_2,...,x_n)=0$则转到左孩子，否则转到右孩子。这里=还可以换成¡和¡=。另一方面，一个yes或no的回答和每个叶子结点联系。

代数决策树的阶 对于某整数d¿=1，如果所有与树的内部结点联结的多项式次数最多为d，称此代数决策树的阶为d。如果d=1，称此树为线性代数决策树。设Π是一个判定问题，它的输入是n个实数序列$x_1, x_2, ..., x_n$，则与联结的是一个n维空间E^n的子集W，使得点$(x_1, x_2, ..., x_n)$在W中当且仅当问题Π在输入为$x_1, x_2, ..., x_n$时回答是yes。称代数决策树T判定W的成员资格，如果每当计算起始于T的根和某点$p=(x_1, x_2, ..., x_n)$，控制最终到达一个yes叶子当且仅当$(x_1, x_2, ..., x_n) \in W$。

设#W是W的连通分量的个数。在线性决策树下，叶子对应于多个线性函数的交集，即若干个超平面开半空间和闭半空间的交，它一定是凸的，因此一定连通。这样，每个叶子对应一个连通分量，因此树的高度至少为[log(#W)]。在高阶情形下，每

个叶子可能对应多个连通分量，更复杂的数学分析导致了下面定理：

定理 设W是En的一个子集，d为一个固定的正整数，则接受W的任意一个d阶代数决策树的高度为Ω(log(#W)-n)。

刚才提到的基于比较的排序是一个线性决策树，每个多项式形如x_i-x_j¡=0。可以证明，在高阶代数决策树中仍有：排序问题的下界为Ω(nlogn)。

元素唯一性问题 作为代数决策树的例子，考虑下面的问题：给出n个实数的集合，判断它们是否有两个数相等。设W为任两维都不相等的点集，则1∼n的每个排列p(1), p(2), ..., p(n)对应于一个连通分量W_p={(x_1, x_2, ..., x_n) ｜$x_{p(1)}$ ¡ $x_{p(2)}$ ¡ ... ¡ $x_{p(n)}$}。显然任意两个连通分量都不相交，且它们的并就是W，因此#W=n!，根据刚才的结论，下界为Ω(nlogn)。

小测验

1. 什么是决策树？n个叶子的k元决策树的高度下界是怎样的？
2. 什么是代数决策树？证明线性代数决策树下下界为log(#W)并用此结论证明元素唯一性问题在此模型下的下界为Ω(nlogn)
3. 什么是计算模型？为什么计算模型的选取十分重要？

2.1.2 对立法

本界给出一个很有意思的下界寻找方法：对立法。这是一种非常基本的方法，一般用来确定一些最基本问题的下界。

寻找最大值 根据基本决策树，最大值有n种输出，因此下界是Ω(logn)。可很明显这个下界是不准确的。至少需要把n个值都看一遍才可能知道最大值，因此肯定是Ω(n)的。更严格地，假设输入并不是一组确定的数据，而是一个“对立机器”，它能生成可变的数据。这样一来，算法不能去“猜测”答案，因为即使猜对了，对立机器可以换一组数据，只要和先前给出的询问回答是相容的。在本题中，对立机器可以先准备一组数据使得x_i=i。每次询问x_i和x_j的关系时返回i和j的关系并把其中较小的那个做一个标记（表示它不能是最大值，否则和回答冲突）。

如果比较次数不超过n-2，则一定至少有两个未做标记，设其中最小的那个为x_k，那么如果算法输出是k，答案是错误的（应该是n）；如果算法输出n，则对立机器悄悄把序列变成1, 2, 3, ..., k-1, n, k+1, ..., n-1，使得算法的输出错误。由于变化后的新序列完全符合算法的所有询问结果，因此算法不能指控对立机器“耍赖”。怪只怪目前已有

的信息不能确定一个唯一的结果。

寻找子串01 用简单的对立法可知：判断一个01串是否含有1必须检查每个字符。那寻找子串01呢？不一定。下面证明串长n为奇数时不需要，偶数时需要。

n为奇数 检查第2, 4, 6, ...n-1个字符。如果存在i¡j使得S_i=0且S_j=1，则一定存在子串01（想一想，为什么），不用继续检查；如果所有字符形如1111...1000..000，则最后一个1和第一个0之间只有一个字符，不管它是啥都不可能构成01串，因此不用检查；如果所有字符都是0，第一个字符不用检查；如果所有字符都是1，最后一个字符不用检查。不管遇到哪种情况，都至少有一个字符不需要检查，最多只需要检查n-1个字符。

n为偶数 下面使用对立法证明必须检查所有字符。对立机器假装数据是11111...000，并维持两个指针l和r，使得两个指针中间的字符**全部没有**被检查过，且这些字符有偶数个。初始时l=0，r=n+1。由于中间有偶数个字符，因此r-l-1为偶数，即$r-l$为奇数。

图 2.3: 子串问题n为偶数时的对立法

如果算法询问一个字符i，如果在i¡=l则返回1；如果在i¿=r右边则返回0。否则一定在l和r中间。如果$r-i$为偶数，则返回1且把l改为i；如果$i-l$为偶数，则返回0且把r改为i。两者情况至少出现一种（想一想，为什么？），因此总能保持中间有偶数个字符。

考虑算法没有检查所有字符就退出的情况。如果算法回答Yes，则把输入串变为1111...0000，从而让算法返回值错误；如果算法回答No，有三种情况（注意A[l]和A[r]一定已经检查过）：

情况一 l前面有字符没有检查。把它改成0就构造出了一个01子串。

情况二 r后面有字符没有检查。把它改成1就构造出了一个01子串。

情况三 中间有字符没有检查。由于中间有偶数个字符，所有至少有两个字符没检查，让里面填一个01即可。

这样我们证明了寻找01子串需要检查所有字符当且仅当n为偶数。如果检查一个01串是否含有字符s需要检查所有字符，称s是**可逃脱(evasive)** 的。可以证明只有0和1对所有长度n 都是逃脱的。

图的连通性 判断一个图是否连通必须检查所有的边吗？直观的说，如果没有检查的边全部加在一起仍然不会连通的话，这些边就不用检查了，然而对立法可以避免这样的情况：记确定存在的边集合为Y，没有检查过的边与Y中的边合起来为集合M，询问边为e，如果M-{e}连通则返回“e不存在”（因此需要把e从M中删除），否则满足“e存在”。容易证明：在任何情况下：

1. Y是M的子图
2. M是连通的；
3. 如果M含有一个圈，则圈上任何一条边都不在Y内。
4. Y是无圈的（由前1,2,3可得）
5. 如果Y不等于M，则Y是非连通的。（因为M至少多一条边，而连通无圈图任意加一条边e都将得到一个带圈，此圈中除了e外的其他边都在Y中，和3矛盾）

如果算法并没有检查所有边，Y和M不相等，因此Y非连通而M连通。算法并不能区分开Y和M，因此无法确定图到底是连通还是不连通。

如果一个性质至少有一个图满足一个图不满足，称此性质是非平凡(nontrivial)的。如果一个图满足此性质蕴涵它的所有子图[2]满足此性质，则称此性质是单调的。

猜想*(Aanderraa, Karp, Rosenberg)* n个结点的图上非平凡的单调性质都是可逃脱的。

最大最小值 第4章介绍过求最大最小值可以只用[1.5n]-2次比较。下面证明这是最优的。类似于最大值问题，给每个数一个+标记（表示它有可能是最大值）和-标记（表示它有可能是最小值）。每次比较以后，如果两个数都有两个标记：清除较大值的-标记和较小值的+标记，而其他情况都可以选择一种回答使得只有一个标记被清除（例如，x有两个标记而y只有-标记。比较x和y时如果返回x¡y则会清除x的+标记和y的-标记，而如果返回x¿y则只能清楚x的-标记）。一开始一共有2n个标记，最终目标是2个标记（一个+一个-）。显然最多只有[n/2]次比较可以一次清除两个标记，因此至少共需要2n-2-[n/2]=[1.5n]-2次比较。

[2] 这里只生成子图，即顶点集不变，边集为原边集的子集

小测验

1. 什么是对立法？为什么可以用它来求出问题的下界？
2. 给出“在一个偶数长度的01串中寻找子串10”的下界。
3. 在最大最小值问题的下界证明中，“对立”体现在何处？

2.1.3 归约法

本节介绍归约法。它是一种间接的方法，常用来确定一些较为复杂问题的下界。

线性时间归约 设A是一个问题，已知它的下界为Ω(f(n))，其中n=o(f(n))，例如f(n)=nlogn。设B是一个问题，如果可以

步骤一 把A的输入转化为B的输入

步骤二 求解问题B

步骤三 把输出转化为A的输出

且步骤一和步骤三的都可以在线性时间（即O(n)）内完成，则称问题A在线性时间内**归约到(reduce to)** 了问题B，并且表示如下：$A \propto_n B$。直观的说，B至少和A一样难（它可能严格比A难）。因此B的下界至少也为Ω(f(n))。再强调一次：

从A线性归约到B，则可以用B来解决A。这说明B至少和A一样难

SORTING $\propto_n$ **CONVEX HULL**。凸包问题将在第三部分详细介绍，这里只证明它至少和排序问题一样难。构造点(x_i, x_i^2)，则凸包上点的顺序就是排序的结果。

一个更有说服力的框图如下。

ELEMENT UNIQUENESS $\propto_n$ **CLOSEST PAIR**。第5章介绍的最近点对问题至少和元素唯一性一样难，只要构造点$(x_i, 0)$，则元素唯一当且仅当最近点对距离不为0。

SORTING $\propto_n$ **EUCLIDEAN MINIMUM SPANNING TREE**。构造点$(x_i, 0)$，则最小生成树为连接它们的线段。通过从最左点开始遍历树，可以在O(n)时间给出排序结果。

这样，我们通过线性时间归约的方法证明三个问题的时间复杂度下界均为Ω(nlogn)。事实上，三个问题都已经找到了O(nlogn)的算法（参见附录中的算法索引），因此这些算法在代数判定树模型下是渐进最优的。

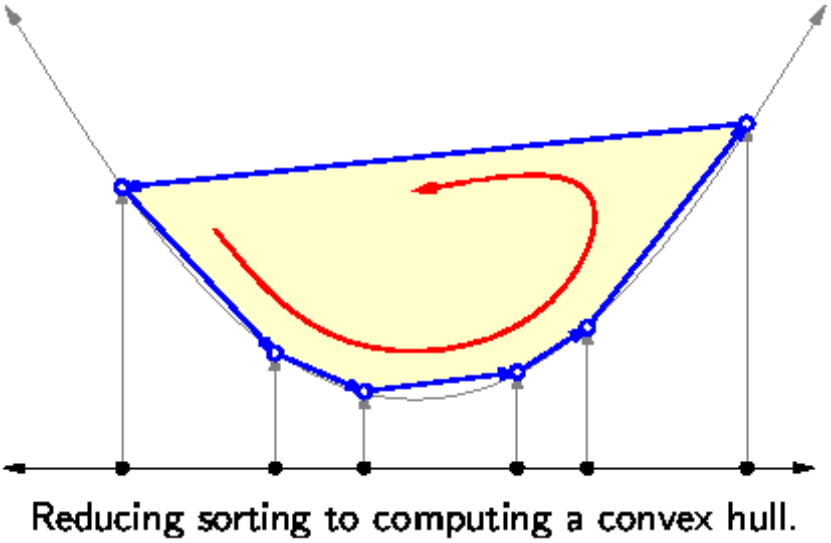


图 2.4: 凸包的归约

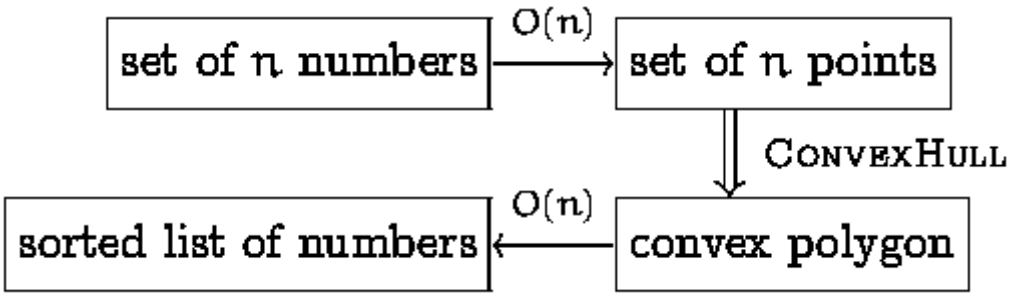


图 2.5: 凸包归约框图

3SUM问题 给n个实数，是否存在三个数x_i, x_j和x_k使得$x_i+x_j+x_k=0$？最简单的方法是一个三重循环判断，时间复杂度为$O(n^3)$。如果先排序，则可以在枚举i¡j，然后用二分查找判断它的和$-x_i-x_j$是否存在，时间复杂度为$O(n^2\log n)$。还可以进一步优化。规定i¡j，则枚举i后随着j的增大，x_i+x_j也将增大，只需要从上次的k开始从大到小枚举k即可，时间复杂度为$O(n^2)$。下面是示意图：

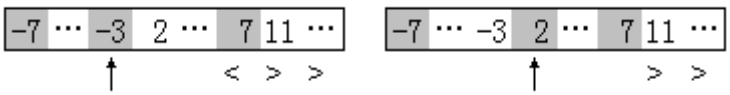


图 2.6: 3SUM问题的平方算法

左图是x_i=-7, x_j=-3的情形，枚举到x_k=7时停止，因为后面的x_k一定过大，不可能有解。现在j增加1，x_j=2，仍然从x_k开始枚举，因为x_k之后仍然过大，不可能有解。由

于固定i时j和k一共只扫描两次数组，因此总时间是$O(n^2)$的。事实上，在一些相对比较弱的计算模型下，可以证明$\Omega(n^2)$是下界。

共线点问题 给n个点，是否有三点在同一条非水平的直线上？如果我们承认3SUM问题的下界是$\Omega(n^2)$，那么共线点问题也是的$\Omega(n^2)$，因为有如下从3SUM到共线点问题的线性时间归约：把3SUM的每个输入元素a拆成三个点(a, 0), (-a/2, 1)和(a, 2)。这样，n个数变成了3n个点，分布在三条水平直线y=0, y=1和y=2上。若三点(a, 0)、(-b/2, 1)和(c, 2)共线，斜率既为-b/2-a也为c+b/2，整理得a+b+c=0。反过来，若a+b+c=0，此三点一定共线。这样，我们完成了从3SUM到共线点问题的线性归约。3SUM问题是一个很基本的问题，它可以归约到很多问题，这些问题称为3SUM-难度问题。在计算几何部分，我们将介绍更多此类问题。

线段分离问题是3SUM-难度的 线段分离问题询问是否存在一条直线把n条线段分离成两个不相交的非空集合。为了完成归约，我们先把3SUM问题归约到共线点问题，并把每个点换成一个孔，穿过空（共线）才可以分离线段，如下图：

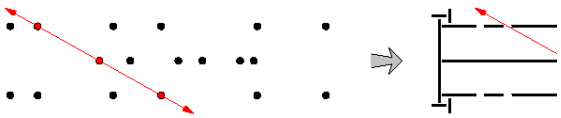


图 2.7: 线段分离问题是3SUM难度的

由于有三行，每行n个孔，因此有3n+3条线段。为了避免直线“从旁边走”。需要给左右加上“护栏”。为了避免直线把护栏单独分离出来，还需要在上下加上“护栏的护栏”（水平短线段）以及“护栏的护栏的护栏”（竖直短线段），一共10条护栏。这样，n个数被转化成了3n个点，最后变成了3n+13条线段。孔需要无限小，但计算机无法直接处理无限小的数。好在可以在O(nlogn)时间内用最近点对距离求出一个“足够小”的大小，把孔的直径设为此值即可。这样，我们证明了线段分离问题是3SUM-难度的。

路径规划问题是3SUM-难度的 考虑如下路径规划问题：机器人是一条长长的线段，障碍是线段集合。给出机器人的初始、目标位置和朝向，问路径规划可以完成吗？为了完成归约，我们从线段问题开始，构造下图。

把机器人设计成足够长的，则它必须通过共线的孔。可以画出如下框图：

需要特别注意的是：前面提到的已有的3SUM的下界不能直接搬到这里，因为在路径规划问题中，可以使用的计算模型是很强的，没有理由使用“弱模型下的3SUM下

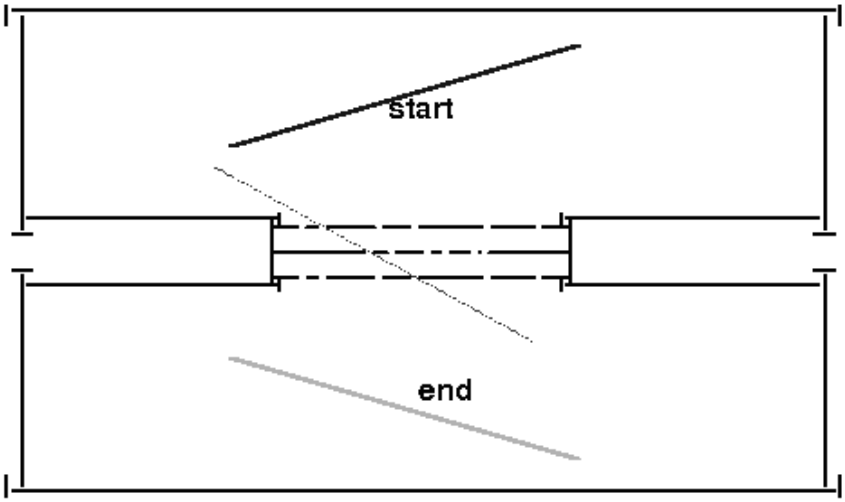


图 2.8: 路径规划问题是3SUM难度的

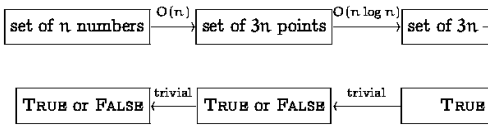


图 2.9: 路径规划问题的归约框图

界”。希望读者对归约中涉及到的模型差异引起重视。

小测验

1. 完成以下线性时间归约：SORTING到TRIANGULATION；BINARY SEARCH到NEAREST POINT; CLOSEST PAIR到ALL NEAREST POINTS。
2. 归约涉及到的两个问题可能会有怎样的模型差异？这种差异将带来怎样的问题？
3. 证明路径规划问题是3SUM-难度的。这是否意味着路径规划问题是$\Omega(n^2)$的？

2.2 NP完全问题

前面提到过，多项式时间算法比指数时间要好，因为它通常增长得不太快。一个很自然的问题是：是否所有问题都有多项式时间算法？答案是否。事实上，存在一些问题，不管给多少时间都无法解决[3]，更别说多项式时间了。本章重点讨论一个很有意

[3] 如著名的“停机问题”

思的问题类：NP完全类。这些问题不知道有没有多项式时间算法，虽然很可能没有。没有人找到它们的多项式时间算法，但也没有人能证明它们不可能在多项式时间内被解决。这类问题的另一个有意思的特性是：只要发现了其中一个问题的多项式时间算法，通过前面介绍的归约法（这里是多项式时间归约而不是线性时间归约）可以得到其他所有问题的多项式时间算法。

2.2.1 Cook定理和3-CNF-SAT

在深入讨论之前，需要定义三类**判定问题(decision problem)**，即答案是“是”或“否”的问题。

P类问题 可以在多项式时间内正确的回答“是”或“否”的判定问题。
NP类问题 如果回答是“是”，则存在一个**证据(certificate)**，可以在多项式时间验证这个问题的答案的确是“是”。
co-NP类问题 如果回答是“否”，则存在一个证据，可以在多项式时间验证这个问题的答案的确是“否”。

显然P既在NP中也在co-NP中（不用管证据，重新算一遍即可），但是主要注意的是：NP并不一定等于co-NP，因为验证“是”和验证“否”并不是一回事。

计算复杂度理论的中心问题之一上：P=NP吗？目前还没有人知道。同样的，我们也不知道是否有NP=co-NP。但大多数人强烈相信两个等式都是不成立的，因此三个问题类看上去应该是下面这个样子：

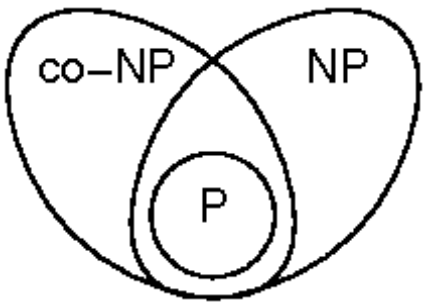


图 2.10: P, NP和co-NP的关系

类似于前面的线性时间归约，我们可以定义多项式时间归约。如果一个问题L满足：

1. L在NP中

2. 对于NP中的任意问题L'，L'可以多项式时间归约到L

则说L是**NP完全(NP-Complete)**的。这个定义是很直观的，因为如果一个NP完全问题得到完美解决，所有NP问题都将借助于归约得到完美解决。可问题在于：存在这样的问题吗？答案是肯定的。第一个被发现的NP完全问题是电路满足问题。

电路满足问题 给定一个只由AND，OR和NOT组成的n输入-单输出布尔组合电路，是否存在一个输入指派，使得电路输出为1？一个布尔组合电路不能包含有向环。

Cook定理 电路满足问题是NP完全的。

有了这个定理，只要电路满足问题可以归约到某问题L，那么任何NP问题都可以多项式归约到L （先归约到电路满足问题，再归约到问题L），我们称这样的问题为NP-难度的。显然，如果一个NP-难度问题同时也在NP中，它一定是NP完全问题。有了NP完全问题和NP-难度问题，复杂度类的图变得复杂了一些，如图8.1.2。

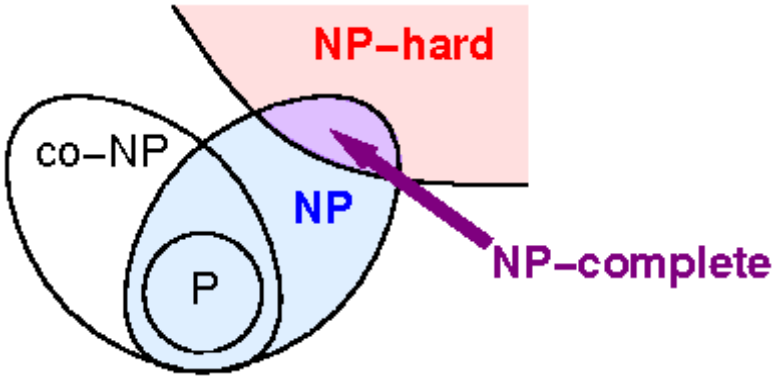


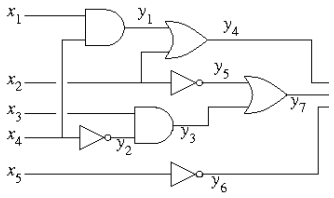
图 2.11: NP完全问题

由于我们并不知道P和NP的关系，所以这个图只是"大多数人认为"的，并不是严格被证明的。

*NP*完全性证明 有了Cook定理，我们可以分两步证明一个问题L是NP完全的。首先证明L在NP中，然后选择一个已知道的NP完全问题L'，并证明L'可以多项式归约到L（即证明L是NP-难度的）。这个证明方法是简单的，至少比证明Cook定理要容易得多。

SAT是NP完全的 **公式满足（formula satisfiability，SAT）** 问题的输入是一个布尔公式，由n个布尔变量x_1, x_2, ..., x_n和m个布尔连接词组成。布尔连接词可

以是AND，OR，NOT，蕴涵和等价，还可以有括号。使用刚才介绍的方法，可以证明SAT是NP完全的。SAT显然是多项式可验证的（只需要简单计算即可），因此SAT在NP中。为了证明它的NP-难度的，我们考虑把电路满足问题多项式时间归约到SAT。显然可以用构造性的方法用布尔公式来表达布尔电路，但公共项的存在可能会让这样的构造可能需要指数时间！借用动态规划的思想，增加一些中间变量，利用等价连结词吧电路写成简单的布尔公式。



$$(y_1 = x_1 \wedge x_2) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5$$

图 2.12: SAT是NP完全的

这样，我们完成了SAT的NP完全性证明。

3-CNF-SAT是NP完全的 考虑SAT的一种简单实例。用文字(literal)表示一个变量或它的非，子句(clause)表示若干文字的或，而CNF（conjunctive normal form）则是若干子句的并。如果每个子句恰好包含三个不同文字，则称它为3-CNF。需要注意的是，可以在一个子句中同时出现一个变量和它的非。为了证明3-CNF-SAT的NP完全性，可以把SAT归约到它，或者直接把电路满足问题归约到它。首先把一个k输入的门变成一棵斜的二叉树（每次让两个变量进行运算），然后按刚才的方法写出布尔公式，然后改写成CNF。注意到直接写出的布尔公式只有以下三种形式（非常好的形式！已经是CNF且每个子句最多三个文字）。

最后把CNF变为3-CNF，即把变量个数为1或2的加以补充，公式如下：

由于3-CNF-SAT是SAT的特例，因此它在NP中。这样，我们证明3-CNF-SAT也是NP完全问题。

小测验

1. 什么是P类问题、NP类问题和co-NP类问题？
2. 什么是NP完全问题和NP难度问题？二者的关系如何？如何证明一个问题是此二类问题？
3. 什么是3-CNF-SAT？用Cook定理证明3-CNF-SAT是NP完全的。

2.2.2 更多NP完全问题

本节介绍更多NP完全问题及它们的NP完全性证明。这些证明是有启发性的，而结论也是读者应当熟知的。

最大团问题CLIQUE 给一个图G，求包含尽量多元素的结点子集，使得这些点诱导出完全子图。我们从3-CNF-SAT归约到CLIQUE。假设有n个子句，给每个子句的每个出现的文字建立一个结点。一共有3n个结点（可能有重复的文字，每出现一次都要建立）。两个结点有边相连当且仅当它们出现在不同子句中且不冲突（冲突的情况只有一种：一个变量和它的非。它们是不可能同时满足的）。例如，公式

被转化到下图（注意观察不存在的边），每个子句的三个顶点被画在一起：

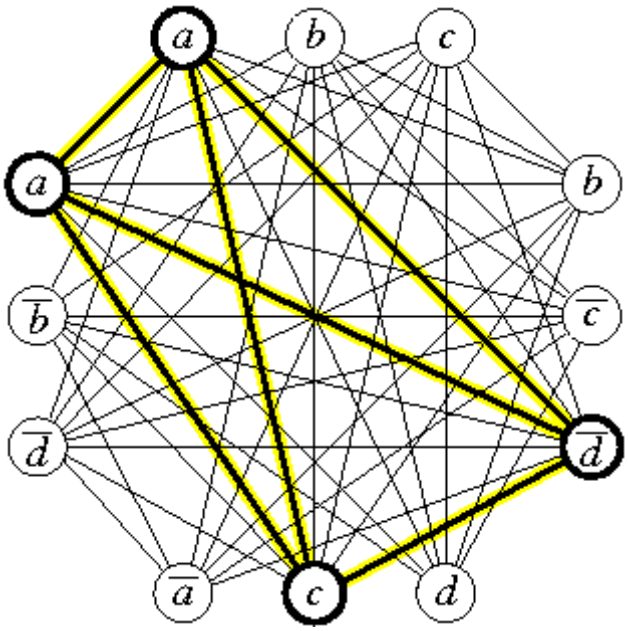


图 2.13: 最大团问题

如果原3-CNF可以被满足，那么恰好每个子句至少有一个文字为真，这些为真的文字互不冲突，因此在图上构成一个n点团。反过来，任何一个n点团都对应n个文字，它们在不同子句且互不冲突，因此可以构造出3-CNF的指派。

由于独立集是团的补，因此我们顺便也证明了最大独立集是NP完全问题。

顶点覆盖VERTEX COVER。顶点覆盖是一个结点集合，每条边至少有一个结点在此集合中。VERTEX COVER问题（简称VC）即寻找一个具有最少结点数的顶点覆盖。证明是简单的，因为对于图G的任意独立集I，V-I是顶点覆盖。类似地，对于任

意顶点覆盖C，V-C是独立集（想一想，为什么）。这样，只需要寻找最小顶点覆盖，就得到了最大独立集。

哈密顿回路HAM-CYCLE。一个图的哈密顿回路为包含所有顶点的简单回路。显然它是多项式可验证的，下面我们把VC归约到它：给定原图G和整数k，构造图G'使得G有大小为k的顶点覆盖当且仅当G'有H回路。这个构造非常有意思，它首先把原图的边(u,v)拆成如下的形状：

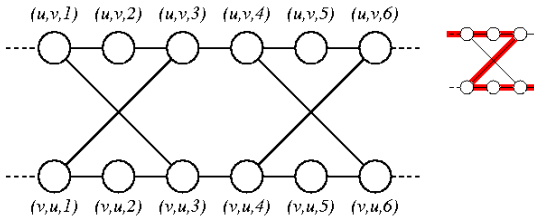


图 2.14: Hamilton回路的子结构

其中上下左右有四条边和外界相连。可以证明H回路穿过它只有右图三种形式（把它们记为形式a, b, c）。首先建立k个附加点，编号为1~k，然后对于原图中的任意结点u，把它的所有邻接边(u, v)所对应的新顶点穿在一起，然后让链上的第一个点连到所有附加点上，最后一个点也连到所有附加点上，如下图（这只是一个结点v所对应的子图，这里(w, v)、(x, v)等边暂时没有连接）：

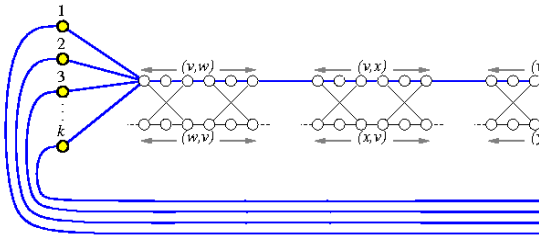


图 2.15: Hamilton回路的转化

如果G有大小为k的顶点覆盖(u_1, u_2, ..., u_k)，那么G'有Hamilton回路：1-¿u_1链-¿2-¿u_2链-¿...-¿k-¿u_k链。如果u在顶点覆盖中而v不在，则访问u-v后立刻要用形式(a)和形式(b)，访问v-u（因为不存在v链，此时若不访问以后再也无法访问），否则u-v和v-u应该分别在访问u链和v链时访问，即每次使用形式(c)。

这是H回路，因为每一条边都至少和一个u_i连接，也就在u_i链被访问。反过来，G'的任意一条H回路一定形如p_1-¿u_1链-¿p_2-¿u_2链-¿..-¿p_k-¿u_k链，其中p_i是1~n的

一个排列。每条u链访问了u的所有邻接边u-v（和v-u，如果v不在链中）。由于是H回路，所有u_i的所有邻接边应包含了G的所有边，故这些u_i是一个顶点覆盖。这个证明是直观的，更加严格的写法可以参考其他书籍。

下面是一个例子。{v, w}是一个顶点覆盖，G的H回路为：1-¿v链-¿2-¿w链。因此访问(v,u)时必须顺便访问(u,v)（因为不存在u链），而访问(v,w)时不能访问(w,v)而是把它留给w链。

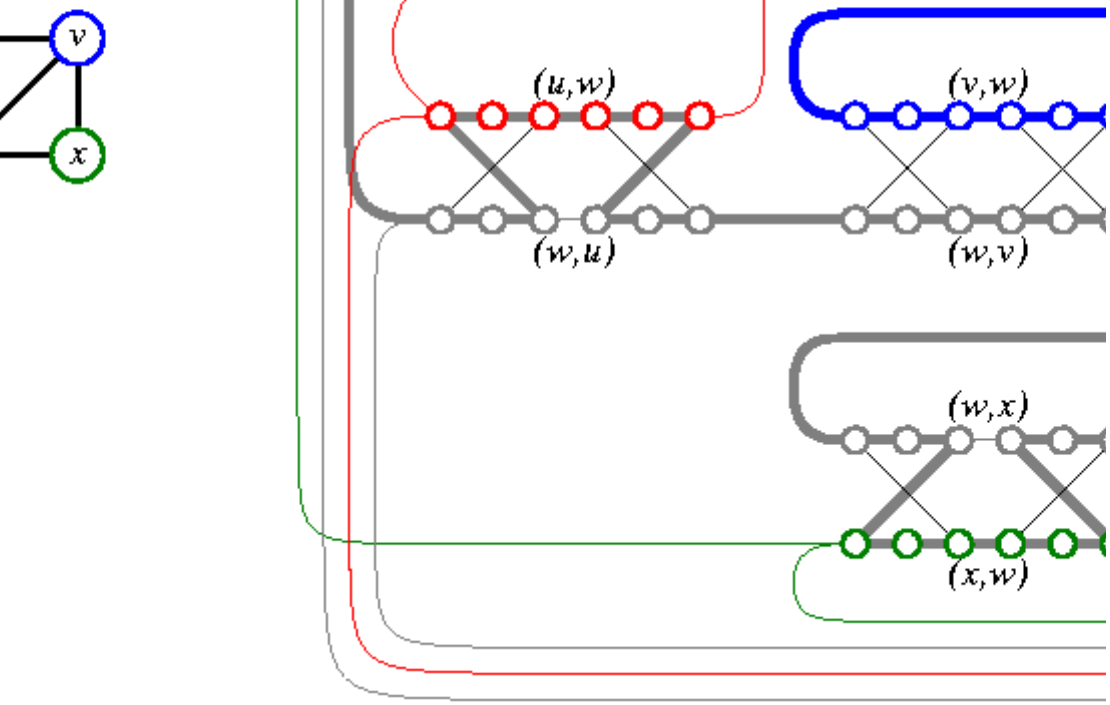


图 2.16: Hamilton回路转化举例

作为HC完全性的推论，TSP也是NP完全的。TSP的输入是一个完全图，每个边上有一个权，要求找到权和最小的HC。很容易把HC归约到TSP：只需要构造一个完全

图G'，把G中有的边权设为0，没有的边权设为1，运行TSP。如果G存在HC，TSP一定返回G'的一条权为0的HC。这样，HC归约到了TSP。显然TSP可以多项式时间验证，因此TSP也是NP完全的。

3着色问题3COLOR。给一个图G，是否可以给顶点3着色，使得任意两个同色顶点不相邻？显然它是多项式可验证的，因此只需要从3-CNF-SAT问题归约到它。首先建立一个所谓的真值三角形。对于每个变量和它的非建立所谓的变量三角形，然后对每个子句增加五个辅助结点形成子句结构，如下图：

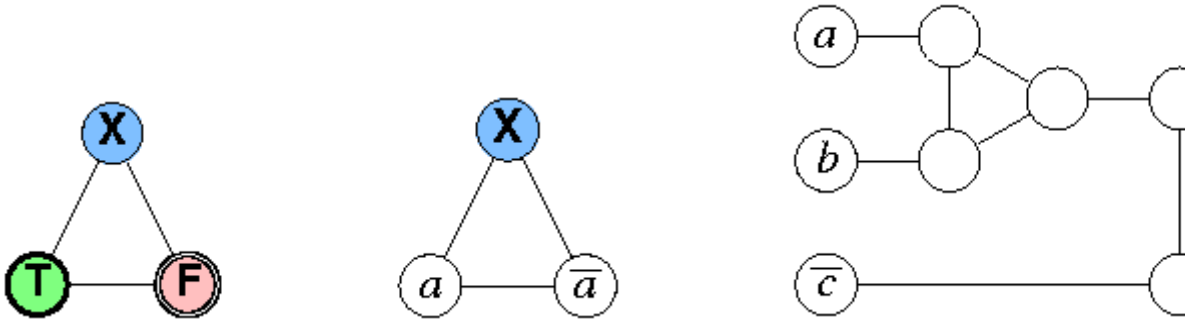


图 2.17: 3着色问题的基本结构

真值三角形保证了T、F、X分别是三种不同颜色，变量三角形保证了每个变量要么是真（T色）要么是假（F色）。子句结构保证了三个文字不能均为假（想一想，为什么？）。这样，原3-CNF-SAT有解当且仅当图G可三着色。

为了加深理解，下面给出

的完整图。

小测验

1. 通过怎样的归约，我们证明了CLIQUE，VC，HC，TSP和3COLOR是NP完全的？

2. 用最少的颜色数给图着色是NP完全问题吗？判定哈密顿路的存在性是NP完全问题吗？试通过本节的结论说明

3. 证明子集和数问题和0-1整数规划问题是NP完全的（从3-CNF-SAT归约）。

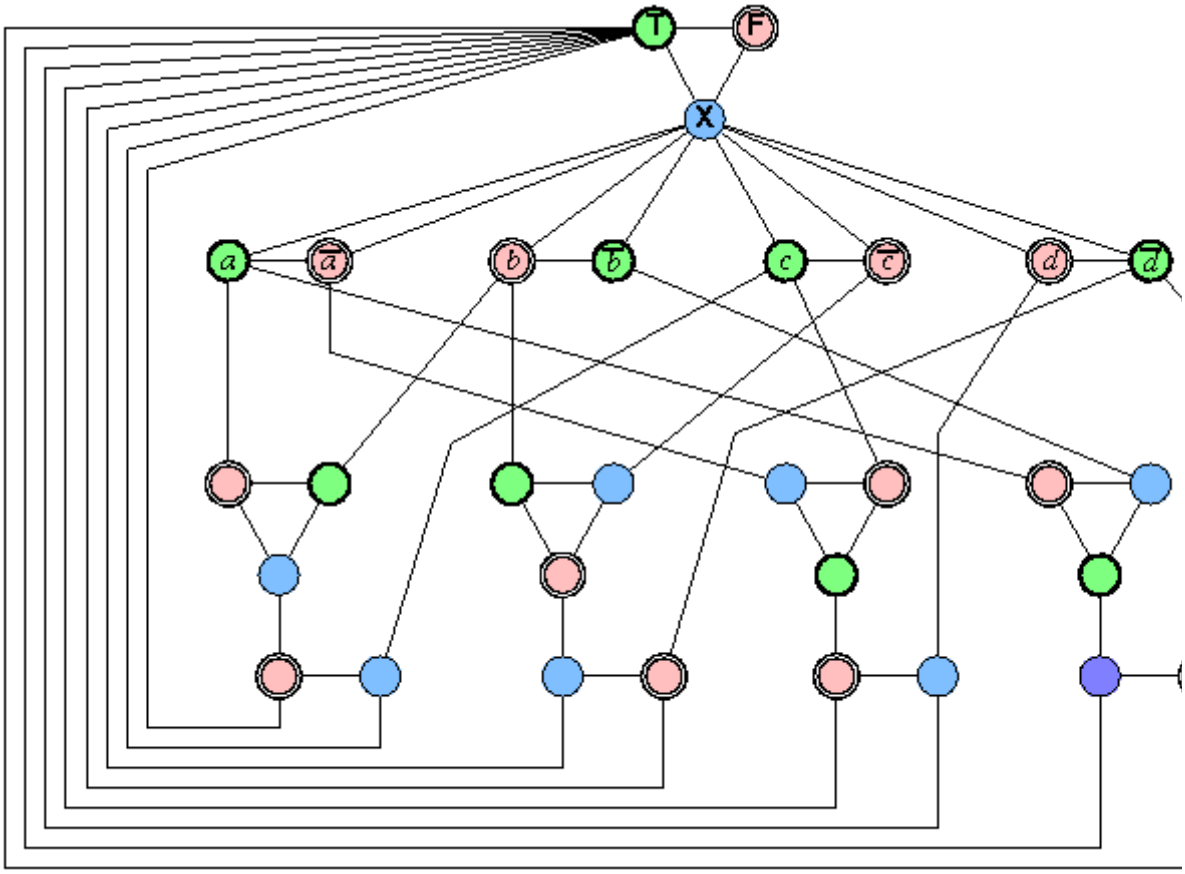


图 2.18: 3着色问题转化举例

2.3 图灵机和计算复杂性理论

上一节的NP完全理论虽然直观，但是不严密。我们没有给出Cook定理的证明，因为在证明这个定理之前需要给“问题”下一个严格定义，否则是没有办法说明什么是“NP问题”，更别提证明任何一个NP问题都可以多项式归约到它了。此外，对“算法”也需要进行严格证明，否则没有办法定义归约。如果说上一节是从感性上认识问题复杂性和NP完全理论，那么从这一节开始正式介绍相关理论。

2.3.1 问题和语言

在深入讨论之前，需要先对“问题”做一个严格定义。**抽象问题(abstract problem)** 是一个I和S的二元关系，其中I是**实例(instance)** 集合，S是**解(solution)** 集合。NP完全理论只考虑**判定问题(decision problem)**，即S={0, 1}。对于优化问题，

我们很容易改写出一个相应的判定问题："此题的最优解不超过x吗？"这个问题显然不比优化问题困难。

为了让计算机可以读取问题实例，需要把它进行**编码(encoding)** 。编码后的问题不再是抽象的了，我们称它们为**具体问题(concrete problem)** 。具体问题的每一个实例用一个01串表示。如果给定长度为n = |i|的具体问题实例i，某算法A可以在O(T(n))时间内得到解，则称该问题可在O(T(n))时间内被算法A解决。

抽象问题到具体问题的编码方式重要吗？当然重要！假设有一个问题，它的输入为一个整数k，运行时间为O(k)，那么整数k的编码方式将会影响到时间复杂度。如果k用k个1表示，那么输入规模n=k，因此运行时间是O(n)的；如果k用二进制表示，输入规模n=$\log_2$k，因此运行时间是O(k)=O(2^n)，不是多项式的！

既然如此，我们有必要规定一个"标准编码方式"，即用二进制表示一个整数，元素列表和分割符表示一个集合，并由此推出三元组、图、公式等标准数学对象的编码，并用¡G¿来表示数学对象G的标准编码。只要使用标准编码[4]，以后可以避开编码方式而直接讨论抽象问题，这给我们的分析带来了很大的方便。但读者应当警惕一些用非标准方法编码的问题。如有遇到，本书将会明确提到。

形式语言(formal language) **字母表(alphabet)** Σ是一个有限符号的集。在Σ上的**语言(language)** L是由Σ上字符构成的一些串的集合。例如Σ={0, 1}，则集合L={10, 11, 101, 111, 1011, 1101, 10001, ...}是Σ上的一个语言，它是所有素数的二进制表示集合。一般用ε表示空串，而Φ表示空语言。由Σ可以组成的所有串的语言记为Σ*，例如当Σ={0, 1}时Σ*={ε, 0, 1, 00, 01, 10, 11, 000, ...}。显然，Σ上的任何语言都是Σ*的子集。

语言的运算 由于语言是串的集合，集合论里的运算交和并可以直接拿来用。补语言被定义为Σ*-L，即所有不在L中的串组成的集合。两个语言L_1和L_2的连接被定义为所有形如x_1x_2的串的集合，其中x_1在L_1中，x_2在L_2中。L的**闭包(closure)** 也称**克莱因星(Kleene star)** ，定义为$L^*=\{\varepsilon\}\cup L\cup L^2\cup L^3\cup...$，其中$L^k$由L自己和自己连接k次而成（因为连接运算类似于乘法）。

由于有了编码，任何一个问题的实例集都是Σ*（保留不合法实例的编码，假装它是输出为0的合法实例，这样任何一个01串都是所有问题的实例），因此各个问题唯一的不同点是从实例x到解Q(x)的映射关系。换句话说，我们把问题Q看作一个语

[4] 或者与它多项式相关的编码方式（即可以在多项式时间内从一种编码转化到另一种编码）

言L={x∈ Σ* | Q(x)=1}。

算法和P类问题 给一个算法A，如果对某串x输出1，我们说算法**接受(accept)**串x；如果对它输出0，我们说算法**拒绝(reject)** 串x。算法A接受的语言L被定义为所有被A接受的串所构成的语言。请注意：即使一个语言L被算法A接受，算法A并不一定能拒绝所有不在L中的串，因为算法有可能永远执行不完！为了排除这种情况，我们定义语言L可被算法A**判定(decide)** ，如果算法A接受L中的所有串并拒绝其他串。

进一步地，我们可以定义算法A多项式时间可接受的语言和多项式时间可判定的语言。根据上一节的知识，我们应该这样定义P类问题（这样避免了不接受时的无限循环）：

P = {L | L是{0,1}*的子集且存在算法A，使得L可被算法A多项式时间判定}

实际上，由于P类问题的特殊性，把“判定”改成“接受”也行，即：

定理 P = {L | L是{0,1}*的子集且存在算法A，使得L可被算法A多项式时间接受}

证明不不困难。既然在接受时运行时间有多项式上限cn^k，可以把算法稍微修改一下：如果在cn^k时间后还没有被接受，那么拒绝此串。

验证算法和NP类问题 下面考虑一种特殊的算法，它用来验证一个串是否在某语言中。一个验证算法有两个参数x和y，其中x是问题实例而y是**证据(certificate)** 。对于问题实例x，如果存在证据y使得A(x, y) = 1，称实例x可被算法A验证。可被算法A验证的所有串的集合称为可被A验证的语言。根据上一节的知识，我们应该这样定义NP类问题：

NP = {L | L是{0,1}*的子集且存在算法A，使得L可被算法A多项式时间验证}

前面说过，我们并不知道P是否等于NP，我们甚至不知道NP关于补运算是否封闭。即一个问题属于NP并不代表它的补也属于NP。但有一点可以肯定，P是NP的子集，也是co-NP的子集。

小结 刚才的叙述过于形式化，这里做一个小小的回顾。我们首先把问题看作是从实例到解的映射，然后把讨论范围局限在判定问题上。为了方便讨论，我们把抽象问题编码成01串，并限定必须采用标准编码方式或者与它多项式相关的编码方式。最

后，借助于形式语言的工具，我们把判定问题进一步简化定义成一个语言。任何一个判定问题对应于{0,1}上的一个语言，而任何一个{0,1}上的语言都对应一个判定问题。这个语言里面的每一个串都对应一个问题实例，该判定问题对这些问题实例回答“是”，而所有其他实例回答“否”。回忆前面的语言L={10, 11, 101, 111, 1011, 1101, 10001, ...}，它对应判定问题“k是素数吗？”

接下来的讨论涉及到算法，并通过算法和语言的关系重新定义了P类问题、NP类问题和co-NP类问题。虽然通过形式语言严格的定义了“问题”，可是仍然没有定义算法，因此理论基础仍然不够。在下一节中，我们介绍图灵机，并通过另外一种方式严格定义什么是P类问题，什么是NP类问题，并证明新的定义和前面介绍的等价。

小测验

1. 抽象问题是什么？为什么需要将它做编码？为什么编码方式是重要的？应如何编码？
2. 为什么只讨论判定问题？如何定义判定问题所对应的语言？
3. 算法接受/拒绝一个串是什么意思？算法可以多项式时间接受/判定语言L意味着什么？算法可以多项式时间验证语言L又意味着什么？

2.3.2 图灵机

和本章第一节中的决策树类似，图灵机也是一个计算模型。这个计算模型相当强大，它的计算能力等同于任何一台PC机（它对应于随机存取模型RAM）。

图灵机(turning machine) 直观地说，图灵机由记忆单元和控制单元组成。控制器有若干个状态，而记忆单元通常由一条或数条带(tape)组成。每条带被分成无限个小方格。每个格子可以记忆一个符号。控制器和带之间用探头联系起来（每个带对应一个探头），在每一个时刻，一个探头只扫描一个方格。

首先考虑单带图灵机。它的运作由一系列移动组成。每个移动含有四个动作：

步骤一 阅读探头所扫描的方格中的符号
步骤二 擦掉这个符号，写上新的符号
步骤三 探头向左或向右走一格
步骤四 改变控制器的状态

需要注意的是，擦掉的符号和新写的符号可能相同，而控制器的新状态和旧状态也可能相同，但每次探头不能不动，也不能移动多格。后三个步骤取决于目前控制器的状态和步骤一所读到的符号，它可以用一个转移函数来表达。

严格地说，图灵机是一个七元组M=(Q, q_0, F, Σ, Γ, B, δ)，其中Q为状态集合，$q_0 \in$Q为初始状态，接受状态F（F是Q的子集），输入符号集合Σ，带上可用符号Γ（Σ是Γ的子集）以及空格符号B和转移函数δ。其中转移函数为(Q-F)×Γ的子集到Q×Γ×{R, L}的映射，描述下一个状态、写入的字符和探头移动方向。如果转移函数是单值的，称为**确定图灵机(Deterministic Turning Machine, DTM)**；如果是多值的（任何一个值都是可能的被执行的），称为**非确定图灵机(Nondeterministic Turning Machine, NTM)**。在没有说明的情况下“图灵机”指的是DTM。

转移函数是部分函数，对于有的(q, a)可以没有定义。如果当控制器处于状态q，读写头读到a且δ(q, a)没有定义，则图灵机停止运转。

为了方便讨论，我们假设Q和Γ没有交集，则可以用瞬态xqy来表示图灵机的当前位置，其中xy是带上的内容，读写头处于y的第一个字符，且控制器的当前状态为q。图灵机在输入w上**停机(halt)** 当且仅当存在瞬态序列$\alpha_0, \alpha_1, \alpha_2, ..., \alpha_n$，使得$\alpha_0$=$q_0$w（初始瞬态，状态为$q_0$且读写头在w的第一个字符处），且每个$\alpha_i$转移到$\alpha_{i+1}$，除了最后一个$\alpha_n$，它无法转移（停机）。需要注意的是：在DTM中，每个瞬态要么没有“下一瞬态”，要么有唯一的“下一瞬态”；而NTM的每个瞬态可能对应多个“下一瞬态”，且每种情况都有可能出现。

和上小节类似，如果图灵机M在输入w上能停机并且停在接受状态中（即当前状态q在F中），称M接受输入w。所有为M所接受的输入串集合为一个语言，记为L(M)，即M所接受的语言。如果停机时α_n=qy且q在F中，称y为M(w)的输出，其中M(w)表示输入串为w的图灵机。

正式地，我们定义如下几种集合：

如果Σ上的一个语言A，存在图灵机M使得A=L(M)，则称A为**图灵可接受集**或**递归可枚举集**，简称r.e.集。

如果Σ上的一个语言A和它的补集都是递归可枚举集，称其为**图灵可判定集**或**递归集**。

如果对部分函数f: $\Sigma^* \to \Sigma^*$，存在图灵机M使得将f定义域内的w作为输入时M的输出为f(w)，且将定义域外的w作为输入时无法停机，称f为**可计算函**

数，或者**部分递归函数**。

如果一个部分递归函数的定义域等于Σ*，称f为**全递归函数**，或**递归函数**。

容易证明，A是递归集当且仅当A的特征函数（对于任意元素x，x属于A时值为1，其他情况值为0）是可计算函数。因此又称递归集为可计算集，并且当图灵机M计算A的特征函数时，称M计算A。

多带机和非确定机 刚才考虑的是单带DTM。如果可以使用多条带（因此会有多个探头，每次需要确定每个探头写的符号和移动方向），计算能力是否会增强呢？下面的命题回答了这个问题：

*Church-Turing*命题 如果一个函数在某个合理的计算模型上可计算，那么它在图灵机上也是可计算的。

什么是“合理”的计算模型呢？没有办法给出定义，因此这个命题也就无法被证明。可是历史上出现过的人们认为合理的模型均被证明符合该命题，因此该命题获得了人们的广泛承认。多带（确定型）图灵机至少有三条带，一条输入带，一条输出带，其余为工作带。每条带有一个探头，由一台公共的控制器操作。输入带只能读，输出带只能写，只有工作带的探头能读又能写，才可称为读写头。另外，探头可以原地不动。这样，k带图灵机仍然可以用七元组来描述，但转移函数变成了$(Q\text{-}F)\times\Gamma^{k-1}$的子集到$Q\times\Gamma^{k-1}\times\{R, L,S\}^k$的映射，其中S代表“不动”。前一处k-1是因为输出带不可读，而后一个k-1是因为输入带不可写。虽然k带机、NTM和DTM所能判定的语言完全一样[5]，但是它们的效率是有差别的。在今后的讨论中，只考虑k带机。

小结 在本节中，我们讨论了图灵机。这是一种强大的计算模型（有多强大？参见Church-Turing命题），一个TM可以看作一个程序，或者一个算法。这就解决了前面遗留的问题：算法的精确定义。

小测验

1. 什么是图灵机？它的转移函数是怎样的？多带机和非确定机和单带确定机相比有什么不同之处？
2. 什么是递归可枚举集和递归集？

[5] 事实上，可以用DTM来“模拟”k带机和NTM，虽然用的时间可能增多，但可判定的语言一致。有兴趣的读者可以一试

3. 简述Church-Turning命题？为什么它不可证明，但有很多人相信它是正确的？

2.3.3 计算复杂性类

本节将从图灵机的角度定义时间和空间，并给出P和NP类问题的定义。读者可以把这里的新定义和前面的定义作对比，找到它们的联系。

时间和空间 DTM的时间是从计算开始到终止的移动次数，DTM M对输入w的计算时间记为$\text{time}_M(\text{w})$。如果k-2条工作带上被扫描过的方格数分别为$\text{n}_1, \text{n}_2, ..., \text{n}_{k-2}$，则空间$\text{space}_M(\text{w})=\max\{\text{n}_1, \text{n}_2, ..., \text{n}_{k-2}\}$。以下两个定理说明用带少的DTM代替带多的DTM并不会多花费太多的时间，而空间相同：

定理1 对于任何多带TM M，存在双带TM M_1，其接受的语言和计算的函数和M相同，并且对任何输入w，$\text{time}_{M1}(\text{w})=\text{O}(\text{time}_M(\text{w})\log(\text{time}_M(\text{w})))$。

定理2 对于任何多带TM M，存在单工作带TM M_1，其接受的语言和计算的函数和M相同，并且对于任何输入w，$\text{space}_{M1}(\text{w})=\text{O}(\text{space}_M(\text{w}))$。

DTM的时空复杂性 和前面类似，对所有输入取最大值，我们定义DTM M的时间复杂性$\text{t}_M(\text{n})=\max\{\text{time}_M(\text{w}) \mid |\text{w}|=\text{n}\}$，而空间复杂性$\text{s}_M(\text{n})=\max\{\text{space}_M(\text{w}) \mid |\text{w}|=\text{n}\}$。如果对几乎所有n（即除有限个n外）都有$\text{t}_M(\text{n})¡=\text{t}(\text{n})$，我们说$\text{t}_M(\text{n})¡=\text{t}(\text{n})$，并称M是t(n)-时间DTM。类似可定义s(n)-空间DTM。NTM稍微复杂一些，我们需要考虑各种情况中最好的一种，即$\text{time}_M(\text{w})$和$\text{space}_M(\text{w})$分别取各种接受路中最少的时间和空间。NTM M的时间复杂性和空间复杂性分别需要和n+1和1取较大值，避免平凡情况。

问题的复杂性类 定义DTIME(t)={L|存在t(n)-时间DTM M接受L}，类似可以定义DSPACE(t)，NTIME(t)和NSPACE(t)。把函数t替换成函数族，还可以定义DTIME(C)等于C中所有函数t所对应的DTIME(t)的并。注意，这里定义的是问题的复杂性而不是DTM的复杂性！可以预见（类似于前面的结论），时间和空间上增加或减少一个常数因子不会影响这个复杂性类，这就是：

带压缩定理 对c¿0，

$$DSPACE(s(n)) = DSPACE(c * s(n))NSPACE(s(n)) = NSPACE(c * s(n))"$$

线性加速定理 若n趋于无穷时t(n)/n趋于无穷，则对c¿0，

$$DTIME(t(n)) = DTIME(c * t(n))NTIME(t(n)) = NTIME(c * t(n))$$

特别的，可以定义一些特殊的复杂性类：P = DTIME(poly)，NP = NTIME(poly)，PSPACE= DSPACE(poly), NPSPACE = NSPACE(poly)。对于函数还可以定义FP和FPSPACE，而另一些重要的类包括LOGSPACE = DSPACE(logn)，NLOGSPACE=NSPACE(logn)。

需要注意的是，这些复杂性类有些和模型相关的，如DTIME(n^2)，单带DTM和多带DTM所代表的问题集显然是不同的，但是也有一些类与模型独立，如P，FP和PSPACE。

和这个事实相关的一个命题是：

广义Church-Turing命题 如果一个函数在某个合理的计算模型上使用合理时间复杂性度量是多项式时间可计算的，那么它在图灵机上也是多项式时间可计算的。

相信这个命题的人远没有相信Church-Turing命题的人多。事实上，在量子TM上，整数因子分析问题已经找到了多项式时间解，而在DTM上还没有找到。

通用图灵机 假设我们不仅需要输入数据，还要输入程序（即TM），比如判定问题“输入一个TM M和一个串w，对于输入w，M能停机吗？”就需要输入一个TM。要输入TM，必须把TM加以编码。如果x是DTM M的码，定义r(x) = M。如果x不是任何DTM的码，定义r(x) = M_0，其中M_0是一个不接受任何输入的DTM。这样，r是一个从{0,1}*到所有TM的一个映射。这里不讨论TM的编码方式，而给出一个编码系统r应该（但不是必须）满足的三个条件。

1. 对于所有x，r(x)表示一个DTM。

2. 每个单带DTM M至少有一个x满足表示r(x) = M

3. DTM r(x)的转移函数可以很容易由x解码得到。

如果编码系统r满足前两条，称r为单带DTM的枚举。如果还满足第三条，称该枚举支持通用图灵机。这种说法是很直观的，因为根据x可以很容易解码出r(x)的转移函数，从而方便的“模拟”这个r(x)。我们把可以模拟另一个DTM的DTM称为通用图灵机。关于模拟单带、多带图灵机的通用图灵机，有以下定理：

定理1 存在DTM M_u，对任何输入¡x, y¿，M_u模拟M_x以y为输入的运算过程，从而使得L(M_u) = {¡x, y¿ | y ∈ L(M_x)}。并且对于每个M_x，存在一个常数c使得

$$time_{Mu}(< x, y >) <= c * time_{Mx}(y)$$

定理*2* 存在DTM M_u，对任何输入¡x, y¿，M_u模拟多带TM M_x以y为输入的运算过程，从而使得$L(M_u) = \{¡x, y¿ \mid y \in L(M_x)\}$。并且对于每个$M_x$，存在一个常数c使得

$$time_{Mu}(< x, y >) <= c * time_{Mx}(y) log(time_{Mx}(y))$$

定理*3* 存在DTM M_u，对任何输入¡x, y¿，M_u模拟多带TM M_x以y为输入的运算过程，从而使得$L(M_u) = \{¡x, y¿ \mid y \in L(M_x)\}$。并且对于每个$M_x$，存在一个常数c使得

$$space_{Mu}(< x, y >) <= c * space_{Mx}(y)$$

对于NTM也有类似定理，但时间方面只能保证对每个M_x存在一个多项式p 使得对于所有y，$time_{Mu}(¡x, y¿) ¡= p(time_{Mx}(y))$。

不可计算函数 由枚举DTM可以知道，所有可计算函数组成之族是可数的。下面我们证明，{0,1}*映入{0,1}的函数族是不可数的。假设它是可数的，记为$\{f_0, f_1, f_2, ...\}$。设a_i是{0,1}*中依字典序的第i个串。定义函数f为$f(a_i) = 1 - f_i(a_i)$，则它不是任何f。但它确实是{0,1}*映入{0,1}的函数，与可数性矛盾。比较一下：可计算函数族是可数，而全部函数族是不可数的，因此*一定存在不可计算的函数*。例如停机问题$K=\{x \in \{0,1\}^* \mid M_x$对输入x会停机}就是不可计算的（显然K是r.e.集，因为通用TM存在，所以只需要证明K的补不是r.e.集。用反证法，假设K的补可以被M_x接受，讨论x是否属于K）

小结 本节考虑了时空复杂度。在定义了图灵机后，需要定义时间和空间。我们从DTM对于确定输入w的情况出发，定义了DTM的时空复杂度，然后定义了问题的时空复杂度类并介绍了带压缩定理和线性加速定理。为了把TM作为输入，我们讨论了TM的编码和枚举，并介绍了关于通用TM的几个定理，其中单带、多带DTM和NTM略有差别。使用对角线方法，我们证明了存在不可计算函数，并举了一个例子（停机问题）。最后需要提到的是：还记得最初对NP类的定义吗？它被定义为“多项式时间可验证的”。而本节重新定义NP是“非确定图灵机多项式时间可接受的”。可以证明：这两个定义是一致的。

小测验

1. 区分以下概念：DTM对某输入w的时空复杂度；DTM的时空复杂度；问

题的时空复杂性类。

2. 什么是通用图灵机？为什么需要对图灵机编码？什么是DTM的枚举？为什么枚举的存在说明了不可计算函数的存在？

3. 什么是广义Church-Turing命题？为什么相信它的人远没有相信Churn-Turing命题的人多？

2.4 本章小结

本章所讨论的问题相当理论且复杂，这里只是介绍了最基本的原理，目的在于让读者记住一些重要的结论，并了解本书的理论基础。

本章首先讨论了问题的下界，说明了为什么要研究下界，计算模型和基本方法：对立法和归约法。接下来介绍了P类、NP类、co-NP类和NP完全问题并以电路满足问题证明了SAT、3-CNF-SAT、CLIQUE、VC、HC等问题的NP完全性，这些证明富有技巧而有启发性，但缺乏严密的理论基础。

下一节让前面的讨论严密化了。首先给出了问题的形式化描述，用形式语言为工具把抽象判定问题转化为了{0,1}*上的语言，并给出了P类和NP类的第二套定义。接下来，介绍了图灵机，把它作为计算模型并用它描述算法。图灵机的计算能力可以用Church-Turing命题来说明。有了计算模型，我们重新定义算法的时空复杂度，以及问题所属的复杂度类，给出了P类和NP类的第三套，也是最严密的一套定义。这套定义和最初的定义不同（一个是基于可验证性，一个基于NTM），但二者是等价的。最后，我们讨论了通用图灵机，并用它证明了不可计算函数的存在性，并举了一个例子。这至少可以说明：计算机不是万能的。

另外一些没有涉及到的话题有：多项式时间分层、多项式空间、线路复杂性（可作为并行计算模型）、NC问题、概率图灵机、RP和BPP类（P类问题的概率扩展）、计数复杂性、交互证明系统（多项式分层和PSPACE类的概率扩展）、PCP（概率可验证明系统）、数值计算问题的复杂性、优化问题的可近似性以及平均NP-完全性理论等。

最后，再次强调一点：本章第三节的讨论基于图灵机，在其他计算模型下，有些结论不再成立。

第3章　数据结构原理

本书第一章向读者展示了本学科的轮廓、程序设计竞赛的概况、C++程序语言概述和数据结构的基本概念，是学习的开头；第二章对计算理论作了简单介绍，让读者开阔了眼界。本章从理论走向实际，在代码层次上详细介绍数据结构的设计方法和技巧，为进一步学习打下基础。

3.1　线性表、树和图

第一章中曾经提到过，逻辑结构分线性表、树和图三种，每一种都可以用数组和链结构实现。本节介绍这些结构的常用实现方法，并通过对相关经典问题的分析，加深读者的理解。

3.1.1　线性表

线性结构是最简单的结构，元素之间只有前驱和后继关系。可逻辑上的简单关系并不意味着线性表在物理上也是简单的。

数组

数组(array) 是大多数程序设计语言提供的基本数据类型，它是相同类型的元素序列，在物理上连续。基于数组表示法的算法大都比较直观，现仅举几例，展示它们的基本思想和需要注意的问题。在所有例子中，假设元素个数为n，元素保存在a[1], a[2], ..., a[n]，而元素a[0]保留。

数组的插入 在数组中插入元素涉及到元素的移动。

如图 3.1(a)，在插入元素3之前需要把元素4，5，...，n往后移动一个位置，然后插入元素3。由于我们的计算机一次只能移动一个元素，所以必须考虑移动顺序。不能

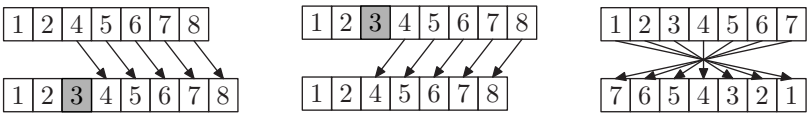


图 3.1: 数组的基本操作。(a) 数组的插入算法。(b) 数组的删除算法。(c) 数组反转算法。

先移后面再移前面，因为移动后面时会把前面的覆盖掉，最后别忘了把n加1，因为这时数组元素已经多了一个。

数组的删除 如图 3.1(b)，删除元素也需要移动，不过这次是往前移动。稍微思考一下就会发现：这次应该改成从前往后移动了。同样地，别忘了元素个数n减少了1。

数组反转 数组的最后一个例子是反转，即把所有元素反过来储存，如图 3.1(c)。现在不管从前往后还是从后往前都会覆盖序列原来的元素。这种情况有两种办法处理，一是事先把原数组备份，二是每次不是直接覆盖，而是交换$1 \leftrightarrow 72 \leftrightarrow 63 \leftrightarrow 5 \ldots$。

注意交换操作主要有两种实现方法。一是用辅助变量，二是使用异或运算符。辅助变量法相信读者已经熟悉，下面简单介绍异或法。异或法的程序非常简单，只需依次执行$a = a \text{ xor } b$, $b = a \text{ xor } b$, $a = a \text{ xor } b$即可。设原来a和b的值分别为x和y，则执行完第一步后$a = x \text{ xor } y$, $b = y$，执行完第二步后$a = x \text{ xor } y$, $b = x$; 执行完第三步后$a = y$, $b = x$。

> **提示：** 数组的元素集中在一起，这给调试带来了很大的方便。一般来说，在设计数组算法时，只需要验证程序产生的结果数组中每个位置的元素是否正确以及元素个数n是否正确。

链表

基于链表的表示法要麻烦一些，也更容易出错，且有较多的变化。一般来说，我们习惯把链接关系next和数据a分离的，不需要修改a只需要修改next。

> **提示：** 链表的关键是链接关系，必须定义清楚，并且维持下去。链接关系的维护是最容易出错的。

如果定义了“第1个元素”位置first，那么当往链表的第一个位置之前再插入一个元素时，一定要修改这个first。通常我们不愿意单独处理这样的特殊情况，而设置一个

虚拟元素，把它作为“第0个元素”，它始终是a[0]。这样，不管遇到什么样的情况，改变的都是某个元素的链接域，不需要单独处理first元素。

单向链表 只需用两个数组即可。int a[]表示元素数据（可根据需要把int换成其他类型），而int next[]表示每个元素的后继元素，next为0的元素是最后一个元素。插入和删除过程只需要修改链接关系。图 **??**(a)和(b)是在元素p后插入和删除元素q的情形。

需要特别注意的是：在删除的时候，结点q仍然是存在的，而且“下一个元素”仍然是r。只不过从链表开头遍历时永远都到不了结点q。在这个意义下，结点q 确实已经从链表中删除了。

> **提示：** 在处理链结构时，我们可以不破坏无用的链接。这样做速度快但容错性不够好。

双向链表 刚才的程序显得不太自然，不能直接指定“删除元素x”，因为没有办法找到它的前一个元素。一个比较自然的方法是：除了“下一元素指针”next外，再设置“上一元素指针”prev。

基于这种表示方法，插入和删除如图 **??**所示。和单向链表类似，删除结点q时，并不需要故意破坏它的前向和后向指针。

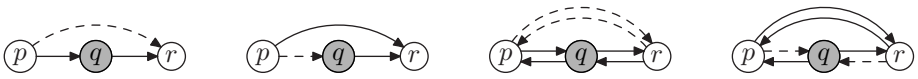


图 3.2: 单向链表和双向链表。(a) 往单向链表的插入元素q。(b) 从单向链表删除元素q。(c) 往双向链表的插入元素q。(d) 从双向链表删除元素q。

和STL一样，我们把函数接口设计为“在结点前”插入元素，则在末尾添加结点相当于在0前插入。在结点i前插入（注意x是结点编号而不是值！）和删除结点x的代码为：

```
void insert_node(int i, int x)
{
    prev[x]=prev[i];
    next[x]=i;
    next[prev[x]]=x;
    prev[i]=x;
}

void delete_node(int x)
{
```

```
    next[prev[x]]=next[x];
    prev[next[x]]=prev[x];
}
```

在插入的时候，需要设置四个链接关系，我们应该先设置新结点的prev/next，然后再设置它前后元素和它的链接关系。如果写成next[prev[x]]=x和prev[next[x]]=x将是非常容易理解的，不过由于速度关系，这里写成prev[i]=x，因为next[x]就是i。删除的时候只需设置x的前一个和后一个相互指向即可。

为了方便，我们再规定next[0]表示第一个元素，prev[0]表示最后一个元素，这样的效果就是：好比由虚拟元素0连接起来的环状链表一样。事实上，这种表示方法确实很容易转化为循环链表：当发现next[x]=0时继续读next[0]，发现prev[x]=0时继续读prev[0]。而对于普通的双向链表，到达元素0意味着遍历结束。二者的区别如图 3.3 所示。

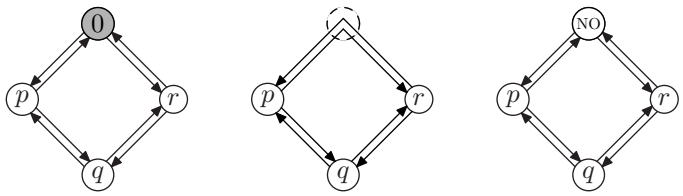


图 3.3: 循环链表和双向链表。(a) 真实链接关系。(b) 虚拟结点做中转是循环链表。(c) 虚拟结点断开是普通链表。

结点管理 在前面的算法中，我们遗漏了两个很重要的问题。在插入操作中，我们需要给出结点编号x，而在实际应用中我们拿到的只有数据本身，没有结点编号，因此需要**分配(allocate)** 结点编号。同样的，删除结点后，应该把这个结点**释放(free)** 到“回收站”中等待分配。由于分配和释放的都只是结点编号，而且分配每个编号都是等价的，因此只需要另外设计一个空闲结点编号队列。如何设计空闲结点编号队列？这就是我们下一小节要考虑的问题。

链表的遍历 链接关系的出现让链表的调试变得困难。在单链表中，只要从头往尾遍历一次没有出现差错，说明链表正确；但在双向链表中，需要确保反向遍历也是正确的，这一点很容易被忽略。这里给出单链表的遍历-输出过程以供调试。双向链表类似，读者可自行写出。

```
p = 0;
while(p = next[p]) output(a[p]);
```

小测验

1. 设计基于数组的算法要特别注意哪些方面？链表呢？
2. 为什么基于链表的算法需要考虑结点编号的管理？为什么数组不需要？
3. 如何方便的调试数组和链表？

栈和队列

前面提到过，栈是后进先出表，而队列是先进先出表，它们都是易于实现的，但应用广泛。

栈的实现 栈只能在一头进行操作，相对比较容易实现。用一个数组int stack[]和栈顶指针top即可，插入和删除（也称push和pop）如图 3.4(a)所示，代码分别为：

```
stack[++top] = x; /* push */
x = stack[top--]; /* pop */
```

和链表删除类似，出栈时并不需要让stack[top]变为0。由于top已经减1了，在逻辑上原来的栈顶元素已经不在栈中，虽然在物理上这个元素本身没有一点变化。

队列的实现 队列的实现需要更多的技巧，原因在于操作是在两端进行的。删除总是从队首端进行，称为**出队(dequeue)**，插入总是在队尾端进行，称为**入队(enqueue)**。

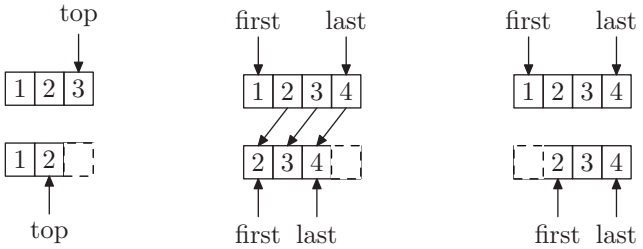


图 3.4: 栈和队列的实现。(a) 栈的实现。(b) 队首始终为数组第一个元素时的队列删除。(c) 队首可变时的队列删除

图 3.4(b)和(c)表示了对于删除操作的两种可能的实现。第一种方法在删除队首时移动元素，始终保持队首是第一个元素（正如栈的数组表示中，栈底始终是第一个元素）。这种操作引起大量元素移动，到队列元素多时非常慢。第二种方法不修改数组本身，而修改队首指针。第一个元素仍然是1，但它不再是队首，队首变成了第二个元素。在这种情况下，入队的时间复杂度为O(1)，比第一种情况好很多。可以借鉴STL的

规矩，把first作为第一个元素而last 作为最后一个元素的后一个位置，则插入和删除的代码如下：

```
x = queue[first++]; /* enqueue */
queue[last++] = x; /* dequeue */
```

虽然第二种方法时间效率高，但是空间利用率却很低。考虑一个空队列不断执行enqueue,dequeue,enqueue,dequeue $\cdots$的操作，first将变得很大（意味着需要一个很大的数组，否则queue[first] 将产生下标越界），但队列里其实什么都没有！这显然是对空间的浪费。

解决的方法是使用循环队列，把数组看成是环状的。插入和删除的代码变为：

```
x = queue[first]; first = (first+1)%n; /* enqueue */
queue[last] = x; last = (last+1)%n; /* dequeue */
```

其中n为数组的长度。只要队列中的元素个数始终不超过n，不管经过多少操作都行。

小测验

1. 为什么栈比队列容易实现？
2. 可以用链表实现队列吗？为什么？
3. 在种子填充中，四个方向考虑的顺序有关系吗？为什么？

3.1.2 树、二叉树和图

前面提到过，树是一种层次结构。二叉树是一种有根有序树，而我们平常所讲的树一般是指**无序树(unordered tree)** 。在后面的学习中我们也将遇到**无根树(unrooted tree)** ，但本节我们的所有讨论都针对**有根树(rooted tree)** 。这些区别请读者注意。

二叉树

二叉树(binary tree)一般采用链结构。和链表类似，这里采用 int a[] , int left[] 和 int right[] 分别表示数据、左儿子编号和右儿子编号。

图 **??**把树描述成递归结构：根root，左子树L和右子树R。在很多情况下，这种简化可以方便的进行算法讨论。一个最典型的例子就是二叉树的**遍历(traversal)** ，即按某种顺序访问树的所有结点。特别地，空树的遍历序列为空。

图 3.5: 二叉树。(a) 二叉树的递归定义。(b) 一棵二叉树。

设树T的前序、中序、后序遍历序列为PreOrder(T)，InOrder(T)和PostOrder(T)，则有：

$$\text{PreOrder(T)} = \{\text{Root(T)}, \text{PreOrder(L)}, \text{PreOrder(R)}\}$$
$$\text{InOrder(T)} = \{\text{InOrder(L)}, \text{Root(T)}, \text{InOrder(R)}\}$$
$$\text{PostOrder(T)} = \{\text{PostOrder(L)}, \text{PostOrder(R)}, \text{Root(T)}\}$$

递归程序非常简单：

```
void PreOrder(int x)
{
    if(x)
    {
        cout << x << "␣";
        PreOrder(left[x]);
        PreOrder(right[x]);
    }
}

void InOrder(int x)
{
    if(x)
    {
        InOrder(left[x]);
        cout << x << "␣";
        InOrder(right[x]);
    }
}

void PostOrder(int x)
{
    if(x)
    {
        PostOrder(left[x]);
        PostOrder(right[x]);
        cout << x << "␣";
    }
}
```

二叉树重建

根据二叉树的前序遍历和中序遍历求后序遍历。

.

图 3.6: 从前序、中序遍历恢复二叉树

根据前序遍历，我们很容易找到根（它就是前序遍历的第一个结点）。在中序遍历中找到根的位置，就可以找到左子树和右子树各有哪些结点，分别用灰色和白色表示。如果在前序遍历中所有灰色结点都在白色结点之前，则递归地把灰色结点和白色结点分别建成左右子树（如箭头所示），然后加上根。如果有的白色结点在灰色结点之前，无解（用叉表示），如图 3.6用这样的方法可以先建树，再进行后序遍历，像这样：

```
bool BuildTree(int pl, int pr, int il, int ir, int& root)
{
    if(pl > pr) /*空树  */
        { root = 0; return true; }

    int p = index[pl]; /*根在原后序遍历中的位置  */
    int q = p1+p-il;  /*最后一个灰色结点在原前序遍历中的位置  */
    for(int i=pl+1; i<=q; i++)
        if(index[i] > p) return false; /*检查灰色是否越界  */

    root = ++nodecount; /*给根分配结点编号  */
    if(!BuildTree(pl+1, q, il, p-1, left[root])) /*递归建立左子树  */
        return false;
    if(!BuildTree(q+1, pr, p+1, ir, right[root])) /*递归建立右子树  */
        return false;
    return true;
}
```

由于在任何时候，递归的参数都是原前序/后序遍历的子序列，因此只需要把前序遍历在原序列中的起止位置(p_l, p_r)和后序遍历的起止位置(i_l, i_r)传递进去即可。由于两个数组都是排列，可以事先预处理算出 pre[i] 在 in 中的下标 index[i] , 方便主程序编写。

和链表类似，设置虚拟结点0作为空子树标记。仍然可以使用队列来实现空闲结点列表，但由于在本例中没有删除操作，只需要简单设置已分配结点个数的计数器即可。

事实上，如果只需要求后序遍历的话，本题不用建树，可以直接递归打印，读者可以试一试。

树

本节讨论的树仅限于无序有根树，即各个儿子之间的顺序是无关的。一般情况下我们使用左儿子-右兄弟表示法存储有根树，即用 int a[] , int son[] 和 int bro[] 表示数据、左儿子编号和右兄弟编号。很多和树有关的题目在预处理时需要把无根树转化成有根树。换句话说，给出一个边列表，需要建立一棵有根树。

一般说来，给出边列表后先要进行预处理，规定每条边(u, v)满足$u < v$，然后按u排序，如图 3.7(b)。更进一步，我们可以设置一个 st[i] 数组，和STL类似， [st[i], st[i+1]) 恰好为以结点i开头的边下标。

图 3.7: 树的表示。(a) 一棵树。(b) 该树的前向星表示。

这样的表示不仅适合于树，也适合于图，称为**前向星(forward star)** 表示法。下面的程序把树的前向星表示转化成左儿子-右兄弟表示，以方便后续算法实现。

```
void star2lsrs()
{
    memset(son, 0, sizeof(son));     /*清零， 为零代表链表为空son */
    for(i = 1; i <= n; i++)
    /*按逆序考虑各个结点，则最后的链表是顺序的  */
        for(j = st[i+1]-1; j >= st[i]; j--)
        {
            bro[j] = son[i];
            son[i] = v[j];   /*插到链表首部  */
        }
}
```

小测验

1. 由二叉树的前序遍历和后序遍历可以确定中序遍历吗？
2. 为什么不用儿子数组的方式表示一棵树，即son[i][j]表示结点i的第j个儿子？
3. 如何遍历一棵树？

图的表示

图分为**无向图(undirected graph)** 和**有向图(directed graph)** 两种，在程序中往往可以统一处理（在网络流的前向星表示中要注意后向弧和前向弧的相互链接以方便

增广）。需要特别注意的是**自环(self-loop)** 和**平行边(parallel edge)** 。自环一般可以直接处理，而平行边往往是给边增加计数器。如果边上还有权，需要尽量把多条边合并成一个（如果可以的话）。

图最常用的表示法是邻接矩阵和邻接表。对于静态图（建图完毕后不再修改图的结构）往往用前向星来代替邻接表，节省空间和时间。

邻接矩阵 不管输入格式如何，总是很容易得到邻接矩阵，只需要注意平行边的情况。

前向星 邻接矩阵本身就包含了顶点序，因此很容易转化为前向星：

```
void matrix2star()
{
    /*上一条的第一端点 初始化为（表示未出现），边数初始化为u0m0 */
    u = m = 0;
    for(i = 1; i <= n; i++)
    for(j = 1; j <= n; j++)
    {
        if(a[i][j])
        {
            v[++m] = j;
            while(u < i)
                st[++u] = m;
        }
    }
}
```

在程序中，u代表上一条边的第一个顶点编号，当$u < i$时代表这条边的第一端点还没有出现过，设置$st[u+1] \dots st[i]$为m。

把边列表转化成前向星的方法类似，只需要把第一顶点相同的结点串成链表，用计数器法进行结点编号分配，和前向星转化成左儿子-右兄弟一样每次插入到链表首部，在O(m)时间内可以建立前向星表示。当然，也可以按第一顶点为关键字直接进行快速排序，不过速度稍微慢一些。

图的连通分量

考虑一个网格，每个格子要么是白色要么是黑色。两个方格如果有一条公共边，则称两个格子连通。所有连通在一起的同色格子称为一个四连通块。类似地，可以定义八连通块，即有公共顶点的连通块。图 3.8有8个黑四连通块，6个黑八连通块。

有一种称为**种子填充(floodfill)** 的方法可以有效的找到连通块。算法的基本思想

图 3.8: 黑白棋盘上的问题。(a) 黑白棋盘。(b) 黑四连块。(c) 以某白点为源的距离标号。

是：从上到下从左到右地检查每一个格子，每找到一个黑格就从它出发找到一个完整的连通块。如何找到一个完整的连通块？设置一个队列，初始时只有那个黑格。每次从队首取出一个黑格，把它相邻的黑格中没有放入过队列 的那些全部放入队列中。

下面的代码中，格子的颜色保存在 a[i][j] 中，其中$0 \le i < n, 0 \le j < m$。0表示白色，1表示黑色。为了节省队列空间，把坐标(x, y)压缩为一个整数 $x \times m + y$，显然不同的坐标对应不同的整数，如图 3.9(a)。

图 3.9: 二维数组的一维化。(a) 编号规则。(b) 邻居编号。

由上可知，对于任何一个编号x, 存在上一行当且仅当$x \ge m$, 存在下一行当且仅当$x < n \times m - m$; 存在左一列当且仅当$x \bmod m \ne 0$，而存在右一列当且仅当$(x + 1) \bmod m \ne 0$。

由于已经讨论过队列的实现，下面的程序中使用 clear_queue() , queue_empty() , enqueue() 和 dequeue() 等函数来表示队列操作，读者很容易写出这几个函数或者调用STL。一般情况下我们经常直接在代码里嵌入队列的实现（或者写成宏调用），而不用另外单独写队列操作函数。以下是求黑四连通块的代码。其中 mark[x] 代表 x 是否已经考虑过。

```
void floodfill()
{
    memset(mark, 0, sizeof(mark));
    for(i=0;i<n;i++)
    for(j=0;j<m;j++)
        if(a[i][j] == 1 && !mark[i *m + j]) /*未标记过的黑点   */
        {
            clear_queue(); /*队列清空   */
            enqueue(i*m+j); /*第一个黑点入队列   */
            while(!queue_empty())
            {
                x = dequeue();
                if(x>=m && !mark[x-m]) /*上*/
                    { mark[x-m]=true; enqueue(x-m); }
                if(x<n*m-m && !mark[x+m]) /*下*/
                    { mark[x+m]=true; enqueue(x+m); }
```

```
                if(x%m!=0 && !mark[x-1]) /*左*/
                    { mark[x-1]=true; enqueue(x-1); }
                if(x%m!=m-1 && !mark[x+1]) /*右*/
                    { mark[x+1]=true; enqueue(x+1); }
            }
        }
}
```

图上的最短路径

利用种子填充可以顺便求出两个格子的最短路径，只需要把mark数组改为路径长度数组即可，初始时起点自身的mark值为0，其他所有点的mark均为-1（表示正无穷），在每次enqueue之前把mark[y]=true 改为mark[y]=mark[x]+1，判断!mark[x]改为mark[x]<0即可。

图 3.8(c)列出了从某格子（用0表示）到所有可达格子的最短路长度。根据这个矩阵很容易得到它到任何一个格子的最短路（只需要让沿途经过的格子数字递增即可）

由于每个格子最多访问1次，检查4次，所以时间复杂度为O(nm)。算法的主要问题在于附加空间也达到了O(nm)，这一点希望读者注意。

最小转弯问题。这里有另外一个例子。图 3.10(a)是一个矩形地图，S代表起点，T代表终点。每一步可以往前走，左转90度或者右转90度。从S走到T最少要转几次弯？注意：长度短的路径可能转弯次数反而多。

图 3.10: 变形的迷宫问题。(a) 最小转弯问题。(b) 带钥匙的迷宫问题。

虽然仍然是矩阵，但是这道题目不能直接用种子填充法求最短路径，因为这里的"最短"不是路径而是转弯次数。如果能构造一个辅助图使得最短路径长度等于最小转弯次数，则问题迎刃而解。

一种方法是扩展相邻关系，让每个格子和它同行同列中能到达的所有格子都直接相邻。如图 3.11(a)所示，每个格子不仅直接和同一列的上一行格子相邻，还和上两行、上三行...直接相邻。这样一来，改造后的最短路径就等于最少转弯次数加1，沿途经过的结点（除了终点）就是那些转弯的点。

这种方法虽然简单，但是时间效率不高。本来图里只有不到4mn条边，现在却达到了O($mn(m+n)$)，高了一个数量级。能不能在保持时间复杂度的情况下进行图的改造呢？

图 3.11: 建图。(a) 最小转弯问题构图。(b) 最小转弯问题的改进构图。(c) 带钥匙的迷宫问题构图。

可以在权上作文章。在种子填充中，图的每条边长度都为1，表示从一个格子走到相邻格子将使路径长度增加1。而在本题中，从一个格子走到相邻格子**不一定**会让转弯次数增加1：和当前方向相同则不变，不同则转弯次数增加1。这提示我们把一个格子拆成四个，代表四个方向，如图 3.11(b)。灰色的四个格子对应于原图中的同一个格子。

其中灰色边的权为0，黑色边的权为1。点数变为了原来的四倍，而每个格子的出度仅为3（没有180度转弯），总边数为$12mn$，比方法一少了一个数量级。可新的问题产生了：这个图的权有的为1有的为0，无法直接进行宽度优先搜索。解决方法是设计两个队列，读者可以试一试。可以证明：结点扩展是按照路径长度从小到大的顺序，和原始的宽度优先搜索一样。

小测验

1. 为什么说种子填充是特殊的图遍历？
2. 如何把前向星表示转化成邻接矩阵？
3. 如果只允许右转，如何求出两点间的最少转弯次数？

3.2 常用数据结构

本节介绍一些常用数据结构，包括二叉堆、并查集、哈希表和排序二叉树。这些数据结构代码简单、使用方便且时间效率高，被广泛的应用到问题求解中。希望读者熟练掌握每一种数据结构的实现，并体会其思想。很多高级数据结构都是从它们出发变化而来的。

3.2.1 二叉堆

二叉堆(binary heap) 是本节介绍的第一个高级数据结构，它的思想简单，代码短，然而用处非常大。二叉堆一般被用来实现**优先队列(priority queue)**，如图3.12所示。

换句话说，优先队列应该提供以下两种操作：

.

图 3.12: 优先队列的功能

Insert(S, x)：把元素x插入到优先队列中。

deleteMin(S, x)：从优先队列中取出最小元素x并从队列中删除。

优先队列经常被用来实现贪心算法（见第四章）和离散时间模拟系统，是一个基础数据结构。STL专门提供了 priority_queue 容器适配器，读者可以在学习本节后比较它和自己的二叉堆实现法的效率。

二叉堆是一棵完全二叉树，即所有叶子在同一层，且集中在左边。二叉堆满足**堆性质(heap property)**：根的值在整棵树中是最小的。不仅如此，根的两棵子树分别构成堆。图 3.13是一个堆。

.

图 3.13: 一棵有14个结点的堆

用 heap 数组来表示各个元素，则根是 heap[1]，最后一个元素是 heap[n]（heap[0] 不使用）。k 的左儿子是 $2k$，右儿子是 $2k+1$，父亲是 $\lfloor k/2 \rfloor$。

删除最小值(deleteMin) 先用最后一个元素代替根，如图 3.14(a)。

.

图 3.14: 堆的删除最小值操作

由于这一步会导致根的元素比儿子大，因此需要向下调整。向下调整的方法很简单，就是把根和较小儿子做比较，如果根比较大，则交换它和儿子，如图 3.14(b)。算法好比石沉大海，因此有的书把它称为sink过程。代码如下：

```
void sink(int p)
{
    int q=p<<1, a = heap[p];
    while(q<=hs)
    {
        if(q<hs && heap[q+1]<heap[q]) q++;
        if(heap[q]>=a) break;

        heap[p]=heap[q];
        p=q;
        q=p<<1;
    }
    heap[p] = a;
```

```
}

int deleteMin()
{
    int r=heap[1];
    heap[1]=heap[hs--];
    sink(1);
    return r;
}
```

插入(insert) 插入是类似的，先插入到最后，然后向上调整。向上调整只需要比较结点和父亲，比向下调整更容易。算法好比从海底游回水面，因此称为swim过程。如图 3.15。

图 3.15: 堆的插入操作

代码如下：

```
void swim(int p)
{
    int q = p>>1, a = heap[p];
    while(q && a<heap[q])
    {
        heap[p]=heap[q];
        p=q;
        q=p>>1;
    }
    heap[p] = a;
}

void insert(int a)
{
    heap[++hs]=a;
    swim(hs);
}
```

删除任意元素(delete) 删除任意元素类似于删除最小元素，先用最后一个元素代替被删除元素，再进行调整。不过值得注意的是，最后一个元素被交换后可能要向下调整，也有可能要向上调整。具体实现可参照以上两段代码。

建立(build) 给n个整数，如何构造一个二叉堆？可以一个一个插入，但是有更好的方法：从下往上一层一层向下调整。由于叶子无需调整，因此只需要从 [hs/2] 开始递减到1。由数学归纳法可以证明：循环变量为i时，以$i+1$, $i+2$,..., n为根的树都是堆。代码如下：

```
void build()
{
    for(int i=hs/2;i>0;i--)
        sink(i);
}
```

时间复杂度 由于堆是完全二叉树，因此高度是O($\log n$)的。插入和删除最多从根到叶子，因此时间复杂度为O($\log n$)。建立操作计算起来稍微麻烦一些。高度为h的结点有$n/2^{h+1}$个，时间为(h)，对$0 \ldots \log n$的h求和，得O(n)。故：建立的时间复杂度为O(n)。

小测验

1. 描述二叉堆的数组表示法，证明树的高度为$\log n$
2. 为什么插入和删除只需要调整一条从根到叶子的路径而不会影响到其他？
3. 如果减小了一个结点的值（称为decreaseKey操作），需要对此结点进行怎样的调整，才能重新得到一个堆？

3.2.2 并查集

并查集(union-find set) 维护一些不相交的集合，它是一个集合的集合。每个元素恰好属于一个集合，好比每条鱼装在一个鱼缸里。每个集合S有一个元素作为“集合代表”rep[S]，好比每个鱼缸选出一条”鱼王”。并查集提供三种操作：

MakeSet(x)：建立一个新集合x。x应该不在现有的任何一个集合中出现。
Find(S, x)：返回x所在集合的代表元素。
Union(x, y)：把x所在的集合和y所在的集合合并。

森林表示法 可以用一棵森林表示并查集，森林里的每棵树表示一个集合，树根就是集合的代表元素。一个集合还可以用很多种树表示，只要树中的结点不变，表示的都是同一个集合。

合并操作 只需要将一棵树的根设为另一棵即可。这一步显然是常数的。一个优化是：把小的树合并到大树中，这样会让深度不太大。这个优化称为启发式合并。

.

图 3.16: 并查集的合并操作

查找操作 只需要不断的找父亲，根就是集合代表。一个优化是把沿途上所有结点的父亲改成根。这一步是顺便的，不增加时间复杂度，却使得今后的操作比较快。这个优化称为路径压缩，如图 **??**所示。

.

图 3.17: 并查集的查找操作

时间复杂度 可以证明（但比较复杂，略），m次操作的总时间复杂度为$\mathrm{O}(m\alpha(n))$，几乎是线性的。其中$\alpha(n)$为让$A_k(1) \geq n$的最小k，它对所有想象得到的n，值均不超过4。$A_k(j)$表示 Ackermann 函数，它是一个增长非常快的函数。

下面给出所有操作的代码。用 p[i] 表示 i 的父亲，而 rank[i] 表示 i 的秩，并用秩来代替深度做刚才提到的启发式合并。

```
void makeset(int x)
{
    rank[x] = 0;
    p[x] = x;
}

int findset(int x)
{
    int px = x, i;
    while(px != p[px]) px = p[px]; // find root
    while(x != px) // path compression
    {
        i = p[x];
        p[x] = px;
        x = i;
    }
    return px;
}

void unionset(int x, int y)
{
    x = findset(x);
    y = findset(y);
    if(rank[x] > rank[y]) p[y] = x;
    else
    {
        p[x] = y;
        if(rank[x] == rank[y]) rank[y]++;
```

```
    }
}
```

小测验

1. 什么形态的树对并查集是好的？即：合并查找操作能较快进行
2. 直观上说明为什么要进行启发式合并和路径压缩
3. 描述基于秩启发式合并规则。为什么秩和深度的作用类似？

3.2.3 哈希表

在很多情况下，我们需要实现一个符号表，里面保存我们用到的所有符号。每个符号有一个关键码key（不同符号的关键码也不同），其余部分可能非常庞大。换句话说，符号表应该提供以下操作：

Search(T, k)：查找关键码k是否在表中
Insert(T, x)：把x添加到表中
Delete(T, x)：从表中删除元素x

有时也把符号表称为"字典"，它最经典的实现方法是哈希表。

直接映射表(direct-access table) 如果关键码的x的取值范围并不大，可以做一个直接映射表，即用一个表 $pos[key]$ 表示关键码等于 key 的符号的位置。这样做是有局限性的。如果 key 的取值范围很大，那么空间无法承受。在很多时候，虽然关键码的取值范围很大，但是符号并不是很多。即使空间可以承受，也是很大的浪费。

哈希表(hash table) 哈希表的核心思想是设计一个哈希函数$h(k)$，把关键码k映射到$0 \ldots m-1$的整数，然后开一个大小为m的直接映射表。如果表里的每个符号都映射到不同的整数，那么直接映射表得以完成。但如果有多个符号映射到了同一个整数（称为冲突），需要用特殊的方法解决**冲突(collision)** 。解决冲突的方法一般有两种：链方法和开放地址法。

链地址法(chaining) 链地址法把映射到同一个槽内的元素串成一个链表，如图3.18(a)所示。

图 3.18: 哈希表的冲突处理。(a) 链地址法。(b) 开放地址法。

设n为符号个数，m为槽的个数，$\alpha = n/m$称为装载因子，则链地址法的期望时间复杂度为$O(1+\alpha)$。

开放地址法(open addressing) 开放地址法的基本思想是："既然我的位置已经被占了，那么我只好去占别人的位置。" 换句话说，第一次查找失败后，算法将尝试下一个位置。如果下一个位置还是失败，再尝试第三个位置...这样的尝试是*系统的*(systematic)，即这个序列只由关键码key决定。

因此，如果用$h(k,i)$表示关键码为k的第i次探测位置，对于每个k应该满足：

$$< h(k,0), h(k,1), h(k,2), \ldots, h(k,m-1) > \text{是} \{0,1,\ldots,m-1\} \text{的排列}$$

这样的要求是合理的，因为任何一个位置都不需要探测多次，而所以位置都应该能被探测到。需要特别注意的是：开放地址法很难删除元素。显然当符号个数超过槽的个数时，插入无法完成，因此使用开放地址法时，应保证装载因子小于1。理论分析指出（不证明）：期望探测次数为$1/(1-\alpha)$，因此当元素装满一半时，期望查找次数为2, 90%装满时，期望查找次数为10。

哈希函数的设计 由于冲突取决于哈希函数，我们希望哈希函数能让不同符号的函数值尽量在$[0, m-1]$内平均分布。另外，由于哈希函数计算频繁，它应该能容易被算出来。通常用三种方法：

- **取余法**。取$h(k) = k \bmod m$，显然得到的是$0 \ldots m-1$之间的数，一般取m为不太接近2或10的幂的素数。
- **乘积法**。取$h(k) = (A \cdot k \bmod 2^w) rsh (w-r)$。这里$w$为字长，因此$A \cdot k \bmod 2^w$实际上就是$A \cdot k$不管溢出而得到的值。这样得到的$m = 2^r$。
- **点积法**。取随机向量a，与k的n进制向量做点积。

对于不同特点的对象分别有各自常用的哈希函数，这里略去。当元素较多时，通常在额外的地方保留所有原始数据，而用链方法处理冲突。查找和插入的代码如下：

```
int find(int k)
{
    int h = hash(k);
    int p = first[h];
    while(p) // lookup in the linked list
    {
```

```
        if(key[p] == k) return p;
        p = next[p];
    }
    return 0;
}

void insert(int x)
{
    int h = hash(key[x]);
    next[x] = first[h]; // insert at the head of the list
    first[h] = x;
}
```

程序中， first[i] 表示哈希函数值为 i 的第一个数据下标， key[i] 表示第 i 个数据的 key ， next[i] 表示第 i 个数据的下一个数据。注意 hash 是长度为 m 的整数而 data 是长度为 n 的符号数组， next 是长度为 n 的整型数组。数据下标从1开始，0是虚拟结点。

小测验

1. 列举出直接映射表能够应用的例子。如果关键码是1 9的排列，可以做成直接映射表吗？
2. 为什么链方法中的链不是双向链表而是单向的？
3. 为什么开放地址法难以处理删除操作？链方法呢？

3.2.4 排序二叉树

上一节中，我们曾经提到过二叉树。二叉树的一个重要变形是排序二叉树，它给一棵二叉树的每个结点赋予一个权值，并且保证每个结点的权值大于等于左子树所有结点的权值，但小于等于右子树所有结点的权值。本节假设所有权值都不相同。对于同样的结点集合，可以构造出多棵排序二叉树，如图 3.19。

图 3.19: 关键码1, 2, 3组成的三棵不同的排序二叉树

虽然每一棵都是排序二叉树，但是它们并不是相互等价的。很快我们就会看到，树的高度越小越好，因此中间那棵树最好。

排序二叉树也可以用来实现符号表，即提供以下操作：

Find(S, x)：查找符号x是否在表中

Insert(S, x)：把x添加到表中

Remove(S, x)：从表中删除元素x

从使用上和哈希表是一样的。读者也许会问：为什么要同时学习两种数据结构？答案是：它们有各自不同的变形。排序二叉树的变形很多，大部分变形都是哈希表所无法实现的。

查找 考虑Find操作的实现。根据定义，只需要根据情况往左或者往右走即可。这里仍然是用0表示空树。查找最小值和最大值是简单的，这里分别用递归和非递归来写：

```
// recursive
int find(int x, int p)
{
    if(!p) return 0;
    else if(x == a[p]) return p;
    else if(x < a[p]) return find(x, left[p]);
    else return find(x, right[p]);
}

// recursive
int findmin(int p)
{
    if(!p) return 0;
    else if (!left[p]) return p;
    else return findmin(left[p]);
}

// non-recursive
int findmax(int p)
{
    if(p) while(right[p]) p = right[p]; // walk along right-son
    return p;
}
```

可以看出，待查找结点的深度决定了查找的效率。考虑最坏情况，树的高度应越小越好。这就回答了刚才提出的问题：三棵不同的树，为什么中间那棵好。

插入 现在考虑Insert操作。显然Insert操作发生在“想找却找不到”的时候。事实上，当查找失败时，到达的空树所在的位置就是新结点的位置！如图 3.20所示。

.

图 3.20: 排序二叉树的查找和插入

因此，可以稍微修改find过程而得到insert过程：

```
// pass p by reference to make it change
void insert(int x, int& p)
```

```
{
  if(!p) p = newnode(x);
  else if(x < a[p]) insert(x, left[p]);
  else if(x > a[p]) insert(x, right[p]);
}
```

删除 最复杂的是remove操作，需要分两种情况处理。当p最多只有一个儿子时，只需要修改p父亲的指针即可，如图 3.21(a)所示。

图 3.21: 排序二叉树的删除。(a) 单子女的情形。(b) 双子女的情形。

和链表一样，结点4删除以后需要放到“回收站”里，虽然它的左儿子仍然是3，但是已经不在树结构中了。

如果p有两个儿子，需要找到比它大的最小结点q，用q替换p，再删除p，删除2的过程如图 3.21(b)所示。由于q是比p大的最小结点，这样操作以后仍然是一棵合法的排序二叉树。

综合两种情况，我们得到完整的remove代码：

```
void remove(int x, int& p)
{
    if(!p) return;
    if(x < a[p]) remove(x, left[p]);
    else if (x > a[p]) remove(x, right[p]);
    else if(left[p] && right[p]) // two sons
    {
        int q = findmin(right[p]);
        a[p] = a[q]; // copy element
        remove(a[p], right[p]);
    }else{
        int q = p; // backup p, since ‘‘deletenode’’ destroys it
        p = left[p] ? left[p] : right[p]; // no matter 1 or 0 sons
        deletenode(q);
    }
}
```

这个代码效率并不是很高，因为它实际上做了两遍，一遍是findmin，另一遍再递归删除这个最小值。如果记录好需要替换的位置然后不回溯的直接往下走，只需要一遍即可完成。这个改进留给读者去实现。

需要说明的是：如果删除操作并不是很频繁，可以用**懒删除(lazy deletion)** 方法，即置一个删除标记但不修改结构。

平衡二叉树 不仅是查找操作，插入和删除的时间复杂度也依赖于树的高度。我们希望树在高度上尽量的平衡，在这种情况下时间复杂度接近O($logn$)。如果所有结点形成一条链，那么查找叶子的时间复杂度将变为O(n)！

有两种方法可以让树平衡。第一种情况需要预先知道关键码的分布。例如，关键码取值恰好是$1 \ldots n$，那么可以预先把这n个数建成完全二叉树，给每个结点设置计数器。只要找到元素对应的结点（一定可以找到！），插入相当于计数器加1，删除时减1，查找就是判断计数器是否大于0。即使元素取值范围很大，只要预先知道所有可能取值，而且这些取值并不是很多，都可以采取离散化的方法，同样通过这样的静态平衡二叉树实现。

如果关键码可以任意，情况就复杂许多了。有很多种动态的平衡二叉树可以使每种操作的最坏情况或者期望、平摊的时间复杂度为O($\log n$)。至少可以避免排序二叉树“在空树中依次插入$1, 2, 3, \ldots, n$的总时间高达O(n^2)”这样的窘境。我们将在下一节中介绍它们。

小测验

1. 对于排序二叉树来说，为什么树的深度越小越好？深度的最小值和最大值各是多少？平均深度呢？
2. 为什么说插入操作和查找是几乎一样的？
3. 叙述删除操作。可以改用比它小的最大结点代替它吗？为什么？

3.3 平衡二叉树及其变种

基本的排序二叉树有可能是不平衡的。如果事先不知道可能出现的关键码，就必须建立动态的平衡二叉树。动态的平衡二叉树有很多种，大都相当复杂。这里介绍4种，即AVL树、伸展树、跳跃表和Treap。

3.3.1 AVL树

AVL树的设计思想是：限制树的形状，让每次操作后的结果严格满足要求。一个直观的想法是：左右子树一样高。可惜这样的树不一定是平衡的，它可以是两条链，高度为$n/2$，如图 3.22(a)。如果要求严格一点，规定每个结点的左右子树一样高，那么12个结点的树没有一棵是满足要求的。如图 3.22(b)，这是12个结点的树中最平衡

的，可是结点1、3、6都不满足“左右子树高度相同”的条件。

图 3.22: 不合理的平衡二叉树定义。(a) 太宽松：根结点左右子树等高。(b) 太严格：所有结点左右子树等高。

自然地，我们规定每个结点的左右子树高度差不超过1（空树的高度为-1，单结点树的高度为0），这样的树称为AVL树。

AVL树的高度。一个很自然的问题是：这样放宽的定义是否保证了高度是平衡的？答案是肯定的。设$S(n)$为高度为n的AVL树的最少结点数，则$S(n) = S(n-1) + S(n-2) + 1$，而$S(0) = 1, S(1) = 2$，经过数学计算可得：$n$个结点的AVL树的最大深度为约$1.44 \log_2 n$，是平衡的。

旋转操作 给定一棵AVL，如果按照一般的排序二叉树进行插入操作，结果可能变成一棵非AVL树。从根到插入点路径上的那些结点的左右子树高度差（称为平衡因子）可能引起改变（其他结点的平衡因子一定不变）！如果改变后的值不是1，0或者-1，那么违反了AVL树的平衡性质，需要进行调整。

假设从底向上第一个违反AVL性质的结点为p，那么插入的情况有四种：

情况*1*：在p的左儿子的左子树上插入了一个结点
情况*2*：在p的左儿子的右子树上插入了一个结点
情况*3*：在p的右儿子的左子树上插入了一个结点
情况*4*：在p的右儿子的右子树上插入了一个结点

情况1和情况4是对称的，可以用一次单旋转进行修正；情况2和情况3也是对称的，可以用一次双旋转进行修正。

单旋转(single rotation) 如果在p的左儿子q的左子树插入了一个结点，可以按图3.23的方法解决。注意三角形的高度也代表着树的高度。只需要修改left[p]=right[q]和right[q]=p即可。

注意：这里的旋转是以儿子为轴的（有些书把这样的旋转称为rotatewithLeftChild和rotatewithRightChild）。一般意义下的单旋转需要读父亲指针，在介绍伸展树操作时会遇到。

这里只给出左单旋代码，右单旋类似，修改right[p]和left[q]即可。读者可以验证旋转前后X，Y，Z和q、p的相对大小关系是否一致。

图 3.23: AVL树的单旋转：以左儿子为轴旋转

```
// rotate, axis is p's left child
void rotatewithLeftChild(int& p)
{
    int q = left[p];
    left[p] = right[q];
    right[q] = p;
    height[p] = max( height[left[p]], height[right[p]] ) + 1;
    height[q] = max( height[left[q]], height[p] ) + 1;
}
```

双旋转(double rotation) 在图 3.23中，如果不是X过高而是Y过高，一次单旋转仍然无法解决问题（旋转后p的左子树比q的左子树深2个单位）。这时候需要进一步画出Y的内部结构，如图 **??**。经过一次所谓的左右双旋（情况2）后得到平衡。情况3是对称的，只需要进行右左双旋。

图 3.24: AVL树的双旋转：以q的右儿子为轴旋转，再以p的左儿子为轴旋转

双旋可以用两次单旋转实现，如左右双旋就是先以left[p]的右儿子，再以p的左儿子为轴旋转，即rotatewithRightChild (left[p]); rotatewithLeftChild(p);

AVL树插入的程序实现 虽然可以在每个结点中保存平衡因子，但是为了清晰起见，这里并没有这样做，而是用height[p]来保存结点p的高度，特别地，height[0]=-1。

```
void insert(int x, int& p)
{
    if(!p) p = newnode(x);
    else if(x < a[p])
    {
        insert(x, left[p]);
        // when not balanced, the difference must be 2
        if(height[left[p-]]  height[right[p]] == 2)
        {
            if (x < a[left[p]]) // left rotate
                rotatewithLeftChild(p);
            else // left-right rotate
            {
                rotatewithRightChild(left[p]);
                rotatewithLeftChild(p);
            }
        }
    }
```

```
    else if(x > a[p])
    {
        insert(x, right[p]);
        if(height[right[p－]]  height[left[p]] == 2)
        {
            if(x > a[right[p]]) // right rotate
                rotatewithRightChild (p);
            else // right-left rotate
            {
                rotatewithLeftChild(right[p]);
                rotatewithRightChild(p);
            }
        }
    }
    else
        ;  // duplicated note, nothing to do

    height[p] = max(height[left[p]], height[right[p]]) + 1;
}
```

删除操作比插入操作更复杂，这里略。

3.3.2 伸展树

和AVL树不一样，伸展树并不保证每次操作的时间复杂度为O($\log n$)，而保证任何一个m个操作的序列总时间为O($m \log n$)。

伸展树的基本思想是：每个结点被访问时，使用AVL树的旋转操作把它移动到根。由于旋转是自底向上的，所以需要设置父亲指针，而不像AVL树那样以儿子为轴旋转。

伸展操作(splaying) 伸展树的核心是伸展操作Splay(x,S)。它是在保持伸展树有序性的前提下，通过一系列旋转将伸展树S中的元素x调整到树的根部。在调整的过程中，要根据x的位置分以下三种情况分别处理。

情况一: 节点x的父节点y是根节点。

这时，如果x是y的左孩子，我们进行一次Zig（右旋）操作；如果x是y的右孩子，则我们进行一次Zag（左旋）操作。经过旋转，x成为二叉查找树S的根节点，调整结束。如图 3.25所示。

图 3.25: 伸展树旋转：zig旋转和zag旋转

两种旋转不仅代表了伸展操作的情况一，而且也是后两种情况的基础，因此把代码列在这里。由于旋转过程中需要修改父亲，因此需要记录父亲指针。这里仍然让0充当虚拟结点，因此可以随意修改它的父亲和儿子。注意建立关系时必须同时修改儿子的父亲指针和父亲的儿子指针，在下面的代码中这样的成对操作被写在同一行中。一共有三组成对操作。

情况二：节点x的父节点y不是根节点，y的父节点为z，且x与y同时是各自父节点的左孩子或者同时是各自父节点的右孩子。

这时，我们进行一次Zig-Zig操作或者Zag-Zag操作。如图 3.26所示

图 3.26: 伸展树旋转：zig-zig旋转

情况三：节点x的父节点y不是根节点，y的父节点为z，x与y中一个是其父节点的左孩子而另一个是其父节点的右孩子。

这时，我们进行一次Zig-Zag操作或者Zag-Zig操作。如图 3.27所示

图 3.27: 伸展树旋转：zig-zag旋转

综合三种情况的伸展操作是：

```
void splay(int& x, int& s)
{
  int p;
  while(father[x])
  {
    p = father[x];
    if(!father[p])  // Zig & Zag
    {
        if(x == left[p]) RightRotate(x);
        else LeftRotate(x);
        break;
    }
    if(x == left[p])
    {
        if(p == left[father[p]])
            { RightRotate(p); RightRotate(x);} //Zig-Zig
        else
```

```
            { RightRotate(x); LeftRotate(x); } //Zig-Zag
    }
    else
    {
        if(p == right[father[p]]
            { LeftRotate(p); LeftRotate(x); } //Zag-Zag
        else
            { LeftRotate(x); RightRotate(x); } //Zag-Zig
    }
  }
  s = x;
}
```

下面是一个例子。如图 3.28(a)所示，执行Splay(1,S)，我们将元素1调整到了伸展树S的根部。再执行Splay(2,S)，如图 3.28(b)所示，我们从直观上可以看出在经过调整后，伸展树比原来“平衡”了许多。伸展操作的过程并不复杂，只需要根据情况进行旋转就可以了，三种旋转都是由基本的左旋和右旋组成的，实现较为简单。

.

图 3.28: 伸展操作举例

五种基本操作 利用Splay操作，我们可以在伸展树S上进行五种基本运算。Find和Insert只是简单的进行通常的操作后伸展操作元素x，需要重点说明的是Delete、Join和Split。

Delete(x,S)：将元素x从伸展树S所表示的有序集中删除。用伸展树查找找到x的位置，则x到了根的位置。合并x的左右子树即可。

Join(S1,S2)：将S1与S2合并，其中S1的所有元素都小于S2的所有元素。首先，我们找到伸展树S1中最大的一个元素x，再通过Splay(x,S1)将x调整到伸展树S1的根。然后再将S2作为x节点的右子树。这样，就得到了新的伸展树S，如图 3.29所示。

.

图 3.29: 伸展树的join操作

Split(x,S)：将S分离为S1和S2，其中S1中元素都小于x，S2中元素都大于x。先查找，将元素x调整到根，则x的左子树就是S1，而右子树为S2。如图 3.30所示。

.

图 3.30: 伸展树的split操作

其他操作 除了上面介绍的五种基本操作，伸展树还支持求最大值、求最小值、求

前趋、求后继等多种操作，这些基本操作也都是建立在伸展操作的基础上的。各种操作的代码如下：

```
int find(int x, int s)
{ int p = BST_Search(x, s); splay(p, s); return p; }

void insert(int x, int& s)
{ int p = BST_Insert(x, s); splay(p, s); return p; }

void remove(int x, int& s)
{ int p = find(x,s); join(left[p], right[p]); }

int maximum(int s)
{ int p = s; while(right[p]) p = right[p]; splay(p,s); return p; }

int minimum(int s)
{ int p = s; while(left[p]) p = left[p]; splay(p,s); return p; }

int prev(int x, int& s)
{ int p = find(x, s); p = left[p]; return maximum(p); }

int next(int x, int& s)
{ int p = find(x, s); p = right[p]; return minimum(p); }

int join(int& s1, int& s2)
{
  if(!s1) return s2; if(!s2) return s1;
  int p = maximum(s1); right[p] = s2;
  return p;
}

void split(int x, int&s, int& s1, int& s2)
{
  int p = find(x, s);
  s1 = left[p]; s2 = right[p];
}
```

伸展树最有意思的地方在于伸展操作结束以后，伸展结点一定在根处，因此才有了join和split这样有意思的操作，delete也变得比普通BST更加简单。

结论*(不证)*：n个结点的伸展树m次操作的总时间开销为O($mlogn$)。

伸展树有一种优化形式，不需要父亲指针，只需要O(1)的附加存储，所有操作自顶向下，维持L和R两个临时树。有兴趣的读者可以参考相关书籍。

小测验

1. 直观来说，伸展操作的效果如何？

2. 叙述伸展操作的三种情况。它们的条件和操作是怎样的？

3. 如何用伸展操作实现查找、插入、删除、最大最小值、前驱、后继、合并和分离操作？它们的平摊时间复杂度如何？

3.3.3 跳跃表

跳跃表是对链表的直接改进。链表的查找操作最坏情况下是O(n)的，但是如果加一些捷径的话，情况会有所好转——增加第二层、第三层、第四层...链表。我们就得到了本节要介绍的跳跃表：

.

图 3.31: 跳跃表上的查找操作

具体来说，每个结点维护下一个元素next[p]和下一层元素down[p]。对于原表的每一个元素，我们不断的抛硬币，如果是正面，则增加一层；如果是反面，则停止。查找操作是简单的，好比一个斜着的排序二叉树一样，当且仅当到达最后一层还往下时失败。

```
int find(int x, int s)
{
    v = s; // starting point
    while(v && key[v] != x)
        v = key[right[v]] > x ? down[v] : right[v];
    return v;
}
```

图 3.31是搜索5的情形。

插入操作 由于插入位置固定，可以在查找失败的同时确定插入位置。可问题是：插入高度是多少？我们把抛硬币改一下，初始列高为1，每次得到0 1的随机数，如果小于p则高度加1，否则停止。为了插入一列，需要保留每层的“最后一个访问元素”，如上图的黄色元素，以便进行链表插入操作。可以证明：p取$1/e$时最好。

删除操作 首先执行查找操作，如果没找到则退出。否则和刚才类似，根据每层的最后访问元素来对每层链表进行删除操作，删除多余的空链。

可以证明：三个操作的期望时间复杂度都是O($logN$)。

小测验

1. 为什么说跳跃表很像是斜着的排序二叉树？试比较二者的查找操作

2. 插入的过程是怎样的？删除时应该注意什么问题？
3. 写出插入和删除操作的详细代码

3.3.4 Treap

Treap是排序二叉树和堆的结合，对键值来说是排序二叉树，对优先级来说是堆。

图 3.32: 一棵Treap和对应的平面剖分

也可以这样来看待treap：它是按照优先级递增的顺序一个一个结点插入而成的排序二叉树。或者把treap看成是平面的划分，如图 3.32。

我们用treap来实现平衡二叉树。显然，查找操作和普通的排序二叉树是一样的。很容易证明：当每个结点和键值和优先级固定时，treap的形态是唯一的（而不像排序二叉树那样有很多种！）。虽然treap也可以很不平衡，但是它不是因为插入顺序不好，而是因为键值和优先级分布不好。由于实际上只有键值是有意义的，所以我们可以随机选择优先级，像快速排序一样，把坏情况从数据转移到了随机数发生器。

基本操作 插入结点时，首先按照排序二叉树的标准插入法插入，但这时可能违反了堆性质。因此自下而上进行旋转操作。直到堆性质得到满足。删除是相反的，先把优先级设置为最低，自上而下转移到叶子，然后删除。以x为基准的分离操作和伸展树类似：只需要插入一个键值为x，优先级最高的结点，则插入以后此结点将变成根，则它的两棵子树为所求。合并操作是相反的：先建立一个虚拟结点成为两棵树的根，调用删除操作即可。

图 3.33: Treap上的插入和删除。从左到右为插入，从右往左为删除

代码如下（newnode和deletenode的功能是分配结点编号和把结点放到空闲队列中）：

```
void insert(int x, int& p)
{
  if(!p) p = newnode(x, myrand()); // random priority
  else if(x < key[p])
  {
      insert(x, left[p]);
      if(priority[left[p]] < priority[p]) rotateWithLeftChild(p);
```

```
  }
  else if(x > key[p])
  {
      insert(x, right[p]);
      if(priority[right[p]] < priority[p]) rotateWithRightChild(p);
  } // duplicated node, do nothing
}

void remove(int x, int& p)
{
  if(p)
  {
    if(x < key[p]) remove(x, left[p]);
    else if(x > key[p]) remove(x, right[p]);
    else{
      // delete node directly
      if(!left[p] && !right[p]) deletenode(p);

      // rotate, delete recursively
      if(priority[left[p]] < priority[right[p]])
        { rotateWithLeftChild(p); remove(right[p]); }
      else
        { rotateWithRightChild(p); remove(left[p]); }
    }
  }
}
```

需要注意的是我们假设空树的优先级为无穷大，则上面的代码可以正确处理只有一个儿子的情形。

结论 所有操作的期望时间复杂度均为O($logn$)。从编程复杂度来说，Treap是最容易实现的数据结构。不仅没有任何特例需要考虑，而且同时在期望O($logn$)时间内支持插入、删除、分离和合并。

小测验

1. 什么是treap？为了把它用做平衡二叉树，如何处理优先级？
2. 插入和删除的过程是怎样的？
3. 写出合并和分离的操作的代码

3.4 可并优先队列

二叉堆很好的实现了优先队列的各种操作，但是却很难将两个堆合并起来–只能将其中一个堆的元素一个一个取出来插入另一个堆。如果两个堆的元素都有n个，

需要花$nlogn$的时间。本节介绍四种可并优先队列，其中左偏树和斜堆实现简单，实用性高。二项堆是一种非常巧妙的数据结构，实现难度适中，也是Fibonacci堆的基础。Fibonacci堆的理论时间复杂度非常优秀，特别是基于层叠提升法的decreaseKey操作，不仅思想巧妙，而且平摊时间复杂度仅为O(1)。

3.4.1 左偏树和斜堆

左偏树(leftist heap) 可以用来实现可并优先队列，支持Insert, DeleteMin和Merge操作。

左偏树是一棵二叉树，每个结点定义了距离(dist)，即它到后代中“没有两个儿子（即左儿子为空或右儿子为空）”的结点的最小距离。叶子的距离为0，而空结点的距离为-1。

左偏树满足两个性质：

堆性质：根的键值小于儿子键值。

左偏性质：根的左儿子的距离大于或等于右儿子的距离。这个性质是递归的，即左偏树根的左右子树分别是左偏树。

左偏树的距离定义为根的距离，根据左偏性质，它是树中“最右路径” 的长度。左偏树并不是高度越小越好，而是越“左偏”越好，这样距离才会比较小。图 3.34是一棵左偏树。

.

图 3.34: 左偏树

由左偏性质，结点距离等于右儿子距离加1。由于左儿子的距离至少和右儿子一样大，因此n个结点的左偏树距离至多为$[log_2(n+1)]-1$。

合并操作 合并操作是左偏树的核心操作。在合并操作中，最简单的情况是其中一棵树为空（也就是，该树根节点指针为NULL）。这时我们只需返回另一棵树。

第一步 如图 3.35(a)。若A和B都非空，我们假设A的根节点小于等于B的根节点（否则交换A,B），把A的根节点作为新树C的根节点，剩下的事就是合并A的右子树right(A)和B了。

.

图 3.35: 左偏树的合并操作

第二步 如图 3.35(b)。合并了right(A) 和B之后，right(A) 的距离可能会变大，当right(A) 的距离大于left(A) 的距离时，左偏性质会被破坏。在这种情况下，交换left(A) 和right(A)。

第三步 由于right(A) 的距离可能发生改变，因此A的距离也可能随之而改变，所以我们应更新A的距离：dist[A]=dist[right[A]]+1。

```
int merge(int a, int b)
{
  if(!a) return b;
  else if(!b) return a;
  if(key[b]<key[a]) swap(a, b);

  right[a] = merge(right[a], b);
  if(dist[right[a]] > dist[left[a]]) swap(left[a], right[a]);
  if(!right[a]) dist[a] = 0;
  else dist[a] = dist[right[a]] + 1;

  return a;
}
```

合并的时间复杂度 下面我们来分析合并操作的时间复杂度。从上面的过程可以看出，每一次递归合并的开始，都需要分解其中一棵树，总是把分解出的右子树参加下一步的合并。根据性质3，一棵树的距离决定于其右子树的距离，而右子树的距离在每次分解中递减，因此每棵树A或B被分解的次数分别不会超过它们各自的距离。根据距离和结点数的关系，分解的次数不会超过$[log_2(N_1+1)]+[log_2(N_2+1)]-2$，其中$N_1$和$N_2$分别为左偏树A和B的节点个数。因此合并操作最坏情况下的时间复杂度为$O(logN_1+logN_2)$。

插入和删除 插入操作相当于合并一棵单结点树。由于最小值在根上，因此删除最小值相当于合并根的两棵子树，都是很容易实现的，这里不再赘述。

建树操作 需要一提的是建树操作。和二叉堆一样，有比逐点插入更快的算法：

步骤一：将n个节点（每个节点作为一棵左偏树）放入先进先出队列。
步骤二：不断地从队首取出两棵左偏树，将它们合并之后加入队尾。
步骤三：当队列中只剩下一棵左偏树时，算法结束。

由于前$n/2$次合并的是单结点子树，后$n/4$次合并的是双结点子树...接下来$n/2^i$次合并的是有2^{i-1}个结点的子树。由于合并两个2^i 个结点的子树时间为$O(i)$，因此总时间

为

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

删除已知结点 如果不把左偏树看作单纯的可并优先队列，它还需要支持删除操作（好比取消原来将要发生的事件）。首先注意到：删除结点不能像二叉堆那样先交换到根再删除，因为左链可能很长！因此只能直接合并两棵子树为p。如果删除的是根，那么任务完成，否则需要进一步讨论。新树p的距离有可能比原来x的距离要大或小，这势必有可能影响原来x的父节点q的距离，因为q现在成为新树p的父节点了。于是就要仿照合并操作里面的做法，对q的左右子树作出调整，并更新q的距离。这一过程引起了连锁反应，我们要顺着q的父节点链一直往上进行调整。

.

图 3.36: 左偏树的删除操作

情况一 dist(p)+1=dist(q) 。那么不管p是q的左子树还是右子树，我们都不需要对q进行任何调整，此时删除操作也就完成了。

情况二 dist(p)+1<dist(q) 。那么q的距离必须调整为dist(p)+1，而且如果p是左子树的话，说明q的左子树距离比右子树小，必须交换子树。由于q的距离减少了，所以q的父节点也要做出同样的处理。

情况三 dist(p)+1>dist(q) 。在这种情况下，如果p是左子树，那么q的距离不会改变，此时删除操作也可以结束了。如果p是右子树，这时有两种可能：一种是p的距离仍小于等于q的左子树距离，这时我们直接调整q的距离就行了；另一种是p的距离大于q的左子树距离，这时我们需要交换q的左右子树并调整q的距离，交换完了以后q的右子树是原来的左子树，它的距离加1只能等于或大于q的原有距离，如果等于成立，删除操作可以结束了，否则q的距离将增大，我们还要对q的父节点做出相同的处理。

删除任意已知节点操作的代码如下：

```
void remove(int x)
{
  q = father[x];
  p = merge(left[x], right[x]);
  father[p] = q;
  if(q && left[q] == x) left[q] = p;
  if(q && right[q] == x) right[q] = p;
  while (q)
  {
```

```
    if(dist[left[q]] < dist[right[q]]) swap(left[q], right[q]);
    if(dist[right[q]]+1 == dist[q]) break;
    dist[q] = dist[right[q]] + 1;
    p = q;
    q = father[q];
  }
}
```

左偏树的一个变种称为**斜堆(skew heap)**，它也是基于合并的，但不记录距离信息，因此不保证左偏性质。合并仍然是在右子树上进行，但是通过在每个结点处交换子树而避免右链过长。

```
int merge(int a, int b)
{
  if(!a) return b; else if(!b) return a;
  if(key[b] < key[a]) swap(a,b);
  right[a] = merge(right[a], b);
  if(right[a]) // no right subtree, not need to swap
      swap(left[a], right[a]);
  return a;
}
```

其他操作仿照左偏树。有一点需要特别注意：由于没有左偏性质，右链可能在某些时候很长。直接的递归实现可能引起栈溢出。

小测验

1. 左偏树和斜堆最基本的操作是什么？叙述这个操作的详细过程。如何用它实现插入、删除最小值和建堆？
2. 最好的左偏树是什么样子的，最坏的呢？左偏树的左链可能很长，删除任意结点后是否会导致向上调整很多次最后到达根？
3. 为什么说AVL和伸展树的关系类似于左偏树和斜堆的关系？

3.4.2 二项堆

二项堆(binomial queue) 不是一棵树，而是若干树的集合，每棵树叫做二项树。二项树的形态非常有意思。高度h一定时，二项树的形态是固定的，且恰好有2^h个结点。在二项堆中，任意两棵二项树的高度都不相同。由于任意自然数的二进制表示是唯一的，因此包含任意结点数的二项堆在形态上是唯一的。为了讨论方便，我们把所有二项树按高度从小到大排列。

.

图 3.37: 二项树

合并操作 二项堆的合并好比两个数的二进制加法一样，对应数位上的数相加，逢二进一。由于树一共只有$logN$棵，合并两棵二项树的时间是常数的，因此总时间复杂度为O($logN$)。和前面一样插入操作只是合并的特殊情形。就像二进制计数器问题一样，虽然单独一次插入可能需要O($logN$)的时间，但连续多次插入的平摊时间却是常数的。

> 注意：合并操作是一个概念上简单，而实现起来比较麻烦。如果看成二进制加法的话，需要考虑$0+0$, $1+0$, $0+1$, $1+1$以及有进位c的情况一共8种。一个好的二项堆实现应该能够简洁的统一处理这些情形，并且能快速合并两个相同高度的树。

删除最小值操作 首先需要查找最小值x以后，首先在原二项堆H里删除x所在的树B，得到二项堆H'，然后把B的根去掉，得到二项堆H''，然后合并H'和H'' 即可。

由于在得到H''时需要列举x的所有儿子，可以使用左儿子-右兄弟表示法实现每棵树，而用数组（不一定连续）储存所有树，即 tree[i] 表示高度为i的二项树。由于二项树需要按照高度排序，在左儿子-右兄弟表示法中需要另外规定左边的儿子高度大。

其他操作 如果最小值即时更新的话，查找最小值只需要O(1)的时间，但删除仍然需要O($logN$)。二项堆还可以支持decreaseKey和已知结点的remove操作，这里不再赘述。

小测验

1. 什么是二项树？它的形态是怎样的？它的结点数和高度有什么关系？
2. 什么是二项堆？为什么结点数一定时，二项堆的形态是固定的？
3. 叙述二项堆的合并和删除最小值操作。如何使最小值查找是O(1)的？

3.4.3 Fibonacci堆

Fibonacci堆(Fibonacci heap) 是对二项堆的修改。Fibonacci堆的decreaseKey, insert,merge和findmin的平摊时间复杂度仅为O(1)，而只有deleteMin的平摊时间复杂度为O($logN$)。常数时间复杂度是怎么来的呢？merge和insert的O(1) 来源于懒合并，

而decreaseKey的O(1)来源于类似于伸展树和左偏树的“切割子树”操作。显然，类似二叉堆的向上调整法肯定不能让decreaseKey达到O(1)。

Fibonacci堆由二项堆演变而来，但同一个高度的树可能不止一棵。我们用循环链表来保存所有树，并维护最小值指针。各棵树本身需要在左儿子-右兄弟表示法的基础上增加父亲和左兄弟指针，因此每个结点有四个指针：father, child, prev和next。为了讨论方便，我们把所有树组成的链表称为主链表。

. .

图 3.38: Fiboancci树。(a) 精简表示。(b) 实际结构。

插入、合并和取最小值 由于维护了最小值指针，最小值在O(1)时间内找到。插入合并操作像插入或合并普通循环链表一样（别忘了检查是否需要更新最小值）。因次三种操作都可以在O(1)时间内完成。

删除最小值 首先把最小值删除，然后把它的子树添加合并到主链表中。由于子树本来就是一个循环链表，因此除了更新父亲指针之外，这一步是O(1)的。

下一步是重新扫描主链表，找到新的最小值并顺便更新父亲指针。这一步最坏是O(n)的。为了让平摊时间复杂度不这么高，这里使用一个顺便的清理操作，合并高度相同的树。

清理操作 清理操作是不得不进行的合并，它扫描链表，对于每个结点，执行一系列合并操作，以确保没有另外一棵树和它高度相同，如图 3.39

.

图 3.39: Fibonacci堆的清理操作

清理操作以及连续合并子程序如下($B[i]$保存任一棵高度为i的树)：

```
void mergedupes(int& v) // v may be changed
{
    int w = B[deg[v]];
    while(w)
    {
        B[deg[v]] = 0;
        if(key[v] <= key[w]) std::swap(v, w);
        list_remove(w); // (**)
        link(w, v); // link w to v
        w = B[deg[v]];
    }
    B[deg[v]] = v;
```

```
}

void cleanup()
{
    // newmin can be replaced by any node in root list
    int newmin = head(root_list);
    for(int i = 0; i <= n >> 2; i++)
        B[i] = 0; // null

    // for all nodes v in the root list
    int v = head(root_list);
    do{
        parent[v] = 0;  // (*)
        if(key[newmin] > key[v]) newmin = v;
        mergedupes(v);
        v = next[v];
    }while(v != root_head);
}
```

由于(*)行对每个结点恰好执行一次，而(**)对每个结点最多执行一次，因此总时间复杂度为$O(r + deg(min))$, r为原来主链表中的结点数，$deg(min)$为原最小值的子树数目。不过执行完DeleteMin之后，主链表的元素个数变成了$O(\log n)$，因为它已经是一个二项堆了。

DecreaseKey操作 这个操作并不是像二叉堆那样不断往上交换，而是把它和父亲的边切掉，直接提升到主链表里。另外还有一个提升规则：只要一个结点x 恰好有两个儿子被切掉，则提升x。为了满足这个规则，我们把恰好有一个儿子被切掉的结点做一个标记mark。只有第二个儿子也被切掉时，这个结点的mark才会被清除，而它同时被提升到了主链表中。如图 3.40，这样的提升是层叠式的。

.

图 3.40: Fibonacci堆的DecreaseKey操作

代码框架如下：

```
void DecreaseKey(int v, int k)
{
    key[v] = k;
    if(k < key[min]) // update pointer to min
        min = k;
    promote(v);
}

void promote(int v)
{
```

```
    mark[v] = 0;
    if(parent[v])
    {
        delete_child(parent[v], v);
        link_insert(root_list, v);
        if(mark[parent[v]]) promote(parent[v]);
        else mark[parent[v]] = 1;
    }
}
```

Fibonacci树的结构 由于decreaseKey进行了切割结点操作，每棵树已经不一定是二项树了。那么Fibonacci树到底是什么样子呢？设以v为根的树结点总数为$|v|$，我们有：

定理：Fibonacci树中的任意结点v满足$|v| >= F_{deg(v)} + 2$。

作为定理的最坏情况，我们在二项堆里删除尽量多的结点而不引起任何结点的提升。由于不能删除两个儿子，因此最坏情况是删除结点数最多的儿子，如图 3.41所示。这也从直观上“证明”了刚才的定理。

图 3.41: 一道六阶Fibonacci树的结构，虚线结点因被提升而离开树

或者更直观的，考虑层叠式提升。f_7由f_5和f_6组成，好比fibonacci数一样，因此把树f_n称为fibonacci树，fibonacci堆也因此得名。

图 3.42: 递推关系。(a) 二项树的递推关系。(b) Fibonacci树的递推关系。

结论*(不证)*：Fibonacci树的insert, merge操作最坏情况时间复杂度为O(1)，deletemin和decreaseKey的平摊时间复杂度为O($\log n$)和O(1)。

小测验

1. 为什么Fibonacci堆的主链表是循环链表而不是单链表？每个结点的儿子链表呢？

2. 描述清理操作的详细过程，并证明删除操作的时间复杂度为O($r + deg(min)$)。为了高效的支持哪一个操作，原本优美的二项树结构被破坏了？

3. 什么是fibonacci树？为什么它是fibonacci堆的最坏情况？它是由二项树经过怎样的操作得来的？它的高度和根结点的度数是怎样的？

3.5 本章小结

本节的内容比较多，学习难度比较大。3.1和3.2的所有内容都是非常重要的，需要知道每个细节并能熟练写出代码。3.3和3.4节有不少内容启发性很强，但是在实际应用中并不一定经常使用。

在各种平衡二叉树和变种中，Treap写起来也许是最容易的，且支持包括分离和合并在内的所有常见操作，AVL提供了一个通用的方法维持树的平衡，虽然思想简单但实现并不容易，所以本书并没有给出全部代码。伸展树是自调整数据接结构很好的例子，只要熟练掌握了伸展操作，所有其他操作都是容易写的。跳跃表的思想也很简单，算法描述也很容易，只是程序实现上有人觉得很简单，有人觉得不习惯。需要说明的是：在所有平衡树中，旋转操作都是最基础的，AVL和Treap只以儿子为轴旋转，相对比较简单，而伸展树需要以自身为轴旋转，因此必须记录父亲指针。

本节介绍的四种可并优先队列中，左偏树和斜堆是最容易写的，它们的关系类似于AVL和伸展树。二者的所有操作基于合并操作（好比伸展树所有操作基于伸展操作），因此学习的关键在于合并操作。虽然理论复杂度不错，二项堆和Fibonacci堆并不实用，因此读者平时可以只使用左偏树，而二项堆和Fibonacci堆只需要学习原理，获得思维上的启发（这也是本书没有给出它们的完整代码的原因）。

二项堆的思想非常简单，读者可以拿数的二进制表示和它比较。二项堆的合并操作完全类似于二进制加法，删除最小值操作只是简单把结点所在的树单独提取出来，删除根后和原来的二项堆合并。Fibonacci堆最合并的思想就是“懒”。由于不及时合并，Fibonacci树的结构变得复杂，需要用双向链表保存，且在删除最小值后必须执行清理操作，合并高度相同的树。DecreaseKey操作是Fibonacci堆另一个特别之处，切割结点法使得DecreaseKey的平摊时间复杂度是O(1)，但正是由于切割结点的产生，Fibonacci堆没有了二项堆优美的结构，它的最坏情况–Fibonacci树是Fibonacci堆名称的由来。

第4章　算法设计方法

本章是第二部分的开头，通过几类经典算法设计思想向读者展示算法设计的一般方法。本章是第二部分后续章节的基础，也是第三部分算法的出发点。

4.1　递归与分治

递归是一种思考问题的方法，而不仅仅是一类具体的算法。事实上，在数据结构的学习中，我们已经看到了很多递归的例子。排序二叉树、堆等的定义都是递归的。本节介绍递归思想，以及它的一个典型应用：分治算法。本章强调时间复杂度的分析，读者需要理性的知道为什么基于分治的算法比普通算法好，好多少。

4.1.1　递归思想

递归的思路是简洁的，有时候可以把看似困难的问题巧妙的解决掉。这里举三个例子。

棋盘覆盖问题 有一个$2^k \times 2^k$的方格组成的棋盘, 恰有一个方格是黑色的, 其他为白色。需要用三个方格的L型牌覆盖所有白色方格。黑色方格不能被覆盖, 任两个方格不能有重叠部分。试找一个方案。图 8.1为四种L型牌。

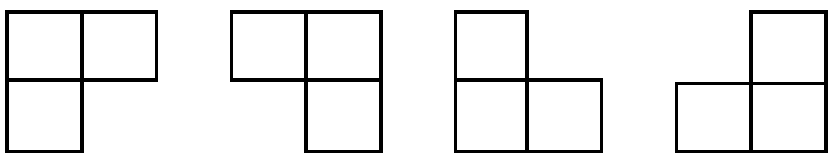


图 4.1: L型牌

本题的棋盘是一个$2^k \times 2^k$的，很容易想到分治。把棋盘切为四块，则每一块都是一个$2^{k-1} \times 2^{k-1}$的。有黑格子的那一块可以递归解决，但其他三块并没有黑格子，应该怎么办呢？可以构造出一个黑格子，如图8.1。当$k = 1$时一块牌就够了。

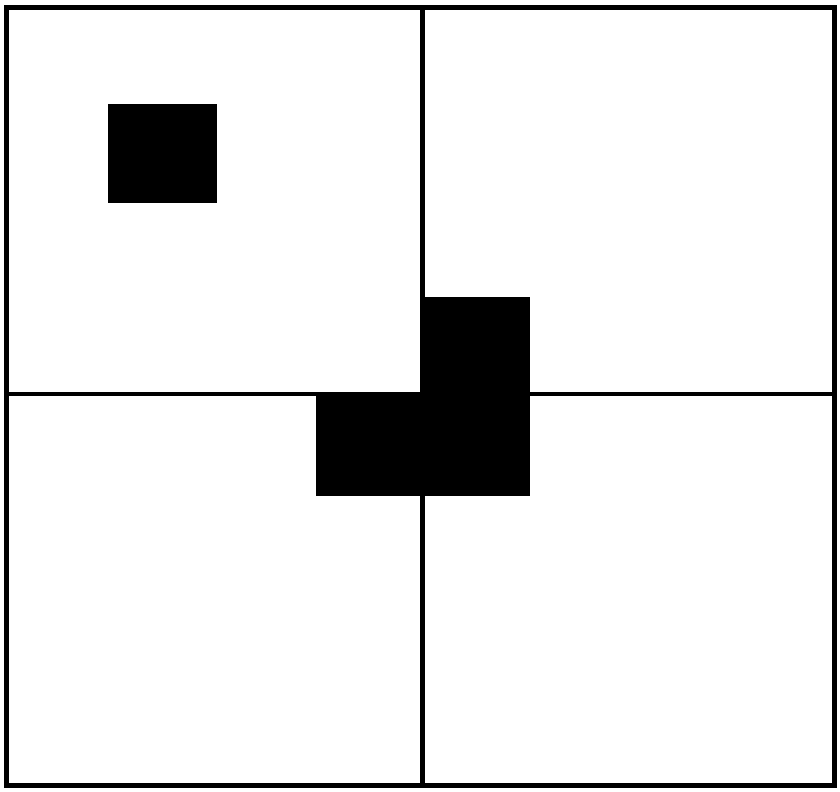


图 4.2: 棋盘覆盖问题的递归法

循环日程表问题 有$n = 2^k$个运动员要进行网球循环赛, 需要设计比赛日程表。每个选手必须与其他$n-1$个选手各赛一次；每个选手一天只能赛一次；循环赛一共进行$n-1$天。按此要求设计一张比赛日程表，它有n行和$n-1$列, 第i行j列为第i个选手第j天遇到的选手。

本题的方法有很多，递归是其中一种比较容易理解的方法。下图是$k = 3$的一个可行解，它是四块拼起来的。左上角是$k = 2$的一组解，左下角是左上角每个数加4得到，而右上、右下角分别是左下、左上角复制得到。

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

图 4.3: 循环日程表问题k=3的解

巨人与鬼 在平面上有n个巨人和n个鬼，没有三者在同一条直线上。每个巨人需要选择一个鬼，向其发送质子流消灭它。每个巨人只能选择一个鬼，不同的巨人必须选择不同的鬼。质子流由巨人发射，沿直线行进，遇到鬼后消失。由于质子流交叉是很危险的，所有质子流经过的线段不能有交点。请设计一种给巨人和鬼配对的方法。

由于只需要任意一种配对方法，从直观上来说本题一定是有解的。由于每一个巨人需要找一个目标，我们不妨先给“最特殊”的巨人找。

考虑y坐标最小的点（即最低点）。如果有多个这样的点，考虑最左边的点（即其中最左边的点），则所有点的极角在范围$[0, \pi)$内。不妨设它是一个巨人，然后把所有其他点按照级角从小到大排序。

如果第一个点是鬼，那么配对完成，剩下的巨人和鬼仍然是一样多，而且肯定不会和这一条线段交叉，如图 8.1(a)所示。

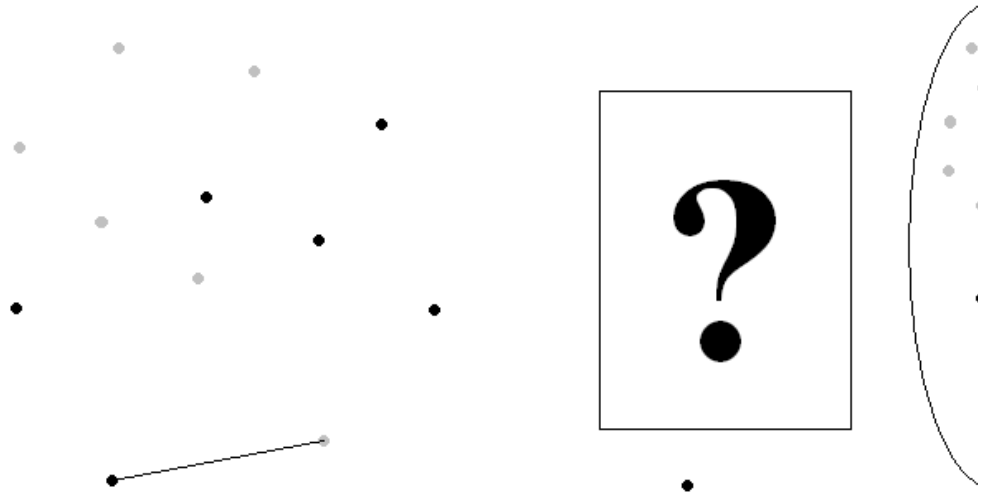


图 4.4: 巨人与鬼问题：极角排序后的直接配对和递归求解

如果第一个点是巨人，那么继续检查，直到鬼和巨人一样多为止，如图 8.1(b)所示。这样的情况一定会出现，因为第一点出现时鬼少，而最后一个点检查完是鬼多，而巨人和鬼的数量差每次只能改变1。从少到多的过程中一定会有“一样多”的时候。找到了这个配对区间后，只需要把此区间内（如图 8.1(b)的椭圆区域）的点配对，再把区域外的点配对即可。这个配对过程是递归的，好比棋盘覆盖中一样。

小测验

1. 描述本小节介绍的三个问题和它们的解法。
2. 棋盘覆盖问题中，如果边长不是2的整数幂，原来的算法还能使用吗？日程表问题呢？巨人和鬼问题呢？
3. 日程表问题也可以按照和递归相反的顺序进行，即按照$k = 0, 1, 2 \ldots$ 的顺序构造。棋盘覆盖问题可以这样构造吗？巨人和鬼问题呢？

4.1.2 三个经典的分治算法

本节讨论三个经典问题并做出相应的分析。这些问题有很多方法，但是最容易想到的方法往往不够好。在这样的情况下，**分治法(divide-and-conquer)** 往往是一个不错的选择。分治法使用递归的思考方式，遵守以下方法：

划分(divide) 把问题划分成若干**子问题(subproblem)**
求解(conquer) 递归求解子问题。

合并(combine) 把子问题的解合并成原问题的解。

其中第二个步骤需要递归调用，而划分子问题和合并子问题的解需要仔细设计算法。下面给出三个例子，它们的划分、求解、合并方法各不相同。

最大最小值 给n个实数，求它们之中的最大值和最小值，要求比较次数尽量小。

这个问题并不难解决，相信大多数读者可以写出下面的程序：

```
void minmax(int a[], int n, int& min, int& max)
{
    if(a[0] > a[1])
        { max = a[0]; min = a[1]; }
    else
        { max = a[1]; min = a[0]; }

    for(i = 2; i < n; i++){
        if(a[i] > max) max = a[i];
        else if(a[i] < min) min = a[i];
    }
}
```

即：首先比较前两个数，确定其中的较大值max和较小值min，以后每个元素分别和max和min比较，并更新。最坏情况下以后每个数都需要比较两次，共$1+2(n-2)=2n-3$次比较。

用递归的方法可以做得更好。假设需要$T(n)$的时间求n个数的最大值和最小值，显然$T(2)=1$。我们按照分治三步法进行算法设计。

划分 把n个数均分为两半。

递归求解 求左半的最小值min_L 和最大值max_L以及右半最小值min_R和最大值max_R。

合并 所有数的最大值为$max\{max_L, max_R\}$，最小值为$min\{min_L, min_R\}$。

程序如下。划分点为$m=(l+r)/2$，两个区间为$[l,m]$和$[m+1,r]$（注意不是$[1,m-1]$和$[m,r]$）：

```
void minmax(int l, int r, int& min, int& max)
{
    int minl, minr, maxl, maxr;
    if(r == l){ max = min = a[l]; }
    else if(r == l+1)
    {
```

```
        if(a[l] > a[r]
            { max = a[l]; min = a[r]; }
        else
            { max = a[r]; min = a[l]; }
    }
    else
    {
        int m = (l + r) >> 1;
        minmax(l, m, minl, maxl);
        minmax(m+1, r, minr, maxr);
        max = maxl > maxr ? maxl : maxr;
        min = minl < minr ? minl : minr;
    }
}
```

递归程序比顺序扫描程序长一些，那么比较次数如何呢？我们有递归方程$T(n) = 2T(n/2) + 2$，表示总时间等于两边分别递归计算的时间$2T(n/2)$加上合并时间2。分析分治算法大都需要求解这样的递归方程。我们把系统的求解方法放在下一小节中介绍，而本节只采取归纳法和**替代(substitution method)法** 。

从n比较小的情况开始讨论。边界是$T(2) = 1$。

$$T(4) = 2T(2) + 2$$
$$T(8) = 2T(4) + 2 = 4T(2) + 6$$
$$T(16) = 2T(8) + 2 = 8T(2) + 14$$
$$T(32) = 2T(16) + 2 = 16T(2) + 30$$

可以归纳出（也容易用数学归纳法证明）：$T(n) = n/2 * T(2) + n - 2 = 1.5n - 2 < 2n - 3$，比刚才的方法好。读者可能会说刚才的推理不严密，因为n不一定是2的幂，$T(2n+1)$应该等于$T(2n) + T(2n+1) + 2$，但事实上这样的分析和实际的结果非常接近。在未加说明的情况下，本书在算法分析中不区分$n/2$取上整和下整的情况，一律写作$n/2$。

有序表查找问题 给出从小到大排列的n个不同数$a[0] \sim a[n-1]$，试判断元素x是否出现在表中。

最简单的方法是一个个寻找，时间复杂度为$O(n)$。这个方法并没有用到“n个数从小到大排列”这一个关键条件，因而时间效率低下。

仍然考虑分治算法，遵循三步法。假设需要在$a[l] \sim a[r]$中查找元素x。

划分 检查某个元素$a[m]$（$l \le m \le r$），如果$a[m] = x$则查找成功，返回m。

递归求解 如果$a[m] > x$，那么元素只可能在$a[l] \sim a[m-1]$中；如果$a[m] < x$，那么元素只可能在$a[m+1] \sim a[r]$中。

合并 不需要合并。

这个算法称为**二分查找(binary search)** 。需要解释一下递归求解过程。假设需要寻找元素10，发现$a[5] = 7$，如图 8.1。由于所有元素从小到大排列，显然10如果出现则只能在灰色区域中，因此递归在$a[6] \sim a[9]$中寻找。

图 4.5: 二分查找示意图

记$T(n)$为在n个数中找出元素x的时间，那么$T(n) = max\{T(k), T(n-k-1)\} + 1$。其中$k$表示左边的元素个数，则$n-k-1$为右边元素个数。这里取$max$是因为我们考虑的是最坏情况时间复杂度，因为最好情况是直接碰巧发现$a[m] = x$，只比较了一次。这样的好情况是平凡的，在本书中一般不考虑。显然k太小时$T(n-k-1)$太大，而k太大时$T(k)$太大，最好让$T(k)$和$T(n-k-1)$一样大，即取$k = n/2$（这里再次忽略了取整方向的问题）。

这样，程序如下：

```
int bsearch(int l, int r, int x)
{
  if(l > r) return -1;
  int m = (l + r) >> 1;
  if(a[m] == x) return m;
  else if (a[m] > x) return bsearch(l, m-1, x);
  else return bsearch(m+1, r, x);
}
```

递归程序一定要注意边界的处理。

只有一个元素时，则$m = l = r$。如果$a[m] = x$，则程序成功退出；如果$a[m] > x$，则递归查找$[l, l-1]$，它会立刻返回-1；如果$a[m] < x$，则递归查找$[l+1, r]$，它也会立刻返回-1。如果这两个边界情况没有单独处理，则二分查找程序将无限递归下去。

只有两个元素时，$m = l = r - 1$。划分成的两半为$[l, l-1]$和$[r-1, r]$，分别转化为了空序列和一个元素的情况，可以正确处理。

元素个数更多时，转化成元素个数少的情况。由数学归纳法容易证明整个算法的正确性。这个算法往往被写成非递归的形式，即：

```
int bsearch(int l, int r, int x)
{
  int m;
  while(l <= r)
  {
    m = (l + r) >> 1;
    if(a[m] == x) return m;
    else if(a[m] > x) r = m - 1;
    else l = m + 1;
  }
}
```

把这种简单的递归转化成迭代的形式，通常是一个好主意。

下面进行时间复杂度分析。由于$m = n/2$，递归方程化成了$T(n) = T(n/2) + 1$。用和刚才类似的方法有：

$$T(1) = 1$$
$$T(2) = T(1) + 1 = 2$$
$$T(4) = T(2) + 1 = 3$$
$$T(8) = T(4) + 1 = 4$$

归纳得：$T(2^n) = n + 1$，即$T(n) = log_2 n + 1$。渐进分析给出：$T(n) = O(\log n)$。

快速幂 给出非负整数a和n，求出a^n的值（不考虑精度问题）。

和查找问题一样，最简单的方法是一个循环，时间复杂度为$\mathrm{O}(n)$。

```
int power(int a, int n)
{
  int x = 1;
  for(i = 1; i <= n; i++) x *= a;
  return x;
}
```

这个问题同样可以用分治法来解决。

划分 如果n是偶数，考虑$k = n/2$，否则考虑$k = (n-1)/2$

递归求解 计算a^k。

合并 如果n是偶数，则a^n=$(a^k)^2$，否则a^n=$(a^k)^2 \times a$

根据这个方法很容易写出程序：

```
int power(int a, int n)
{
    if(n == 0) return 1;
    else if(n & 1)
    {
        int x = power(a, (n-1)>>1);
        return x*x*a;
    }
    else
    {
        int x = power(a, n>>1);
        return x*x;
    }
}
```

可能有读者会这样写：

```
int power(int a, int n)
{
    if(n == 0) return 1;
    else if(n & 1)
        { return power(a, (n-1)>>1)*power(a, (n-1)>>1) * a; }
    else
        { return power(a, n>>1)*power(a, n>>1); }
}
```

和刚才那个程序的差别是：这里直接进行了两次递归调用，而不像第一个程序那样先保存到一个变量x中。这个程序不仅看起来罗嗦，而且时间效率低，因为它重复的进行了两次递归调用，递归方程为$T(n) = 2T(n/2) + 1$，而第一个程序的递归方程为$T(n) = T(n/2) + 1$。

由于方程和二分查找完全一样，程序一的时间复杂度也是$\mathrm{O}(\log n)$的，而程序二的时间复杂度和求最大最小值是一样的，为$\mathrm{O}(n)$（想一想为什么？）。这个例子告诉我们：应该尽量减少递归调用，尤其不要进行重复的计算。这就是下一节“动态规划”的基本思想，我们稍后还会详细分析。

本小节介绍的三个问题都不困难，但基于分治的算法比最普通的算法时间效率高。这些算法的时间复杂度分析都需要解递归方程，这将是下小节中讨论的主题。

小测验

1. 叙述分治三步法。哪一步涉及到递归调用？

2. 直观说明为什么求最大最小值的分治算法和普通算法相比节约了比较次

数？

3. 二分查找中，如果递归的区间是$[l, m]$和$[m, r]$，会有怎样的结果？

4.1.3 递归树和主定理

上一小节遇到的三个递归方程

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

都是以下递归方程的特例：

$T(n) = aT(n/b) + f(n)T(1) = 1$

其中$f(n)$是关于n的函数。注意：方程一虽然不满足$T(1) = 1$，但可以做替换$T'(n) = T(2n)/2$，则$T'(n) = T(2n)/2 = T(n)+1 = 2T'(n/2)+1T'(1) = T(2)/2 = 1$，满足此方程。

本节讨论这个方程的解法。

主定理(Master Theorem) 记$p = log_b a$，则$T(n)$的解分三种情况

情况1：$f(n) = O(n^{p-\varepsilon})$，则$T(n) = \Theta(n^p)$
情况2：$f(n) = O(n^p \log^k n)$，则$T(n) = \Theta(n^p \log^{k+1} n)$
情况3：$f(n) = O(n^{p+\varepsilon})$，则$T(n) = \Theta(f(n))$

证明略，但是可以从8.1.1的递归树中体会到主定理的思想：

高度$h = \log_b n$，第i层的代价和为$a^i f(n/b^i)$，叶子的个数为$a^h = n^p$。考虑级数

$$c(n) = f(n) + af(n/b) + a^2 f(n/b^2) + \ldots + np \tag{4.1}$$

三种情况下，此级数的各项接近几何级数增长，当公比不为1时和与最大项同阶。

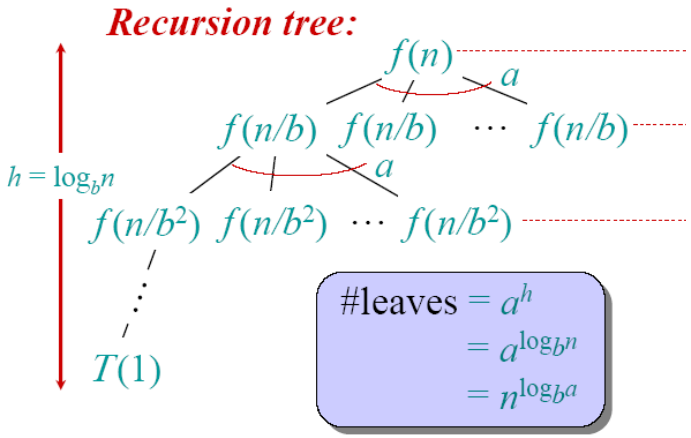


图 4.6: 递归树

情况一：$f(n)$小多项式级别，最大项是n^p

情况二：$f(n)$大对数级别，每项看作相等，因此需要乘以高度$\log n$

情况三：$f(n)$大多项式级别，最大项是$f(n)$

这并不是一个严格证明，但是可以直观说明递归树方法的思想。

嵌入问题 把一棵有n个结点的完全二叉树嵌入到网格中，使得占用面积尽量小。

本题的方法有很多，这里只举两种，作为主定理的应用。

基本方案 如8.1.1，两棵子树左右并排，然后高度加一层。

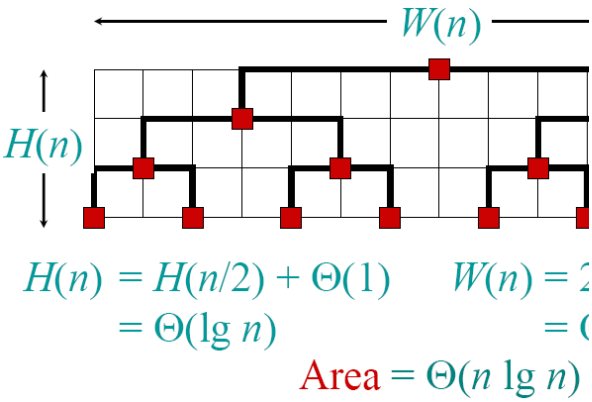


图 4.7: 基本方案

为了计算面积，我们分别计算高度$H(n)$和宽度$W(n)$。

高度的递归方程是$H(n) = H(n/2)+\Theta(1)$，$p = \log_b a = \log_2 1 = 0$。$f(n) = 1$和$n^p = n^0 = 1$同阶，符合情况二，解为$H(n) = \Theta(\log n)$。

宽度的递归方程是为$W(n) = 2W(n/2) + \Theta(1)$，$p = \log_b a = \log_2 2 = 1$，$f(n) = 1$比$n^p = n$小多项式级别，取大者$n^p = n$，解为$\Theta(n)$。

面积为$H(n) \times W(n) = \Theta(n \log n)$

H型方案 如下图，两棵子树交替进行左右并排和上下并排，形成H型。

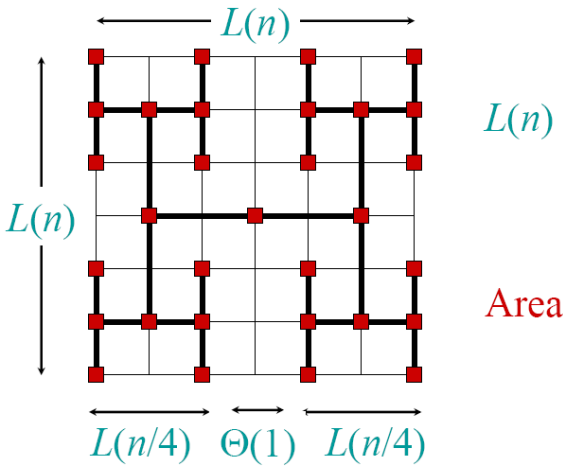


图 4.8: H型方案

这是一个正方形，边长$L(n)$的递归方程为$2L(n/4) + \Theta(1)$。$P = \log_b a = \log_4 2 = 2^{1/2}$。由于$f(n) = 1$比$n^p = n^{1/2}$小多项式级别，取大者$n^p = n^{1/2}$，因此$L(n) = \Theta(n^{1/2})$，因此面积为$\Theta(n)$。

最后，我们介绍主定理的一般形式。

Akra-Bazzi定理 对于递归方程$T(n) = a_1T(n/b_1) + a_2T(n/b_2) + ... + a_kT(n/b_k) + f(n)$，用$p$表示方程$a_1/b_1^p + a_2/b_2^p + ... + a_k/b_k^p = 1$的唯一解，则情况和对应的结论同主定理。

小测验

1. 用自己的话叙述主定理。
2. 解递归方程$T(n) = T(n/4) + T(n/2) + 1$。
3. 解递归方程$T(n) = 2T(n/2) + n/\log n$。（提示：使用递归树方法）

4.1.4 运算的加速

本节讨论一个很有意思的话题：利用分治法给普通的运算进行加速。首先考虑整数乘法的Karatsuba算法，然后再考虑矩阵乘法的Strassen算法，最后简单介绍FFT。

Karatsuba算法 考虑两个n位整数x和y的乘法。基本的算法是$O(n^2)$的。以10进制为例，用bigint表示大整数类型，s[1]表示个位，len为位数。乘法的程序如下：

```
bigint mult(bigint a, bigint b)
{
  bigint c;
  c.len = a.len + b.len;
  for(i = 1; i <= c.len; i++) c.s[i] = 0;
  for(i = 1; i <= a.len; i++)
    for(j = 1; j <= b.len; j++)
      c.s[i+j-1] += a.s[i]*b.s[j];
  for(i = 1; i <= c.len; i++)
      { c.s[i+1] += c.s[i] / 10; c.s[i] %= 10; }
}
```

这个算法实际上是先计算每一位上的原始数，然后从低位往高位调整使得每一位都是0~9。这个算法需要原始数不上溢。对于第k位，满足$i+j-1=k$的数对(i,j)最多有n个，因此原始数最多$81n$，在时间可以承受的范围之内都不会溢出。

现在考虑分治算法。取$m=(n+1)/2$，把x写成$10^m \times a+b$的形式，y写成$10^m \times c+d$的形式，则a, b, c, d都是m位整数(如果不足m位，前面可以补0)。

$$xy=(10^m a+b)(10^m c+d)=10^{2m}ac+10^m(bc+ad)+bd \tag{4.2}$$

递归方程为$T(n)=4T(n/2)+\Theta(n)$，其中系数4为四次乘法ac, bd, bc, ad，附加代价n为最后一个return语句的两次高精度加法。方程的解为$T(n)=\Theta(n^2)$，和暴力乘法没有区别。

Anatolii Karatsuba在1962年提出一个改进方法（并由Knuth改进）：用ac和bd计算$bc+ad$，即：

$$bc+ad=ac+bd-(a-b)*(c-d) \tag{4.3}$$

这样一来，只需要进行三次递归乘法，即递归方程变为了$T(n)=3T(n/2)+\Theta(n)$，解为$T(n)=\Theta(n^{\log 3})=\Theta(n^{1.585})$，比暴力乘法快。

计算整数乘法的最快算法是基于FFT的，它的时间复杂度为$O(n\log n)$。

Strassen矩阵乘法 一个$n \times n$矩阵是n行n列的数排列起来的方阵。矩阵A第i行第j列数记为A_{ij}。定义两个n阶矩阵的乘法$C = A \times B$，其中$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj}$

基本的矩阵乘法是O(n^3)的，程序如下：

```
void matmult(matrix a, matrix b, matrix& c)
{
  c.m = a.m; c.n = b.n
  for(int i = 1; i <= a.m; i++)
  for(int j = 1; j <= b.n; j++)
  {
      c.num[i][j] = 0;
      for(int k = 1; k <= a.n; k++)
          c.num[i][j] += a.num[i][k] * b.num[k][j];
  }
}
```

考虑基本分治算法。把矩阵分两四个子矩阵。

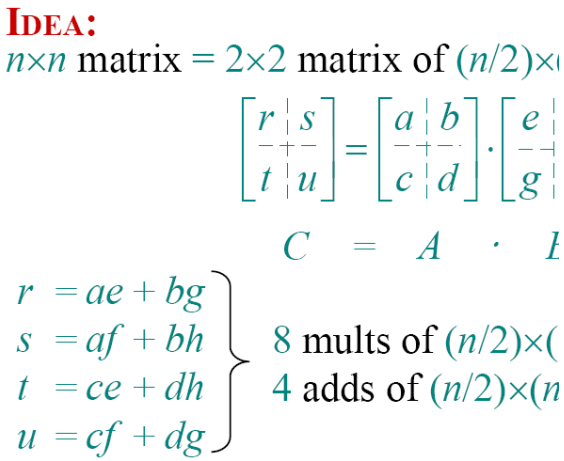


图 4.9: 矩阵分割

则r = ae + bg, s = af + bh, t = ce + dh, u = cf + dg，一共需要8次乘法和4次加法。因为每次加法需要Θ(n^2)的时间，因此递归方程为T(n)=8T(n/2)+ Θ(n^2)，解为O(n^3)，和暴力乘法一样。

Strassen的改进方法和Karatsuba快速乘法类似。记$P_1 = a(f - h)$, P_2=$(a + b)h$, P_3=$(c + d)e$, $P_4 = d(g - e)$, P_5=$(a + d)(e + h)$, P_6=$(b - d)(g + h)$, P_7=$(a - c)(e + f)$，则：$r = P_5 + P_4 - P_2 + P_6$, $s = P_1 + P_2$, $t = P_3 + P_4$, $u = P_5 + P_1 - P_3 - P_7$。一共7次乘法，18次加减法，而且并没有使用过乘法交换律（矩阵乘法不满足交换律）！

这样，方程变为$T(n) = 7T(n/2) + \Theta(n^2)$，它的解为$O(n^{\log 7})$=$O(n^{2.81})$，比暴力乘法略好。目前最好的矩阵乘法算法时间复杂度为$O(n^{2.376})$，但是人们还不知道矩阵乘法的复杂度下限是多少。

小测验

1. Karatsuba算法和Strassen算法的基本思想是什么？
2. 为了实现Karatsuba算法，需要实现哪些高精度运算？要考虑哪些问题？
3. 为了让Strassen算法适用于任意n，需要做哪些修改？

4.1.5 排序与顺序统计

本小节介绍基于分治的排序算法和顺序统计算法。所谓排序，即把n个数从小到大排成一行；所谓顺序统计，即给定$1 <= k <= n$，在n个数中找出第k大的数。

归并排序(merge sort) 。按照分治三步法来说，归并的过程是：

划分 把序列分成元素个数相等的两半。

递归求解 把两半分别排序。

合并 把两个有序表合成一个。

显然，前两部分是很容易完成的，关键在于如何把两个有序表合成一个。下图演示了一个合并的过程。每次只需要把两个序列中当前的最小元素加以比较，删除较小元素并加入合并后的新表。由于需要一个新表来存放结果，所以附加空间为n。

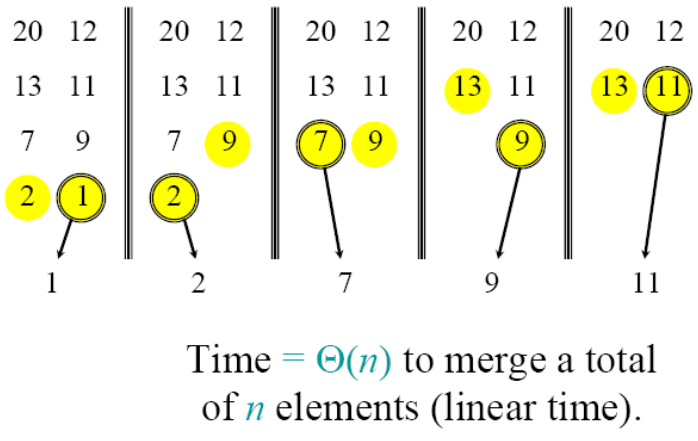


图 4.10: 合并有序表

这样，时间复杂度为$T(n) = 2T(n/2) + n$，由主定理得时间复杂度为$\mathrm{O}(n \log n)$。

逆序对数(inverse number) 给一列数$a_1, a_2, \ldots, a_n$，求它的逆序对数，即有多少个有序对i, j使得$i < j$但$a[i] > a[j]$。最简单的方法是二重循环，时间复杂度为$\mathrm{O}(n^2)$。如果利用归并排序，我们可以轻松做到这一点。划分和递归求解部分完全一致，只有合并部分需要修改：需要加上跨越两边的逆序对。考虑右表中的任意元素k。当k从右表中删除时，左表剩下的所有元素x都与k构成(x, k)逆序对，因此应该在总答案中加上左表此时的元素个数。

下面的程序中，左表初始为$[l, m]$而右表为$[m + 1, r]$，两个表各有一个指针i和j，因此在任意时刻左表为$[i, m]$而右表为$[j, r]$。如果右表为空或者两表都不空且$a[i] \leq a[j]$时，元素i从左表中移出；否则元素j从右表中移出，并累加左表元素个数$m - -j + 1$。

```
void sort(int l, int r)
{
  if(l >= r) return 0;

  int m = (l + r) >> 1;
  sort(l, m);
  sort(m+1, r);

  // merge [l,m] and [m+1,r]
  int i = l, j = m+1, c = l;
  while(i <= m && j <= r)
    if(j > r || (i <= m && a[i] <= a[j]) )
        t[c++] = a[i++];
    else
        { t[c++] = a[j++]; cnt += m -  i + 1; }

  // copy back
  for(i = l; i <= r; i++) a[i] = t[i];
}
```

只要把cnt += m - i + 1;这一句去掉，就是标准的归并排序。时间复杂度也是$\mathrm{O}(n \log n)$，比普通的$\mathrm{O}(n^2)$算法快很多。需要注意的是附加空间t数组是$\Omega(n)$的。

快速排序(quick sort) 快速排序也许是最快的通用排序算法，它由Hoare于1962年提出，不像归并排序那样需要辅助空间，而是原地排序。我们先来看它的各个分治步骤。

> 划分 把数组的各个元素调整成以下的形式：以某元素x为轴心，左边的元素比x小，右边的元素比x大。

递归求解 左右两边分别排序。

合并 不用合并，因为此时数组已经完全有序。

在快速排序中，递归求解和合并都是平凡的，关键在于划分过程。划分过程有很多种，这里介绍两种。

方法一 以$a[p]$为轴心，在任意时刻都维持以下形状。

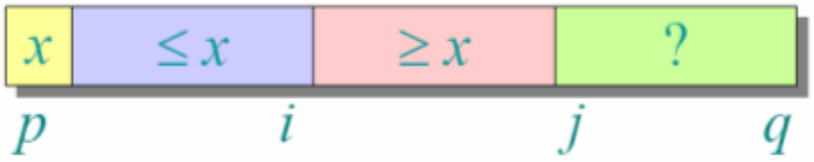


图 4.11: 第一种划分法

j不停往右移，如果发现小于等于x的元素就让i加1并交换$a[i]$和$a[j]$。最后交换$a[p]$和$a[i]$，这样可以让划分平均一些。

```
int partition(int p, int q)
{
  int x = a[p], i = p;
  for(j = p+1; j <= q; j++)
      if(a[j] <= x) swap(a[++i], a[j]);
  swap(a[p], a[i]);
  return i;
}
```

方法二 也是维持两个指针，确保i前面的都比x小，j后面的都比x大。然后i不断加1，碰到不满足条件的停下来；j不断减1，碰到不满足条件的停下来。由于$a[i] > x$, $a[j] < x$，只需要交换$a[i]$和$a[j]$，则二者都将满足条件。

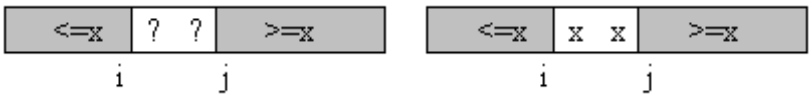


图 4.12: 第二种划分方法

程序如下。

```
int partition(int p, int q)
{
  int x = a[p], i = p, j = q + 1;
  while(i < j)
```

```
  {
      while(a[++i] < x);
      while(a[--j] < x);
      swap(a[i], a[j]);
  }
}
```

快速排序的时间复杂度取决于划分后左右部分各有多少个元素，而两部分各有多少元素并不容易控制。通常采取随机的方法选择pivot，期望时间复杂度为$O(n \log n)$，程序如下：

```
void qsort(int p, int r)
{
  if(p < r)
  {
    int i = p + rand()% (r -  p + 1);
    swap(a[i], a[p]);
    int q = partition(p, r);
    qsort(p, q-1);
    qsort(q+1, r);
  }
}
```

快速选择(quick select) 利用快速排序很容易求出n个数中第k大的数，因此在划分后如下图。

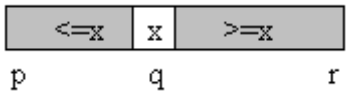


图 4.13: 划分后数组示意图

左半（包括q）有$j = q - p + 1$个元素。如果$k < j$，只需要在左半找第k大元素；如果$k > j$，需要在右半找第$k - j$大元素。可以证明，期望时间复杂度为$O(n)$。

```
int select(int p, int r, int k){
  if(p == r) return p;
  int i = p + rand()%(r -  p + 1); swap(a[i], a[p]);
  int q = partition(p, r);
  int j = q -  p + 1;
  if(k == j) return q;
  else if(k < j) return select(p, q, k);
  else select(q+1, r, k-j);
}
```

线性时间选择(linear-time select) 快速选择在期望意义下是线性的，事实上还

存在最坏情况线性的算法。它由Blum, Floyd, Pratt, Rivest和Tarjan于1973年提出。它的思想是：pivot不是随机选择，而是采取某种确定性的策略，使得每次递归调用后保证长度至少缩短为原来的a倍($a < 1$)。这样，T(n)≤ T(an)+O(n)，由主定理得T(n)=O(n)。

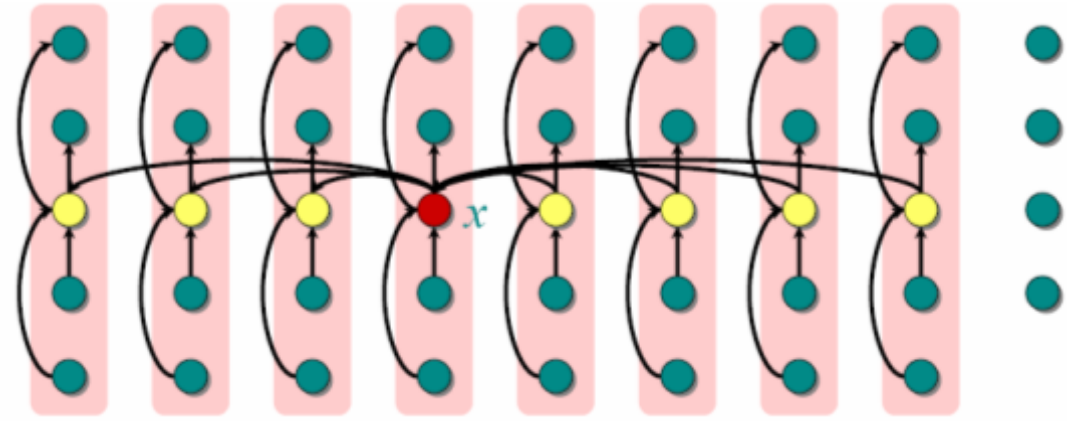


图 4.14: 线性时间选择算法

如图 8.1.2所示，把所有元素分成5个一组共约$n/5$组，每组用暴力法求出中位数（图中为中心行白点所示），一共有$n/5$个中位数。用递归法求出这$n/5$个数的中位数x，则所有k个中位数中至少有约$n/10$个数比x小。由于每个中位数在自己组中至少有3个数不比它大（也就不会比x大），因此至少有$3(n/10) = 0.3n$个数比x小。更精细的分析指出，当$n \geq 75$时至少有$n/4$个数比x小。同理，也至少有$n/4$个数比x大。求$n/5$个数的中位数需要T($n/5$)的时间，而划分后的选择不超过T($3n/4$)，因此T(n)≥T($n/5$)+T($3n/4$)+O(1)，解为T(n)=O(n)。

小测验

1. 为什么归并排序的时间复杂度好分析，而快速排序不好分析？
2. 既然线性选择保证每次划分时两边差距不会太大，为什么不用它来加速快速排序？
3. 在理论上归并排序的时间复杂度较小，为什么实际上快速排序更快？

4.2 贪心法

贪心是一种解决问题的策略。如果这种策略正确，那么贪心法往往是易于描述易

于实现的。本节介绍可以用贪心法解决的若干经典问题。

4.2.1 几个简单的例子

这里举几个简单的例子，一方面说明贪心策略的含义，另一方面说明贪心法正确性的证明方法。

最优装载问题 给n个物体, 第i个物体重量为w_i, 选择尽量多的物体, 使得总重量不超过C。

由于目标是物体的"数量"尽量多，所以装重的没有装轻的划算。只需把所有物体按重量从小到大排序，依次选择每个物体，直到装不下为止。这就是一种贪心法，因为每次都是选择能装下的最轻的物体，是一种"只顾眼前"的策略。幸好本题很特殊，这样的策略保证能得到最优解。

部分背包问题 有n个物体, 第i个物体的重量为w_i, 价值为v_i, 在总重量不超过C的情况下让总价值尽量高。每一个物体可以只取走一部分，价值和重量按比例计算。

本题在上一题的基础上增加了价值，所以不能简单的像上题那样先拿轻的（轻的可能价值也小），也不能先拿价值大的（可能它特别重），而应该综合考虑两个因素。一种直观的贪心策略产生：优先拿价值/重量比最大的，直到重量和正好为C。由于每个物体可以只拿一部分，因此一定可以让总重量恰好为C（或者全部拿走重量也不足C），而且除了最后一个以外，所有的物体要么不拿，要么拿走全部。

乘船问题 有n个人, 第i个人重量为w_i。每艘船的载重量均为C, 最多乘两个人。用最少的船装载所有人。

考虑最轻的人i，他应该和谁一起坐呢？如果和每个人都无法一起坐同一艘船，则唯一的方案就是每人坐一艘船（想一想，为什么）。否则选择能和i一起坐船的人中最重的一个j。这样的方法是贪心的，因此它只是让"目前船的浪费尽量少"。这个贪心策略是对的，可以用反证法说明。

假设这样做不是最好的，考虑最好方案中i是什么样的。

情况一 i不和任何一个人坐同一艘船，那么可以把j拉过来和他一起坐，总船数不会增加（且可能会减少！），并且符合刚才的贪心策略。

情况二 i和另外一人k同船，由贪心策略，k比j轻。把j和k交换后k原来所在的船仍然不会超重（因为j比k轻），而i和k所在的船也不会超重（由贪心法过程），因此所得到的新解不会更差，且符合贪心策略。

综上所述，虽然可能不采取贪心策略也能得到最优解，但是只考虑贪心策略肯定不会丢失最优解。贪心法往往容易实现。在本题中，只需每次寻找最小值和能与它同船的最大值配对即可。

通过三个例子，我们大致看到了贪心算法的形式和正确性证明的基本方法。在接下来的几小节中，我们通过对几个经典题目的分析加深对贪心法的认识。

小测验

1. 证明部分背包问题的贪心法是正确的。如果每个物体要么完全取要么完全不取，贪心策略还正确吗？

2. 在乘船问题中，是否每次一定要考虑最轻的人？从正确性证明的过程加以说明。

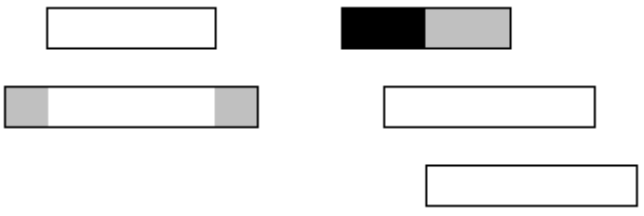
3. 讨论乘船问题的贪心法的时间复杂度。存在更快的算法吗？

4.2.2 区间上的问题

本节讨论几个具有一定相似之处的问题。它们都和数轴上的线段或称区间有关。

不相交的区间选择问题 数轴上有n条开区间(a_i, b_i)，选择尽量多个区间，使得这些区间两两没有公共点。

首先明确一个问题：如果有两个区间x, y，区间x完全包含y。那么显然选x是不划算的，因为x和y最多只能选一个，选x还不如选y，这样不仅区间数目不会减少，而且给其他区间留出了更多的位置。这样，我们按照b_i从小到大的顺序给区间排序。贪心策略是：一定要选第一个区间。为什么？



(a) $a_1 > a_2$ (b) $a_1 < a_2 < a_3$

图 4.15: 贪心策略图示

现在区间已经按照排序了，即$b_1 \le b_2 \le b_3 \ldots$，考虑$a_1$和$a_2$的大小关系。

情况一 $a_1 > a_2$，如图8.1.2(a)所示，区间2包含区间1。前面已经讨论过，这种情况下一定不会选择区间2。不仅区间2如此，以后所有区间中只要有一个i满足$a_1 > a_i$，i都不要选。在今后的讨论中，我们不考虑这些区间。

情况二 排除了情况一，一定有$a_1 \leq a_2 \leq a_3 \leq ...$。如果区间2和区间1完全不相交，那么没有影响（因此一定要选区间1），否则区间1和区间2最多只能选一个。注意到如果不选区间2，黑色部分其实是没有任何影响的（它不会挡住任何一个区间），区间1的有效部分其实变成了灰色部分，它被区间2所包含！由刚才的结论，区间2是不能选的。以此类推，不能因为选任何区间而放弃区间一，因此选择区间一是明智的。

选择了区间一以后，需要把所有和区间一相交的区间排除在外，需要记录上一个被选择的区间编号。这样，在排序后只需要扫描一次即可完成贪心过程，得到正确结果。

选点问题 数轴上有n个闭区间$[a_i, b_i]$。取尽量少的点，使得每个区间内都至少有一个点（不同区间内含的点可以是同一个）。

如果区间i内已经有一个点被取到，我们称此区间已经被满足。受到上一题的启发，我们先讨论区间包含的情况。由于小区间被满足时大区间一定也被满足。所以在区间包含的情况下，大区间不需要考虑。

把所有区间按b从小到大排序（b相同时a从大到小排序），则需要考虑的区间的a值也是从小到大排好序的（其他区间都不需要考虑）。第一个区间应该取哪一个点呢？我们的贪心策略是：取最后一个。

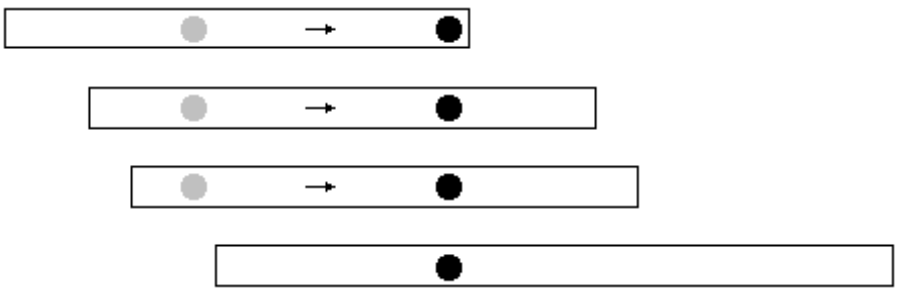


图 4.16: 贪心策略

如果不是取最后一个，比如说灰色点，那么把它移动到最后一个点后被满足的区间增加了，而且原先被满足的区间现在一定被满足。不难看出，这样的贪心策略是正

确的。

区间覆盖问题 数轴上有n个闭区间$[a_i, b_i]$，选择尽量少的区间覆盖一条指定线段$[s, t]$。

本题的突破口仍然是区间包含和排序扫描，不过先要进行一次预处理。每个区间在$[s, t]$外的部分都应该预先被切掉，因为它们的存在是毫无意义的。在预处理后，在相互包含的情况下，小区间显然不应该考虑。

按照a从小到大排序。如果区间1的起点不是s，无解（因为其他区间的起点更大，不可能覆盖到s点），否则选择起点在s的最长区间。选择此区间后$[a_i, b_i]$，新的起点应该设置为b_i，并且忽略所有区间在b_i之前的部分，就像预处理一样。虽然贪心策略比上题复杂，但是仍然只需要一次扫描，如下图，s为当前有效起点（此前部分已被覆盖），则应该选择区间2。

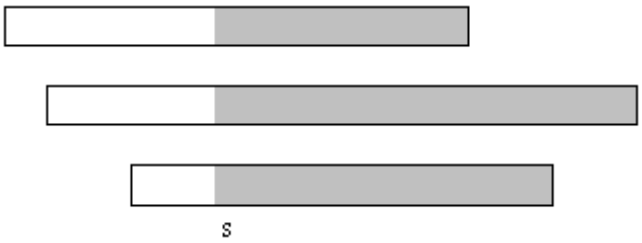


图 4.17: 区间覆盖问题

小测验

1. 选点问题中，把所有区间按照a从小到大排序然后执行贪心法可以吗？为什么？

2. 选点问题中，为什么b相同时要按a从大到小排列？如果a从小到大排列会怎样？

3. 在区间覆盖问题中，如果在排序时a相同时规定按b从大到小排序，每次一定选择区间1，得到的结果是正确的吗？为什么？

4.2.3 Huffman编码

本节介绍的问题同时具有理论和实用价值，它可以用来进行文件压缩。假设有一个文件只有六种字符：a, b, c, d, e, f，我们可以用三个位来表示，如下表：

字符	a	b	c	d	e	f
频率(千次)	45	13	12	16	9	5
编码	000	001	010	011	100	101

表 4.1: 各种字符的编码

字符	a	b	c	d	e	f
频率(千次)	45	13	12	16	9	5
编码	0	101	100	111	1101	1100

表 4.2: 变长码

字符	a	b	c	d	e	f
频率(千次)	45	13	12	16	9	5
编码	0	1	00	01	10	11

表 4.3: 错误的变长码

由于每个字符恰好占3位，一共需要(45+13+12+16+9+5)×1000×3=300,000位。还有一种编码方案，称为变长码，如下：

总长度为(1×45+3×13+3×12+3×16+4×9+4×5)×1000=224,000位，比定长码省。读者可能会说：还可以更省，比如：

总长度只有(1×(45+13)+2×(12+16+9+5))×1000=142,000，不是更省吗？这样的编码是有问题的。如果收到了001，到底是aab还是cb还是ad？这样码有歧义。歧异产生的原因是其中一个字符的编码是另一个码的**前缀(prefix)** 。表2的码没有这样的情况，任一个编码都不是另一个的前缀。我们把满足这样性质的编码称为**前缀码(prefix code)** 。这样，我们可以正式的叙述编码问题。

编码问题 给出n个字符的频率ci，给每个字符赋予一个01编码串，使得任一个字符的编码不是另一个字符编码的前缀，而且编码后总长度（每个字符的频率与编码长度乘积的总和）尽量小。

在解决这个问题之前，首先来看一个结论：任何一个前缀编码都可以表示成所有非叶结点都恰好有两个儿子的二叉树。如下图，每个非叶结点与左儿子的边上写0，与右儿子的边上写1。

每个叶子对应一个字符，编码为从根到该叶子的路径上的01序列。在上图中，N的编码为110，而E的编码为00。为了证明一般情形，我们需要说明两件事情。

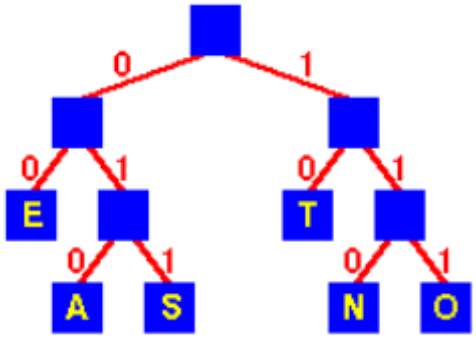


图 4.18: 前缀码的二叉树表示

结论一 n个叶子的二叉树一定对应一个前缀码。如果编码a为编码b的前缀，则a所对应的结点一定为b所对应结点的祖先。而两个叶子不会有祖先后代的关系。

结论二 最优前缀码一定可以写成二叉树。逐一个字符构造即可。每拿到一个编码，都可以构造出从根到叶子的一条路径，沿着已有结点走，创建不存在的结点。这样得到的二叉树可能有节点只有一个儿子。只需要用这个儿子代替父亲，得到的仍然是前缀码且总长度更短。

接下来的问题变为：如何构造一棵最优的编码树。

Huffman算法 把每个字符看作一个单结点子树，每棵子树的权值等于相应字符的频率。每次取权值最小的两棵子树合并成一棵新树，并重新放到树集合中。新树的权值等于两棵子树权值之和。

我们分两步证明Huffman算法的正确性。

结论一 设x和y是频率最小的两个字符，则存在前缀码使得x和y具有相同码长且仅有最后一位编码不同。换句话说：贪心选择一定保留最优解。

证明：假设深度最大的结点为a，则a一定有一个兄弟b。不妨设$f(x) \leq f(y)$, $f(a) \leq f(b)$，则$f(x) \leq f(a)$, $f(y) \leq f(b)$。如果x不是a，把x和a交换；如果y不是b，把y和b交换。这样得到的新编码树不会比原来的差。

结论二 设T是加权字符集C的最优编码树，x和y是树T中两个叶子且为兄弟，z是它们的父亲。若z看成是具有频率$f(z) = f(x) + f(z)$的字符，则树$T' = T - \{x, y\}$是字符集$C' = C - \{x, y\} \cup \{z\}$的一棵最优编码树。换句话说：原问题的最优解包含子问题的最优解。

证明：设T'的编码长度为L，其中字符{x, y}的深度为h，则把字符{x, y}拆成两个后长度变为$L - -[f(x) + f(y)] \times h + f(x) \times (h+1) + f(y) \times (h+1) = L + f(x) + f(y)$。

因此T'必须是C'的最优编码树，T才是C的最优编码树。

结论一通常称为贪心选择性质，结论二通常称为最优子结构性质。根据这两个结论，Huffman算法正确。

在程序实现上，可以先按照频率把所有字符排序成表P，然后另外设置队列Q。每次合并两个结点后放到队列Q中。由于后合并的频率和一定比先合并的频率和大，因此Q内的元素是有序的。类似有序表的合并过程，每次只需要检查P和Q的首元素即可找到频率最小的元素，时间复杂度为O(n)。加上按频率排序的过程，总O($n\log n$)。也可以用堆把表P和表Q合成一个，渐进时间复杂度仍为O($n\log n$)，但时间效率略低。

小测验

1. 什么是贪心选择性质？证明前两小节六个问题的贪心选择性质。
2. 什么是最优子结构性质？证明前两小节六个问题的最优子结构性质。
3. 如果编码有0、1、2三个字符，Huffman算法应该怎样修改？

4.2.4 两个调度问题

本节讨论两个看起来困难的问题。

流水作业调度问题 有n个作业要在两台机器M_1和M_2组成的流水线上完成加工。每个作业i都必须先花时间a_i在M_1上加工，然后花时间b_i在M_2上加工。确定n个作业的加工顺序，使得从作业1在机器M_1上加工开始到作业n在机器M_2上加工为止总时间最少。

直观上，最优调度一定让M_1没有空闲，而让M_2的空闲时间尽量少。下面的Johnson算法是贪心的，保证能得到最优解。

Johnson算法 设N_1为$a < b$的作业集合，N_2为$a \geq b$的作业集合，将N_1中的作业按照a非减序排序，N_2中的作业按照b非增序排序，则N_1作业接N_2作业构成最优顺序。显然算法的核心只是一个排序过程，时间复杂度为O($n\log n$)，关键在于它的正确性证明。

考虑任务i刚开始在M_1执行的时候。设此时M_2还需要等待t时间才可以执行任务i。显然M_1需要a_i的时间可以完成，关键是求出M_2需要的时间$f(t,i)$。

情况一 此时任务i已经在M_1加工完成（即$a_i \leq t$），再执行b_i时间即可。

情况二 此时任务i还没有加工完成，需要等待加工完成，即花费$a-t+b$时间。

综合一下，M_2需要的时间为$f(t,i) = max\{a_i - t, 0\} + b_i$。

考虑两个相邻任务i和j的顺序。

图 4.19: 相邻任务的可能顺序

显然M_1需要的时间一定是$a_i + a_j$。因此只需要考虑M_2。如果i在j前面，执行i后M_2还需要的时间为$f(t,i)$，因此执行完i和j后，M_2还需要的时间为$max\{a_j - f(t,i), 0\} + b_j$，即$max\{a_j - -(max\{a_i - -t, 0\} + b_i), 0\} + b_j$，化简得

$$b_i + b_j - a_i - a_j + max\{t, a_i + a_j - b_i, a_i\} \tag{4.4}$$

前四项和t都是定值，因此要$max\{a_i + a_j - b_i, a_i\} = a_i + a_j + max\{-b_i, -a_j\}$尽量大。由于前两项是定值，因此要$max\{-b_i, -a_j\}$尽量大，即当且仅当$max\{-b_i, -a_j\} \geq max\{-b_j, -b_i\}$，即$min\{b_i, a_j\} \geq min\{b_j, a_i\}$时，$i$应该排在$j$前面。这个条件称为Johnson不等式。

可以进一步把这个不等式写成更容易懂的形式。设i和j是相邻任务，则有

结论一 如果$a_i < b_i$且$a_j \geq b_j$，则i排在j前面。

证明：用不等式放缩得：$min\{b_i, a_j\} > min\{a_i, a_j\} \geq min\{a_i, b_j\}$

结论二 如果$a_i < b_i$，$a_j < b_j$，且$a_i \leq a_j$，则i排在j的前面

证明：用不等式放缩得：$min\{b_i, a_j\} \geq min\{b_i, a_i\} = a_i \geq min\{a_i, b_j\}$

结论三 如果$a_i \geq b_i$, $a_j \geq b_j$, 且$b_i \geq b_j$，则i排在j的前面

证明：用不等式放缩得：$min\{b_i, a_j\} \geq min\{b_j, a_j\} = b_j \geq min\{a_i, b_j\}$

显然，这样的序关系有传递性，它是序关系。因此Johnson不等式可以推广到不相邻的任意两个任务之间。这就证明了一开始提出的贪心算法。

带限期和罚款的单位时间任务调度 有n个单位时间任务，每个任务需要单位长度时间。第i个任务的限期是d_i，超时罚款为f_i，给n个任务排序使得所有未按时完成的任务罚款总和尽量小。

本题的特殊之处在于所有任务都是单位时间的，因此对于时间的计算是简单的。为了解决这个问题，先介绍矩阵胚理论。

矩阵胚(matroid) 给定非空有限集S，I是S的一类具有遗传性质的独立子集族（即空集属于I，且对于I里的任意元素B，B的任意子集也属于I）。我们把I中的元素称为独立子集。如果I满足交换性质（即对于I的任意两个元素A和B，若$|A| < |B|$，一定可

以找到一个元素$x \in B - A$，使得$A \cup \{x\}$仍然是独立集），称有序对(S, I)为矩阵胚，或称拟阵。

对于独立集A，如果存在x使得$A \cup \{x\}$仍是独立集，称x为A的可扩展元素。没有可扩展元素的独立集A称为极大独立集。

定理1：矩阵胚的所有极大独立集具有相同的基数。

证明：使用反证法。如果存在两个极大独立集A和B满足$|A| < |B|$，由交换性质得$B - A$中一定由A的可扩展元素，与A是极大独立集矛盾。

最大权独立子集问题 给S的每个元素x赋予正权$W(x) > 0$，独立集的权被定义为其中所有元素的权和。求加权矩阵胚中权最大的独立子集。

贪心算法 从空集开始（权为0），每次选择权最大的可扩展元素加入独立集中，直到不存在可扩展元素为止，此时得到的独立集一定是权最大独立子集。为了证明算法的正确性，我们仍然考察问题的贪心选择性质和最优子结构，证明留给读者。

回到原问题中。称在限期内完成的任务为“早任务”, 收到罚款的任务为“迟任务”。显然早任务紧跟在迟任务之后是不划算的, 因为交换之后总罚款不变。设任务i的执行时间为t_i，考虑相邻两个早任务i和$i+1$。

由于两个任务都是早任务, 因此$t_{i+1} \leq d_{i+1}$。若$d_i > d_{i+1}$, 则$t_{i+1} \leq d_{i+1} < d_i$, 交换以后显然$i+1$的时间更早, 故仍然是早任务；$i$的时间$t_i' = t_{i+1} < d_i$仍然是早任务, 总罚款不变。根据这个结论，我们可以任务表写成规范形式：所有早任务在迟任务前, 且按限期非递减排序。

给一个任务集，先把它们按限期非递减排序。如果有的任务是迟的，那么不管怎么调度都不可能让这些任务都是早任务。如果所有任务都是及时的，那么此任务集称为原任务集S的独立子集。把所有独立子集的集合记为I。

显然迟任务的排列顺序可以随意，因此只需要选择罚款和尽量大的早任务集合（对应罚款和尽量小的迟任务集合）即可。如果(S, I)是矩阵胚，那么可以用贪心算法求解。

引理：对于任意任务子集A，以下命题等价

1. A是独立子集。

2. 对于$t = 1, 2, 3, \ldots n, N(A, t) \geq t$，其中$N(A, t)$表示截止时间不超过$t$的任务数。

3. 把A中任务按截止时间排序，则A中所有任务都是及时的。

证明： 1◇2：如果不满足2，肯定安排不过来，因此一定满足2。

2◇3：由于已经按限期排序，如果任务i的超时了，一定有$d_i < i$，因此N(A, d_i) $\geq i > d_i$，和性质2矛盾，如图 8.1.3。

3◇1：既然已经构造出可行解了，当然是独立子集了。

图 4.20: d_i和i的关系

进一步的，我们有：

定理： 在本题中，(S, I)是矩阵胚。显然I满足遗传性质，只需要证明它满足交换性质。设A和B为独立子集且$|A| < |B|$。取k为满足$N(A,t) \geq N(B,t)$的最大t。由于$N(A,n) = |A| < |B| = N(B,n)$，因此$k < n$，且对于任意$k+1 \leq j \leq n$，有$N(A,j) < N(B,j)$。

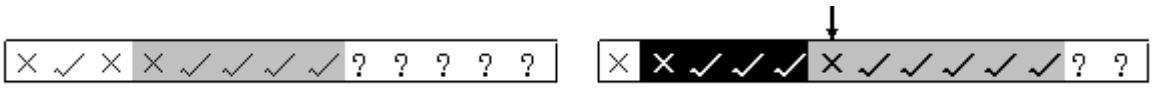
在$B-A$中随便选取截止时间至少为$k+1$的任务。因为$N(A,k+1) < N(B,k+1)$，所以一定找得到。

因为A是独立的，因此对于$1 \leq t \leq k$，有$N(A',t) = N(A,t) \geq t$；又因为B是独立的，因此对$k < t \leq n$有$N(A',t) \leq N(A,t)+1 \leq N(B,t) \leq t$。因此$A'$是独立的。这样，我们证明了$(S, I)$是矩阵胚。

算法实现 如何高效的实现贪心算法？贪心法的流程是：把所有元素按权值从大到小排序；依次考虑各个元素，如果是当前集合的可扩展元素，则加入，否则丢弃。排序的时间复杂度为O($n \log n$)，关键是判断元素x是否当前集合A的可扩展元素。

一个简单的实现法是更新$N(A)$并根据引理的性质2判断，每加入一个元素需要O(n)的时间，总O(n^2)。利用并查集，我们可以把时间复杂度进一步降低。改进的思想是：判断A是否为独立集的同时构造出解，即每次选择限期以内最靠右的位置，以给其他任务腾出空间。如图(a)，打勾的位置表示已经安排了任务，打叉的表示空闲时间。灰色区域属于同一个等价类，因为如果限期是灰色四个格子之一，都是插入到其中的第一个格子。

初始时所有格子打叉，各为一个集合。每次填入一个任务时把它和它左边的集合合并，如图(b)所示，放入箭头指向的位置后，应将黑色和灰色两个集合合并。每次判



(a) 等价类(b) 等价类的合并

图 4.21: 基于并查集的实现

断独立集只需要约O(1)的时间，因此总时间复杂度为O($n \log n$)。如果不需要排序，则接近O(n)。

小测验

1. 在流水作业调度问题中，为什么只需要对相邻任务讨论就可以确定任意两个任务的顺序？

2. 什么是矩阵胚？试举出两个矩阵胚的例子。它们满足怎样的性质？

3. 在单位时间任务调度问题中，描述如何用并查集高效判断独立集的方法。

4.3 动态规划

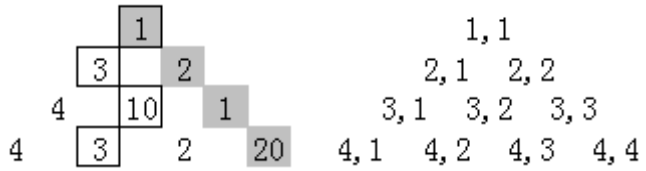
和贪心类似的是，动态规划也是用来求解优化问题的。适合动态规划求解的问题也满足最优子结构性质，却不一定满足贪心选择性质。事实上，在动态规划算法中，贪心选择性质被替换成了无后效性。动态规划思想应用十分广泛，本节不仅介绍其基本思想，还将分析几个经典问题，并提出改进方案。

4.3.1 基本思想

动态规划的思想源于递归，是一种问题转化策略。基于递归的问题转化策略有很多，他们相互之间也不是很容易区分，所以这里并不想严格的讲述什么是动态规划，而是通过例子让读者不断体会。

数字三角形 有一个由正整数组成的三角形，第一行只有一个数，除了最下行之外每个数的左下方和右下方各有一个数，如下图所示。

从第一行的数开始，每次可以往左下或右下走一格，直到走到三角形底端，把沿途经过的数全部加起来作为得分。如何走，使得这个得分尽量大？



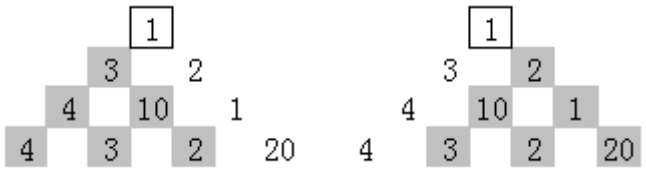
(a) 数字三角形(b) 格子编号

图 4.22: 数字三角形问题

这是一个决策问题：每次选一个方向（左/右）行走。最容易想到的是贪心算法：每次选择数字较大的方向走。可惜和上节不一样，贪心法无法保证得到最优解。在上例中，贪心法讲找到带方框的路线1-3-10-3，和为17，而最优路线为灰色路线1-2-1-20，和为24。显然，贪心的错误在于“目光短浅”，它不知道第一次贪心将错过以后的20。

下面的解法是直观的，虽然正确性并不显然：设以格子(i,j)为首的“子三角形”的最大和为d[i,j]（我们将不加区别的把这个**子问题(subproblem)** 本身也称为d[i,j]），则原问题的解是d[1,1]。从格子(i,j)出发往左走和往右走将分别到达位置(i+1,j)和(i+1,j+1)，转化为子三角形的问题。也就是说，

$$d[i,j] = a[i,j] + max\{d[i+1,j], d[i+1,j+1]\}$$



(a) 子问题d[2,1] (b) 子问题d[2,2]

图 4.23: 两个子问题

例如d[1,1]的求解转化为了上面两个子问题，即d[2,1]和d[2,2]，这两个应该取较大值，因此d[1,1]=1+max{d[2,1], d[2,2]}。其中加上的那个“1”是原三角形顶上的数字。

如图 4.24是d[2,1]的两个子问题d[3,1]和d[3,2]，因此$d[2,1] = 3 + max\{d[3,1], d[3,2]\}$

如图 4.25是d[2,2]的两个子问题d[3,2]和d[3,3]，因此$d[2,2] = 2 + max\{d[3,2], d[3,3]\}$

有了表达式，应该怎样计算呢？

方法一：递归计算

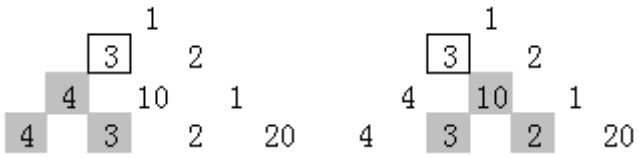


图 4.24: d[2,1]的两个子问题

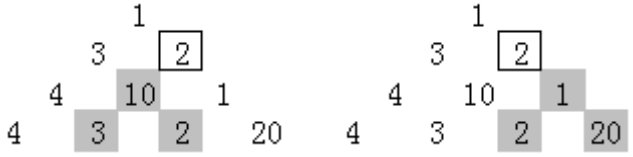


图 4.25: d[2,2]的两个子问题

```
int solve(int i, int j)
{
  if(i == n) return a[i][j];
   else return a[i][j] + max(solve(i+1,j), solve(i+1, j+1));
}
```

这样做是正确的，可惜时间效率太低。低效的原因在于重复计算。

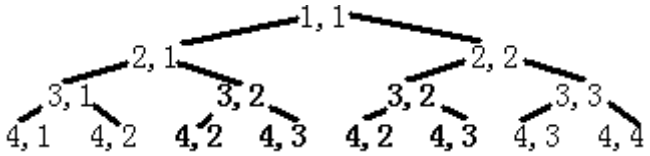


图 4.26: 子问题的重复计算

如上图。子问题d[3,2]被计算了两次。关键在于：这样的重复不是一个结点，而是一棵子树！这样一棵二叉树，有n层，每个结点有两个儿子，因此一共有$\Theta(2^n)$个结点。

方法二：递推计算

```
void solve()
{
  int i , j;
  for(j = 1; j <= n; j++) d[n][j] = a[n][j];
  for(i = n-1; i >= 1; i--)
    for(j = 1; j <= i; j++)
      d[i][j] = a[i][j] + max(d[i+1][j], d[i+1][j+1]);
}
```

这个方法从小到上依次计算每一个值。由于i是逆序枚举，计算到d[i][j]时d[i+1][j]和

d[i+1][j+1]一定已经计算出来了。显然时间复杂度为O(n^2)。

方法三：记忆化搜索

```
// initially, all d[i][j] are -1
int solve(int i, int j)
{
  if(i == n) return a[i][j];
  if(d[i][j] >= 0) return a[i][j];
  d[i][j] = solve(i+1,j), solve(i+1, j+1);
  return d[i][j];
}
```

这个方法和直接递归非常类似，但加入了**记忆化(memoization)**，保证每个结点只访问一次，如下图。

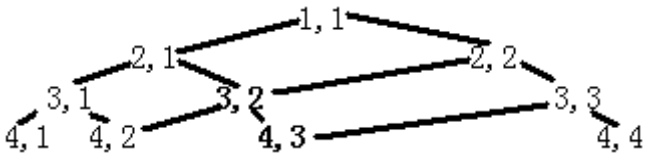


图 4.27: 记忆化搜索

由于$1 \le i, j \le n$，所有不相同的结点一共只有O(n^2)个。不管以怎样的顺序访问，时间复杂度均为O(n^2)。

这就是动态规划的基本思想：建立子问题的描述，建立状态间的转移关系，使用递推或记忆化搜索法来实现。总结一下，动态规划有以下几个要点。

状态定义 用问题的某些特征参数描述一个子问题。在本题中用d[i,j]表示以格子(i,j)为根的子三角形的最大和。在很多时候，状态描述的细微差别将引起算法的不同。在不同的时候使用不同的状态定义是设计动态规划算法时最常见的错误之一。

状态转移方程 即状态值之间的递推关系。这个方程通常需要考虑两个部分：一是递推的顺序，二是递归边界（也是递推起点）。这两点读者应该注意。

从直接递归和后两种方法的比较可以看出：**重叠子问题(overlapping subproblems)** 是动态规划展示威力的关键。

小测验

1. 动态规划的思想来源是什么？它的实现方法有哪两种？

2. 动态规划展示问题的关键是什么？为什么数字三角形问题中直接递归的时间复杂度是指数级别的，而动态规划的时间复杂度是平方级别的？

3. 解释状态定义和状态转移方程。设计状态转移方程应该注意哪两个方面？

4.3.2 动态规划的条件

上一节介绍了数字三角形问题，解法虽然直观，但并不严密。可以把问题稍微变化一下。

数字三角形II 有一个由正整数组成的三角形，第一行只有一个数，除了最下行之外每个数的左下方和右下方各有一个数，如下图所示。

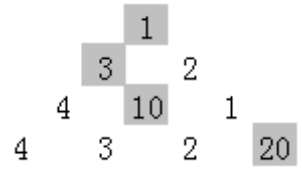


图 4.28: 数字三角形问题II

从第一行的数开始，除了某一次可以走到下一行的任意位置外，每次都只能左下或右下走一格，直到走到最下行，把沿途经过的数全部加起来。如何走，使得这个和尽量大？

稍微修改一下以后，原来的解法不适用了，如果仍然记d[i,j]为以格子(i,j)为根的子三角形的解，那么第一步到底是可以任意移动，还是只能往左下或右下走一格呢？不一定，这取决于解决这个子问题之前做过什么事。

后效性 刚才的状态定义有后效性[1]，即先前的决策可能影响后续的决策。解决方法是扩展状态定义，把有后效性的部分包含进来。记d[i,j,k]为以格子(i,j)为根的子三角形的解，$k = 1$表示可以任意移动，$k = 0$表示不能任意移动，则原来问题的解是d[1,1,1]。当$k = 1$时可以任意移动，然后$k = 0$，而$k = 0$时不能任意移动。状态转移方程需要分类讨论，写成程序是

```
if(i == n) d[i][j][k] = a[i][j];
else{
  d[i][j][k] = max(d[i+1][j][k], d[i+1][j+1][k]); // normal move
```

[1] 无后效性有时也称子问题的独立性

```
  if(k == 1) // teleport
    for(t = 1; t <= i; i++)
        if(d[i][j][k] < d[i+1][t][0]) d[i][j][k] = d[i+1][t][0];
}
```

当然，所有d[i][j][0]的最大值可以边计算边记录，像这样：

```
memset(max, 0, sizeof(max));
for(j = 1; j <= n; j++)
{
  d[n][j][1] = d[n][j][0] = a[n][j];
  if(a[n][j] > max[n]) max[n] = a[n][j]);
}
for(i = n-1; i >= 1; i--){
  for(j = 1; j <= i; j++){
    d[i][j][0] = max(d[i+1][j][0], d[i+1][j+1][0]); // normal move
    if(d[i][j][0] > max[i]) max[i] = d[i][j][0];  // update max
    d[i][j][1] = max(d[i+1][j][1], d[i+1][j+1][1], max[i+1]);
    // normal move/teleport
  }
}
```

免去了一次循环，时间复杂度仍为O(n^2)。

数字三角形III 有一个由正整数组成的三角形，第一行只有一个数，除了最下行之外每个数的左下方和右下方各有一个数，如下图所示。

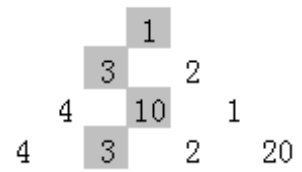


图 4.29: 数字三角形问题III

从第一行的数开始，每次都只能左下或右下走一格，直到走到最下行，把沿途经过的数全部加起来。如何走，使得这个和的个位数尽量大？

这次没有后效性了，但是又有了新的问题。

错误方程一：$d[i,j] = max\{d[i+1,j], d[i,j+1]\} + a[i,j]$。反例：如果$d[i+1,j] = 5$，$d[i,j+1] = 6$，$a[i,j] = 9$，那么加起来是14或15，是两位数；

错误方程二：$d[i,j] = (max\{d[i+1,j], d[i,j+1]\} + a[i,j]) \bmod 10$。反例：$d[i+1,j] = 6$, $d[i,j+1] = 9$，$a[i,j] = 2$，虽然6比9小，但是6+2的个位数比9+2的个位数大。

错误方程三：$d[i,j] = max\{(d[i+1,j] + a[i,j]) \bmod 10, (d[i+1,j+1] + a[i,j]) \bmod 10\}$。这个错误比较隐蔽，它并不是来源于方程本身的错误，而是状态定义的错误。

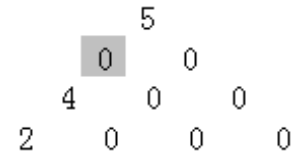


图 4.30: 方程三错误示意图

数字三角形上的数如上图。对于灰色格子(2,1)来说，根据状态定义，d[2,1]=6（从此格子出发路径上数之和的最大个位数），d[2,2]=0（无论怎么走，个位数都是0），根据前面的“递推方程”算出d[1,1]应是1，但实际上d[1,1]等于9。事实上，对于相同的d[2,1]和d[1,1]可能会有不同的d[1,1]，所以不存在从d[2,1]和d[2,2]得到d[1,1]的递推公式！也就是说，并不能怪我们没有找到正确的状态转移方程，实在是因为这样的状态定义根本无法递推！

问题出在哪里呢？关键是：全局最优解5-0-4-0并没有包含子问题最优解0-4-2。由于d值只保留“子问题最优解”，当全局最优解并没有包含子问题最优解时，d值无法递推。

最优子结构 在贪心法介绍中曾经提到过最优子结构。这样的结构在动态规划中也是需要的。不满足最优子结构的情况通常也可以考虑扩展状态定义。在本例中，记d[i,j,k]表示以格子(i,j)为根的子三角形是否存在所有数之和个位为k的路径，则d[i,j,k]为真当且仅当存在t使得$(a[i,j] + t) \bmod 10 = k$，且d[i+1,j,t]或d[i+1,j+1,t]为真。

```
memset(d, 0, sizeof(d));                          // false
for(j = 1; j <= n; j++) d[n][j][a[n][j] % 10] = true;
for(i = n-1; i >= 1; i--)
  for(j = 1; j <= i; j++)
    for(k = 0; k <= 9; k++){
      d[i][j][k] = false;
      for(t = 0; t <= 9; t++) // lookup suitable t
        if ((a[i][j]+t)%10 == k && (d[i+1][j][t] || d[i+1][j+1][t]))
          d[i][j][k] = true;
    }
```

这个程序是对的，但稍稍有点别扭。如果$a[i,j] = 5$和$k = 5$，需要检查的只有$t = 0$，不需要从0到9依次判断一次，即：

```
for(i = n-1; i >= 1; i--)
  for(j = 1; j <= i; j++)
    for(k = 0; k <= 9; k++)
    {
        d[i][j][k] = false;
        t = (10-a[i][j]) % 10;
        if(d[i+1][j][t] || d[i+1][j+1][t])) d[i][j][k] = true;
    }
```

这样就避免了内层循环，时间复杂度为O(n^2)。看上去本题已经很好的解决了，但如果把加法改为乘法呢？还是$a[i,j]=5$和$k=5$的情况，需要检查的t有1，3，5，7或9五个，而不是一个。在这样的情况下，是否仍然可以避免内层循环呢？答案是肯定的，不过要稍微调整一下算法。

不妨把刚才的程序称为“综合法”，即按照一定的顺序依次计算每个状态的值。如果一个状态需要用k状态来计算（比如在刚才的例子中，一个状态需要$t=1,3,5,7,9$五个状态计算），称这个状态的决策量为k。综合法的时间复杂度为O(状态个数×平均每个状态的决策数)，或O(总决策数)。对于特殊的问题，综合法并不是最好的。因为它会检查很多状态。更新法写起来更加简单，且时间效率更高：

```
for(i = n; i >= 1; i--)
  for(j = 1; j <= i; j++)
    for(k = 0; k <= 9; k++) if(d[i][j][k])
      d[i-1][j][(k*a[i-1][j])%10] =
      d[i-1][j-1][(k*a[i-1][j-1])%10] = true;
```

虽然一个状态需要用很多状态计算，但每个状态最多只能更新两个状态。更新法就是利用这一点避免无谓的判断。更新法同时适用于取max或min的情况，但不适合于两个状态需要做运算的情况。

小测验

1. 什么是后效性？数字三角形问题II为什么有后效性？如何解决这一问题？
2. 什么是最优子结构？数字三角形问题III为什么不具备最优子结构？如何解决这一问题？
3. 什么是更新法？它有什么特点和优势？它的适用范围是怎样的？

4.3.3 最优矩阵链乘

本节利用上节介绍的基本方法分析两个经典问题。

最优矩阵链乘 一个n×m矩阵由n行m列共n×m个数排列而成。两个矩阵A和B可以相乘当且仅当A的列数等于B的行数。一个n×m的矩阵乘以一个m×p的矩阵等于一个n×p的矩阵，运算量为m×n×p。矩阵乘法不满足分配律，但满足结合律，因此A×B×C可以按顺序(A×B)×C进行也可以按A×(B×C)来进行。假设A、B、C分别是2×3，3×4，4×5的，则(A×B)×C的运算量为2×3×4+2×4×5=64，A×(B×C)的运算量为3×4×5+2×3×5=90。显然第一种顺序节省运算量。给出n个矩阵组成的序列，设计一种方法把它们乘起来，使得总的运算量尽量小。假设第i个矩阵A_i是$p_{i-1} \times p_i$的。

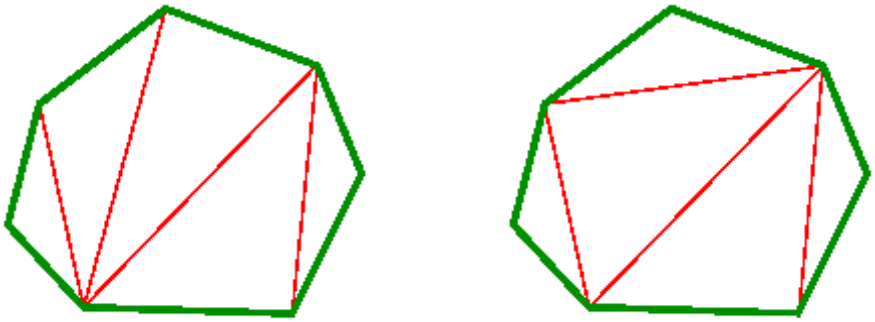
和数字三角形类似，我们让d[i,j]表示从第i个矩阵乘到第j个矩阵所需要的最小代价。显然一共需要$j-i$次乘法。这些乘法中一定有一个“最后一次乘法”A× B，那么A一定是从i个矩阵乘到第k个矩阵的结果，而B是$k+1$个矩阵乘到第j个矩阵的结果。问题满足无后效性（两边乘法互不影响）和最优子结构（代价之间是加法运算），因此可以列出状态转移方程为：$d[i,j] = max\{d[i,k] + d[k,j] + p_{i-1} \times p_k \times p_j\}$，边界为$d[i,i] = 0$。这个方程和数字三角形问题有所不同：它不能按照$i$递减的顺序递推，而应该按照$j-i$递增的顺序递推，设它为$l$。

解的重建 这样的递推法只能直接得到最优值，如果需要具体的加括号方法还需要另外的工作，即在递推的同时记录最优决策。让s[i,j]表示让d[i,j]取到最大值的决策k，那么从s[1,n]开始逐渐按照最优决策分配，最终可以打印出完整的方案。用括号表示运算顺序，程序如下，时间复杂度为O(n)。

```
for(i = 1; i <= n; i++) d[i][i] = 0;
for(l = 2; l <= n; l++)
  for(i = 1; i <= n - l + 1)
  {
    j = i + l - 1;
    d[i][j] = infinity;
    for(k = i; k <= j-1; k++) // enumerate decision k
    {
        v = d[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
        if(v < d[i][j])
            { d[i][j] = v; s[i][j] = k; }
    }
}

void print(int i, int j)
{
    if(i == j) printf("A[%d]", i);
    else{
        printf("(");
        print(i, s[i][j]);
```

```
        print(s[i,j]+1, j);
        printf(")"); // print recursively
    }
}
```

最优三角剖分 给一个有n个顶点的凸多边形，有很多方法进行**三角剖分(polygon triangulation)**。给每个三角形规定一个权函数$f(i,j,k)$（比如三角形的周长或者三顶点的权和），求让所有三角形权和最大的方案。



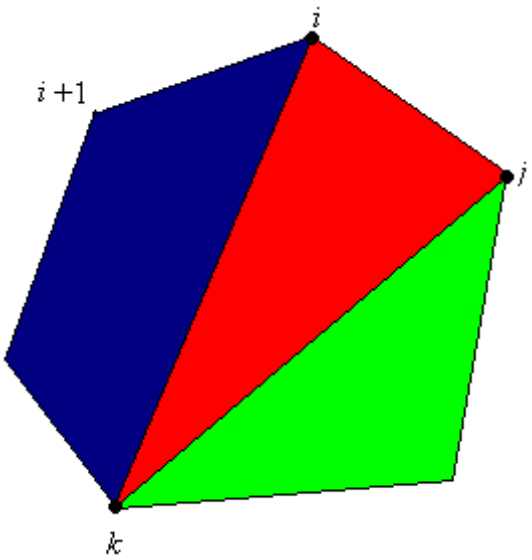
本题和最优矩阵链乘问题十分相似，但是有一个显著不同：链乘的结果反映了决策过程，而剖分的结果不反映决策过程。举个例子，在链乘问题中，方案$((A_1A_2)(A_3(A_4A_5)))$只能是先把序列分成A_1A_2和$A_3A_4A_5$两部分，而对于上图的两个剖分，“第一刀”可以是任何一条对角线。在这样的情况下，我们有必要把决策的顺序规范化。如果允许随意切割，显然任意“半成品”多边形的各个顶点可以是原多边形中随意选取的，很难简洁的定义成状态；然而用稍后介绍的方法，任意时候的子多边形的顶点是原多边形的连续子序列。

定义d[i,j]为从顶点i开始到顶点j所构成的子多边形的最大三角剖分权和，则边$i-j$在此多边形内一定恰好属于一个三角形ijk。枚举k的位置形成灰色三角形并把多边形分割为两部分。注意到两个多边形的顶点序列仍是连续的，因此可以递归求解。状态转移方程为$d[i,j]=max\{d[i,k]+d[k,j]+w(i,j,k)\}$，时间复杂度为$O(n^3)$。原问题的解为d[1,1]，第一次枚举决策$k$后得到的是退化多边形$1-k-1$，权为0。另外这里$i$可以大于$j$，在这种情况下$k$需要枚举$i+1,i+2,...n,1,2,...j-1$。

小结 本节介绍的方法可以称为“分割法”，它类似于分治，但由于划分点不固定，需要枚举决策。

小测验

1. 叙述矩阵链乘问题的状态定义、状态转移方程和它的递推顺序、边界条



件。

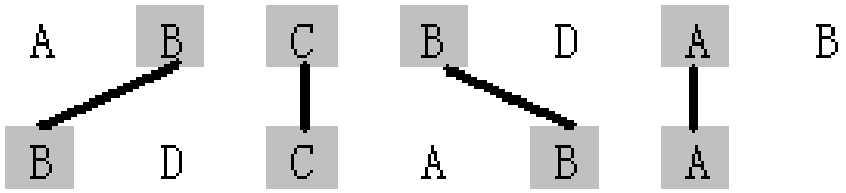
2. 在最优三角剖分问题中，如何记录和输出最优三角剖分方案？

3. 在最优三角剖分问题中，如果可以剖分的最小单位可以是三角形也可以是四边形，动态规划方法仍然适用吗？

4.3.4 最长公共子序列问题

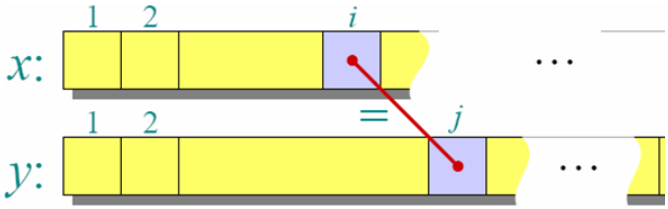
本节介绍最长公共子序列问题。

最长公共子序列问题。给出两个序列$x[1..m]$和$y[1..n]$，找出二者的一个最长公共子序列。一个子序列是原序列删除一些元素后得到的，剩下的元素必须保持相对顺序。例如$x = ABCBDAB$，$y = BDCABA$，则一个最长公共子序列为$LCS(x, y) = BCBA$。这里LCS借用函数记号来表示一个最长公共子序列。



用x[1..i]和y[1..j]分别表示x的前i个字母组成的前缀和y的前j个字母组成的前缀，记d[i,j]表示$LCS(x[1..i], y[1..j])$的长度。应该如何递推呢？

第一种方法是枚举LCS的最后一个字符。设它是x的第a个字符，也是y的第b个



字符，因此剩下的问题就是求x[1..a-1]和y[1..b-1]的LCS了。换句话说，状态转移方程为：$d[i,j] = max\{d[a-1,b-1]+1\}$，枚举条件是$x[a] = y[b]$，$i \le a, j \le b$。这样的方程是对的，但是状态有$mn$个，每个状态的决策最坏也是mn的，时间复杂度为O(m^2n^2)。

一个更好的方法是增量法，即枚举最后一个字符是否出现在LCS中：如果$x[i]$和$y[j]$相等，则$d[i,j] = d[i-1,j-1]$，否则$x[i]$和$y[j]$不可能同时出现在LCS中，因此为$max\{d[i-1,j], d[i,j-1]\}$，程序如下：

```
for(i = 1; i <= m; i++) d[i][0] = 0;
for(j = 1; j <= n; j++) d[0][j] = 0;
for(i = 1; i <= m; i++)
  for(j = 1; j <= n; j++){
    if(x[i] == y[j]){ d[i][j] = d[i-1][j-1] + 1; s[i][j] = 1; }
    else if(d[i-1][j] >= d[i][j-1]){ d[i][j] = d[i-1][j]; s[i][j] = 2;}
    else{ d[i][j] = d[i][j-1]; s[i][j] = 3; }
}
```

和矩阵链乘一样，可以用s[i,j]来重建解，如下图。

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

规划方向 细心的读者也许注意到了，我们完全可以定义一个“相反”的状态：记d[i,j]为$LCS(x[i..m], y[j..n])$的长度，则递推方程完全类似，只是从下到上递推。数字三角形也有两种状态定义方法。它们的区别在于规划方向。在本题中，两种方向难度差不多，但是在某些特殊的题目里，其中一种比另一种更好，希望读者注意。

关于决策矩阵s 读者很容易注意到s数组其实是可以省略的，因为根据d数组可以在O(1)的时间判断出最优决策究竟是哪一个。矩阵链乘问题中，需要O(n)的时间比较才能知道哪个决策是最优的，因此仍然需要s数组。

滚动数组 有一种方法可以在渐进意义下降低空间复杂度。注意到每一行的值只取决于上一行的值，因此每次只保留两行即可，如下：

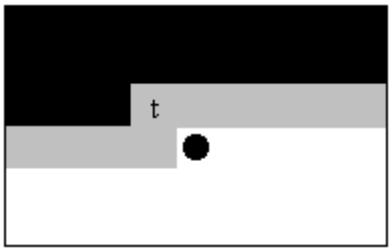
```
for(j = 0; j <= n; j++) d[0][j] = 0; // line 1
for(i = 1; i <= m; i++){
  for(j = 1; j <= n; j++){
    if(x[i] == y[j]) d[1][j] = d[0][j-1] + 1;
    else if(d[0][j] >= d[1][j-1]) d[1][j] = d[0][j];
    else d[1][j] = d[0][j-1];
  }
  // get ready for the next line
  for(j = 0; j <= n; j++) d[0][j] = d[1][j];
}
```

其中当前行为d[1]，上一行为d[0]。最后答案在d[0][n]。这样做空间复杂度降低了，但是时间上有浪费，即行拷贝操作。其实没有必要做行拷贝，像循环队列一样用d[cur]表示当前行，则d[1-cur]为上一行。最后答案在d[1-cur][n]。

```
for(j = 0; j <= n; j++) d[0][j] = 0; // first line
cur = 1;
for(i = 1; i <= m; i++){
  for(j = 1; j <= n; j++){
    if(x[i] == y[j]) d[cur][j] = d[1-cur][j-1] + 1;
    else if(d[1-cur][j] >= d[cur][j-1]) d[cur][j] = d[1-cur][j];
    else d[cur][j] = d[cur][j-1];
  }
  // alter line
  cur = 1-cur;
}
```

事实上，可以进一步把空间需求降低为一行，由于每个格子只被它下一行、右下方和右一列的格子用到，因为在任意时刻需要用到的元素最多$n+1$个，如下图。

黑色部分是已经计算完毕且不需要使用的，灰色部分是已经计算完毕且还会用到的，黑色小球为当前正在计算的位置。用t表示当前元素左上方的值，设当前位置为(i,



j)，则当$k < j$时d[k]表示原来定义中的d[i,k]，而$k \geq j$时表示原来定义中的d[i-1,k]，t是原来定义中的d[i-1,j-1]。递推过程如下：

```
for(j = 0; j <= n; j++) d[j] = 0; //line 1
for(i = 1; i <= m; i++){
  t = 0; // new line, d[i, 0] = 0
  for(j = 1; j <= n; j++){
    if(x[i] == y[j]) v = t + 1;
    else v = max(d[j], d[j-1]);
    t = d[j];  // when updating d[i], backup to t
    d[j] = v;
  }
}
```

从刚才的几个例子可以看出，应用动态规划解题往往分为两个步骤，一是分析问题，设计状态和递推式，二是根据递推式进行计算，加上时空优化。需要注意的是，在降低空间复杂度以后，解的构造变得困难了。如果坚持只能使用O(m)的空间，那么需要重复进行m次动态规划，每次让最优解往上走一行，时间复杂度为O(m^2n)。

小结 本节介绍的方法可以称为“增量法”，它重点考虑删除一个元素后剩下的子问题，而不急着构造出解。这个方法非常类似于第4章中介绍的最少转弯问题，直接构图时边数非常多，拆点简化了问题；本问题直接枚举LCS内的元素情况非常多，拆决策简化了问题。从前面的d值图来看，原先的动态规划算法实际上是允许两个“匹配点”（满足x[i]=y[j]的点）间的任意跳转，情况非常多，而增量法通过限制决策的方式既没有丢失解，还让决策总数少了很多。有时把这种方法称为“细化状态转移”。

小测验

1. 什么是增量法？为什么它隐含了状态转移的细化？
2. 如果有三个串，应该如何修改算法？
3. 什么是滚动数组？它的原理是什么？好处和局限性分别是什么？

```
d[1] = 1;
S={a[1], d[1]}
for(i = 2; i ≤ n; i++){
d_max=getmax(S,a[i]);
d[i] = d_max + 1;
update(S,a[i],d[i])
}
```

4.3.5 其他经典问题及其时间优化

本节讨论几个经典问题和它们的优化。

最长上升子序列问题(Longest Increasing Sequence, LIS) 给一个序列，求它的一个递增子序列，使它的元素尽量多，如下图。

设d[i]为以i结尾的最长上升子序列的长度，那么它的前一个元素是什么呢？设它为j，则$d[i] = max\{d[j]\} + 1$，其中枚举条件是$j < i$且$a[j] < a[i]$。状态有O(n)个，决策O(n)个，时间复杂度为O(n^2)。下面考虑如何把它优化到O($n \log n$)。

把同一个i对应的d[i]和a[i]看成一个二元组(d, a)。在计算d[i]时，考虑所有满足$j < i$的二元组(d, a)。显然它们的i是无关紧要的，只需要找出其中$a[j] < a[i]$的最大d[j]即可。换句话说，程序看起来应该是这样的：

其中getmax(S,a[i])表示在集合S中所有a值小于a[i]的二元组中寻找d的最大值。update(S,a[i],d[i])表示用二元组$a[i], d[i]$更新集合S。

现在问题的关键是实现集合S。考虑二元组(5, 4)和(5, 3)。它们d值相同但a值不同。对于后续决策来说，(5, 4)是合法时(5, 3)一定合法，而且(5, 4)和(5, 3)一样好。因此我们规定：只保留(5, 3)！换句话说：d值相同的a只保留一个。用g[i]表示d值为i的最小a，那么一定有$g[1] \le g[2] \le ... \le g[n]$（想一想，为什么？）。不存在的$d$值对应的$g$设为正无穷。

d[i]的计算 可以使用二分查找得到大于等于a[i]的第一个数j，则$d[i] = j$（本来是找不大于a[i]的最后一个数j，则$d[i] = j + 1$，但这样转化后更方便）。

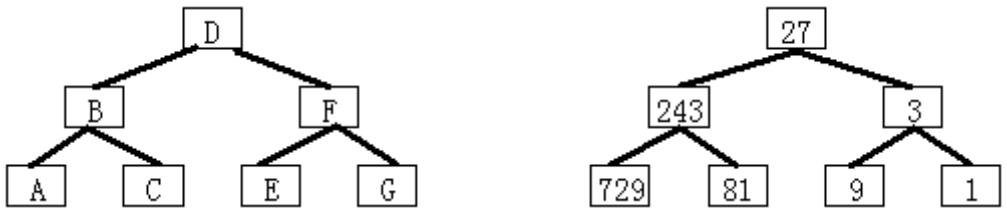
g的更新 由于$g[j] > a[i]$，且i的d值为j，因此需要更新$g[j] = a[i]$。

程序如下：

```
fill(g, g + n, infinity);
for(int i = 0; i < n; i++){
  int j = lower_bound(g, g + n, a[i]) - g;
  d[i] = j + 1;
  g[j] = a[i];
}
```

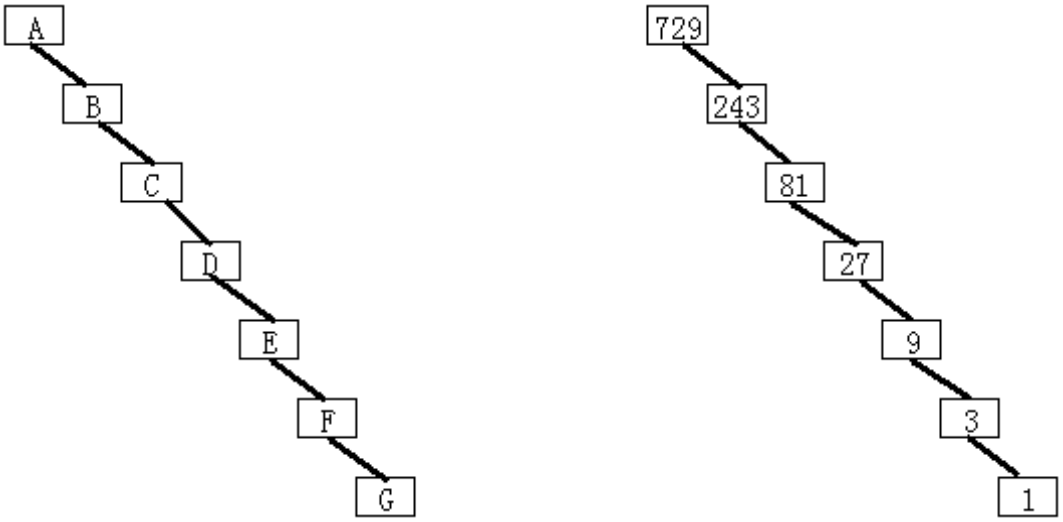
程序使用了第3章介绍的lower_bound函数，时间复杂度为O(nlogn)。

最优排序二叉树(Optimum Binary Search Treem OBST) 需要把n个符号做成符号表。虽然平衡二叉树的高度平衡，但是如果各个符号的频率不同的话，它并不一定是最好的选择。如果有7个符号ABCDEFG（按从小到大顺序排列），频率分别为729，243，81，27，9，3，1，则平衡二叉树为：



这棵平衡二叉树的检索总次数（即所有关键码的频率与深度的乘积相加的结果）为$27 \times 1 + (243 + 3) \times 2 + (729 + 81 + 9 + 1) \times 3 = 2979$。

而下面这棵很斜的树的总检索次数为$729 \times 1 + 243 \times 2 + 81 \times 3 + 27 \times 4 + 9 \times 5 + 3 \times 6 + 1 \times 7 = 1636$，比平衡二叉树好很多。



按照关键码大小给出n个符号的频率f_i，构造出总检索次数最小的排序二叉树。

有了前面的介绍，这个问题不难解决。根据排序二叉树的递归定义，可以先选根，再递归建立左右子树。记d[i,j]为第$i \sim j$的符号所建立的排序二叉树的最小检索次数，如果选根为k，总检索次数应该如何计算？

树根k只需要检索一次，累加上f_k。左子树单独作为一棵树时的总检索次数为d[i,k-1]，但是作为k的子树后所有结点的深度增加了1，因此实际上需要累加f_i+f_{i+1}+...+f_{k-1}。右子树类似。因此，若记$w[i,j] = f_i + f_{i+1} + ... + f_j$，状态转移方程为

$$d[i,j] = max\{d[i,k-1] + d[k+1,j]\} + w[i,j]$$

状态有O(n^2)个，每个决策有O(n)，因此时间复杂度为O(n^3)。

下面把它优化到O(n^2)。

四边形不等式(Monge condition / Quadrangle inequality) 如果对于$i \le i' < j \le j'$总有$w[i,j] + w[i',j'] \le w[i',j] + w[i,j']$，称$w$满足四边形不等式，或$w$满足凸性。

区间包含格上的单调性 如果对于$i \le i' < j \le j'$总有$w[i',j] \le w[i,j']$，称w满足单调性。

在本题中，显然$w[i,j] + w[i+1,j+1] = w[i+1,j] + w[i,j+1]$，因此满足四边形不等式。

定理*(F.Yao)*: 若w满足四边形不等式，则d也满足四边形不等式，即

$$d[i,j] + d[i',j'] \le d[i',j] + d[i,j'],\ i \le i' \le j \le j'$$

证明略，读者可以在配套ppt中找到完整证明，主要是分情况讨论和代数变形。

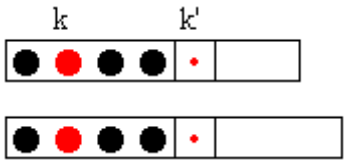
更进一步地，d的凸性可以推出决策单调性。记k[i,j]为让d[i,j]取最小值的决策，有

定理*(F.Yao)*: $k[i,j] \le k[i,j+1] \le k[i+1,j+1],\ i \le j$，即$k$在同行同列都是递增的。

证明：由对称性，只需证明$k[i,j] \le k[i,j+1]$。记$d_k[i,j] = d[i,k-1] + d[k,j] + w[i,j]$，则只需证明对所有的$i < k \le k' \le j$，有$d_{k'}[i,j] \le d_k[i,j]$蕴涵$d_{k'}[i,j+1] \le d_k[i,j+1]$，即：区间加长一个单位后，以前较好的决策现在仍然好。

事实上，可以证明一个更强的结论：$d_k[i,j] - d_{k'}[i,j] \le d_k[i,j+1] - d_{k'}[i,j+1] (i < k \le k' \le j)$，因为当k'在[i,j]更优时，左边≥0，由不等式知右边≥0，因此k'仍更优。

如上图，设k'是[i,j]的最优值，则对于它左边的任意k，k'在[i,j]上更优意味着k'在[i,j+1]上仍最优，因此k'左边的任意k在[i,j+1]上仍然不是最大值，即$k[i,j+1] \ge k[i,j]$。



欲证$d_k[i,j]-d_{k'}[i,j]\le d_k[i,j+1]-d_{k'}[i,j+1](i<k\le k'\le j)$，移项得：$d_k[i,j]+d_{k'}[i,j+1]\le d_k[i,j+1]+d_{k'}[i,j]$。按定义展开，两边消去$w[i,j]+w[i,j+1]+d[i,k-1]+d[i,k'-1]$得：

$$d[k,j]+d[k',j+1]\le d[k,j+1]+d[k',j]$$

这就是d的凸性。

有了决策单调性，可以稍微改造一下程序，决策从$k[i,j-1]$枚举到$k[i+1,j]$即可。这样做时间复杂度降低了吗？当$L=j-1$固定时，

$d[1,L+1]$的决策是$k[1,L]\sim k[2,L+1]$
$d[2,L+2]$的决策是$k[2,L+1]\sim k[3,L+2]$
$d[3,L+3]$的决策是$k[3,L+2]\sim k[4,L+3]$
$d[4,L+4]$的决策是$k[4,L+3]\sim k[5,L+4]$
...

合并起来，当L固定时总决策为$k[1,L]\sim k[n-L+1,n]$，共$\mathrm{O}(n)$个。由于L有$\mathrm{O}(n)$，因此总时间复杂度降为$\mathrm{O}(n^2)$。

小测验

1. 在LIS问题中，如果允许子序列中相邻数相同，算法还是正确的吗？
2. 什么是四边形不等式？什么是区间包含格上的单调性？它们有什么用处？
3. 什么是决策的单调性？可以从实验上验证或猜想决策单调性吗？

4.4 路径寻找问题

学完计算理论之后，您是否有点沮丧？看起来，有很多问题是难以解决的，甚至有一些（在图灵机模型下）是无法解决的。其中有不少问题是那么的自然和普遍，以

至于你无法说服自己“放弃吧，它是NP完全问题。不是我的错，确实没有人能找到多项式算法”。理论上再困难的问题，只要有需要，我们就必须寻找解决它的方法，而且它的各方面表现还应该尽量好。这就是问题求解所应该具备的精神：不管理论分析后得出多么不利的结论，我们还是应该努力去解决需要解决的问题。

本章的主题是人工智能搜索，它是对付NP完全问题的有力武器。搜索是一种传统而通用的问题求解方法。因为其通用性，这个方法相当难以掌握，需要具体问题具体分析。人工智能搜索包含的内容很多，本章只是一个导引，在开拓视野的同时教会读者一些最基本的搜索算法。

4.4.1 引例和基本概念

在棋盘上放置8个皇后，使得她们互不攻击。每个皇后的攻击范围为同行同列和同对角线，如图1所示。

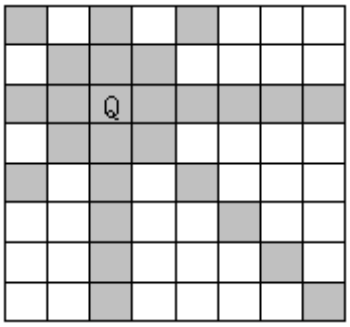


图 4.31: 皇后的攻击范围

图2是一个可行解。它是怎样得来的呢？它是“一个一个试出来”的。由于每行必须恰好有一个皇后，我们依次考虑每一行的每一个皇后。我们把行从上到下编号为0~7，列从左到右编号为0~7，这样，每一个当前摆放皇后的“情况”（不一定是解！）可以表示成一个向量，表示每行的皇后编号。例如图2表示为(0, 4, 7, 5, 2, 6, 1, 3)。如果向量中的元素有不确定的值，可以用“？”表示。由于这时候我们需要进一步确定这些“？”的值，称每个带“？”的向量为“状态”。例如(0, 1, ?, ?, ?, ?, ?, ?)就是一个状态。

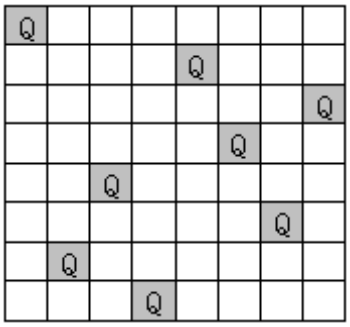


图 4.32: 一个可行解

可惜这个状态并不包含可行解，因为前两个皇后已经相互攻击了。这样的状态称为非法状态。

8皇后问题的情况比较复杂，我们不妨先考察只有4皇后的情形。最开始什么都不知道，状态为(?, ?, ?, ?)，称为**初始状态(initial state)** 。第一个皇后在哪一列呢？不知道，只能一个一个试。于是从初始状态可以**扩展(expand)** 出4个状态，即(0, ?, ?, ?), (1, ?, ?, ?), (2, ?, ?, ?)和(3, ?, ?, ?)。每个状态又继续扩展，直到求出解或者无法扩展为止。这里把完整的过程画出来，如下图。注意：图上并没有画出(0, 1, ?, ?)这样的非法状态。

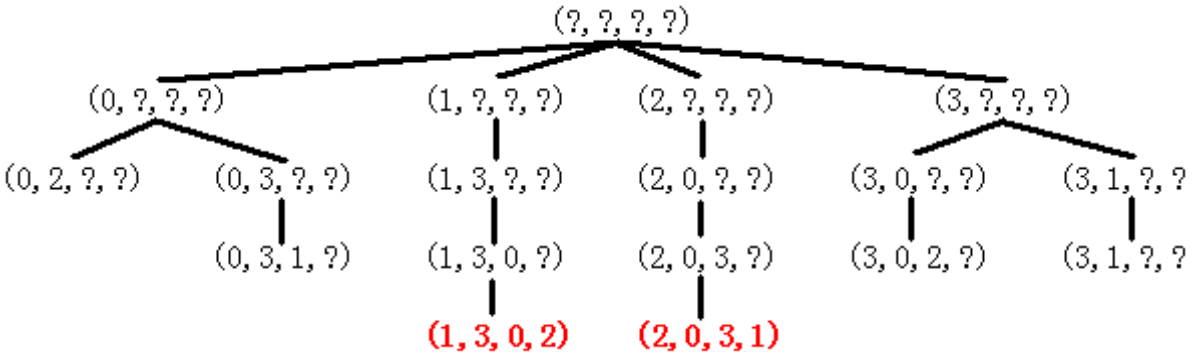


图 4.33: 4皇后问题的解答树

上图是一棵树，它标识从初始状态可以直接或间接扩展出的所有状态。它有一个名字叫“解答树”，每个结点的儿子是该结点能扩展出的所有结点。每个状态的深度恰好是它包含的问号个数。

只要某算法能完整的扩展出这棵树，它一定可以找到深度为4的所有结点，即问题的解。这样的算法有很多，回溯法就是其中的一个。

回溯法(backtracking) 回溯法按照深度优先的顺序遍历解答树。换句话说，它遍历图3的顺序是(?,?,?,?)–>(0,?,?,?)–> (0,2,?,?)–>(0,3,?,?)–>(0,3,1,?)–>(1,?,?,?)–>(1,3,?,?)–>...既然是深度优先遍历，回溯法可以很自然的写成递归的形式。回溯法有多种实现方式，这里暂时不展开讨论，读者明白程序大意即可。根据不同的题目类似，本章的后面几节将对回溯法和其他类似算法进行深入分析。

下面给出完整的程序，它将按顺序给出回溯法遍历到的所有状态并给出解的总数。为了方便判断，程序使用数组used[i][0]，used[i][1]和used[i][2]分别表示编号为i的列、主对角线和副对角线是否已经被攻击到。放置一个皇后需要把一些值设置为真，取走皇后（即回溯）时把这些值重新设置为假。主副对角线分别用y-x和x+y标识各条对角线，如下图：

0	1	2	3	4	5	6	7
-1	0	1	2	3	4	5	6
-2	-1	0	1	2	3	4	5
-3	-2	-1	0	1	2	3	4
-4	-3	-2	-1	0	1	2	3
-5	-4	-3	-2	-1	0	1	2
-6	-5	-4	-3	-2	-1	0	1
-7	-6	-5	-4	-3	-2	-1	0

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

(a)各个格子的y-x值(b)各个格子的x+y值

由于$y-x$的值范围为-7∼7，我们可以把它加上7调整到0∼14，这样，主副对角线的编号均为0∼14了。程序中的附加数组path保存当前路径的上结点，修改前面的常数n定义可以求解任意多皇后的问题（但时间效率不令人满意，读者可以一试）。

```
#include<stdio.h>
const int n = 4;

int tot = 0;
int path[n];
int used[n*2-1][3];
```

```
void search(int dep){
  int i;

  //输出状态值
  printf("(");
  for(i = 0; i < dep; i++)
    { if(i) printf(","); printf("%d", path[i]); }
       for(i = dep; i < n; i++)
          { if(i) printf(","); printf("?"); }
      printf(")\n");

      if(dep == n) tot++;  //递归边界
      else for(i = 0; i < n; i++) //枚举第 行皇后的列编号dep
         if(!used[i][0] && !used[i-dep+n-1][1] && !used[i+dep][2]){
            path[dep] = i;
            used[i][0] = used[i-dep+n-1][1] = used[i+dep][2] = 1;
            //放置皇后
            search(dep + 1);                    //递归遍历子树
            used[i][0] = used[i-dep+n-1][1] = used[i+dep][2] = 0;
            //取走皇后
          }
}

int main()
{
        search(0);
        printf("%d\n", tot);
        return 0;
}
```

如果只需要统计解的个数，可以删除注释“输出状态值”后面的四行。再次强调，本小节的目的在于介绍回溯法的原理，具体的程序实现技巧、变形和更多的应用将留到后面介绍。

如果尝试求解n比较大的情况，读者会发现上小节给出的程序非常慢，这是回溯法最容易遇到的问题。一般来说最容易想到[2]和实现的时间优化是剪枝。

剪枝(prune) 如果解答树中某个结点的后代不可能有解，则无需扩展此结点。这相当于在解答树中剪掉此结点下方的整棵子树，我们形象的把这种方法称为剪枝。剪枝的具体方法和问题的相关度非常大，需要具体问题具体分析。

当前状态不可能扩展出可行解时剪枝，可以进行**可行性剪枝** 。如果问题需要最优解，则即使当前状态可以扩展出可行解，只要它无法扩展出最优解，都可以进行**最优性剪枝** 。如果问题只需要求一组可行解，则即使当前状态可以扩展出可行解，只要可

[2] 理论性比较强的优化方法将在后面介绍

以证明在其他枝叶中也存在解，也可以进行**替代剪枝**，忽略当前状态。

小结 回溯法属于“通用解题法”，它的思想是用一种系统的方法(systematic approach)枚举所有情况。如果情况是有限的，算法一定可以得到想要的结果，但可能花费很长的时间。不同的题目在使用回溯法时的细节可以有很大不同，而同一个题目也可以有不同的回溯方法。因此本节和前两章最大的不同之处是：本节强调**框架(framework)** 而不是具体的可重用法**模块(module)**，从而更加要求读者有“具体问题具体分析”的精神和素质。

需要特别强调的是：回溯法的效率是值得注意的：通常情况，它是指数级别的。回溯法的优势在于空间节省，调试方便，是最容易掌握的状态空间搜索算法，应用也最广泛——甚至在有更好算法的同时会被滥用。它的正确性往往是显然的，但一般来说是迫不得已才会使用的算法。

小测验

1. 什么是回溯法？它的缺点和优点是什么？
2. 什么是解答树？回溯法用怎样的顺序遍历解答树？
3. 证明n皇后问题的状态空间搜索时间复杂度为O(n!)。

4.4.2 状态空间

在上一节中，我们介绍了回溯法，并且从感性上认识了什么叫“用系统的方法一个一个试”，但并没有从理性上深入进行学习。本节学习路径寻找问题，并从理论上把解答树扩展为状态空间，回溯法扩展为状态空间搜索。

路径寻找问题的通常形式为“解谜题”，往往被制作成各种玩具和游戏，集趣味性和挑战性于一身。

15数码问题 1878年，“美国最伟大的谜题专家”Sam Loyd发明了15数码谜题。这个谜题由一个正方形盒子和15个编号为1，2，...，15的正方形滑块组成，每个滑块的边长是盒子边长的1/4，因此可以把它们拼成图8.1中左图的形式，留下一个空格。每次可以把任何一个与空格相邻的滑块移动到空格中（因此在左图中可以把滑块12和15移动到空格中），而此滑块原来的位置变成新的空格。图8.1中左图和右图非常接近，那么是否可以从左图的状态出发，在若干步之后变成右图的状态呢？Sam Loyd“悬赏”1000美元奖给第一个解决这个问题的人。

游戏规则很容易理解，因此这个问题引起了很多不同职业不同阶层的的人的兴

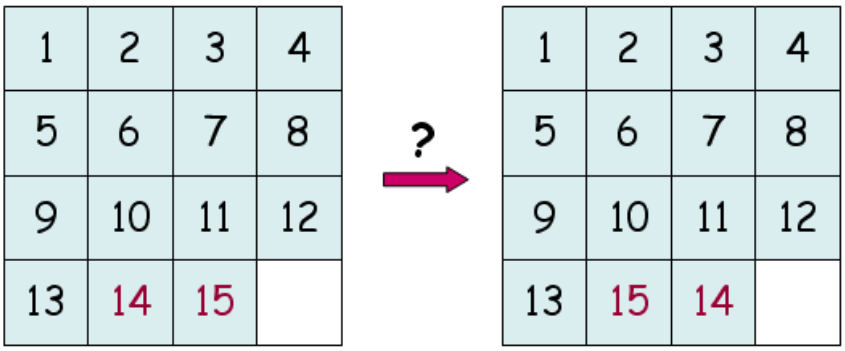


图 4.34: 15数码问题

趣，可惜始终没有一个人拿到这1000美元。事实上，这个特殊的15数码问题是无解的（在下一章中我们将使用数学方法证明这个结论），但在很多情况下，是可以从一个布局A转化成另一个布局B的。本章贯穿这样的一般性**15数码问题(15-puzzle)**：给出任意两个布局A和B，用最少的步数把A转化为B。

为此，我们引入**路径寻找问题(path-finding problem)** 。它包含五个要素：状态定义、初始状态、后继函数、目标测试和路径耗散。15数码问题的状态被定义为15个滑块的位置（15个滑块位置确定后空格的位置也就定了），初始和目标状态取决于问题实例，后继函数产生上、下、左、右中合法移动所导致的下一状态（可以有2~4的后继状态），目标测试是简单的把状态和目标状态进行比较，而每一步的路径耗散始终为1。

读者一定想到了回溯法。个状态可以扩展出最多4个子状态，这样得到了一棵解答树。遍历解答树即可得到问题的解。这个算法没错，但是效率太低。仔细观察解答树（画解答树的工作留给读者），我们会发现解答树有重复状态！

由于数字移动相当于空格反方向移动，我们用序列(d_1, d_2, ..., d_k)表示从初始状态开始把空格按方向d_1, d_2, ..., d_k移动之后形成的布局，则(L, R)和(R, L)以及(L, U, D, R)都是初始状态，如果在解答树中完全扩展所有状态，那我们就犯了一个和动态规划“直接递归法”相同的错误：没有利用到“重叠子问题”。

图8.1.1是15数码问题的简化版本：8数码问题的解答树片段，存在重复结点。如果重复结点可以重复访问，那么解答树将是无穷的！事实上，所有可能的状态是有穷的。在8数码问题中，如果空格看成数字9则一个布局对应于1~9的一个排列，而一共只有9!=362880种排列。我们有理由把状态所构成的结构看作是一个有362880个结点的图，如图8.1.1所示，称为**状态空间(state space)** 。

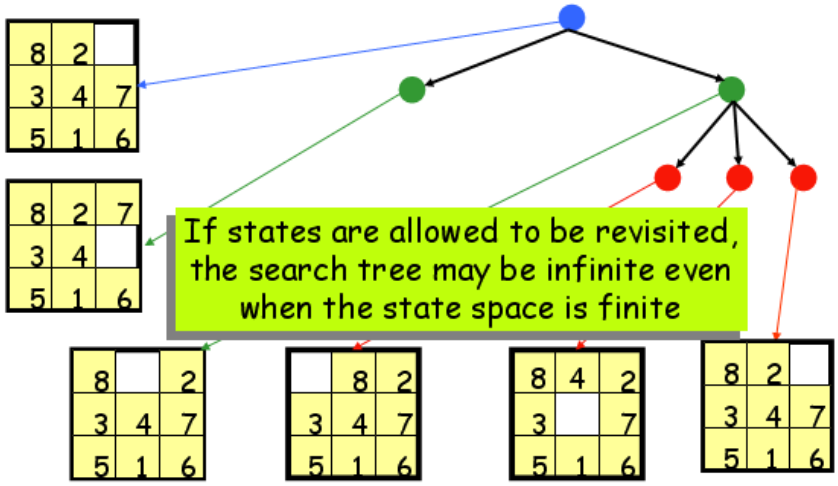


图 4.35: 8数码问题的解答树片段

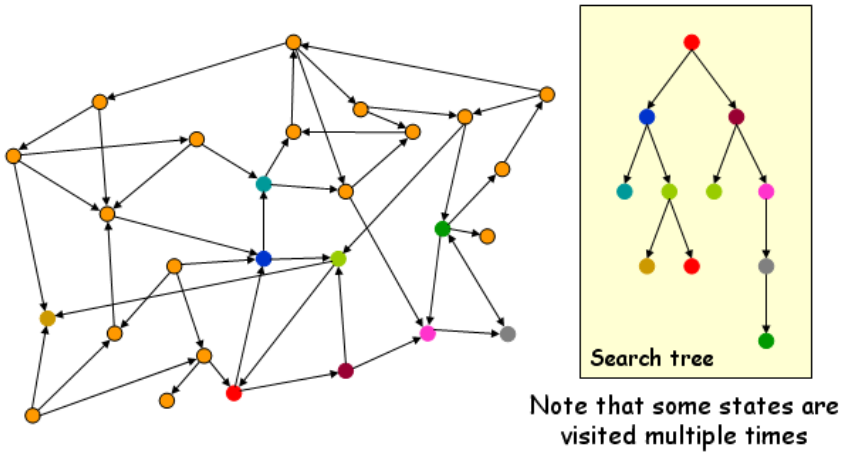


图 4.36: 状态空间

在学习各种不同的搜索策略之前，让我们看看几个重要的概念。

搜索结点 解答树中的结点称为搜索结点，它不仅记录了状态信息，还记录了和搜索有关的附加信息，例如父亲结点、儿子结点和弧开销等。图8.1.1描述了一个搜索结点应当记录哪些信息。

结点扩展 解答树是通过结点扩展一步一步产生的。结点扩展需要后继函数(successor function)，扩展结点N的过程就是先计算SUCCESSOR(N)，然后给函数返回的每一个状态建立一个结点，即把这些子状态包装一下，填入Action, Depth, Path-Cost, Expanded等信息。

搜索边界 在搜索的任何时候，没有被扩展过的结点集合称为搜索边界(fringe)。下

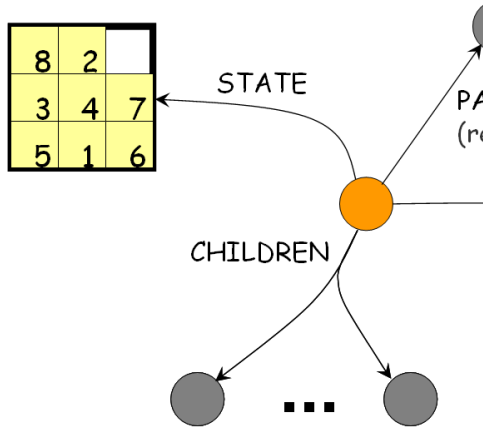


图 4.37: 搜索结点

图就是一个搜索边界。搜索边界上的结点不一定处于同一个层次，只要没有被扩展过的结点都是边界的一部分。

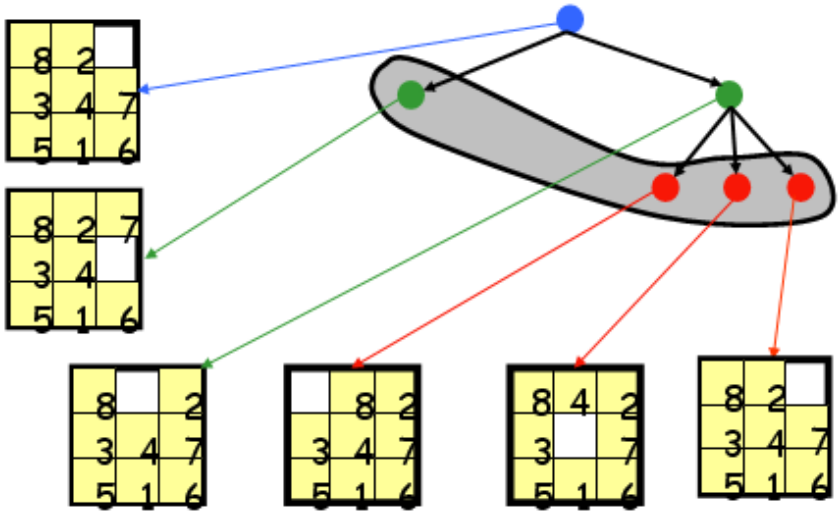


图 4.38: 搜索边界

搜索策略(search strategy) 每次需要选择搜索边界上的一个结点进行扩展，所以搜索边界应被实现为一个优先队列，支持insert（插入结点）和remove（取下一个扩展结点）操作。这个优先队列决定了搜索边界将按怎样的顺序进行扩展，因此把它称搜

索策略。

下面给出一般搜索算法框架。为了使用这个算法，需要实现设定初始状态initial-state，目标状态判断函数GOAL?（因为目标状态可能有多个），搜索边界的优先队列实现（INSERT，REMOVE，EMPTY函数）以及状态后继函数SUCCESSORS。

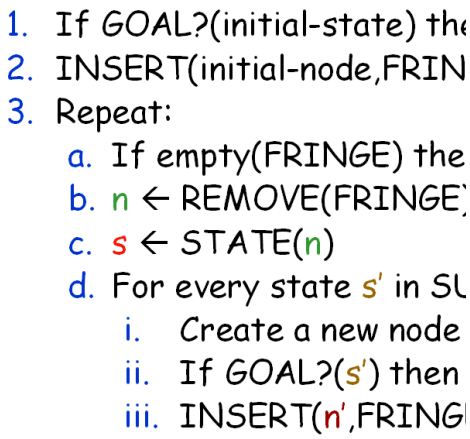


图 4.39: 搜索算法框架

需要特别注意的是，只要发现第一个目标结点，算法将立刻返回。

性能度量 前面说过，搜索策略可以有很多种，如何评价它们的优劣程度？一般说来可以有以下三个指标：

完全性(completeness) 如果有解时某算法一定能找到解，称此算法为完全的。

最优性(optimality) 如果有解时某算法一定范围路径总开销最小的解，称此算法为最优的（想一想，还可以有哪些最优性度量？）

复杂性(complexity) 算法需要多大空间和多大运算量？这相当于前面介绍的算法分析结果。这里的规模一般取三个：每个状态的后继状态数目，即**分支因子（branching factor）** b；**最小解深度（goal depth）** d；**最大叶子深度（maximal leaf depth）** m。

但由于这三个参数往往不能精确的和原始输入参数联系起来，所以对搜索算法的分析常常是非常不精确的，更多的只有理论意义，确保上限而不是实际开销。

盲目搜索和启发式搜索 搜索策略可以分为两大类。盲目搜索算法不关心状态的定义，而启发式搜索通过计算状态的某些指标优先选择比较“有希望”的结点先扩展。如

图8.1.2，盲目搜索认为N1和N2没啥区别，而启发式搜索认为N2比N1更“有希望”，因此将在搜索策略中考虑优先扩展N2（具体的扩展时机后文将详细叙述）。

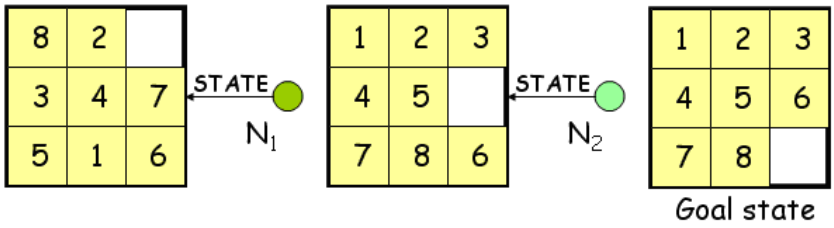


图 4.40: 盲目搜索和启发式搜索

小测验

1. 什么是路径寻找问题？它包含哪几个要素？什么是状态空间？15数码的状态空间有多大？8数码问题呢？
2. 一般的搜索算法框架是怎样的？它需要事先设定哪些东西？其中哪些是和问题相关的，哪些是和问题无相关的（只取决于算法）？
3. 应怎样评价一般的搜索算法？在考虑时空开销时，常用哪几个参数？其中哪个参数只有当状态空间有限时才有意义？为什么说对搜索算法的分析往往是不精确的？

4.4.3 盲目搜索

上一节介绍了盲目搜索算法的基本思想，本节学习几个最常见的盲目搜索算法。

广度优先搜索(Breadth-first search)这里的广度优先搜索可以完全的类比为图的BFS遍历，这里不详细叙述。需要注意的是每次新扩展出来的结点添加到搜索边界的末尾。换句话说，边界是一个队列。

显然，广度优先搜索是完全的，如果每边的开销是1，则它是最优的。生成的结点树不超过$1+b+b^2+...+b^d=(b^{d+1}\text{-}1)/(b\text{-}1)=O(b^d)$，因此时空开销均为$O(b^d)$，如图8.1.2。

如果解根本不存在，则上表的数字是可以达到的（假设不用数学方法判断无解）。

双向广度优先搜索这是广度优先搜索的一个变形，它有两个搜索边界，交替进行搜索（每次先扩展结点数少的一边）。如果两棵树的分支因子均为b，则生成的结点数为$O(b^{d/2})<<O(bd)$。

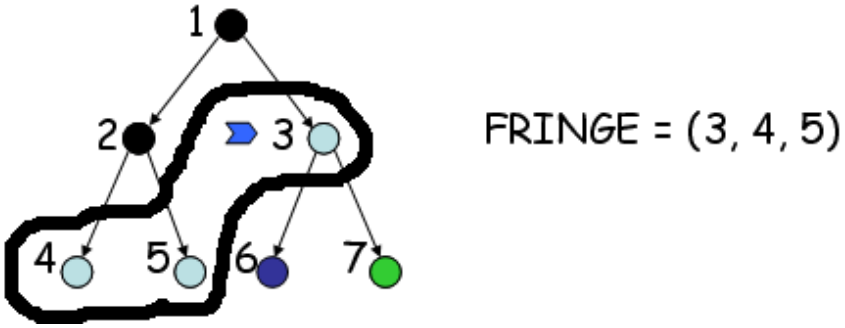


图 4.41: 广度优先搜索

d	# Nodes	Time
2	111	.01 msec
4	11,111	1 msec
6	~10^6	1 sec
8	~10^8	100 sec
10	~10^{10}	2.8 hour
12	~10^{12}	11.6 day
14	~10^{14}	3.2 year

Assumptions: b = 10; 1,000,00

图 4.42: 时空开销表

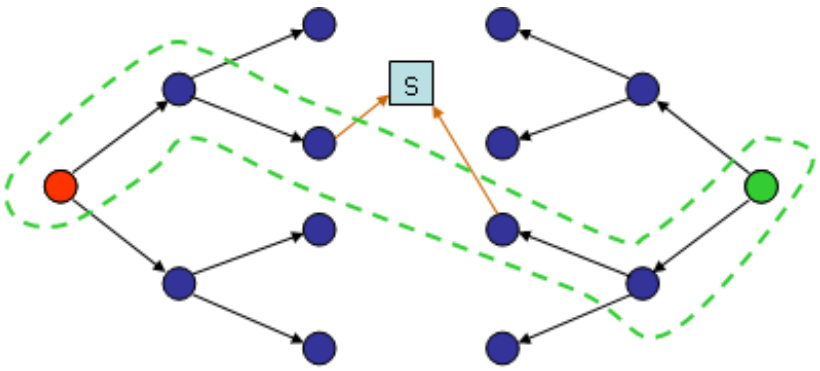


图 4.43: 双向广度优先搜索

深度优先搜索这里的深度优先搜索和前面的讲的回溯法有差异。这里的搜索边界并不是只有一个结点。例如在图8.1.2中，边界为{3,5}。在回溯法中，结点1一旦扩展出结点2就立刻遍历它，而在深度优先搜索中，必须先扩展出所有儿子2和3，然后扩

展2。每次扩展出的结点被插入到边界队列的首部。换句话说，边界是一个栈。

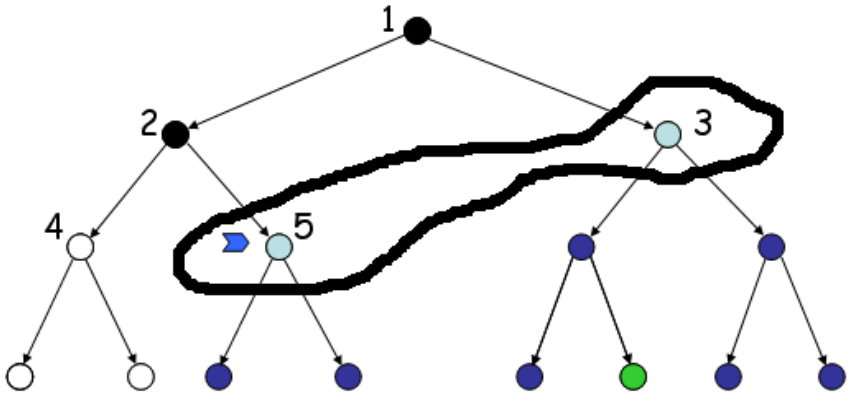


图 4.44: 深度优先搜索

深度优先搜索的理论分析结果不太令人满意。首先，算法只有当状态空间有限时才能保证是完全的。其次，它并不是最优的（想一想，为什么）。而完全性和最优性都可以被广度优先搜索保证。当状态空间有限时，扩展的结点数为$1+b+b^2+...+b^m$=O(b^m)，这也是时间复杂度；而空间复杂度仅为O(bm)。对于广度优先搜索的时间复杂度O(b^d)和空间复杂度O(b^d)，读者可以立刻得出结论：深度优先搜索的最大优势在于空间开销小。

深度优先搜索的一个变形为深度有限搜索(depth-limited search)，它有一个截断深度k，不扩展所有深度超过k的结点。显然，这个方法并不是完全的，也不是最优的。

迭代加深搜索把深度有限搜索中的k设置为从小到大递增的，则得到的算法称为迭代加深搜索(iterative deepening search, IDS)。图8.1.3是深度限制为1和2时的搜索情形。虽然深度限制为1时错过了解，但是深度限制为2时就把它找到了。

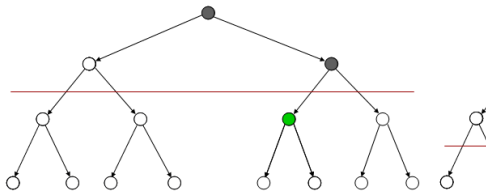


图 4.45: 迭代加深搜索

容易证明：IDS是完全的。如果边开销都是1，则它还是最优的。这一点和广度优先搜索非常相似。但是它有一个很明显的毛病：进行了很多重复工作。深度限制为1时搜索过的结点在深度限制为2时重新被搜索了一遍。这样的重复工作代价有多大

呢？IDS扩展结点数为$(d+1)(1)+db+(d-1)b^2+...+(1)b^d$=O($b^d$)在渐进意义下和广度优先搜索是相同的！而它的空间复杂度仅为O(bd)，不比深度优先搜索差！这样，IDS综合了深度优先搜索和广度优先搜索的优势，是最实用的盲目搜索算法之一。

重复结点前面提到过，15数码问题的解答树里有重复结点，如果把重复结点看成多个不同结点，则状态空间无穷大。为了有效的进行搜索，我们有必要避免重复结点的出现。避免方法有两种，一是记录所有结点，但此时深度优先搜索和IDS的空间复杂度也变得不比广度优先搜索好了，没有一点优势可言，此时广度优先搜索变得最优；第二种方式是只记录部分已出现过的结点，例如当前路径上的结点。这样做保持了渐进空间复杂度，但是判重并不够，内存也没有得到充分利用。实际程序中往往根据内存限制有选择的保存结点，在存得下的情况下尽量多的判重。

边权任意的情况前面提到广度优先搜索和IDS保证最优的前提是每条边的开销均为1，有时这是无法满足的。在权值可以为任意非无穷小的正数的情况下，我们可以使用一种称为**代价一致搜索(uniform-cost search)**的算法，设$g(N)$为从初始状态到N的路径权和，则把边界上的结点按照g值从小到大的顺序进行扩展。需要特别注意的是：重复结点的处理更加复杂。在边权为1的时候后生成的结点的权和不会比先生成的好，但是在一般权值下可能出现后生成的结点比新生成的结点更好（g更小）的情况！

我们设置两个表OPEN和CLOSE保存g值可能会更新和一定不会再更新的（已达到最优值）的结点。当一个结点N被扩展完毕后立刻把它放到CLOSE表中（想一想，为什么）；若新生成的结点N的状态在CLOSE表中，则丢弃N；若新生成的结点N的状态在OPEN表中的结点N'出现过，设保留N和N'中g值更小的一个，而丢弃另一个。这里不打算详细叙述这个算法。在图论部分，我们会详细叙述这个算法，它在图论中被称为dijkstra算法。需要特别注意的是，这里必须把算法框架修改一下，在扩展结点而不是生成结点时判断是否为目标结点。

小测验

1. 描述深度优先搜索、广度优先搜索和它们的变形。证明IDS的时间复杂度为O(b^d)。它扩展的结点数在绝对数值上广度优先搜索多多少？
2. 如果内存足够大，IDS、广度优先搜索、双向广度优先搜索哪个在理论上最优？
3. 如何处理重复结点和边权任意的情况？为什么需要修改搜索算法框架？如何修改？

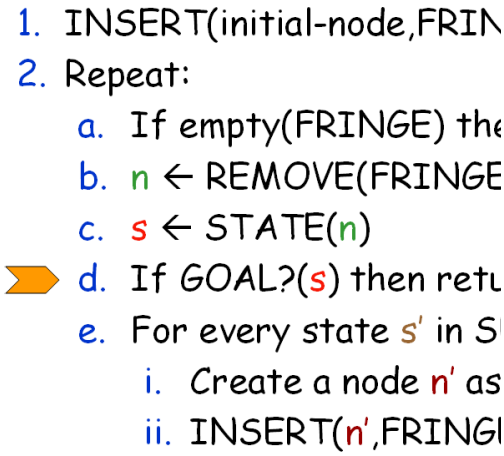


图 4.46: 边权任意的情况

4.4.4 启发式搜索(1)：A*算法

上一节介绍了几个最常见的盲目搜索算法，本节学习启发式搜索的基本思想和应用最广泛的启发式搜索算法：A*算法。

启发式搜索的核心是实际一个启发函数$f(N)$，计算结点N“前途如何”。一般把$f(N)$设计成估计“一条经过N的路径”。既然经过N，路径可以分为两部分：从起点到N的部分和从N到目标的部分。因此经典的评价函数为$f(N) = g(N) + h(N)$，其中$g(N)$为目前已知的从起点到N的最优路径的长度，而$h(N)$是对N到目标的接近程度的**估计**，称为启发函数(heuristic function)。注意，这里的g取决于搜索树（它不是估计，只是当前的最优路径长度），而h和搜索树无关，它是状态的函数。

上图是一棵解答树，每个状态被标上了$g + h$的形式。其中$h(N)$为N不在目标位置上的滑块个数。目标结点的h值为0，而初始状态的g值为0。

h的可接纳性如果允许随意选择h函数，那么很难对算法进行评价：我们甚至无法保证算法是否完全或者最优。一般情况下，我们希望h函数满足：$0 \leq h(N) \leq h*(N)$，其中$h*(N)$表示从N到目标结点的**真实**最优路径长度，称这样的h是可接纳(admissible)的。显然，如果h是可接纳的，$h(G) = 0$。其中G为目标结点。直观的说：可接纳的h是对N到目标结点路径长度的**乐观估计(optimistic estimation)**。例如对于8数码问题的以下三个h函数：$f_1(N) = N$不在目标位置的滑快数目；$f_2(N) = N$所有数字的离家Manhattan距离和；$f_3(N) = N$的逆序对数，其中前两个是可接纳的（想一想，为什么）。

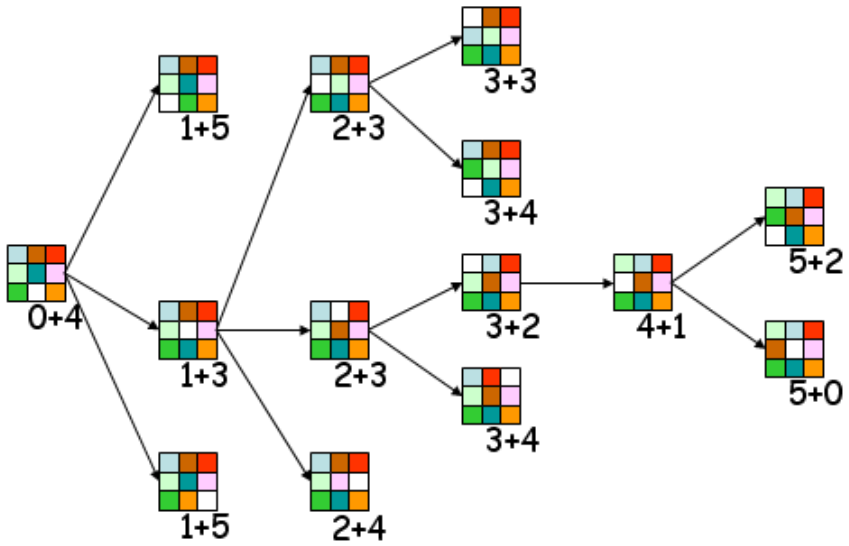


图 4.47: 解答树中的g和h

A*算法叙述了可接纳性之后，我们可以正式的来讨论A*算法：定义$f(N) = g(N) + h(N)$，其中$g(N)$是从初始状态到N的最优路径代价，$h(N)$是一个可接纳启发函数，所有边权$c(N, N') > \varepsilon > 0$，如果把搜索边界按照$f(N)$从小到大排序，则**修改后**的搜索算法称为A*算法。

图8.1.3和图8.1.3是一个路径规划的例子，$h(N)$为N到目标的Manhattan距离。

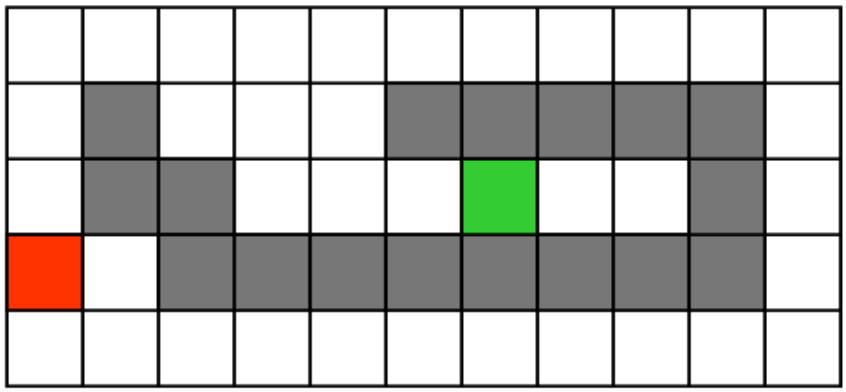


图 4.48: 路径规划问题

这个算法到底如何呢？我们证明关于它的三个结论：

结论一：如果重复结点全部保存，则A*是完全的，也是最优的。

完全性搜索边界上的任意结点N满足$f(N) \geq g(N) > d(N) * \varepsilon$，其中$d(N)$为N的深

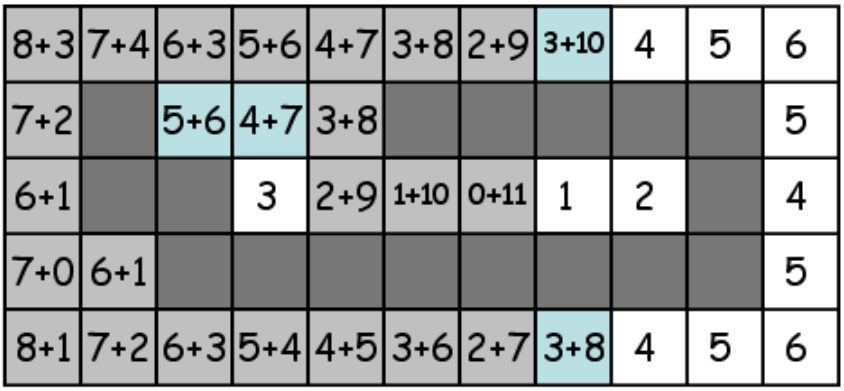


图 4.49: 路径规划问题各格子的g和h

度。如果问题有解，则搜索边界上一定存在某结点K使得从K开始有解。由于每次扩展一个结点后f值至少增加1，所以有限多次扩展后一定会扩展K。这也看出了为什么必须限制$c(N, N') > \varepsilon$：如果每次老是用权为0的边扩展，扩展出的新结点权值不变，则将永无止境的扩展它们和由它们扩展出来的新结点，而永远无法扩展K。

最优性由于采用的是修改后的搜索算法，应证明当目标结点被扩展（而不是生成）时，对应的路径是最优的。设最优解为$C*$，则对于边界上的任意非最优目标结点G'，有$f(G') = g(G') + h(G') = g(G') > C*$, 而边界上任意一个在最优路径上的结点K（K本身不一定是目标结点）都满足$f(K) = g(K) + h(K) \leq C*$，所以不会先扩展G'。这里看到了$h$的可接纳性带来的好处：乐观估计让最优路径上的点优先被扩展。

重复结点结论一假设所有结点都保留下来了。是否可以丢弃新结点呢？图8.1.3的例子给出了否定的回答：新结点的f值（2+90=92）比重复结点的f值（4+90=94）要小，如果把新结点丢弃，则无法扩展出最优解102。

但问题在于：如果保留所有结点，那么搜索树的大小也许和实际的状态数呈指数关系，如图8.1.3：

通常有两种方法处理。第一种方法是像代价一致搜索一样，丢弃较差的结点而不总是丢弃新结点。如果丢弃了的老结点是边界上的，问题还不大；但如果丢弃的是已经扩展的老结点（即在CLOSED表中），必须对新结点重新扩展（想一想，为什么）。也就是说：以前扩展工作等于白做了！最坏情况下这不保留所有结点好多少，并不令人满意。

第二种方法是给$h(n)$加上一个更强的限制：**一致性(consistency)**，或称**单调性(monotone)**。启发函数h被称为一致的，如果$h(G) = 0$，且对任意结点N和它的儿

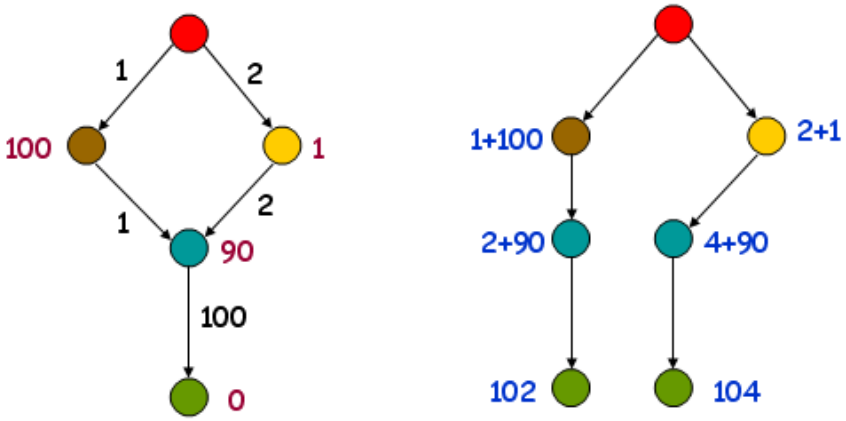


图 4.50: 丢弃新结点可能产生错误

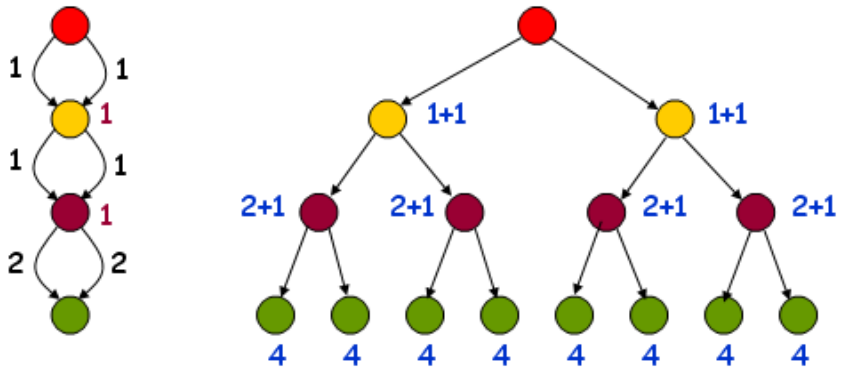


图 4.51: 保留所有结点则搜索树大小可能与实际状态数呈指数关系

子N'，有$h(N) \leq c(N, N') + h(N')$。直观来说，这表示随着搜索的加深，$h$函数越来越精确。它还可以被理解为三角形不等式，如图8.2.1。

可以证明：一致的h一定是可接纳的，但是可接纳的h不一定是一致的。幸运的是：一致性并不难满足。很多可接纳启发函数都是一致的。例如前面提到的8数码问题的启发函数h_1和h_2都是一致的。

结论二：若启发函数h是一致的，则每扩展一个结点N时都已经找到了起点到它的最优路径。

证明：对于任意结点N和它的儿子N'，$h(N) \leq c(N, N')+h(N')$，两边加上$g(N)$得$f(N) \leq g(N) + c(N, N') + h(N') = g(N') + h(N') = f(N')$，因此$f$是单调的。由于A*算法按$f$给搜索不边界排序，因此对于搜索边界上的所有其他结点K'都满足$f(K') \geq f(K)$。由于其他路径一定是从搜索边界继续扩展得到的，但结点扩展会使f越变越大，因此当前没有得到的路径不可能比K更优。在这里我们看到了一致性的好处：保证f的单调性，使

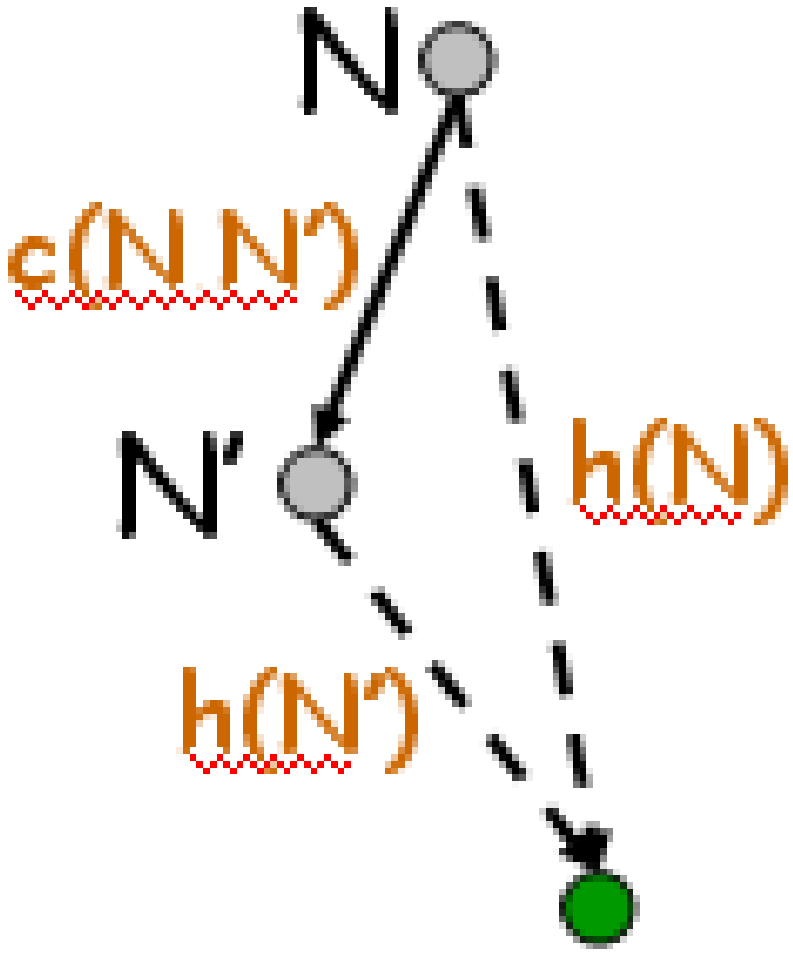


图 4.52: 三角形不等式

得算法变得和代价一致搜索、dijkstra算法本质相同。

这样，虽然仍然无法避免边界上的重复结点出现更新的情况，但是对于**已经扩展完毕**（即CLOSED表中）的结点来说：重复结点一定可以立刻丢弃！如图8.2.1：

这样，我们得到了基于一致启发函数的A*的算法：

A*算法的理论分析如果状态空间有n个状态和r条边，如果每个状态的后继数为O(n)，则r=O(n^2)，图是稠密的；如果后继数为O(1)的，则r=O(n)，图是稀疏的。再大多数搜索问题中，状态空间图是稀疏的。假设CLOSED表被实现为哈希表，平均访问时间为O(1)，而h是一致的，则有两种实现搜索边界的策略：

对于稠密图，方案一理论更优，而对于稀疏图（大多数情况如此！），方案二理论更优。

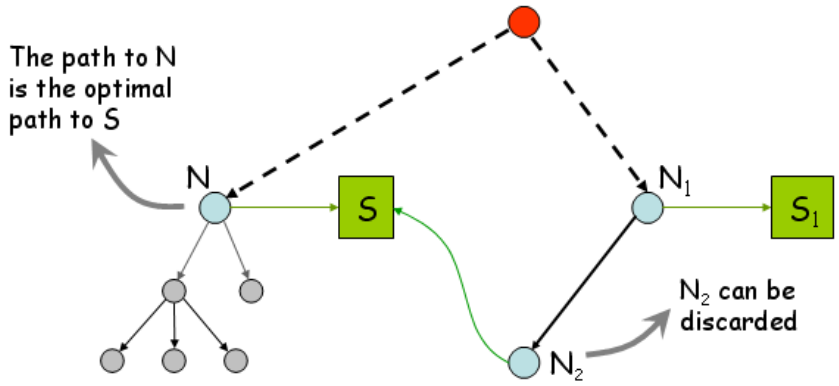


图 4.53: 与CLOSED表中结点重复的结点可以立刻丢弃

- When a node is expanded, store its state into CLOSED
- When a new node N is generated:
  - If STATE(N) is in CLOSED, discard N
  - If there exists a node N' in the fringe such that STATE(N') = STATE(N), discard the node - N or N' - with the largest f

图 4.54: 基于一致启发函数的A*算法

小测验

1. 解释$f(n) = g(n) + h(n)$的含义。哪些部分是和搜索过程有关，哪些只和状态有关？

2. 描述A*算法。为什么A*算法规定边权大于无穷小正数，且h必须是可接纳的？

3. 为什么重复结点有害？有哪两种处理重复结点的方法？哪一种更好？为什么？试比较采取一致启发函数的A*算法和代价一致搜索的最优性证明。

4.4.5 启发式搜索(2)：启发函数和其他算法

启发函数的效果虽然理论分析并没有说明启发函数的作用，但使用不同的启发函数，算法的效率显然是不一样的。比如$h = 0$时其实一点启发信息都没有使用，但

	方案一	方案二
搜索边界实现	无序表	二叉堆
搜索边界的插入次数	O(r)	O(r)
每次插入时间	O(1)	O(logn)
结点扩展次数	O(n)	O(n)
选择扩展结点的时间	O(n)	O(logn)
时间复杂度	$O(r + n^2) = O(n^2)$	O(rlogn + nlogn) 稠密图$O(n^2logn)$ 稀疏图O(nlogn)
空间复杂度	O(n) （无须指向儿子的指针）	O(n)

这个启发函数是一致的，A*算法仍然有效。事实上，$h = 0$时A*退化为了代价一致搜索，如果$c(N, N')$也恒等于1，则退化为了广度优先搜索。令h_1和h_2为两个一致的启发函数，如果对于所有结点N，都有$h_1(N) \leq h_2(N)$，称h_2比h_1更精确(accurate)，或更有信息(informed)的。对于8数码问题，离家距离和函数h_2比不在位数码个数h_1更精确。

结论三：设启发函数h_2比h_1更精确，采用h_1的A*算法为A_1*，采用h_2的A*算法为A_2*，则被A_2*扩展的非目标结点一定会被A_1*扩展。

证明：设最优解为$C*$，则所有满足$f(N) < C*$，即$h(N) < C* - g(N)$的结点都会被扩展。由于$h_1(N) \leq h_2(N)$，因此对于所有非目标结点，如果被A_2*扩展就一定被A_1*扩展。至于目标结点，A_1*和A_2*会各扩展一个，但不一定是同一个。

有效分支因子为了量化的描述一个启发函数的精确程度，我们引入有效分支因子(effective branching factor)。设n为A*算法扩展的结点数，而d为解的深度，则有效分支因子$b*$为满足$n = 1 + b* + (b*)^2 + ... + (b*)^d$的数$b$。由于不同的问题实例所对应的有效分支因子不同，所以采取了多次随机取平均的方法。图8.2.1给出了8数码问题的启发函数$h_1(N) = N$的不在位数码个数和$h_2(N) = N$的离家距离和的比较，括号里的数字是扩展的结点数。

启发函数的设计从上面的例子可以看出，启发函数的设计很重要。应当如何设计启发式函数？这个工作相当难，目前没有找到系统的方法。可接纳性告诉我们：应当进行乐观估计，因此很多时候启发函数的值被设计为**松弛问题(relaxed problem)**的解，即放宽的限制，得到一个容易精确求解的问题。由于新问题的限制更少，所以求出的解是可接纳的。

8数码问题的启发函数h_2相当于是解决以上8个松弛问题：单独移动每个滑块，假

d	IDS	A_1*	A_2*
2	2.45	1.79	1.79
6	2.73	1.34	1.30
12	2.78 (3,644,035)	1.42 (227)	1.24 (73)
16	--	1.45	1.25
20	--	1.47	1.27
24	--	1.48 (39,135)	1.26 (1,641)

图 4.55: 8数码问题的两个启发函数

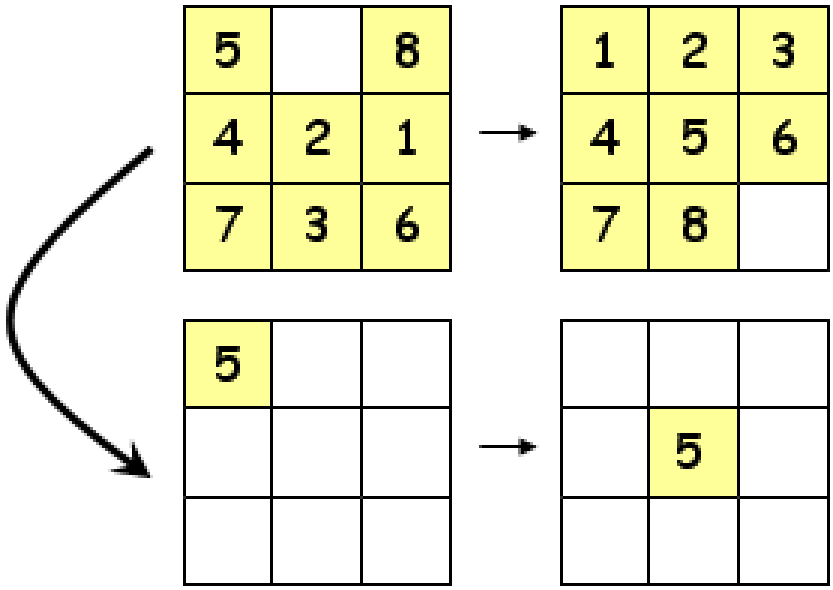


图 4.56: 8数码问题的松弛问题

设每个滑块互不影响。由于这个新问题非常容易解决（每个滑块的移动距离就是初始位置和目标位置的Manhattan距离），所以我们成功的设计出了一个易于计算的启发函数。我们可以进一步扩展这个思想，不把限制放得太宽，只分解为两个子问题。

则新的启发函数h_3=d_{1234} + d_{5678}。可这两个子问题并不容易求解，无法直接算出。一般我们事先离线计算出这些子问题后存放到数据库中，然后解决新问题时读取数据库的值。我们甚至可以设计更多的数据库如1-3-5-7和2-4-6-8数据库，然后让$h_3 = max\{d_{1234} + d_{5678}, d_{1357} + d_{2468}\}$。

这样设计方法实际上用到了两个技巧：一是启发函数的合并，取多个启发函数的最大值，得到的新启发函数仍然是可接纳的；二是无交集的模式数据库，因为移

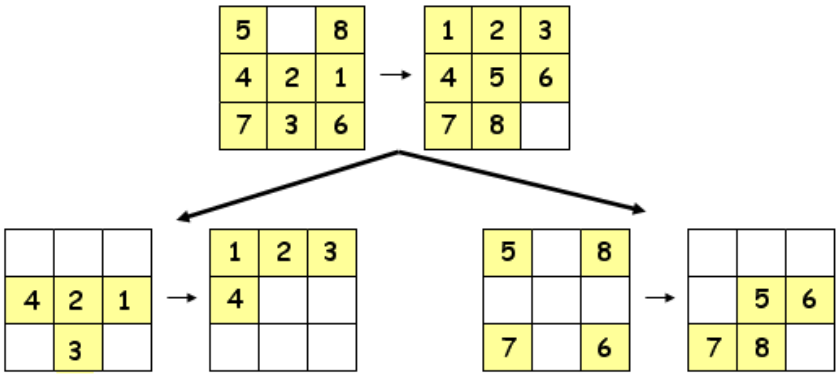


图 4.57: 8数码问题的一个复杂松弛问题

动1234和移动5678是互不影响的。这样的方法在随机15数码问题中扩展的结点数是基于Manhattan距离的A*算法的万分之一。

理论和实际的差别虽然理论上一致启发函数是好的，但问题在于有时候根本找不到这样的函数。本章一开始就强调状态空间搜索注重理论和实际结合，也提醒读者本章的算法分析都是不准确的。A*算法在理论上是完全的，在实际使用中可能需要过长的时间（需要运行1万年和无限循环只有理论上的区别）。一个方法是设定时间限制，另一个方法是寻找虽然理论上不可接纳，但实际上效果还不错的函数。总之，应用状态空间搜索法解题要多从实际出发，不能过分依赖于理论分析。

IDA*算法A*算法的最大问题在于空间消耗太大。和IDS一样，IDA*是A*的简化版本，空间限制少，编程也更加简单。IDA*在编程上几乎和IDS完全一样，只有三点不同：

初始深度限制设置为$f(S)$（而不是1），其中S为初始结点。

当$f(N) \leq cutoff$时才扩展结点（而不是当前深度$g(N) \leq cutoff$时）。

设置下一次的$cutoff$为上一次迭代时碰到了没有扩展的结点的最小f值（而不是简单的把$cutoff$ 加1）。

图8.2.1是cutoff为4时的运行过程（注意只保留当前路径上的结点，在图中画以彩色）。

图8.2.1是$cutoff = 5$时的算法运行过程。

完全性和最优性是显然的，节约了空间且不需要给搜索边界排序。它的缺点是无法避免重复扩展结点，当权值为实数时会有困难，而且空间利用不充分。

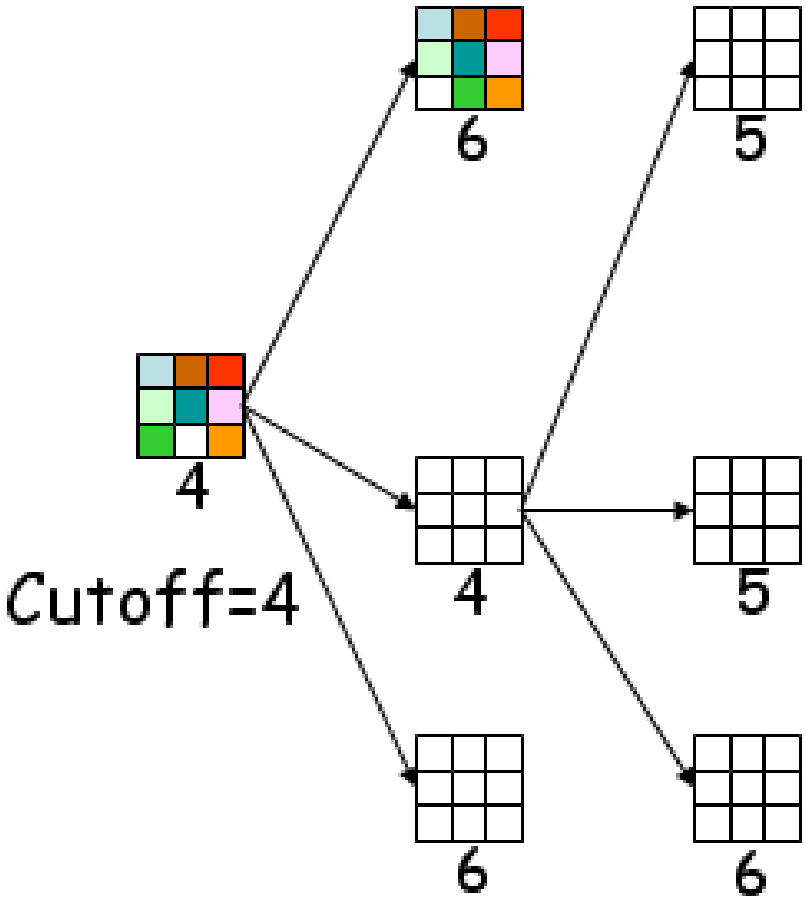


图 4.58: cutoff为4时的运行过程

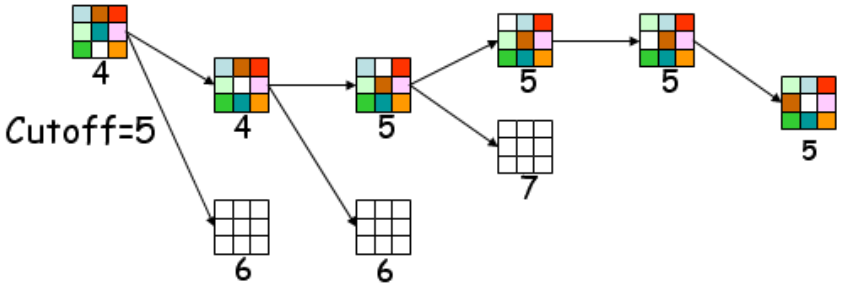


图 4.59: cutoff为5时的运行过程

RBFS算法[3]。递归最佳优先搜索(Recursive-Best-First-Search, RBFS)是一个模仿标准最佳优先搜索的递归算法，和IDA*一样只用线性内存。算法的特别之处在于它

[3] 实际的RBFS比这个复杂，这里只是给出它的大体思想

记录从当前结点的组合可得到的最佳可替换路径的f值，如果当前结点超过这个限制，则递归将自动转回替换路径上。当递归回溯时，对回溯前的当前路径上的每个结点，RBFS用其子结点的最佳f值替换其f值。这样，RBFS能记住被它遗忘的子树中的最佳叶子结点的f值，并因此能够在以后某个时刻决定是否值得重新扩展该子树。

算法框架如下。初始调用为RBFS(S, ∞)，其中S为初始结点。返回的new_f_limit实际上是子树中的最小f值。

```
bool RBFS(Node n, int f_limit, int& new_f_limit){
  if(Goal(n)) return true;
  expand(n, successors);
  if(empty(successors)) { new_f_limit = inf; return false; }
  for(i = 0; i < successors.count(); i++)
    successors[i].f = max(successors[i].g + successors[i].h, n.f);
  while(true){
    best = successors.min();  //选取 最小的后继结点f
    if(best.f > f_limit) { new_f_limit = best.f; return false; }
    alternative = successors.min2(); //选取 第二小的后继结点作为替换路径f
    if(RBFS(best, min(f_limit, altenative.f), best.f)) //递归搜索，在回溯时
更新的值bestf
      return true;
  }
}
```

容易看出，如果不计重复扩展，RBFS仍然**按照f递增的顺序扩展结点**，因此它和A*一样是完全的和最优的。但由于它只保留线性个数的结点，所以在扩展新结点时必须先重新扩展老结点，这是一种用时间换空间的方法。

递归回溯时儿子往父亲“备份f值”实际上是记录此树“下一个需要扩展结点”的f值（父亲老的f值已经失去作用，现在它唯一的用处就是用来扩展出它备份的儿子），而f限制实际上是说：“如果当前结点的儿子f都太大，还不如考虑以前发现的替换结点”。

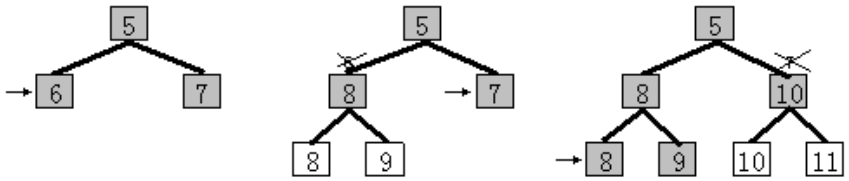


图 4.60: RBFS运行举例

图8.2.1是一个例子。首先从初始结点扩展出两个结点，f值分别为6和7。先扩展值为6的点，因为有一个f为7的点可以“替补”，所以f限制为7。可惜扩展出来8和9，都

超过了f限制，因此回溯，顺便把父亲的f值从6改为8，表示以它为根的子树中未扩展结点的最小f值为8。现在扩展7，并把刚才那个8作为替换结点。这次扩展再一次失望了：得到的两个结点f值为10和11。于是算法重新考虑刚才放弃的那个$f = 8$的结点，并把$f = 10$作为限制，只有当此子树中f不超过10的结点全部扩展完毕才会去考虑替代结点。

通过这里例子，读者直观的体会到了：RBFS和A*一样也是按f递增的顺序扩展结点，但它的优点是空间需求很小。图中只有灰色结点才是当前保留的，$f = 8$的结点被生成了两次：第一次生成时因为f值太大（当时有一个替补$f = 7$）被放弃了。后来由于那个$f = 7$的结点已经被扩展，f限制被放宽到10，才得以重新生成并被扩展。

SMA*算法RBFS的思想很有意思，但空间利用仍然不充分。继续贯彻RBFS的思想可以得到两个算法MA*(Memory-bounded A*)和SMA*(Simplified MA*)。限于篇幅，这里只简略介绍SMA*。

在内存放满前，SMA*和A*完全一样，而当内存放满后，如果不丢弃老结点，将无法加入新结点。SMA*选择丢弃在搜索边界上f值最大的一个结点，并把f值备份到它的父亲。算法有一个细节需要注意：当最佳/最老结点有多个时应如何处理？SMA*规定：扩展最新的最佳结点，删除最老的最差结点。当搜索边界的所有结点f值全部一样时，这样可以避免选择同一个结点进行扩展和删除。只有当搜索边界只有一个结点时才是同一个结点。在这种情况下，搜索树必然是一条链，即使有解也存不下了，丢弃此结点并不是SMA*的错，只怪内存太小。

如果一个解的深度d小于内存大小，那么SMA*一定可以找到这个解，称此解是可达到的。因此如果有解是可达到的，则SMA*是完全的，而如果有最优解是可达到的，则SMA*是最优的。如果所有最优解都不可达到，则SMA*将返回可达到的解中最优的。但正如RBFS那个例子中读者所看到的，如果扩展结点时老是在两条路径里“换来换去”，这样的附加开销可能使一个可解的问题变得（时间上）不可解。

小测验

1. 如何直接比较两个启发函数的精确性？如何从数值上度量启发函数的精确性？设计启发函数有哪些技巧？
2. 什么是IDA*算法？它有什么优点和缺点？
3. 什么是RBFS和SMA*？为什么说它们的扩展顺序和A*类似？两种算法采取怎样的方法进行时空平衡？

4.5 约束满足问题

路径寻找问题把状态看成是一个黑盒子，它可以采用任意的数据结构表示，只需要提供后继函数、启发函数和目标测试。用户感兴趣的是从初始状态到目标状态的一条路径，而不是目标状态本身。

约束满足问题是另一类重要的问题，它不是要找到解的路径，而只需找到一个可行方案。形式化地说，**约束满足问题(Constraint Satisfaction problems, CSP)** 是由一个变量集合X_1, X_2, ..., X_n和约束集合C_1, C_2, ..., C_m定义的。每个变量X_i有一个非空的可能值域D_i，每个约束集合包括一个变量子集，并指定了这些变量之间的取值所应满足的关系。

问题的一个状态是由对一些或全部变量的一个赋值$\{X_i = v_i, X_j = v_j, ...\}$定义的，一个不违反任何约束条件的赋值称为**相容(consistent)**的或者**合法(legal)**的。完全赋值是每个变量都参与的赋值，而CSP问题的解是满足所有约束的完全赋值。某些CSP问题还要求问题的解能使目标函数最大化。

如果每个变量的值域都是有限集，称此CSP为有限CSP；否则为无限CSP。引例“八皇后问题”就是一个有限CSP。本节只讨论有限CSP。

需要说明的是：CSP和枚举、计数问题也是相通的，本节介绍的一些方法同样可以用来给可行解计数或者枚举。

4.5.1 再论回溯法

在引例中我们介绍过回溯法，但并未上升到理论高度。本节从CSP算法的角度重新介绍回溯法，并考虑更多的实现细节。

我们把CSP看成一个路径寻找问题：

初始状态：空赋值{}，其中所有变量都是未赋值的。

后继函数：可以给任何未赋值的变量赋一个值，倘若该值和先前赋值的变量不冲突

目标测试：检验当前的赋值是否完全

路径耗散：每一步的耗散都是常数（例如1）。

则回溯法实际上是每次选择一个变量然后依次尝试它的每一个取值。由于路径是无关的，所以只保存当前状态即可。

图8.2.1是三变量的回溯法的执行例子，灰色结点表示曾经生成过但目前并未保存的结点。

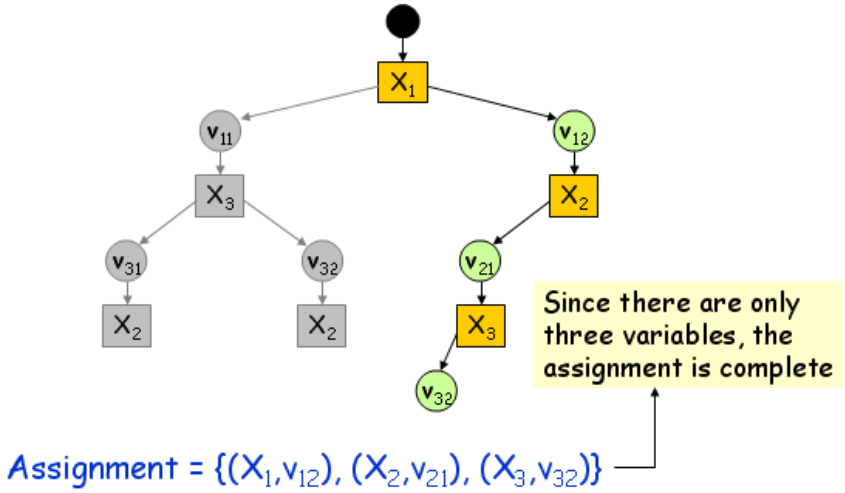


图 4.61: 三变量回溯法的执行例子

回溯法框架如图8.2.1，初始调用为CSP-BACKTRACKING({})。

```
CSP-BACKTRACKING(A)
1. If assignment A is com
2. X ← select a variable r
3. D ← select an ordering
4. For each value v in D d
     a. Add (X←v) to A
     b. If A is valid then
          i. result ← CSP-BACk
          ii. If result ≠ failure
5. Return failure
```

图 4.62: 回溯法框架

注意到这个框架和前面的八皇后算法是有区别的。在八皇后问题中，我们并没有在每层结点中保存整个状态A，而只保存了每一层增加的**新赋值**。这种办法比较节约内存，速度也有所提升（避免了已有赋值的拷贝）。

我们可以框架稍微优化一下。有时候当前变量赋值后会引起未来的某个变量的每个赋值都将引起冲突，导致无解。刚才的算法并不能发现这一点，因此需要引入**前向检查(Forward Checking, FC)**技术，即根据新增加的赋值更新每个变量的值

域var-domains。由于做了检查，所有的取值都是没有冲突的，因此不需要检查A是否合法。

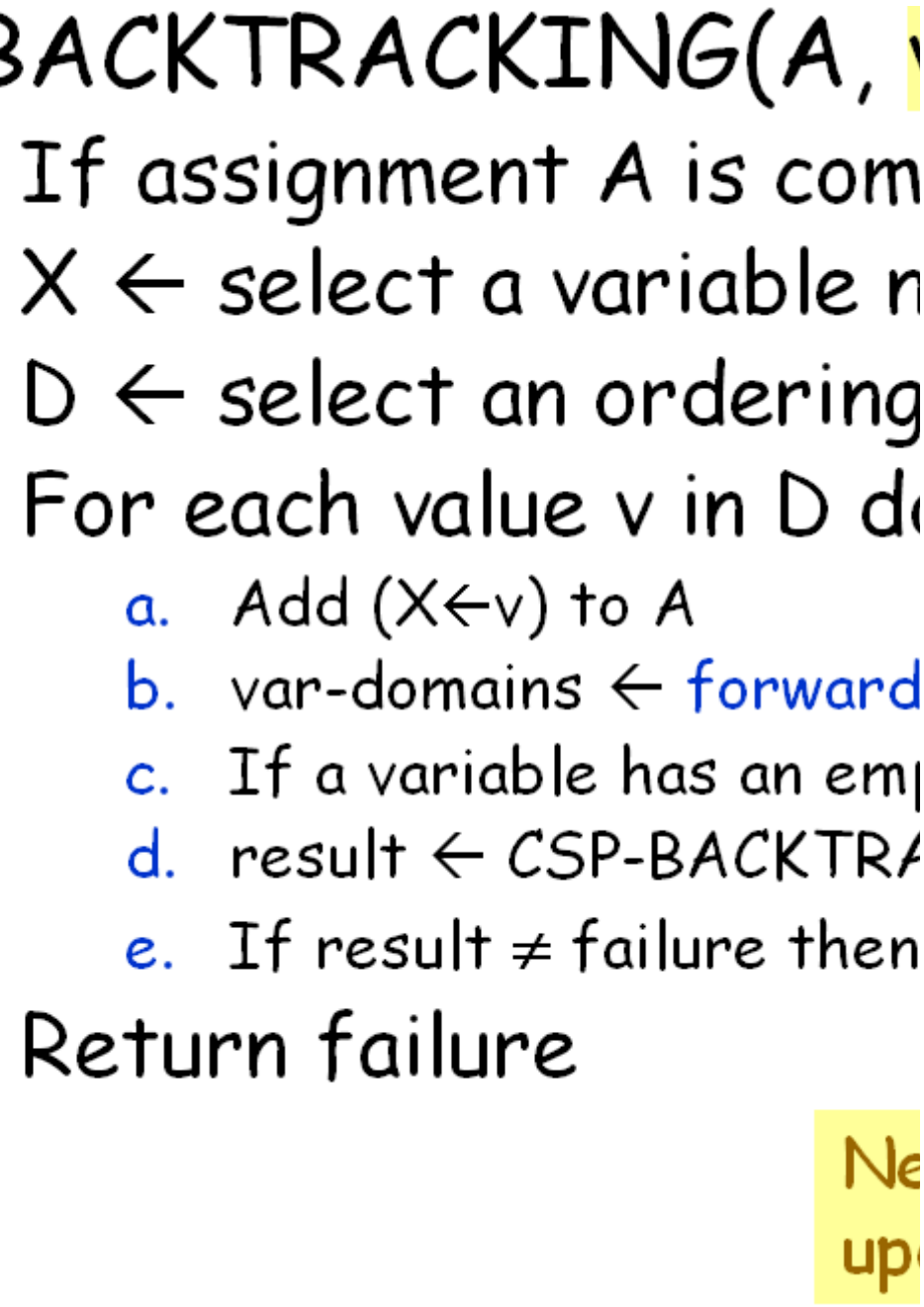


图 4.63: 优化后的回溯法框架

如果读者对前面八皇后问题的程序还有印象，那么可以看到used数组就是这里的var-domains。不过在那个程序中var-domains并没有作为参数传到回溯函数中，而是

采取了更新-恢复的策略，只保留“当前取值集合”。需要注意的是在一些问题中恢复先前的取值集合并不容易，因此仍然需要保留所有取值集合，或者把它作为参数传进回溯函数。

框架中有两个关键点，即两个“select”。如何选择一个未被赋值的变量？如何确定值的排列顺序？这两个问题的解决方法是重要的，因为：

若当前状态无法扩展成为合法解，选择合适的未赋值变量进行赋值将有助于早点发现这一事实；

若当前状态可以扩展成为合法解，先尝试正确的值将能更早的找到这个解。

对于第一个问题，通常采取最少剩余值（MRV）启发式，也称最受约束变量(Most constrained variable)启发式，即选择取值范围最窄的变量。它的直观想法是降低分支因子。如果有多个可能的选择，取在约束中出现最多的一个（即对其他变量约束最大的一个），称为度启发式，它的直观想法是通过约束其他变量来间接降低分支因子。

对于第二个问题，通常采取最少约束值启发式，即按照“排除其他未赋值变量取值范围”从少到多排序。因为我们的目标是要找到解，排除其他变量的取值范围越多越不利。当然，如果需要求出所有解的话，排序规则就没有意义了。注意：这个启发式需要一定的计算量，因为在排序时需要对每个取值进行一次前向检查，而不只是对已经选好的值进行。

需要特别说明的是，有时候把回溯法简称BT（Backtracking），而前向检查简称为FC（Forward Checking）。理论上FC并不是BT的一种优化形式，而是和BT不同的一个算法，因为它并没有回溯（没有检查A的合法性，因为它一定是合法的）。在大多数应用中，FC+MRV启发式都是理想的选择。

小测验

1. 什么是CSP？如何把CSP看作路径寻找问题？它和一般路径寻找问题有什么不同？
2. 什么是BT？什么是FC？叙述二者的算法框架。二者有什么不同之处？
3. 在BT和FC中，选择被赋值的变量和决定值的选择顺序有哪些启发式？它们的直观想法是什么？

4.5.2 约束传播和AC3算法

前向检查能发现许多矛盾，缩减取值范围，但它不能检验出所有矛盾。约束传播

是将一个变量的约束内容传播到其他变量上的一般术语，它可以检测到更多的矛盾。

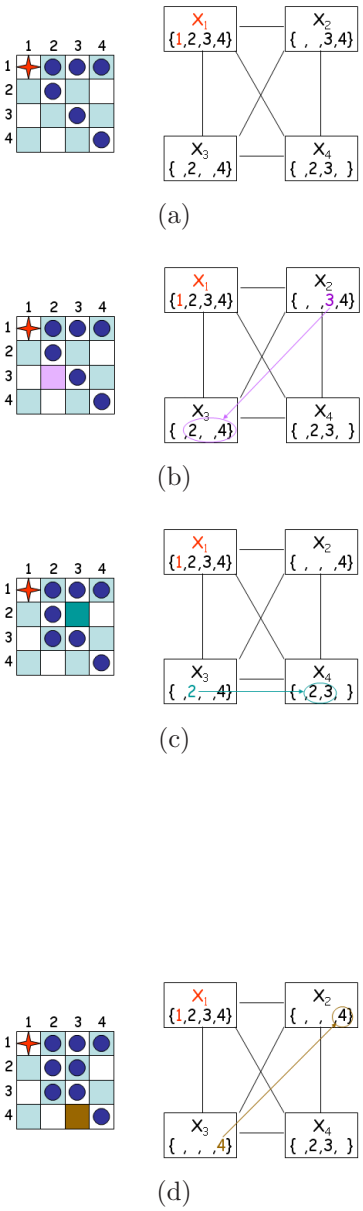


图 4.64: AC3算法运行举例

考虑4皇后问题。假设已经在左上角放了一个皇后，先进行FC，结果如图(a)，已经把X_2，X_3和X_4的取值范围缩小了。此时可以继续判断：X_2如果取3则一定和X_3冲突（不管X_3等于2还是4！），因此X_2不能取3，如图(b)。类似的，X_3不能取2，因为它一定和X_3冲突，如图(c)。最后，X_2必须为4而X_3也必须为4，而这两个“必须取的值”是不相容的。因此我们得出结论：“左上角放皇后”这个状态无解。

刚才的推理过程使用的就是约束传播的思想。显然，FC是约束传播的一种简单情况，它只排序和当前变量不相容的情况。

下面是另外一个例子。3×3的格子有9个变量，要求3行3列和主对角线的和都是6，即有7个约束。如下图。

X_1	X_2	X_3	This row must sum to 6
X_4	X_5	X_6	This row must sum to 6
X_7	X_8	X_9	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

图 4.65: 9变量问题的初始状态

当选择左上角元素为1后前向检查都无法发现矛盾，因为这里的约束是三元的。但实际上通过$1+X_2+X_3=6$可以知道X_2和X_3只能是2和3，不可能是1，因此X_2和X_3都不是1。同理，X_4, X_7, X_5, X_9都不是1。考虑第2列，如果$X_8=3$，则X_2和X_5的可能取值无法满足$X_2+X_5+3=6$。因此X_8不可能为3。同理X_6也不能为3。约束传播后的取值范围如图8.2.2

1	2, 3	2, 3	This row must sum to 6
2, 3	2, 3	1, 2	This row must sum to 6
2, 3	1, 2	2, 3	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

图 4.66: 9变量问题一次传播后

现在约束传播结束，我们尝试取$X_2=2$，经过约束传播后得到了一个解，如图8.2.2：

可见，约束传播比简单的前向检查要强得多。

AC3算法AC3算法进行**二元约束(binary constraints)**的传播，即只适用于每个约束恰好涉及到两个变量的情形。二元约束可以用约束图来表示，结点代表变量，弧代表约束。虽然只适用于二元约束，AC3算法提供了一个很好的思路，可以用来扩展到高阶约束。算法的基本思想是对于每个变量X考虑所有和它共处同一个约束c的变量Y，

1	2	3	This row must sum to 6
2	3	1	This row must sum to 6
3	1	2	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

图 4.67: 9变量问题两次传播后

如果Y的某个取值v使得X的每个取值无法满足约束c，则从Y的取值范围里删除v。这一过程可以从图8.2.2中的框架来描述：

其中REMOVE-VALUES(X, Y)就是刚才提到的更新定义域的过程。注意到如果Y的定义域被更新，这个更新也许还能继续传播，所以需要把Y插入队列。

下面分析算法的时间复杂度。设n为变量个数，d为每个变量的初始定义域大小，s为每个变量所涉及到的最大约束数目（对应于约束图的最大度数），则每个变量最多被插入队列d次（因为每次插入前定义域大小至少减1），因此出队次数不超过nd，因此REMOVE-VALUES被调用最多nsd次。显然REMOVE-VALUES的时间复杂度为O(d^2)，因此总时间为O(nsd^3)。对于完全图，$s = n - 1$，因此总时间为O(n^2d^3)。

扩展AC3算法并不适合高阶约束，它甚至对于二元约束也不是完全的：有一些二元约束的矛盾无法被AC3所检测到，例如图8.2.2

要想检测到这样的矛盾，必须考虑约束图上的三角形而不只是弧约束，而考虑三角形的代价是时间开销上升了一个数量级。AC3的思想同样可以扩展到高阶约束，但代价仍然是时间开销。

使用AC3的回溯算法我们把AC3应用到回溯算法中，得到图8.2.2中的算法框架。需要特别注意的是这里在枚举取值时仍然使用了FC，因为它能快速检测到一些“明显”的矛盾。由于FC和AC3的使用，这里仍然不需要检查A是否是合法的：它一定是合法的。

小测验

1. 什么是约束传播？为什么说前向检查是一种简单的约束传播？
2. 什么是二元约束？什么是约束图？AC3算法的基本思想是什么？时间复杂度如何？
3. 二元约束中的冲突一定只由两个变量组成吗？如何扩展AC3算法，使它

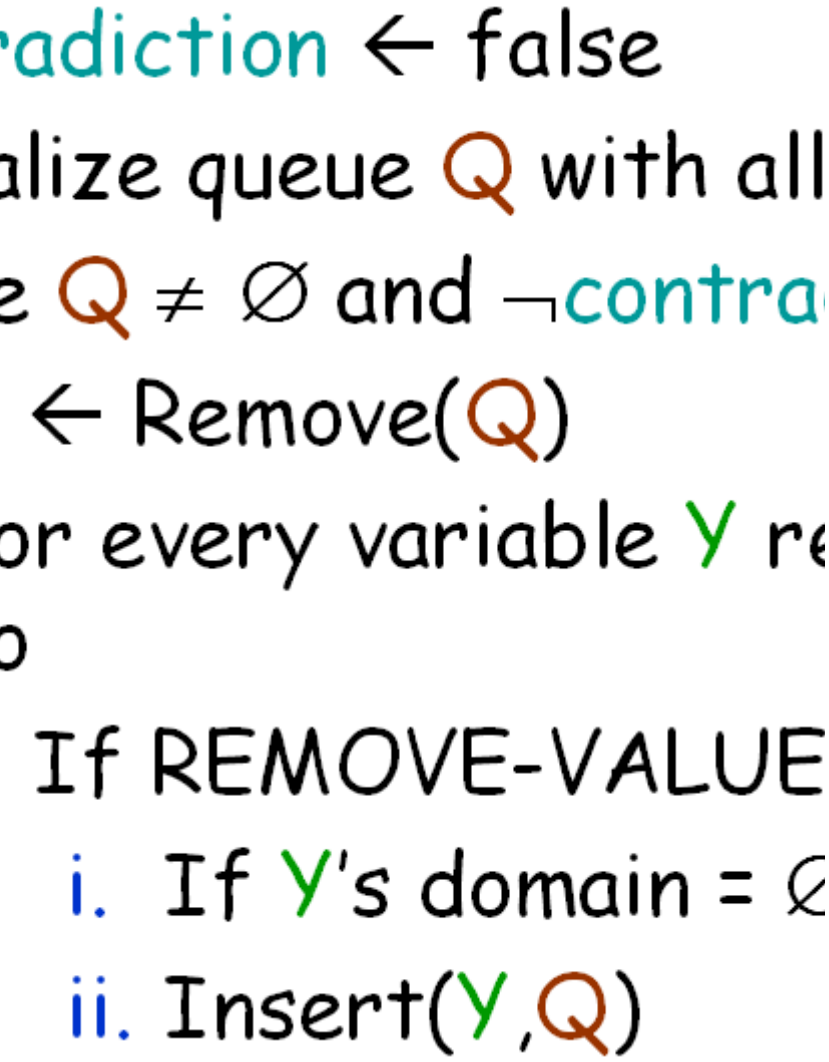


图 4.68: AC3算法框架

能检测更复杂的二元约束中的冲突，以及高阶约束中的冲突？这样做的代价是什么？

4.5.3 CSP问题的结构

仍然考虑二元约束。假设我们已经有了一张约束图。究竟什么样的约束图是“简单”的？即：可以更快的找到解？我们考虑着色问题：给一个地图，给每个区域涂一种颜色，使得任两个有公共边的区域被涂上不同的颜色，如图8.2.2。

我们很快发现T和其他部分是不相连的。显然，约束图的不同连通分量可以分别处

REMOVE-VALUES(X,Y)

1. removed ← false
2. For every value v in the domain of Y do
   - If there is no value u in the domain of X such that the constraint on (x,y) is satisfied then
     a. Remove v from Y's domain
     b. removed ← true
3. Return removed

图 4.69: REMOVE-VALUES框架

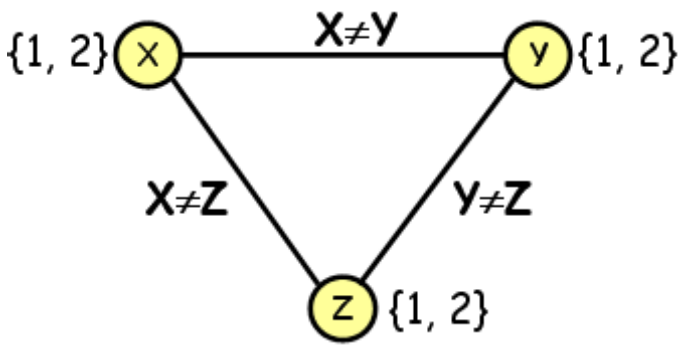


图 4.70: 二元约束中无法被AC3所检测到的矛盾

CSP-BACKTRACKING(A, var-

1. If assignment A is complet
2. var-domains ← AC3(var-do
3. If a variable has an empty
4. X ← select a variable not ir
5. D ← select an ordering on t
6. For each value v in D do
   a. Add (X←v) to A
   b. var-domains ← forwarc
   c. If a variable has an em
   d. result ← CSP-BACKTR
   e. If result ≠ failure then
7. Return failure

图 4.71: 使用AC3的回溯法

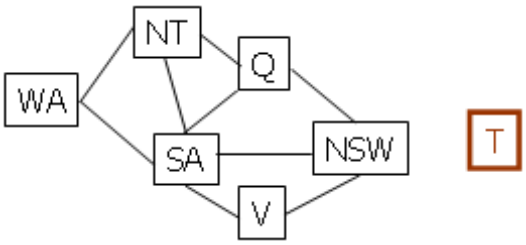


图 4.72: 约束图举例

理，这是我们得到的第一个结论。可是这个结论并不总能派上用场，我们必须继续分析。

由于树是一种特殊的图，我们很自然的考虑约束图为树状的CSP是怎样的。首先对这棵树进行自根向叶子的排序（通过一次BFS即可实现），得到一个拓扑顺序X_1, X_2, ..., X_n。下一步类似于建堆以后的调整：按照$j = n$, n-1, n-2...的顺序调用REMOVE-VALUES(X_j, X_i)，其中X_i是X_j的父亲。然后给根X_1随便赋一个合法值，再按照j=2, 3, ..., n的顺序给每个X_j任选一个和X_i相容的值。

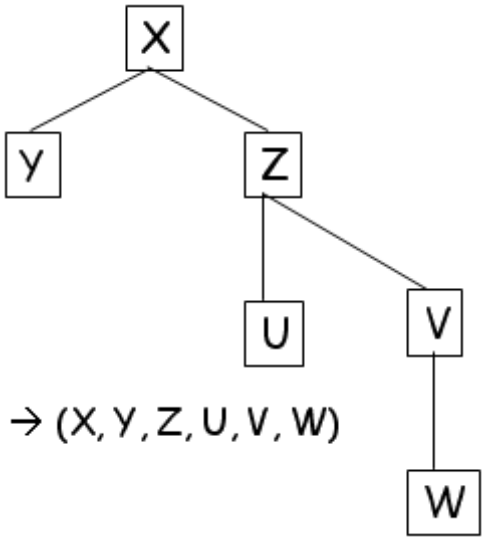


图 4.73: 树状CSP

算法的正确性不难看出：自底向上过程保证任何被删除的值不会危及已经处理过的弧的相容性，而处理之后CSP是有向弧相容的，因此自顶向下过程是不需要回溯的。算法的时间复杂度为O(nd^2)，因为它只是执行了n次REMOVE-VALUES和n次选择。

现在我们对树状CSP有了高效算法，因此需要想办法把一般的图简化成树。如何简

化呢？通常有两种方法。

方法一：删除结点在刚才的着色问题中，如果能够确定SA的值，那么在约束传播后问题化为了图8.2.2。

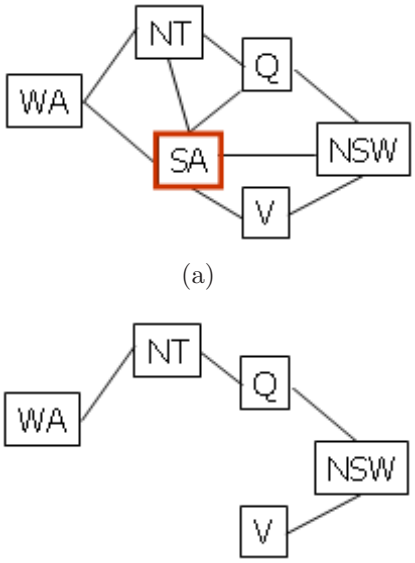


图 4.74: 删除结点法

一般地，算法如下：

步骤一：选择一个变量子集S，使得约束图在删除S后成为一棵树。S称为环割集。

步骤二：对于满足S所有约束条件的S中变量的每个可能赋值，从剩余变量值域中删除与S的赋值不相容的值，如果去掉S后的CSP有解，把解连同S的赋值一起返回

这个方法称为割集调整法，如果S的大小为c，那么运行时间为$\mathrm{O}(d^c \times (n-c)d^2)$。找最小环割集是NP难度的，但是已经有一些高效的近似算法。

方法二：合并结点这个方法实际上是分治的：是把约束图分解为相关联的子问题集，独立求解然后合并。和多数分治算法一样，如果没有哪个子问题特别大，那么效果会很好。这样的分解称为**树分解(tree decomposition)**。图8.2.3是着色问题中大的连通分量的树分解，它把原问题分成了5个子问题。

一个树分解必须满足三个条件：

每个变量至少在一个子问题中出现

如果两变量有约束相连，则它们至少同时出现在同一个子问题中（连同约束）

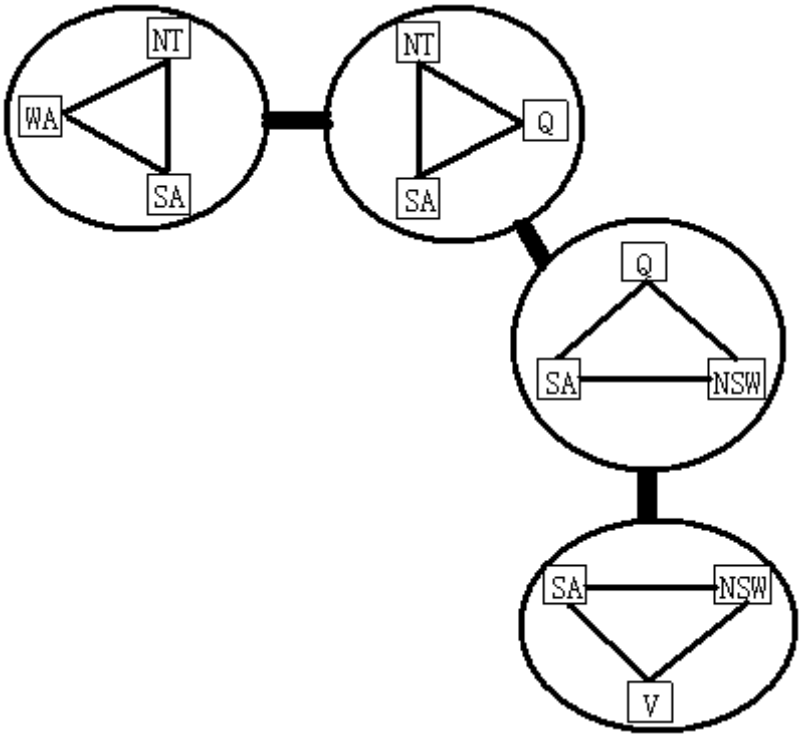


图 4.75: 合并结点法

如果一个变量出现在树中的两个子问题中，它必须出现在连接这些子问题的路径上的所有子问题里。

前两个条件是显然需要满足的，最后一个条件只是反映了任何给定变量在每个子问题中必须取值相同的约束。如果一个变量所在的子问题连成一条路径，那么根据传递性，在这些所有子问题中，该变量的取值都会相等，这也为连接子问题提供了可能。

进行树分解以后，我们先独立求解所有子问题。注意我们必须求出每个子问题的所有解，并把这些解作为每个子问题所对应的“大变量”的取值。这样，我们得到了一个树状二元CSP：每个结点对应每个子问题的“大变量”，两个“大变量”之间的约束是：共享变量必须有相同的取值。根据树分解的第三个条件，共享变量的取值将传遍每个变量出现的整条链。

一个约束图有多种树分解，显然子问题越小越好，因为定义树分解的树宽为最大子问题大小减1，而图本身的树宽是所有树分解的最小树宽。如果能找到一个树宽为w的树分解，那么这个算法的时间复杂度为O(nd^{w+1})，因此树宽有界的CSP是多项式

时间内可解的。找到最小树分解是NP难度的，不过有一些效果不错的启发式方法。

约束满足问题的内容还有很多，这里只是进行了最简单的介绍。除了回溯法这样的搜索算法之外，Kirkpatrick等人在模拟退火方面的工作使得局部搜索算法在约束满足问题中得到了普及。应用最小冲突启发式的局部搜索可以在很短时间内求解百万皇后问题。

约束传播方法被证明在许多问题中能完全消除回溯的需要。AC3算法实现的是弧约束，更好的AC4算法由Mohr和Henderson于1986年提出。在每个变量赋值之后的完全弧相容检测算法被Sabin和Freuder称为MAC算法，有证据说明在更难的CSP问题上MAC算法是成功的。

本节介绍的CSP问题结构方面的研究起源于Freuder。树宽的概念由Robertson和Seymour于1986年引入。

小测验

1. 为什么约束图的不同连通分量可以单独考虑？
2. 叙述树状二元CSP的多项式算法。为什么它是正确的？可以把它扩展为枚举所有解的算法吗？
3. 把一般约束图转化为树有哪两种方法？叙述环割集和树宽的定义，并说明环割集和树宽对CSP算法时间复杂度的影响。

4.6 对抗搜索

本节考虑对抗搜索。在前面的讨论中，状态空间的转移只取决于单一的智能体，而在很多情况下可以有第二个智能体参与。典型的例子是对抗类游戏。在这样的情况下，单智能体搜索变成了对抗搜索，它有一些不一样的特征。

本节只讨论双人对弈游戏，目的在于提出一个和单人游戏（如前面介绍的15数码问题）一样**通用**的方法。特殊双人游戏的数学方法将在本书的第三部分加以讨论。

为了方便讨论，我们假设我们的主人公叫MAX，而它的对手叫MIN[4]。假设MIN的目的是希望MAX输，而MAX的目的是让MIN输，即游戏是对抗性，而不是合作性的。正规的说，本节只讨论信息完全的、确定性的、轮流行动的、两个游戏者的零和游戏。这样的双人游戏很多，其中经典项目有：国际象棋、西洋跳棋，黑白棋、

[4] 读者稍后就会知道MAX和MIN这两个看起来奇怪的名字的来由

围棋等。这些问题非常难于求解，例如国际象棋的平均分支因子大约是35，一盘棋一般每个游戏者走50步，所以搜索树大约有35^{100}即10^{154}个结点（尽管整个状态空间"只"约10^{40}个不同结点）。和现实世界一样，游戏要求即使无法找到最优决策也必须能做某种决策，而不能花费太多的时间。换句话说，这些游戏有严格的**时间限制(time limit)**。所以对博弈的研究也产生了一些有趣的思想，如何尽可能充分的利用好时间。

4.6.1 问题定义和MINIMAX算法

考虑两个游戏者MAX和MIN，MAX先行，轮流出招，直到游戏结束。在游戏最后，给胜者加分，给败者罚分。类似于CSP，我们可以把双人游戏写成类似于路径寻找问题的严格定义：

初始状态：包括棋盘局面和确定该哪个游戏者出招

后继函数：返回(move, state)列表，每一项表示一个合法招数和对应的结果状态。

终止测试：判断游戏是否结束。游戏结束的状态称为终止状态。

效用函数：也称目标函数或收益函数，是终止状态的得分。国际象棋中赢、输、平分别是1，-1和0分，而围棋、黑白棋等可以有更多的结果。

考虑一个简单的游戏：井字棋。在3×3的棋盘上，MAX划叉，MIN划圆圈。任何一种图案占据了一整行、一整列或者一整条对角线（一共有两条），相应的游戏者胜利。初始局面为空，而双方轮流走步将得到这样一棵类似于搜索树的**博弈树(game tree)**。

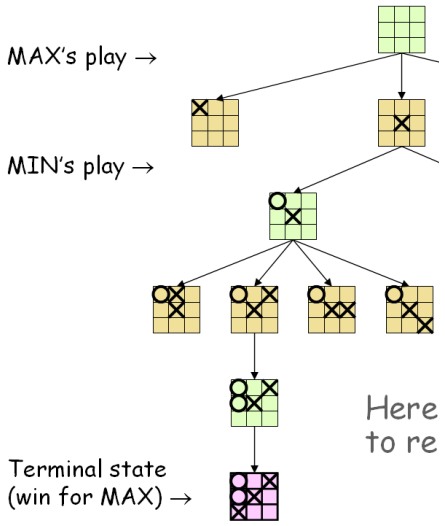


图 4.76: 井字棋游戏

仔细观察图8.2.3中这棵树，目前只有叶子有评价函数值（它们是终止状态）。我们通过自底向上的方法把每个结点的值“备份”到它的父亲，直到根结点有值。如何备份呢？对于MAX来说，评价函数越大越好，而对于MIN来说评价函数越小越好。对于最聪明的对手来说，它一定会选择对自己最有利，即后继状态值最小的走步。换句话说：MAX结点的值是它所有儿子的最大值，而MIN结点的值是它所有儿子的最小值。

请注意：这个结论的前提是对手总是采取最优策略。如果一个MIN结点有三个儿子，评价值分别为3，4，-1。最聪明的对手一定会选择那个-1的儿子（这样对MAX最不利），而如果对手并没有发现这个走步（或者并不觉得它的后继状态对MAX最不利），它可能选择的是3或者4。

可惜完整的博弈树往往过大，无法完整的计算出来，所以通常需要在一个指定的深度处截断。这样，即使截断处并不是终止状态也必须给它一个评价值$e(s)$。其中$e(s) > 0$表示对MAX有利，且值越大对MAX越有利；$e(s) < 0$表示对MIN有利，且负得越厉害对MIN越有利。$e(s) = 0$表示局面是中性的，双方优势相当。对于井字棋游戏，一个可能的评价函数是：

e(s) = (MAX可能占有的行/列/对角线数) - (MIN可能占有的行/列/对角线数)

其中“可能占有”的意思是此行/列/对角线上不含对方的符号。更复杂的评价函数往往是对各种特征的线性加权。图8.2.3是采用这样的评价函数时截断深度为2时的部分博弈树。

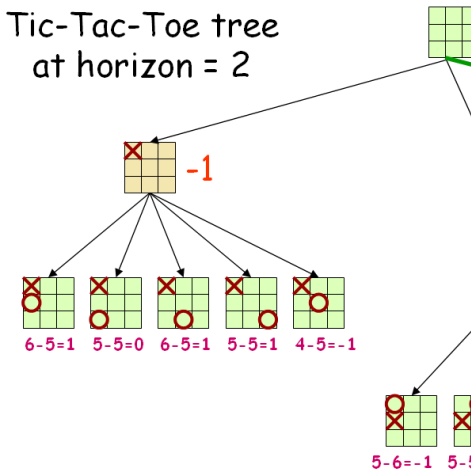


图 4.77: 评价函数

读者可以验证：MIN结点的值取儿子中最小的一个；MAX结点的值取儿子中最大的一个。于是MAX选择走中间。假设此时MIN走第一行正中，则此时MAX的部分搜索

树如图8.2.3：虽然叶子的评价函数有很多2和3，可惜由于它们的父亲是MIN结点，所以这些值并没有被备份到父亲结点。

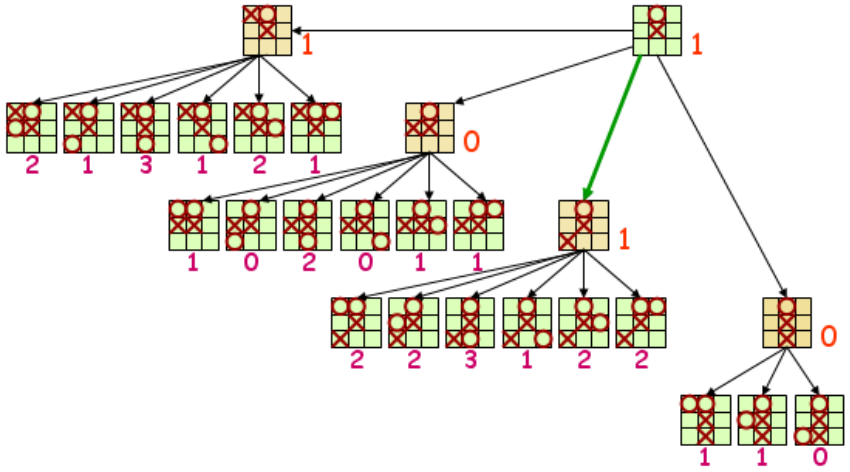


图 4.78: MAX和MIN的含义

如果你相信随着游戏的进行，评价函数e将越来越准确，那么这样的搜索无疑比直接选择e值最大的儿子要好。

MINIMAX算法刚才叙述的算法称为MINIMAX算法。我们采取递归计算的方法，整个算法是易于描述的。

```
int max_value(int dep, state s){
  if (terminal(s)) return e(s);          //终止状态
  if(dep == maxdepth) return e(s);   //深度截断，返回评价函数
  v = -inf;                          //初始化为负无穷
  succ = make_successors(s);         // succ[i为第]个后继状态i
  for(i = 0; i < succ.count; i++)
    v = max(v, min_value(succ[i])); //计算所有儿子的最大值
  return v;
}

int min_value(int dep, state s){
  if (terminal(s)) return e(s);          //终止状态
  if(dep == maxdepth) return e(s);   //深度截断，返回评价函数
  v = inf;                           //初始化为无穷大
  succ = make_successors(s);         // succ[i为第]个后继状态i
  for(i = 0; i < succ.count; i++)
    v = min(v, min_value(succ[i])); //计算所有儿子的最小值
  return v;
}
```

主算法计算当前结点s的max_value，设为v，然后选择评价值为v的任何一个走步。

小测验

1. 什么是双人游戏？本节讨论的双人游戏满足哪些条件？
2. 什么是评价函数？非终止结点的评价函数应满足怎样的条件？
3. 什么是博弈树？博弈树非叶结点评价值应如何计算？描述MINIMAX算法。它的时间复杂度是怎样的？

4.6.2 $\alpha-\beta$剪枝和其他优化

完全的MINIMAX算法有很多不必要的计算，下面有一个例子。

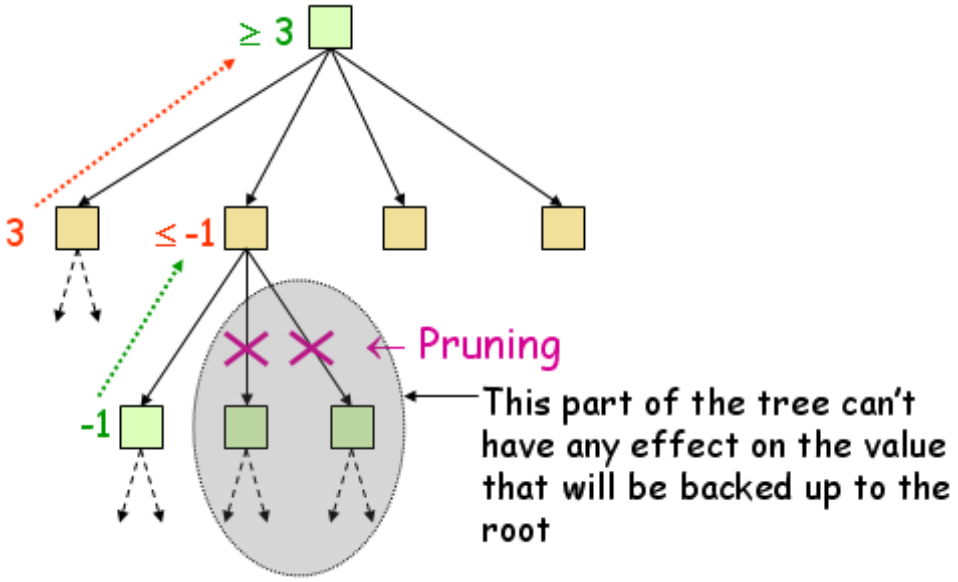


图 4.79: $\alpha-\beta$剪枝

假设正在计算一个MAX结点s。它的第一个儿子s_1的MIN值为3，因此s的**当前MAX值**3，它的真实MAX值至少为3。它的第二个儿子s_2的MIN值还没有计算完，但已经知道它的第一个儿子的MAX值为-1，因此s_2的**当前MIN值**为-1，它的真实MIN值最多为-1。

既然s_2的MIN值不超过-1，它根本不可能影响s的MAX值（想一想，为什么）。虽然还没有算完，但其实已经不需要计算了。我们把s_2的其他子树**剪枝(prune)**掉。

我们把这个想法严密化，即定义：

MAX结点的α值为它的评价值下限。它在计算过程中可能增加，但不会减少。

MIN结点的β值为它的评价值上限。它在计算过程中可能减少，但不会增加。

这样，刚才的剪枝是因为MIN结点的β值小于它父亲的α值。这是一个有效的剪

枝规则，但并不是$\alpha-\beta$剪枝的全部！事实上，如果MIN结点的β值小于或等于它的某个祖先（并不一定是父亲！）的α值，都应该被截断（想一想，为什么）。同样的，如MAX结点的α值大于或等于它的某个祖先的β值也应该被截断。

这样的剪枝条件适合手工操作，但是不太好直接写到程序里。注意到MINIMAX过程是递归的，我们可以简单的在递归函数中加入两个附加参数alpha和beta，表示当前结点的祖先中**最大α值和最小β值**。加入$\alpha-\beta$剪枝的MINIMAX过程框架如下：

```
int max_value(int dep, state s, int alpha, int beta){
  if (terminal(s)) return e(s);        //终止状态
  if(dep == maxdepth) return e(s);  //深度截断，返回评价函数
  v = -inf;                        //初始化为负无穷
  succ = make_successors(s);       // succ[i为第]个后继状态i
  for(i = 0; i < succ.count; i++){
    v = max(v, min_value(succ[i], alpha, beta));  //计算所有儿子的最大值
    if (v >= beta) return v;                     //β剪枝
    alpha = max(alpha, v);                      //更新α为最大值
  }
  return v;
}

int min_value(int dep, state s){
  if (terminal(s)) return e(s);        //终止状态
  if(dep == maxdepth) return e(s);  //深度截断，返回评价函数
  v = inf;                         //初始化为无穷大
  succ = make_successors(s);       // succ[i为第]个后继状态i
  for(i = 0; i < succ.count; i++){
    v = min(v, min_value(succ[i], alpha, beta)); //计算所有儿子的最小值
    if(v <= alpha) return v;                    //α剪枝
    beta = min(beta, v);                       //更新β为最小值
  }
  return v;
}
```

结点排序下面的例子说明了：剪枝效果和结点检查顺序有关。同一棵树，如果先检查-1的结点则可以把4剪掉；而如果先检查4则什么都剪不掉。

考虑一棵高度为h，分支因子为b的博弈树，MINIMAX将检查所有$O(b^h)$个结点，而$\alpha-\beta$剪枝在最坏情况下什么都剪不掉。一个很自然的问题产生了：在什么样的情况下，$\alpha-\beta$剪枝算法将访问最少的结点？Knuth和Moore在1975年证明了：当MAX结点的MIN儿子按照评价值递减、MIN结点的MAX儿子按照评价值递增的顺序排列时，$\alpha-\beta$剪枝算法只检查$O(b^{h/2})$个结点，即分支因子变为$b^{1/2}$（对于国际象棋来说，35变为6），或者在相同时间内可以检查两倍的深度。

可惜这只是理想情况，无法保证达到——如果能保证完美排序，就不用搜了。如

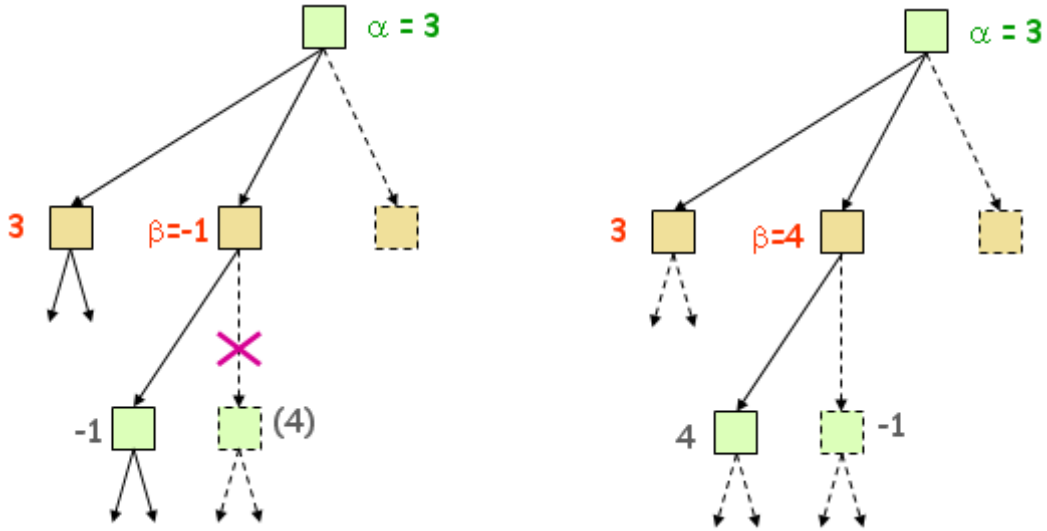


图 4.80: 结点排序

果随机检查结点，则要检查的结点数大约是O($b^{3d/4}$)。对于国际象棋来说，用一种相当简单的排序函数（如吃子优先，然后是威胁、向前走子和向后走子）效果很好。如果增加动态行棋方案，参考直接评价值和上次迭代的备份值进行排序，则可以让我们非常接近理论极限。

其他优化手段前面提到过，国际象棋的搜索树约10^{154}个结点，但实际不同的状态数只有10^{40}，这说明搜索过程中有大量重复结点。重复的来源是**调换(transportation)**，因此可以采用调换表消除重复，它的作用类似于路径寻找问题中的哈希表。真实的对弈程序的截断深度是可变的或者迭代的，对于“明显优”的走步，可以不考虑其他结点，甚至对他们给予“多搜索几层”的奖励。对于国际象棋来说，**空招数(Null-move)**启发式和无效剪枝也是非常有效的。

小测验

1. 为什么说MINIMAX可能会进行不必要的计算？解释α值、β值和$\alpha-\beta$剪枝规则。在MINIMAX程序中如何加入这个剪枝？
2. $\alpha-\beta$ 剪枝在何时是最优的？此时分支因子和检查的结点数以及相同时间内检查的步数有何变化？
3. 实用的对弈程序通常还会加哪些优化？

4.7 本章小结

本章介绍了一定数量的经典问题，通过这些题目展示算法设计方法。一方面希望读者掌握这些经典问题的算法，另一方面希望读者通过这些具体的问题体会抽象意义下算法设计的原则、方法和技巧。

递归是本章介绍的第一个思想，也是最重要的思想。事实上，本章的动态规划和后面将要介绍的回溯法都是基于这样的思想。递归是一种“问题转化”的思考方式，它把问题转化为本质相同的其他实例。为了理解递归，读者需要区分问题和问题实例，把算法看成是带有参数的过程，那么递归算法就是自己调用自己的算法。分析递归调用时常常画出一个递归调用的树，树中的结点表示递归调用的参数。在本章的动态规划的分析和后面回溯法的分析中，都用到了这样的树。

分治法是递归思想的典型应用，它往往可以用更短的时间解决问题。本章举的各种类似，包括最大最小值、二分查找、归并排序和线性时间选择算法将在下一章被证明是理论最优算法。分治法的几个有趣例子，包括快速整数运算和矩阵乘法虽然不实用，但是从理论上告诉我们：最直观的方法不一定是最好的。分治法部分非常强调算法分析，理论工具是主定理和递归树，二者希望读者熟练掌握。

贪心法理论性并不是很强，所以这部分更多的是给出实际的例子。本书提到的几个区间上的问题算法很直观，正确性证明方法不止本书提到的一种。Huffman编码是一个非常经典的算法，而且被实际用来进行文件压缩。本书在证明Huffman编码正确性时引入了最优子结构性质和贪心选择性质，二者都是相当重要的，可以作为证明贪心法正确的一般框架。贪心法的最后介绍了两个截然不同的调度问题，推导了Johnson算法并简单介绍了矩阵胚理论。

动态规划算法广泛的应用在优化问题中，本章只对它做了最简单的介绍。首先通过数字三角形问题引入了状态定义和状态转移方程，说明了动态规划的优势：重叠子问题，然后比较了三种计算方法。接下来，通过考察此问题的两种变形，深入讨论了无后效性和最优子结构的含义，并介绍了递推法的两种实现：综合法和更新法。接下来的几小节主要讨论了几个经典问题。矩阵链乘问题具有一种特殊的结构，很多类似问题和它等价。解决它的思路是“分割法”；LCS问题的思路是“增量法”，它可以用滚动数组法节约空间；LIS问题和OBST问题都有自己的优化方法，读者在遇到其他问题时都可以尝试这些优化思路：合理组织状态以加快决策；应用四边形不等式证明决策单调性。

本章接下来讨论了人工智能搜索，作为对计算理论中悲观结论的回应。这里的基本出发点是：虽然很多问题是难解的，但是现实要求我们必须寻求尽量好的解决方法。本章最主要的算法思想是："一个一个试"，或者称为穷举。穷举法的思想虽然简单，但是——正如读者已经看到的——它可以扩展出五花八门的算法，其中有的方法有效得令人吃惊。本章属于人工智能中问题求解的研究范畴，有兴趣的读者可以参考经典的人工智能书籍。本章的很多素材都取自Stuart Russell和Peter Norvig《人工智能——一种现代方法（第二版）》的第二部分。

本章从引例八皇后问题开始，首先通过回溯法简单的介绍了状态、解答树等基本概念，然后依次介绍了路径寻找问题、约束满足问题（CSP）和对抗搜索，每个部分都是引人入胜的。

很多智力游戏都可以看作是路径寻找问题，即寻找从初始状态到目标状态的路径。本书首先介绍了路径寻找问题的几个要素，这也成为后面CSP和对抗搜索的定义提供了一份参考。路径寻找问题的搜索算法可以粗略的分为盲目搜索和启发式搜索两类。

盲目搜索类似于图的遍历，只是这里的状态空间图是隐式的，没有必要实现把整个图构造出来。每种盲目搜索算法的特点不一样，双向广度优先搜索理论时间复杂度较低，而IDS在只用很少空间的情况下几乎达到了广度优先搜索的时间效率。

启发式搜索用到了对状态的主观估计，最常见的启发式搜索是A*算法，本章介绍并证明了和A*算法有关的三个结论，前两个分别考虑可接纳h和一致的h。结论一是说：如果h只是可接纳的，A*将无法避免的重复扩展一些结点，但完全性和最优性可以得到满足；结论二是说：如果h是一致的，那么可以轻易处理重复结点：如果老结点在CLOSED表中则直接丢弃新结点；如果在OPEN表中则只保留g较小的一个。结论三是说：在一致的情况下，h越接近$h*$越好。本章还讨论了启发函数的设计：可以通过解决松弛问题来设计h，也可以把多个h合并起来。最后，我们再次强调理论和实际的差别：有时候不得不使用连可采纳性都不满足的h，但这并不代表这样的h实际效果差。

约束满足问题是本章的第二个主题，首先介绍了CSP的严格定义并借助路径寻找问题给它建立了状态空间。接下来深入讨论了回溯法，介绍了前向检查技术FC，然后讨论了两个关键问题：如何选择决策变量；按照何种顺序考虑此变量的各种取值。对于第一个问题，介绍了MRV启发式和度启发式，它们和FC配合时被证明常常是有效的；对于第二个问题，介绍了最少约束值启发式，但需要一定的计算量。

接下来把FC进行扩展，通过实例介绍了约束传播的一般方法，并重点阐述了适用

于二元约束的AC3算法，它保证了弧约束，但无法检查更隐蔽的矛盾。AC3算法很简单，也并不是最有效的，但它为约束传播提供了一种思路。这种思路很容易扩展到高阶约束。最后，本书从约束图出发介绍了CSP问题的结构，讨论了树状二元CSP的有效算法并提出了基于环割集和树分解的两种方法把一般图转化为树。

本书对对抗搜索的介绍并不算多，但是阐述了其中最基本的算法：极大极小过程+$\alpha-\beta$剪枝。极大极小过程采取了理想对手假设，并把解答树扩展为了博弈树，成为博弈算法的基本工具。$\alpha-\beta$剪枝基于一个很简单的事实，非常容易编程实现且效果相当好。虽然本节的内容很简单，但它确实是当今包括Deep Blue在内的实用博弈程序的核心。实用的博弈程序需要考虑很多细节，这里并未多加阐述，有兴趣的读者可以参考相关书籍。

第5章　数学概念与算法

在前两部分的铺垫下，从本章开始，读者讲学到一些特定领域的算法。第一部分是最基本的入门导引，第二部分是问题求解的方法论，这一部分则是分领域介绍知识和算法。这些内容自成体系，同时又都属于问题求解这一大主题中。通过这些专门算法的学习，读者可以全方位感受问题求解的精妙。

5.1　数论基础

本节介绍数论的基本概念和算法。由于我们将用到大整数运算，所以需要把“多项式算法”的定义变一下，即必须是关于$\log n$的多项式而不是n的多项式。一般来说，我们可以认为四则运算是基本操作，但如果涉及到的整数太大，即使是加减法也是需要耗费一定时间的，乘除法更慢。在这样的情况下，我们需要给出**比特运算(bit operation)**数目。给出两个b比特的整数，加减法可以在$O(b)$的时间内完成，且这是最优的; 一般乘法在$O(b^2)$时间内完成。前面介绍过$O(b^{1.59})$的分治算法，而很快我们将学到时间复杂度为$O(b \log b \log\log b)$的算法，但在实用中$O(b^2)$算法已经足够快了，所以在分析里我们认为乘法是$O(b^2)$的。

在本章中，我们同时使用基本运算数和比特运算数来衡量算法的时间效率。

5.1.1　基本概念(1)：整除性、素数、最大公约数

用Z表示整数集合{..., -2, -1, 0, 1, 2, ...}，N表示正整数（自然数）集合{0, 1, 2, ...}。如果存在整数k使得$a = kd$，则称d**整除(divides)** a，记为$d \mid a$。任何整数都整除0。

如果d整除a，我们说a是d的**倍数(multiple)** 。如果$d > 0$，我们说d是a的**约数(divisor)** 。注意到如果$d \mid a$，则$-d \mid a$，因此定义约数必须是正数是合理的。对于任意整数a，它的约数不小于a不大于$|a|$，例如24的约数为1, 2, 3, 4, 6, 8, 12, 24。在因

数分解算法中，我们经常用到“非平凡因子”这个术语，它表示除1和n之外的n的其他因子。

显然a能被1和a整除，如果没有其他整数整除a，则称a是**素数(prime)** 。任何大于1的整数如果不是素数就被称为**合数(composite)** ，它除了1和a之外还有其他约数。1既不是素数也不是合数，0和负数也既不是素数也不是合数。

小于100的素数有25个，即2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97。

模n等价类 小学里学过的带余除法的存在并不是“显然”的，它由以下定理给出：

定理：对于任意整数a和正整数n，存在唯一的整数对(q,r)满足$0 \leq r < n$且$a = qn + r$。

我们把$q = [a/n]$称为**除法(division)** 的**商(quotient)** ，$r = a \bmod n$称为除法的**余数(remainder或residue)** 。显然，$n|a$当且仅当$a \bmod n = 0$。

这样，我们把根据模n的余数把所有整数划分成n个等价类。包含任意整数a的等价类为：$[a]_n = \{a + kn : k \in Z\}$，例如$[3]_7$={..., -11, -4, 3, 10, 17, ...}，它也可以表示为$[-4]_7$或者$[10]_7$。如果两个整数a和b属于同一个等价类，即$a \in [b]_n$，我们说a和b关于模n同余，记为$a \equiv b (\bmod n)$。我们把所有等价类的集合记为Z_n=$\{[a]_n : 0 \leq a \leq n-1\}$，或者简记为$Z_n$=$\{0,1,...,n\text{-}1\}$，二者显然是等价的。读者应当明白$Z_n$的每个元素是一个等价类而不是一个数，当考虑$[-1]_n$时要能立刻明白它实际上也是$[n\text{-}1]_n$

最大公约数 如果d既是a的约数也是b的约数，则称d是a和b的公约数。例如30的约数为1, 2, 3, 5, 6, 10, 15, 30，因此24和30的公约数为1, 2, 3, 6。只要a和b不同时为0，它们的公约数中一定有一个最大值，称它为a和b的**最大公约数(greatest common divisor)** ，记为$gcd(a,b)$只要a和b不同时为0，则$1 \leq gcd(a,b) \leq \min(|a|, |b|)$，它显然满足以下性质：

$$\begin{aligned} &gcd(a,b) = gcd(b,a) = gcd(-a,b) = gcd(|a|,|b|) \\ &gcd(a,0) = |a| \\ &gcd(a,ka) = |a| \text{对于任意整数k} \end{aligned} \tag{5.1}$$

下面的定理从另外一个角度刻画了最大公约数，它在理论和实际应用中都是重要的：

定理：设a和b为不同时为0的整数，则$gcd(a,b)$是集合$\{ax + by : x, y \in Z\}$中的最小正整数。这里的集合表示a和b的**线性组合(linear combination)** 。

证明：设这个最小正整数为s，可以表示为$s = ax + by(x, y \in Z)$。令$q = \lfloor a/s \rfloor$，则$a \bmod s = a - qs = a - q(ax + by) = a(1 - qx) + b(-qy)$

因此$a \bmod s$也是a和b的线性组合，而且比s还小（$a \bmod s < s$）。由于我们设s是最小的正整数，只能有$a \bmod s = 0$，即$s|a$。同理，$s|b$，因此s是a和b的公约数，故$gcd(a, b) \geq s$（因为gcd是最大的）。

注意到等式$s = ax + by$的右边是$gcd(a, b)$的倍数，因此左边也必须是$gcd(a, b)$的倍数，因此$gcd(a, b) \leq s$。综合$gcd(a, b) \geq s$得：$gcd(a, b) = s$。

下面的三个推论是显然的，但很有用。

推论1：a和b的所有公约数都是$gcd(a, b)$的约数。

证明：由于$gcd(a, b) = ax + by$，任何一个公约数整除等式右边，必能整除等式左边。

推论2：对于非负整数n，有$gcd(an, bn) = n \cdots gcd(a, b)$

证明：$n = 0$时显然成立。否则考虑集合$\{anx + bny\}$和$\{ax + by\}$的关系即可。

推论3：如果$n|ab$且$gcd(a, n) = 1$，则$n|b$

证明：由于$gcd(a, n) = 1$，则存在$x, y \in Z$使得$ax + ny = 1$。两边乘以b得$abx + nby = b$，n显然能整除左边，因此也必能整除右边。

互素。如果a和b的最大公约数为1，则称a和b**互素(relatively prime)**。例如8和15是互素的，虽然8和15都不是素数。如果k个整数每两个的最大公约数都是1，则称它们**两两互素(pairwise relatively prime)**。如果a和p互素且b和p互素则ab和p互素，即：

定理：对于整数a, b, p，若$gcd(a, p) = gcd(b, p) = 1$，则$gcd(ab, p) = 1$

证明：根据题设，存在x, y, x'和y'满足$ax + py = 1$，$bx' + py' = 1$，两式相乘并整理，得$ab(xx') + p(ybx' + y'ax + pyy') = 1$，因此1是$ab$和$p$的线性组合，它显然是最小的，因此$gcd(ab, p) = 1$。

唯一分解定理。任何一个合数a有唯一的方式写成如下形式：

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$

其中p_i是素数($p_1 < p_2 < \ldots < p_r$，e_i是正整数。例如$6000 = 2^4 \cdots 3 \cdots 5^3$。这里不给出证明，但请读者注意：这是一个定理而不是“显然成立”的公理。根据唯一分解定理，如果把两个数a和b写成如下的形式：

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$
$$b = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$

其中指数可以为0（这样可以确保素数是对应相同的），则最大公约数$gcd(a,b)$和最小公倍数（a和b公共倍数中最小的一个）$lcm(a,b)$分别满足：

$$\gcd(a,b) = p_1^{\min(e_1,f_1)} p_2^{\min(e_2,f_2)} \cdots p_r^{\min(e_r,f_r)}$$
$$lcm(a,b) = p_1^{\max(e_1,f_1)} p_2^{\max(e_2,f_2)} \cdots p_r^{\max(e_r,f_r)}$$

因此有了分解式很容易计算出gcd和lcm（从上式还很容易看出$gcd(a,b) \cdot lcm(a,b) = ab$）。可惜这个定理并没有告诉我们如何把一个合数进行分解。后面我们将会看到：分解一个合数是困难的。

小测验

1. 解释以下名词：整除、倍数、约数、素数、合数。1是素数吗？1是合数吗？

2. 除法、商和模n等价类是如何定义的？解释$[-1]_n$的含义。

3. 什么是两个整数的最大公约数？它是此二数的线性组合有什么关系？什么是互素和两两互素？什么是唯一分解定理？

5.1.2 基本概念(2)：Z_n和Z_n^*群

如果把整数的加减法和乘法的结果替换成{0, 1, ..., n-1}中在同一个等价类中的元素，则这样的运算被称模算术。例如$4 + 5 = 94 - 5 = -14 \cdot 5 = 20$，而在模6下$4 + 5 = 34 - 5 = 54 \cdot 5 = 2$。利用群论的知识，我们可以对模算术进行更为严密和深入的讨论。

有限群(finite group) 群(S, $\odot$)是一个集合S和定义在S上的二元运算组成的。它满足如下性质：

封闭性(closure) 对于任意$a, b \in S$，有$a \odot b \in S$。

单位元(identity) 存在元素$e \in S$，使得对于任意$a \in S$ 都有$e \odot a = a \odot e = a$。我们称$e$为单位元。

结合性(associativity) 对于任意$a, b, c \in S$有$(a \odot b) \odot c = a \odot (b \odot c)$。

逆元(inverses) 对于任意$a \in S$，存在唯一元素$b \in S$，使得$a \odot b = b \odot a =$

e，称b为a的逆。由逆元的存在性很容易证明：群满足左右消去律。

如果这个群还满足交换律$a \odot b$=$b \odot a$，则称它为**交换群** 或**Abel群(abelian group)**。如果S是有限集，则称它为有限群。

最简单的例子就是大家熟知的群$(Z, +)$，其中Z是整数集，$+$是整数加法。封闭性和结合性是显然的，0是单位元，a的逆元是$-a$。根据定义，这个群还是一个Abel群。

模算术中的加法和乘法分别可以构成一个群，下面分述之。

群$(Z_n, +_n)$ 把一个等价类[a]$_n$看成一个元素，则可以定义等价类加法[a]$_n$+[b]$_n$ = [a+b]$_n$。如果用{0, 1, ..., n-1}来代替各个等价类，则可以把等价类加法理解为两个0∼n-1的整数相加后对n取模。

定理：(Z_n, $+n$)是一个有限Abel群。

证明是根据定义一条一条验证，并不困难但稍显罗嗦，因此略去。

群$(Z_n^*, \cdot_n)$ 类似地，可以定义等价类乘法$[a]_n \cdot [b]_n = [ab]_n$。可问题产生了：($\mathrm{Z}_n$, $\cdot_n$)不是一个群！封闭性、结合性都没有问题，单位元也能找到（就是1），可是并不是所有元素都存在逆。事实上，0就不存在逆，因为对于任何元素a，$0 \times a \bmod n = 0$而不是1。

既然(Z_n, $\cdot_n$)并不是群，我们只能退而求其次，在Z_n中选一些元素，使得它们在$\cdot_n$下构成群。考虑集合

$$Z*_n = \{[a]_n \in Z_n : gcd(a, n) = 1\}$$

则我们有：

定理：$(Z_n^*, \cdot_n)$是一个有限Abel群。

定理的证明需要用到后面讲到的EXTENDED-EUCLID算法，我们把它放到后面去，在证明逆存在的同时给出求逆的有效算法。注意这个集合的定义是没有问题的，因为如果$gcd(a, n) = 1$，等价类$[a]_n$中的其他元素$a + kn$也满足$gcd(a + kn, n) = 1$。

为了简单起见，在今后的讨论中，我们用等价类的代表元素直接代替等价类，而用+和可省略的·表示等价类加法和乘法，因此$ax \equiv b (\mathrm{mod} n)$和$[a]_n \cdot_n [x]_n = [b]_n$是等价的。当没有歧异的时候，可以只用S来代表群$(S, \odot)$，例如我们将用$Z_n$和$Z_n^*$分别表示$(Z_n, +_n)$和$(Z_n^*, \cdot_n)$

a关于乘法的逆通常用$(a^{-1} \bmod n)$来表示，而除法定义为$a/b \equiv ab^{-1}(\mathrm{mod}n)$。例如在$Z_{15}^*$中，我们有$7 * 13 \equiv 91 \equiv 1(\mathrm{mod}15)$，因此$7^{-1} \equiv 13(\mathrm{mod}15)$，$4/7 \equiv 4 * 13 \equiv 7(\mathrm{mod}15)$。

显然Z_n的大小为n，那么Z_n^*有多大呢？它应该等于比n小且于n互素的数的个数。我们把它记为$\phi(n)$，称为**欧拉函数(Euler's phi function)** 。稍后我们会讨论$\phi(n)$的性质和计算方法。一个显然的结论是：对于$n > 1$，当且仅当n为素数时$\phi(n) = n - 1$。

子群(subgroup) 对于群$(S, \odot)$，如果S的某个子集S'满足$(S', \odot)$也是群，那么称$(S', \odot)$是$(S, \odot)$的子群。例如所有偶数关于整数加法构成的群是$(Z, +)$的子群。关于子群有两个很有用的定理：

定理一：对于有限群$(S, \odot)$，如果S的子集S'关于运算封闭，则$(S', \odot)$是$(S, \odot)$的子群。

定理二(Lagrange定理)：如果有限群$(S', \odot)$是有限群$(S, \odot)$的子群，则$|S'|$是$|S|$的约数。

定理一提供了一个证明子群的好办法（只需要验证封闭而不用管其他条目），而定理二揭示了一个有限群及其子群的大小关系。两个定理这里都不证明，有兴趣的读者可以参考抽象代数的相关书籍。

生成子群(generated subgroup) 下面我们讨论一种特殊的子群，即一个元素a的生成子群。定义

$$a^{(k)} = a \odot a \odot a ... \odot a(\text{共k个a}) \tag{5.2}$$

例如在Z_6中，对于$a = 2$有：$a^{(1)}, a^{(2)}, \ldots$是$2, 4, 0, 2, 4, 0, 2, 4, 0, \ldots$。特别地，在$Z_n$中有$a^{(k)} = ka \bmod n$，而$Z_n^*$中$a^{(k)} = a^k \bmod n$。

a生成的子群(subgroup generated by a)被定义为：

$$\langle a \rangle = \left\{ a^{(k)} : k \geq 1 \right\}$$

它显然满足封闭性，因此它确实是子群。包含我们说a生成$< a >$，或称a为$< a >$的**生成元(generator)** 。由于S是有限集，$< a >$是S的一个有限子集。它可能包含S中的所有元素，也可能只包含部分元素。例如在Z_6中，考虑把各个元素作为生成子的情形：

$< 0 >= \{0\}$，序列为$0 \to 0 \to 0 \to \ldots$
$< 1 >= \{0,1,2,3,4,5\}$，序列为$1 \to 2 \to 3 \to 4 \to 5 \to \mathbf{0} \to 1 \to 2 \to \ldots$
$< 2 >= \{0,2,4\}$，序列为$2 \to 4 \to \mathbf{0} \to 2 \to 4 \to \ldots$

而在Z_7^*中，有

$< 1 >= \{1\}$，序列为$1 \to 1 \to 1 \to \ldots$
$< 2 >= \{1,2,4\}$，序列为$1 \to 2 \to 4 \to \mathbf{1} \to 2 \to 4 \to \ldots$
$< 3 >= \{1,2,3,4,5,6\}$，序列为$3 \to 2 \to 6 \to 4 \to 5 \to \mathbf{1} \to 3 \to 2 \to \ldots$

可以看出，元素a“总有一天”会变成单位元e，然后开始循环。我们把满足$a^{(t)} = e$的最小正整数t称为a的**阶(order)**，记为$ord(a)$。根据刚才的观察，$ord(a) = | < a > |$。这个结论是正确的，但推导不严密。是否可能a永远都变不到e呢？如果变不到，那么只有一种可能：a在中途开始循环，即存在$i < j$使得$a^{(i)} = a^{(j)}$，而根据结合律$a^{(j)} = a^{(i)} \odot a^{(j-i)}$，两边左消去$a^{(i)}$得$a^{(j-i)} = e$。这说明$a$最终是会变成$e$的，矛盾！注意：这个例子只对有限群成立。另外，证明的过程还提示我们：一个元素a和它的逆的阶是相同的（把循环倒过来）。

现在，我们可以放心大胆的说：序列$a^{(i)}$是周期性的，周期$t = ord(a)$。因此$a^{(i)} = a^{(j)}$当且仅当$i \equiv j(\mathrm{mod}t)$。和用$\{0,1,...,n-1\}$代替等价类一样的方法，把$\mathrm{a}^{(0)}$定义为$e$，$a^{(i)}$定义为$a^{(i \bmod t)}$，其中$t = ord(a)$。

根据Lagrange定理，$ord(a)|\ \ |S|$，故$|S| \equiv 0(\mathrm{mod}t)$，因此我们有：

定理：$(S, \odot)$为一个有限群，单位元为e，则对于任意$a \in S$，$a^{(|S|)} = e$

这个定理在Z_n^*中就是著名的欧拉定理。

小测验

1. 什么是群？它需要满足哪四个条件？什么是Abel群？
2. 解释群Z_n和$Z*_n$。它们的大小如何？为什么Z_n关于模乘法不构成群？
3. 什么子群？什么是元素a的生成子群？关于子群和生成子群有哪些重要结论？

5.1.3 模方程(1)：Euclid算法和中国剩余定理

介绍完基础知识后，从本节开始每节讨论数论中的不同主题。本节介绍求解线性

同余方程和方程组的算法。在介绍算法之前，先给出扩展的Euclid算法和线性不定方程的求法。有了这些基本工具，模线性方程就变得容易求解了。

Euclid算法 前面提到过：给出唯一分解式后很容易求出两个数的最大公约数。可惜进行唯一分解是困难的。求最大公约数的euclid算法是最经典的数论算法之一，它利用了$gcd(a,b) = gcd(b, a \bmod b)$。这个性质可以直接转化为递归或递推两种实现，如下：

```
int gcd(int a, int b) //递归实现
{
    return (b? gcd(b, a%b) : a);
}

int gcd(int a, int b) //非递归实现
{
    while(b){ int t = a % b; a = b; b = t; }
    return a;
}
```

它的时间复杂度为$\mathrm{O}(\log b)$。更进一步地，次数$step \leq 4.785 \lg N + 1.6723$。其中$lgN$为常用对数，且最坏情况是计算$gcd(F_n, F_{n-1})$。这个结论被称为Lamé定理，平均次数为$12ln2/\pi^2 lgN = 0.843lgN$。

如果除法的开销很大，有一种二进制算法可以只用减法、移位和判奇偶。

二进制算法 $gcd(a,a) = a$, 当$a > b$时，分情况讨论：

a和b均为偶数，$gcd(a,b) = 2 \cdot gcd(a/2, b/2)$

a为偶数b为奇数，$gcd(a,b) = gcd(a/2, b)$

a和b均为奇数，$gcd(a,b) = gcd(a-b, b)$

代码如下。虽然迭代次数增多，但是实际的时间效率和euclid算法相差无几。

```
int gcd(int a, int b)
{
    int t = 1, c, d;
    while(a != b){
        if(a < b) swap(a, b);
        if(!(a&1)){ a >>= 1; c = 1; }else c=0;
        if(!(b&1)){ b >>= 1; d = 1; }else d=0;
        if(c && d) t <<= 1;
        else if(!c && !d) a -= b;
    }
    return t * a;
}
```

线性不定方程 线性不定方程是指$ax+by=c$。给出整数a, b, c，先求出一组解，然后讨论如何表示通解（所有解）。首先设$gcd(a,b)=d$。如果c不是d的倍数，那么方程无解，因为左边是d的倍数，而右边不是。在有解的情况，设$c=c'\times d$，则可以先求出$ax+by=d$的一组(x_0,y_0)，则$ax_0+by_0=d$，两边乘以c'得：$a(c'x_0)+b(c'y_0)=c$，因此$(c'x_0,c'y_0)$是原方程的一组解。换句话说，核心问题是：求$ax+by=gcd(a,b)$的解。

下面的扩展euclid算法求出了这样一组解。

```
void gcd(int a, int b, int& d, int& x, int& y)
{
    if(!b){ d = a; x = 1; y = 0; }
    else{
        gcd(b, a%b, d, y, x);  // (**)
        y -= x*(a/b);
    }
}
```

用数学归纳法可以证明它的正确性。归纳基础：$b=0$时，$gcd(a,0)=a$，取$x=1, y=0$，有$ax+by=a\cdot 1+0\cdot 0=a=gcd(a,0)$，成立。

先假设i次迭代以后，(**)行得到了正确的结果，即返回值y'和x'满足：

$$by'+(a \bmod b)x'=gcd(b,a \bmod b)=gcd(a,b)=d \tag{5.3}$$

经过步骤$y-=x'*(a/b)$后，新的数对(x,y)满足：

$$ax+by=ax'+b(y'-x'(a/b))=ax'+by'-bx'(a/b)=s \tag{5.4}$$

这里的a/b是整除运算，它等于$(a-a \bmod b)/b$，因此

$$s=ax'+by'-bx'(a-a \bmod b)/b=ax'+by'-ax'+(a \bmod b)x'=by'+(a \bmod b)x'=d \tag{5.5}$$

其中前两步是乘除法和加减法相抵消，而最后一步用到了归纳假设。由数学归纳法知：任意步骤以后，函数都返回正确的结果，且时间复杂度和euclid算法一样。

通解 假设现在已经求出一组解，如何得到所有解？假设有两组解(x_1,y_1)和(x_2,y_2)，则有$ax_1+by_1=ax_2+by_2=c$，因此$a(x_1-x_2)=b(y_2-y_1)$。消去$d=gcd(a,b)$后得$a'(x_1-x_2)=b'(y_2-y_1)$。由于此时a'和b'互素，因此x_1-x_2一定是b'的整数倍，设为kb'，计算得$y_2-y_1=ka'$。因此：

给一组特解(x, y)，通解为$(x + kb', y - ka')$，其中k取任意整数。

多变元的情形 事实上，刚才的结论可以推广到多变元的情况，如果所有系数的最大公约数d（利用$gcd(a, b, c) = gcd(gcd(a, b), c)$）不是$c$的约数，则无解；否则有无数解。

前面说过，$a_n x_n + a_{n+1} x_{n+1}$的值域为$\gcd(a_n, a_{n+1})$的整数倍。因此，对于任意整数$y$，方程$a_n x_n + a_{n+1} x_{n+1}$=$\gcd(a_n, a_{n+1})y$有无数解。这样方程减少了一个字母，变成了：

$$a_1 x_1 + a_2 x_2 + \ldots + a_{n-1} x_{n-1} + gcd(a_n, a_{n+1})y = c \tag{5.6}$$

可以理解为：先求出解$(x_1, x_2, \ldots, x_{n-1}, y)$，然后再根据$y$求出$x_n$和$x_{n+1}$。

单变元的模线性方程 既然是单变元的，方程应该形如$ax \equiv b(\mathrm{mod}n)$。在讨论方程组之前，先讨论每个单个方程$ax \equiv b(\mathrm{mod}n)$。

这个式子相当于存在y使得$ax - b = ny$，移项得$ax - ny = b$，是一个标准的线性不定方程。模方程和一般方程不一样的地方在于：虽然解有无穷多组，但是剩余系解是有限的。换句话说，如果$x = 1$是解，那么显然$x = n + 1$也是解。由于1和$n + 1$属于同一个剩余系，因此我们把它们看作同一组解。以后没有说明的情况下，我们只考虑$0 \sim n - 1$之间的解

逆的求法 $ax \equiv 1(\mathrm{mod}n)$的解$x$称为$a$关于模$n$的逆。显然，逆不一定存在，因为转化后的方程$ax - ny = 1$有解当且仅当$a$和$n$互素（这也可以由前面的$Z_n^*$群的由来加以说明）。在有解的情况下，下面的代码可以用来求逆。返回–1代表逆不存在，否则返回一个$0 \sim n - 1$之间的逆。下面的mod可以对负数进行调整，在很多算法中都要使用。

```
int mod(int x, int n)
{
    return (x%n+n)%n;
}

int inv(int a, int n)
{
    int d, x, y;
    gcd(a, n, d, x, y);
    return d == 1 ? mod(x, n) : -1;
}
```

如果逆存在，它一定是唯一的。因为如果有两个剩余系x和y满足$ax \equiv 1(\mathrm{mod}$

$m)$和$ay \equiv 1(\mathrm{mod}m)$，则$a(x-y)$是$m$的倍数。由于$a$和$m$互素，因此$x-y$是$m$的倍数，说明$x$和$y$其实是在同一个剩余系中。这个性质对于其他群也成立。事实上，它是群的通性。

一般方程的求法 有了求逆的过程，可以把一般方程向它转化。前面已经把方程变为了$ax-ny=b$了，设$gcd(a,n)=d$，则b不是d的倍数时无解，否则两边同时除以d，得到：$a'x-n'y=b'$，即$a'x \equiv b'(\mathrm{mod}n')$

由于此时a'和n'已经互素，因此只需要左乘a'对n'的逆，则解为$x \equiv (a')^{-1} * b'(\mathrm{mod}\ n')$

问题并没有结束。这里的唯一解表示成模n'的剩余系，然而在很多情况下，需要还原为模n的剩余系。还原的方法也很简单：一共有d个解。显然第一个解为$(a')^{-1} * b'$，以后每个解等于前一个解加上n'，即n/d。代码如下：

```
void linear_mod_equation(int a, int b, int n, int sol[])
{
    gcd(a, n, d, x, y);
    if(b%d) d = 0;          // no solution
    else
    {
        // ax+ny=, dso a'x+n'y=1, x is the inverse of a' mod n'
        sol[0] = x * (b/d) % n;  // first solution
        for(i = 1; i < d; i++)
            sol[i] = (sol[i-1] + n/d) % n;
    }
}
```

方程组的情形 下面考虑方程组的情形。由于单独的方程要么无解(b不是$gcd(a,n)$的倍数时)要么可以转化成$x \equiv x_0(\mathrm{mod}n')$（其中$x_0 = (a')^{-1} * b'$），因此只需转化成多个形如$x \equiv x_0(\mathrm{mod}m)$ 的方程联立构成的方程组。在解方程前，还可以继续把方程进行变形。如果a和b互素，则$x \equiv x_0(\mathrm{mod}ab)$可以拆为$x \equiv x_0 \bmod a(\mathrm{mod}a)$和$x \equiv x_0 \bmod b(\mathrm{mod}b)$。这是因为$x \equiv x_0(\mathrm{mod}ab)$等价于$x = x_0 + kab$。两边同时对$a$和$b$取模，得$x \equiv x_0 \bmod a(\mathrm{mod}a)$和$x \equiv x_0 \bmod b(\mathrm{mod}b)$。反过来，当$a$和$b$互素时，两个方程组联立后在$[0 \sim ab)$内有唯一解（不证）。因此所有方程都可以变为模为素数的幂的形式。

中国剩余定理 由刚才的变形，我们只需考虑所有m两两互素的情况。令M为所有m的乘积，$w_i = M/m_i$，则$(w_i, m_i) = 1$，因此存在p_i和q_i使得$w_i p_i + m_i q_i = 1$成立。令e_i为等式的前一半，即$w_i p_i$。等式两边模m_i可知$e_i \bmod m_i = 1$。

可以验证$e_1 a_1 + e_2 a_2 + \ldots + e_n a_n$是问题的解，因为它对$m_i$取模时，所有其他$e_j$模都

等于0（因为w_j包含因子m_i），而e_i的模为1。代码如下：

```
int china(int n, int a[], int m[])
{
  int M = 1, dummy;
  for(i = 0; i < n; i++) M*=m[i];
  for(i = 0; i < n; i++){
    w = M / m[i];
    gcd(m[i], w, dummy, dummy, y); // don't care about others
    x = (x + y*w*a[i]) % M;        // accumulate e*的和a
  }
  return (n+x%M)%M; // adjust to [0, M-1]
}
```

小测验

1. 叙述扩展的euclid算法和二进制算法。如何用euclid算法求解线性不定方程的特解？通解如何得到？
2. 模n的逆何时存在？如何求逆？如何用逆来求解单变元模线性方程？
3. 叙述中国剩余定理？如何求解单变元模线性方程组？

5.1.4 模方程(2)：指数和原根

上一节我们讨论了a的倍数模n的结果，即$0a, 1a, 2a, \ldots$模n；我们同样可以考虑a的幂，即$a^0, a^1, a^2, a^3 \ldots$模$n$。还记得前面引入的生成子群和阶的概念吗？这里有一份$3^i \bmod 7$的表格：

i	**0**	**1**	**2**	**3**	**4**	**5**
$3^i \bmod 7$	1	3	2	6	4	5
i	**6**	**7**	**8**	**9**	**10**	**11**
$3^i \bmod 7$	1	3	2	6	4	5

表 5.1: $3^i \bmod 7$的值

注意到从6开始有循环，因此在Z_7^*中$ord(3) = 6$，故$3^{11} \bmod 7 = 3^{11 \bmod 6} \bmod 7 = 5$。由于本节讨论乘法，因此记$Z_n^*$中$a$的指数为$ord_n(a)$。这样，$ord_7(3) = 6$。回忆我们在该小节中介绍的最后一个定理：$(S, \odot)$为一个有限群，单位元为$e$，则对于任意$a \in S$，$a^{(|S|)} = e$（用Lagrange定理容易证明）。在$Z_n^*$中，这个定理和它的一个特殊情况为：

欧拉定理：给定整数$n > 1$，对于任意$a \in Z_n^*$，$a^{\phi(n)} \equiv 1 (\bmod n)$

费马小定理：若p是素数，则对任意$a \in Z_p^*$，$a^{p-1} \equiv 1(\mathrm{mod}n)$

它们事实上已经被我们证明了。根据欧拉定理，实际上我们得到了计算逆的另一个方法。若$ax \equiv 1(\mathrm{mod}n)$，两边乘以$a^{\phi(n)-1}$，得$x \equiv a^{\phi(n)-1}(\mathrm{mod}n)$，问题转化为了求$a^p \bmod n$。

幂取模问题 刚才的方法需要对非负整数a, n, p计算$a^n \bmod p$的值。一个显然的方法是循环n次，不断算出$a \bmod p, a^2 \bmod p \ldots$，时间复杂度为$\mathrm{O}(n)$。一个改进是：把$n$写成二进制$n = (n_k n_{k-1} \ldots n_0)_2$，设$d_i = a^{2^i} \bmod p$，则$a^n \bmod p$就是把$n_i$为1的那些$d^i$全部乘起来，例如：$11 = (1011)_2$，因此$a^{11} = a^8 \cdot a^2 \cdot a^1$。由于$a^8 = (a^4)^2, a^4 = (a^2)^2, a^2 = (a^1)^2$，可以在从右往左扫描$n$的二进制位的同时递推出所需要的$d_i$。

从右往左递推，每次如果n的最低位为1，则把当前d值乘到最终答案里，然后递推$d_i = (d_{i-1})^2$，并把n往右移动一位，直到为0。需要注意的是递推d_i时，乘法可能溢出！即使是使用ll类型，两个ll类似的数相乘还是可能溢出？怎么办呢？

Head算法 这里介绍一个不会超出字长的算法，它由Head于1980年提出。假设计算机机器字大小为w而模n满足$n < w/2$。令$T = \lfloor \sqrt{n} + 1/2 \rfloor t = T^2 - n$，则显然$|t| \leq T$。任何数$x$都可以表示为$x = aT + b$，其中$0 \leq a \leq T, 0 \leq b < T$。注意$a$可以等于$T$，$b$不可以。这一步取$a = x/T, b = x \bmod T$即可。

为了计算$xy \bmod n$，我们先把x和y写成$x = aT + b, y = cT + d$，再把ac写成$ac = eT + f$。

令$z = (ad + bc) \bmod nv = (z + et) \bmod n$。把$v$写成$v = gT + h$，则

$$xy \equiv hT + (f + g)t + bd(\mathrm{mod}n) \tag{5.7}$$

如果按照$j = \lfloor (f + g)t \rfloor \bmod n, k = (j + bd) \bmod n, ans = (hT + k) \bmod n$的顺序计算，最后$ans$就是$xy \bmod n$的值且每步运算不会超过计算机字长。程序如下：

```
ll mul_mod(ll x, ll y, ll n){
        ll T = floor(sqrt(n) + 0.5);
        ll t = T*T-n;

        ll a = x/T; ll b = x%T;
        ll c = y/T; ll d = y%T;
        ll e = a*c/T; ll f = a*c%T;

        ll v = ((a*d + b*c)%n + e*t) % n;
        ll g = v/T; ll h = v%T;
```

```
        ll ans = (((f + g)*t%n + b*d)%n + h*T)%n;
        while(ans < 0) ans += n;
        return ans;
}
```

最后一步调整是因为刚计算出的ans可能是负数。这样，可以写出幂取模的过程：

```
ll pow_mod(ll a, ll n, ll p)
{
        ll ans = 1, d = a % p;
        do{
                if(n & 1) ans = mul_mod(ans, d, p);
                d = mul_mod(d, d, p);
        }while(n >>= 1);
        return ans;
}
```

原根 在刚才的表中，我们看到了由3生成了整个Z_7^*，即$Z_7^* =< 3 >$，或者说$ord_7(3) = 7 - 1$。考虑一般情况，如果给定一个群Z_n^*，存在一个元素g使得g生成整个Z_n^*，或者等价地，$ord_n(g) = |Z_n^*|$，则称g为Z_n^*的**原根(primitive root)** 。这样，3是模7的原根，但2不是模7的原根（因为在Z_7^*中，$< 2 >= \{1, 2, 4\}$，还有3、5和6无法由2生成）。如果Z_n^*有原根，我们说Z_n^*是循环群。

有原根g是一件很幸福的事情，因为此时任意$a \in Z_n^*$都可以写成g^i的形式。换句话说，$g^1, g^2, \ldots, g^{\phi(n)}$构成了$Z_n^*$。而且这其中还有不少元素也是$n$的原根。

为了说明这个问题，我们先证明下面的结论：

定理：若$ord_n a = t$，则对于正整数u，$ord_n(a^u) = t/gcd(t, u)$

我们先尝试直观理解。考虑用a^u来生成一个群时的情景。显然t次肯定已经有循环了，因为$(a^u)^t \equiv (a^t)^u \equiv 1(\mathrm{mod} n)$。但这时也许已经是好几个循环节了，因此$ord_n(a^u)$应该是$t$的约数。如果$gcd(t, u)$大于1，那么在生成的时候每次得到的$a^i$的$i$都是这个$gcd(t, u)$的倍数，因此实际上原来的循环节中只有$1/gcd(t, u)$被遍历到了，故答案为$t/gcd(t, u)$。

这里举一个例子。前面已经知道$ord_7(3) = 6$，它的生成过程为$\mathbf{1} \to 3 \to \mathbf{2} \to 6 \to \mathbf{4} \to 5 \to \mathbf{1} \to \ldots$。考虑$< 3^2 >$，它的生成过程应该是在3的基础上“隔一个数选一个”得到的，即：$1 \to 2 \to 4 \to 1 \to \ldots$。原来循环节长度为6，这次每一个数跳了一个，因此只选到了3个数，循环节变为了6/2=3。对于一般情形，只有在$gcd(t, u)$的倍数的地方才

会被选到，这些a^i满足：

$$i \equiv 0(modgcd(t,u)) \tag{5.8}$$

而i$\equiv 1,2,3,...(\mathrm{mod}gcd(t,u))$的数都被“抛弃”了。新循环节数为原来的$1/gcd(t,u)$，即$ord_n(a^u) = t/gcd(t,u)$。

作为定理的直接推论，若a是模n的原根，则当且仅当$gcd(u,\phi(n)) = 1$时a^u也是模n的原根。换句话说，对于任意有原根的Z_n^*，不同余的原根个数为$\phi(\phi(n))$个。

很自然地，下一个问题就是：哪些正整数n作为模时才有原根呢？我们有如下定理：

定理：$n > 1$时，使得Z_n^*为循环群的n只有$2,\ 4,\ p^e$或者$2p^e$，其中p为任意奇素数，e为任意正整数。

证明篇幅不短且和本书主题关系不大，故略去。但它是一个很优美的结论，读者应当记住。

离散对数。若g是模n的原根，则对于任意$a \in Z_n^*$，存在z使得$g^z \equiv a(\mathrm{mod}n)$。$z$称为模$n$下以$g$为底$a$的**离散对数(discrete logarithm)** 或**指数(index)** ，记为$ind_{n,g}(a)$。如果模n是固定的，也写作$ind_g(a)$。离散对数有类似于对数的性质，但都在模$\phi(n)$下进行。

显然，若g是模n的原根，则$g^x \equiv g^y(\mathrm{mod}n)$当且仅当$x \equiv y(\mathrm{mod}\phi(n))$（考虑$g$生成$Z_n^*$的过程，使用原根的定义即可），利用这一点可以简化模方程。

例如$6x^{12} \equiv 11(\mathrm{\ mod\ }17)$。由于3是模17的原根（因为$3^8 \equiv -1(\mathrm{\ mod\ }17)$），计算$Z_{17}^*$中所有整数的离散对数（考虑3的生成序列，然后填表），如下

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ind_3a	16	14	1	12	5	15	11	10	2	3	7	13	4	9

表 5.2: Z_{17}^*中所有整数的离散对数

方程两边同时取以17为模3为底的离散对数，则$ind_3(6x^{12}) \equiv ind_311 \equiv 7(\mathrm{mod}\ 16)$（注意模变为了$\phi(17) = 16$）。根据对数运算法则，左边可以化简为$ind_36 + 12 \cdot ind_3x \equiv 15 + 12 \cdot ind_3x(\mathrm{mod}16)$，这样，方程变为$15 + 12 \cdot ind_3x \equiv 7(\mathrm{mod}16)$，即$12 \cdot ind_3x \equiv 7(\mathrm{mod}16)$。根据上一小节介绍的方法，可以计算出$ind_3x \equiv 2,6,10,14(\mathrm{mod}$

16)，因此$x \equiv 3^2, 36, 3^{10}, 3^{14}(mod 17)$，即

$$x \equiv 9, 15, 8, 2(mod 17) \tag{5.9}$$

由于以上每步可逆，因此$6x^{12} \equiv 11(\mathrm{mod}17)$有且只有这四个根。

另一个例子是$7^x \equiv 6(\mathrm{mod}17)$。由于3是模17的原根，两边取17为模3为底的离散对数，则$ind_3(7^x) \equiv ind_3 6 = 15(mod 16)$，化简得$11x \equiv 15(\mathrm{mod}16)$，直接求出$x \equiv 13(mod 16)$。

虽然求出了结果，但是这样的方法并不实用，因为离散对数和因数分解一样，是难以求出的。我们将在后面专门讨论离散对数的算法。

最后，需要说明的是：虽然这里的结论都只针对模有原根的情形，但它们是可以扩展到任意模的，方法如下：

虽然$k \geq 3$时2^k没有原根，但对于任意整数a都存在唯一数对(α, β)，使得$a \equiv (-1)^{\alpha} 5^{\beta}(\mathrm{mod}2^k)$，其中$\alpha = 0$或1，$0 \leq \beta \leq 2^{k-2} - 1$。我们把这个数对$(\alpha, \beta)$称为$a$**关于模$2^k$的指数系统(index system of a modulo 2^k)** 。

更一般地，给定整数n，考虑它的唯一分解式

$$n = 2^{t_0} p_1^{t_1} p_2^{t_2} \cdots p_m^{t_m}$$

设a为和n互素的整数，$i \geq 1$时记r_i为$p_i^{t_i}$的原根$(i \geq 1)$，$\gamma_i = ind_{r_i} a \left(\mathrm{mod}\ \phi\left(p_i^{t_i}\right)\right)$，如果$t_0 \leq 2$则类似定义$r_0$和$\gamma_0$，否则取$(\alpha, \beta)$为$a$关于模$2^k$的指数系统。

这样，我们可以定义a关于模n的指数系统：t_0¡=2时为m+1元组$(\gamma_0,\gamma_1,\gamma_2,...,\gamma_m)$，当$t_0 \geq 3$是为m+2元组$(\alpha,\beta,\gamma_1,\gamma_2,...,\gamma_m)$。这样的指数系统虽然比较复杂，但是和原始定义的指数运算规则是类似的。

大步小步算法 离散对数的计算是困难的，困难程度和因子分解相当。在很多情况下被作为子程序调用，这里只介绍**Shank的大步小步算法(Shank's Baby-Step-Giant-Step Algorithm)** ，它对于不太大的数比直接枚举好得多。

这里只考虑n为素数的情况。由欧拉定理, 只需要检查$x = 0, 1, 2, \ldots, n-1$的情况, 时间复杂度为O(n)。由于n为素数, 只要a不为0, 一定存在逆a^{-1}。

大步小步算法首先检查前m项, 即$a^0, a^1, \ldots, a^{m-1}$模$n$的值是否为$b$, 并把$a^i \bmod n$保存在$e_i$里。求出$a^m$的逆$a^{-m}$。

下面考虑$a^m, a^{m+1}, \ldots, a^{2m-1}$。如果其中有解，相当于存在$i$使得$e_i \cdot a^m \equiv b(\mathrm{mod}\ n)$。两边左乘$v = a^{-m}$得到：$e_i \equiv v \cdot b(\mathrm{mod} n)$，这只需要检查是否有$e_i$等于给定值$v \cdot b \bmod n$。可以先给$e$排序再二分查找。接下来每一轮都只需要做一次二分查找。

需要O(m)的时间计算e数组，O($n/m \cdot \log m$)的时间进行剩下的工作（共n/m轮，每轮$\log m$的时间二分查找）。取$m = \sqrt{n}$，则时间复杂度为O($\sqrt{n} \log n$)。下面的程序使用一个map，x[j]表示满足$e_i = j$的最小i（这样的i可能有多个）。

```
ll log_mod(ll a, ll b, ll n){
        int m, v, e, i;
        map<int,int> x;

        m = (int)ceil(sqrt(n));
        v = inv(pow_mod(a, m, n), n);

        e = 1; x[1] = 0;
        for(i=1; i<m; i++){ e=mul_mod(e, a, n); if(!x.count(e)) x[e]=i; }
        for(i=0; i<m; i++){
                if(x.count(b)) return i*m+x[b];
                b=mul_mod(b,v,n);
        }
}
```

程序中用到了前面介绍的Head算法mul_mod。

小测验

1. 叙述欧拉定理和费马小定理。要用欧拉定理求逆，需要哪两个辅助算法？叙述模乘法的Head算法和幂取模的倍增算法。

2. 什么是原根？哪些模有原根？如果有，不同余的原根有多少个？一个元素a的阶和幂a^u的阶有什么关系？

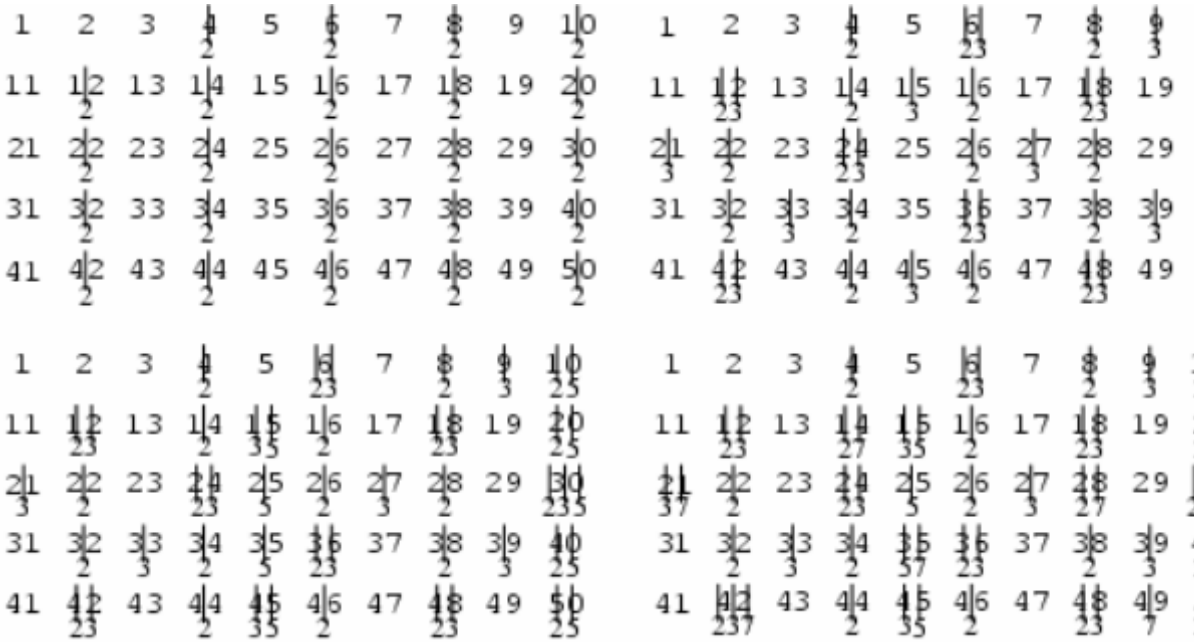
3. 什么是指数？如何把它扩展为原根不存在的模？举例说明如何用指数求解几类特殊的模方程。叙述Shank的大步小步算法。

5.1.5 素数及其判定

和素数有关的问题在数论中非常基本。很多题目需要一开始生成一个素数表，不管是做查表用还是遍历用，都是很方便的。

Eratosthenes筛法 用数组mk_i表示i是否一定为合数。True表示一定为合数，而false表示可能是素数。从小到大检查所有元素，如果发现mk_i为假，则它一定为素数，然后把i的所有倍数的mk_i值都设为真。

下面是一个$n = 50$的例子，得到了不超过50的所有素数。



在得到一个素数p时，具体需要筛去哪些数呢？p的倍数有$2p, 3p, 4p, \ldots$ 但当$k < p$时，kp早就被k的一个素因子被找到时筛掉了，因此只需要筛$p \cdot p, (p+1) \cdot p, \ldots$，直到$n$。

代码如下：

```
void GenPrimes(int n){
  int i, j, k; pc = 0;
  memset(mk, 0, n+1);
  for(i=2,k=4; i<=n; i++, k+=i+i-1)
  if(!mk[i]){
    primes[pc++] = i;
    if(k <= n)
    for(j=k; j<=n; j+=i)
        mk[j] = true;
  }
}
```

时间复杂度 显然每个数至少检查了一次，因此时间复杂度为O(n)+筛的时间复杂度。仅当i为素数的时候，筛了不超过n/i个数，因此总时间复杂度为$n + n/2 + n/3 + n/5 + n/7 + \ldots$，一直加到不超过$n$的最大素数。当$x$为素数时，从2到$x$的所有素数$p$的倒数和为$\ln \ln x + B_1 + o(1)$。其中$B_1$为Mertens常数0.2614972128。因此总时

间约为$n \cdot (1 + \ln\ln n)$，几乎是线性的。事实上，$n = 10^7$可以在约一秒之内出解，而$n = 10^8$遇到了空间问题。

空间复杂度 筛法的致命缺点是空间复杂度高。找到不超过n的所有素数需要长度为n的bool数组，而结果需要$\pi(n)$个整数（不超过n的素数个数）。

素数定理：$\pi(n)$和$n/\ln n$很接近。

下面是$n = 100, 1000, \ldots, 10^9$内素数的个数以及对应的$n/\ln n$。

	$\mathbf{10^2}$	$\mathbf{10^3}$	$\mathbf{10^4}$	$\mathbf{10^5}$	$\mathbf{10^6}$	$\mathbf{10^7}$	$\mathbf{10^8}$	$\mathbf{10^9}$
π**(n)**	25	168	1229	9592	78498	664579	5761455	50847534
n/lnn	22	145	1086	8686	72382	620421	5428681	48254942

表 5.3: 10^9内素数的个数

下面是一些更大的值：

	$\mathbf{10^{10}}$	$\mathbf{10^{11}}$	$\mathbf{10^{12}}$	$\mathbf{10^{13}}$	$\mathbf{10^{14}}$
π**(n)**	455052512	4118054813	37607912018	346065536839	3204941750802
n/lnn	434294482	3948131664	36191206825	334072678387	3102103442166

表 5.4: 10^{14}内素数的个数

可以看出，虽然差距越来越大，但是比值越来越接近1。和后面将要介绍的方法类似，如果只考虑和记录除以6余1和5的那些数，空间可以优化到现在的1/3，代码略。

由刚才的分析，辅助空间是$O(n)$的，但结果保存只需要$O(n/\log n)$。

然而素数表的空间开销很大，在一些只需要判断个别大整数是否为素数的场合不仅空间无法承受，而且是一种时间上的浪费。

素数判定 素数判定是一个基本的问题，它的算法有很多，其中不少算法涉及到高深的理论，程序也不容易编写。这里介绍两类实用中比较好的算法，一类是试除法，一类是概率判定。二者的适用范围不一样，不是在任一种情况下某一种都比另一种好。

试除法 试除法实际在判断素数时顺便找出了它的一个因子，因此它也可以用来分解因数。从2开始从小到大枚举i，如果p能整除i，则失败退出。虽然根据定义，需要枚举到$p-1$，但实际上只需枚举到$\lfloor\sqrt{p}\rfloor$。因为当$a > \sqrt{p}$时，$p/a < a$一定已经被枚举过了。因此当$\sqrt{p}$内没有约数时，p一定是素数。代码如下：

```
bool isprime(int p)
```

```
{
    int i;
    for(i = 2; i * i <= p; i++)
        if(p % i == 0) return false;
    return true;
}
```

如果p不是素数，那么它一定有一个素因子。注意到偶数中只有2是素数，而且除了3之外所有素数除以6的余数均为1或5，因此可以从5开始，交替以2、4为步长增加i。由于计算$\sqrt{p}$时间比较长，而每次计算i^2时间也是一种浪费，可以利用$(i+1)^2 = i^2 + 2i + 1$进行递推。变量j即是保存i^2的值，k是步长。代码如下：

```
bool isprime(int p)
{
    if(p==2||p==3) return true;
    else if(p%2==0||p%3==0) return false;
    for(i=5,j=25,k=4; j<=p; i+=(k=6-k), j=i+i-1)
        if(!p%i) return false;
    return true;
}
```

代码风格并不好，但是由于这是一段很基础的代码，这里首要考虑了紧凑性。读者可以很容易把它改写成风格良好但代码稍长的形式。虽然代码变长，但这样的优化以后，速度明显提升。读者可以根据实际应用选择更合适的书写方式。

另一种改进是只考虑比$\sqrt{p}$小的所有素数。由于需要读写素数表，因此稍微复杂一些且读素数表的时间比较长。优化后速度有所提升，但效果并不是特别明显。

概率测试 概率测试的特点是：只根据素数的条件进行判定，认定它是合数却不知道它的因子。最基本的概率测试是直接基于前面证明过的：

Fermat小定理：如果n是素数，则对于所有不是n倍数的a，有$a^{n-1} \equiv 1 (\mathrm{mod} n)$

由此定理，假设$a = 2$时不满足此式，可以确定的说n是合数。然而如果$a = 2$时成立，并不能确定n是素数，因为$2^{340} \bmod 341 = 1$，而$341 = 11 \cdot 31$不是素数。$a = 3$也有反例，如$3^{90} \bmod 91 = 1$，而$91 = 7 \cdot 13$不是素数。

最糟糕的情况是：n是合数，但$1 \sim n-1$中所有与n互素的a全部满足上式，这样我们不管选哪个a都将得到错误的结果。我们称这样的数为Carmichael数。不与n互素的数a显然不满足上式，因为同余符号右边一定是$gcd(n, a)$的倍数。因此Carmichael数是最糟糕的情况。

需要说明的是：Carmichael数有无穷多个，10^8范围内有255个，最小的几个是$561 = 3 \cdot 11 \cdot 17$$1105 = 5 \cdot 13 \cdot 17, 1729 = 7 \cdot 13 \cdot 19, 2465 = 5 \cdot 17 \cdot 29$。

直接使用Fermat小定理，可以得到如下的算法（n不小于5的正奇数）：

```
bool isprime(int n)
{
    int a = random(2, n-2);
    if (pow_mod(a, n-1, n) == 1) return true; // very likely
    else return false; // must be
}
```

顺便说一句，程序里应加入如下判断：如果a是p的倍数则直接输出0。

直接应用Fermat小定理的测试不一定是对的，它的失败概率（是合数却报告素数）有多大呢？回忆群Z_n^*的特点，定义

$$F_n = \{a \in Z_n^* | a^{n-1} \equiv 1 (\mathrm{mod} n)\} \tag{5.10}$$

则如果n是素数或Carmichael数，$F_n = Z_n^*$。如果不是，则二者不相等。容易验证F_n在模n乘法下也形成一个群，因此它是Z_n的真子群。由Lagrange定理，$|F_n|$是$|Z_n|$的约数。因此F_n的元素最多是Z_n的一半。这就证明了：

定理：如果n不是Carmichael数，则用Fermat定理判定素数的算法测出合数的概率至少是1/2

Miller-Rabin测试 可惜Carmichael数有无穷多，我们需要继续变形。设n为大于等于5的奇素数，写成$n-1 = 2^r s$。由于$n-1$是偶数，因此$r \geq 1$。由Fermat定理，序列

$$a^s \bmod n, a^{2s} \bmod n, a^{4s} \bmod n, \ldots, a^{n-1} \bmod n \tag{5.11}$$

必定以1结束，而且在第一次出现1之前的值必定是$n-1$。这是因为n是素数时，$x^2 \equiv 1 (\mathrm{mod} n)$的唯一解为$x = \pm 1$（方程两边取离散对数即可）。

考虑除了最后一个数外的其他数。如果第一个数是1或者任何一个数为-1（意味着下一个数为1），都说明n可能是素数，否则是合数。这样的测试称为Miller-Rabin测试，代码如下（注意做乘法时应使用前面介绍的mul_mod函数而不能直接乘）：

```
bool miller_rabin(ll a, ll n)
{
        ll r = 0, s = n - 1, j;
        while(!(s&1)){ s >>= 1; r++; }
        ll x = pow_mod(a, s, n);
        if(x == 1) return true;
        for(j = 0; j < r; j++, x = mul_mod(x, x, n))
            if(x == n - 1) return true;
```

```
        return false;
}
```

定理：Miller-Rabin测试输出合数时，n一定是合数。

证明：如果n是奇素数，那么$a^{n-1} \bmod n = 1$。由于$j = 0$时x不为1也不为-1（否则算法将输出素数），因此$j = 1$时不为1 （只有1和-1的平方模n才为1），也不为-1（否则算法将输出合数），因此$j = 2$时也不为1。由数学归纳法易得：$j = r - 1$时（倒数第2个数）也不为1和-1。因此a^{n-1}也不为1。由Fermat小定理知：n为合数。

显然这个方法对于非Carmichael数的出错概率不会比刚才更大，而事实上，包括Carmichael数在内的所有合数，出错概率不超过1/4。

为什么这个方法对Carmichael数也有效？这里不给出证明，只举一个例子，直观上的感受一下。考虑最小的Carmichael数561。$561 = 2^4 \cdot 35 + 1$。从$x = 2^{35} \bmod 561 = 263$开始，有$263 \to 166 \to 67 \to 1$，在没有出现-1的情况下直接跳到1，因此判断561为合数。只有当n确实为奇素数时，1才只有两个平方根。即使是Carmichael数，也会出现67这样的其他根，避开-1而直接变成1，从而被正确的判断为合数。

出错限和时间复杂度。取$k = \lfloor \log n \rfloor / 2$，重复执行$k$次。如果每次都返回素数则n确实为素数的概率为$1 - 1/n$，任何一次返回合数时一定为合数。当$n$很大时$1/n$的出错概率可以忽略不计。

虽然Miller-Rabin测试不一定正确，但是对于小整数($n \leq 2.5 \cdot 10^{10}$)只测试$a = 2, 3, 5, 7$只会有合数3,215,031,751（它有因子151）会被误判为素数；通过2, 3, 5, 7, 11测试的最小合数为2,152,302,898,747（它有因子6763），而通过2, 3, 5, 7, 11, 13, 17的最小合数为341,550,071,728,321（它有因子10670053）。因此对于n比较小的情况，可以放心使用Miller-Rabin测试而不用担心会产生错误的结果。

这样，一个判断$2 * 10^9$之内的素数的Miller-Rabin测试是：

```
bool isprime(ll n)
{
    for(int i = 2; i < 1000 && i < n; i++)
        if(n % i == 0) return false;
    if(!miller_rabin(2, n)) return false;
    if(!miller_rabin(3, n)) return false;
    if(!miller_rabin(5, n)) return false;
    if(!miller_rabin(7, n)) return false;
    return true;
}
```

前面的试除不仅可以排除小素因子的干扰，而且对于Miller-Rabin测试是必须的：

用a测试时必须保证n不是a的倍数。其中1000是经验性的，可根据情况调整，但不能比最大的a小。

欧拉ϕ函数的计算。前面遗留的一个问题是：如何计算ϕ函数？$\phi(n)$函数有一个重要性质：它是是**积性(multiplicative)** 的，即对于互素的m, n，有$\phi(mn) = \phi(m) \cdot \phi(n)$。有了这个定理，我们可以推导出$\phi(n)$的计算公式。

考虑$\phi(p^k)$，其中p是素数，k是正整数。显然$1 \sim p^k$中与p^k不互素的数只有$p, 2 \cdot p, 3 \cdot p, \ldots p^{k-1} \cdot p$这些，一共$p^{k-1}$个。由加法原理，与它互素的数有$p^k - p^{k-1} = p^k(1-1/p)$个。

把任意n分解素因数，由刚才的讨论以及欧拉函数的积性可知：$\phi(n) = n \cdot \Pi(1 - 1/p)$。

有趣的是：欧拉函数的计算和素数判定很类似：单个欧拉函数可以用试除法，而$1 \sim n$的所有欧拉函数可以用筛法，时间复杂度仍分别为$\mathrm{O}(\sqrt{n})$和$\mathrm{O}(n \log\log n)$。

单个欧拉函数 由刚才的讨论，只需要设置初始结果为n，然后对n做素因数分解，对于所有素因子p，乘以$1 - 1/p = (p-1)/p$即可。代码如下：

```
int euler_phi(int n)
{
    phi = n;
    for(i=2,j=4;j<=n;i++,j+=i+i-1)
    if(!(n%i)){
        phi = phi / i * (i-1);
        while(!(n%i)) n/= i;
    }
    if(n>1) phi = phi / n * (n-1);
    return phi;
}
```

这里有个技巧是：得到一个素因子i后，立刻把n全部消去，因此这样不会找到非素因子pq，因为在这之前已经消去了p和q的所有幂，不再有因子pq了。

$1 \sim n$的所有欧拉函数 和素数的筛法是类似，求出一个素数p后，它对所有倍数$2p, 3p, \ldots$ 都有$(p-1)/p$的贡献，用一个循环即可完成求解。初始设所有ϕ值为0，因此如果再累乘时发现一个0值，先初始化为n，再乘以$(p-1)/p$。另外由于还没有被筛过的数ϕ值为0，因此这同时也是素数标志。

```
void all_phi(int n, int phi[])
{
    phi[1] = 1;
    for(i = 2; i <= n; i++)
    if(!phi[i]) //prime
```

```
    for(j=i; j<=n; j+=i){  //for each multiple
        if(!phi[j]) phi[j] = j;  // first time, initialize
        phi[j] = phi[j] / i * (i-1);  // multiply
    }
}
```

和素数筛法一样，时间复杂度为$O(n \log\log n)$。

小测验

1. 叙述Eratosthenes筛法并给出它的时空复杂度。复杂度的分析用到了哪两个结论？
2. 叙述直接基于Fermat小定理的概率测试，它对哪类数效果特别不好？为什么？叙述Miller-Rabin测试。为什么它可以有效的对付Carmichael数？
3. 什么是积性函数？叙述ϕ函数的计算方法，包括单个函数和1∼n的所有函数值。

5.1.6 因数分解(1)：基本算法

因数分解是一个很基本的问题，但是却比素数判定要难得多。17世纪时，Fermat发明了一种很有趣但不是很实用的分解方法，而从1970年开始，因数分解算法开始多了起来。

这些算法中，比较早期的是J.M.Pollard发明的Pollard rho算法和Pollard p-1算法。另一个比较简单的算法是基于连分数的，Morrison和Brillhard 把这种算法加以变形，成为70年代的主流算法。这个算法的时间复杂度是亚指数级别的，即$n^{a(n)}$，其中$a(n)$是n的减函数。一般用$L(a,b)$表示比特运算是$O(exp(b(\log n)^a(\log\log n)^{1-a}))$的。刚才提到的Morrison-Brillhard算法是$L(1/2, 3/2^{1/2})$的，它在1970年成功的分解了一个63位的大数。

1981年，Carl Pomerance发明了Quadratic sieve，分解了超过百位的数而不需要此数具有任何特别的性质。当加了若干优化后，它是$L(1/2,1)$的，并分解了一个129位的整数（RSA-129）。对于超过115位数时，number field sieve更加有效，它是被Pollard提出，由Buhler, Lenstra和Pomerance改进，它是$L(1/3,(64/9)^{1/3})$的，成功分解了155位的RSA-155，但是当位数不太大时仍然没有Quadratic sieve有效。

Fermat分解算法 从$t = n^{1/2}$开始，依次检查$t^2-n, (t+1)^2-n, (t+2)^2-n\ldots$，直到出现一个平方数$y$，由于$t^2-y^2 = n$，因此分解得$n = (t-y)(t+y)$。显然，当两个因数很

接近时这个方法能很快找到结果，但如果遇到一个素数，则需要检查$(n+1)/2-n^{1/2}$个整数，比试除法还慢得多。虽然方法并不是很有效，但是为我们提供了一个思路。

Lehman算法 下面的算法现在已经很少使用，但它是一个渐进时间复杂度严格比试除法好的方法。算法思想是先把n限制成$n=pq$的形式，其中p和q都为素数。这一步只需要试除到$B=n^{1/3}$即可，因为如果$n=pqr(p\le q\le r)$，则$n\ge p^3$，因此$p\le n^{1/3}$。经过这一步后，如果n是合数，则$n=pq$且p和q为素数。

现在把k从1循环到B，检查在范围$4kn\le a^2\le 4kn+B^2$内且满足$a\equiv r(\mathrm{mod}m)$的所有a，令$c=a^2-4kn$。如果c是完全平方数，设$c=b^2$，则$gcd(a+b,n)$是n的非平凡因子。这里，k为偶数时取$m=2,r=1$，而k为奇数时$m=4,r=(k+n)$。

```
ll lehman(ll n)
{
    ll a, b, c, t, r, m, B = pow(n, 1.0/3.0) + 1;

    // trial division
    for(ll i=2; i<=B; i++)
        if(n%i == 0) return i;

    for(ll k=1; k<=B; k++)
    {
        // set m and r
        if(k%2==0){ m = 2; r = 1; }
        else{ m = 4; r = (k + n)%m; }

        // initialize t and a
        t = 4 * k * n;
        a = sqrt(t); while(a*a<t)a++;
        a+=(m+r-a%m)%m;

        // try all possible a, test if c is a perfect square
        while(a*a < t+B*B)
        {
            c = a*a - t; b = sqrt(c);
            if(b*b == c) return gcd(a+b, n);
            a += m;      // next a
        }
    }
    return 0;  // prime
}
```

由于需要计算$4kN$（不取模，因此无法用Head算法），最坏情况是$4N^{4/3}$，所以为了让ll不溢出，应该有$4N^{4/3}\le 4\cdot 10^{18}$，即$N\le 3\cdot 10^{13}$。不过实际上可以先近似计算出$b$，如果$gcd(a+b,n)$不是平凡因子就算了。此算法的最坏情况时间复杂度为$\mathrm{O}(n^{1/3})$，

比试除法好。读者可以自己尝试证明这个结论。

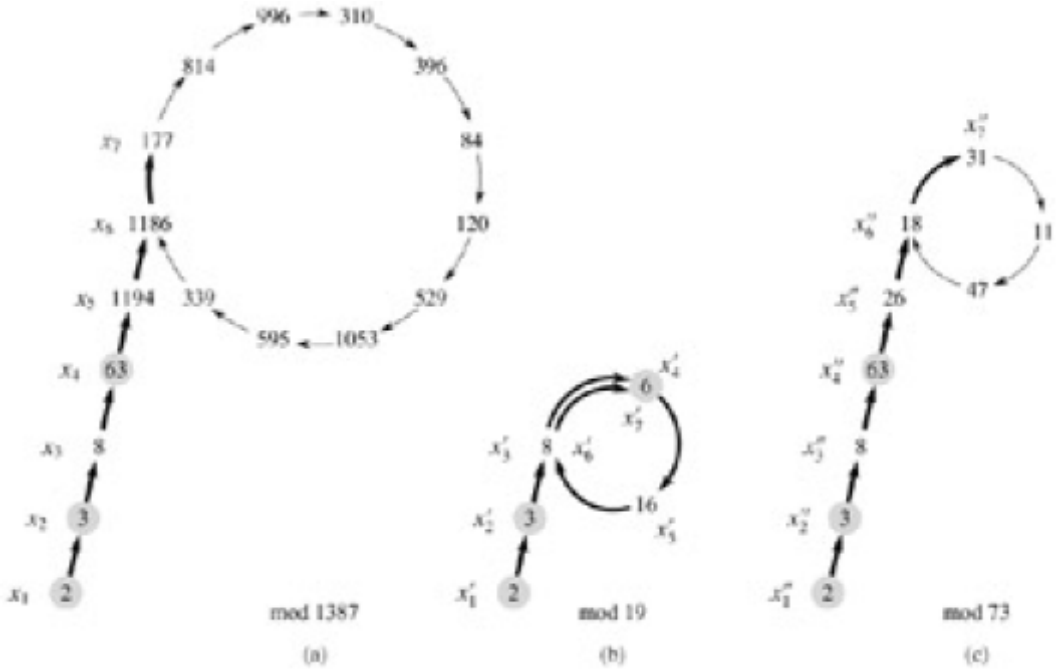
Pollard的rho算法 有一个很有意思的算法称为rho算法，它的实际效果不错，但是并没有理论保障。算法的核心思想是：如果d是n的非平凡因子且两个数x_i和x_j满足

$$x_i \equiv x_j (\mathrm{mod} d), x_i \not\equiv x_j (\mathrm{mod} n) \tag{5.12}$$

则$gcd(x_i - x_j, n)$是n的非平凡因子。这是因为$d|(x_i - x_j)$，但$n \nmid (x_i - x_j)$。$gcd(x_i - x_j, n)$不是1（d是公因子且$d > 1$）它也不是n（因为$n|(x_i - x_j)$），而只能是n的非平凡因子。

问题在于：如何找到这样的x_i和x_j？rho算法的基本思想是：生成+判圈。生成采用“近似随机”的多项式，而判圈采用Floyd“追逐”算法。

从一个x_1开始，每次选择一个多项式计算$x_i = f(x_{i-1})$（计算在Z_n中进行因此每次自动模n）。例如$n = 1387 = 19 \cdot 73, f(x) = n^2 - 1$且$x_1 = 2$，则生成序列为：$2 \to 3 \to 8 \to 63 \to 1194 \to \mathbf{1186} \to 177 \to 814 \to 996 \to 310 \to 396 \to 84 \to 120 \to 529 \to 1053 \to 595 \to 339$。$f(339) = 1186$，产生循环。我们可以把这个生成序列画成一个好看的形式：



图形是不是很像希腊字母ρ（发音为“rho”）？这就是rho算法名字的由来。

稍微运算一下就会发现：$gcd(63-6,1387)=gcd(1194-16,1387)=gcd(1186-8,1387)=\ldots=19$，但由于我们实现并不知道$d$，所以也没有办法直接检查是否有$x_i \equiv x_j (\mathrm{mod} d)$，只能计算$gcd(x_i-x_j,n)$是否为1或者$n$。如果为1则继续，如果为$n$的话，说明大序列已经循环。在这个范围之内如果找不到因子，则继续生成也无济于事。

最简单的方法是对于每个x_i，检查$x_i-x_1 x_i-x_2\ldots$，但这样做计算量太大。Floyd提出一种很有意思的算法：比较x_i和x_{2i}。把x_i和x_{2i}想象成速度为1和2的两个小孩子，那么如果有圈，则最终两人都将进入圈内。一旦进入圈内，不管相对位置如何，速度为2的小孩子每次以相对速度*1*靠近另一个小孩子，因此一定会追上。换句话说，圈存在当且仅当存在i使得$gcd(x_{2i}-x_i,n)$是n的非平凡因子。这就是原始的Pollard-rho算法。

如果假设生成序列在Z_d内是随机的，那么循环出现的期望位置是$d^{1/2}$。由于最小因子$p \le n^{1/2}$，因此循环出现的最坏期望位置是$n^{1/4}$，比Lehman算法优。但实际上我们并不能指望生成序列是真正随机的，所以有可能失败（在Z_d和Z_n中同时循环），在这个时候我们应当寻找一个另外的多项式，而不只是另外一个初值。典型的选择是$x_{i+1}=(x_i^2-c) \bmod n$，除了$c=0$和$c=2$不应该被使用。

原始的Pollard-rho算法代码如下：

```
ll rho(ll n)
{
    ll x, y, d, c = -1;
    while(1){
        y = x = 2;
        while(1){
            x = mul_mod(x, x, n); x = (x - c) % n;
            y = mul_mod(y, y, n); y = (y - c) % n;
            y = mul_mod(y, y, n); y = (y - c) % n;
            d = gcd(abs(y - x), n);
            if(d == n) break;
            else if(d > 1) return d;
        }
        c--;
    }
}
```

每次发现循环就让c减1。由于一开始是-1，所以能避免0和2。由于使用了mul_mod过程，所以即使只用ll也只能支持到10^{18}之内的。

Pollard的$p-1$算法 这是由Pollard提出的另一个很有意思的算法。先随便选$a \in$

Z_n，再取一个“能被很多素数幂整除”的数k，例如$k = lcm(1, 2, \ldots, B)$，且B越大成功的希望越大。下面计算$a_k = a^k \bmod nd = gcd(a_k - 1, n)$。如果$1 < d < n$，则$d$是$n$的非平凡因子，成功退出；否则再选一个$a$或者/并再选一个$k$。

这个方法非常通俗易懂，也很容易写成程序：

```
ll p_1(ll n){
    ll a, k, ak, d;
    a = 2; k = 840; // any a and k is ok
    while(1){
        ak = pow_mod(a, k, n);
        d = gcd(ak - 1, n);
        if(d > 1 && d < n) return d;
        a++;  // failed. modify a (or increase k)
    }
}
```

但它并不总能成功。事实上，它对满足“存在素因子p，使得$p-1$无大因子”的n最有效。因为此时$(p-1)|k$，而$p \nmid a$。由费马定理知$a^k \equiv 1(\mathrm{mod}p)$，因此$a^k - 1$是$p$的倍数，因此

$$p|gcd(a^k - 1, n) \tag{5.13}$$

之所以要让k“能被很多素数幂整除”是为了让$(p-1)|k$。最坏情况下$(p-1)/2$是素数，此时$p-1$和试除法一样。这个算法有一个改进，即把一个B改成两个数B_1和B_2，用二阶段法代替一阶段法，有兴趣的读者可以阅读相关书籍。

小测验

1. 描述Fermat分解算法。它最坏情况运算量和试除法哪个大？它最适合分解怎样的n？
2. 描述Lehman算法，证明它的最坏情况时间复杂度为$\mathrm{O}(n^{1/3})$。对于怎样的n，它的时间效率和试除法完全一样？
3. 描述Pollard提出的rho算法和$p-1$算法。他们各适合分解怎样的n？最坏情况时间复杂度有保障吗？

5.1.7 因数分解(2)：连分数、椭圆曲线和其他

Legendre同余式 上一节的算法除了Lehman之外，其他算法的最坏情况表现都不佳，它们的分析往往是根据经验或者依赖于n的某些特殊性质。本节介绍因数分解的三

个高级算法：CFRAC，QS和NFS并重点讨论CFRAC，它们的特点是：对于任何n的分解时间只和n的大小有关。

这三个算法全部依赖于以下同余式：

$$x^2 \equiv y^2 (\mathrm{mod} n), 0 < x < y < N, x \neq y, x + y \neq n$$

由于n整除$(x+y)(x-y)$，但n不整除$x+y$和$x-y$，因此$gcd(x+y,n)$和$gcd(x-y,n)$都可能是n的非平凡因子。这个同余式称为**Legendre同余式** 。为了用它做因数分解，我们需要做两件事：

步骤一 找$x^2 \equiv y^2 (\mathrm{mod} n)$的一组非平凡解
步骤二 计算$(d_1, d_2) = (gcd(x-y,n), gcd(x+y,n))$。

例如$12^2 \bmod 119 = 5^2 \bmod 119$，因此计算$(gcd(12-5,119), gcd(12+5,119)) = (7,17)$。事实上，正好$119 = 7 \cdot 17$。

第二步很简单，因此关键在于如何找Legendre同余式的解。三个算法的思路是一致的：寻找一组所谓的**素数基(factor base, FB)** ，即一个“小素数”集合$\{p_1, p_2, \ldots, p_m\}$，并构造很多这样的同余式：

$$x_k^2 \equiv (-1)^{e_{0k}} p_1^{e_{1k}} p_2^{e_{2k}} \cdots p_m^{e_{mk}} \ (\mathrm{mod}\ n) \quad \ldots \tag{5.14}$$

然后选一些式子乘起来，使得右边每个p_i的指数都是偶数，则右边配成了完全平方，而左边已经是一些完全平方的乘积，这样就构造出了Legendre同余式的解。可是如何选择呢？设第i个方程选时$\varepsilon_i = 1$，不选时$\varepsilon_i = 0$。这样，右边配出平方当且仅当

$$\sum_{1 \le k \le n} \varepsilon_k \left(e_{0k}, \cdots e_{mk}\right) \equiv (0, \cdots, 0) \pmod 2$$

这是一个向量方程，如果把m个维全部拆开，得到的就是一个Z_2的线性方程组，可以用高斯消元法求出一组解。

显然构造同余式5.14是问题的关键。一个显然的方法是枚举x，并尝试分解$x^2 \bmod n$，但由于很多x都无法用FB分解开，所以这个方法耗费的时间可能很长。

CFRAC,QS和NFS用不同的方法构造同余式(5.14)，但都是当构造的同余式足够多以后进行高斯消元，所以下面只讨论如何构造(5.14)。

用良好近似构造同余式 如果t很小且$P^2 \equiv t(\mathrm{mod}N)$，则可以写成$P^2 = t + Q^2kN$，因此$(P/Q)^2 - kN = t/Q^2$是一个很小的数（因为$t$“很小”）。换句话说，$P/Q$是$(kN)^{1/2}$的一个良好近似。

反过来，一旦得到$(kN)^{1/2}$的一个良好近似P/Q，记$t = P^2 - Q^2kN$，则我们得到了同余式$P^2 \equiv t(\mathrm{mod}N)$。因为此时$t$很小，因此“很有希望分解成素数基中素数幂乘积”，如果成功分解，则我们得到了一个(5.14)的解。

连分数逼近 连分数展开(continued fraction expansion) 可以用来进行有理逼近。如果

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ddots}}}$$

我们简记为$x = [a_0, a_1, a_2, a_3, \ldots]$。如果有理近似$a/b$是给定的，那么$a_i$可以简单的用Euclid算法求出。$x$的连分数展开有限当且仅当$x$是有理数。首先$a_0$是很容易得到的，它就是$x$的整数部分。聪明的读者也许已经发现：如果用Euclid算法求x, y的最大公约数，则每次商排列起来就是x/y的连分数表示！程序如下，设置n为项数，如果提前完结（$y = 0$）则返回项数。

```
ll CRE(ll x, ll y, ll n, ll a[])
{
    for(int i = 0; i < n; i++){
        if(y == 0) return i;
        else a[i] = x / y;
        ll t = x; x = y; y = t % y;
    }
    return n;
}
```

读者可以运行程序，验证$1281/243 = [5, 3, 1, 2, 7]$。

对于无理数的情形，算法需要修改一下，每一步让$a_i = [x_i]$，而$x_{i+1} = 1/(x_i - a_i)$（此式很容易通过代数恒等变形来得到）。程序如下：

```
const double eps = 1e-8;

int dcmp(double x)
{
    if(x < -eps) return -1;
    else return (x > eps);
}
```

```
ll CRE(double x, ll n, ll a[])
{
    for(int i = 0; i < n; i++){
        a[i] = (int)x;
        if(dcmp(x - a[i]) == 0) return i;
        x = 1.0 / (x - a[i]);
    }
    return n;
}
```

请注意：因为用到了实数，所以这里判断“分母为0”不能直接用等号，而需要判断它的绝对值是否足够小。这个函数dcmp是实数的三出口函数，它非常有用，在计算几何部分我们还将继续讨论。有兴趣的读者可以试试$3^{1/2}$的连分数展开，它是$[1,1,2,1,2,1,2,1,2,...]$，从第二个数开始是一个长度为2的小循环。读者可以再试式$5^{1/2}$和$7^{1/2}$。

只取到a_n的连分数称为第n个**收敛分数(convergent)**，记为$C_k = P_k/Q_k$，每个收敛分数都是原数的近似。它们可以用以下公式来计算（读者可以用恒等变形验证这些式子）：

$$\begin{aligned}
&\frac{P_0}{Q_0} = \frac{q_0}{1} \\
&\frac{P_1}{Q_1} = \frac{q_0 q_1 + 1}{q_1} \\
&\cdots\cdots \\
&\frac{P_i}{Q_i} = \frac{q_i P_{i-1} + P_{i-2}}{q_i Q_{i-1} + Q_{i-2}}, \quad i \geq 2
\end{aligned}$$

有了连分数逼近的知识，现在我们可以完整叙述CFRAC了。

CFRAC算法 取小整数k（往往是1）和不超过某个整数的所有素数作为FB。计算$(kN)^{1/2}$的连分数展开，每次求出一组P和Q，计算$W = P^2 - Q^2 kN$，并尝试用FB把它进行因数分解。如果成功，则得到一个同余式，加到方程组中。当方程足够多后，用高斯消元法求出方程的一组解，对应Legrendre同余式的一组解。一般来说，FB越大越好，而k只有当循环节太小时才设置为非1的值。

CFRAC的运行时间是

$$O\left(\exp\left(\left(\sqrt{2}+o\left(1\right)\right)\sqrt{\log N \log\log N}\right)\right) = O\left(N^{\sqrt{(2+o(1))\log\log N/\log N}}\right)$$

它是一个猜想而不是定理，因为它需要一些没有被证明的假设的支持。

总结：和前面的因数分解算法相比，CFRAC是一个比较复杂的算法，用到了多次问题转化。首先，它把“因数分解”的目标转化为“构造Legrendre同余式的解”，然后再次把问题转化为“构造同余式(5.14)的解”。这也许是最重要的一步问题转化，它把问题变成了：找足够数量的同余式$x^2 \equiv y(\mathrm{mod}n)$，只要右边的$y$可以分解成FB中素数幂的乘积，而不需要$y$本身是完全平方（如果是，则Legendre同余式已解出）。当然，由于n往往很大，Z_n中很多数都无法分解成FB中素数幂的乘积，因此CFRAC进行了最后一次问题转化：使用连分数求$(kN)^{1/2}$的逼近，这样可以让y比较小，从而让它“比较有希望成功分解”。QS和NFS使用其他方法构造$W = P^2 - Q^2kN$，两边取余同样有$W \equiv P^2(\mathrm{mod}N)$。

MPQS算法。和CRFAC算法类似，QS（Quadratic Sieve）算法的目标也是要构造出同余式(5.14)，同样也希望$x^2 \bmod n$不要太大（这样比较有希望分解开）。考虑多项式

$$Q(a) = \left(\left\lfloor\sqrt{N}\right\rfloor + a\right)^2 - N$$

显然当$x = \left\lfloor\sqrt{N}\right\rfloor + a$时$Q(a) = x^2 - N$，因此$Q(a) \equiv x^2(\mathrm{mod}N)$。只要$a$非常小，则$Q(a)$也会很小。CRFAC算法用连分数展开计算$W$，而这里不用找就有一大堆，因为每个小$a$都对应唯一的一个$x$。我们只需要保留所有能用$FB$分解得开的$Q(a)$即可。

如何保留所有能分解得开的$Q(a)$呢？一种办法是从小到大枚举a，只要发现$Q(a)$可以用FB分解开就加到方程组里。但这种方法不可取[1]，因为有很多$Q(a)$实际上是分解不开的（别看$Q(a)$小就以为一定能分解开！）。怎么办呢？既然是“一大堆数中找满足条件的数”，一个很自然的想法就是：筛。和素数筛法类似，如果$m|Q(a)$，则对于任意整数k都有$m|Q(a + km)$，细节这里不再赘述。

QS的思想简单，且有很多变形，最著名的一个是**MPQS算法(Multiple Polynomail Quadratic Sieve)** 。从名字可以看出来，它使用多个多项式，即$Q(x) = Ax^2 + 2Bx + C$。其中$A > 0, B^2 - AC > 0$，且$N|B^2 - AC$。

容易验证，$AQ(x) = (Ax + B)^2 - (B^2 - AC) \equiv (Ax + B)^2(\mathrm{mod}N)$，是一个形如$x^2 \equiv y(\mathrm{mod}n)$的式子，只要$AQ(x)$能分解开，我们就成功的构造了一个同余式5.14。

和前面一样，我们仍然希望$Q(x)$越小越好（这样$AQ(x)$比较有希望在FB中分解开），如果如果筛法作用的区间长度为M，这个区间应该是关于$Q(x)$的最小值对称的

[1]虽然枚举a再计算x比起直接枚举x来说已经好很多了

区间，即$I=[-B/A-M,-B/A+M]$。

为了让边界处绝对值尽量相等，即$Q(-B/A)\approx -Q(-B/A+M)$，解得

$$A\approx \frac{\sqrt{2\left(B^2-AC\right)}}{M}$$

代入得$\max_{x\in I}|Q(x)|\approx \frac{B^2-AC}{A}\approx M\sqrt{\left(B^2-AC\right)/2}$。由于我们希望$|Q(x)|$越小越好，而又需要满足$N|B^2-AC$，因此取$B^2-AC=N$。

总结一下。首先选定筛法执行长度M，然后选一个接近$(2N)^{1/2}/M$的A，使得A是素数且$x^2\equiv N(\mathrm{mod}A)$有根（即Legendre符号为1）。求出平方根$B$使得$B^2\equiv N(\mathrm{mod}\ A)$，再令$C=(B^2-N)/A$。

下面对于每个素数幂$m=p^k$，用它来筛。由于$m|Q(a)$推出$m|Q(a+km)$。因此只需要找到第一个满足$m|Q(a)$的a，即$Q(a)\equiv 0(\mathrm{mod}m)$的根。由于取$B^2-AC=N$，因此它的两个根是$(-B+a)/A$和$(-B+b)/A$，其中$a$和$b$分别是$x^2\equiv N(\bmod m)$的两个解，而除以$A$的运算是在$Z_m^*$中进行的。MPQS的好处是：如果筛法失败，可以选择另外一个A，从而得到另外一个多项式，比原始的QS更具灵活性。

Lenstra的ECM算法ECM很像Pollad的p-1算法，也需要取一个k被“很多素数幂整除”，不过它的主运算不是幂取模，而是椭圆曲线运算。

令N是一个正整数且$gcd(N,6)=1$，则Z_N上的**椭圆曲线(elliptic curve)** 被定义为：

$$Ey^2=x^3+ax+b \tag{5.15}$$

其中$a,\ b$是整数且$gcd(N,4a^3+27b^2)=1$。E上的点集为此方程的所有解，加上一个特殊的点：无穷点O_E。

定义椭圆曲线上点的加法为：

$$P_1\oplus P_2=\begin{cases} O_E & x_1=x_2,y_1=-y_2 \\ \left(\lambda^2-x_1-x_2,\lambda\left(x_1-x_3\right)-y_1\right) \end{cases}$$

$$\lambda=\begin{cases} \frac{3x_1^2+a}{2y_1}, & P_1=P_2 \\ \frac{y_2-y_1}{x_2-x_1}, & otherwise \end{cases}$$

对于定义在Z_N^*上的椭圆曲线，以上的加、减、乘、除都是在模N下进行的。可以写出点加法的程序：

```
struct elliptic_curve{ ll a, b, n; };
elliptic_curve ec;

struct ecp{ bool oe; ll x, y; };
ecp operator+(ecp a, ecp b)
{
    ecp c; ll l;
    if(a.x == b.x && mod(a.y + b.y, ec.n) == 0)
        c.oe = true;
    else
    {
        if(a.x == b.x && a.y == b.y)
            l = div_mod(mod(mul_mod(mod(3*a.x, ec.n), a.x, ec.n) + ec.a, ec.n), 2*a.
        else
            l = div_mod(mod(b.y - a.y, ec.n), mod(b.x - a.x, ec.n), ec.n);
        c.oe = false;
        c.x = mod(mul_mod(l, l, ec.n) - a.x - b.x, ec.n);
        c.y = mod(mul_mod(l, mod(a.x - c.x, ec.n), ec.n) - a.y, ec.n);
    }
    return c;
}
```

在程序中，我们设置了全局变量ec，另外写了mod和div_mod来进行取模和除法取模运算。如果读者准备写自己的椭圆曲线代码，可以实验$a=-2, b=-3, n=7, p=(3,2)$时$2P, 3P, \ldots, 10P$是否分别为$(2,6),(4,2),(0,5),(5,0),(0,2),(4,5),(2,1),(3,5),O_E$。

现在我们可以正式叙述Lenstra的ECM算法了。首先随机选择$0 \leq a, x, y < N$，让$b = y^2 - x^3 - ax$，则我们实际上得到了一条曲线$Ey^2 = x^3 + ax + b$ 和一个点$P(x, y)$。如果$gcd(4a^3 + 27b^2, N) \neq 1$，则E不是椭圆曲线，重新随机选择。

接下来的处理和Pollard的p-1算法类似：选择一个“被很多素数幂整除”的k，例如$k = lcm(1, 2, \ldots, B)$或者$k = B!$。下一步是计算kP，如果$kP \equiv O_E(\mathrm{mod}N)$，则令$z = m_2$并计算$d = gcd(z, N)$。如果$1 < d < N$，则$d$是$N$的非平凡因子。如果同余式不成立或者$d$等于1或$N$，重新随机选择$k$或者$E$和$P$。需要注意的是$kN$应该用倍增法求，而如果在运算过程中如果发现无法进行除法（分母的逆不存在），则算法立即停止，因为此时分母和N的最大公约数一定是N的非平凡因子。

SQUFOF算法 这个算法是基于实二次域理论的，但是在算法描述中我们可以忽略它。虽然它并不是特别快，但是对于小数(10^{19}以内的数)效果特别好。

定义1：设$f = (a, b, c)$为判别式为整数D的二次型，f是最简的当且仅当

$$\left|\sqrt{D} - 2\,|a|\right| < b < \sqrt{D}$$

定义2：设$D>0$为判别式，若$a\neq 0$和b都是整数，$r(b,a)$为满足$r\equiv b(\mathrm{mod}2a)$且

$$\begin{array}{ll} -|a|<r\leq|a| & |a|>\sqrt{D} \\ \sqrt{D}-2|a|<r<\sqrt{D} & |a|<\sqrt{D} \end{array}$$

的唯一整数r，则定义化简操作符

$$\rho(a,b,c)=\left(c,r(-b,c),\frac{r(-b,c)^2-D}{4c}\right)$$

算法首先判断N是否为素数或者完全平方数。然后选择一个**判别式(discriminant)** D，它应是N的倍数。这里取$N\equiv 1(\mathrm{mod}4)$时$D=N$，否则$D=4N$。接下来我们从**全等型(identity form)** 开始不断使用化简运算符，每次把$|A|$放入集合Q中，A是完全平方数（此时称为平方型）且算术平方根a没有在Q中出现过。

如果$s=gcd(a,B,D)$大于1，则s^2是因子；否则把$(a,-B,-aC)$化简到最简，设为(a,b,c)。最后进行循环，每次设置$b_1=b$，继续使用化简运算符，直到b_1和b相等。如果a是奇数则输出$|a|$，否则输出$|a/2|$。

```
ll squfof(ll n)
{
    triple f, g;
    set<int> Q;
    ll D, d, b, L, b1;

    if(n % 4 == 1)
        { D=n; d=ll_sqrt(n); b=2*((d-1)/2)+1; }
    else
        { D=4*n; d=ll_sqrt(n); b=2*(d/2); }

    f.A = 1; f.B = b; f.C = (b*b-D)/4;
    L = ll_sqrt(d);
    Q.clear();

    for(int i = 1; ; i++)
    {
        f = rho(f);
        if(i % 2 == 0)
        {
            ll a = ll_sqrt(f.A);

            // square form found
            if(a*a == f.A && !Q.count(a))
            {
                ll s = gcd(gcd(a, f.B), D);
                if(s > 1) return s*s;
```

```
                g.A = a; g.B = -f.B; g.C = a*f.C;
                while(!reduced(g)) g = rho(g);

                do{
                    b1 = g.B; g = rho(g);
                }while(b1 != g.B);

                if(a % 2 == 1) return a;
                else return a/2;
            }
            if(f.A == 1) return 0; // period too short, failure
        }
        if(abs(f.A) < L) Q.insert(abs(f.A));
    }
}
```

注意到a, b, c都是$N^{1/2}$这个数量级的，因此基本运算非常快。

5.2 数值计算

本节讨论一些基本的数值计算问题，包括基本的大数运算、非线性方程求根、矩阵运算、线性方程组和FFT。

5.2.1 大整数运算

本节需要用到整除（注意负数的情况）、素数、合数的概念、算术基本定理、带余除法和同余的记号。需要注意的是：数论算法中最需要注意的问题是：负数、情况讨论和溢出。为了缓解溢出的危险，我们在程序中使用ll数据类型。它并不是C++的标准类型，在VC++中为__int64，而gcc中为long long。

数论中经常要用到大整数运算。根据应用不同，需要编写不同类型的大整数运算。

方法论 使用C++通常有两种方法写大数运算，一种是写成一个类，另一种是写成结构，然后定义运算符。我们推荐使用后者，因为在很多情况下，用到的只是运算的一个子集。这样写起来很容易“装卸”各种算法，即：

```
const int base = 100000000;
const int num_digit = 8;
const int maxn = 1000;
struct bign{
  int len;
```

```
  int s[maxn];
  bign(const char* str){ (*this) = str; }
  bign operator=(const char* str){
  int i;
  int j = strlen(str) - 1;
  len = j / num_digit + 1;
  for(i = 0; i <= len; i++) s[i] = 0;
  for(i = 0; i <= j; i++){
    int k = (j - i) / num_digit + 1;
    s[k] = s[k] * 10 + str[i] - '0';
  }
  return *this;
  }
};
```

这里采用1亿（10^8）进制，即每位是0 ~ 99999999的数。为了方便，提供字符串为参数的赋值和构造函数（赋初值也会调用构造函数而不是赋值运算符）。读者可以很容易写出其他赋值和构造函数，根据需要选用。s[0]是保留位置，s[len]~s[1]是各个数位上的数，因此可以写出打印函数：

```
void print(bign a){
  for(int i = a.len; i >= 1; i--)
    printf("%d", a.s[i]);
}
```

比较运算 有一个很重要的函数：比较。

```
int compare(bign a, bign b){
  if(a.len > b.len) return 1;
  if(a.len < b.len) return -1;
  int i = a.len;
  while((i > 1)&&(a.s[i] == b.s[i])) i--;
  return a.s[i] - b.s[i];
}
```

利用它可以很方便的定义运算符<，==和其他派生符号。注意比较的前提是整数没有前导0。如果有的话，算法会认为“001>99”，因为算上前导0后前者的数位多大。这也提示我们：在所有大数运算的末尾要注意去掉前导0。

加法和减法 加法和减法很容易写出，只需注意不要忽略前导0。加法和减法有两种特殊的形式：自加自减和与单精度的运算。这两种特殊情况可以节省时间和空间，把优化方法留给读者。

```
bign operator+(bign a, bign b){
bign c;
```

```
for(int i = 1; i <= a.len || i <= b.len || c.s[i]; i++){
  if(i <= a.len) c.s[i] += a.s[i];
  if(i <= b.len) c.s[i] += b.s[i];
  c.s[i+1] = c.s[i] / base;
  c.s[i] %= 10;
}
c.len = i-1;
if(c.len == 0) c.len = 1;
return c;
}

bign operator-(bign a, bign b){
bign c;
int i, j;
for(i=1,j=0; i<=a.len; i++){
  c.s[i]=a.s[i]-j;
  if(i<=b.len) c.s[i] -= b.s[i];
  if(c.s[i] < 0){ j = 1; c.s[i] += base; }
  else j = 0;
}
c.len = a.len;
while(c.len > 1 && !c.s[c.len]) c.len--;
return c;
}
```

这里，减法的前提是a>b。

乘法 乘法需要谨慎一点，因为直接做乘法（二重循环，a.s[i]*b.s[j]加到c.s[$i+j-1$]中）是会溢出的。如果不想改成每位都是ll（这样多占用内存且速度变慢），需要一位一位计算。注意当确定当前计算位置k后，i的最小值和最大值分别为$max(k+1-b.len, 1)$和$min(k, a.len)$。注意临时变量tmp必须是ll型的，且乘法$a.s[i] \cdot b.s[k+1-i]$必须临时转化为ll再做。

```
bign operator*(bign a, bign b){
bign c;
ll g = 0;
int i, k;
c.len = a.len + b.len;
c.s[0] = 0;
for(i = 1; i <= c.len; i++) c.s[i] = 0;
for(k =1; k <= c.len; k++){
  ll tmp = g;
  i = k+1-b.len;
  if(i < 1)i = 1;
  for(; i <= k && i<=a.len; i++)
    tmp += (ll)a.s[i]*(ll)b.s[k+1-i];
  g = (int)(tmp / base);
  c.s[k] = (int)(tmp % base);
```

```
}
while(c.len > 1 && !c.s[c.len]) c.len--;
return c;
}
```

除法 除法是四则运算中最伤脑筋的一个，通常用两种方法。一种是试除法，每一位依次尝试；第二种是二分法。试除法的程序如下，当base很大时它非常慢。其中c和d分别是商和余数。

```
void divide(bign a, bign b, bign& c, bign& d){
int i, j;
for(i = a.len; i > 0; i--){
  if(!(d.len == 1 && d.s[1] == 0)) {
    for(j = d.len; j > 0; j--)
      d.s[j+1] = d.s[j];
    ++d.len;
  }
  d.s[1] = a.s[i];
  c.s[i] = 0;
  while( (j = compare(d, b)) >= 0) {
    d = d-b;
    c.s[i]++;
    if(j==0) break;
  }
}
c.len = a.len;
while(c.len > 1 && !c.s[c.len]) c.len--;
}
```

二分法可以用在局部和整体，即可以一位一位二分也可以整体二分。我们在这里推荐一位一位二分。这样，只需要找到满足$a \cdot base^n \cdot b \leq c < (a+1) \cdot base^n$的$a$。程序留给读者编写，这里提醒一下：二分中用到的乘法需要单独写（而不能用已有的乘法操作符），以节省移位开销。这一点和前面的分治整数乘法的实现是类似的。

对小整数取模 给一个n位大整数p和正整数m，用$p[0], p[1], \ldots, p[n-1]$表示大整数的各个数位，如何计算$p \bmod m$的值？根据同余的性质，有：$ab \bmod m = ((a \bmod m)(b \bmod m)) \bmod m (a+b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$

设p的前k位整数为$q[k]$，那么有递推公式$q[k] = q[k-1] \cdot 10 + p[k]$，两边对$m$取模，记$t[k]$为前$k$位整数$q[k]$除以$m$的余数，则有：$t[k] = (t[k-1] \cdot 10 + t[k]) \bmod m$。由于所有$t[k]$都只用一次，我们用一个变量$d$记录它，即：

```
ll bigmod(bign a, ll m){
int d = 0;
```

```
for(i = 0; i < a.len; ++i){
  d = mul_mod(d, base, m);
  d = mod(d + a.s[i], m);
}
}
```

开平方 开平方可以用试除法，二分法或者迭代法。试除法比较慢，这里不推荐使用；二分法略，这里给出牛顿法。牛顿法实际上是求解$x^2 = n$，或者写成$f(x) = x^2 - n = 0$。迭代过程为$y = (x + n/x)/2$。程序如下：

```
bign sqrt(bign n){
bign x, y = n;
do{
  x = y;
  y = (x + n/x) / 2;
}
while(y < x);
return x;
}
```

需要用到高精度除高精度数(n/x)和高精度数除2。

小测验

1. 叙述高精度运算包的两种实现方法。它们各有什么利弊？
2. 叙述高精度四则运算的基本思想。base选取和这些运算的关系怎样？
3. 叙述开平方的迭代算法，并与二分法比较。

5.2.2 多项式与FFT

上一小节叙述了高精度乘法，它的时间复杂度是$O(n^2)$。利用FFT可以把它改进为$O(n \log n)$，这就是本小节的主题。

多项式 定义在数域F（通常为复数域C）上的单变量**多项式(polynomial)** 被定义为如下的和式：

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

其中a_i被称为**系数(coefficient)**，它们都是F里的元素。如果一个多项式的最高位非零系数是a_k（即$i > k$时a_i全为0），称这个多项式的**次数(degree)** 为k。

两个次数分别为a和b的多项式$A(x)$和$B(x)$的和和乘积分别为次数为$max\{a,b\}$和$a+b$。

多项式的表示 多项式有两种常见表示法：系数表示法和点-值表示法。

系数表示法记录系数向量$\mathrm{a}=(\mathrm{a}_0,\mathrm{a}_1,...,\mathrm{a}_{n-1})$。在本小节中，向量被当作列向量处理。给两个多项式的系数表示a和b，多项式加法对应于向量加法a+b，而多项式乘法对应于向量**卷积**（**convolution**）$a\otimes b$。上一节的多项式算法实际上就是在计算卷积。

点值表示法是一个点-值对的集合$\{(\mathrm{x}_0,\mathrm{y}_0),(\mathrm{x}_1,\mathrm{y}_1),...,(\mathrm{x}_{n-1},\mathrm{y}_{n-1})\}$，其中所有$\mathrm{x}_i$两两不等且$\mathrm{y}_k$=A($\mathrm{x}_k$)。这是一种间接表示法，选择不同的$\mathrm{x}_k$，得到的表示也不一样。事实上，一个多项式有唯一的系数表示，但有无穷多种点值表示。反过来，给定一组点值表示是否有唯一的多项式和它对应呢？下面的定理说明了这个问题：

定理：给n个点-值对$\{(\mathrm{x}_0,\mathrm{y}_0),(\mathrm{x}_1,\mathrm{y}_1),...,(\mathrm{x}_{n-1},\mathrm{y}_{n-1})\}$，只有唯一的次数小于n的多项式以它作为点值表示。

定理告诉我们：如果限定多项式的次数严格小于n，点值表示和多项式是一一对应的。点值表示法适合于很多运算。如果两个多项式所选取的点$\{\mathrm{x}_i\}$是一致的，则两个多项式的加法和乘法只需要把对应的y_i乘起来，两种操作都是O(n)的[2]。如果能够快速进行系数表示法和点值表示法的相互转换，则序列卷积就可以快速被求出。

求值和插值我们希望快速进行以下两种操作：

求值(evaluate)：从系数表示法转化为点值表示法。
插值(interpolate)：从点值表示法转化为系数表示法。

其中求值过程可以随意选择点集，但插值过程的答案是唯一的。原理上说选择任意点集都可以得到正确的结果，但正如我们很快就将看到的，如果选择**单位复根**（**complex roots of unity**）为求值点的话求值可以通过FFT求出（否则只能连续用n次Horner规则在O(n^2)时间内算出），而插值则通过逆FFT求出。由于FFT的时间复杂度为O($n\log n$)（比O(n^2)好很多！），我们达到了快速转换目的。

快速多项式乘法 这样，给出多项式的系数表示法，我们可以设计如下算法计算两个次数小于n的多项式A和B的乘积C：

步骤一（添零） 往高位添0，得到两个“次数为$2n$”的多项式。
步骤二（求值） 把2n次单位根作为求值点，用FFT计算两个多项式的点值

[2]乘法操作需要实现把n个点值对扩充为2n个

表示。

步骤三（乘法） 把两个点值表示的对应点的值乘起来得到C的点值表示。

步骤四（插值） 用FFT从C点值表示求出系数表示。

框图如下：

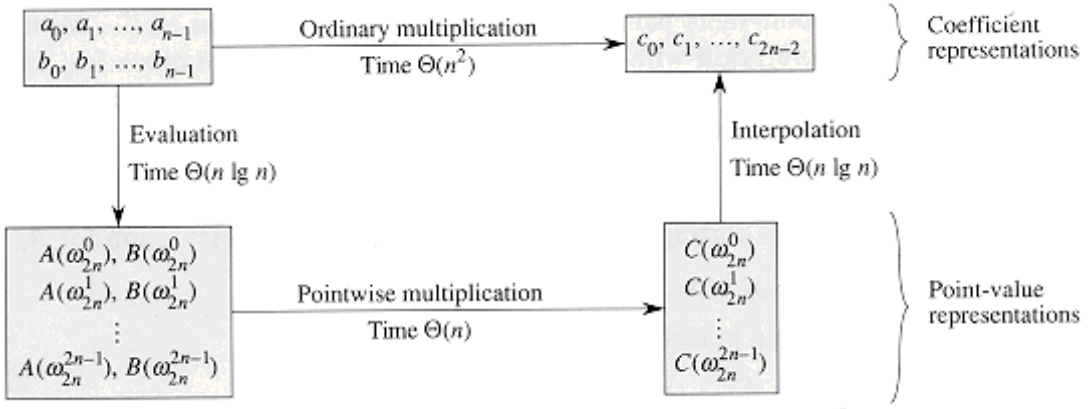


图 5.1: 用FFT求多项式乘法框图

单位根定义n次单位根为满足ω^n=1的复数ω。恰好有n个单位根$e^{2\pi ik/n}$(k=0, 1, ..., n-1)，其中i是虚数单位(i^2=-1)，而

$$e^{iu} = \cos u + i \sin u$$

这样，我们可以把单位根看作是单位圆上的n等分点，如下图：

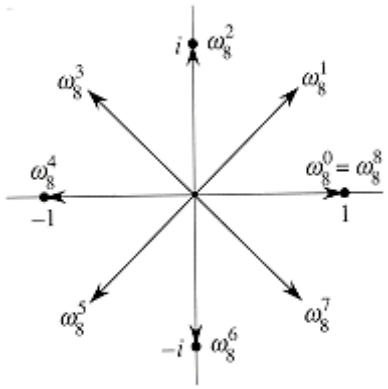


图 5.2: 单位根

定义$\omega_n = e^{2\pi i/n}$，称为n次主单位根(principal nth root of unity)，则所有其他单位根都是它的幂。这样，n个单位根可以写成$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$，它们在乘法下构成群，且结构和加法群$Z_n$一样。容易看出，$\omega_n^k = -\omega_n^{k+n/2}$。

DFT假设n是2的幂（若不是，可以补0），我们希望计算多项式$A(x) = \sum_{j=0}^{n-1} a_j x^j$在$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$处的值。定义$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} \omega_n^{kj}$，则向量y = ($y_0$, y_1, ..., y_{n-1})称为系数向量a = (a_0, a_1, ..., a_{n-1})的离散傅里叶变换（Discrete Fourier Transform, DFT），记为y=$DFT_n(a)$。

FFT有一种称为**快速傅里叶变换（Fast Fourier Transform, FFT)** 的方法用到了n次单位根的特殊性质，从而在$O(n \log n)$时间内算出$DFT_n(a)$，在直观的$O(n^2)$基础上产生了飞跃。这个方法是分治的，它把奇数项和偶数项分开处理：

$$A^{[0]}(x) = a_0 + a_2 x + \cdots + a_{n-2} x^{n/2-1}$$
$$A^{[1]}(x) = a_1 + a_3 x + \cdots + a_{n-1} x^{n/2-1}$$

那么$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$。前面提到过$\omega_n^k = -\omega_n^{k+n/2}$，因此$(\omega_n^k)^2 = (\omega_n^{k+n/2})^2$。这样，我们可以同时计算A(x)在这两个点的值。这样，计算A(x)在n个点上值的任务被转化成了计算$A^{[0]}(x)$和$A^{[1]}(x)$在n/2个点上值的任务了。

设$A(x)$, $A^{[0]}(x)$和$A^{[1]}(x)$在ω_n^k处的值分别为$y_k, y_k^{[0]}, y_k^{[1]}$，则我们有公式：

$$y_k = y_k^{[0]} + \omega_n^k y_k^{[1]}, y_{k+n/2} = y_k^{[0]} - \omega_n^k y_k^{[1]}$$

上式的操作实际上是用$y_k^{[0]}$和$y_k^{[1]}$算y_k和$y_{k+n/2}$，这样的操作称为**蝴蝶操作(butterfly operation)** ，如下图：

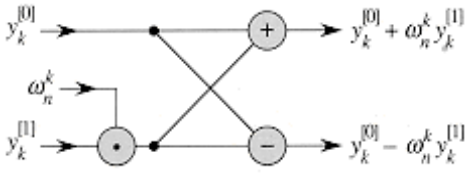


图 5.3: 蝴蝶操作

这样，只需要花$O(n)$的时间就可以把序列$y^{[0]}$和$y^{[1]}$合并成序列y了。递归方程为$T(n)=2T(n/2)+O(n)$，因此解为$T(n)=O(n \log n)$。

迭代FFT实现 虽然可以用递归法实现，但为了让速度更快，一般把FFT写成迭代形式。为了说清楚迭代顺序，我们先来看一个典型的递归树，如图8.1.1：

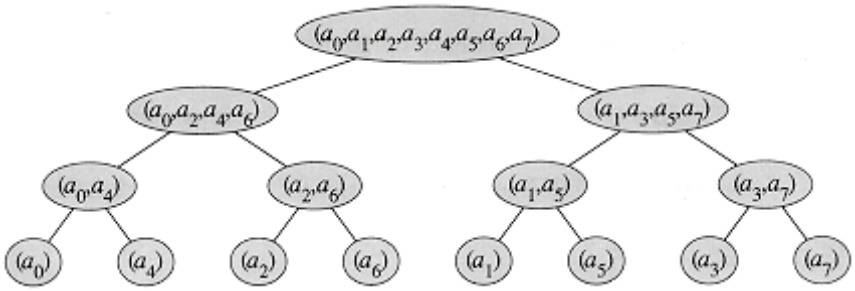


图 5.4: FFT的递归树

如果一开始就把序列排成叶子的顺序，那么就可以把递归变成迭代：一共迭代$\log n$次，每次处理一层，每次用$m/2$次蝴蝶操作合并两个长度为$m/2$的子序列，得到长度为m的序列。这样，伪代码如图8.1.1：

```
ITERATIVE-FFT (a)
 1  BIT-REVERSE-COPY (a, A)
 2  n ← length[a]          ▷ n is a power of 2.
 3  for s ← 1 to lg n
 4       do m ← 2^s
 5          ω_m ← e^(2πi/m)
 6          for k ← 0 to n - 1 by m
 7               do ω ← 1
 8                  for j ← 0 to m/2 - 1
 9                      do t ← ωA[k + j + m/2]
10                         u ← A[k + j]
11                         A[k + j] ← u + t
12                         A[k + j + m/2] ← u -
13                         ω ← ω ω_m
```

图 5.5: 迭代FFT框架

注意我们使用了辅助变量u和t以方便代码阅读。

剩下的事情就是如何排列叶子了。仔细观察发现实际上叶子顺序是把比特反转后的顺序。例如在刚才的递归树中，叶子0, 4, 2, 6, 1, 5, 3, 7的二进制为000, 100, 010, 110, 001, 101, 011, 111，反转后恰好为0~7。设k的二进制反转结果为rev(k)，则BIT-REVERSE-COPY过程就是简单的对于0~n-1的所有数k，设置A[rev(k)] = a_k，时间复

杂度为O($n \log n$)。有更快的实现方法，留给读者思考。

插值操作插值操作是DFT的逆过程IDFT，它和DFT有什么联系吗？略去数学推导，这里只给出结果：$\mathrm{DFT}_n^{-1}(\mathrm{y})$的结果为：

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega^{-kj}$$

和DFT比较得：只需要用ω_n^{-1}代替ω_n，并把结果的每个元素除以n即可使用FFT的程序来计算IDFT。这样，经过补0、求值、乘法、插值四个步骤，我们在O($n \log n$)时间内完成了多项式乘法。

快速整数乘法作为多项式乘法的应用，我们可以用它做整数乘法。在b进制表示中，整数可以看作以各个数位为系数的多项式在$x = b$处的值。例如数123,456,789可以看作

$$A(x) = 789 + 456x + 123x^2$$

在x=1000处的值。这样，我们可以用FFT在O($n \log n$)的时间计算多项式乘法，然后再进行进位处理，以保证每个系数都是0∼999的整数。但这里有一个问题：FFT中用到了复数，会引起误差。事实上整数乘法中多项式的系数是整数，结果也是整数，可不可以避开复数呢？回答是肯定的。

方法一 取m=$2^{tn/2}$+1，2^t作为模m下的n次主单位根。这个方法是可行的，但是需要保留O(n)个比特的数据（因为模m=O(2^n)），运算量比较大，而且很容易溢出。这个算法也称为Schönhage-Strassen算法。

方法二 寻找最小的k使得$p = kn + 1$为素数。令g为$Z*_p$的一个原根，取n次主单位根$\omega = g^k \bmod p$。可以预料p并不会太大，因此所有操作都不会溢出。实用中可以取p=15×2^{27}+1=2,013,265,921，而31是$Z*_p$的原根。可以支持的最大n为2^{27}=134,217,728，而每个位储存15个比特。取模p下的n次单位根为$31^{15} \bmod p = 440{,}564{,}289$，而2的逆（用在IDFT中）为1,006,632,961。

小测验

1. 什么是多项式？叙述它的两种表示法和求值、插值的概念。
2. 什么是n次单位根？n次单位根的0次、1次...n-1次幂构成怎样的代数结构？

3. 叙述FFT的迭代实现。如何用它实现大整数乘法？为什么需要把讨论从复数域转换到Z_p^*中？

5.2.3 矩阵和线性方程组

本节介绍矩阵和代数方程组。**矩阵(matrix)** 是一个由数排列成的矩形，例如

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

是一个2×3矩阵$A=(a_{ij})$，其中i=1,2, j=1, 2, 3。第i行第j列的元素用a_{ij}表示。一般用大写字母表示矩阵，小写字母表示它的元素。A的转置A^T通过交换A的行和列得到，例如

$$A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

向量(vector) 是一个n×1矩阵，即有n行但只有1列。我们也把向量称为列向量，类似可以定义行向量，如x^T=(2 3 5)。行和列均为n的矩阵称为n阶**方阵(square)** 。有很多特殊的方阵，如**对角阵(diagonal matrix)**：

$$diag(a_{11}, a_{22}, \cdots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}$$

还有**单位阵(identity matrix)** I_n=diag(1, 1, ..., 1)，上三角阵、下三角阵等。如果$A=A^T$，我们说A是**对称阵(symmetric matrix)** 。

矩阵的运算 矩阵的加法和减法被简单的定义为两个大小相等的矩阵对应元素相加、相减。而矩阵乘法需要进一步说明。如果A的列数等于B的行数，则可以定义乘法C=A×B。如果A是$m \times n$矩阵，B是$n \times p$矩阵，那么C是一个$m \times p$矩阵，c_{ij}满足：

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

矩阵乘法满足结合律和对加法的分配律，根据定义很容易设计出一个O(n^3)的算法进行矩阵乘法。如果把列向量x看成n×1矩阵的话可以定义向量**内积(inner product)**

$x^T y$和**外积(outer product)** xy^T，前者是一个数（或者说1×1矩阵）而后者是一个$n \times n$矩阵。

矩阵的逆对于n阶矩阵A，如果存在A^{-1}使得$AA^{-1}=I_n=A^{-1}A$，则称A^{-1}是A的**逆(inverse)**。存在逆的矩阵称为**可逆矩阵(invertible matrix)**或**非奇异矩阵(nonsingular matrix)**。如果A和B都是非奇异矩阵，则$(BA)^{-1}=\mathrm{A}^{-1}B^{-1}$，$(A^{-1})^T=(A^T)^{-1}$。矩阵的可逆性和前面提到过的线性组合有着密切联系。n个向量x_1, x_2, ..., x_n是**线性相关(linearly dependent)**的，如果存在不全为0的系数c_1, c_2, ..., c_n使得这些向量的线性组合$cx_1 + cx_2 + ... + cx_n$=0。例如x_1=(1 2 3), x_2=(2 6 4)和x_3=(4 11 9)是线性相关的，因为$2x_1$+3x_2-2x_3=0。如果向量组不是线性相关的，我们称它们是**线性无关(linearly independent)**的。

矩阵的秩矩阵的**列秩(column rank)**是最多的线性无关列向量组的元素个数，类似可定义**行秩(row rank)**。可以证明：行秩一定等于列秩，我们把它称为矩阵的**秩(rank)**。如果n阶方阵的秩是n，我们说它是满秩(full rank)的。n阶矩阵是满秩的当且仅当它是可逆的，它的另一个充要条件是：不存在非零向量x使Ax=0。

行列式定义n阶矩阵的**行列式(determinant)**，即$\det(A)$如下：记$A[ij]$为A删除第i行和第j列后得到的新矩阵，则有递归式

$$\det(A) = \begin{cases} a_{11} & n = 1 \\ \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(A_{[1j]}) & n > 1 \end{cases}$$

直接用这个定义计算需要指数时间，通常使用行列式的以下性质：

1. 若某行或某列为0，则det(A)=0。
2. 若某行或某列的所有元素同时乘以c，则行列式乘以c。
3. 若某行加到另外一行（原来那行保持不变）上或某列加到另外一列上，则行列式不变。
4. 若某两行（或两列）交换，则行列式乘以-1。
5. 将矩阵转置不改变行列式值。
6. 对角阵的行列式值为对角元素乘积。特别地，$\det(\mathrm{I}_n)$=1。

另外，$\det(AB)=\det(A)\det(B)$，因此一个矩阵是可逆的当且仅当它的行列式不为0。对于上/下三角矩阵，行列式值和对角阵一样等于所有对角元素之积。这样，我们可以在$O(n^3)$的时间求出矩阵的行列式，方法是不断使用性质2、3、4，直到得到一个上/下三角阵。

线性方程组 考虑n个未知数n个方程的线性方程组：

$$\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1\\
&a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2\\
&\vdots\\
&a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n
\end{aligned}$$

可以把它写成矩阵形式。令$A=(a_{ij})$，$x=(x_i)$，$b=(b_i)$，则$Ax = b$。如果A是非奇异的，则方程两边同时乘以A^{-1}有$x=A^{-1}b$，它是唯一解。在这种情况下，我们可以对矩阵A进行LUP分解，即写成$PA = LU$的形式，L和下三角阵，U是上三角阵。先解$Ly = Pb$，得到y（相当于得到高斯消元后新的b'），然后解$Ux = y$，即回代过程。

消元过程 考虑当前方程为i，当前变量为j时的情形。首先把x_j项系数绝对值最大的方程和方程i交换。如果所有项系数都为0，则放弃该变量（j加1），否则对于所有k=i+1～n，把方程i乘以$-a_{kj}/a_{ij}$后加到第k个方程中。经过这样的过程，当未知数或方程处理完毕时结束，方程组成为阶梯型。虽然最后可能会有一些没有未知数的“方程”，但它们要么是多余方程，要么是矛盾方程（想一想，为什么），很容易处理。在有未知数的方程中，后一个方程总是比前一个方程的未知数严格少。

回代过程 回代过程有一些特殊情况需要处理。首先是如$x_1+x_2+x_3=3$这样的方程。实际上我们只能说$x_2=k_1$，$x_3=k_2$，而$x_1=3-k_1-k_2$。由于第一个未知数系数非0（想一想，为什么），因此如果一共有m个变量，则有m-1个基变量和1个非基变量（即可以根据基变量算出的变量）。我们统一规定非基变量是当前方程的第一个变量。

考虑下面这种情况：

$$\begin{cases} x_1 + 2x_2 + 2x_3 + 2x_4 = 8\\ \qquad x_2 + x_3 + x_4 = 3 \end{cases}$$

这已经是阶梯形式了。虽然x_2、x_3和x_4都是不定的，但决不能以“方程1中有无法确定的x_2，x_3和x_4”来断定x_1也是不确定的。事实上，很容易知道$x_1=2$是确定的！解决这一问题的方法是回代时消元，消去非基变量。如果这样做还有“不确定变量”，则该方程剩下的变量一定都是不确定的。容易知道，回代时消元后仍然能保证每次处理时第

一个变量的系数非0。

这样做有一个附加的好处：回代过程遇到的全部是基变量，因此很容易得出所有变量的表达式，进而得到更紧凑的解形式。例如x_1=3，x_2=k_1，x_3=k_2，x_4=5-k_1-k_2，可以写成x=(3, k_1, k_2, 5-k_1-k_2)=(3, 0, 0, 5)+k_1(0, 1, 0, -1)+k_2(0, 0, 1, -1)，这里k_1和k_2是独立的，后两项可以作看(0, 1, 0, -1)和(0, 0, 1, -1)的任意线性组合。需要说明的是：如果当前方程只有一个未知数就不要消元了，因为消元的开销是比较大。

小测验

1. 什么是矩阵、方阵、列向量和行向量？如何定义向量的内外积和矩阵乘法？什么是矩阵的逆？矩阵的奇异性与行/列向量相关性、矩阵的秩和行列式值有何关系？
2. 给出计算行列式的算法并分析时间复杂度。
3. 给出解线性方程组的消元过程和回代过程。为什么会出现多余方程和矛盾方程？为什么在回代时也要消元？如果A是可逆的，还需要回代时消元吗？

5.3 组合计数

本节讨论组合数学中最基本的问题：组合计数。计数问题形如“满足性质P的物体有多少个？”，看起来简单但往往不容易解决。前面提到过，优化问题可以归结为存在性的判定问题，而存在性的判定问题可以归结为检查“满足此条件的物体个数大于等于1吗？”来解决，因此计数问题是普遍存在的，且比判定问题更困难。本节不讨论困难的计数问题，而着眼于最基本的计数问题，重点是阐述计数方法，分析经典问题。

5.3.1 计数原理和离散概率

本节介绍最基本的计数原理。**加法原理(rule of sum)** 告诉我们：如果做一件事有两类解决途径，分别有a种和b种具体方案，则一共有a+b种方案。**乘法原理(rule of product)** 告诉我们：如果做一件事需要分为两个步骤，分别有a种和b种具体方案，则一共有ab种方案。这两个原理是直观的，用集合论的符号表示是：

加法原理 若两个有限集A和B交集为空，则$|A \cup B| = |A| + |B|$

乘法原理 对于任两个有限集A，B，有$|A \times B| = |A| \cdot |B|$

它们分别通过分类和分步来进行计数。作为例子，考虑这样一些简单问题：

字符串。字母表的大小为n，那么长度为k的**字符串(string)** 有多少个？由于第1个字母有n种选法，第2个字母也是，...第k个也是。由乘法原理，一共有n^k个。

排列。字母表的大小为n，那么选k个不同的字母组成**排列(permutation)** 有多少个？由于第1个字母有n种选择，第2个字母有n-1种（不管第1个字母是啥，由于字母不能重复，第2个字母必然少一个选择）...第k个字母有n-k+1种。由乘法原理，总数为：

$$P_n^k = n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!}$$

有时候需要对排列的大小进行估计，这时需要用到n!的估计式，即Stirling公式：

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \text{œfflO}'^{-}O\text{œ}\tilde{}X100! = 9.33 * 10^{157}\text{flfflO}\text{œ}(J9.39 * 10^{157}\text{”}$$

组合。字母表的大小为n，它有多少个含k个元素的子集？这样的子集也称**组合(combination)** ，它和排列相比多了一个无序性。假设答案为C_n^k。排列可以分成两步得到：先得到组合，然后把选出来的k个数进行排列。由乘法原理得$P_n^k = C_n^k \times P_k^k$，因此：

$$C_n^k = \frac{P_n^k}{P_k^k} = \frac{n!}{k!(n-k)!}$$

需要补充说明的是：组合数的计算是值得注意的。虽然可以利用$C_n^k = C_n^{n-k}$（想一想，为什么）把k变小。但当n并不是很大时分子已经溢出。为了避免这样的情况，可以先进行约分再计算，具体过程留给读者思考。另一个常用的过程是一次计算出很多组合数，它利用公式$C_i^j = C_{i-1}^j + C_{i-1}^{j-1}$，而long可以存下的最大i是21。

```
For(i=1;i<=21;i++){
  c[i][0] = c[i][i] = 1;
  for(j=1; j<i; j++)
    c[i][j]=c[i-1][j]+c[i-1][j-1];
}
```

利用这个递推式还可以画出著名的杨辉三角。

可重组合。如果每个数可以选多次，那么从n个数中选k个有多少种方法？这相当于方程$x_1+x_2+...+x_n=k$有多少组非负整数解。其中x_i表示第i个数被选了多少次。为了解这个方程，我们先引入辅助变量$y_i=y$，方程两边加n得：$y_1+y_2+...+y_n=n+k$。问

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1

图 5.6: 杨辉三角

题转化为求此方程的正整数解的个数。想象有n+k个盒子排成一列，每个y_i表示连续的y_i的盒子，那么只需要把这n+k个盒子分成非空的n段即可，如下：

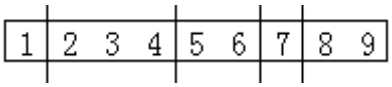


图 5.7: 可重组合的转化

这样，我们只需要确定这n-1个“分界线”的位置。由于一共有n+k-1个可能的位置，所以n-1个分界线一共有C_{n+k-1}^{n-1}种可能的选择，这就是问题的答案。

离散概率 概率(probability) 与计数密切相关。定义一些**基本事件(elementary event)**，我们称所有基本事件的集合为**样本空间(sample space)** S。一个基本事件可以认为是实验的一个可能结果。例如抛两枚不同的硬币，样本空间为S={HH, HT, TH, TT}，其中每一个基本事件都是一个可能的实验结果。有了样本空间S，我们把S的某些子集称为**事件(event)**，比如“一个正面一个反面”的事件为{HT, TH}。S本身称为**必然事件(certain event)**，因为它必然发生；空集称**空事件(null event)**，因为它必然不发生。每个基本事件对应的单元素集合看作它对应的事件。如果两个事件A和B的交集为空，称它们**互斥事件(mutally exclusive events)**。概率分布Pr{}是从S中事件到实数的映射，它必须满足以下概率公理：

对于任意事件A，Pr{A}≥0

Pr{S} = 1

对于有限或可数多两两互斥的事件A_1, A_2, .., 有

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\}$$

我们称Pr{A}为事件A的概率。如果概率分布定义在有限或可数集S上，则称此概率分布是**离散(discrete)** 的。在这种情况下，$\Pr\{A\} = \sum_{s \in A} \Pr\{s\}$。如果所有基本事件s都满足$\Pr\{s\} = 1/|S|$，则称它为S上的均匀分布。在这种情况下，实验通常被描述为“随机在S中选择一个元素”。考虑把硬币抛n次的实验，让事件A表示恰好k次正面朝上，则$A = C_n^k$，而样本空间大小为2^n，因此$\Pr\{A\} = C_n^k/2^n$。通过这个例子我们看到：计数和离散概率有着密切联系。

连续均匀分布 如果S是实数上的闭区间[a, b]，那么如何定义“均匀分布”？直观上应该让“每个点都等概率被取到”，但点有无穷多，因此如果单个点的概率为正实数，那么整个区间的总概率将是正无穷，违反公理2。在这样的情况下，我们不能让S的任意子集都代表事件，而规定只把闭区间$[c, d]$看作事件，定义

$$\Pr\{[c, d]\} = \frac{d - c}{b - a}$$

条件概率和独立性 在B条件下（即：已知B是已经发生的事件），事件A的概率称为**条件概率(conditional probability)** ，它被定义为：

$$\Pr\{A|B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}}$$

既然B已经发生，Pr{B}必不为0。这个公式很直观，类似于乘法原理。直观地，我们定义满足$\Pr\{A \cap B\} = \Pr\{A\} \Pr\{B\}$的事件A和B为**独立(independent)** 的。如果B的概率不为0，它等价于Pr{A|B}=Pr{A}，即B发生与否和A的概率没有影响。

Bayes公式。由于“集合交”运算满足交换律，我们有

$$\Pr\{A \cap B\} = \Pr\{B\} \Pr\{A|B\} = \Pr\{A\} \Pr\{B|A\}$$

变形得：

$$\Pr\{A|B\} = \frac{\Pr\{A\} \Pr\{B|A\}}{\Pr\{B\}}$$

这个公式称为**Bayes定理(Bayes's theorem)** 。进一步可以把公式变形为

$$\Pr\{A|B\} = \frac{\Pr\{A\} \Pr\{B|A\}}{\Pr\{A\} \Pr\{B|A\} + \Pr\{\bar{A}\} \Pr\{B|\bar{A}\}}$$

假设有两个硬币，一个是正常硬币，两面朝上的概率都是1/2，而另一个是异常硬币，总是正面朝上。现在在两个硬币中任选一枚抛两次后发现都是正面朝上。问它是异常硬币的概率有多大。A表示选到异常硬币的事件，B表示两次正面朝上的事件，则目标是求Pr{A|B}。已经知道$\Pr\{A\} = 1/2, \Pr\{B|A\} = 1, \Pr\{\bar{A}\} = 1/2, \Pr\{B|\bar{A}\} = 1/4$，代入公式得Pr{A|B}=4/5。在这里，变形的Beyes公式简化了计算，因为在这里条件概率比较好求出。如果在这里和A平行的还有其他事件，可以进一步进行分类，用加法公式来计算Pr{B}。

离散随机变量 随机变量被定义为S到实数的映射函数。比如在抛两次硬币的实验中，“正面朝上的次数”就是一个随机变量X，它定义为X(HH)=2, X(HT)=X(TH)=1, X(TT)=0。由此可见，随机变量包含的内容非常丰富，它实际上是给实验的每种可能的结果定义了一个值。：在这里我们只讨论离散随机变量。定义随机变量X的概率密度函数为：

$$f(x) = \Pr\{X = x\} = \sum_{\{s \in S: X(s) = x\}} \Pr\{s\}$$

这个直观定义只是简单的把值为x的所有基本事件的概率都加起来，正如在很多情况下，我们并不关心实际情况是哪一个基本事件，而只关心出现的此基本事件的某个数值属性，比如在抛硬币实验中，我们也许并不关心到底哪次正面，哪次反面，只关心一共有几次正面。

实际上，我们根据“正面朝上次数”把基本事件划分成了若干等价类，等价类i里的基本事件的正面朝上的次数都是i，它们的概率加起来就是“正面朝上次数为i的概率”。注意这里体现了一个转化思想：即把事件的概率转化为了事件的函数（即离散变量）的概率。这样处理在很多时候非常方便和自然。

随机变量的独立性 可以定义两个随机变量的**联合概率密度函数(joint probability density function)** 为：

$$\Pr\{X = x | Y = y\} = \frac{\Pr\{X = x, Y = y\}}{\Pr\{Y = y\}}$$

可以类似定义X和Y独立性当且仅当对于所有x和y，X=x和Y=y是两个独立事件，或者说对于所有x和y，Pr{X=x,Y=y} = Pr{X=x}Pr{Y=y}。

对于在同一个样本空间中定义的随机变量，可以定义四则运算和其他运算，和普通变量一样。

期望和方差 有两个值用来刻画随机变量各种取值的概率分布。**数学期望(expected value/expectation)** 被定义为所有取值依概率加权之和，记为E[X]，即：$E[X] = \sum_x x \Pr\{X = x\}$。期望最重要的性质是线性性：E[X+Y]=E[X]+E[Y]，很多其他性质都可以由它推导出。显然只用期望刻画随机变量是不够的，期望为0的随机变量可能恒为0，也可能是等于100或-100的概率均为1/2。直观地，我们用**方差(variance)** 来刻画随机变量在期望周围的"波动"期望。方差越大，波动越厉害。它的定义是Var[X]=E[(X-E[X])^2]。注意E[X]是一个数，而X-E[X]是另外一个随机变量。Var[X]被定义为此随机变量平方的期望。根据期望的线性性质，不难推出Var[X]=E[X^2]-E^2[X]。方差的常用性质有：

1. Var[aX]=a^{2}Var[X]
2. X和Y独立时Var[X+Y]=Var[X]+Var[Y]

定义标准差σ为方差的算术平方根，因此也用σ^2表示方差。

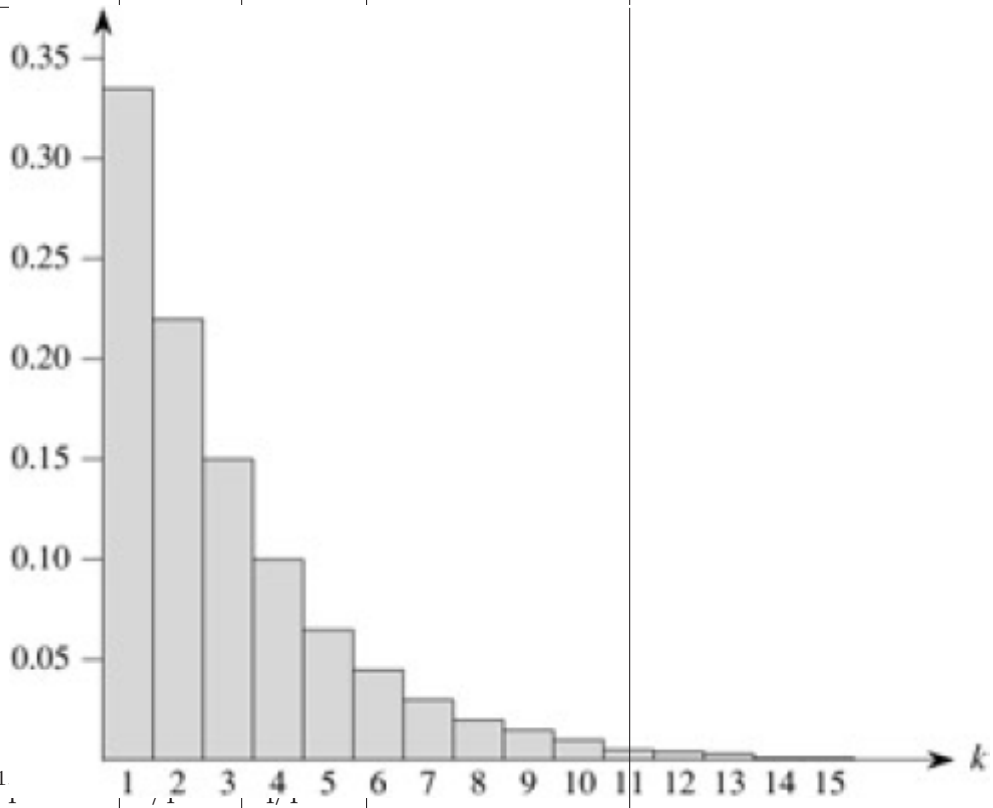
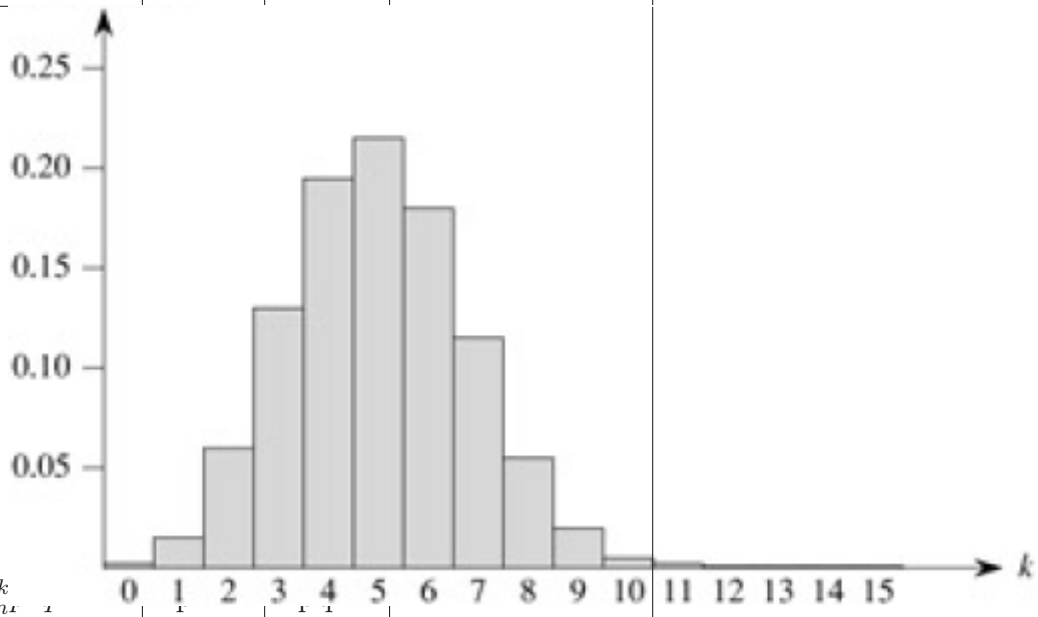
几何分布和二项分布 抛硬币是一个典型的Bernoulli实验，即只有两种结果的实验，成功概率为p，失败概率为q=1-p。考虑连续进行独立Bernoulli实验的情形，并考察两个随机变量：第一次成功之前进行的实验次数X和n次实验中成功的次数Y。我们称这里的X满足**几何分布(geometric distribution)**，而Y满足**二项分布(binomial distribution)**。数学推导给出下表：

小测验

1. 计数问题具有怎样的形式？叙述加法原理和乘法原理，并用它们推导排列数和组合数的公式。
2. 什么是离散概率？概率的定义需要满足哪三条概率公理？叙述条件概率和事件的独立性，并说明Bayes公式的内容和用途。
3. 什么是随机变量？如何定义它的概率密度函数、期望和方差？如何定义两个随机变量的联合概率密度函数和独立性？什么是几何分布和二项分布？符合此二种分布的随机变量的期望和方差是多少？

5.3.2 编码、解码和枚举

本节讨论三个看起来不相关的问题：计数、序号和枚举。在很多时候，我们需要把满足一定条件的物体排列起来，依次编号。这样，我们可以：

名称	描述	f(k)	期望	方差	图例
几何分布	连续进行独立Bernoulli实验，第一次成功前的实验次数	q^{k-1}			
二项分布	n次独立Bernoulli实验中	C_{n}^{k}			

编码 给物体，求编号，或者

解码 给编号，求物体，或者

枚举 以编号从小到大的顺序列出所有物体

下面，我们以各种常见物体为例讨论这三种问题的联系。

二进制串 可以把{0,1}上长度为3的字符串排列如下。

序号	1	2	3	4	5	6	7	8
字符串	000	001	010	011	100	101	110	111

读者也许注意到了，字符串就是序号减1的2进制表示。因此字符串到序号、序号到字符串的转换都可以直接利用二进制和十进制的转化（如果需要用十进制输出的话）。

排列 这里是1~4的24个排列的前8个，它们以**字典序(lexicographical order)** 排列。

序号	1	2	3	4	5	6	7	8
字符串	1234	1243	1324	1342	1423	1432	2134	2143

编码方式并不是显然的。有两种方式考虑这个问题，一种是所谓的变进制计数法，另一种是编码和解码的一般方法。

编码实际上是求“不大于x的物体有多少个”，因此是一个计数问题。由于在字典序下第一个不一样的位就决定了顺序。这样，可以考虑用“分类”的思想，一位一位求。例如求426315的编码可以用图8.1.2中的递归树来表示：

其中灰色为叶子结点，可以直接计算。需要注意的是同层的叶子结点所代表的物体数目是一样的，例如421???，423???和425???都有3!=6种，因此应该每次计算当前层有多少个叶子结点，然后乘以它们所对应的个数（对于一般情况，同层叶子对应的个数可能不一样）。在这里例子中，426315的编码为5!×3+4!×1+3!×3+2!×1+1!×1=405。

反过来，解码可以一位一位的确定，方法是逐步累加，直到确定当前位上的数。仍然以编号405为例，先来确定第一位。

分类	1?????	2?????	3?????	4?????
个数	120	120	120	120
累加值	120	240	360	480

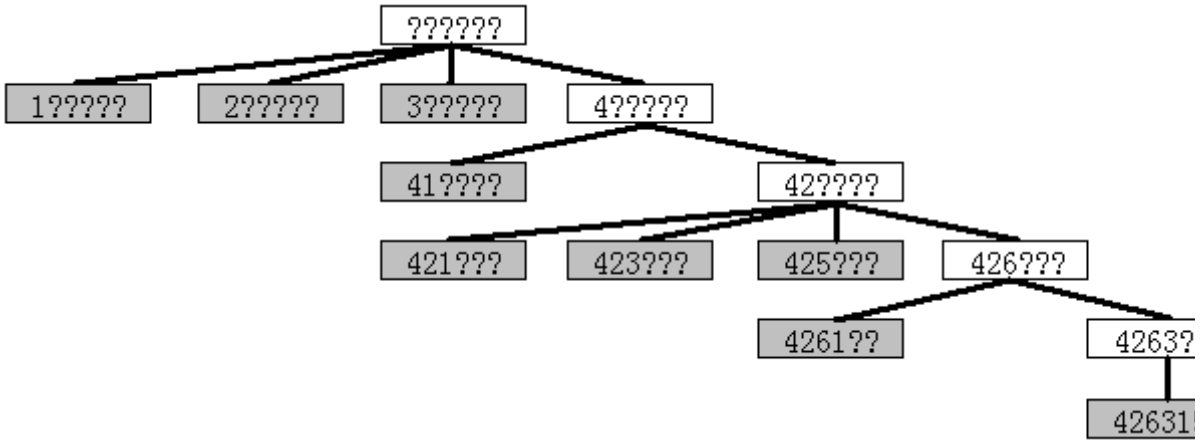


图 5.8: 排列426315的编码

由于360<405而480≥405，因此第一位为4。目前累加到3?????，即360。下面确定第二位。

分类	**41????**	**42????**
个数	24	24
累加值	384	408

因此第二位为2。目前累加值为384。下面确定第三位。

分类	**421???**	**423???**	**425???**	**426???**
个数	6	6	6	6
累加值	390	396	402	408

因此第三位为3。重复上面步骤，最终确定编号405为426315。

可重排列 我们把排列扩展一下：可以选择的元素是可重的。例如可重集{a, a, b, c}上的全排列按顺序写出是：

序号	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
串	aabc	aacb	abac	abca	acab	acba	baac	baca	bcaa	caab	caba	cbaa

我们可以按类似于排列的方法进行编码和解码。这里我们仍然需要解决一个问题：把比它小的物体进行分类，并对形如42????这样的“模板”进行计数。在排列问题中，有k个问号的模板对应于$k!$的物体，但在可重排列中，事情并没有这么简单。

在集合{a, a, b, c}上，模板a???对应6个物体但b???只对应于4个物体。一般模板实际上是在原集合中删除部分元素以后的可重全排列。全异排列中集合的元素个数决定

了它的所有性质，但在可重排列中，每个元素的重数也起到关键作用。假设模板中可选集中各个元素的有n_1，n_2，...，n_k个，一共有n个元素，则全排列的个数为：

$$\frac{n!}{n_1!n_2!\cdots n_k!}$$

这个公式的推导方法类似于组合数：如果在可重排列的基础上在给所有相同元素标上特殊标记，则将得到n个全异元素的全排列。给第i种元素编号相当于给n_i个元素做全排列，方案数为$n_i!$，由乘法原理即得。

编码 设字母表大小为k，字符串长度为n，则每处理一个字符需要累加最多k个字符，每次累加需要进行模板计数，时间复杂度为$\mathrm{O}(k)$，因此处理一个字符需要$\mathrm{O}(k^2)$的时间，总$\mathrm{O}(k^2n)$。在处理过程中需要记录“当前各字母可选次数”数组c。注意到累加的过程中相邻两次计数用到的c实际上非常接近，所以可以用递推的方式从上一次计数计算出本次计数，每次计数只需O(5.14)的时间，总$\mathrm{O}(kn)$。

解码 和编码非常类似，也可以采取递推加速，是$\mathrm{O}(kn)$的。注意到全异排列的每一步是只需计算当前元素在可选集合里是第几大的。可以在$\mathrm{O}(\log n)$时间用二分查找得到结论。对于可重排列，也可以用类似的方法加速：把当前可选数目相等的元素看成同一种元素，当k很小或者很大时速度都很快。

枚举 考虑了编码和解码问题后，我们考虑枚举，即从小到大列举出所有元素。枚举通常有两种方法：递归枚举和加1枚举。递归枚举的思想是在当前可选集中依次选择各个元素，而加1枚举通过不断调用next函数，从一个元素转移到“比它大的最小元素”。和二进制加法类比，找到“比它大的最小元素” 分为两个步骤：需要找到“进位”的地方进一位，然后把后面改为最小值，例如：1010**0**1111的进位点是从右往左的第一个0位，而可重排列cba**c**dcba的进位点是从右往左第一个满足$c_i < c_{i+1}$的i，即i的右边是逆序，但(c_i, c_{i+1})是逆序。把c_i和c_{i+1}交换，然后c_{i+1}及其右边的所有字母颠倒过来（因此变成升序）。这就是STL中next_permutation的实现方法。如果发现整个串都是升序，那么加1失败，因为此串已经是最大串了。

小测验

1. 叙述编码问题、解码问题和枚举问题。它们和计数问题有联系吗？
2. 证明可重全排列计数公式，并用它进行编码和解码。
3. 枚举问题有哪两种常见思路？可以用编码和解码进行加1操作吗？这样做有什么优点和缺点？

5.3.3 递推关系与生成函数

递推关系是解决计数问题的有力武器，下面以常见的递推关系为例进行说明。

Fibonacci数 设一对雌雄兔子出生两个月后开始每月生下一对雌雄幼兔。如果开始有一对兔子，那么n个月后一共有几对兔子？假设答案为F_n，那么这些兔子有两部分组成，一部分是$n-1$个月时已有的兔子活下来的，另一部分是$n-2$个月时有的兔子生下来的，因此$F_n=F_{n-1}+F_{n-2}$，其中$F_1=F_2=1$。它的前10项为1, 1, 2, 3, 5, 8, 13, 21, 34, 55，读者应当记住。

Fibonacci数有很多有意思的性质，例如它可以由杨辉三角得到,如图8.1.2：

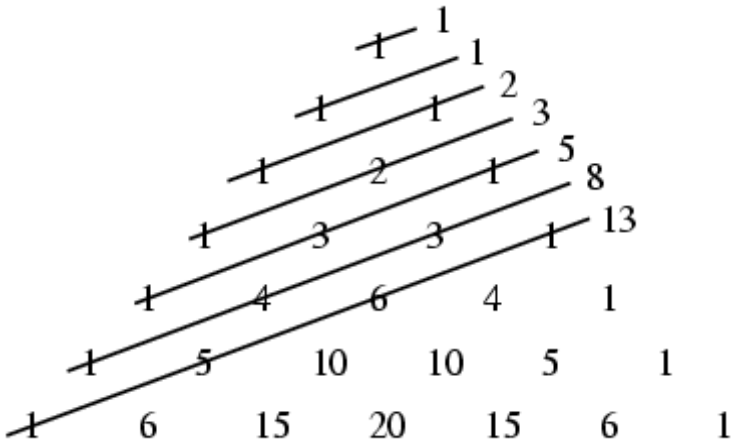


图 5.9: Fibonacci树和杨辉三角

如何计算第n个Fibonacci数？有两种方法，一种是直接按照定义计算，依次计算F_3, F_4, F_5，...直到算出F_n。由于只需要保留最近的两个数，因此时间复杂度为$O(n)$，空间复杂度为O(5.14)。事实上有更快的方法，时间复杂度仅为$O(\log n)$（忽略高精度运算）。

回忆前面学过的矩阵乘法。容易验证

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} b \\ a+b \end{pmatrix}$$

因此如果让a=F_{k-1}，b=F_k，则一次递推对应一次矩阵乘法。由于矩阵乘法满足结合律，因此

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-1} \times \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

和幂取模一样，我们用倍增法在O($\log n$)次矩阵乘法求出F_n。这个方法可以推广到任意常系数线性递推方程$f_i = a_1 f_{i-1} + a_2 f_{i-2} + \cdots + a_k f_{i-k}$中，只需取矩阵

$$A = \begin{bmatrix} a_1 & a_2 & a_3 & \cdots & a_k \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

和F_0=(f_{k-1} f_{k-2} ... f_0)，则F_n=A^n*F_0的最后一行即为所求。Fibonacci数还是{1, 2, ..., n}的不包含相邻整数的子集个数（包括空集）。请读者自己证明这个结论。

Catalan数 有三种等价的方式可以定义Catalan数C_n($n \geq 2$)。

定义*1* 将正n边形用对角线剖分为三角形的方法数为C_n

定义*2* n个数的乘积$a_1a_2...a_n$的不同结合方法数为C_{n+1}

定义*3* 在整数坐标平面的格子上，从点(0, 0)到点(n, n)由垂直和水平线段组成的路径，要求中间点(a, b)满足$a \leq b$（即必须在$y = x$的上面走），则路径条数为C_{n+2}

很容易得到前两种定义满足递推式

$$C_{n+1} = C_2C_n + C_3C_{n-1} + \cdots + C_nC_2(n \geq 2)$$

它是Catalan数最常用的一个递推式。在第3种定义中，递推式需要用如下方法得出：设路径中第一次到达的$y = x$上的点为(k, k)，则从(0, 0)到(k, k)必然是先向上走一步，然后不再经过直线$y = x$，到达(k-1,k)，进而到达(k,k)。由于这个过程并不在经过直线$y = x$，因此相当于在网格中按题目要求从(0,0)走到(k-1,k-1)（做一次坐标平移即可），次数为C_{k+1}。接下来从(k,k)走到(n,n)时没有限制，即相当于从(0,0)走到($n-k$,$n-k$)，次数为C_{n-k+2}。去$k = 1, 2, 3, ...n$，得$C_{n+2} = C_2C_{n+1} + C_3C_n + \cdots + C_{n+1}C_2(n \geq 1)$，它和刚才的递推式是一样的。

$C_2 \sim C_{10}$的值分别为1, 1, 2, 5, 14, 42, 132, 429, 1430，建议读者记住。直接根据递推式需要花费O(n^2)的时间才能计算出C_n，好在C_n可以写成更为简洁的形式：

$$C_n = \frac{1}{n-1} C_{2n-4}^{n-2}$$

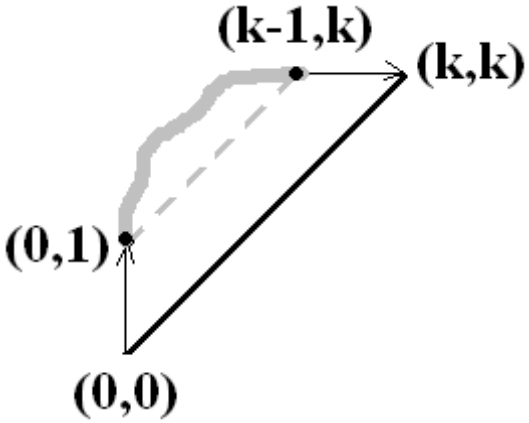


图 5.10: Catalan数的第三种定义

因此计算Catalan数归结到了计算组合数。

集合的划分 n元集合划分为k类的方案数记为$S(n,k)$，称为第二类Stirling数，它有递推公式：$S(n+1,k)=S(n,k-1)+kS(n,k)$，边界是$S(n,1)=S(n,n)=1$。这个递推式每次把$n$减小1，直到$n$和$k$一样大，有$S(n,n)=1$。这个递推式是容易理解的，因为$n+1$元集合是$n$元集合增加一个元素后得到。这个新元素被划分到哪一个集合呢？如果是新集合，则剩下还需要把n个数划分到$k-1$个集合；如果是已经有的k个集合中的一个，则只需要把n个数划分到k个集合，然后从k个集合中选一个。

用此递推式计算$S(n,k)$的时间复杂度为O(nk)。n元集合的所有划分数记为B_n，称为Bell数，它显然满足$B_n=\sum_{k=1}^{n}S(n,k)$，还满足递推公式：

$$B_{n+1}=\sum_{k=0}^{n}C_n^kB_k$$

这个递推式也是显然的。假设考虑元素1所在的集合除了1之外还有k个元素，则此类划分可以分两步构造：先选出k个元素，然后划分剩下的$n-k$个元素。由乘法原理即得。

采用两种方法计算的时间复杂度均为O(n^2)。B_0=1，从B_1开始的10个Bell数为：1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975。

生成函数 生成函数(generating function) 是处理数列的有力工具，如果把n作为输入的计数问题的解a_n看作一个数列的话，生成函数也可以解决计数问题。序列a_0, a_1, ...的生成函数是

$$A(x) = a_0 + a_1x + a_2x^2 + \cdots$$

它把整个数列的信息浓缩到了一个多项式中。它并没有给出数列的通项公式，甚至也没有给出有效计算方法，但它却是数学推导的有利工具。

Fibonacci数的递推公式是F_n=F_{n-1}+F_{n-2}，边界是F_1=F_2=1。设它的生成函数是A(x)，则

$$\begin{aligned} A(x) &= F_0 + F_1x + F_2x^2 + F_3x^3 + \cdots \\ xA(x) &= 0 + F_0x + F_1x^2 + F_2x^3 + \cdots \\ x^2A(x) &= 0 + 0 + F_0x^2 + F_1x^3 + \cdots \end{aligned}$$

第一个等式减去后两个等式，则x^2和更高项全部消去，因此有

$$(1 - x - x^2)A(x) = F_0 + F_1x - F_0x = x$$

这样，我们得到了F_n的生成函数：

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

一般排列组合问题。有k种元素，均有无穷多个。规定第i种元素选取的个数c_i必须属于一个给定的集合S_i，求选r个元素的排列数和组合数。显然，最基本的排列组合问题对应的每个S_i均为{0,1}，表示要么选（选1个），要么不选（选0个）。

对于每个元素k，我们把限制S_k写成序列A，其中A_i=1当且仅当i在集合S_k中（即：选i个元素k是允许的）。假设有苹果、香蕉和桃子三种水果，苹果只能选不超过3个，香蕉的个数必须是4的倍数，而桃子只能选2、3或5个，则三个水果的序列是：

水果	限制集合S	序列
苹果	{0, 1, 2, 3}	1, 1, 1, 1, 0, 0, 0, 0, 0, ...
香蕉	{4, 8, 12, ...}	0, 0, 0, 0, 1, 0, 0, 0, 1, ...
桃子	{2, 3, 5}	0, 0, 1, 1, 0, 1, 0, 0, 0, ...

设r组合数为c_r，那么c_r的生成函数是每个序列的生成函数之积。在这个例子中，c_r的生成函数是：

$$(1 + x + x^2 + x^3)(x^4 + x^8 + x^{12} + \cdots)(x^2 + x^3 + x^5)$$

选20个水果的组合数就是上式中x^{20}项的系数。

设r排列数为p_r，那么p_r的指数生成函数是每个序列的指数生成函数之积。其中序列a_i的指数生成函数被定义为

$$a_0 + \frac{a_1}{1!}x + \frac{a_2}{2!}x^2 + \frac{a_3}{3!}x^3 + \cdots$$

因此选20个水果的排列数为

$$(1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3)(\frac{1}{4!}x^4 + \frac{1}{8!}x^8 + \frac{1}{12!}x^{12} + \cdots)(\frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \frac{1}{5!}x^5)$$

的x^{20}的系数乘以20!的结果。生成函数还有很多应用，限于篇幅这里不再讨论，有兴趣的读者可以参考相关书籍。

小测验

1. 什么是递推关系？给出Fibonacci数、Catalan数、第二类Stirling数和Bell数的组合意义和递推公式。
2. 什么是生成函数？什么是指数生成函数？为什么说它们是序列信息的浓缩？
3. 简述一般排列组合问题，并证明本节介绍的生成函数解法是正确的。

5.3.4 等价类计数

本节介绍等价类计数问题。为此，先介绍置换群的概念。

置换。可以用这样的形式表示1被a_1取代，2被a_2取代，...，n被a_n取代。其中a_1，a_2，...，a_n是1∼n的一个排列。

$$\begin{pmatrix} 1 & 2 & \cdots & n \\ a_1 & a_2 & \cdots & a_n \end{pmatrix}$$

可以定义置换的连接运算：

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{pmatrix}\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{pmatrix}\begin{pmatrix} 3 & 1 & 2 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix}$$

可以验证全体置换关于连接运算构成群，根据子群判定定理，只要一个置换集合关于连接是封闭，那么它们也构成群，我们把这样的群称为置换群。设G是1∼ n的某个

置换群，k是1$\sim$ n的某个元素，把G中让k保持不变的置换集合，记为Z_k，称为k不动置换类。k在G作用下的轨迹记为E_k，也就是通过置换G能变换到的元素集合。

定理：若G是一个置换群，对于1$\sim$ n的任意元素k，有$|E_k| \cdot |Z_k| = |G|$

这个定理是本节其他定理的基础，证明略。

循环和循环节定义n阶循环如下：

$$(a_1 a_2 \cdots a_n) = \begin{pmatrix} a_1 & a_2 & \cdots & a_{n-1} & a_n \\ a_2 & a_3 & \cdots & a_n & a_1 \end{pmatrix}$$

每一个置换都可以分解为不相交的循环乘积，这一步只需要从1个元素出发“顺着置换走一遍”，在O(n)时间即可各个循环。例如

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 1 & 4 & 2 \end{pmatrix} = (1\ 3)(2\ 5)(4)$$

这个表示是唯一的（想一想，为什么），它包含的循环个数称为循环节。例如上例的循环节是3。

等价类计数问题。考虑这样一个问题：给2×2方格上着黑白两色，有几种方法？答案是16种，如图8.1.2：

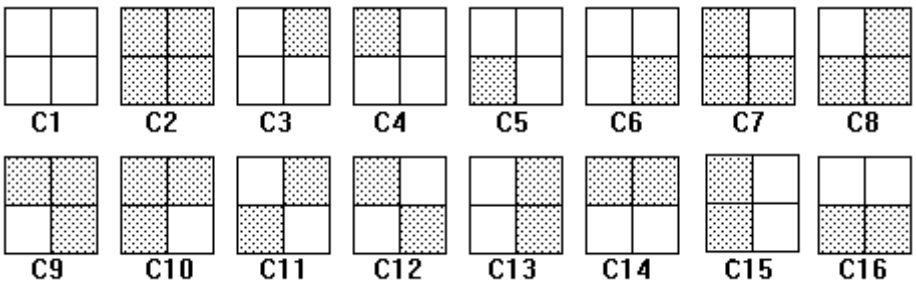


图 5.11: 2 × 2方格的着色方案

但如果我们定义一种“旋转操作”，规定逆时针旋转90度、180度或者270度后相同的方案作为同一种方案，那么答案就变成6种了：

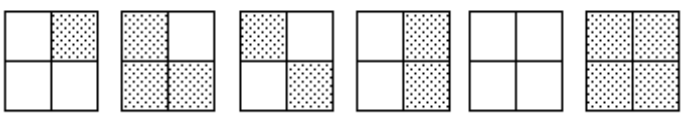


图 5.12: 2 × 2方格的等价着色方案

我们的问题是：对于$n \times n$的格子，涂m种颜色，有多少种涂法？注意到每种旋转操作实际上是一个置换。如果把2×2的格子按图8.1.3中方法进行编号

图 5.13: 2×2方格的编号

那么四个置换实际上是：

编号	旋转方式	置换
1	逆时针转0度	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix}$
2	逆时针转90度	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \end{pmatrix}$
3	逆时针转180度	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix}$
4	逆时针转270度	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$

表 5.5: 2×2方格的四种旋转操作

置换群为G={转0度、转90度、转180度、转270度}。由于四个置换关于连接是封闭的，因此这是一个合法的置换群。

Burnside引理 对于一个置换f，若方案s变换到自己，称它为f的不动点。f的不动点数目记为$C(f)$。注意这里的s是一个方案，即“一个元素到颜色的映射”。这样的映射是很多的。关于$C(f)$，有一个有用的结论：

Burnside引理：等价类数目为所有$C(f)$的平均值。

证明：考虑所有满足“方案s是置换f的不动点”的二元组$(s,\ f)$，则这样的二元组的个数既等于所有C_f之和也等于所有Z_k之和。假设有L个等价类，由于在同一个等价类中的Z_k相同，因此在计算所有Z_k之和时可以把每个等价类合并在一起，即：

$$\sum_f C(f) = \sum_k |Z_k| = \sum_L |E_{L(1)}| \cdot |Z_{L(1)}| = \sum_L |G| = |L| \cdot |G|$$

其中用到了前面介绍的定理，L(1)表示等价类L的任意一个元素。在刚才的问题中，转0度、90度、180度和270度的不动点分别有16、2、4、2个，因此等价类

有(16+2+4+2)/4=6个，和正确答案吻合。Burnside引理提供了一个计算等价类的方法，但直接用它并不能很快计算出结果。使用Burnside引理需要计算每个置换的不动点数。如果根据定义，需要检查每个方案是否在该置换下不动，这是不实际的，因为方案数（即元素到颜色的映射数）可能很大。能不能快速计算出不动点数呢？Pólya定理给出了一个很好的答案。

Pólya定理 给n个元素着k种颜色，设置换f的循环节为m(f)，则它的不动点数C(f)=k$^{m(f)}$。这样，不动点数不用枚举判断了，而只需要计算置换f的循环节，它只需要O(n)的时间。

证明： 考虑f的循环分解。为了让一个方案是f的不动点，在任何一个循环中只要有一种颜色固定，其他元素必须和它同色（想一想，为什么），而其他循环的元素不受任何限制。因此问题相当于给每个循环从k种颜色中选一种，答案自然是$k^{m(f)}$了。2×2的问题中转换0度、90度、180度和270度的循环分解式分别为：(5.14)(2)(3)(4)、(1 4 3 2)、(1 3)(2 4)、(1 2 3 4)，因此循环节分别为4, 1, 2, 1，等价类数目为$(2^4+2^1+2^2+2^1)/4=6$。

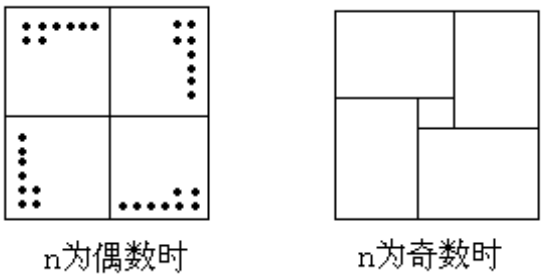


图 5.14: 一般$n \times n$格子着色问题的分析

对于给$n \times n$格子图m种颜色的一般情况，循环节可以通过数学分析直接得到，而不需要进行置换分解，这个工作留给读者去思考，最后整理出的答案是：

$$L = \frac{1}{4}(m^{n^2} + m^{[\frac{n^2+3}{4}]} + m^{[\frac{n^2+1}{2}]} + m^{[\frac{n^2+3}{4}]})$$

小测验

1. 什么是置换？什么是循环？为什么置换的循环分解是唯一的，如何有效的分解一个置换？推导一个1$\sim$ n的置换连接以一个二元循环(ab)（$1 \le a < b \le n$）的结果。

2. 什么是置换群？在置换群G中，任意元素k的不动置换类和轨迹有何关系？
3. 什么是Burnside引理？什么是Pólya定理？给出定理的描述并证明。直接使用二者进行计数的时间复杂度是多少？

5.4 组合游戏

本节讨论一个有趣的话题：组合游戏。这里的游戏指的是双人对弈游戏，我们曾在第4章中讨论过一般的方法。本节的不同之处在于这里的游戏是特殊的：它们很好的数学特性，使得我们能够找到可判定输赢的数学策略，而不需要进行状态空间的搜索。

5.4.1 基本概念和方法

组合游戏应满足以下性质：

1. 有两个游戏者。
2. 有一个可能的游戏状态集。这个状态集通常是有限的。
3. 游戏规则指定了在任何状态下双方的可能的走步和对应的后继状态集。如果在任意状态下双方的走步集合是相同的，那么说游戏是**公平的(impartial)**，否则是**不公平的(partizan)**。国际象棋是不公平的，因为每个人只能移动自己的子。
4. 两个游戏者轮流走步。
5. 当到达一个没有后继状态的状态后，游戏结束。在**普通游戏规则(normal play rule)** 下，最后一个走步的游戏者胜；在**misère游戏规则** 下，最后一个走步的游戏者输。如果游戏无限进行下去，我们认为双方打平，但通常我们会附加规定：
6. 不管双方怎么走步，游戏总能在有限步后结束。

其他规则包括：不允许随机走步（不能扔色子或者随机洗牌），且必须信息完全的（如隐藏走步是不允许的），有限步结束时不能产生平局。在本节中，我们只考虑公平游戏，并且通常只考虑普通游戏规则（最后走步的胜）。

P状态和N状态假设双方都采取最明智的策略，则对于一些状态，**刚完成走步的游戏者(Previous Player)** 一定胜利，而对于其他状态，**下一个走步的游戏者(Next Player)** 一定胜利。我们把两种状态称为**P状态(P position)** 和**N状态(N position)**，且有以下关系：

1. 所有终止状态是P状态

2. 能一步到达P状态的状态为N状态

3. 每一步都将到达N状态的状态为P状态

我们也可以把P状态称为必败状态，N状态称为必胜状态，含义是直观的。以上关系实际上给出了一个递推计算所有状态的P-N标号的算法。只要状态集构成一个n个结点m条边的**有向无环图(directed acyclic graph, DAG)**，则可以按照拓扑顺序在O(m)时间内计算所有状态的标号。可问题在于这样的状态往往有很多，能否通过数学方法直接判断一个状态是P状态还是N状态呢？

减法游戏 有n根火柴，双方轮流拿火柴，每次可以拿1根、3根或4根，最后一个拿的人获胜。用火柴的根数表示状态，显然唯一的终态为0，第一批确定的N状态是1、3和4，因为它们能一步到达P状态。用刚才的规则可以列出一个表5.4.1：

x	0	1	2	3	4	5	6	7
标号	P	N	P	N	N	N	N	P
x	8	9	10	11	12	13	14	15
标号	N	P	N	N	N	N	P	N

表 5.6: 减法游戏的标号

表5.4.1也可以用程序算：从小到大枚举x，检查从x出发所有一步可达的状态中有没有P状态（必输状态）。若有就是N状态，否则就是P状态。如果每次有k种选择，时间复杂度就是O(nk)。

通过表5.4.1很容易看出规律：一个状态x为P状态当且仅当$x \bmod 7$为0或2。这个结论很容易用数学归纳法证明，这里不再赘述。根据这个结论，我们可以在O(1)的时间判断是先手胜还是后手胜，而不用花O(n)时间慢慢递推了。

Ferguson游戏有两个盒子，一开始其中一个有m个糖而另一个有n个糖。把这样的状态记为(m, n)。每次走步是将一个盒子清空而把另一个盒子的一些糖拿到被清空的盒

子中，使得两个盒子各至少有一颗糖。显然唯一的终态为$(1,1)$。如果最后走步的游戏者获胜，那么在初始状态为(m,n)时，先手胜还是败？

Chomp!游戏有一个$m \times n$棋盘，每次可以取走一个方格并拿掉它右边和上面的所有方格。拿到左下角的格子(1, 1)者输。例如在8×3棋盘中拿掉(6, 2)和(2, 3)后的状态为：

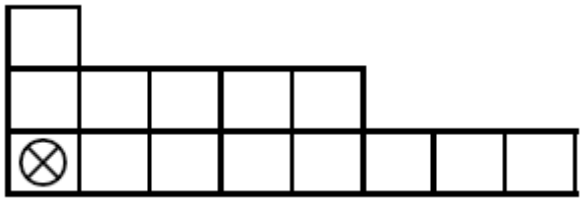


图 5.15: Chomp!游戏

其中取到左下角有标志的格子者输。这是一个N状态（先手胜）。可以证明只要初始状态是完整的矩形，则一定是先手胜（即矩形状态是N状态）。

动态减法游戏可以把前面的减法游戏加以扩展，让每个人取的火柴数取决于上一个游戏者取的游戏。初始有n根火柴，第一个人可以任意取，至少一根最多n-1根；以后每个人至少拿1根，但拿的火柴数

1. 不能超过上一次（另一个游戏者）拿的火柴数，或

2. 不能超过上一次（另一个游戏者）拿的火柴数的两倍

问先手胜还是先手败。

SOS游戏有n个方格排成一列，初始均为空。每次一个游戏者可以在任何一个空格中写上S或者O。如果某个游戏者让某三个连续的方格里写上“SOS”三个字母，则该游戏者获胜。如果所有方格都写满了字母但并没有SOS的出现，则双方和。

小测验

1. 什么是组合游戏？它满足哪些条件？
2. 什么是P状态和N状态？如果状态集形成一个n结点m边的DAG，如何在O(m)的时间计算每个状态的标号？
3. 分析本节提到的6个游戏，找出P状态和N状态，并给出数学证明。

5.4.2 Nim和与Sprague-Grundy定理

本小节介绍组合游戏中著名的Nim游戏和它的变种。

Nim游戏 有三堆火柴，分别有a, b, c根，记为状态(a, b, c)。每次一个游戏者可以从任意一堆中那走至少一根火柴，也可以整堆都拿走，但不能从多堆中拿火柴。最后一个拿火柴者为胜。Nim游戏的重要定理是L.Bouton在1902年给出的：

Bouton定理：在Nim游戏中，状态(x_1, x_2, x_3)为P状态当且仅当x_1 xor x_2 xor x_3=0，其中xor为二进值异或操作。这样的操作也称为**Nim和(Nim Sum)** 。

解释：状态(13, 12, 8)是P状态吗？我们可以进行二进制运算：

$$\begin{aligned} 13 &= 1101_2 \\ 12 &= 1100_2 \\ 8 &= 1000_2 \\ \hline \text{nim-sum} &= 1001_2 = 9 \end{aligned}$$

由于不是0，因此是N状态，先手必胜。第一步应该走什么才能保证胜利呢？显然应该把Nim和变为0，只需要在第1堆拿掉9根火柴即可。

直觉上，这个定理应该适用于4堆、5堆、...的情况，如何证明呢？用数学归纳法。

证明：如果每堆都为0，显然是P状态。下面验证P状态和N状态的后两个递推关系：

一、*每个N状态都可以一步到达P状态* 证明是构造性的。检查Nim和的二进制表示中最左边一个1，则随便挑一个该位为1的火柴堆，根据Nim和进行调整（0变1，1变0）即可。例如Nim和为100101011，而其中有一堆为101110001。为了让Nim和变为0，只需要让操作的火柴数取操作前的火柴数和Nim的异或即可。

项目	值
Nim和(X)	100101011
操作前某堆的火柴数(Y)	101110001
操作后该堆的火柴数(X xor Y)	001011010

表 5.7: 火柴数的二进制表示

显然操作后火柴数变小，因此和合法的。设操作前其他堆的Nim和为Z，则有Y xor Z = X。操作后的Nim和为X xor Y xor Z = X xor X = 0，是一个P状态。

二、每个*P*状态都不可以一步到达*P*状态 由于只能改变一堆的火柴，不管修改它的哪一位，Nim的对应位一定不为0，不可能是P状态。

这样就证明了Bouton定理。

misère规则Nim游戏如果最后取石子的输，情况是怎样的？初看起来问题要复杂很多（因为不能主动拿了，而要"躲着"拿，以免拿到最后一根火柴），事实上确实有很多游戏misère规则比普通规则要困难很多，但对于Nim游戏来说，几乎是一样的：首先按照普通规则一样的策略进行，直到恰好有一个火柴数大于1的堆x。在这样的情况下，只需要把堆x中的火柴拿得只剩1根或者拿完，让对手面临奇数堆火柴，每堆恰好1根火柴。这样的状态显然是必败的。由于你每次操作后需要保证Nim和为0，因此不可能在你操作后首次出现"恰好有一个火柴数大于1的堆"。新游戏得到了完美解决。

阶梯Nim游戏有n级阶梯，每级台阶上有一些硬币。每次可以在某个台阶上拿一些硬币放到下一个台阶。到地上（台阶0）后的硬币自动消失。设台阶i有x_i个硬币，则状态$(x_1, x_2, ..., x_n)$为P状态当且仅当$(x_1, x_3, x_5, ..., x_k)$对于Nim游戏是P状态，其中$n$为奇数时$k = n$，否则$k = n - 1$。

Nim$_k$游戏如果可以同时从k堆火柴里拿（至少要拿一根），会怎样？这样的游戏称为Nim$_k$游戏，由E. H. Moore于1910年提出。结论非常简单：把2进制异或换成k+1进制不带进位加法即可。

刚才提到过，状态构成了一个有限图。如果图是有限的，那么图应该是无环图；否则不仅不能有环，而且必须保证不管怎么走一定可以走到终态。在前面提到的减法游戏中，如果每次可以拿1根、2根或3 根火柴，则图如下：

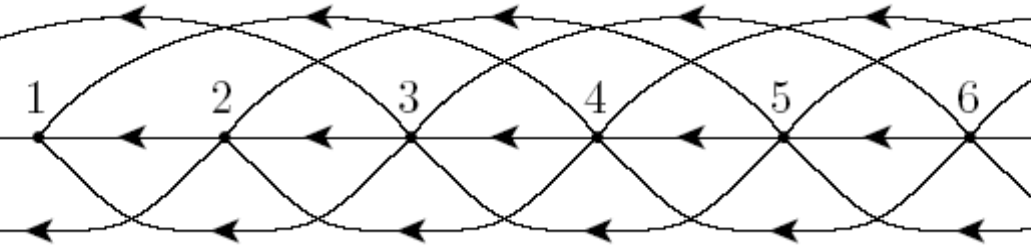


图 5.16: 组合游戏的有向图表示

在图上可以定义**Sprague-Grundy函数** $g(x)$为"不在后继点函数值集合中"的最小非负整数。如果记后继点函数值集合为S，则$g(x) = mex(S)$，其中$mex(S)$为不在S内的最小非负整数。例如，如果x有3个后继点，g值分别为0, 1, 1, 2, 4, 7，则$g(x) = 3$，

因为3是第一个没有出现在后继点函数值中的非负整数。这样，终态的g值显然为0，而其他值由拓扑顺序给出。根据递推关系，对于一个给定的图，SG函数显然是唯一确定的。

SG函数比状态标号有更多的信息。如果已知SG函数，那么显然P状态的SG函数值为0，N状态的SG函数值非0。下面是一个计算SG函数的例子。

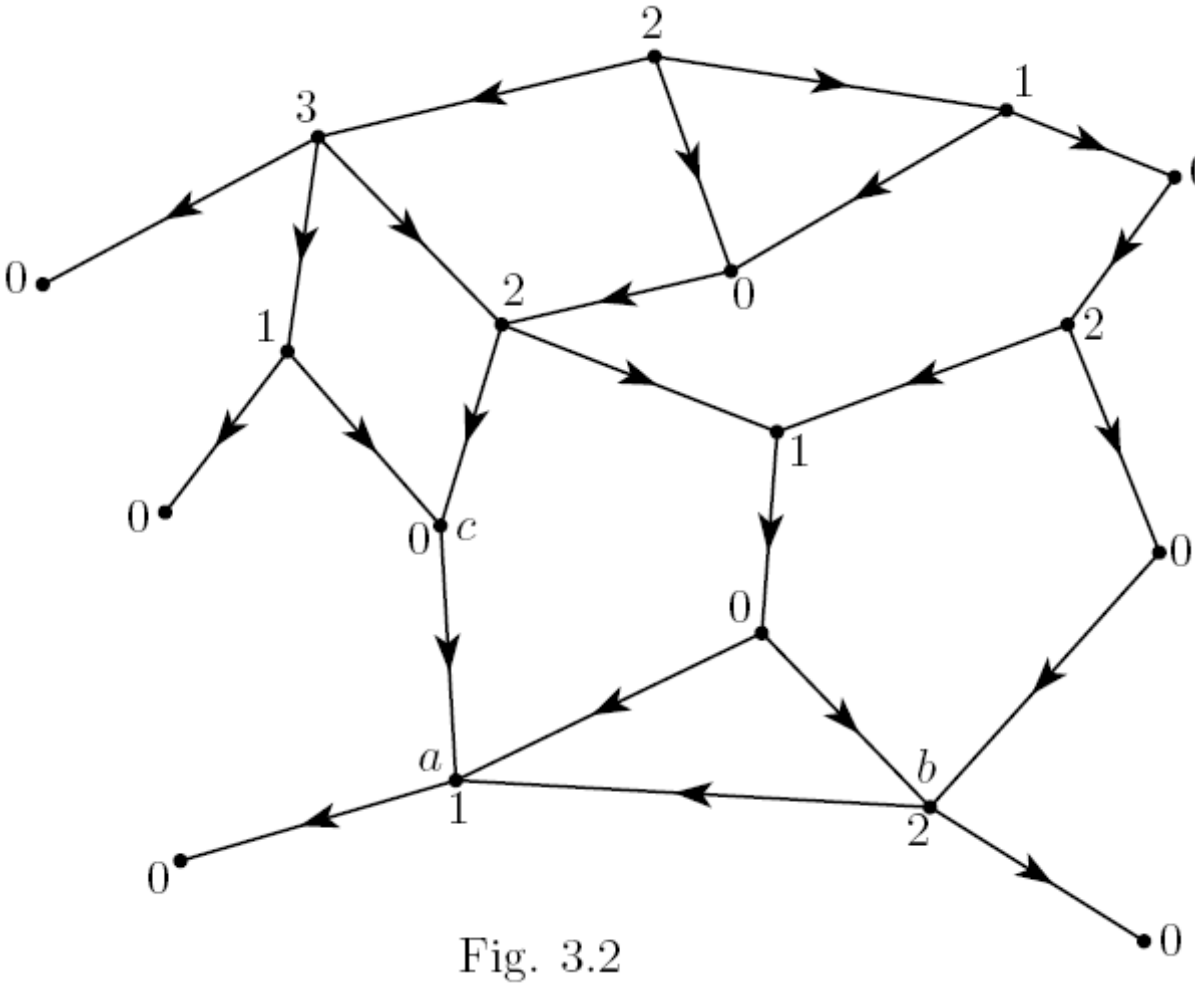


图 5.17: SG函数计算举例

对于取值为{1, 2, 3}的减法游戏，可以按定义计算SG函数如5.4.2：

如果只有一堆火柴，每次至少要取走一半，则SG函数如表5.4.2：

可以证明$g(x)$是满足$2^k > x$的最小k。不过这个游戏很无聊，因为可以一次拿走所有火柴！SG函数因此没有用了吗？不是的，SG函数允许我们进行局面加法，以处理多堆火柴的情形。

x	0	1	2	3	4	5	6
g(x)	0	1	2	3	0	1	2
x	7	8	9	10	11	12	13
g(x)	3	0	1	2	3	0	1

表 5.8: 减法游戏的SG函数计算举例

x	0	1	2	3	4	5	6
g(x)	0	1	2	2	3	3	3
x	7	8	9	10	11	12	...
g(x)	3	4	4	4	4	4	...

表 5.9: Nim游戏的SG函数计算举例

可以把“最少”改为“最多”：只有一堆火柴，且每次最少拿一根火柴，最多拿一半。这样就有两个终态：0和1。它的SG函数留给读者去推导。

Dim+游戏有一堆火柴。每次你可以拿掉c根火柴当且仅当c是火柴总数n的约数。例如在12根火柴中你可以拿走1，2，3，4，6或12根。唯一终态为0。如果不允许拿走整堆（即总数为12时可以拿走1，2，3，4，6根火柴），则游戏称为Aliquot游戏。

Wythoff游戏一个$n \times n$棋盘上有一个皇后。每个人可以把它往左或下或左下45度移动任意多步。把皇后移动至左下角的游戏者获胜。如果把每个格子看作一个状态，可以把棋盘上的格子写满它们的SG函数值。

组合游戏的和 可以用如下方式定义游戏的和：初始局面包含每个子游戏的初始局面，而每次每个游戏者可以任意选一个游戏，进行一次合法走步，而让其他游戏局面保持不变。在图上，我们可以认为游戏的和对应于图结点做直积。3堆火柴的Nim游戏可以作看是3个一堆火柴的Nim游戏之和。一堆Nim游戏是简单的：直接把所有火柴拿掉即可；但3堆火柴却复杂得多。看样子，即使每个游戏都很简单，它们的和也可能很复杂。

虽然看起来很复杂，但**Sprague-Grundy定理** 提供了一个很好的方案解决这一问题：游戏和的SG函数等于各子游戏SG函数的Nim和（二进制异或）。

设$x = (x_1, ..., x_n)$为组合游戏的一个状态，b为子游戏的SG函数和。我们需要证明两件事（根据定义）：

1. 对于任意非负整数$a < b$，$(x_1, ..., x_n)$的某个后继的SG值为a。

2. 不存在的$(x_1,...,x_n)$的某个后继的SG值为b。

证明并不困难，具体过程留给读者。

奇数偶数游戏 有n堆火柴，每次你可以任选一堆拿。在任何情况下，你可以拿偶数根火柴，但不能全部拿走；如果有奇数根火柴，可以全部拿走（也可以拿偶数根火柴）。对于一堆的情形，我们计算SG函数如表5.4.2：

x	0	1	2	3	4	5	6
g(x)	0	1	0	2	1	3	2
x	7	8	9	10	11	12	...
g(x)	4	3	5	4	6	5	...

表 5.10: 奇数偶数游戏的SG函数计算举例

规律是$g(2k) = k-1$，且$g(2k-1) = k$。这样，(10, 15, 20)的SG值为4 xor 7 xor 9 = 10，需要把第三堆的SG值从9变为3，即在第三堆拿走12根火柴，剩8根。而$g(8) = 3$。

有些游戏可以把一堆分成多堆，下面举三个例子。

Lasker's Nim游戏和Nim游戏类似，但每次可以把一堆分为两个不为空的堆（不拿走任何火柴）。这样，2的后继为0、1和(1, 1)，它们的SG函数为0, 1和1 xor 1 = 0，因此$g(2) = 2$。这样，容易得到表5.4.2：

x	0	1	2	3	4	5	6
g(x)	0	1	2	4	3	5	6
x	7	8	9	10	11	12	...
g(x)	8	7	9	10	12	11	...

表 5.11: Lasker's Nim游戏的SG函数计算举例

很容易用数学归纳法证明，$g(4k+1) = 4k+1$, $g(4k+2) = 4k+2$, $g(4k+3) = 4k+4$, $g(4k+4) = 4k+3$。

Kayles游戏有一堆火柴排成一排。每次可以拿走一根或者连续的两根（必须紧挨在一起，中间不能有空隙），最后取的获胜。下面是$g(y+z)$的表格。从$x = 72$开始，以12为周期循环。也就是说，表5.4.2里的最后一行将永远重复下去。最后一个例外是$x = 70$，在这之前的例外均用粗体标出。

Dawson棋盘游戏在一列格子里，双方轮流在格子里划叉，禁止在一个已经有X的相邻格子里划叉。不能画者输。在一堆的中间划叉相当于取走三个格子以后把它分成两堆，在边上画X相当于拿掉2个格子或者1个格子。可以计算出SG函数如表5.4.2：

在最后一个例外$x = 51$后，SG函数以34为周期循环。

yz	0	1	2	3	4	5	6	7	8	9	10
0	**0**	1	2	**3**	1	4	**3**	2	1	4	2
12	4	1	2	**7**	1	4	**3**	2	1	4	**6**
24	4	1	2	8	**5**	4	**7**	2	1	8	6
36	4	1	2	**3**	1	4	7	2	1	8	2
48	4	1	2	8	1	4	7	2	1	4	2
60	4	1	2	8	1	4	7	2	1	8	**6**
72	4	1	2	8	1	4	7	2	1	8	2

表 5.12: Kayles游戏的SG函数计算举例

x	0	1	2	3	4	5	6
g(x)	0	1	1	2	0	3	1
x	7	8	9	10	11	12	13
g(x)	1	0	3	3	2	2	4

表 5.13: Dawson棋盘游戏的SG函数计算举例

小测验

1. 叙述Nim游戏、misère Nim和Nim$_k$游戏的规则和必胜策略。
2. 如何把游戏用图的方式叙述？如何定义图的SG函数？编程计算本节提到的各种游戏的SG函数值。
3. 如何定义组合游戏的和？如何计算游戏和的SG函数？

5.4.3 Nim积与Tartan定理

经典的翻硬币游戏是这样的：一些硬币排成一列，有的正面朝上，有的反面朝上。每次可以选择一些硬币翻过来（正面变反面，反面变正面）。为了让游戏不会无限进行下去，还需附加规定每次翻转的最右边一枚硬币必须是从正面H翻到反面T。这样，每次操作后“最右边的H”都会左移，最终所有硬币变成T，无法继续移动。

这样的游戏一般还规定可以翻的硬币集合完全取决于当前最右边H的位置，而不取决于以前的走步或者其他位置的H。游戏是公平的，即任何状态下双方允许翻的硬币集合是相同的。

翻硬币游戏的一个重要结论是：可以把游戏状态分解，单独考虑各个H。换句话说，如果状态是THHTTH，则$g(THHTTH) = g(TH)$ xor $g(TTH)$ xor $g(TTTTTH)$。比如翻转左数第2个H，相当于在子游戏TTH中操作。也许读者会说：如果还需要把左数第1个H 也翻成T，岂不是必须在两个子游戏中同时操作了吗？不是的。只需要在子

问题TTH中进行这两个翻转操作，则TTH变为了TH，和已经有的TH抵消。这样，我们得到了一个重要的结论：分析这类翻硬币问题时，只需把“最右边的H的位置”作为状态表示，它是一个整数，比用{H, T}组成的二进制串简单多了。这样的状态（最右边的硬币为H，左边均为T）称为基本状态。

Mock Turtle游戏每次可以翻转1个、2个或3个硬币，其中最右边一个必须是H到T。我们从左到右设各枚硬币的位置为0，1，2...。

显然g(0)=1，因为状态H并不是目标状态但它的唯一操作将到达目标状态。g(5.14)=2，因为TH可以转换为HT或者TT，而$g(HT)=1$，$g(TT)=0$。$g(2)=4$，因为TTH可以转化为TTT, HTT, THT和HHT，而$g(TTT)=0$, $g(HTT)=1$, $g(THT)=2$, $g(HHT)=g(H)$ xor $g(TH)=1$ xor $2=3$。见表5.4.3：

x	0	1	2	3	4	5	6	7	8	9	10
g(x)	1	2	4	7	8	11	13	14	16	19	21

表 5.14: Mock Turtle游戏的SG函数计算举例

注意到$g(x)$要么是$2x$要么$2x+1$，究竟是哪一个呢？这要看x的二进制表示中1的个数了。实际上$g(x)$里的数恰好是所有二进制表示中1的个数为奇数的数，称为odious数，而二进制表示中1的个数为偶数的数称为evil数[3]。由数学归纳法容易证明：如果x是odious数，则$g(x)=2x$，否则$g(x)=2x+1$。

有了基本状态的SG函数，我们很容易扩展到任意状态。由于基本状态的SG函数要么是$2x$要么是$2x$+1，所以从第1位起的高位相当于所有x做Nim和。由于奇数个oidous数的Nim和为odious数，因此P状态中一定有偶数个H。这样，我们实际上证明了：H集合$\{x_1, x_2, ..., x_n\}$为Mock Turtle游戏的P状态当且仅当n为偶数且$\{x_1, x_2, ..., x_n\}$是Nim游戏的P状态。

Ruler游戏可以翻任意多枚连续硬币，但最右一枚必须是从H翻到T。这次位置从1开始编号，则$g(n)=mex\{0, g(n-1), g(n-1)$ xor $g(n-2), ..., g(n-1)$ xor ... xor $g(5.14)\}$。计算得表5.4.3：

x	1	2	3	4	5	6	7	8	9	10	11	12
g(x)	1	2	1	4	1	2	1	8	1	2	1	4

表 5.15: Ruler游戏的SG函数计算举例

[3] odious变形自单词odd，evil变形自单词even，具有调侃的意味。

可以找到规律：$g(x)$是能整除x的2的最大整数幂。

二维翻硬币游戏可以把翻硬币游戏扩展一下，把硬币排成一个矩阵，左上角为(0, 0)，任意硬币的坐标为(x, y)，其中$x, y \geq 0$。而“最右硬币必须由H翻成T”的规则被改为：“最右下的硬币必须由H翻成T”，其中(x, y)为被翻转的最右下的硬币当且仅当所有被翻转的硬币(a, b)都满足$a \leq x$, $b \leq y$。有的翻转方式不存在“最右下的硬币”，这样的翻转方式是非法的。

Acrostic Twins游戏每次翻两个硬币，要么同一行，要么同一列，其中右下方的那个必须由H翻成T。显然$g(x, y) = mex\{g(x, b), g(a, y) : 0 \leq b < y, 0 \leq a < x\}$，它等于没有在左上方出现过的最小非负整数。当$y = 0$时等价于普通Nim游戏。容易计算出表5.4.3：

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	0	3	2	5	4	7	8	9	8
2	2	3	0	1	6	7	4	5	10	11
3	3	2	1	0	7	6	5	4	11	10
4	4	5	6	7	0	1	2	3	12	13
5	5	4	7	6	1	0	3	2	13	12
6	6	7	4	5	2	3	0	1	14	15
7	7	6	5	4	3	2	1	0	15	14
8	8	9	10	11	12	13	14	15	0	1
9	9	8	11	10	13	12	15	14	1	0

表 5.16: Acrostic Twins游戏的SG函数计算举例

表5.4.3是令人惊奇的：$g(x, y)$恰好就是x xor y！

Turning Corners游戏每次翻一个矩形的四个角$(a, b), (a, y), (x, b), (x, y)$，其中$0 \leq a < x$, $0 \leq b < y$。显然$g(x, y) = mex\{g(x, b)$ xor $g(a, y)$ xor $g(a, b) : 0 \leq a < x, 0 \leq b < y\}$，显然上边界和左边界的硬币不可能是最右下的一个，因此$g(x, 0) = g(0, y) = 0$。表5.4.3是SG函数表：

Nim积表5.4.3看起来似乎没有一点规律，但这只是表面现象。我们把这样的$g(x, y)$称为x和y的**Nim积(Nim product)**，用$\otimes$表示。Nim积和普通乘法有很多类似的地方：

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	3	1	8	10	11	9	12	14
3	0	3	1	2	12	15	13	14	4	7
4	0	4	8	12	6	2	14	10	11	15
5	0	5	10	15	2	7	8	13	3	6
6	0	6	11	13	14	8	5	3	7	1
7	0	7	9	14	10	13	3	4	15	8
8	0	8	12	4	11	3	7	15	13	5
9	0	9	14	7	15	6	1	8	5	12

表 5.17: Turning Corners游戏的SG函数计算举例

$$
\begin{aligned}
&x \otimes 0 = 0 \otimes x = 0 \\
&x \otimes 1 = 1 \otimes x = x \\
&x \otimes y = y \otimes x \\
&x \otimes (y \otimes z) = (x \otimes y) \otimes z
\end{aligned}
$$

如果和Nim和一起考虑，它还满足分配律：$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$。更重要的，所有非零元素都有一个唯一的逆。这可以在表中看出：每行每列恰好一个1。根据这个结果可以定义Nim除法。事实上，非负整数在Nim和与Nim积下构成一个域。

Nim积的计算。下面考虑如何用非递推的方式计算Nim积。考虑数$F_n = 2^{2^n}$(n=0, 1, 2...)，即2, 4, 16, 256, 65536等，它们有以下性质：

1. 若$x \neq y$, 则$F_x \otimes F_y = F_x \cdot F_y$

2. $F_x \otimes F_x = (3/2)F_x$

根据这两条性质，我们计算任意2的幂，即$f(x, y) = 2^x \otimes 2^y$。把$x$和$y$写成二进制的性质，则

$$
\begin{aligned}
2^x &= 2^{2^{x_0}} \otimes 2^{2^{x_1}} \otimes \cdots \otimes 2^{2^{x_p}} \\
2^y &= 2^{2^{y_0}} \otimes 2^{2^{y_1}} \otimes \cdots \otimes 2^{2^{y_q}}
\end{aligned}
$$

其中x_i和y_i分别是x和y的二进制表示中为1的那些位。两式相乘，统计同时出现在两边的项数k，则$2^x \otimes 2^y$就是所有互不相同的项相乘后再乘以$(3/2)^k$。时间复杂度为$\mathrm{O}(\log n)$。其中$n = max\{x, y\}$。

有了这个结果，现在可以计算任意数x和y的Nim积了。首先把x和y写成二进制，然后用分配律，并展开成$\log^2 n$项乘积（其中$n = max\{x, y\}$），每一项都形如$2^a \otimes 2^b$，其中a和b不超过$\log n$，因此每一项需要$O(\log \log n)$的时间，总$O(\log^2 n \times \log \log n)$。如果用递推的方式计算$f(x, y)$，则时间复杂度可以进一步降低：

$$f(x + 1, y) = f(x, y) \otimes 2$$

代入前面的式子，根据情况可以知道2^0是新多出来的一项（直接乘进去）还是重复项（乘3/2），无论哪种情况都可以在常数时间内算出。类似也可以递推出$f(x, y + 1)$。这样，计算Nim积的时间复杂度降为了$O(\log^2 n)$。

Tartan定理 给定两个二维翻硬币游戏G_1和G_2。如果$x_1, x_2, ..., x_m$在G_1里是合法操作，而$y_1, y_2, ..., y_n$在G_2里是合法操作，则(x_i, y_j)（$1 \le i \le m, 1 \le j \le n$）是$G_1 \times G_2$里的合法操作（同时满足“右下”规则）。这样Turning Corner实际上就是Twins×Twins。如果G_1和G_2都是一维翻硬币游戏，那么有：

Tartan定理：如果$g_1(x)$是G_1的SG函数，$g_2(x)$是G_2的SG函数，则$G_1 \times G_2$的SG函数值为：$g(x, y) = g_1(x) \otimes g_2(y)$。

例如Twins的$g(x) = x$，因此Turning Corners的$g(x, y) = x \otimes y$。另一个有趣的游戏是Rugs（或称Turnablock），它等于Ruler×Ruler，即可以翻一个连续的矩形。它的SG函数可以由Ruler的SG函数作Nim积得到，如表5.4.3：

	1	2	1	4	1	2	1	8
1	1	2	1	4	1	2	1	8
2	2	3	2	8	2	3	2	12
1	1	2	1	4	1	2	1	8
4	4	8	4	6	4	8	4	11
1	1	2	1	4	1	2	1	8
2	2	3	2	8	2	3	2	12
1	1	2	1	4	1	2	1	8
8	8	12	8	11	8	12	8	13

表 5.18: Rugs游戏的SG函数计算举例

Tartan定理虽然很容易计算SG函数，但有时却不容易找到必胜策略。一个典型的例子是Mock Turtle×Mock Turtle。由于Mock Turtle规定只能1个、2个或者3个硬币，因此Mock Turtle×Mock Turtle是选1∼3行再选1∼3列，翻转交叉点上的硬币。下面是

一个游戏局面。前面说过Mock Turtle的SG函数是1, 2, 4, 7, 8, 11, ...，因此可以得到表5.4.3。

T	H	T	T	T	T
T	T	T	T	T	T
T	T	T	T	T	T
T	T	T	T	T	T
T	T	T	T	T	H

	1	2	4	7	8	11
1	1	2	4	7	8	11
2	2	3	8	9	12	13
4	4	8	6	10	11	7
7	7	9	10	4	15	1
8	8	12	11	15	13	9

表 5.19: Mock Turtle×Mock Turtle游戏的SG函数计算举例

这个局面的SG值为2 xor 9 = 11，因此是一个N局面。如何走才能保证胜利呢？我们需要让操作后的SG值变为0。由于最后一行和一列已经固定（根据最右下规则），因此还需要选择0∼2行和0∼2列。选择0∼2行有1+4+4×3/2=11种选择，选择0∼2列有1+5+5×4/2=16种选择，因此一共有176种可能的走步。哪一种走步后SG值为0呢？

假设状态为(x, y)，SG函数为$g_1(x) \otimes g_2(y) = v$，且你希望把它变成SG值为$u$。令$v_1 = g_1(x)$且$v_2 = g_2(y)$，首先在Turning Corners里找出$(v_1, v_2)$到$u$的策略$(u_1, u_2) - (v_1, v_2)$。接下来在一维$G_1$里寻找从$g_1(x)$到$u_1$里操作$M_1$，同理找到$M_2$，则$M_1 \times M_2$为所求。

用刚才的例子。$(v_1, v_2) = (8, 11)$，找到$(u_1, u_2) = (3, 10)$（只需验证$(u_1 \otimes u_2) \oplus (u_1 \otimes v_2) \oplus (v_1 \otimes u_2) = u$）接下来分别寻找一维情形下从8到3和11到10的操作，前者是{1, 2, 5}，后者是{2, 5, 6}。

假设有n行n列，则寻找u_1和u_2需要O(n^2)次枚举。一维情形下如果有f种操作，则只需要2f的操作就可以通过枚举找到M_1和M_2，一共O($n^2 + f$)，而不是直接枚举的O(f^2)。当f很大时（例如Mock Turtle游戏的f为O(n^2)），这个方法比直接枚举快。

小测验

1. 叙述Mock Turtle游戏、Ruler游戏、Twins游戏和Turning Corners游戏。

2. 什么是Nim积？它有哪些性质？如何有效计算两个数的Nim积？
3. 叙述Tartan定理。如何寻找一个Tartan游戏的必胜走步？

第6章　离散结构上的算法

本章讨论离散结构上的算法。典型的离散结构，例如序列、树、图和字符串有各自的数学特性，由此产生出各种各样的算法，其中不乏经典之作。

6.1　序列上的问题

本节介绍序列上的算法，包括排序、统计问题、序列上常用的数据结构等。虽然序列的结构比较简单，但本节介绍的很多算法都是非常巧妙的。

6.1.1　线段树和树状数组

有很多动态问题需要用到专门的数据结构。本节介绍其中两种。

线段树 线段树(interval tree) 是把区间逐次二分得到的一树状结构，它反映了包括归并排序在内的很多分治算法的问题求解方式。

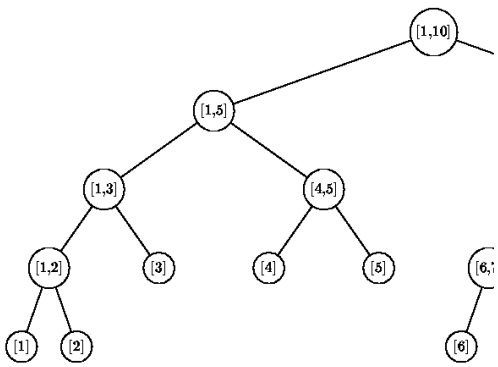


图8.1是一棵典型的线段树，它对区间[1,10]进行分割，直到单个点。这棵树的特点是：

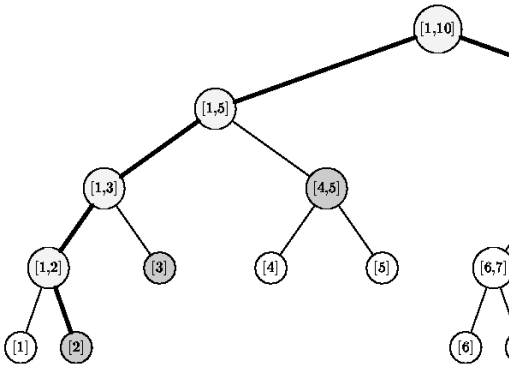
1. 每一层都是区间$[a,b]$的一个划分，记$L=b-a$

2. 一共有$\log_2 L$层

3. 给定一个点p，从根到叶子p上的所有区间都包含点p，且其他区间都不包含点p。

4. 给定一个区间$[l,r]$，可以把它分解为不超过$2\log_2 L$条不相交线段的并。

其中第四点并不是很显然，图8.1显示了[2, 8]的分解方式，深灰色结点为分解后的区间，浅灰色结点是递归分解过程中经过的结点。为了叙述方便，下面称树中的结点所对应的区间为树中区间。



从第3点和第4点可以得出结论：修改一个点只用修改$\log_2 L$个树中区间信息，而统计一个区间只需要累加$2\log_2 L$个树中区间的信息，且访问的总结点数是$O(\log L)$的。两个操作都很容易实现。

动态统计问题I 有一个包含n个元素的整数数组A，每次可以修改一个元素，也可以询问某一个区间$[l,r]$内所有元素的和。如何设计算法，使得修改和询问操作的时间复杂度尽量低？

方法一 直接做，则修改操作是$O(1)$的，但询问需要进行累加，时间复杂度为$O(r-l)$，最坏情况为$O(n)$。

方法二 建立一棵线段树，每个树中区间记录该区间的元素和，则由刚才的结论，修改元素时只需要修改$\log_2 L$个树中区间的元素和，而统计时只需要累加$2\log_2 L$个树中区间的元素和，两个操作都是$O(\log n)$的，比方法一好很多。

动态统计问题II 有一个包含n个元素的整数数组A，每次可以同时给一个区间$[l,r]$内的数同时增加一个数d（d可以为负），也可以询问某一个数A_i的值。如何设计算法，使得修改和询问操作的时间复杂度尽量低？

如何快速修改一个区间？修改一个树中区间$[a,b]$将影响到以$[a,b]$为根的整棵子树和它的所有祖先，所以如果沿用刚才的线段树定义，是不可能快速实现区间修改操作的。

解决方法是借用数据结构中常用的“懒操作”的思想，只记录有哪些操作需要执行，而不去真正执行这些操作。换句话说，在需要给树中区间$[l,r]$的元素同时增加d时，我们只记录“曾经有一条指令add(l, r, d)”就可以了。我们把记录的这个值称为树中区间的add值，则叶结点的元素值为它所有祖先的add值之和。

根据前面的结论，每一条任意区间的修改指令可以分解成$2\log_2 L$个树中区间的修改指令，且修改操作具有叠加性，因此修改操作的时间复杂度为$\mathrm{O}(\log n)$。查询操作只需要累加叶子的所有祖先，它也是$\mathrm{O}(\log n)$的。

动态统计问题III 有一个包含n个元素的整数数组A，每次可以同时给一个区间$[l,r]$内的数同时增加一个数d（d可以为负），也可以询问一个区间$[l,r]$内所有元素的和。如何设计算法，使得修改和询问操作的时间复杂度尽量低？

显然前两个问题都是此问题的特殊情况，因此这个问题比前两个问题难度更大。注意到上一个问题线段树只能提供叶结点的真正元素和，因为对于任何一个树中区间$[l,r]$来说，影响它的指令所修改的区间不仅包括它的所有祖先，也包括它的所有后代。所以$[l,r]$内的所有元素和应该等于$[l,r]$的所有祖先的add值加上$[l,r]$所有后代的add值。

但后代有很多，直接累加的时间开销过大。这里需要再一次利用线段的树的区间相加性质，记录一个附加值sum_of_add，表示以$[l,r]$为根的子树上所有树中区间的add值之和。

修改操作 把区间分解为树中区间，修改它们的add值，并且要顺便修改它们父亲的sum_off_add值。由于区间分解过程访问的总结点是$\mathrm{O}(\log L)$的，因此修改操作是$\mathrm{O}(\log n)$的。

查询操作 把区间分解为树中区间，并统计它们所有祖先的add值之和，再与这些树中区间本身的sum_off_add相加即可。

线段树的应用并不限于此，以后我们会看到更多例子。

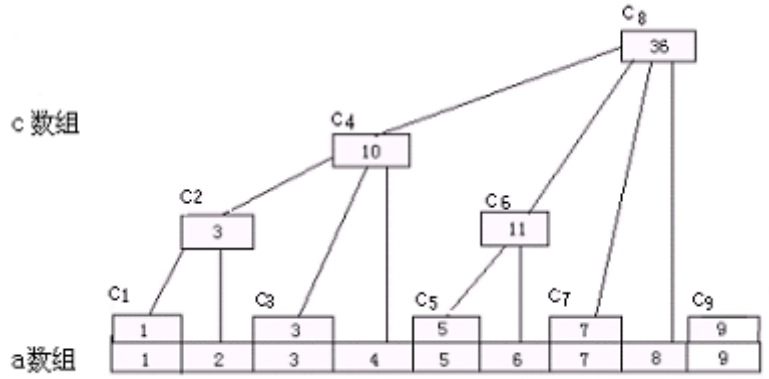
树状数组 **树状数组** 是一个很有意思的数据结构，它的应用没有线段树广，但是对于一类特定问题，它的程序非常容易编写。

再谈动态统计问题I 设数组$C[i]=a[i-2^k+1]+[i-2^k+2]+...+a[i]$，其中$k$为$i$在二进制形式下末尾0的个数，则和式的起点是把$i$的最后一位1变成0，再加1。根据定义，

我们有

i	C[i]的定义式	C[i]的递推式
1	a[1]	a[1]
2	a[1]+a[2]	c[1]+a[2]
3	a[3]	a[3]
4	a[1]+a[2]+a[3]+a[4]	c[2]+c[3]+a[4]
5	a[5]	a[5]
6	a[5]+a[6]	c[5]+a[6]

可以把递推关系整理成一棵树，其中i为奇数时$c[i] = a[i]$。



修改操作 修改一个$a[k]$时，设受到影响的C序列为$C[p_1], C[p_2], ..., C[p_m]$，则$p_1 = k$, $p_{i+1} = p_i + 2^{l_i}$，其中l_i为p_i在二进制中末尾0的个数。这样，修改元素的方法是：如$a[k]$需要增加x，则数组$c[p_1], c[p_2], ..., c[p_m](p_m \le n < p_{m+1})$都应该增加$x$。例如在$a[1]$, $a[2]$, ..., $a[9]$中，往$a[3]$增加x，则$p_1 = k = 3$, $p_2 = 3 + 2^0 = 4$, $p_3 = 4 + 2^2 = 8$, $p_4 = 8 + 2^3 = 16 > 9$，因此应该给$c[3]$, $c[4]$, $c[8]$增加x。

求和操作 有一个类似的公式是$k_1 = k$, $k_{i+1} = k_i - 2^{si}$，其中s_i为k_i在二进制数中末尾0的个数，则$S = c[k_1] + c[k_2] + c[k_3] + ... + c[k_m]$，其中$k_{m+1} = 0$。例如计算$a[1] + a[2] + ... + a[7]$，则$k_1 = 7, k_2 = k_1 - 2^0 = 6, k_3 = k_2 - 2^1 = 4, k_4 = k_3 - 2^2 = 0$，因此结果为$c[7] + c[6] + c[4]$。

这样，修改和求和都是$\mathrm{O}(\log n)$的，且空间开销比线段树省，程序也简单。

小测验

1. 什么是线段树？如何修改和查询点？如何修改和查询区间？其中用到了数据结构中的哪些思想？

2. 叙述三个动态统计问题和解法。

3. 什么是树状数组？推导修改和求和操作时下标的递推公式。

6.1.2 RMQ问题和扩展

下面介绍RMQ问题和它的扩展。**范围最小值（Range Minimal Query，RMQ）**问题是一种动态查询问题，它不需要修改元素，但要及时回答出数组A在区间$[l,r]$中最小的元素值。这个问题可以通过线段树来解决：每个树中区间记录该区间的最小元素，则每次询问只需要比较分解后的$\log n$个树中区间。建树需要O(n)的时间，而询问需要O($\log n$)的时间。

对于这样的问题，我们通常关心两方面的时间：预处理时间$f(n)$和查询时间$g(n)$，并用$f(n)-g(n)$来描述算法的时间效率。换句话说，刚才基于线段树的算法是O(n)-O($\log n$)的。

一般RMQ问题的Sparse Table(ST)算法：这个算法记录了所有长度形如2^i的所有询问的结果。用$d[i,\ j]$表示从i开始的，长度为j的区间内的RMQ，则有递推式：

$$d[i,\ j] = \min\{d[i,\ j\text{-}1],\ d[i+2^{j-1},\ j\text{-}1]\}$$

即用两个相邻的长度为2^{j-1}的块的取更新长度为2^j的块。预处理的时间复杂度为$O(n\log n)$。

询问时只要取k=[$\log_2(j-i+1)$]，那么令A为从i开始的长度为2^k的块，B为到j结束的长度为2^k的块，那么A和B都是$[i,j]$的子区间，但是A和B一起将覆盖$[i,j]$。这样，A的最小值与B的最小值的较小者就是RMQ(i,j)，即：

$$\text{RMQ}(i,j)=\min\{d[i,k],d[\ j\text{-}2^k+1,k]\}$$

这个算法不难编写，它是O($n\log n$)-O(1)的，对于询问很多的情况下比线段树优。

±1-RMQ问题。下面介绍一种特殊的RMQ问题：±1-RMQ。它的特点是：相邻两个元素要么相差1，要么相差-1（不能相同）。如何利用这个性质呢？显然，把一个数组的所有元素同时减去一个相同的数，则修改后的数组对于询问将给出和原数组相同的最小值位置！换句话说，在±1-RMQ问题中，长度为n的数组本质不同的只有2^n个！这样，一个新的算法产生了：

把数组A划分成每部分为L=$\log_2 n/2$的小块，则一共有$2n/\log_2 n$块

用O(n)的时间求出所有小块的最小值，令A'[i]表示第i个小块的最小值。

可以用O($2n/\log_2 n\times$ $\log(2n/\log_2 n)$)=O(n)时间内做好ST算法的预处理

下面的任务是在O(1)的时间内回答A'上的RMQ询问。

对于一般的询问RMQ(i, j)，可以先求出i所在的块编号x和它在块中的下标a，以及j所在的块编号y和它在块中的下标b。

如果$x = y$，则执行块内RMQ：In-RMQ(x, a, b)，表示第x块中下标a到b的最小值。

否则区间[i, j]分成三部分，即在块x中，从i到块末的最小值In-RMQ(x, a, L)，在块y中从块首到j的最小值In-RMQ(y，1，b)以及第x+1块到y-1块的最小值RMQ(A'，x+1，y-1)。RMQ询问的结果为：min{In-RMQ(x,a, L), In-RMQ(y, 1, b), RMQ(A', x+1, y-1)

	第x块下标a到块末下标L)		第x块块首（下标1）到下标b	

第x块第x+1～ y-1块第y块

图1-28 区间划分

由于使用了ST算法，已经可以在O(1)的时间内计算出RMQ(A'，x+1，y-1)了，因此我们的重点是计算第x块内从下标a到下标b的最小值In-RMQ(x，a，b)。还记得刚才的定理吗？由于所有的小块长度均为L=$\log_2 n/2$，所以它们最多只有$2^L = n^{1/2}$种本质不同的数组，每个数组最多有$L \times L = O(\log^2 n)$种询问。因此用完全递推法用O($n^{1/2}\log^2 n$)的时间事先求出所有可能数组的所有询问，再用O(n)的时间计算出每个小块属于哪个数组，就可以用查表的方法在O(1)的时间内算出所有In-RMQ的值，问题得以解决。

一般RMQ问题可以转化为±1-RMQ问题，具体方法将在后面LCA部分学到。

允许修改的情形 对于有修改的情况，刚才的算法一律失效，但线段树仍然可用。修改一个元素将影响$\log_2 n$个树中区间的最小值，而询问只需要比较分解后的$2\log_2 n$个树中区间记录的最小值。这里需要注意的是，修改操作比动态统计问题I复杂。在动态统计问题I中，给一个元素A_i增加值d只需要把叶子i的所有祖先的值增加d，而在这里，每个祖先修改后的值需要采取递推式计算：min(父亲) = min(左儿子, 右儿子)。

把这个现象加以推广，我们得到了基于线段树两个相当重要的条件：

1. 要想进行修改操作，父亲的值必须可以由左儿子和右儿子算出

2. 要想进行区间统计操作，整体的值必须可以由局部的值推出。

查询k小值的情形 如果我们需要的查询范围内的第k小值，又改如何修改线段树？注意，父亲的k小值不能由左右儿子的k小值推出，因此它不满足刚才所说的条件，因此在修改时无法进行递推。同理，由于整体的k小值不能由局部的k小值推出，所以它也无法进行区间统计。

怎么办呢？我们只能采取一种迂回的方法求k小值，把它转化为一种可以递推的东西。为了让修改可以进行，我们必须设计一种二分数组的方法求数组里的k小值。这里需要一点创造性思维：假设已经知道一个"可能的k小值"x，如何判断它是否为正确答案呢？我们只需要统计有多少个数比x小即可。如果恰好有$k-1$个数比x小，那么它一定是第k小数。（这里假设所有元素都不同。有相同元素的情况要复杂一些，需要记录和x相同的数有多少个，但基本方法是一样的）。如果比x小的数太小，说明x需要增大；如果比x小的数太多，则x需要减小。这样，只需二分x，每次判断x是否为正确答案即可。

这个方法很迂回，但好处在于"统计有多少个数比x小"的工作可以分到左半线段和右半线段分别完成再合并在一起，所以这个方法可以应用到线段树中。

"统计有多少个数比x小"可以通过扫描数组完成，但显然如果事先将数组排序，这一步将做得更快（二分查找）。换句话说：线段树中的每个树中区间必须保留该区间内所有元素的有序数组。这看起来需要较多的时间，但实际上只用一个归并排序就可以完全建立整棵线段树。由于每一层恰好包含所有n个元素，因此建树的时间复杂度仍为$\mathrm{O}(n\log n)$。

综合起来一下。

预处理 用一个归并排序完成，时间复杂度为$\mathrm{O}(n\log n)$。设$A[l,r]$表示树内区间$[l,r]$所有元素对应的有序数组。

查询 二分答案x，根据比x小和相等的元素个数决定x应增大还是减小。统计元素个数可只需要把分解后的$2\log_2 n$个区间的统计结果相加，而每个区间的统计需要一次有序数组中的二分查找。假设所有元素均为正整数，且不超过C，则二分次数不超过$\log C$次，总时间复杂度为$\mathrm{O}(\log C\log^2 n)$。实际的次数比它小得多，因为这二分次数、树中区间个数和树中区间长度都可以远比估计值小。

最困难的是修改操作，因为在一个有序数组里修改元素可能会很慢。由于我们使用有序数组只是希望快速的进行"统计有多少个数比x小"，所以为了让修改操作变得快

速，需要把有序数组改为排序二叉树。这样，我们得到了一个以排序二叉树为结点的线段树，是一种数据结构的复合。这样：

修改 以修改元素对应的叶结点的所有祖先的排序二叉树中更新元素（删除旧的，增加新的），时间复杂度为$\log n + \log(n/2) + \log(n/4) + ... = \log n + \log n - 1 + \log n - 2 + ... = \mathrm{O}(\log^2 n)$。

小测验

1. 什么是RMQ问题？这里介绍的几种算法分别适用于怎样的情形？
2. 为了在线段树上修改和进行区间统计，树中结点记录的信息需要满足怎样的条件？
3. 什么是数据结构的复合？以动态范围k小值问题为例说明。

6.1.3 更多排序算法

到现在为止我们学习了快速排序和归并排序，并且学习了它们的变形，包括随机选择算法、逆序对统计算法等。排序算法很多，在这里不可能一一介绍，因此我们把它们分类，重点叙述其思想而不是算法细节。未加说明的情况下，我们按照升序排列。

交换排序 通过交换逆序元素来达到排序的目的，例如冒泡排序就是这样的排序。如果每次只能交换相邻元素，则时间复杂度受到逆序对数的制约，不可能很快完成。

选择排序 依次得到第1小、第2小、第3小...元素。这些算法包括一般选择排序（每次遍历数组以选择当前最小元素）、堆排序（先建堆，然后每次从堆中取出最小值并删除）。选择排序的好处是很方便的修改成部分排序，即只取前k小元素。

增量排序 依次考虑各个元素，使得考虑下一个元素时可以借助当前结果，而不需要重新计算。插入排序的好处是在输入有间隔的情况下可以有效的利用时间，而不需要像选择排序一样必须等到所有数据输入完毕才能开始工作。插入排序就是这样一个算法。

另外，还需要注意到基本有序数据时插入排序较快，而快速排序是不稳定的（关键码相同时，原来在前面的排序后不一定还在前面）。

这些排序方法都是基于比较的，也就是说排序对象必须满足**全序** 关系。在一般情况下统一使用快速排序，需要建立线段树或者逆序对统计用归并排序框架，而需要增量排序或者部分排序时用堆排序，数据基本有序时用插入排序，几乎不需要用到其他

排序方法。但是如果元素并不是黑盒子，除了比较之外还可以访问元素的内部结构，那么下面介绍的算法也许会快很多。前面证明过基于比较的排序的下界为$\Omega(n \log n)$，而下面的算法并不受到此限制。

计数排序 如果关键码的范围不太大，**计数排序(counting sort)** 是合适的。它简单的把关键码为k的元素扔到桶k中。如果用链队表实现各个桶，则算法是稳定的。

基数排序 下面将要叙述的这种来自"洗牌机"的算法称为**基数排序(radix sort)** 。它把元素看成是d元组，目标是按照字典序给它们排序。

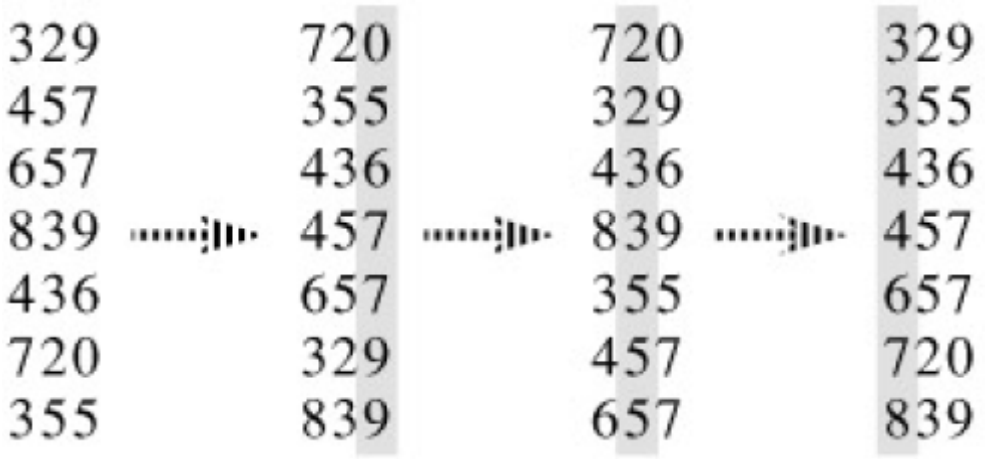
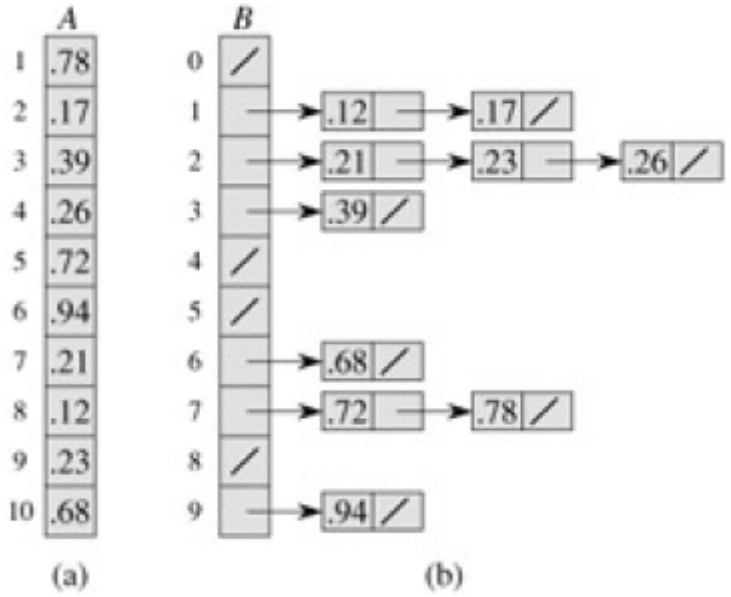


图8.1显示了基数排序的思想：从最低位开始，每次以一位为关键码进行稳定排序，通常采用计数排序。由于是按照字典序排列，基数排序经常被用于给诸如日期（年、月、日三元组）、定长字符串等对象排序。如果每部分有k种取值，则排序时间为$O(d(n+k))$。显然，把不超过2^{32}的整数看成每元65536种取值的2元组比计数排序快很多（而且空间也节省很多）。

桶排序 和基数排序类似，**桶排序(bucket sort)** 也假设元素具有某种性质：在[0, 1)之间平均分布。算法把[0, 1)分成n个相等的区间，像哈希表一样每个区间使用一个桶。如果分布不太平均，会出现多个元素在同一个桶中的情况，需要用插入排序给每个桶进行排序，然后从小到大连接各个桶。

字符串排序 给变长字符串排序可以简单的用快速排序实现，但这样做并不是最好的：每次字符串比较实际上包含了很多次字符比较。可以用基数排序来给字符串排序。假设字符串的最大长度为d，那么给n的字符串排序等价于给n个d元组排序，其中较短字符串需要添加"比所有字符都小"的特殊字符。这样，排序的时间复杂度为$O(d(n+k+1))$，其中k为字母表大小。这样的算法显然是不明智的，因为如果有



一个字符串极长而其他字符串都很短，那么算法的初期将耗费大量时间在“把一个字符和本来不存在的特殊字符比较” 这样的愚蠢事情上。如果n个字符串的长度分别为1，1，1，1...，1，d，那么算法所做的绝大多数事情都是不必要的。

如果事先建立一个列表length_i，表示长度恰好为i的字符串列表，则给第i位排序时，只需要把length_i附加进来考虑即可，每个字符恰好参与一次运算，总时间复杂度为$O(L+k)$，其中L为所有字符串的字符总数。显然对于刚才的例子，新算法只需要$O(n+d+k)$的时间，比原来了$O(d(n+k+1))$提高了不少。

小测验

1. 什么是基于比较的排序算法？它们的最坏情况时间复杂度下限是什么？有哪三种设计思路？
2. 叙述计数排序、基数排序和桶排序。为什么基数排序必须以稳定排序为子过程？
3. 如何在$O(L+k)$时间内给字母表大小为k，总长为L的若干字符串排序？

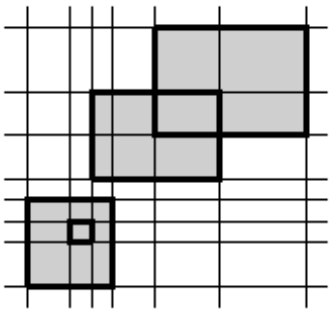
6.2 矩形的算法

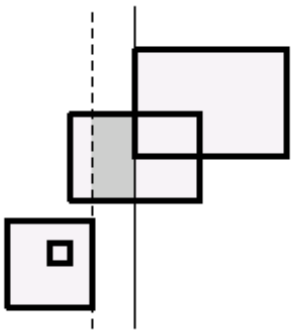
本节介绍矩阵上的算法，包括常用数据结构、最大空矩形、最大空正方形等问题，理论性不强，但算法很有启发性。

6.2.1 数据结构和算法思想

很多数据结构是从一维情形推广而来，但却比一维要灵活得多。线段树和树状数组都可以推广到二维情形，但有的东西并不是显然的。

扫描法 虽然有二维情形下的数据结构，但如果可能，最好降维。对于静态统计问题来说，矩形问题降维的最常见方法是扫描。这里有一个例子：给出n个边平行于坐标轴的矩形，它们并的面积。由于矩形的坐标范围没有确定，但是取值是固定的，因此可以先进行离散化，如图8.1.1。





这样，我们得到了$n \times n$个“元矩形”，每个元矩形要么完全被覆盖，要么完全没有被覆盖，没有中间状态。这样，可以设计一个布尔数组表示每个元矩形是否被覆盖，每次考虑一个矩形时标记它所覆盖的所有元矩形为“已覆盖”。由于每个矩形最多覆盖n^2个元矩形，这个算法是$O(n^3)$的。有没有更好的方法呢？我们借助扫描法。

如图8.1.1，在相邻两条竖线之间的区域是矩形。先把所有竖线按照从左到右排序，然后从左到右进行扫描，左右竖线（在程序实现中分别用+1和-1来表示）分别标

志着矩形进入和退出阴影区域，也就会涉及到线段的插入和删除操作。在每一个竖条中被覆盖的矩形面积等于当前的线段总长度乘以竖条宽度，因此需要在扫描过程中累加每个竖条面积，框架如图8.1.1：

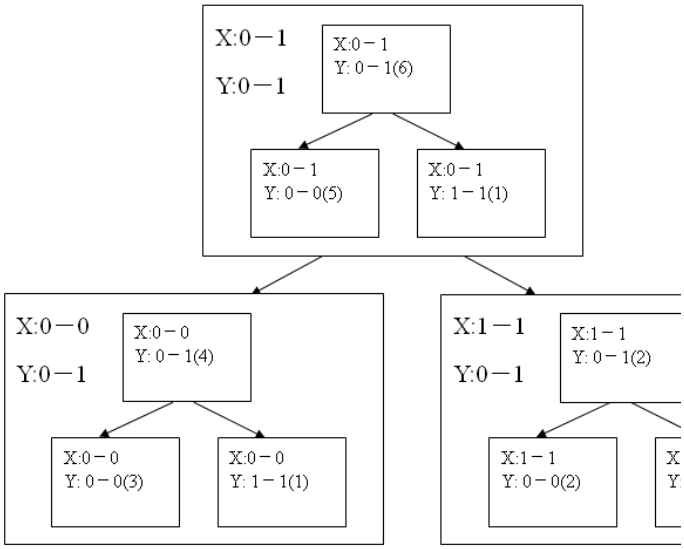
```
1:  last_x := 0;
2:  for e = GetLeftmost(Edges) do begin
3:     (x, y0, y1, sign) := e;
4:     area := area+sum[1]*(x-last_x);
5:     last_x := x;
6:     if (sign = +1) then
7:        modifyInterval(1, y0, y1, 0, max_
8:     else
9:        modifyInterval(1, y0, y1, 0, max_
10: end;
```

现在需要设计线段树高效的支持增加线段、删除线段和计算总长度三种操作。

由于修改的是区间，所以这里仍然采取“懒方法”，记录每个树中区间上已有的线段条数，又由于统计的也是区间，我们记录每个树中区间“被其他线段覆盖的总长度”。这里的“其他线段”为没有完全覆盖该线段的线段。显然记录的两种信息满足递推性质，所以修改和统计可以进行。

二维线段树 动态统计问题往往需要借助二维数据结构。给一个$n \times n$矩阵，可以修改每一个元素的值，也可以统计某个矩形区域$(l,t)-(r,b)$内所有元素和。这显然是一维的动态统计问题I的推广。它的解决方案——线段树是否也能推广到二维情形呢？答案是肯定的。首先按照X分，每个结点对应一棵树，此树是X范围固定时关于Y的线段树。其中主树中的结点称为结点树。如图8.1.1。

这又是数据结构的复合。在这样一个复杂的复合线段树上，如何进行修改和统计呢？



修改操作 对于主树来说，修改点p的所有祖先结点树（这些矩形的Y范围都是$1 \sim n$）都应该修改，因为这些矩形的x区间包含了该点。这些祖先每个都是一棵树，因此修改操作应该在每棵树中进行，修改对象仍然是修改点的所有祖先矩形，因为这些祖先矩形的y区间包含了该点，且根据本结点树的性质，x区间也包含该点。由于主树有$\log n$个结点树被修改，而每个结点树有$\log n$个结点被修改，因此修改的时间复杂度为$O(\log^2 n)$。

统计操作 矩形内的树之后可以经过变形转化为4个形如$(0,0)$-(a,b)的统计，类似于一维情形下的“前缀和”方法，如下图。

0,0						
					R,T－1	
		L,T				
	L－1,B				R,B	

这样，我们仍然采取类似于一维的情形，每次往右走时处理左边的结点。在主树中最多处理$\log n$个结点树，而在每个结点树里最多处理$\log n$个结点，因此总时间不超过$4\log^2 n$。其实也可以不做转化直接在线段树中遍历，则一个矩形内分解为不超过$4\log^2 n$个矩形。

修改矩形 如果像一维动态统计问题一样，修改操作是给一个矩形的每个元素增加

一个定值d，又当如何？还可以使用懒方法吗？如果采用懒方法，修改自然没有问题，那么统计呢？对于X坐标来说，仍然需要累加它的后代和祖先中Y坐标包含p点的矩形的add值。这样的累加是有选择的，而不像一维情形中一样是全部加起来。

一维情形的处理方法是：记录“后代中所有add值之和”。这是可以递推的，递推时间仅为O(1)。对于二维情形，可以记录每个矩形“后代中所有add值之和”。这也可以理解为每个树中矩形“被其他操作修改” 的总量。总结一下，

修改：把矩形分解为$4\log^2 n$个树中矩形，修改每个矩形的add值和所有祖先的sum_of_add。虽然矩形有$O(\log^2 n)$个，每个的修改时间最坏为$O(\log n)$，但由于和一维情形类似的道理，这些矩形的祖先结点个数和为$O(\log^2 n)$，因此修改的时间复杂度为$O(\log^2 n)$。

统计 把矩形分解为$4\log^2 n$树中矩形，则总修改量为每个分解矩形的add和sum_of_add之和。

这样，两个操作的时间复杂度均仍是$O(\log^2 n)$。

二维树状数组只是简单的把每个一维的C继续进行分解，这里不再赘述。有兴趣的读者可以自己试一试。

小测验

1. 以计算矩形并的面积为例，叙述如何用扫描法进行降维，并用离散化减小线段树规模
2. 叙述如何在二维线段树中修改点、统计矩形和修改矩形。
3. 把树状数组推广到二维情形。

6.2.2 数字矩形相关问题

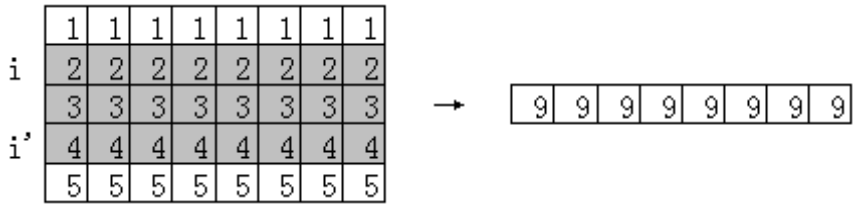
本节讨论若干和数字矩形有关的问题，包括和最大子矩形，最大空矩形等问题。在这些问题中，输入都是$n \times m$矩阵，每个格子里有且仅有一个数，左上角为$(1, 1)$，右下角为(n, m)。

和最大子正方形 找一个正方形，使得里面所有数的和尽量大。注意格子里的数可以是负的，因此不一定面积大的正方形里所有数之和也大。为了考虑“所有数的和”，一般采取前面提到过的转化方法：转化为四左上角固定的矩形中数的和。这样，只要计算出所有左上角为$(1, 1)$的矩形内的数之和，就可以用O(1)的时间计算任意矩形内数之和了。

如何计算所有左上角为(1, 1)的矩形内数之和？设$d[i,j]$表示矩形(1, 1)-(i, j)的数之和，则可以由$d[i,j]$计算$d[i+1,j]$。$d[i+1,j]$和$d[i,j]$的差别如图8.1.2所示，是第i+1行前j个元素的和$s[i+1,j]$，它可以再通过递推得到：$s[i+1,j]=s[i+1,j-1]+A[i+1,j]$。由次，用O($nm$)时间计算出每行$i$的前缀和$s[i]$，再递推即可得到$d[i,j]$。

这样，只需要枚举左上角和边长，在O($nm \times min\{n,m\}$)时间内可以解决本题。

和最大子矩形 找一个矩形，使得里面所有数的和尽量大。如果仍然用刚才介绍的方法，需要枚举左上角和右下角，时间复杂度为O(n^2m^2)。这里可以使用前面提到的降维思想，枚举起始行和终止行后，把整个矩形压缩成一维的，如图8.1.2。



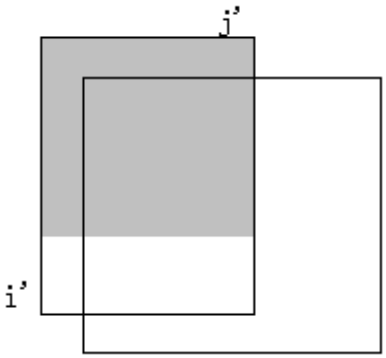
这个一维问题是简单的：设$d[i]$表示以第i个数结尾的最大连续子序列之和，则当$d[i-1]$为正时$d[i]=d[i-1]+A[i]$，否则$d[i]=A[i]$，整个问题的答案为所有$d[i]$的最大值。注意每个“压缩数”$d[i]$可以由两个列前缀和相减，因此整个算法的时间复杂度为O($nm \times min\{n,m\}$)。如果把行压缩该为列压缩，那么列前缀的预处理也可以省略（因为已经计算了行前缀）。

最大空正方形 如果所有数非0即1，我们可以把1看成障碍，0看成空地。一个经典问题是：求最大空矩形。和刚才一样，我们先考虑正方形的情形。设$d[i,j,0]$和$d[i,j,1]$表示以(i,j)为左上角的最大空正方形的行列编号。

如何计算$d[i,j]$？可以从$d[i+1,j+1]$递推，如图8.1.2。显然$d[i,j]$的行列不会超过$d[i+1,j+1]$的行列。实际上，这个最大空正方形取决于从(i,j)往右和往下能够延伸到的最远处。设它往右最远可以延伸到第j'列，而往下最远可以延伸到i'列，则i', j'和$d[i+1,j+1]$共同决定了$d[i,j]$：i'超过$d[i+1,j+1,0]$是没有意义的，同理j'超过$d[i+$

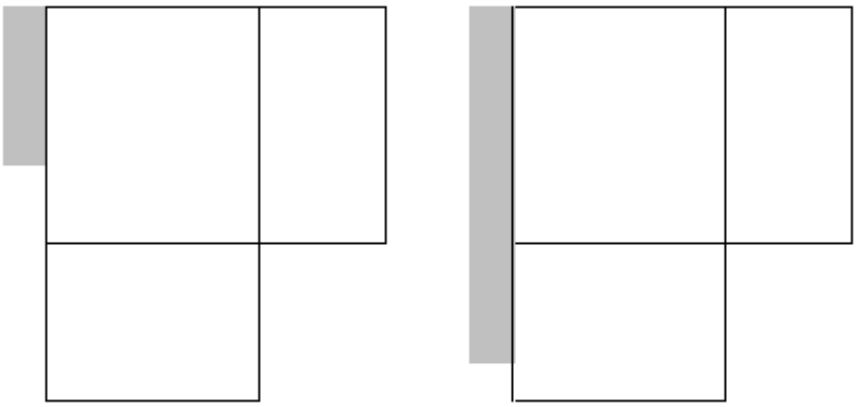
$1, j+1, 1]$也是没有意义的，所以先更新i'为$min\{i', d[i+1, j+1, 0]\}$, 更新$j'$为$min\{y', d[i+1, j+1, 1]\}$。

当前$j'-j$和$i'-i$的较小值为以(i, j)为左上角的最大空正方形边长。



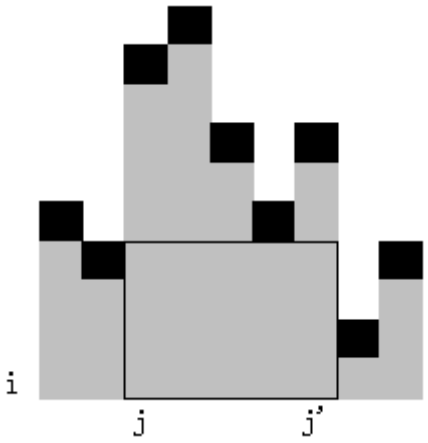
这样，递推过程是O(1)的。如何求一个点往右往下能走多远呢？这一步也是递推的，留给读者思考。这样，问题在O(nm)的时间内得到了解决。

最大空矩形 把刚才的问题扩展到任意矩形的情况，则$d[i, j]$的递推式变得复杂了。$d[i, j]$甚至不可能由$d[i, j+1]$递推得到：



以$d[i, j+1]$为左上角的最大空矩形可能有两个，如果从(i,j)出发往下延伸得比较短，选择列数多的矩形比较划算（图8.1.2中左图），否则选择列数少的矩形比较划算（图8.1.2中右图）。如果我们只记录了一个最大矩形，势必会在其中一种情况下丢失最优解；如果两个同时记录，那么在矩形有很多的情况下记录信息过多，速度很慢。

解决方法是从上到下逐行扫描。显然对于当前行来说，每列j延伸到的最上行$last[j]$才是关键的，如图8.1.2。



对于任意一个列区间$[j, j']$，以它为底的最大矩形的起始行为$i' = max\{last[j], \ldots, last[j']\}$，因此最大矩形面积为$(j' - j + 1) \times (i - i' + 1)$。每增加一行，我们需要在尽量短的时间内求出让这个值最大的区间$[j, j']$。

显然对于每个区间来说，起决定因素的是那个最大的last。如果枚举这个“最低列”j，那么只需要把它尽量往左延伸（只要延伸到的列不比它更低），再尽量往右延伸，就得到了以它为最低点的最大区间。延伸过程可以用类似的递推法来得到，从两个方向扫描第i 行各一次即可。

这样，每一行需要O(m)的时间（更新last值，扫描求出最左最右延伸值，枚举最低列），总时间复杂度为O(nm)，达到了理论下界。

小测验

1. 如何把一个子矩形的和转化为4个以(1,1)为左上角的矩形内的数之和？
2. 如何求解和最大矩形？如何把它扩展到和最大长方体？
3. 如何求解最大空矩形？为什么说它达到了理论下界？

6.2.3 平面矩形和点相关问题

本节讨论几个平面矩形和点的相关问题。这些问题一般涉及到一个平面区域和n个点，需要求一个满足条件的矩形。本节的题目都要求找到的矩形边界与坐标轴平行，且落在一个给定区域内部。由于点的坐标可以任意，所以往往需要先离散化。这些问题和上一节介绍的问题的区别是：点没有长度和宽度，而且输入规模往往用点数n而不是离散化后的长、宽来度量。上一节各个问题的理论时间复杂度下界为O(nm)，而本节为O(n)。

最大空矩形 找一个内部不包含任意点的面积最大的矩形（边界和顶点处可以有点）。借助离散化，可以用上一小结介绍的方法求解，这里不再赘述。这里介绍纯粹另外几种算法。首先需要确定的是：最大矩形的每条边上至少有一个点，否则可以继续向该方向扩展，得到更大的矩形。这样，我们得到了一个有用的结论：最大矩形一定是极大矩形。

这样，我们可以枚举矩形的四个边界所对应的点，再判断是否有点在该矩形内，总时间复杂度为$O(n^5)$，显然无法承受。如果只枚举下边界和左边界，我们可以发现当上边界往上移动时，右边界往左移动。这样时间复杂度降为$O(n^3)$。另一种方法也是$O(n^3)$的：枚举上下边界，然后转化为一维问题，与前面介绍的和最大子矩形类似。

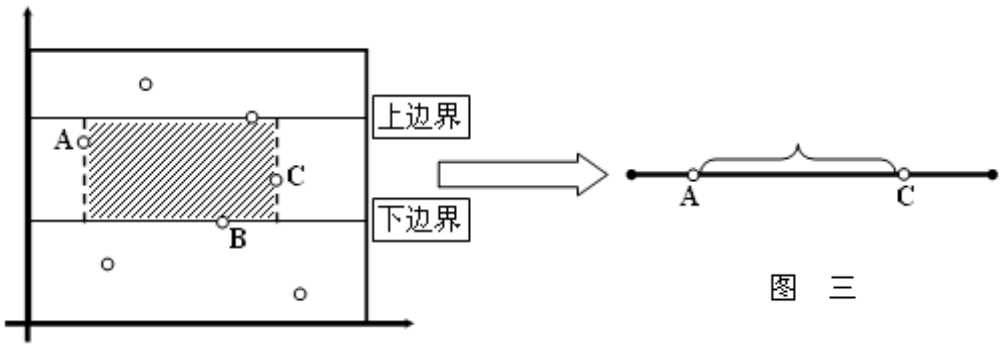


图 三

注意到下边界往下移动时，得到的新一维问题和刚才非常相似（仅仅是增加一个点），如图8.1.3：

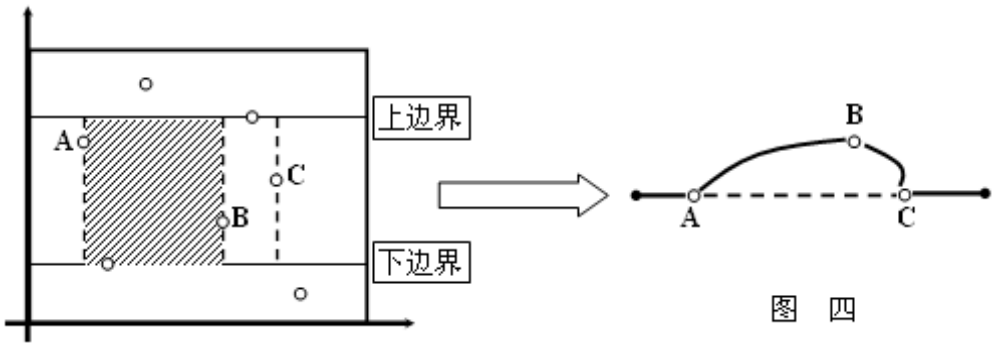
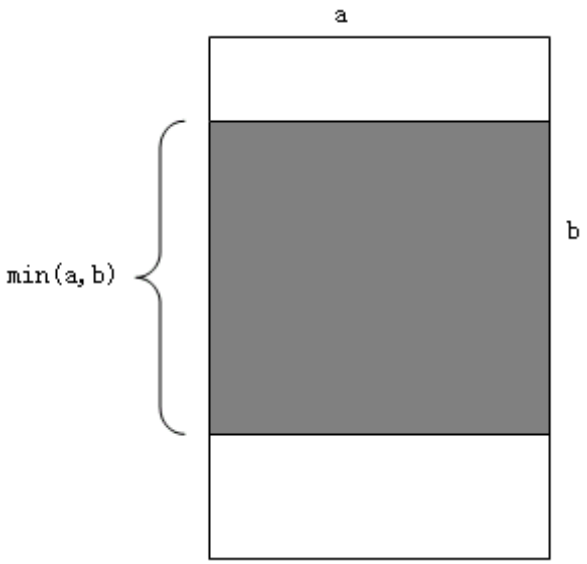


图 四

这样，我们需要维护当前信息，支持增加点操作和查询最大线段长度。经过离散化后，线段树或者静态排序二叉树都可以维护各个点之间的序关系。增加点B时只需要在长度集合里删除长度AC并添加长度AB和BC。由于每次需要找最长线段，用堆表示长度集合是合适的。这样所有操作均为$O(\log n)$时间内完成。只需要枚举上边界，然后不断移动下边界即可，总时间复杂度为$O(n^2 \log n)$。

细心的读者注意到了：在确定上边界后，下边界也可以是从下往上移动的，这样需要在点集中不断删点。由于每次删点后将得到一条更长的线段，所以只需要不断比较当前最长线段和新生成的线段即可。点集用链表维护，因此删除和求新线段长度都是O(1)的，总时间复杂度仅为O(n^2)。

最大空正方形 把刚才的题目限制一下，要求找出的矩形必须是正方形。虽然刚才的方法表面上不适用了，但是应注意到每个极的空正方形都可以扩展为极的空矩形。只要最大空矩形的算法枚举了所有极大空矩形（例如最后介绍的反向移动下边界的算法），都可以用来解决最大空正方形问题：只需要找出每个$a \times b$的极大空矩形中包含的最大正方形，它的边长为$min\{a, b\}$，如图8.1.3。



包含指定点数的最小正方形。假设所有点的x、y坐标都不同。给定$2 \leq k \leq n$，找出内内部恰好包含k的点的最小正方形。首先仍然是离散化，接下来二分正方形的边长。极小正方形的边长应该是某两点的某坐标差，因此一共至于O(n^2)种可能的边长。边长给定后，考虑n^2个可能的左下角位置，统计每个正方形中包含的点数。为了快速统计，在离散化后先要进行递推处理（类似于上一小节），就可以在O(1)时间内求出任意正方形内的点数。但由于需要知道此正方形右上角所在的离散点位置，需要再次进行二分查找。这样，在边长固定的时候需要O($n^2 \log n$)时间，总O($n^2 \log^2 n$)。

包含点数至少为且尽量接近指定数的矩形 假设所有点的x、y坐标都不同。给定$2 \leq k \leq n$，找出内部至少包含k个点的矩形，包含尽量少的点。选择矩形必须以两个

不同点的两个相对顶点（这两个点也算做被此矩形所包含）。

最简单的方法就是枚举两个顶点再统计点的个数，时间复杂度为O(n)。例如前面用过很多次的技巧，可以进行预处理，即在离散化后计算每个交叉点左上方的点数，但这样做需要O(n^2)的空间。一个比较好的方法是先固定一个点，并计算它和其他点形成的n个交叉点左上方的点数。但这n个点可以只在O(n)时间内递推出来吗？这个问题留给读者思考。

下面的算法看上去直观一些：把所有点从左到右排序。每次增加一棵树x时考虑把它作为右顶点的情形。只需要花O(n)的时间可以递推出所有与x形成的交叉点左上方的点数，因此时间复杂度为O(n^2)，空间仅为O(n)。

小测验

1. 如何求解最大空矩形问题？最大空正方形问题呢？
2. 如何求解包含指定点数的最小正方形问题？
3. 如何求解包含点数至少为且尽量接近指定数的矩形问题？

6.3 树的算法

本节介绍树上的经典问题和算法，包括LCA问题、计算与统计问题、构造与计数问题。

6.3.1 LCA问题

树的最近公共祖先(Lowest Common Ancestor)问题是树结构上最经典的问题之一。给一棵树T，每个询问形如：“点u和点v 的公共祖先是哪个点？”，此问题的答案被记为LCA(T, u, v)。LCA问题的算法分为在线和离线两种，前者要求在回答后一个问题之前必须给出前一个问题的输出，而离线问题允许在读入所有询问之后一次性给出所有问题的答案。

LCA问题的应用很多，例如它可以用来回答这样的询问“点u和点v的距离是多少？”由于在树中两点的简单路是唯一的，所以这个距离等于u到LCA(T, u, v)再到v的距离，关键仍然是LCA。

离线LCA的Tarjan算法 对于离线LCA问题，有一个十分精巧的算法如下所述：用LCA($root[T]$)调用下面的过程。最开始处理所有询问，如果存在询问LCA(u, v)，就

把v保存在集合$Q(u)$里。初始时所有结点颜色均为WHITE。

```
void LCA(u){
MAKE-SET(u);
ancestor[FIND-SET(u)] = u;
for(u的每个儿子v){
LCA(v);
UNION(u, v);
ancestor[FIND-SET(u)] = u;
}
color[u] = BLACK;
for(Q(u)中的所有元素v){
if (color[v] == BLACK)
printf("LCA(%d , %d) = %d\n", u, v, ancestor[FIND-SET(v)]);
}
}
```

其中MAKE-SET(u)、UNION(u, v)和FIND-SET(u)对应于第4章中介绍的并查集操作。可以看出，在任何时候一个集合里面的元素都形成了一棵树，每处理完一棵子树就把它并到父亲所在的集合中。该算法执行的是深度优先遍历，因此对于任何处理过（即黑色）的结点v，v当前所在的集合的代表元就是v和当前处理结点u的LCA。

算法的时间复杂度取决于MAKE-SET, UNION和FIND-SET的实现细节。把这个问题留在1.4中继续讨论。这是一个相当经典的离线算法，算法的巧妙之处在于利用了并查集，使得和当前处理元素u有关的所有询问都可以立即得出，因此平均下来每次询问时间几乎为常数。

在线LCA的算法 在线LCA有一个十分经典的算法。首先把边列表转化为左儿子-右兄弟表示法，并计算每个点u的父亲放在anc$[u][0]$中：

```
for(i = 1; i < n; i++){
    cin >> j >> k; anc[j][0] = k;
    next[j] = son[k]; son[k] = j;
}
```

然后用一次dfs计算出每个结点的第2^i级父亲，代码如下：

```
void dfs(int d, int p){
  int i, j;
  stack[d] = p; depth[p] = d;
  for(j=1;i=2; i<=d; j++, i<<=1) anc[p][j] = stack[d-i];
  for(j=son[p]; j; j=next[j]) dfs(d+1, j);
```

```
}
```

只需要调用dfs(0, root)，即可得到深度数组depth和结点u的第2^i级父亲。有了anc数组，可以写出返回结点u的第k级祖先的程序，它是O($\log n$)的：

```
int ancestor(int u, int k){
  while(k > 0){ u = anc[u][log[k]]; k -= (1<<log[k]); }
}
```

其中log[k]为k以2为底的对数取下整，它可以通过预处理得到。取$low = 0$，$high = \min\{\text{depth[u]}, \text{depth[v]}\}$。如果ancestor($u$, depth[u]-high)=ancestor(v, depth[v]-high)，说明u和v是祖先-后代关系，返回这个值；否则LCA的深度在区间$[low, high)$中。

算法一 二分查找真正的LCA的深度w，使得w是满足

$$\text{ancestor(u, depth[u]-w)} = \text{ancestor(v, depth[v]-w)}$$

的最大值。有两种二分法可以奏效。第一种是不断检查上式是否成立，如果不成立说明w太大，令$high = mid$，否则$low = mid$，始终保证low满足上式而$high$不满足（又类似于STL使用的左闭右开区间）。注意此时的结束条件是$low \geq high$。每次二分需要调用ancestor函数，因此每次查询的总时间复杂度为O($\log^2 n$)。

算法二 令u=ancestor(u, depth[u]-$high$)，v=ancestor(v, depth[v]-$high$)，即把u和v上升到同一个高度。接下来每次取祖先级数$lvl = \log high - low - 1$，计算$lu = \text{anc}[u][lvl]$, $lv = \text{anc}[v][lvl]$，并设$mid = high - (1 << lvl)$。如果$lu = lv$，说明mid处是公共祖先，设$low = mid$，否则设$high = mid$，并更新$u = lu$, $v = lv$，让它们往上走一步，在保持高度相同时不影响LCA的值。这样做显然是正确的，但时间复杂度并不明显。区间长度为n=2^k+1时有可能区间长度只减少1，表面上最坏情况需要减O(n)次才能确定LCA，但这样的分析是不精确的。

和并查集一样，不可能老是遇到这种情况。当区间长度n=2^k时$mid = 2^{k-1}$（注意取的是n-1的对数），不管哪种情况区间长度都缩短一半。这样，一旦区间长度变为2^k，接下来的二分次数一定是O($\log n$)的，而在此之前每次二分（其实已经不是二分了）都把区间长度缩短了1/2，因此总次数为O($\log n$)。程序如下：

```
int lca(int u, int v){
  int lu, lv, lvl, low, mid, high;
  low = 0;
  high = depth[u] < depth[v] ? depth[u] : depth[v];
```

```
  u = ancestor(u, depth[u]-high);
  v = ancestor(v, depth[v]-high);
  if(u == v) return u;
  while(low+1<high){
    lvl = log[high-low-1];
    lu = anc[u][lvl]; lv = anc[v][lvl];
    mid = high - (1<<lvl);
    if(lu == lv) low = mid; else
    { high = mid; u = lu; v = lv; }
  }
  return anc[u][log[high-low]];
}
```

这样，我们得到了一个不错的O($n \log n$)-O($\log n$)的算法。

LCA问题和±1-RMQ问题 LCA问题可以转化为±1-RMQ问题，从而用前面介绍的O(n)-O(1)算法或O($n \log n$)-O(1)算法解决。

给树T做深度优先遍历，并记录下每次到达的结点。第一个记录的结点是根root(T)，每经过一条边都记录它的端点。由于每条边恰好经过了两次，因此一共将记录2n-1个结点。我们用E[1，...，2n-1]来表示这个数组，并用$R[i]$来表示E数组中第一个值为i的元素下标，那么对于任何$R[u] < R[v]$的结点u,v来说，DFS中从第一次访问u到第一次访问v所经过的路径应该是$E[R[u]，\ldots，R[v]]$。虽然这些结点会包含u的后代，但是其中深度最小的结点一定是u和v的LCA。即：令数组$L[i]$表示结点$E[i]$的深度，那么当$R[u] \leq R[v]$时，LCA(T, u, v) = RMQ(L, $R[u]$, $R[v]$)；类似地，如果$R[u] > R[v]$，LCA(T, u, v) = RMQ(L, $R[v]$,$R[u]$)。

这样，在O(n)时间内把LCA问题转化为了±1-RMQ问题。

一般RMQ问题和LCA问题 前面遗留下来的问题是：如何高效解决一般RMQ问题？我们利用笛卡儿树把它转化为LCA，再转化回±1-RMQ问题。

笛卡儿树(Cartesian Tree) 是一种特殊的堆，它根据一个长度为n的数组A建立。它的根是A的最小元素位置i，而左子树和右子树分别为A[1...i-1]和A[i+1...n]的笛卡儿树。

给定一个数组，如何建立它的Cartesian Tree呢？最简单的方法就是直接按照定义，用顺序选择法来找最小值，最坏情况需要O(n^2)，下面给出一个O(n)的算法。

Cartesian树的建立算法 从A[1...1]开始建立，每次加入一个数，修改Cartesian树。不难发现，每次加入的数A[i]一定在旧树C[i-1]的最右路径上，而且一定没有右儿子。只要沿着最右路径自底向上把各个结点p和A[i]做比较，如果p小，则A[i]插入，作为p的右儿子，否则把p作为A[i]的左儿子。由于每个结点最多进入和退出最右路径各一次，

因此平摊时间复杂度为$O(n)$。

RMQ定理 数组A的Cartesian树记为C(A)，则RMQ(A, i, j) = LCA(C(A), i, j)。

定理非常直观，请读者自己证明。这样，我们把一般RMQ问题转化为了LCA，再转化为±1-RMQ问题，最终得到了一个O(n)-O(1)算法。

小测验

1. 叙述LCA问题、±1-RMQ问题和一般RMQ问题的转换关系。
2. 叙述离线LCA问题的Tarjan算法。为什么它不能解决在线LCA问题？
3. 什么是Cartesian树？如何高效的建立一个数组所对应的Cartesian树？

6.3.2 树的计算和统计问题

树有很多经典问题，本节选择其中最有代表性的一些加以讨论。

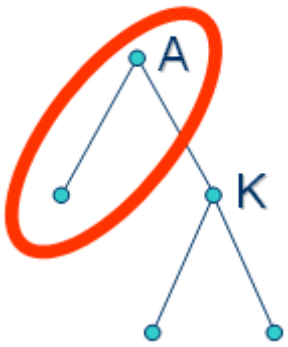
树的直径 在树中，任意两点的简单路是唯一的，它包含的边数称为这两个点的距离。树中最远的两个点的距离称为树的直径。如何快速计算出一棵树的直径？首先仍然是把无根树转化为有根树，即把边列表转化为左儿子-右兄弟表示。显然直径对应于两个结点的距离，这个距离在什么位置呢？可能是两棵子树的最深结点（情况一），也可能是某棵子树的直径（情况二），因此需要记录两个值：以i为根的子树直径d[i]和以i为根的子树深度g[i]。

不难得出：$g[i]$ = max{depth[i], g[j]}，其中j遍历i的儿子。depth[i]表示结点i在原树中的深度。用从下到上的顺序递推可以得到所有的g，时间复杂度为O(n)。

$d[i]$的递推式复杂一些，需要记录所有儿子中g的最大值g[u]和第二大值g[v]，则d[i] = max{g[u]+g[v], d[j]}，其中j遍历i的所有儿子。这一步也是O(n)的，整个算法时间复杂度为O(n)。

每个结点的最远点 如何快速的计算出树中每个点的最远点？如果它可以在O(n)时间内完成（平均每个点O(1)），那么树的直径也可以在O(n)时间内求出，因此这个问题更加基本。设d[i]为i的最远点到i 的距离，那么如何去出d[i]？如果写成d[i] = max{g[j]} - depth[i]，就错了，它只考虑了i到后代的距离，而忽略了图8.1.3中划圈的部分，它们也可能离i最远。

现在我们需要“子树”的概念扩展一下。把无根树变为有根树后，K在有根树中子树为“下方子树”，而其他部分在原来的无根树中也是K的子树，在有根树中称为K的“上方子树”。这样，可以分别写书两个子树的递推式。细心的读者可能已经发现了：在这



样的情况下有根树相对于无根树的优势已经不存在了，我们还不如不进行转化，直接在无根树上进行递推：设T(u, v)为原树删除边(u, v)后以v为顶点的有根树（u为0表示不删除任何边），d$[u, v]$为T(u, v)的深度，则有递推式d$[u, v] = \max\{0, \text{d}[v, w] + 1\}$，其中$w$是$v$不等于$u$的儿子。状态数为$2n - 2$，且每个状态只被使用一次，因此总时间复杂度为O$(n)$。

有了d$[u, v]$，则每个结点i的最远点离它的距离就是d$[0, i]$，时间复杂度为O(n)。如果读者对T(u, v)的定义感到奇怪，你可以设想一下直接设d$[v]$表示为以为v为根的有根树，然后进行递推。如果有一条边$(1, 2)$，那么计算d[1]需要d[2]，而计算d[2]又需要d[1]，产生了循环引用！引入T(u, v)的目的正是为了消除循环引用，因为总是用边少的树递推边多的树，具有明确的拓扑顺序。

应用刚才的方法很容易解决下面的问题：

树的质心 求出一个结点K，使它所有子树的结点数的最大值最小。借助于刚才的思想，我们以T(u, v)为状态进行递推，很容易得到每棵子树的结点数和它们的最大值。这样，我们仍然在O(n)的时间内完成了求解。

两点间的权最小边如果树是加权的，如何求出任意两点的唯一简单路上权最小的边？这个问题非常基本，常常被用在网络优化算法中，我们将在后面继续讨论。这个问题仍然可以通过动态规划解决。设需要找u和v的路上权最小的边，则以u为根转化为有根树，令d$[i]$为i到u的路上的权最小边，自底向上递推即可。

最后，让我们来看一个比较难的问题。

距离统计 给一个正权树和距离限制L，统计距离不超过L的结点对。显然在固定根时可以用O(n)时间内求出其他所有点到根的距离，因此在O(n^2)时间内计算出每对结点的距离。但这样的时间复杂度并不令人满意。

我们仍然把无根树转化为以A为根的有根树，并且把所有结点分成两部分，如

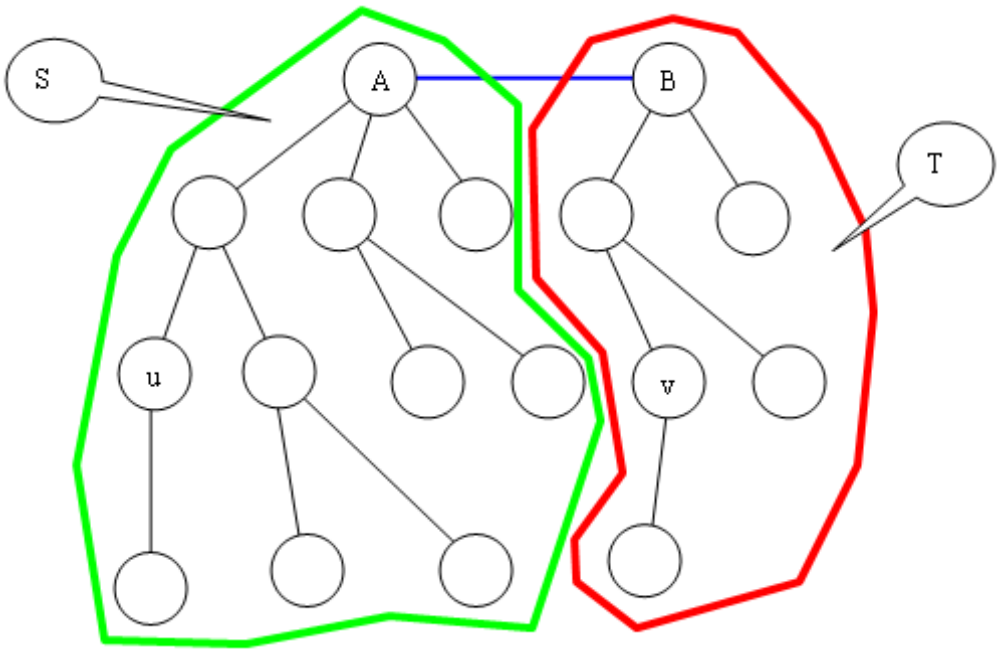
图8.1.3: 所有点对分为两种：

1. 全部在S中或者全部在T中。

2. 一个点在S中另一个点在T中。

显然第一类只需要简单的递归调用，而第二类需要先从A开始进行一次DFS，求出所有点到A的距离。设u在S中，v在T中，则d(u, v)=d(A, u)+d(A, v)。为了使d(u, v)$\leq L$，必须有d(A, u)$\leq L$-d(A, v)。因此，必须对每一个v统计出满足d(A, u)$\leq L$-d(A, v)的u的个数。如果事先给S和T里的d值排序，再从d值从小到大的顺序枚举v，则满足上式且距离最大的u编号将递减，因此在O(n)的时间内统计出第二类结点对。

以上的解法是基于分治的，递归调用统计第一类，合并过程统计第二类，合并时间为O(n)（从A开始DFS统计第2类点）+O($n\log n$)（给S和T中的d值分别排序）+O(n)（枚举v并统计第3类点）=O($n\log n$)。注意，这里划分成的两棵树用前面的定义来说就是T(B, A)和T(A, B)。



接下来的问题是：如何删除一条边，使得S和T尽量均匀？看起来这并不是一个平凡的问题，所以只能花O(n)的时间计算所有子树的结点个数，再枚举比较每条边。可以证明至少存在一条边使得划分成的两部分在$3n/4$和$n/4$之间（留给读者思考），因此：

划分 $O(n)$
递归 $T(3n/4)+T(n/4)$（最坏）
合并 $O(n \log n)$

用主定理容易得到：总时间复杂度为$O(n \log^2 n)$，比$O(n^2)$优秀。

小测验

1. 如何在$O(n)$时间内求出树的直径？
2. 如何在$O(n)$时间内求出树中每个结点的最远点？和质心？这些算法有什么启发性？
3. 如何用数学归纳法证明关于树的命题？

6.3.3 树的构造和计数问题

树的度序列 由于树有n-1条边，所以如果每个结点的度为$d_1, d_2, ..., d_n$，必然满足$d_1+d_2+...+d_n=2(n-1)$。反过来，对于任意满足$d_1+d_2+...+d_n=2(n-1)$且$d_1 \geq d_2 \geq ... \geq d_n$的正整数序列S，都可以构造出一棵以它为度序列的树。

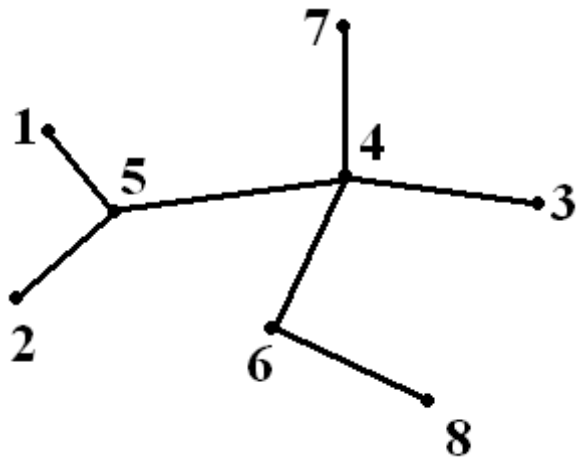
证明是构造性的。用数学归纳法，假设$n = k$时成立，只需证明$n = k + 1$时命题也成立。首先必然有$d_{k+1}=1$，否则序列和至少为$2(k+1)>2k$。我们从序列S中删除d_1和d_{k+1}并增加$d^*=d_1-1$（把它插入到合适的位置）。显然新序列S'的和为$2k-2=2(k-1)$。由归纳假设，可以构造树T'，以S'为度序列。现在增加一个新结点v，把它与d^*对应的结点相连，则d^*的度变为d_1，并新增加一个度为1的结点v，因此新树的结点序列恰为S。

Prüfer编码。树的结构比较复杂，有时候讨论起来不方便。下面介绍的**Prüfer编码** 完全刻画了树T，二者之间是一一对应关系。下面的步骤求出了编码$(a_1, a_2, ..., a_{n-2})$：

1. 令b_1为T编号最小的叶子，令a_1为b_1惟一的邻接点。
2. 从T中删除b_1，产生新树T_1。
3. 令b_2为T'编号最小的叶子，令a_2为b_2惟一的邻接点。
4. 从T_1中删除b_2，产生新树T_2。

5. ...

简单的说，只要每次删除编号最小的叶子，把它的相邻点加到编码中即可，直到只剩下两个结点，我们得到了一个长度为n-2的Prüfer编码。例如图8.1.3中的树编码为(5, 5, 4, 4, 4, 6)。



从Prüfer编码可以很方便的重建这棵树。首先令a_{n-1}=n，即把编码延长到n-1个数。如果知道了b_1, b_2, ..., b_{n-1}，那么(a_i, b_i)就是树中的所有边。所以关键在于求出b_i。根据定义，我们不难设计出重建树的算法：令b_1为不在{a_1, a_2, ..., a_{n-1}}中的最小结点，b_2为不在{a_2, ..., a_{n-1}}中且不为b_1的最小结点，b_3为不在{a_3, ..., a_{n-1}}中且不为b_1也不为b_2的最小结点...直到计算出整个(b_1, b_2, ..., b_{n-1})。对于每个b_i，它是不在{a_i, a_{i+1}, ..., a_{n-1}}且不是b_1 ~b_{i-1}的最小结点。如果我们维护一个“不在当前A序列中的元素集合”，则每次需要取集合最小值b_i并从集合中删除（如果找得到的话），然后可能会增加一个元素a_i（如果它不在后面序列中再次出现的话）。可以用堆在O($\log n$)的时间内完成删除最小值和插入操作，总时间复杂度为O($n \log n$)。

由于Prüfer编码的每一位可以独立取1~n的任意数，所以我们得到：

Cayley定理 不同的n结点标号树的数量是n^{n-2}。

二叉树的平衡三染色 给出一棵二叉树，它的结点分成三个部分A,B,C，使得每一个结点和他的儿子以及他的兄弟（如果有的话）处于不同的集合，使得每两个集合的元素个数最多相差1。这个结论并不是很常用，但是其思想是值得学习的，所以这里叙述如下。我们先对命题进行加强，加强部分为：“根结点位于集合A，且满足Max{|A|,|B|,|C|}=|A|”。我们用数学归纳法证明一定存在这样的划分。

当树为空的时候显然成立：$|A| = |B| = |C|=0$；

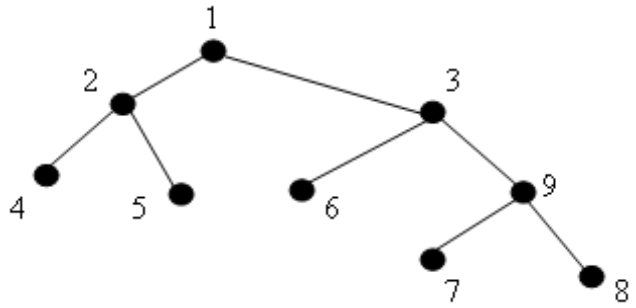
当树只有一个根结点X，无左右儿子时，令A={X},B=C={}便能满足要求；

当树的结点数大于1时，设根结点为X，其左右儿子分别为Y, Z，且X的左右子树均加强满足命题。设左子树的三个集合为A_1,B_1,C_1，Y位于集合A_1中；右子树的三个集合为A_2,B_2,C_2，Z位于A_2中。则有$|A_1| \geq |B_1| \geq |C_1|,|A_2| \geq |B_2| \geq |C_2|$。不妨设$|A_1|=m,|A_2|=n$。则可能出现如下这几种情况：

$(\|A_2\|,\|B_2\|,\|C_2\|)$ $(\|A_1\|,\|B_1\|,\|C_1\|)$	(n,n,n)	(n,n,n-1)	(n,n-1,n)	(n,n-1,n-1)
(m,m,m)	$A=B_1+C_2+X$ $B=A_1+B_2$ $C=A_2+C_1$	$A=C_1+C_2+X$ $B=A_1+B_2$ $C=A_2+B_1$	$A=B_1+B_2+X$ $B=A_1+C_2$ $C=A_2+C_1$	$A=B_1+B_2+X$ $B=A_1+C_2$ $C=A_2+C_1$
(m,m,m-1)	$A=C_1+C_2+X$ $B=A_1+B_2$ $C=A_2+B_1$	$A=B_1+C_2+X$ $B=A_1+B_2$ $C=A_2+C_1$	$A=C_1+C_2+X$ $B=A_1+B_2$ $C=A_2+B_1$	$A=B_1+B_2+X$ $B=A_1+C_2$ $C=A_2+C_1$
(m,m-1,m)	$A=B_1+B_2+X$ $B=A_1+C_2$ $C=A_2+C_1$	$A=C_1+C_2+X$ $B=A_1+B_2$ $C=A_2+B_1$	$A=B_1+C_2+X$ $B=A_1+B_2$ $C=A_2+C_1$	$A=C_1+C_2+X$ $B=A_1+B_2$ $C=A_2+B_1$
(m,m-1,m-1)	$A=B_1+B_2+X$ $B=A_1+C_2$ $C=A_2+C_1$	$A=B_1+B_2+X$ $B=A_1+C_2$ $C=A_2+C_1$	$A=C_1+C_2+X$ $B=A_1+B_2$ $C=A_2+B_1$	$A=B_1+C_2+X$ $B=A_1+B_2$ $C=A_2+C_1$

无论出现哪种情况，都存在方案使得新的集合A,B,C符合命题。

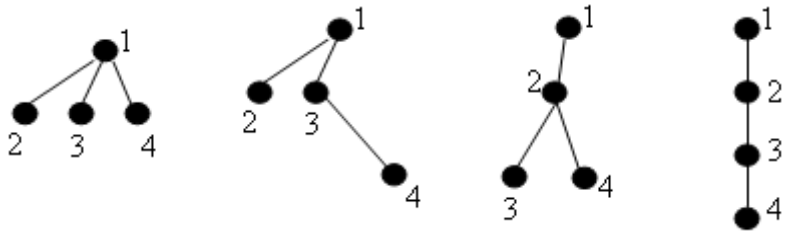
BFS和DFS序列重建问题 给定一棵树，则可以得到它的DFS和BFS序列。比如图8.1.3：



BFS序列为：1,2,3,4,5,6,9,7,8；DFS序列为：1,2,4,5,3,6,9,7,8。

两种遍历法都是从根结点开始扩展，BFS是深度小的优先，而DFS是深度大的优先。两种方法在处理同一个结点的若干儿子时均选取编号小的儿子优先扩展。这样，

对于一棵树，BFS和DFS序列都是唯一的。反过来却不成立，例如若一棵树的BFS 序列是(1,2,3,4)，DFS序列是(1,2,3,4)，那么这棵树可以是四种不同的树，如图8.2.1中所示：



给定BFS和DFS序列，求任意一棵原树。（或者判断无解）

我们按照DFS序列的顺序来构造。很显然DFS[1]=BFS[1]=根，DFS[2]=BFS[2]=根的最小编号儿子。以DFS[1]作为根，DFS[2]作为根的儿子，就构成了一棵规模为2的树。现在假设DFS的前k-1个节点已经建好了树T_{k-1}。下面考虑把第k个节点（DFS[k]）插入到树T_{k-1}中构成T_k（$k \geq 3$）。

因为DFS[1], DFS[2],...,DFS[k]这些点中，DFS[k]是在深度遍历中最后访问到的点，所以DFS[k]在T_k中的位置必然是“极右路径”上最末段的点。也就是说T_k是通过在T_{k-1}的极右路径上插入DFS[k]而得到的。

可以设T_{k-1}的极右路径是$a_1, a_2, \ldots, a_t (t \geq 2)$。设DFS[$k$]在BFS序列中的前驱是$v$。

显然a_1=根=BFS[1]。因为$k \geq 3$，所以DFS[k]≠DFS[2]同时DFS[2]=BFS[2]，于是就有DFS[k]≠BFS[2]。也就是说DFS[k]在BFS序列中的位置不可能是2，所以v≠BFS[1]，即v≠a_1。

下面分四种情况讨论。

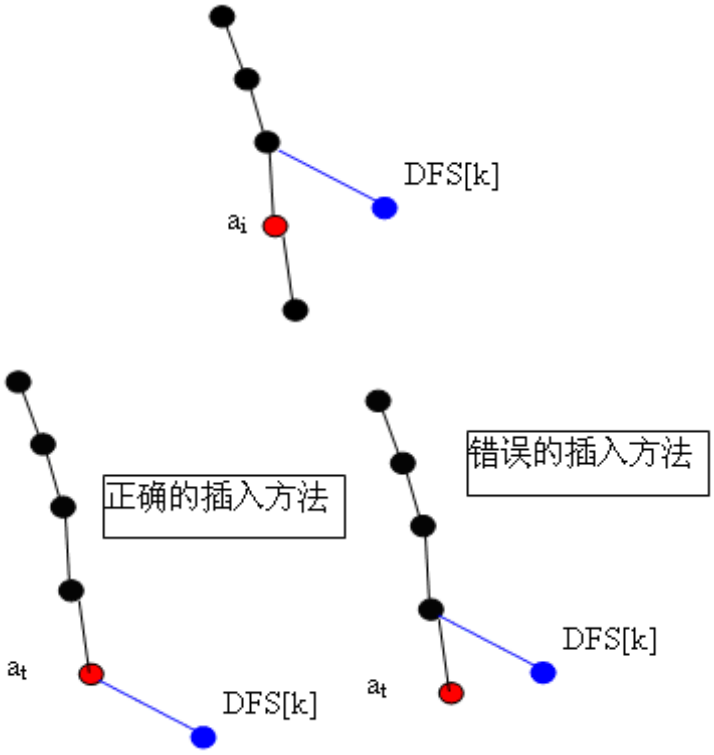
情况一 存在i ($2 \leq i < t$)，满足a_i=v。如图8.2.1，DFS[k]别无选择，只能给a_{i-1}做儿子。

(a)情况一(b)情况二

请注意：这种情况下DFS[k]必须大于a_i。因为如果DFS[k]<a_i，根据“编号小的儿子先遍历”规则，DFS[k]在a_i之前就应该已经被遍历。矛盾。

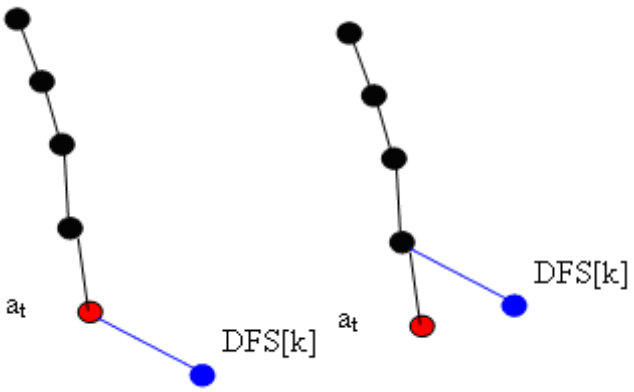
因此，如果在这种情况下发现DFS[k]<a_i，就可以判定无解了。

情况二 a_t=v，且a_t>DFS[k]。



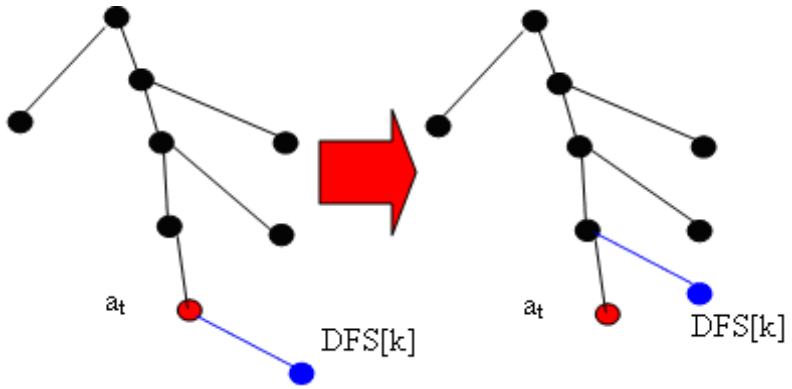
从理论上说，可以有两种插入DFS[k]的方式。但是第二种方案必须满足DFS[k]>a_t。这显然和我们的条件矛盾。因此在第二种情况下，插入方式也是唯一的。

情况三 a_t=v，且a_t<DFS[k]。此时有点棘手。



(a)情况三(b)解决方法

因为a_t<DFS[k]，所以两种情况都是有可能发生的。到底应该取哪种呢？还是都去试探呢？都去试探一次自然是对的，但是复杂度将会很高，相当于是在搜索。我们可



以从两种插入方案对以后的影响来分析问题：

第一个方案中，如果要让DFS[k]最终成为a_t的“BFS后继”，就要求a_t的“右边”、DFS[k]的“左边”不能有任何东西。这实际上就是对以后的插入过程做了一个深度限制。

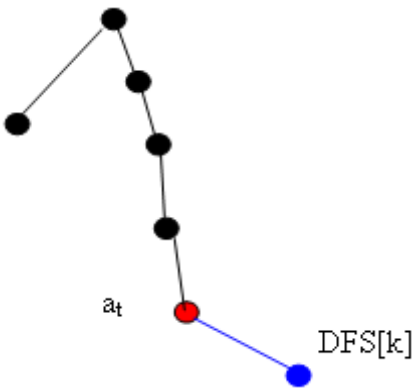
反观第二个方案，不管之前、之后的点如何插入树中，DFS[k]都铁定是a_t的“BFS后继”了。所以第二个方案对之后的插入没有任何影响。

因此我们从直观上来看，应该选择第二个方案插入。

严谨的看，如果第一种方案最终构造成功了可行解，因为DFS[k]是a_t的“BFS后继”，所以在a_t的这一深度上，a_t“右边”必然不存在任何点（否则如果存在，a_t的后继就会是它、而不是DFS[k]）。这样的话，可以把DFS[k]往上“提”，如图8.2.1中图(b)。经过调整之后，无论BFS还是DFS序列，都不会改变。也就是说如果方案一可以构造出可行解，方案二也必然可以构造出可行解。

所以我们可以大胆的选择方案二插入。

情况四 不存在任何的$i(1 \le i \le t)$，满足a_i=v。这时只能把DFS[k]插在a_t之后，如图8.2.1所示。



综上，对所有的情况，我们都能找到唯一的方式插入，最终得到一棵树T_n。

从构造的过程看，T_n肯定完全满足DFS序列。反过来，只要问题有解，T_n一定也可以符合BFS序列（因为构造过程中每一步都是必要的）。

最后只要检验T_n的BFS序列和输入的BFS序列是否相同，即可判断问题是否有解。

进一步的很容易发现，T_n不仅是一组任意解，它也是深度最小的解。这一点很重要。

BFS和DFS序列计数问题 把上一个问题进行扩展：给定BFS和DFS序列，问有多少棵树满足要求。从构造的角度讲，能造成多解的只有上面提到的“情况三”。如果采用左边的方案，必须满足两个条件：

已经插入的点中，不存在比a_t深度大的点。（如果存在，DFS[k]的“BFS前驱”就变成了它，而不是a_t）。

以后插入的点，深度都必须小于a_t的深度。（如果存在深度和a_t相同的点，a_t的后继就会是这个点，而不是DFS[k]）

刚才提到，根据本文已经提出的方法构造，生成的是深度最小的树。而第二点正好要求“以后的点深度都必须小于a_k的深度”。

所以，如果在插入DFS[k]的时候采取了第一种插入方案，那么要检验这种插入方案是不是可行，只要把以后的点都按之前提出的正常方法插入即可。如果最后得到的树的BFS序列仍然和输入的BFS序列全等，那么就证明在第k步的时候选择第一种方案是可行的。

我们首先把深度最小的树按照上一节提出的方法构造出来。

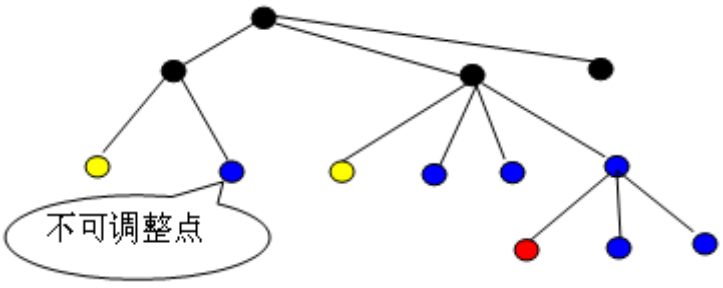
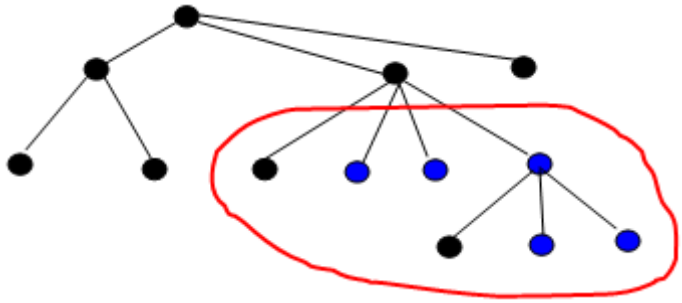


图8.2.1中黑色点是插入过程中的第一种情况，红色点是第二种情况，蓝色点是第三种情况，黄色是第四种情况。

从理论上说，蓝色点都有两种插入方式，但是可以明显的看到，最左边的那个蓝色点是“无可选择”的。

如果把它往下调，就违反了上面提出的第二条“以后插入的点，深度都必须小于a_t的深度”。因为后面的点插入都是按照第一节提出的基本构造法进行的，所以得到的是“深度最小的树”；如果连深度最小的树都不能满足“深度小于a_t的深度”，其他的插入方式就更加不用提了。

就上面那棵树而言，可“调整”的点如图8.2.1中蓝点所示：



有5个点，每个点都有2种插入方式，也就是2^5=32棵不同的树。观察上面用粗红线条框出来的部分，除了每一层的第一个点，其他的点都是“可调整点”。

这个用红线框出来的部分有一个奇妙的性质：它的BFS和DFS序列完全相等。

令L=max{x | BFS$[n-x+1]$, BFS$[n-x+2]$, ..., BFS$[n]$这些点的DFS和BFS序列相同}

在数列BFS$[n-L+1]$, BFS$[n-L+2]$, ..., BFS$[n]$中，设有t个数小于它的前一项（这t个数实际上就是红框中每排的第一个点，即属于构造法中第二种情况的点），那么答案就是2^{L-t}。

小测验

1. 什么是树的Prüfer编码？如何用它证明Cayley定理？
2. 以二叉树的平衡三着色问题为例，说明如何用数学归纳法证明和树有关的命题。
3. 给定树的BFS序列和DFS序列，如何重建树和计树？

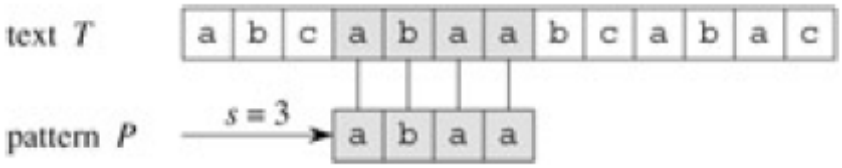
6.4 字符串及其算法

本节介绍字符串及其算法。字符串在文本处理、生物信息学有着重要的应用。

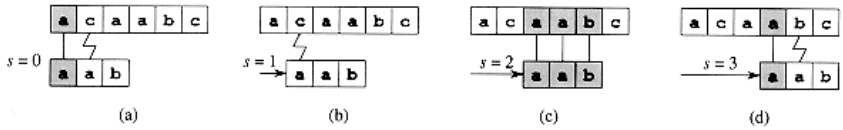
6.4.1 字符串匹配问题

字符串S是一个由n个字符组成的序列，通常用S[1..n]来表示，其中S[i]表示S的第i个字符。每一个字符都是字母表中的一个元素，字母表的大小通常用Σ来表示。例如在Σ={0, 1}上的串称为01串，而Σ={a, b, ..., z}上的串称为字母串。

字符串匹配问题 假设文本是一个长度为n的字符串T，模板是一个长度为m 的字符串P，且$m \leq n$。一般说来，我们认为文本和模板的字母表相同。如果存在整数s 使得$0 \leq s \leq n-m$且T[s+1..$s+m$]=P[1..m]，我们说模板P在文本T的**偏移(shift)** s出现，也可以说在位置s+1出现（因为T[s+1]=P[1]）。字符串问题就是要找出所有合法的偏移。图8.2.2是P在T的偏移3，也就是位置4出现。有时候也把合法偏移称为匹配点。



朴素的匹配算法 即判断每个位置s是不是一个匹配点。检查匹配点需要O(m)的时间，而可能的匹配点有O($n-m$)个，因此最坏情况时间复杂度为O($m(n-m)$)。有一个简单的优化：检查匹配点的合法性时一旦发现其中一个字符不相同，理解停止比较，换下一个匹配点。如图8.2.2所示：



关于朴素匹配算法的一个有趣结论是：如果文本和模板都是随机生成时，朴素匹配算法是期望O(n)的。优化后的朴素匹配算法如图8.2.2：

Rabin-Karp算法 假设字母表为{0,1,2,3,4,5,6,7,8,9}（读者很容易把下面的结论推广到其他形式的字母表），那么一个长度为m的串（例如模板串）就可以看作是一个m位的十进制数。用$P[i]$表示模板串的第i个字符，$T[i]$表示主串的第i个字符，p表示模板串，t_s表示串中从第s+1个字符开始，$s+m$个字符结束的长度为m的串，则模式匹配问题转换为：是否存在一个$i(0 \leq s \leq n-m)$，使得$p = t_i$。在不考虑p代表的数可能很大的情况下，可以把t_0，t_1，...，t_{n-m}都计算出来，然后用p和它们一一比较，只需要进行$n-m$+1次。

```
AlmostBruteForce(T[1..n], P[1..m]):
  for s ← 1 to n − m + 1
      equal ← true
      i ← 1
      while equal and i ≤ m
          if T[s + i − 1] ≠ P[i]
              equal ← false
          else
              i ← i + 1
      if equal
          return s
  return 'none'
```

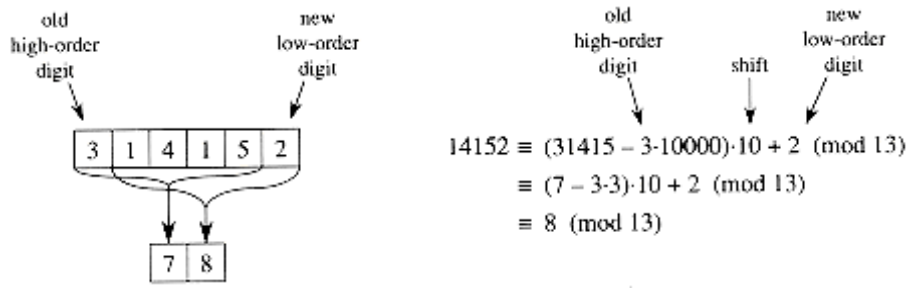
如何计算p和t_i？对于p，可以用Horner规则：

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \ldots + 10(P[2] + 10P[1]))). \tag{6.1}$$

t_0可以用同样的方法计算，时间复杂度为$\Theta(m)$。接下来，可以用递推式计算剩下的：

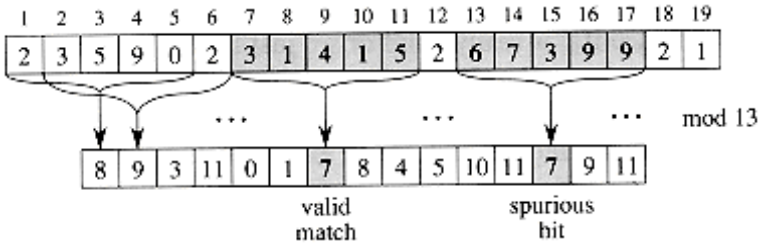
$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \tag{6.2}$$

只要注意到t_s和t_{s+1}的表达式有很多公共部分，读者不难自己证明此式，如图8.2.2：



其中$h = 10^{m-1}$需要在预处理中计算出来（可以直接计算O(m)，也可以用二进制展开m，时间复杂度为O($\log m$)），这样，计算所有t_i的总时间复杂度为O($n+m$)。

这里有个潜在的问题：p和t_i可能很大，不仅储存不方便，而且比较时间也不能看作常数。怎么办呢？我们可以用结果对一个很大的质数q取模。则预处理时$h = 10^{m-1} \bmod q$，而p和t_i的表达式都应对q取模。比较可以在常数时间内完成了，但是如果发现$p \equiv t_i (\bmod q)$，并不能保证$p = t_i$。$p \equiv t_i (\bmod q)$但$p \neq t_i$的情况称为伪命中（spurious hit），如图8.2.2：



为了防止伪命中，我们只能把模相等作为一个预判条件。条件满足时还应该继续判断。

如何进一步判断呢？有如下几种方法：

方法一 直接比较p和t_i是否相等。这样每次验证需要O(p)的时间。

方法二 对多个q取模。如果所有的q都是质数而且乘积大于10^m-1，则由中国剩余定理，如果所有余数相等，那么p和t_i一定相等。

方法三 预先把$p \equiv t_i (\bmod q)$的所有解算出来，设计出一个能更快区别出它们的算法。

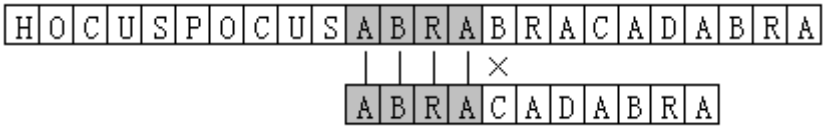
显然，方法三最困难，而且当解很多时时间很慢，且很可能最好的“区别”算法仍然需要比较完整个p和t_i。方法二看起来不错，但是由于取模和比较操作是在任何情况下都需要完成的，所以它延长了平均处理时间（读者不妨仔细算一算实际的工作量）。

推荐方法一，它不仅简单，而且遇到很多次验证的情况毕竟很少。事实上，如果验证次数很少（近似为常数），而q又是比m大的素数，那么RK算法是期望线性的，具体来说，是O(n)+O($m(v+n/q)$)，其中v为匹配点个数，q为模。

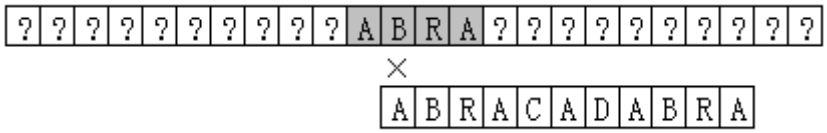
这样的思路称为**指纹（fingerprint）** 方法，它具有相当的普遍性，因此RK算法很容易进行推广。如果所有模板$p_1,\ldots,p_k$的长度相同，均为m。假设所有的$p_i \bmod q$各不相同（一般可以通过试验不同的q来达到这个要求），则仍然只需要O(m)的

时间就计算出所有$t_i \bmod q$。对于每个t_i，可以用二分查找的办法计算出它可能和哪个模板匹配，如果$t_i \equiv p_j (\bmod q)$，则验证t_i和p_j是否相同。平均的时间复杂度为$O(m+n\log k)$。RK算法还可以处理各个模板长度不同以及二维匹配（模板是$m\times m$的矩阵，主串是$n \times n$的矩阵，模板的左上角可能在主串的任何位置出现，但不能被翻转或者旋转），留给读者思考。

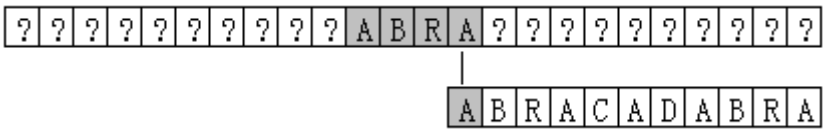
KMP算法 这个算法的出发点是朴素模式匹配算法中的无用比较。如图8.2.3，



在成功匹配四个字符后失败。朴素匹配算法的做法是模板右移一位，但事实上，右移一位肯定仍然匹配不成功。如图8.2.3，不管文本串如何，匹配都将以失败告终。

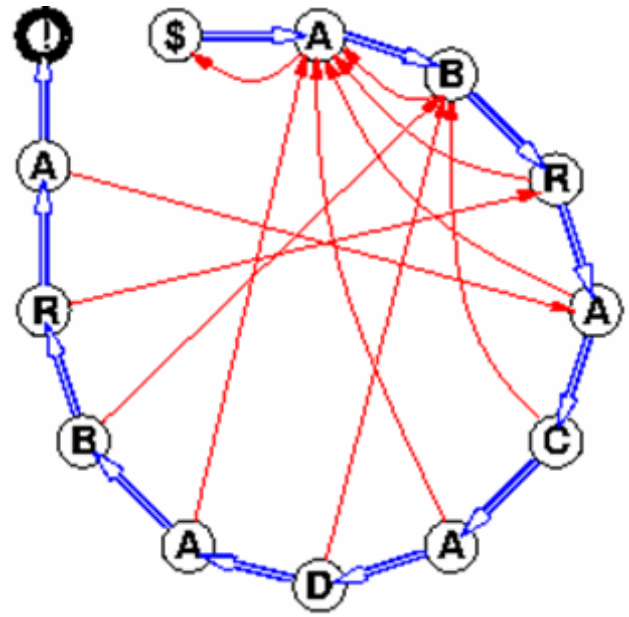


换句话说，如果上一次恰好匹配了4个字符后失败，那么模板右移一位后仍然无法匹配。进一步的，我们发现移动两位仍然不可能匹配，而三位是有可能匹配的：



既然是有可能匹配，我们不能贸然继续右移模板串。我们把这个结论总结一下就是：“如果上一次恰好匹配了4个字符后失败，那么模板可以直接右移3位。”容易看出，当匹配失败时，主串的“当前比较字符”不变，而模板串的“当前比较字符”左移。引入“当前比较字符”指针可以把算法叙述得更方便：“如果匹配失败时模板的指针是5，则把它修改成2，继续比较”。这可以看作一个有向图中位置5到位置2的有向边。

图8.2.3就是这样一个有向图。图中的结点表示模板的指针，匹配成功时沿着粗边走，而失败时沿着细边走。显然，细边一定是往回连的。可以验证刚才的结论：第五个字母C到第二个字母B有一条细有向边。再如匹配到最后一个字母A失败时应跳回到第4个字母A。用fail[i]表示第i个字符比较失败时的新位置，则KMP算法的框架如图8.2.3：




```
KnuthMorrisPratt(T[1..n], P[1..m]):
  j ← 1
  for i ← 1 to n
      while j > 0 and T[i] ≠ P[j]
          j ← fail[j]
      if j = m        ⟪Found it!⟫
          return i − m + 1
      j ← j + 1
  return 'none'
```

外层循环不断增加主串指针，而在匹配失败时不断修改模板的指针j，直到匹配可以继续。由于i最多增加n-1次，因此j也最多增加n-1次（注意j增加的时机）。由于j减少的总量不超过增加的总量，因此j的减少次数也不超过n-1次，主算法时间复杂度为O(n)。问题的关键是fail函数的计算。

fail[i]是比较失败时模板指针的新位置，它应该满足两个条件：

1. 不要指向一个肯定不会产生匹配的地方
2. 不要漏掉可能会产生匹配的地方

换句话说，fail应该指向最后一个可能产生匹配的地方（即回溯得尽量少，或者说fail值尽量大）。用数学语言描述就是：P[1..fail[j]-1]是“P[1..j-1]的真前缀中，是T[1..i-1]的后缀中最长的一个”。其中“是T[1..i-1]的后缀”表明i是第一个匹配失败的字符。

图8.2.3是ABRACADABRA的fail函数，读者可以验证刚才的定义。

P[i]	A	B	R	A	C	A	D	A	B
fail[i]	0	1	1	1	2	1	2	1	2

有了定义之后，可以用朴素方法加以验证：对于给定的j，从大到小枚举fail[j]，直到P[1..fail[j]-1]是P[1..j-1]的后缀。检查后缀需要O(m)，枚举fail[j]需要O(m)，枚举j需要O(m)，总预处理时间为O(m^3)

KMP的过人之处（也是它让人迷惑的地方）在于：它用模板自己匹配自己，在O(m)的时间求出了所有的fail函数。算法很像KMP的主过程，依次求出fail[1], fail[2], ..., fail[m]，每次会用到以前算出的fail值。在下面的算法中，i是T指针，j是P指针，这里原来的模板在“自己匹配自己”的过程中又做主串又做模板。

```
ComputeFailure(P[1..m]):
  j ← 0
  for i ← 1 to m
      fail[i] ← j        (*)
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

显然fail[1]是正确的。下面用数学归纳法证明所有的fail[j]都是正确的。假设fail[i]是正确的，那么考虑fail[i+1]的计算。刚执行(*)计算出fail[i]时j=fail[i]，因此P[1..j-1]是P[1..i-1]的最长真前缀也是后缀，定义符合fail如图8.2.4（fail^{0}无定义）：

用数学归纳法容易证明：所有形如P[1..fail^c[j]-1]的串同时是P[1..i-1]的前缀和后缀，且最长的P[1..i-1]的前缀/后缀也形如P[1..fail^c[j]-1]，其中c是满足P[fail^c[j]]=P[i] 的

$$fail^c[j] = fail[fail^{c-1}[j]] = \widehat{fail[fa}$$

最小c（这就是里层循环体做的事情）。这个过程还可以看成动态规划，不断用当前c的fail值进行更新：

$$fail[i] = \begin{cases} 0 \\ \max\limits_{c\geq 1}\left\{ fail^c[i-1]+1 \;\middle|\; P[i-1] = P[f \right. \end{cases}$$

如果P[j]和P[fail[j]]是同一个字符，那么回溯后马上又匹配失败，跳到P[fail[fail[j]]]。此时我们还不如直接令fail[j]=fail[fail[j]]，少一次跳转。这个过程可以通过修改预处理过程或者算出fail之后加一个优化过程来实现，如图8.2.4：

```
ComputeOptFailure(P[1..m]):
    j ← 0
    for i ← 1 to m
        if P[i] = P[j]
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j]
            j ← fail[j]
        j ← j + 1
```

```
OptimizeFailure(P[1..m], fail[1..m]):
    for i ← 2 to m
        if P[i] = P[fail[i]]
            fail[i] ← fail[fail[i]]
```

这个方法虽然不降低时间复杂度，但往往让比较次数少了很多，例如图8.2.4是ABBABBABABB优化前后的fail值比较。fail值越小越好，因为这意味着模板往右移动的幅度更大。

P[i]	A	B	B	A	B	B	A	B	A	B	B
unoptimized fail[i]	0	1	1	1	2	3	4	5	6	2	3
optimized fail[i]	0	1	1	0	1	1	0	1	6	1	1

小测验

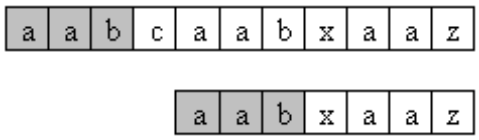
1. 什么串？什么是串匹配问题？
2. 叙述Rabin-Karp算法。它的主要思想是什么？期望时间复杂度是多少？最坏情况呢？
3. 叙述KMP算法。为什么匹配失败后模板移动的幅度只和模板有关？叙述失配函数的含义、朴素计算法和动态规划计算法。如何优化fail函数？

6.4.2 Z算法和BM算法

本节首先介绍Gusfield提出的一种字符串处理方法，它本身可以得到一个简单的线性时间匹配算法，然后讨论Boyer-Moore算法。

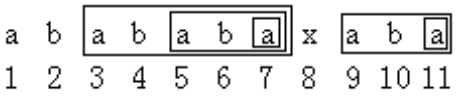
对于字符串S[1...n]和$i \in \{2, \ldots, n\}$，定义Z_i为S和S[i...n]的最长公共前缀的长度。例如对于串S[1...11] = aabcaabxaaz，则Z_2=1, Z_3=Z_4=0, Z_5=3（注意S[5...11]=aabxaaz）...，Z_9=2, Z_{10}=1, Z_{11}=0。用枚举法可以在O($n-i$)时间内求出Z_i，因此总时间为O(n^2)。下面我们给出一个方法能在O(n)时间内算出所有Z_i。

对于Z_i >0，定义i处的Z-box为$S[i...i+Z_i-1]$，即从i开始的极大前缀。如图8.1，在刚才的S=aabcaabxaaz中，i=5处的Z-box为aab，因为$c \neq x$。



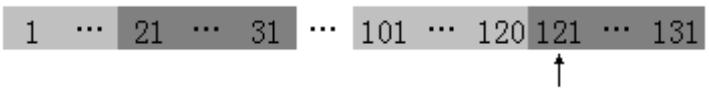
对于$i \geq 2$，r_i为所有位于$j \leq i$处的Z-box的最右端点（如果$j \leq i$处的Z-box全都不存在，r_i=0）。若r_i >0，定义l_i为,形如S[j...r_i]（$j \leq i$）的Z-box中的左端点，否则l_i=0。注意这里只需要让Z-box的右端点尽量往右，在这个情况下，如果仍有多个Z-box则随便取哪个都可以。

如图8.1，一共有五个非空的Z-box，由此可得各个位置的Z/r_i/l_i值如表8.1.1：



i	2	3	4	5	8
Z_i	0	5	0	3	0
r_i	0	7	7	7	7
l_i	0	3	3	5(或3)	7(或3，5)

可以从左到右扫描，利用已经计算过的Z_i值。变量l和r代表最近计算的l_i和r_i，即r是到现在为止曾经见过的所有Z-box的最右端点。计算Z_2需要比较S[1...n]和S[2...n]，直到第一次失配。



如图8.1，当前字符k=121，而(l_{120},r_{120})=(101,131)，因此我们现在位于$S[101...131] = S[1...31]$这个Z-box中（根据定义，一个Z-box总是和S的一个前缀相等）。因此$S[121...131] = S[21...31]$。因此如果$Z_{21}$=9则$Z_{121}$=9。把这个想法加以一般化（并考虑刚才没有出现的其他情况），我们得到：

Z算法初始化$l = r = 0$，然后对k=2, 3, ..., n依次计算Z_k。

for $k := 2, \ldots, n$ either case 1 or case 2 applies:

1. if $k > r$ then
   $Z_k := \max\{j \le n - k + 1 \mid S[1 \ldots j] = S[k \ldots k + j - 1]\}$;
   If $Z_k > 0$, set $l := k$ and $r := k + Z_k - 1$;
2. if $k \le r$, we're inside Z-box $S[l \ldots r] = S[1 \ldots Z_l]$, and thus $S[k \ldots r] = S[k' \ldots Z_l]$ for $k' = k - l + 1$.
   (Draw a picture!)
   Let $t = |S[k \ldots r]|$;
   (a) If $Z_{k'} < t$, we know to set $Z_k := Z_{k'}$.
   (b) Otherwise $S[k \ldots r] = S[k' \ldots Z_l] = S[1 \ldots t]$. Find
   $j := \max\{j \le n-r \mid S[r+1 \ldots r+j] = S[t+1 \ldots t+j]\}$;
   and set $Z_k := t + j$, $r := r + j$, and $l := k$;

情况1是k不在任何一个Z-box里，因此只能老老实实用定义计算Z_k，如果算出Z_k >0则更新l和r，因为这个Z-box的右端点肯定比其他Z-box要右。

需要详细说明的是情况2，即在一个Z-box里的情况。前面的例子对应于这里的2(a)情况。当$Z_{k'}$ <t，即还没有比较到这个Z-box的右边界时就已经失败了，则一定有Z_k=$Z_{k'}$；否则从r+1开始继续比较，相当于仍然按定义计算Z_k，只不过因为以前已经比较过了一些，这次只需要从中途继续比较。

算法的正确性是显然的，但时间复杂度并不是很明显，因为并不是每个Z_k计算都是常数的。不过由于每次除字符比较之外的部分都是常数，因此只需要考虑比较总次数。

œg：每次失配终止一次迭代，因此失配次数≤|S|；

œg：每次匹配给r至少增加1，因此匹配次数≤|S|。

因此总比较次数≤2|S|，时间复杂度为O(|S|)。

基于Z算法的线性时间匹配算法用Z算法可以得到一个线性时间匹配时间，它也许是最简单的线性时间匹配算法了。

假设模板是P[1...n]，而文本是T[1...m]，则令新串S=P$T，其中$是P和T中都没有出现过的字母，然后对i=2,...,$m+n$+1计算Z_i(S)，时间复杂度为O($n+m$)。由于$的存在，对于所有i都有$Z_i \le n$。对于所有$i > n$+1，当且仅当$Z_i$=$n$时P在T的位置$i$-($n$+1)出现。

完全使用原始的Z算法需要O($n+m$)空间，但是对于匹配来说并不需要。由于$的关系只需要储存满足$i \le n$的$Z_i$，因此空间复杂度仅为O($n$)。

Boyer-Moore算法（BM算法）这是几乎最实用的精确匹配算法。在字母表大（如自然语言）或模板长（如生物应用）的时候尤其有效。一般来说文本T[1..m]中不是全部字符（往往只有约m/n个字符）会被算法检查。BM算法的核心思想是尽量把模板右移（每次移动后都将重新匹配，这一点和KMP算法不同），有三个基本规则：

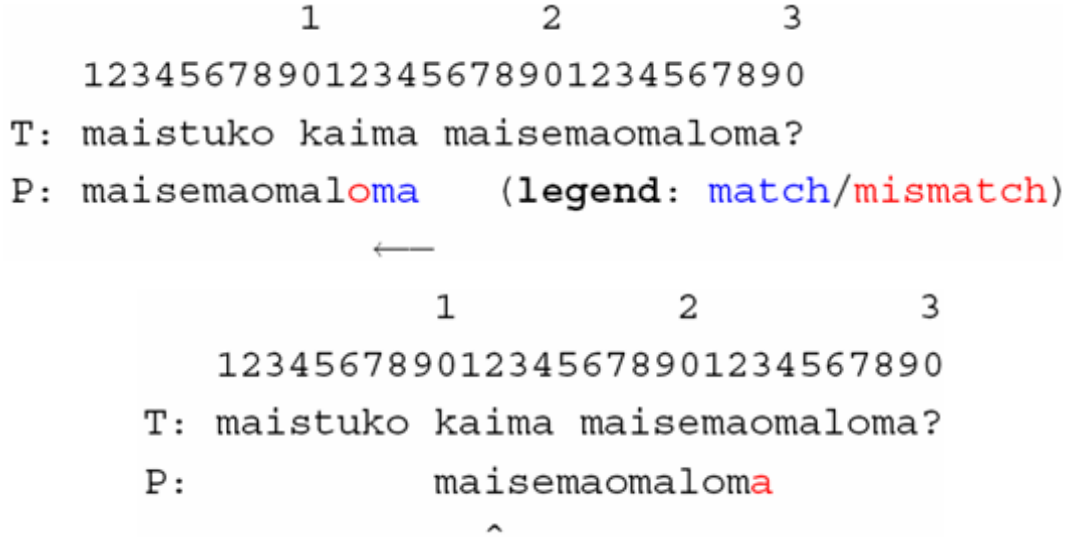
从右到左扫描：从右往左检查模板P，即P[n], P[n-1], ...

坏字符规则：避免在同一个字符上的连续失配

好后缀规则：移动时总是让已匹配的模板字符和已经匹配到的目标字符对齐

每一条规则都可以单独使用，但结合使用时更有效。第一条规则很简单，如图8.1：

坏字符规则也不难理解，如图8.1.1，



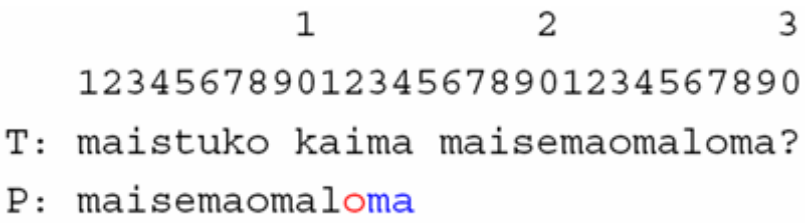
正式地说，首先用$\Theta(|\Sigma|+|P|)$时间计算$R(x)=\max(\{0\}\cup\{i<n|P[i]=x\})$，然后当$P[i]\neq T[h]=x$时把模板P右移$\max\{1,i-R(x)\}$个单位，即：

如果x的最右边位置j在i的左边，则把它和T[h]=x对齐。

如果x的最右边位置j在i的右边，则右移一个单位（因为j左边每个位置都可能有x）。

如果x根本不在P[1...n-1]里，则shift=i，整个模板P和T[$h+1...h+n$]对齐。

坏字符规则对于英文文本非常有效（因为失配经常出现），但字母表很小时可能每个字母的最右位置离右端点都很近，因此坏规则起不到太大的作用。在这种情况下需要考虑P中已经成功匹配的后缀，方法就是规则三的强好后缀规则，它比原始BM算法中的（弱）好后缀规则更为有效。考虑图8.1.1的情形



则右移后P在12~14处的字母应该是xma，其中$x\neq o$这样才可能继续匹配。注意这里不能直接要求P在12~14处的字母是ima，因为这和T相关！为了使用好后缀规则，我们需要对P进行预处理，这个处理不能和T有关，因此只能规定“右移后P在12~14处的字母应该是xma，其中$x\neq o$”，这条规则的所有信息都只依赖于P。

假设P[i...n]已经和T的子串t匹配上，则有两种情况：

：P[i-1]失配，且P中有另外一份和P[i...n]内容完全一样的字串，且它的前一个字符不是P[i-1]，则让i左边且最考虑i的这份子串和t对齐。

′：如图8.1.1，虽然没有相同子串，但有一个“潜在”的匹配子串

```
          1         2         3
 123456789012345678901234567890123
T: mahtava talomaisema omalomailuun
P: maisemaomaloma
```

应该如图8.1.1这样移动：

```
          1         2         3
 123456789012345678901234567890123
T: mahtava talomaisema omalomailuun
P:             maisemaomaloma
               ^^
```

正式地说：如果不符合情况1，则把P右移尽量少个字符，使得t的一个后缀和P的一个前缀匹配。作为特例，这个t的（最长）后缀可以是空的，在这种情况下P右移|P|个字符。注意一次完全匹配也属于情况2。

情况1的预处理对于i=2, ..., n+1，令满足“不紧跟字符P[i-1]之后的，内容和P[i...n]一样的子串”集合为S_i，则定义L’(i)为S_i中右端点的最大值。如果S_i为空，则L’(i)=0。显然$0 \le L'(i) < n$，且若$L'(i) > 0$，它就是情况1中所求子串的右端点，因此移动$n - L'(i)$个字符。另外，由于$P[n+1...n] = \varepsilon$，L’(n+1)是满足$P[j] \neq P[n]$的最大j，若所有字符都相等则$L'(n+1) = 0$。

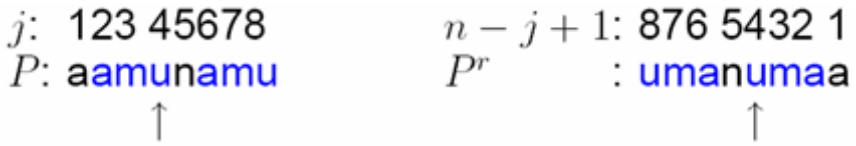
```
          1
 12345678901234
P: maisemaomaloma
```

如图8.1.2，L’(15)=13，L’(14)=0，L’(13)=7, L’(12)=10, L’(11)=L’(10)=L’(9)=...=L’(2)=0。

定义$N_j(P)$为P[1...j]和P的最长公共后缀长度，则$0 \le N_j(P) \le j$。例如对于上面的例子，$N_0(P)=N_1(P)=0$, $N_2(P)=2$, $N_3(P)=...=N_6(P)=0$, $N_7(P)=2$, $N_8(P)=N_9(P)=0$, $N_{10}(P)=3$, $N_{11}(P)=...=N_{13}(P)=0$, $N_{14}(P)=14$。显然N_j和Z_i代表的串互为逆序的，因此

$$N_j(P) = Z_{n-j+1}(P^r)$$

例如图8.1.2：



这样我们可以在O(|P|)时间内在P的逆序串P^r上用Z算法得到N_j值。有了N_j，我们可以立刻得到L'(i)值，因为我们有：

定理：若$L'(i) > 0$，则它是满足$N_j(P) = |P[i...n]| = n - i + 1$的最大$j$（$j < n$）。

这样，我们很容易像图8.1.2中那样计算出L'：

for $i := 2$ **to** $n + 1$ **do** $L'(i) := 0$;
for $j := 1$ **to** $n - 1$ **do** $L'(n - N_j(P) + 1) := j$;

情况2的预处理。对于$i \ge 2$，令l(i)为满足“$P[1...l(i)]$是$P[i...n]$的后缀”的最大值。例如对于P=P[1..5]=ababa，l(6)=0, l(5)=l(4)=1（a）, l(3)=l(2)=3（abc）。关于$l(i)$我们有以下定理：$l(i) = \max\{0 \le j \le |P[i...n]| \,|\, N_j(P) = j\}$，这样我们可以利用$N_j$在线性时间内计算出$l(i)$。总结一下，我们有：

好后缀规则：若P[i-1]是一次失配，则可以考虑使用好后缀规则。若$L'(i) > 0$（情况一），则右移$n - L'(i)$字符，否则（情况二）右移n-l(i)个字符。注意如果P[n]失配，则$i = n + 1$，好后缀规则仍然适用。当找到一个匹配后，模板应右移动n-l(2)个位置（想一想，为什么）。

完整的BM算法组合两个规则，我们得到了完整的BM算法：

预处理：对每个字符x计算R(x)，并对i=2,...,n+1计算L'(i)和l(i)

主过程：图8.1.2中循环。

时间复杂度。BM算法的分析比较复杂，这里只给出结论：当P不在T中出现时最坏情况下是线性的，否则最坏情况是$\Theta(nm)$的。Galil规则修正了这个瑕疵，得到了最

```
while k ≤ m do
    i := n; h := k;
    while i > 0 and P[i] = T[h] do
        i := i − 1; h := h − 1;
    endwhile;
    if i = 0 then
        Report an occurrence at T[h + 1 . . . k];
        k := k + n − l(2);
    else // mismatch at P[i]
        Increase k by the maximum shift given by the
        bad character rule and the good suffix rule;
    endif;
endwhile;
```

坏情况线性的修正BM算法。对于自然语言来说BM算法的运行时间几乎总是亚线性的。

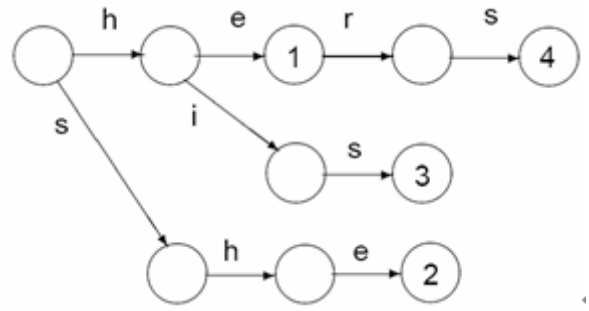
6.4.3 集合匹配和Aho-Corasick算法

本节讨论**精确集合匹配问题(exact set matching problem)**，即在文本T[1...m]中找模板集合$P = \{P_1, \ldots, P_k\}$中任何一个串的匹配。记$n = \sum_{i=1}^{k} |P_i|$，则精确集合匹配问题可以通过k次串匹配在O($n + km$)时间内解决，但当k很大时仍然不理想。Aho-Corasick算法(AC)是解决集合匹配问题的经典方法，它的时间复杂度为O($n + m + z$)，其中z为匹配的次数。Aho-Corasick算法是基于关键码树的，我们先讨论它。

模板集P的**关键码树(keyword tree)**或**字母树(trie)**是一个有根树K，它的每条边标记为一个字符，从同一节点出发的不同边标记有不同的字母。定义节点v的标记L(v)为从根到v路径上所有边的标记顺次连接形成的字符串，则trie的任意叶子v的标号都是P中的字符串，且对于P中的每个字符串P_i都能找到trie中的一个叶子v使得v的标号为P_i。

图8.1.2是字符串集合P={he, she, his, hers}对应的trie。可以看出，trie可以用来表示字符串字典(dictionary)。

Trie的构造和查找根据定义很容易得到trie的构造和查找算法。构造过程是增量的，每加入一条串时只需要眼着树走，遇到叶子（且串没有结束）时增加子树，并沿



途设置边的标号。显然构造过程是O(n)的，但有一个隐含条件：可以在O(1)时间内找到每个结点出发的任意标号的边。显然这可以通过给每个结点储存一个儿子数组来实现，但当字母表很大时空间开销太大。这个问题暂时留在这里，以后我们再讨论。注意到构造过程实际上已经包含了查找过程，因此查找是O(|P|)的。

由这个查找算法很容易得到集合匹配问题的O(nm)算法：对于文本的每个开始位置在trie上执行一次查找操作即可。下面通过把trie转化为自动机来支持线性时间查找。

Aho-Corasick的自动机。事实上，Aho-Corasick算法是KMP算法在字符串集合上的推广。和KMP类似，它把trie里每个结点看成一个状态，初始状态为根0，且自动机包含三个函数：

一、*goto函数g(q, a)*：即当前状态匹配到字符a后的下一个状态。

若(q, v)的标号为a，则g(q,a)=v；对于根出发的边中不存在的字符a，g(0,a)=0（在根处扫描非匹配字符时仍在根中），其他情况$g(q,a)=\emptyset$。

二、*失配函数f(q)*（$q\neq 0$）：即一次失配后的下一个状态。

f(q)是L(q)的最长真后缀w对应的结点，其中w是,ø的前缀。注意f(q)总是有定义的，因为$L(0)=\varepsilon$是任何模板的前缀。

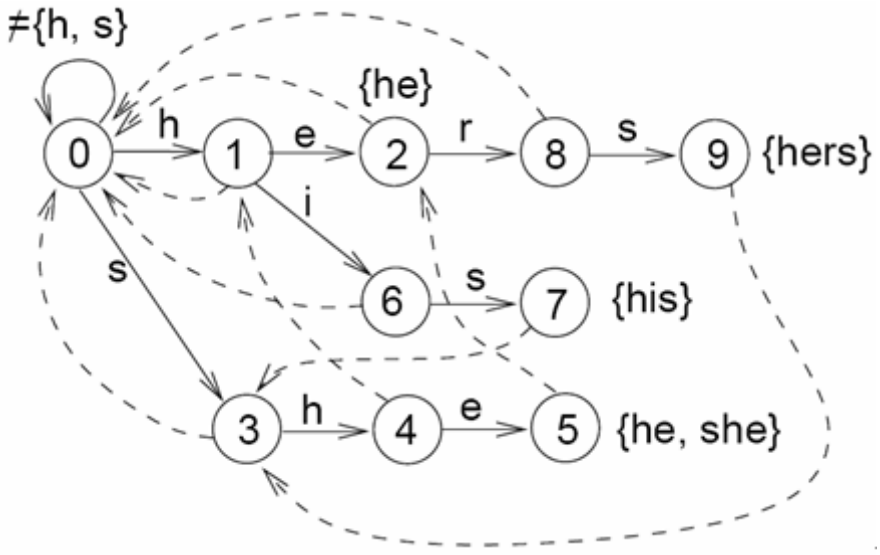
三、*输出函数out(q)*：进入状态q时被匹配的所有模板构成的集合。

图8.1.2给出了P={he, she, his, hers}对应的AC自动机，其中虚线是失配函数转移。可以看出，它只是在trie上增加了根到自己的转移边和失配转移函数。

有了AC自动机，我们很容易得到AC算法：

这进一步证实了它是KMP算法的推广。

时间复杂度在AC中寻找文本T[1...m]的时间复杂度为O($m+z$)，其中z是模板出现的总次数。证明：对于每个目标字符，自动机进行0次或多次失配转移，接着一次匹配转移。匹配：每次匹配转移要么留在根，要么让当前状态q的深度增加1，因此q深度的




```
q := 0; // initial state (root)
for i := 1 to m do
    while g(q, T[i]) = ∅ do
        q := f(q); // follow a fail
    q := g(q, T[i]); // follow a goto
    if out(q) ≠ ∅ then print i, out(q);
endfor;
```

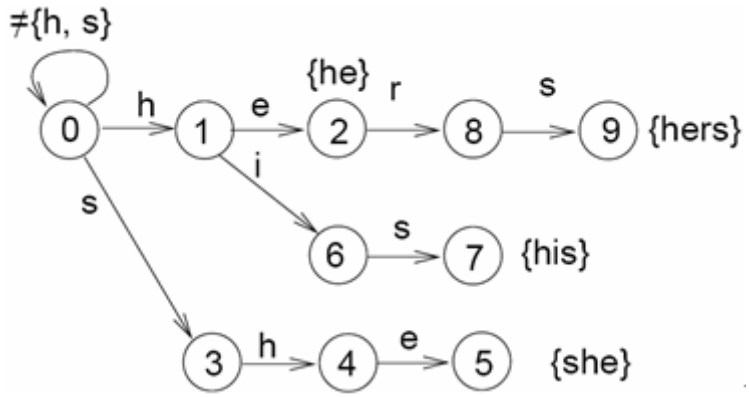
增加次数不超过m次。

失配：每次失配将把当前状态q移到离根更近的地方，即严格减少q的深度。因此q深度的减少次数不超过m次。

而每次匹配成功需要O(1)的时间报告，因此总时间为O($m+z$)。AC自动机的确为我们带来了线性时间的集合匹配，但剩下的问题是：如何给模板集合构造AC自动机呢？

AC自动机的构造AC自动机的构造分为两步，一是构造trie并设置goto函数，二是计算失配函数。显然这一步只需要O(n)时间（假定字母表大小为常数），注意设置每个字符串P对应的结点v的out(v)={P}，并对根出发边里不存在的字符a设定g(0,a)=0。注意out函数不一定出现在叶子中，如图8.1.3。

第二步和KMP一样，是一个让人困惑的步骤。KMP算法中计算f[u]用到了$v < u$的f值，这里也一样。不同的是这里是树结构，因此应该是利用祖先的信息。容易想到



用BFS完成这一过程，如图8.1.3：

```
Q := emptyQueue();
for a ∈ Σ do
    if q(0, a) = q ≠ 0 then
        f(q) := 0; enqueue(q, Q);
while not isEmpty(Q) do
    r:= dequeue(Q);
    for a ∈ Σ do
        if g(r, a) = u ≠ ∅ then
            enqueue(u, Q); v := f(r);
            while g(v, a) = ∅ do v := f(v); // (*)
            f(u) := g(v, a);
            out(u) := out(u) ∪ out(f(u));
```

其中*步和KMP算法的道理类似，而输出函数合并是因为恰好是在f(u)被识别的模板是L(u)的真后缀，因此也应该在u中被识别。和KMP算法的分析类似，对于任何一个串，*步一共执行不超过l_i次，因此总执行次数不超过O(n)。out集合可以用链表实现，则合并的时间复杂度是常数级别的。

这样，我们在O(n)时间内构造出了AC自动机。

6.4.4 后缀树与Ukkonen算法

本节讨论后缀树。考虑子串匹配问题。给一个文本串T[1...m]，要求在预处理O(m)时间下使得每个模板串P[1..n]可以在O(n)时间内匹配。这个问题非常实际，比

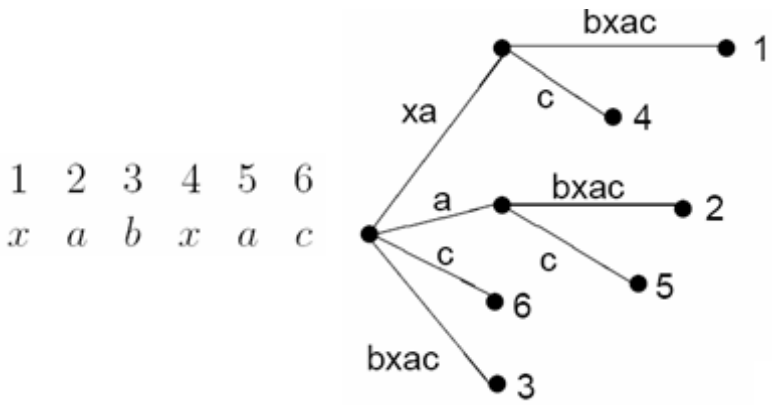
如T是一个庞大的数据库，而P是人工查询的字符串，长度一般很小，则即使O($m+n$)的时间也是无法忍受的，必须要O(n)才行。

为了实现O(n)查询，我们想到了trie：如果把T的所有子串构造出一棵trie，则可以达到O(n)查询。但构造时间呢？子串一共O(m^2)个，则构造时间高达O(m^3)！这对于巨大的m来说仍然是无法承受的。后缀树的巧妙之处在于它只储存T的所有后缀。储存后缀有用吗？答案是肯定的。任意子串T[$i...j$]是是T的后缀T[$i...n$]的前缀，而读者不难发现trie支持查询的同时实际上也是支持前缀查询的，因此后缀组成的trie组以支持O(n)时间子串查询了。但稍微分析后发现后缀组成的trie空间复杂度为O(m^2)，建立时间至少需要O(m^2)，仍然无法满足要求。但这是一个好的思路，我们不妨顺着它走下去。

后缀树(suffix tree)。S[1...m]的后缀树是一棵有根树T，它有m个叶子，标记为1,..,m。除根之外每个内结点都至少有两个儿子，每条边上有S的一个非空子串，且同一结点出发的任两条边上标记的子串的第一个字母都不相同。和trie一样用L(v)表示结点v的标记，即从树根走到v时经过边的子串的连接。后缀树最重要的性质是：对于i=1,...,m，有L(i)=S[$i...m$]。

容易看出后缀树和后缀的trie很接近，唯一的区别是边上的标记上一个子串而不是单个字符，因为也就多了个度数限制（度数为1的话可以和父亲合并）。最后，所有后缀必须和叶子对应，而不能是内结点。

图8.1.3是字符串xabxac的后缀树：



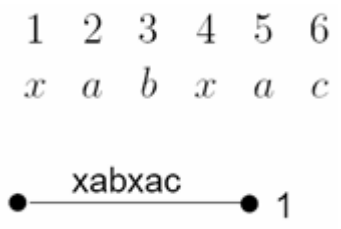
细心的读者也许注意到了：有时候无法构造出一个字符串S的后缀树！如果S有一个后缀为另一个后缀的前缀，则短的那个一定无法是叶子。为了避免这种情况，我们通常要求S以一个特殊字符$结尾（如果没有，就加一个）。$是S中不曾出现的字符，

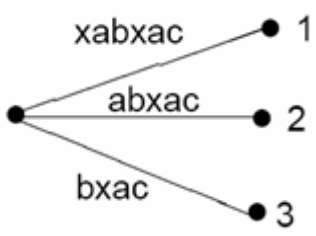
往往称为**终结符(termination character)**。这样，每个后缀一定可以和叶子对应，后缀树一定存在。

有了后缀树，我们可以这样解决子串问题：首先给T建立一棵后缀树，然后按照trie查找算法进行匹配。需要注意的是匹配结束（不管是成功还是失败）的位置可能在一条边标记串中间。匹配成功时位置的所有后代叶子都对应于一个匹配点，它们可以在O(z)时间内得到。

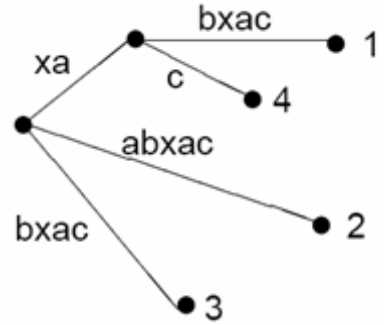
后缀树的构造后缀树显然可以通过逐一插入后缀来实现，和trie类似：初始时只有两个点：根和叶子1，以一条边连接，标记串为S\$。每次插入一个后缀K_i=S[i...m]\$时，按照查找算法从上到下走，直到一次失配（一定会出现）。这时在失配处插入一条边和叶子i，但有可能还需要在失配处把边一分为二。

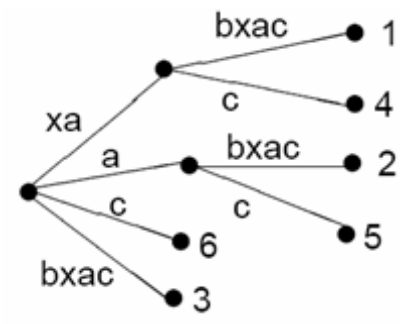
图8.1.3是串xabxac的构造过程。首先插入后缀1，2，3：



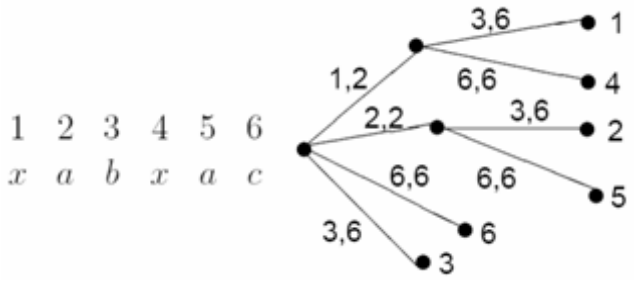


插入后缀4（xac）时引起第一条边分裂，而插入后缀5（ac）时引起第二条边分裂，如图8.1.3





可以证明，后缀树上所有边的子串长度和是O(m^2)的，但边的个数只有O(m)个，因此需要用更简单的方法表示后缀树。如果直接记录边上的子串，由于空间是O(m^2)的，时间不可能达到线性。由于边上的标记串是S的子串S[i...j]，所以只需要记录二元组(i,j)而不是整个子串内容，如图8.1.3：



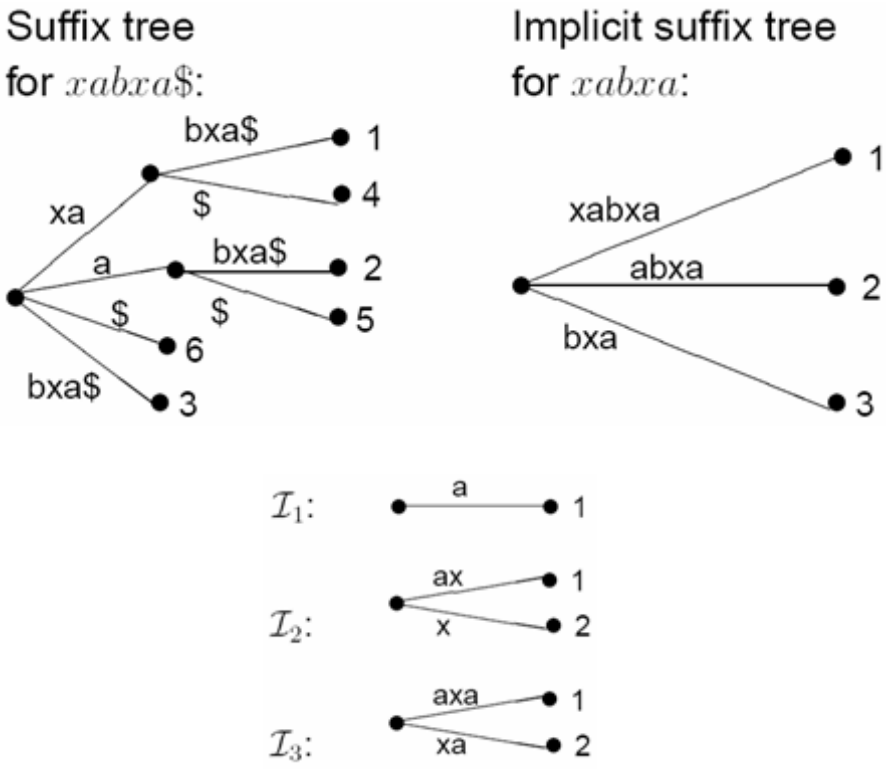
每条边上的空间是常数的，因此总空间消耗是O(m)，有希望在这种表示下找到构造后缀树的线性时间算法。

在线性时间内构造后缀树并不是一件简单的事。Weiner于1973年提出了第一个线性时间构造算法，可惜该算法太复杂，不易理解。McCreight于1976年提出了一个内存更省的线性时间算法，但仍然很复杂。Ukkonen于1995提出了一个相对比较简单的算法，拥有Weiner和McCreight算法的所有优点，并且内存更省。下面我们将介绍这一算法。

Ukkonen构造算法虽然已经相对简单，Ukkonen算法仍然比较复杂。在深入学习之前，我们需要了解Ukkonen算法的梗概。

Ukkonen算法是一个在线算法。换句话说，它从左到右扫描每个字符，在扫描第i个字符时，前i-1个字符组成的串的“后缀树”已经被构造好了。这里需要给后缀树加上引号，因为前i-1个字符构成的串不一定存在后缀树——它的结尾并不是$。图8.2.1的例子能让这个结论更加直观。

在刚才的例子中，后缀xa和a都没有对应于叶子，而是对应于两条边的“中间”。更正式地说，可以给每条边的任意“中间位置”定义所谓的**串路径(string path)**，它是该边指向结点标号的前缀，并把这种后缀可能对应中间位置的“后缀树”称为**隐式后缀树(implicit suffix tree)**。设前i个字符对应的隐式后缀树为T_i，则Ukkennon算法就是从T_1出发，构造T_2，T_3，...，T_{m+1}，其中S[m+1]=$，因此$T_{m+1}$是真正的后缀树。每次构造一棵新树成为一个**阶段(phase)**，整个这个过程被形象的称为“生长过程”，如图8.2.1。



可以看出，把树T$_i$生长成T$_{i+1}$过程就是给每个后缀S[j...i]对应的路径（称为“**生长点(growing point)**”）最后加一个字符。具体来说，要分成三种情况。假设我们要扩展生长点S[j...i]，给它加上字符S[i+1]。

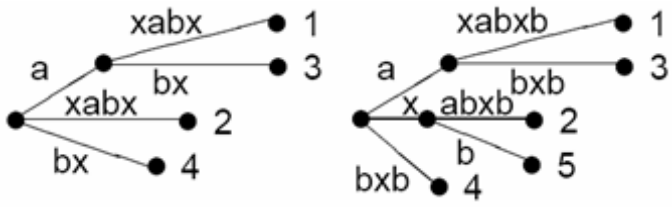
情况一：S[j...i]以叶子结尾，则直接在S[i+1]的标号后增加字符S[i+1]即可。

情况二：S[j...i]在中间结束，且后一个字符不是S[i+1]，则在该处增加一个分支，连接到一个新叶子f，该边的编号为S[i+1]。如果结束点在一条边的中间，还需要进行边分裂。

情况三：S[j...i]的后一个字符本来就是S[i+1]，则什么都不做，因为希望增加的后缀S[j...i+1]已经在树中了。

前两种情况称为非空扩展(non-void extension)，而情况三称为空扩展(void extension)。图8.2.1是给axabx对应的隐式后缀树生成字符b的情况。给后缀1，2，3，4生长时符合情况1，直接在最后一条边后面增加b即可。后缀5生长时符合情况2，分裂一条边并增加新结点。后缀6（对应于空）生长时符合情况3，什么也不做。

用最简单的方法，扩展长度为l的后缀需要O(l)的时间（先查找，再扩展），因此一次迭代（生长一棵树）的时间为O(i^2)，生长n次的总时间为O(m^3)，完全无法忍受。



为了进一步讨论，我们先来证明如下的：

三阶段定理：每一次树生长的过程是：先进行一些（至少一次）情况1扩展，再进行一些（可能0次）情况2扩展，最后进行一些（可能0次）情况3扩展。注意这里的顺序后缀S[1...i]，S[2...i]，...，S[i...i]，S[i+1...i，即按树上le的顺序扩展各个生长点。

三阶段定理表明：在一次树生长过程中，各生长点的扩展是很有规律的，三个情况按顺序出现。

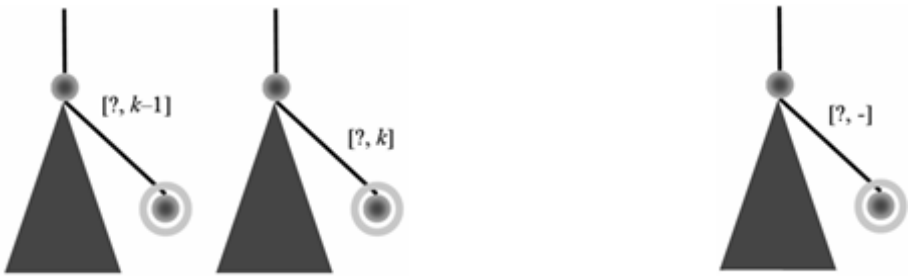
另两个有趣结论是关于Œ的生长情况：

结论一：一日为叶子，终身为叶子。

结论二：在所有阶段中，情况二的次数一共O(|S|)次。

现在程序实现细节。让我们更加深入的观察三种扩展情况。

情况一：只需要更改叶子标号，从$[?.k-1]$改到$[?,k]$。但如果我们改一下用$[?,-]$来标记边，则什么都不用改了，树结构没变化，甚至结点标号和生长点位置都不变！



情况二：前面提到过，所有情况二一共只有O(|S|)个，不管是匹配到内结点（图8.2.1中(a)）还是边的中间（图8.2.1中(b)），都可以在常数时间内扩展完毕。

情况三：不管新的生长点在结点上还是仍在边的中央，树的结构都未发生改变，只有生长点移动。

由于情况一什么都不用做，情况二一共只有O(|S|)次，因此问题的关键在于如何快速处理情况三。事实上，情况三的总数可以达到$\Omega(|S|^2)$，因此必须用某种办法“忽略”一些扩展。如果一一处理的话，哪怕每次只用常数时间也不行。

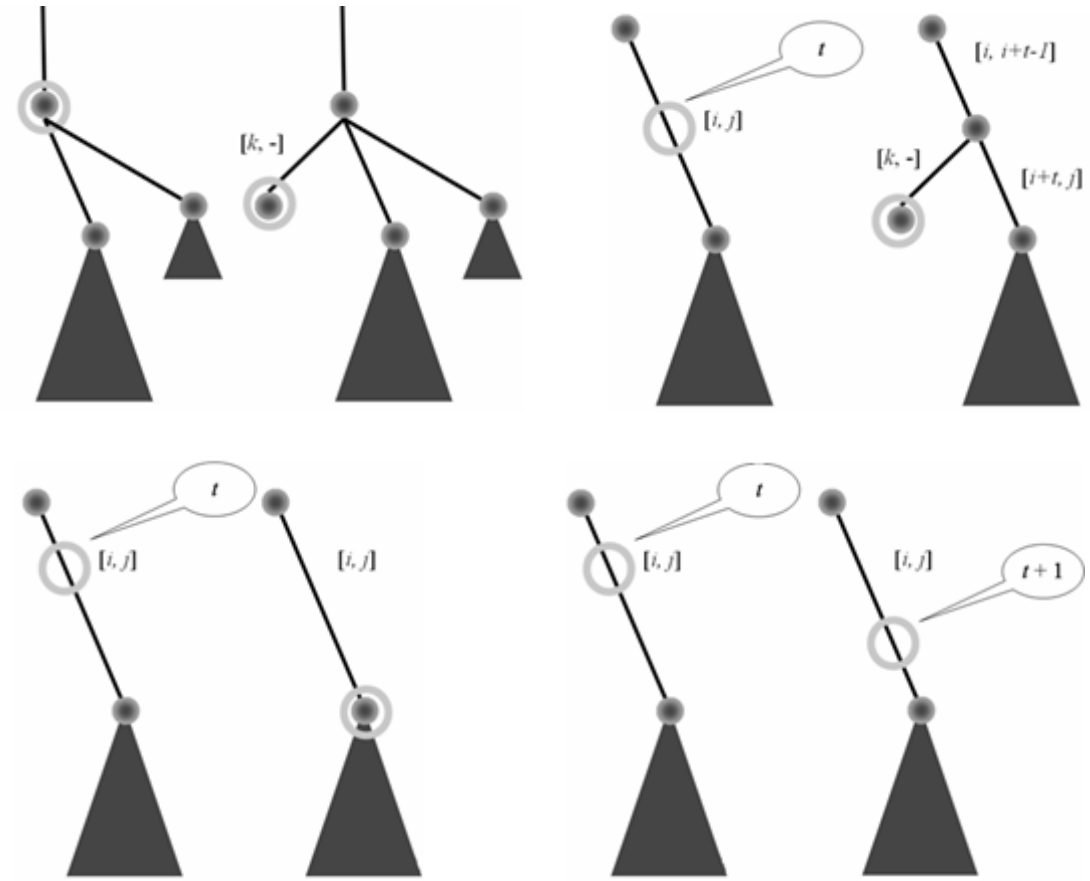
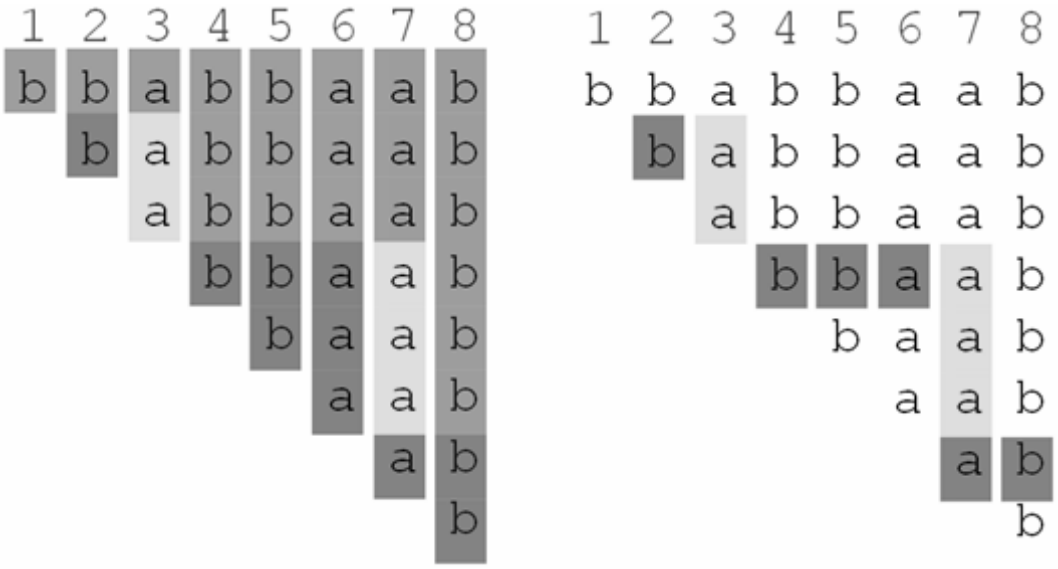


图8.2.1或许对理解后面要介绍的suffix link有帮助。左图是保留所有生长点的方法，每一行对应一个生长点，第i列代表算法的第i阶段，每一阶段从上到下依次扩展每一个生长点。中灰、浅灰和深灰分别代表情况一、二、三，在同一列（阶段）中总是按情况一、情况二、情况三这样的顺序出现。由于“一日为叶子，终身为叶子”，情况一的边界应该是阶梯型的。我们去掉所有情况一，每列只保留第一次情况三（如果有），则可以得到图(b)，它是一条从左上角到右下角的路径！Ukkonen算法的巧妙之处在于：它只保留一个生长点（活动生长点），让它按这条路径从左上角走到左下角，完成所有需要的工作。显然这条路径的长度为O(|S|)的，因此Ukkonen算法的时间复杂度是线性的。

图8.2.1非常关键。许多自学Ukkonen算法遇到困难的人都是因为始终没明白为什么会出现suffix link这个东西，以及为什么可以把明明需要处理（即移动生长点）的情况三给“忽略”掉。

读者也许还记得数据结构部分介绍的“惰性删除”（只做删除标记，不真正删除，

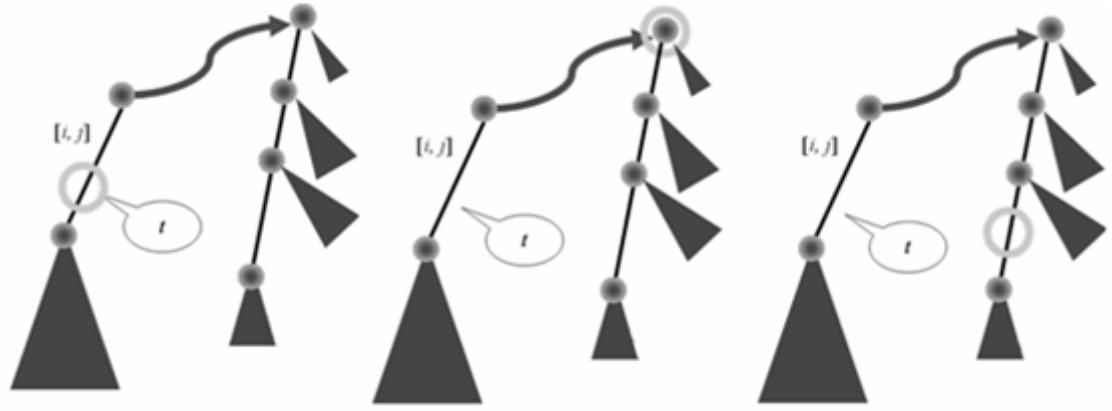


而到了特定时候再删除所有标记结点），以及Fibonacci堆的惰性合并（简单的插入到链表中，只有在删除最小值时不得不得执行清理操作），那么这里的生长点也是在“惰性扩展”！情况一本来就可以完全忽略（什么都不用做，生长点也不动），情况三是在不改变树结构的情况下仅仅移动生长点。如果一个生长点连续进行k次情况三扩展，那么它完全可以前k-1次啥都不做（惰性），最后一次扩展一下子移动k步。这样的情况反映到图(b)上就是每个阶段只处理一个情况三扩展，因为其他情况三扩展总是可以延迟，等到以后一起处理。

活动生长点算法下面考虑这个“活动生长点”算法。显然水平移动非常简单：就是把活动生长点往下移动即可，复杂的是垂直移动：活动生长点改变了，在树上看的话就是生长点“没有规律的乱跑”。因此问题的关键是：在情况二扩展（即垂直移动）时，活动生长点应该跑到哪里。Ukkennon算法在处理这点的时候遵循了“前人种树，后人乘凉”的原则，第一次找到垂直移动的目的地时，把这次移动的起点和终点用一个链接记录下来，下次再需要从同样的起点出发时就可以直接走到终点去，而不用另找一次。这些链接称为**后缀链接(suffix link)**，它有时也被称为断头指标，因为终点对应字串是起点对应字串的**“断头串”(second suffix)**，即去掉第一个字符以后剩下的串。

显然后缀链接的起点是内结点，而不会是叶子或者边的中间，且任意内结点都是某个后缀链接的起点，图8.2.1是利用suffix link进行垂直移动的例子。

方法非常直观：首先往上走到最近的内结点（设走了t个字符），然后顺着suffix link走（它已经存在），然后往下匹配t个字符。最后别忘了创建新的suffix tree，方便



后人。这个过程显然可以在O(t)时间内得到，但更聪明的做法只需要O($1+d$)的时间，其中d是在第二阶段中经过的内结点总数。设第i次情况二扩展的d值为d_i，我们只需证明$d_1+d_2+\ldots+d_{|S|}=|S|$

设生长点的势能Φ为该生长点离它上方最近内结点的距离，则每次情况三扩展最多让Φ增加1（有时还会让它减少），但第i次情况二扩展总是让Φ减少至少d_i，因为势能原来是t，现在为最多t-d_i（走过了d_i个内结点后最多只能往下走t-d_i步），因此势能减少了至少d_i。由于势能在任何时候都是非负的，因此$d_1+d_2+\ldots+d_{|S|}=|S|$。

这样，我们完成了整个算法，时间复杂度为O(|S|)。最后不要忘记加入$符号，然后拆除所有后缀链接，只保留后缀树。

6.4.5 后缀数组和Skew算法

上节我们学习了后缀树及其线性时间构造算法：Ukkonen算法。这是一个相当巧妙的算法，但细心的读者也许已经发现了它的致命弱点：空间开销大且对大字母表时间效率不理想。本节介绍后缀树的一个有效替代品：后缀数组，它在字母表较大时尤其有效，且构造简单，时间效率也颇高。虽然Farach于1997年提出了一个针对整数字母表的线性时间后缀树构造算法，但该算法和本节介绍的线性时间后缀数组构造算法比较起来太复杂了。

和后缀树一样，在使用之前我们给字符串增加一个特殊字符$，规定$比字母表中其他字符都小。为什么要规定大小呢？这是因为后缀数组实际上是S所有后缀的排序。排序是根据字典序进行的，规定$最小才不会让这个辅助字符$改变了原来的顺序。*后缀数组和后缀树的关系相当于有序字符串集合和它的Trie之间的关系*。始终记住这一点，将对理解后缀数组及其算法有很大帮助。

前面说过，给字符串排序时一般使用间接排序，即只需得到一个排列P_i，使得$S_{P_1} \leq S_{P_2} \leq \ldots \leq S_{P_n}$，而不需要移动字符串本身，因为字符串往往较长，移动开销很大。后缀数组也一样，它的含义是S的各个后缀的排列SA，它满足：

$$Suffix(SA[i]) < Suffix(SA[i+1]) \quad (1 \leq i < n)$$

注意这里不用等号，任何两个后缀的长度不同，因此不可能相等。为了方便，我们另外定义一个名次数组Rank=SA^{-1}，即若SA[i]=j，则Rank[j]=i，因此Rank[i]保存的是后缀Suffix(i)在所有后缀中从小到大的“名次”。

后缀数组的构造显然后缀数组可以由后缀树构造出来：给后缀树进行一次深度优先遍历（扩展时按照字母顺序从小到大），则叶子的访问顺序就是后缀数组SA。但这违背了我们的初衷：我们研究后缀数组的目的是避开后缀树！根据定义，后缀数组可以通过给n个后缀进行排序得到。给字符串排序的最好方法是基数排序，但无法突破下界$\Omega(n^2)$，因为这n个后缀的字符总数就已经达到了！这也提醒我们：要想快速构造后缀树，*必须利用后缀之间的联系*。我们不是在给n个不相干的字符串排序。

对于字符串u，我们定义u的k-前缀(k-prefix)为u的前k个字符组成的字符串（如果k过大则为整个u），定义k-前缀的比较关系$<_k$为：u$<_k$v当且仅当$u^k<v^k$。类似可以定义$=_k$和$\leq_k$。这些运算的直观含义是：最多比较k个字符，比较结束之前如果分出大小就立刻停止，否则到第k个字符比较完后也停止，忽略后面部分。

有了k-前缀比较关系，我们有三个非常重要的性质：

51：对于$k \geq n$，则Suffix(i)$<_k$Suffix(j)当且仅当Suffix(i)<Suffix(j)

这是显然的，因为k太大了，还没有比较到第k个字符就已经分出大小了。

52：Suffix(i)$=_{2k}$Suffix(j)等价于

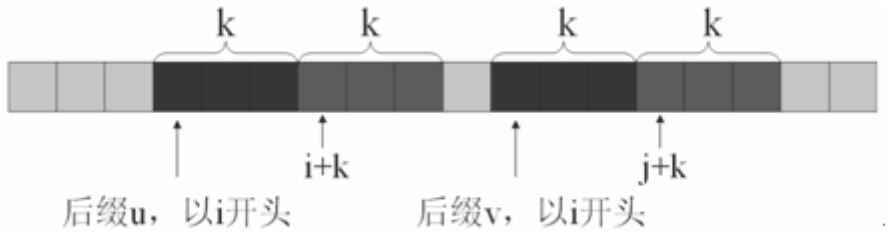
$$\text{Suffix}(i)=_k\text{Suffix}(j) \text{ and } \text{Suffix}(i+k)=_k\text{Suffix}(j+k)。$$

53：Suffix(i)$<_{2k}$Suffix(j)等价于

$$\text{Suffix}(i)<_k\text{Suffix}(j) \text{ or } (\text{Suffix}(i)=_k\text{Suffix}(j) \text{ and } \text{Suffix}(i+k)<_k\text{Suffix}(j+k))$$

这两个式子看起来比较复杂，但图8.2.1很清楚的表示了它的含义：

要比较$2k$个字符，可以先比前k个，再比后k个。这样，性质2和性质3也是显然的了。这里有一个潜在的问题是如果$i+k$或者$j+k$太大，超过n了怎么办？这个字符不就



没意义了么？不要忘了最后一个字符是\$，比较到这里时肯定可以分出大小，因此不会让指针移到串外的。

根据k-前缀比较运算，我们可以定义k-后缀数组SA_k和k-名次数组Rank_k，和传统的后缀数组、名次数组的差别是比较运算：从通常的字典序比较运算变成了k-前缀比较运算。这两个数组构成了后缀数组的增量算法的核心，因为从SA_k出发可以在$\mathrm{O}(n)$时间内得到Rank_k，而从Rank_k可以在常数时间内比较任意两个后缀：

$$Suffix(i) <_k Suffix(j)\ iff\ Rank_k[i] < Rank_k[j]$$
$$Suffix(i) =_k Suffix(j)\ iff\ Rank_k[i] = Rank_k[j]$$

这样，把所有后缀按$2k$-前缀比较关系排序相当于把后缀$\mathrm{Suffix}(i)$以$\mathrm{Rank}_k[i]$为主关键字，$\mathrm{Rank}_k[i+k]$为次关键字按k-前缀比较关系进行字典序排序。这个字典序排序可以通过基数排序完成，时间复杂度为$\mathrm{O}(n)$。这样我们在$\mathrm{O}(n)$时间内从SA_k和Rank_k得到了SA_{2k}。

最后一个问题是：如何求出SA_1和Rank_1。这实际上是把S的所有字符提取出来排序，用快速排序或者基数排序都可以轻易完成。这样，我们从SA_1和Rank_1求出SA_2和Rank_2，再求出SA_4和Rank_4，..，直到SA_m和Rank_m，其中m=2^k且$m \geq n$。根据性质1，SA_m和SA是等价的，因此一共只需要进行$\mathrm{O}(\log n)$次排序，每次排序的时间为$\mathrm{O}(n)$，总时间复杂度为$\mathrm{O}(n \log n)$。

Skew算法Juha Karkkainen和Peter Sanders于2003年提出了Skew算法，在整数字符集上用线性时间构造出了后缀数组。这个算法不仅效率高，而且非常简洁。在两位作者提出Skew算法的论文中，甚至有该算法完整的C++语言代码——只有50行之短。和复杂的后缀树构造算法相比有着天壤之别。

我们以S='mississippi'为例介绍Skew算法。首先把字符串的末尾添加\$（假设\$小于S中的所有字符），直到长度是3的倍数加1。Skew算法的设计者遵循了C语言的数组规范，因此如果设S为$[0...n]$，则n是3的倍数。我们把后缀$\mathrm{S}[i...n]$的起点记为i。

Skew算法分析为三步。

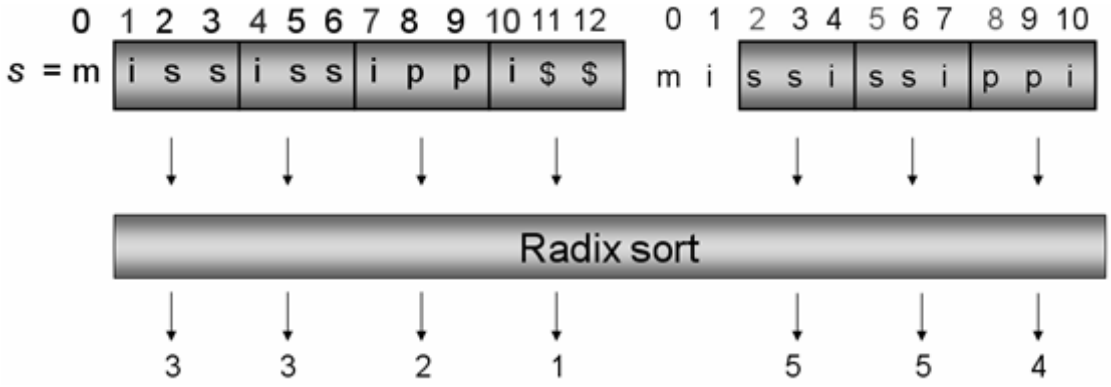
步骤一：把所有起点不是3的倍数的后缀排序，放到数组SA_{12}中。

步骤二：把所有起点是3的倍数的后缀排序，放到数组SA_0中。

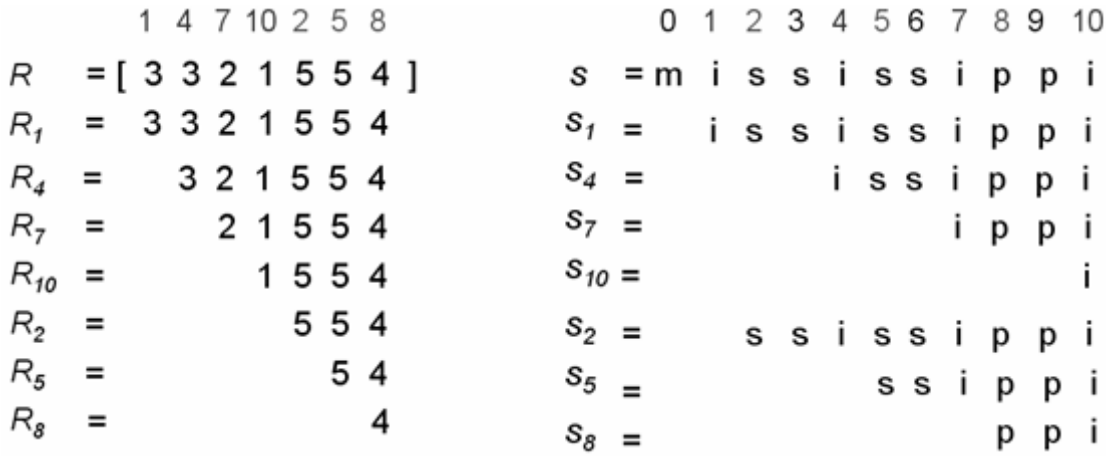
步骤三：合并SA_{12}和SA_0，得到SA。

其中步骤一最麻烦，步骤二和步骤三都非常简单（惊奇吗？）。我们一一叙述如下。

步骤一看起来比较神奇。考虑后缀S[1...n]和S[2...n]，把它们三个一个分组，得到$2n/3$个分组，然后进行基数排序，如图8.2.1。



得到名次数组R=(3,3,2,1,5,5,4)，但注意这里的串序号已经变了，是(1,4,7,10,2,5,8)。Skew算法的聪明之处在于利用了“给R进行后缀排序的结果就是SA_{12}”。图8.2.1便是一个例子。



在这个例子中，R的后缀数组=(10,7,4,1,8,5,2)（注意排序后要代入串序号），和SA_{12}完全一样。换句话说，R_i<R_j当且仅当S_i<S_j，其中S_i=S[i...n]。这样，问题转化为了求R的后缀数组，时间花费为T($2n/3$)。

步骤二非常简单，根本不需要像步骤一样三个一组的划分这么麻烦。把每个起点为3i的后缀写成二元组(S[3i], S[3i+1...n])，则二元组的两部分都可以在常数时间内比较大小（别忘了步骤一中已经把所有形如S[3i+1...n]的后缀排序好了），用一次基数排序即可完成步骤二，时间复杂度为O(n)。

步骤三也很简单，它只是个有序表合并问题。有序表合并可以在O(n)时间内完成，但前提是两个表中的元素都可以在常数时间内比较，即一个位于3i+1或3i+2的后缀和一个位于3j的后缀相比较。受到步骤二的启发，读者不难写出两种比较的规则：

规则一：后缀3i+1和后缀3j的比较时，比较前两个字符，然后再比较3i-1和3j-2。

规则二：后缀3i+2和后缀3j比较时，比较第一个字符，然后再比较3i+1和3j-1。

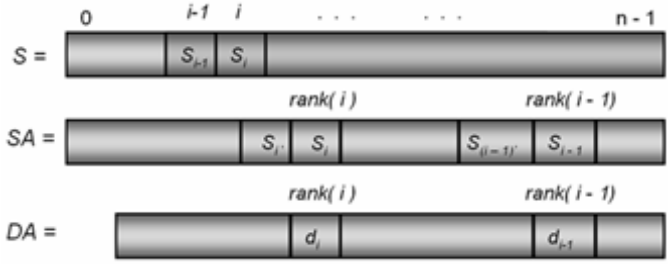
因此每两个后缀的比较是常数时间的（两个规则都转化为了直接读取Rank_{12}），因此步骤三的总时间为O(n)。

这样，我们得到了递归方程T(n)=T(2n/3)+O(n)，解得T(n)=O(n)。

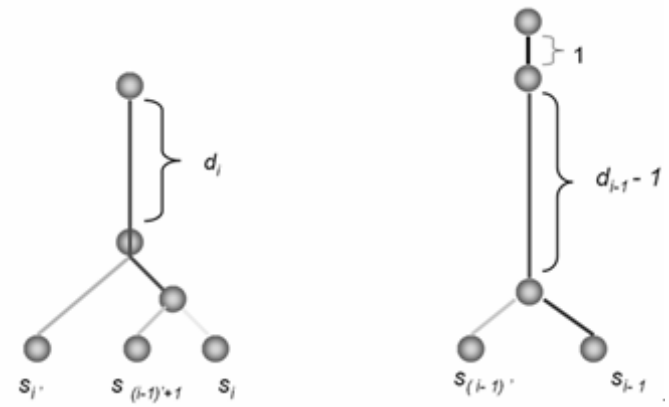
高度数组和LCP计算后缀数组只保留的排序结果，和后缀树比较起来缺少了很多信息。和后缀数组搭配使用的最常用武器是高度数组height[i]，它代表串SA[i]和SA[i+1]的最长公共前缀长度。为了计算height[i]，我们定义h[i]=height[Rank[i]]。注意Rank[i]是说后缀i在SA中的位置，因此h[i]就是：*在后缀数组中*，后缀i和它左边相邻后缀（注意这里的相邻是在SA中相邻而不是在S中相邻）的最长公共子串长度。可以证明如下引理。

引理：对i>1且rank[i]>1，有h[i]≥h[i-1]-1。

这个引理很直观。如图8.2.2，我们讨论的对象是S_{i-1}和S_i，它们实际上只相差一个字符！



用反证法。注意到把S_{i-1}去掉第一个字符后，两条树上的遍历路径应该是一样的。假设h[i]<h[i-1]-1，则右边走到最近公共祖先时左边已经超过了，这时需要增加一个分支。根据图8.2.2中右图的大小关系，这个分支应该应该加在左边！这样这个多出来的叶子夹在了S[i]和它在SA中的左相邻后缀之间，和定义矛盾。



这样，我们证明了引理。有了该引理，我们可以从小到大依次计算各个h值。h[1]只能完整比较S_1和$S_{1'}$，而从h[2]开始，h[2]只需要从h[i-1]个字符开始比较S_i和$S_{i'}$。第i（i¿1）次迭代的时间为(h[i]-h[i-1]+1)+1 = h[i]-h[i-1]+2，因此总时间为：

$$h_1 + \sum_{i=2}^{n-1}(h_i - h_{i-1} + 2) = h_1 + (h_{n-1} - h_1 + 2(n-2)) = h_{n-1} + (2n-4) = O(n)$$

这样，我们在O(n)时间内计算出了高度数组height。

有了高度数组，我们可以计算任何两个后缀的最长公共前缀（注意height是相邻两个后缀的最长公共前缀）。如果在后缀树上考虑这个问题的话，我们不难得出如下结论：

LCP定理：设$i < j$，则$LCP(i,j) = \min\{LCP(k-1,k) | i+1 \le k \le j\}$

这是一个RMQ问题。用本章开头介绍的标准RMQ算法可以在O(n)预处理时间后支持O(1)的LCP询问。

高度数组的一个有趣应用是从后缀数组构造后缀树。这个过程是线性的，留给读者思考。

值得一提的是：如果给后缀数组增加一个附加表，可以得到一种所谓的**增强后缀数组(enhanced suffix array)**。M.I.Abouelhoda, S.Kurtz和E.Ohlebusch在“Replacing

suffix trees with enhanced suffix arrays”中证明后缀树上的任意算法都可以用增强后缀数组来实现，有兴趣的读者可以阅读该论文。

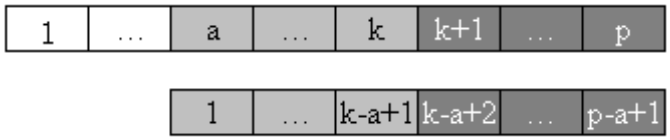
6.4.6 最长回文子串问题

最长回文子串问题：正序和逆序相同的串称为回文串(palindrome)，如abba或madamimadam，但abc不是回文串，因为abc和cba不相等。给长度为n的串S，求S的最长回文子串。

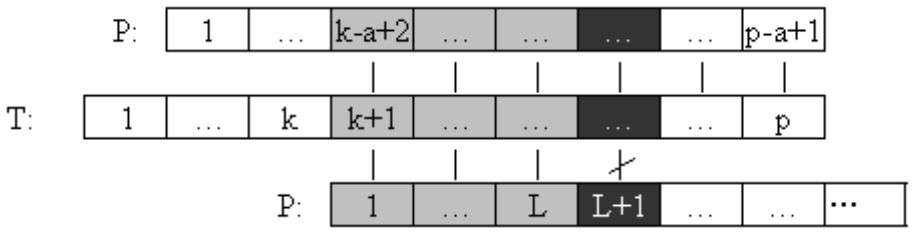
我们分别用分治算法和后缀树/后缀数组给出本问题的解决方案，前者的时间复杂度为$O(n\log n)$，后者为$O(n)$。为此，我们先介绍扩展的KMP算法。

扩展的KMP算法在本节的开始我们曾介绍过KMP算法，它的作用是求出模板在文本中的所有匹配。现在考虑一个更复杂的问题：求一个数组ext[1...m]，其中ext[i]代表在文本的位置i开始能匹配到的最大字符数。显然这个扩展问题包含了原问题的解。模板匹配位置就是那些让ext[i]=n的i。

设辅助函数next[i]为P和P[i...n]的最长公共前缀长度。设ext[1...k]已经计算好，且以前在匹配过程中到达的最远位置（即i+ext[i]-1）为p，且取到这个最大值的i是a，因此有：T[a...p]=P[1...$p-a$+1]，因此T[k+1...p]=P[$k-a$+2...$p-a$+1]，如图8.2.2。



令L=next[$k-a$+2]，则有两种情况。第一种情况如图8.2.2，如果$k+L<p$，则灰色字符对应相等，但黑色字符一定是不相等的，否则根据传递性，最上面的P和最下面的P的对应位置也应该相等，因此next[$k-a$+2]> L，矛盾！由此可知ext[k+1]=L。

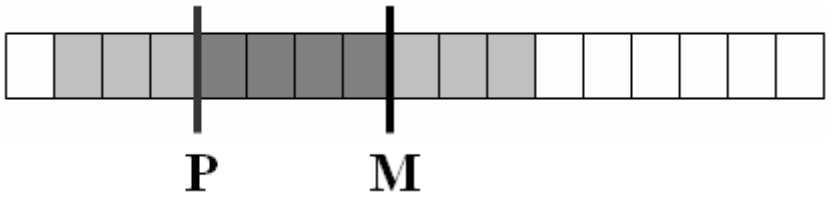


第二种情况如图8.2.2。由于p+1是以前从来没有比较过的，因此必须加以比较，即比较T[p+1]和P[$p-k$+1]，T[p+2]和P[$p-k$+2]...，直到比到不相等为止，这样就计算出了ext[k+1]。计算出以后需要更新p和a。

由于p是只增不减的，因此第二种需要真正比较的次数不会超过m次，时间复杂度是线性的。

下面遗留的问题是：如何计算next函数？聪明的读者一定已经看出了这个问题和KMP中遇到的前缀函数求解问题一模一样！KMP求解前缀函数的方法是自己匹配自己，而扩展KMP算法也是一样，这里不再赘述。

分治法考虑分治算法，把串平均分成两半，则问题的核心为求扩展分界线的回文串。



如图8.2.2，二分点为M。考虑中心在左边的串。以串中心为对称中心把M“反射”过去得到P，则P左边的字符数和M右边的字符数相同。我们把这里的P称为该子串的对称分界点。图上的串为回文串当且仅当PM中间的串是回文串，且从P往左，M往右得到的两个串完全相同。

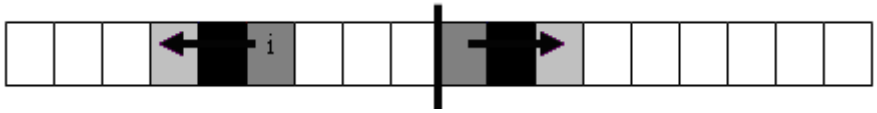
不难发现，条件一是必要条件（如果不满足，不可能是回文串），而条件二是优化条件（可以从P和M同时往两边扩展，扩展得越远越好）。因此对于一个给定的对称分界点P，对应的最长回文子串可以这样求出：

步骤一：判断PM之间是否为回文串。

步骤二：从P往左，M往右不断扩展，直到两边字符不相同。

用最笨的方法可以在O(n)时间内完成一个P点的检测，对于所有P点的总时间为O(n^2)。我们可以对这个方法进行改进，使得对一个P点的检测只需要O(1)的时间。为什么会这么快？因为我们可以用预处理得到想要的信息，而不用一次次重新算。

如图8.2.2，把左边的串记为L，右边的串记为R，则



步骤一：以r(L)为模板，L为主串，从P右边第一个字符开始能匹配到PM的中间么？

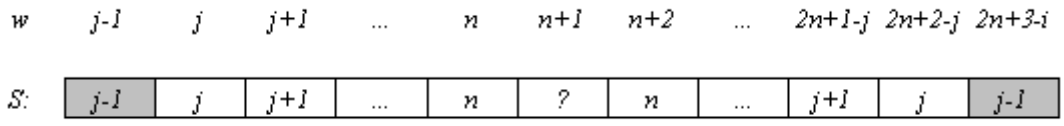
步骤二：以R为模板，r(L)为主串，从P左边第一个字符开始能匹配到多远？

由于扩展KMP可以算出每个位置能匹配到多远，因此只要两次扩展KMP以后（步骤一、二各一次），检测每个P点的时间都只需O(1)。

这样，我们分治法的时间复杂度满足$T(n)=2T(n/2)+O(n)$，解得$T(n)=O(n\log n)$。需要注意的是还需要枚举P在右边的情况，这时匹配顺序取逆。

用类似的方法还可以解决重复子串问题：形如xx的串称为重复子串，给长度为n的串S，求S的最长重复子串，读者可以试一试。这个问题还可以进一步扩展为：求出最大的K，使得形如xxx...x(重复K次)的子串在S中出现。

基于后缀树和后缀数组的方法利用后缀树，本题可以在线性时间内得到解决（如果字母表大小为常数的话）。做一个新串$w=S?S^R$，其中?为不在字母表中出现的元素。



如图8.2.2，$w[n+1]=?$，且$w[n+2]=S[n]$, $w[n+3]=S[n-1]$...。考虑以j为中心的奇数长度的回文子串，我们希望同时往左往右扩展。由于w的后半部分是S的逆序，这个扩展过程相当于是从$w[j]$和$w[2n+2-j]$同时往右扩展，即w的两个后缀$w[j...n]$和$w[2n+2-j...n]$的最长公共前缀，即LCP。不管是后缀树还是后缀数组，在建立后都可以在O(1)时间内完成任意两个前缀的LCP询问，即在O(1)时间内算出以任意字符为中心的最长回文子串，因此只需要枚举中心，总时间复杂度为$O(n)$。长度为偶数的回文串类似可求，时间复杂度也是$O(n)$。

6.4.7 小结和应用举例

本节大多数材料来源于生物信息学相关书籍和课程。读者可以参考台湾国立大学

的课程“生物信息学中的算法”：

http://www.iis.sinica.edu.tw/~hil/bioinfo/

还有芬兰UKU大学的类似课程。

http://www.cs.uku.fi/~kilpelai/BSA05/

两个课程的主要参考书都是Dan Gusfield的“Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology”，其中前者还推荐Pavel A. Pevzner的“Computational Molecular Biology – An Algorithmic Approach”。

下面是一个不错的介绍suffix tree的页面：

http://homepage.usask.ca/~ctl271/857/suffix_tree.shtml

由于suffix tree，尤其是Ukkonen算法在生物信息学中的广泛应用，读者可以直接在搜索引擎中查找“Ukkonen's Algorithm”以获得更多的资源。

本节的算法和程序列表如下：

算法	程序	备注
字符串模式匹配经典算法	smatch.cpp	朴素模式匹配，Rabin-Karp算法，KMP算法和优化的KMP算法。
Z算法和BM算法	bm.cpp	Z算法和Boyer-Boore算法。
Aho-Corasick算法	ac.cpp	集合匹配的Aho-Corasick算法。
Ukkonen算法	ukkonen.cpp	后缀树构造的Ukkonen算法
后缀数组相关算法	suffixarray.cpp	分治构造法、Skew算法和高度数组计算、常数时间LCP询问。
扩展KMP算法	ext.cpp	扩展KMP算法。
最长回文子串问题	palin.cpp	分治法、基于后缀树的算法和基于后缀数组的算法。

第7章　图论问题和算法

本章介绍图论问题和算法，包括图结构方面的问题和网络优化。

7.1　图的结构

本节介绍图论的基本问题，它们是学习网络优化的基础。这些内容包括：图的遍历与边分类、拓扑排序与欧拉回路、有向图和无向图的连通性和特殊图类的判定与特殊算法。

7.1.1　图的遍历及其应用

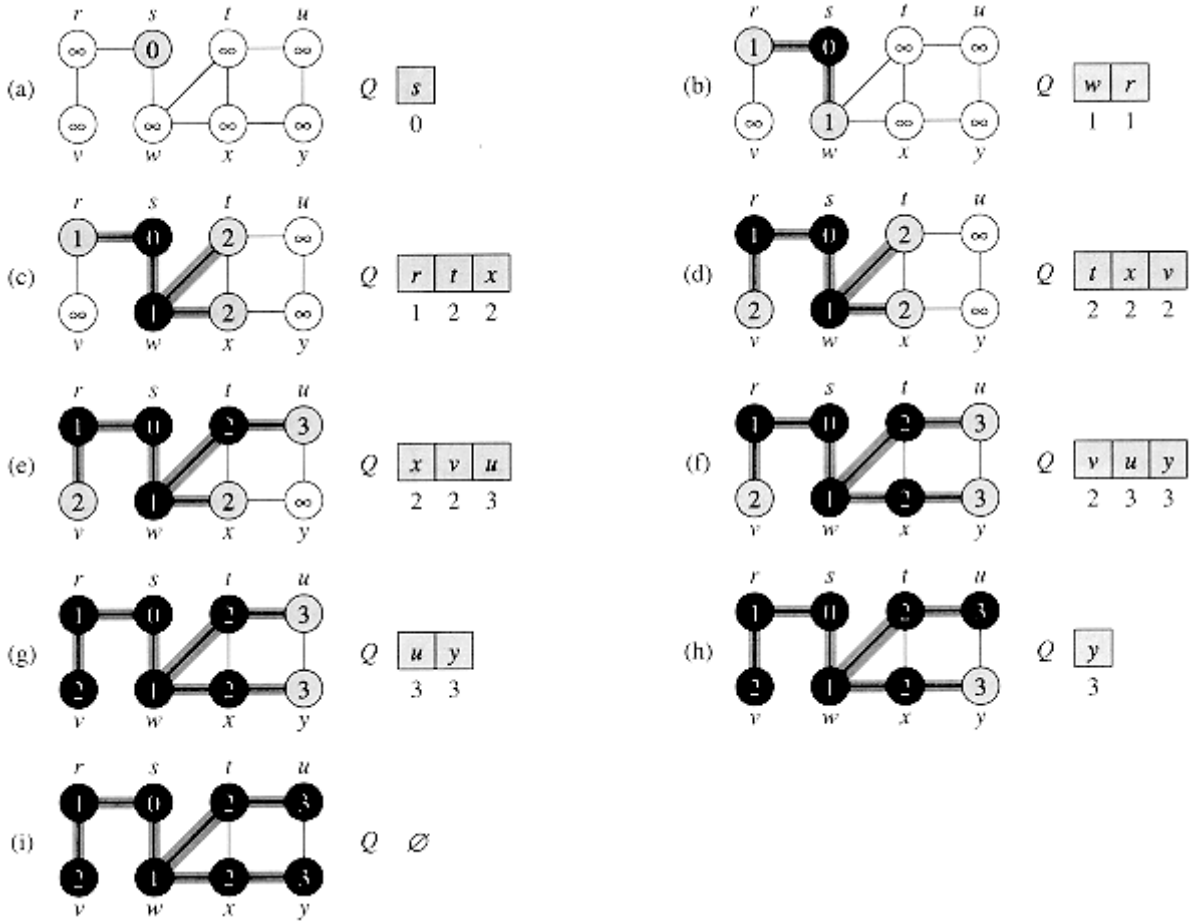
图的很多问题以遍历为辅助算法。虽然宽度优先遍历和深度优先遍历已经介绍和应用过，但在图论中，以它们为辅助算法可以扩展出很多算法。

BFS算法 宽度优先遍历由Moore和Lee独立提出，它的思想是按照从近到远的顺序遍历各个点。算法顺便可以求出从s到各点的距离。为了叙述方便，规定算法过程将给结点着色：白色为没有考虑过的点，黑色为已经完全考虑过的点，灰色为发现过但没有处理过的点。灰色点组成了遍历边界，好比在第7章中介绍的搜索边界一样。

BFS算法首先把一个给定点s着成灰色，用队列保留所有灰色结点。每次从中取出一个点u，并对于它的所有关联边(u, v)，把其中白色点v着成灰色并加入到BFS树中。在BFS树中u是v的父亲或称前驱(predecessor)，到BFS根的距离d[v] = d[u]+1。下面是一个典型的BFS遍历过程，BFS树用黑色标出。

利用d[i]的信息可以求出任意点i到s的所有最短路。最短路长度为d[i]，所以只要每走一步到达的目标点的d值减1，一定可以扩展成为一条最短路。虽然最短路可能有指数多条，但这样的方法时间复杂度和输出量成正比，已经是理论下界了。

二分图判定 利用BFS可以判断一个图是否为二分图，即是否可以给所有点分成X和Y两个集合，使得每条边的两个端点一定处于不同集合。起点s显然可以随意



定色，每次考虑一条边(u, v)时显然u所在集合已定，因此当v为白色时把它放到不包含u的那个集合，而当v为灰色或黑色时（此时v所在集合已经确定）检查v和u是否在同一个集合。如果是，则该图不是二分图，失败退出。如果是否没有失败，则算法实际上已经构造出了这两个集合。在忽略s所在集合的情况下，这个集合分划是唯一的（算法步骤中的每一步都是强制的）。

DFS遍历 DFS遍历优先扩展新发现的结点，它的过程可以看作是递归的。算法将得到DFS森林，类似于BFS树。DFS的特别之处在于可以进行边分类。我们首先需要在遍历时加上**时间戳(time stamp)** 。

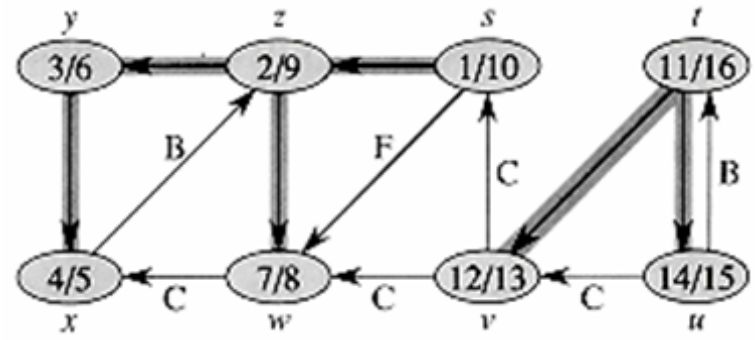
发现时间 $d[u]$ 结点变灰的时间
结束时间 $f[u]$ 结点变黑的时间

初始化time为0，所有点为白色，DFS森林为空。则只需要对每个白色点u执行一

次DFS-VISIT(u)，每处理一个点前和处理完后都把当前时间加1，则所有点u的时间戳满足1¡=d[u]¡f[u]¡=2|V|。显然算法的时间复杂度为O(n+m)。

```
DFS-VISIT(u)
1  color[u] ← GRAY          ▷ White vertex u has just been discovered.
2  d[u] ← time ← time + 1
3  for each v ∈ Adj[u]      ▷ Explore edge (u, v).
4       do if color[v] = WHITE
5             then π[v] ← u
6                  DFS-VISIT(v)
7  color[u] ← BLACK         ▷ Blacken u; it is finished.
8  f[u] ← time ← time + 1
```

这里有一个例子：



所有时间戳都不同，且为1到2|V|的整数。从下图还可以直观看到，对于任意结点对(u, v)，区间[d[u], f[u]]和[d[v], f[v]]要么完全分离，要么相互包含。如果u的区间完全包含在v的区间内，则在DFS树中u是v的后代。

关于DFS树，我们有重要的：

白色路径定理 在DFS森林中v是u的后代当且仅当在u刚刚被发现时，v可以由u出发只经过白色结点到达。定理十分直观，证明略。

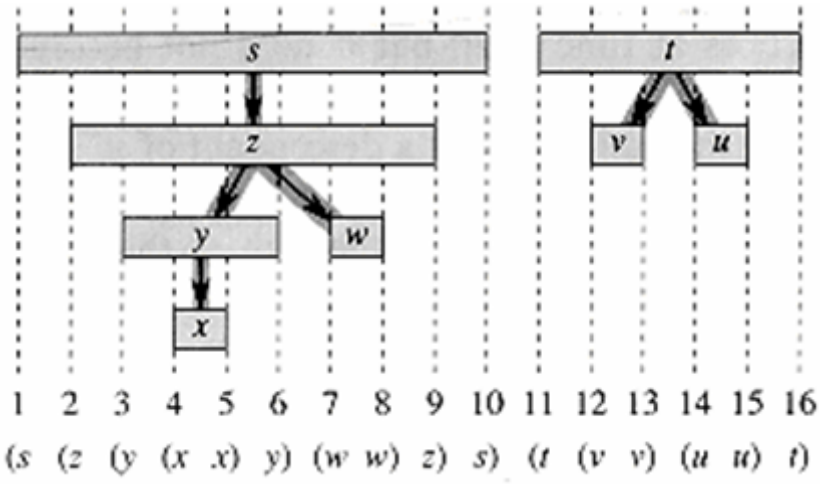
DFS-VISIT可以顺便将边(u, v)分成以下几类：

树边(Tree edges, T) v通过边(u, v)发现。

后向边(Back edges, B) u是v的后代。

前向边(Forward Edges, F) v是u的后代。

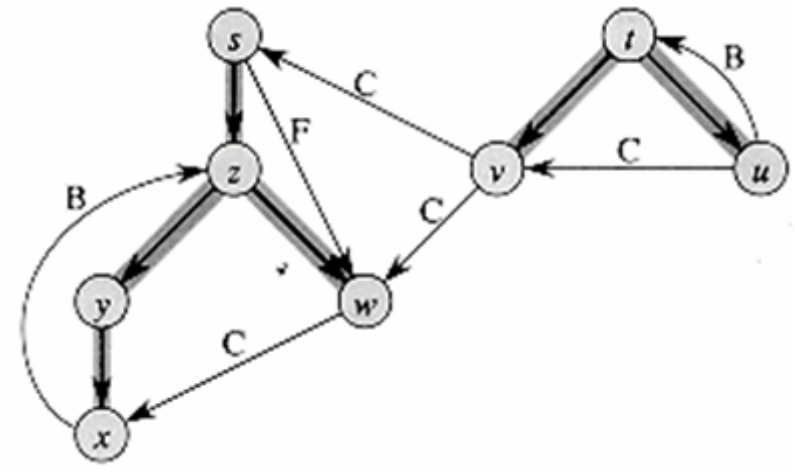
交叉边(Cross Edges, C)：其他边。可以连接同一个DFS树中没有后代关系的两个结点，也可以连接不同DFS树中的结点。



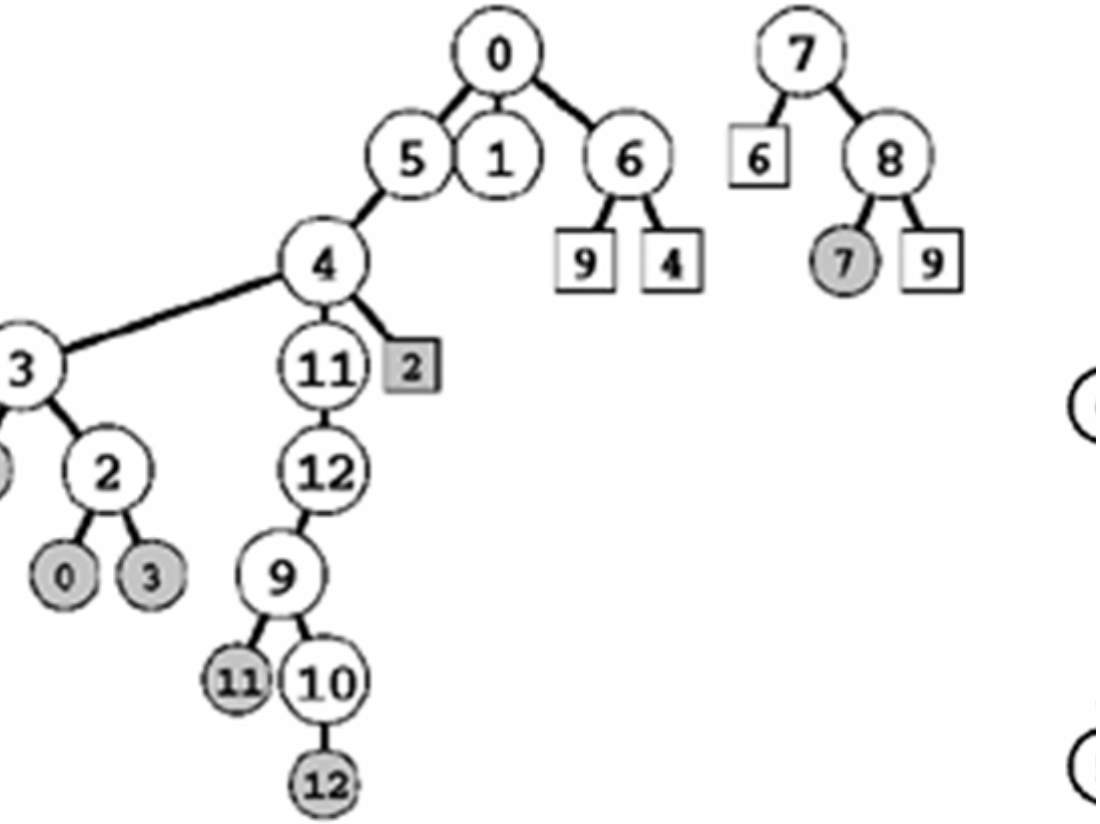
边分类算法 把分类规则落实到程序实现中，只需在考虑边(u, v)时检查v的颜色：

v是白色，(u, v)是T边
v是灰色，(u, v)是B边，因此只有此时u的祖先是灰色
v是黑色，继续判断。若d[u]¡d[v]，说明v是u的后代，因此是F边，否则为C边。

DFS的时间复杂度仍为O(n+m)。容易证明：无向图只有T边和B边。下面的图把前面的例子中的DFS树画得更加直观，读者可以仔细体会T、B、F、C边是什么样的



遍历编号 有的书在叙述DFS时不使用时间戳而用遍历编号pre[u]和post[u]，它相当于把时间戳分成两个，给发现事件和结束事件以单独的时间编号。



	0	1	2	3	4	5	6	7	8	9	10	11	12
e	0	9	4	3	2	1	10	11	12	7	8	5	6
t	10	8	0	1	6	7	9	12	11	3	2	5	4

这样，边分类算法应改为：pre[v]为-1时为T边，否则当post[v]为-1时为B边，否则当pre[v]¿pre[u]是为F边，否则为C边。初始时所有pre和post值均为-1。

拓扑排序。用有向边表示偏序关系，则一个有向图可以理解为n个元素的偏序关系图。拓扑排序的任务就是根据偏序关系整理出全序关系。一个有向图有拓扑排序当且仅当它不含有向环。这样的图称为**有向无环图(Directed Acyclic Graph, DAG)** 。

定理：一个有向图是DAG当且仅当DFS时没有发现B边。

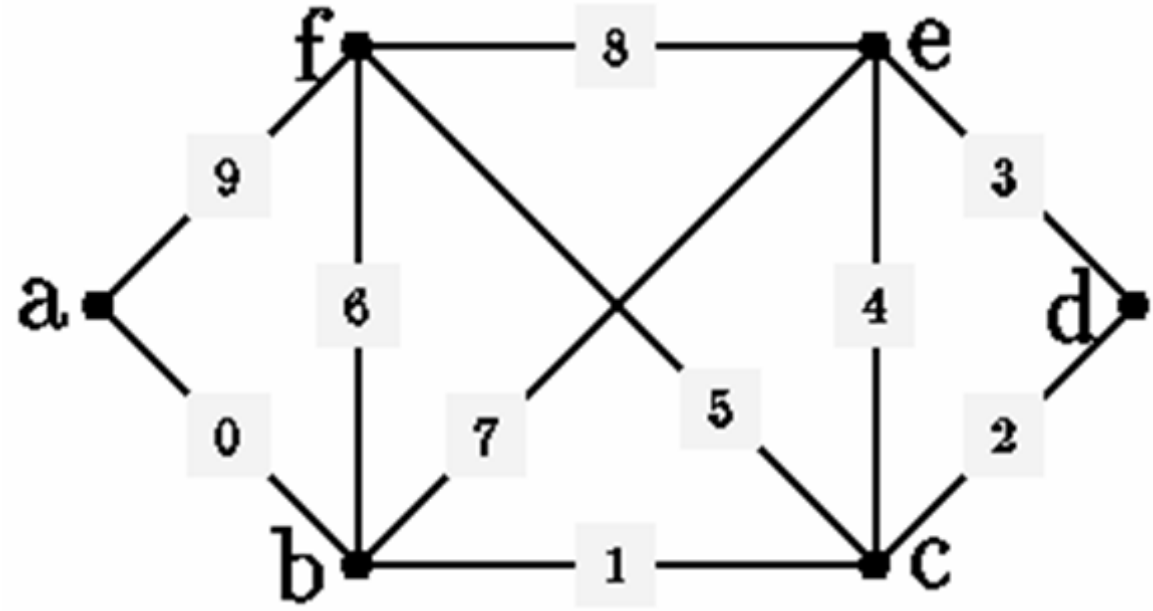
证明：如果存在B边(u, v)，则从v到u再回到v就是一个有向环；如果有环C，考虑其中第一个被发现的结点v，环中v的上一个结点为u，则沿环的路径v-¿u是白色路径。由白色路径定理，u是v的后代，因此(u, v)是B边。

用DFS算法每次把一个结点变黑时把它加到栈中，则各结点出栈的顺序是一个正确

的拓扑顺序。这样，我们在O(n+m)时间内求出一个拓扑排序。需要说明的是：这个方法并不能求出所有拓扑顺序（不管按什么顺序考虑结点，都有可能漏掉）。有没有办法求出所有拓扑排序呢？我们需要跳出DFS这个框框，寻找其他解决方案。

第一个结点是什么？它应该是一个入度为0的点，而且没个入度为0的点都可以是拓扑序下的第一个点。把它删除后剩下的图中入度为0的点都可以是下一个点。这样，只需要维护每个点的当前入度，每删除一个点时修改它所有后继点的入度，并把其中入度为0的点加入到队列中。在有多个入度为0的点时只要选择合适的顺序，每种拓扑顺序都可以取到。

欧拉回路 每条边经过一次且仅一次的回路称为图的**欧拉回路(euler cycle, euler circuit)** 。



一个无向图有欧拉回路的必要条件是：它是连通的。在连通的情况下，存在欧拉回路的充要条件是每个点的度数均为偶数。下面的算法利用DFS求出无向图的一条欧拉回路。

其中inv[i]为边i的反向边，因为标记(u, v)时也需要标记(v, u)。注意这个过程和前面所讲的DFS不一样，因为每个结点可以被访问多次。算法的正确性可以用数学归纳法说明：Euler-Route把当前图中start所在连通分量（如果是有向图，就是底图的连通分量）的欧拉回路压入栈中，因为在标记一条边(u, v)为“已访问”并从v开始DFS时，递归边界必然回到u（因为只有u的度数为奇数），因此递归调用往栈里放入了一

```
void Euler_Route(int u)
{
for(i = st[u]; i ¡ st[u+1]; i++) if(!mk[i]){
mk[i] = mk[inv[i]] = 1;
Euler_Route(v[i]);
push(i);
}
}
```

条$u \to v \to u$的回路，把所有回路并起来就得到了欧拉回路。

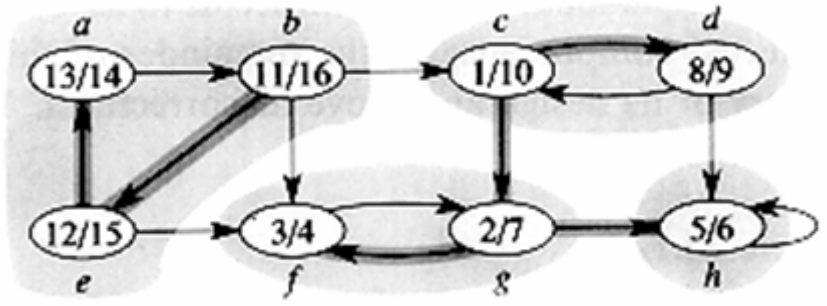
对于有向图来说，必要条件改为每个点的出度等于入度，而程序里只需要标记一条边即可，不用标记inv[i]。

小测验

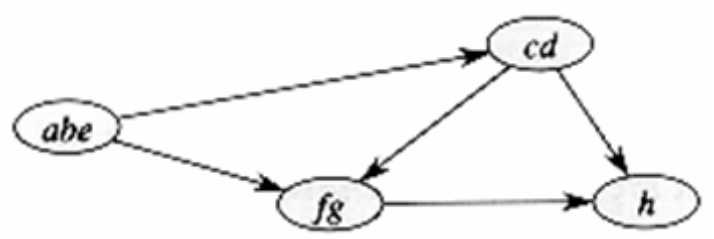
1. 叙述BFS算法并用它做二分图判定
2. 叙述DFS算法、嵌套区间定理、白色路径定理。分别用时间戳和遍历编号说明如何给有向图的边进行分类。无向图只有哪些类型的边？
3. 如何用DFS进行拓扑排序？如何用DFS找欧拉回路？

7.1.2 有向图的连通性

本节讨论有向图的连通性。有向图的边都是单向的，因此可达性不具有传递性：u可以到达v，不见得v可以到达u。但如果u和v相互可达，那么对于任意其他结点w来说，w、u之间的可达性与w、v之间的可达性相同。“相互可达”关系是一个等价关系，因此可以根据这个关系把所有结点分成若干集合，同一个集合内的点相互可达，不同集合内的点不相互可达，如下图。



每一个集合称为有向图的一个**强连通分量(Strongly Connected Componenet, SCC)** 。如果把一个集合看成一个点，那么所有SCC构成了一个SCC图：



从abc到cd有一条有向边，意味着从a可达c，但c不可达a。这个图中不可能有环（想一想，为什么），因此它是一个DAG。

如何求出有向图的各个强连通分量呢？可以通过DFS。以上图为例，如果从h开始DFS，将得到只包含h的一棵DFS树。下面再从f出发，得到了以fg为结点的DFS树。再从c出发，将得到cd为结点的DFS树，最后从a出发，将得到以abc为结点的DFS树。这样，每次DFS得到的DFS树恰好对应一个SCC。可问题在于我们并不知道应该先从h开始DFS。如果一开始就不幸选择从a进行遍历，由白色路径定理，这棵DFS树将包含整个图！也就是说，这次不明智的DFS把所有SCC混在了一起，什么也没得到。

很明显，我们希望按SCC图拓扑顺序的逆序进行遍历。这样才能这次DFS得到一个SCC，而不会把两个或多个SCC混在一起。

Kosaraju算法 也许最容易理解的一个算法是Kosaraju于80年代提出的，它执行两次DFS。第一次DFS得到了关于各个SCC拓扑顺序的有关信息，而第二次DFS按照这个拓扑顺序的逆序进行DFS，从而把每个SCC分开。算法步骤如下：

第一步 调用DFS(G)，计算出每个结点的f[u]或者post[u]

第二步 计算G的转置G^T（即把所有有向边(u, v)变为有向边(v, u)）

第三步 调用DFS(G^T)，在主循环中按照f[u]或者post[u]递减的顺序执行DFS-VISIT，则得到的每个DFS树恰好对于一个SCC。

算法的时间复杂度为O(n+m)，但注意并不是在图的所有表示法中都很容易求出转置。为什么算法是正确的呢？我们首先证明以下定理：

定理： 设d(U)和f(U)表示集合U所有元素的最早发现时间和最晚完成时间，则对于G中的两个SCC C和C'，如果C到C'有边，则f(C)¿f(C')。

证明分两种情况。

情况一 d(C)¡d(C')。考虑C中第一个被发现的点x，则x被发现时C'全为白色。而C到C'有边，故x到C'中的每个点都有白色路径。这样，C的其他点和C'的所有点都

是x的后代，因此f(C) ¿ f(C')。

情况二 d(C)¿d(C')。由于从C'不可到达C，所以必须等C'全部访问完毕后才能访问C，因此f(C) ¿ f(C')。

作为定理的推论，如果G^T中C到C'有边，则f(C)¡f(C')。由于G和G^T的强连通分量完全一致，所以只需要在G^T中按照f值递减的顺序执行DFS-VISIT即可。

可能有读者会认为不需要转置而直接在G中按照f递增的顺序执行DFS-VISIT也能奏效，但可惜这是错误的。对于G中的两个SCC C和C'，如果C到C'有边，刚才的定理说的是f(C)¿f(C')，其中二者取的是C和C'中f值的最大值，而并没有说C和C'中最小的f值哪个大。如果按照f递增的顺序遍历，我们无法知道究竟会先遇到C中的点还是C'中的点。如果在G^T上操作，由推论知f(C)¡f(C')，所以当按f值降序考虑时一定会先遇到C'中的点，从而正确的把C和C'分开。

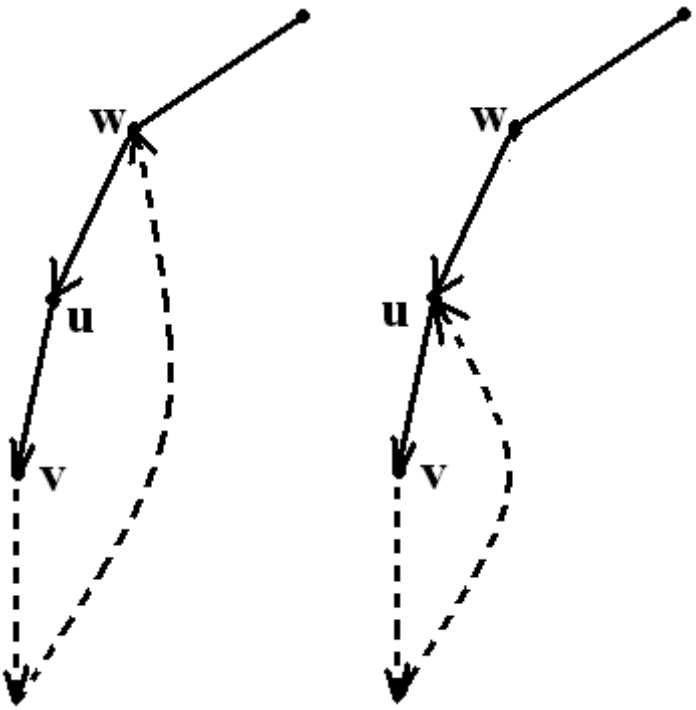
Tarjan算法。这是SCC问题的第一个算法，由Tarjan于1972年提出。算法仍然借助DFS，但它并不是靠遍历顺序来把不同SCC分离到不同的DFS树中，而是允许多个SCC并存于同一棵DFS树中，用某种手段把它们分开。考虑SCC C，设其中第一个被发现的点为x，则由白色路径定理，C中其他点都是x的后代。我们希望在x访问完成时立刻输出C。这样，就可以在同一棵DFS树中区分开所有SCC了。因此问题的关键是：判断一个点是否为一个SCC中最先被发现的点。

如上图。假设我们正在判断u是否为某SCC的第一个被发现结点。如果我们发现从u的儿子出发可以到达u的祖先w，显然u、v、w在同一个SCC中，因此u不是该SCC中第一个被发现的结点。如果从v出现最多只能到u，那么u是该SCC中第一个被发现的结点。这样，问题转化为求：一个点u最远能到达的祖先的d值。注意，这里的“到达”可以通过B边，也可以通过C边，但是前提是只能通过栈里有的点而不是已经确定SCC编号的其他点。图中实线表示一条边，虚线表示一条或多条边。

定义lowlink[u]为u及其后代能追溯到的最早（最先被发现）祖先点v的pre[v]值，则可以在计算lowlink函数的同时完成SCC计算。lowlink函数可以通过递推得到，方法如下：

程序用S栈保存当前SCC中的结点（注意这些结点形成一棵子树而不一定是一个链），v为前向星数组，cnt为当前pre编号，而scnt为SCC计数器，id[i]为i所在的SCC编号。

原始的Tarjan算法的递推方法是：如果pre[w]¡pre[u]且w在栈中，则low[u] = min{pre[w], low[u]}，注意限制w必须在栈中是因为从C边出发可能到达已经输出的SCC中。这里




```
min = low[u] = pre[u] = cnt++;
push(S, u);
for(i = st[u]; i ¡ st[u+1]; i++){
w = v[i];
if(pre[w] == -1) dfs_visit(w);
if(low[w] ¡ min) min = low[w];
}
if(min ¡ low[u]){ low[u] = min; return; }
do{ id[w = pop(S)] = scnt; low[w] = n; } while(w != u);
scnt++;
```

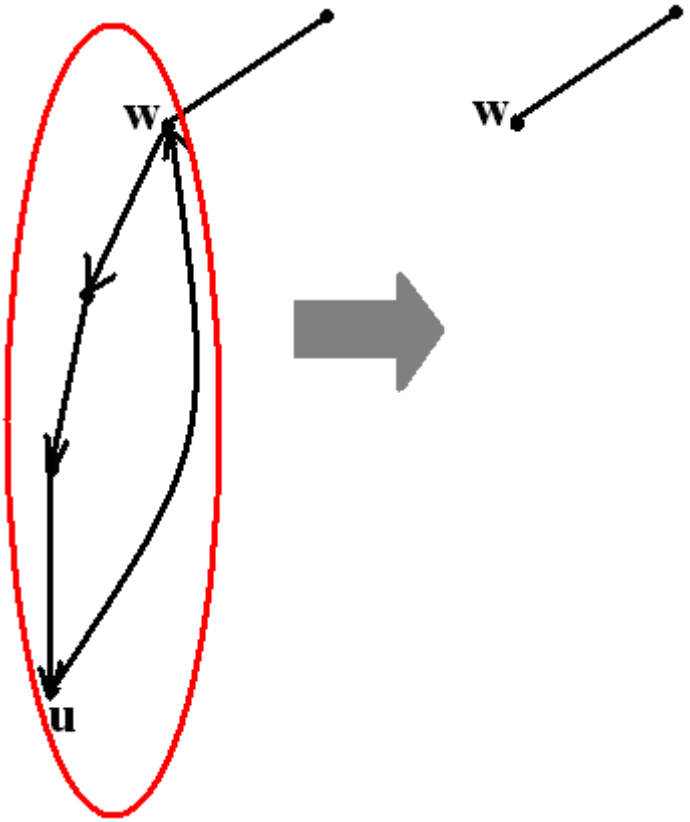
给出的版本更容易理解：只需在输出SCC后设置low值均为最大值n，这样在考虑后面的SCC时就不会通过C 边跑到这个已经结束的SCC中了。

Gabow算法 Gabow算法和Tarjan算法很类似，仍然使用栈S在同一棵DFS树中区分不同的SCC，但Gabow不是用lowlink函数来判断每个SCC的第一个被发现结点，而是使用另外一个栈P保留当前路径中的结点。P上的结点形成一条链，此链上的结点都属于不同的SCC，而原来的栈S作用不变，因此S内的点和P的栈顶在同一个SCC中。当发现反向边(u, w)后P不断出栈，只保留w。程序如下：

如果id[w]不是-1，说明w的SCC已输出，这是一条C边。否则(u, w)是B边或者F边。F边

```
pre[u] = cnt++;
push(S, u); push(P, u);
for(i = st[u]; i ¡ st[u+1]; i++){
w = v[i];
if(pre[w] == -1) dfs_visit(w);
else if(id[w] == -1) while (pre[top(P)] ¿ pre[w]) pop(P);
}
if(top(P) == u) pop(P); else return;
do{ id[w = pop(S)] = scnt; } while(w != u);
scnt++;
```

自动不会执行循环，而B边将会导致把P内的结点一直弹出，直到栈顶为pre[w]为止。这样才能确保P中的结点都属于不同SCC。这个过程可以由下图说明。



如果访问完结点u后u在P的栈顶，说明u是当前SCC的第一个被发现顶点，栈S内的结点就是该SCC内所有结点。

有了SCC，很容易求出有向图的传递必包、点基和汇基等。关于有向图的连通性，DAG的**支配点(dominator)** 问题也是一个基本问题，2-SAT也可以借助于SCC在

线性时间内得到解决。限于篇幅，这里不再叙述。

小测验

1. 什么是SCC？什么是SCC图？为什么它是DAG？
2. 叙述Kosaraju算法并证明其正确性。
3. 比较Tarjan算法和Gabow算法的相同点。它们在处理哪一个关键问题时采取了不同的方法？

7.1.3 无向图的连通性

求无向图的连通分量是平凡的，但无向图的连通性问题远不止于此。如果连通无向图G中存在一个点u，删除u后G不再连通，则称u为G的一个**割顶(articulation point)** 。没有割顶的连通图称为**双连通图(biconnected graph)** 。对于任意两条边e_1和e_2，如果e_1=e_2或者它们在同一个环中，则称它们满足关系R。容易验证R是一个等价关系。根据此等价关系我们可以把G的边分为不相交集E_1, E_2, ..., E_k，设V_i为E_i中包含的点集，则每个子图G_i=(V_i, E_i)称为G的一个**双连通分量(biconnected component, BCC)** ，或称**块(block**)。双连通分量具有如下性质：

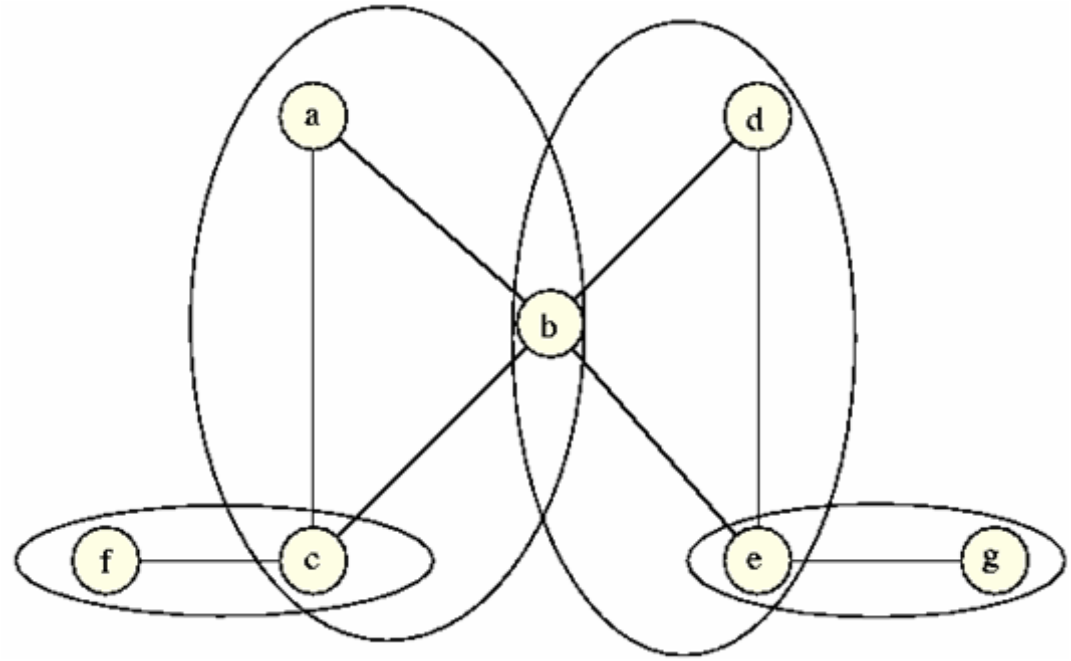
1. 双连通分量都是双连通的。
2. 任意两个不同的双连通分量最多只有一个公共点
3. 结点u是图G的割顶当且仅当u是某两个不同的双连通分量的公共点。

下图是一个图和它的双连通分量。

注意到在无向图中边只有B边和T边两种，因此DFS可以很方便的求出G的所有双连通分量。首先我们需要判断出所有割顶。关于割顶，我们有以下引理：

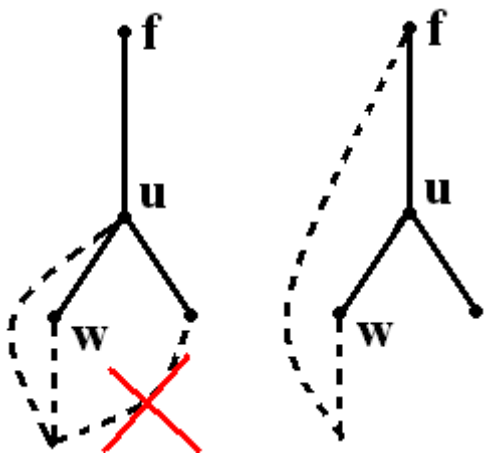
引理：对于连通无向图G=(V, E)，S=(V, T)为G的一个DFS树，则结点u是G的割顶当且仅当下面条件之一被满足：

1. u是T的根且u至少有两个儿子
2. u不是T的根且存在u的某个儿子w，使得从w或者w的后代没有边连回u的祖先（注意，不是连回u本身）。



证明：首先考虑u是根的情况。如果它有两个儿子v和w，其中v先被发现。由于w并不是v的后代，所以v到w没有白色路径。由于v和w是连通的，而且此时只有u被标记为灰，所以v到w的所有点都必须经过u。删除u后v和w不再连通。反过来，如果u只有一个儿子，删除以后显然剩下点在T-{u}中连通，当然在G-{u}中也连通了。

如果u不是根，情况如下图。如果w或者w的后代没有连回u祖先的边，那么w到u父亲f的所有路必须经过u。注意无向图是没有F边和C边的，所以w及其后代只能靠B边连回祖先。



反过来，如果u的每个儿子都可以连回f（如果可以连得更远，顺着树边走最终可

以到f），删除u至少不会引起u的这些儿子和f不连通。有没有可能引起另两个点不连通呢？假设存在x和y使得删除u后引起x和y不连通。显然x和y至少有一个应是u的后代，否则不考虑以u为根的子树x和y也可以通过T的剩下部分连通。这样，设x是u的后代。注意我们假设的是x可以连回f，因此图看起来应该是下面两种情况之一：

情况一 y不是u的后代，这样x先走到f，再沿着T中的边走到y，并没有经过u。

情况二 y也是u的后代，则y也可连回f。这样x先走到f，再从f到y，并没有经过u。

在两种情况下，删除u都不会引起x和y不连通。

LOW函数及其计算 刚才我们遇到了和SCC的Tarjan算法类似的问题：需要计算每个点u及其后代所能回到的最远祖先LOW[u]。和有向图的LOWLINK函数类似，可以在DFS的同时顺便递推每个点的LOW函数：

```
low[u] = pre[u] = cnt++;
for(i = st[u]; i ¡ st[u+1]; i++){
w = v[i]; if(w == u) continue; //不考虑自环
if(prev[w] == -1){
father[w] = u;
dis_visit(w);
if(low[u] ¿ low[w]) low[u] = low[w];
}else if(w != father[u])
if(low[u] ¿ pre[w]) low[u] = prev[w];
}
```

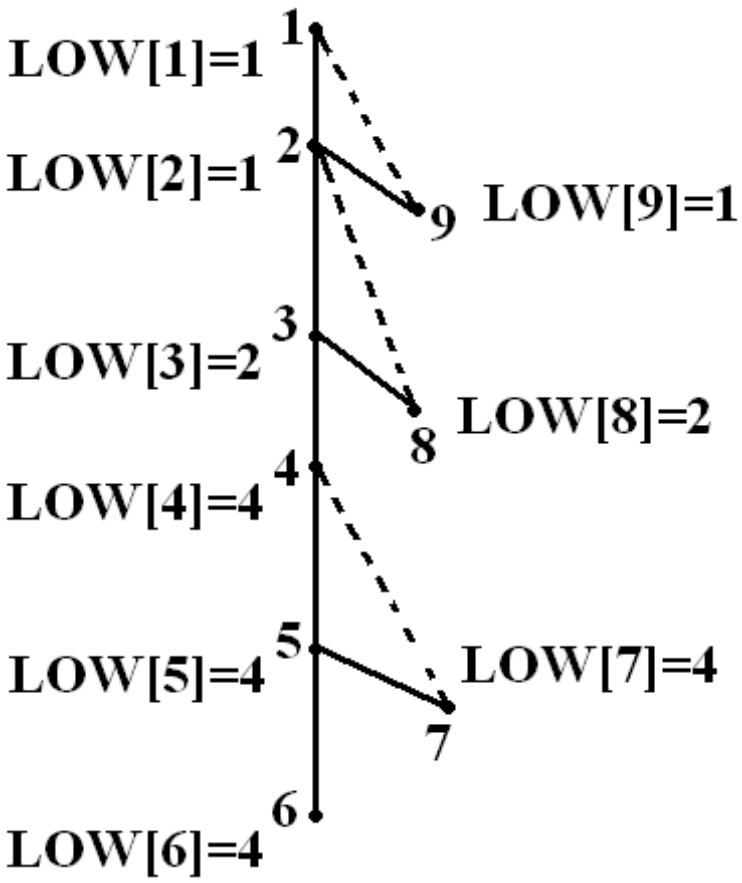
注意必须判断w不是father[u]。如果w=father(u)，说明(w, u)是T边，就不能认为(u, w)是B边了。下面是一个LOW函数计算的例子，虚线为B边，实线为T边。无向图没有F边和C边。

BCC的计算。和SCC的Tarjan算法类似（事实上，这个算法也是Tarjan提出的），我们可以使用栈S来保留在当前BCC中的边（注意不是点而是边）。注意每遇到一条边(u, w)都要加到栈中，不管w是否已编号。比如在上图中在访问7时应把(7, 4)加入栈中，虽然4是灰色的。

和有向图不一样的是：在无向图中一条边会被考虑两次，所以在遇到边(u, w)时需要检查它是否已经在S中。

情况一 如果pre[u]¿pre[w]，边(u, w)已经以(w, u)的形式存在于S中当且仅当w=father[u]

情况二 如果pre[u]¡pre[w]，边(u, w)已经以(w, u)的形式存在于S中当且仅当pre[w]!=-1。



其中情况一是容易理解的，如上图中访问结点2时考虑边(2, 1)。情况二对应于访问结点1时考虑边(1, 9)。由于这条边在访问点9时已经加入到S中，因此这时忽略边(1, 9)。

在LOW函数计算的程序中只需要在递归调用dfs_visit(w)中加一行：如果low[w]¿=pre[u]，则u是割顶或者根。此时从S中不断的弹出边，直到(u, w)被弹出。这些边构成了G的一个BCC。在刚才的例子中，访问完结点5后发现LOW[5]=4，因此4是一个割顶。此时栈S从底到顶各条边依次是：(1, 2), (2, 3), (3, 4), **(4, 5), (5, 6), (6, 4), (5, 7), (7, 4)**。这时需要从栈中弹出加黑部分，构成一个BCC。注意到我们并没有特别处理根，因为不管它是不是割顶都代表着BCC的终止。

割顶和桥 下面我们来显式统计出所有割顶和桥。对于割顶来说，只要在BCC计算中附加判断u为根时度数是否为1。注意最好给每个点做一个割顶而不是直接输出，因为同一个点可能输出多次（因为多个BCC可以包含同一个割顶）。类似于割顶，我们可以定义无向连通图的**桥(bridge)**：如果删除一条边e后无向图G不再连通，称e为G的

桥。

桥的判定也不难，只需要在发现T边(u, v)时进行判断。如果v后它的后代无法连回u或者u的祖先，则删除(u, v)后u和v不连通。即：发现T边(u, v)并递归遍历v后若LOW[v]=pre[v]，则(u, v)为桥。

类似于割顶，我们称没有桥的图为边连通图。如果一个无向图是边连通的，可以把它的边定向，得到一个强连通的有向图。算法留给读者思考。

点连通度和边连通度 把割顶和桥的定义加以推广，可以定义：

k-连通(k-connected)：对于任意一对结点都至少存在结点各不相同的k条路。

点连通度(vertex connectivity)：把图变成非连通图所需删除的最少点数。

这两个定义是互通的，因为我们有：

Whitney定理：一个图是k-连通的当且仅当它的点连通度至少为k。

这样，双连通图是2-连通的，而任意连通图都是1-连通的。类似的，可以定义：

k-边连通(k-edge connected)：对于任意一对结点都至少存在边各不相同的k条路。

边连通度(edge connectivity)：把图变成非连通图所需删除的最少边数。

类似的，我们可以证明一个图是k-边连通的当且仅当它的点连通度至少为k。计算一个无向图的连通度和边连通度都有多项式算法，但它们大都用到了网络流，我们将稍后重新讨论它。

小测验

1. 什么是割顶？什么是桥？什么是双连通图？什么是无向图的双连通分量？什么是点连通度和边连通度？
2. 如何计算LOW函数？它的计算过程和有向图的LOWLINK计算有哪些相同点和不同点？试从交叉边和边的双向性加以说明
3. 叙述割顶和桥的条件和计算BCC的Tarjan算法。栈S里保存的内容和加入方式和SCC的Tarjan算法有什么不同？

7.2 生成树和树型图

若存在一个无向图G=(V, E)，而对于图中的每一条边(u, v)∈E，都有一个权值w(u, v)表示连接u和v需要的代价。我们需要选则一些最小代价和的边，连接了所有的结点。由于要求边的代价和最小，因此在保证连通的前提下选出来的最优边集一定是一棵树（想一想，为什么）。基于此，我们称任何连接了V的所有点的E的无回路子集T为G的**生成树**。而若要求权和w(T)= $\sum_{(u,v)\in T} w(u,v)$最小，则是G的一棵**最小生成树(minimum spanning tree, MST)**。

本节所要研究的问题都与图的生成树有关。首先我们将介绍解决最小生成树问题的两种经典算法：Prim算法和Kruskal算法。其次我们则研究该问题的几个变种：最小度限制生成树和次小生成树等。若将无向图的生成树问题扩展至有向图，则被称为有向图的**树形图(arborescence)**或**有向生成树(directed spanning tree)**，如何判定一个图中是否存在树形图以及如何构造出其中权值和最小的一个，我们将在本节的最末予以研究。

7.2.1 求最小生成树的Kruskal和Prim算法

一般最小生成树算法。已知无向加权连通图G=(V, E)，如何得到G的最小生成树呢？本节介绍的两种算法都运用了贪心法，其贪心的细节有所不同，但都使用了下述的“一般性” 方向：首先令集合A为空，并保证算法运行*自始至终A都是某个最小生成树的子集*，那么每次我们都找一条新的边(u, v)加入A中，且不会破坏上述的性质，则可称这样的边为A的**安全边(safe edge)**。而这一过程可以写成下述的代码形式。

```
GENERIC-MST(G, w)
1  A ← ∅
2  while A does not form a spanning tree
3      do find an edge (u, v) that is safe for A
4         A ← A ∪ {(u, v)}
5  return A
```

执行这一过程，则返回的集合就是我们需要的最小生成树T，而如何寻找安全边，就是两个算法的差异所在了。

Kruskal算法。

引理：权值最小的边一定是空集的安全边。

证明：不妨使用调整法，设w(u, v)的值最小，且任选一颗最小生成树，若其不包括这样的一条边，则u和v一定被一条路径相连：

$$\mathrm{u},\ \mathrm{a}_1,\ \mathrm{a}_2,\ \ldots,\ \mathrm{a}_m,\ \mathrm{v}$$

由于w(u, v)的权值最小，在这棵树中加上边(u, v)且删去任何一条边如(u, a_1)，代价不会更大，而此时存在这样一条链：

$$\mathrm{a}_1,\ \mathrm{a}_2,\ \ldots,\ \mathrm{a}_m,\ \mathrm{v},\ \mathrm{u}$$

不难看出连通性依然满足，因此这一定是一颗包含(u, v)边的最小生成树，即至少有一颗最小生成树包括(u, v)，命题得证。

定理：连接A中不同连通分量的权值最小的边一定是A的安全边。

证明：这里我们将运用缩点技术，将这一命题划归为上述引理。可以构造一个新的图G'，在G'中，A中每一个连通分量都被收缩为一个顶点，G中连接不同连通分量的边在E'中保留，舍去连接相同连通分量的边。那么设新图G'(V', E')最小生成树T'，则T'∪A就是G的最小生成树T。

设E'中权最小的边为(u, v)，由引理可知，一定存在一个T'包含{(u, v)}，则有

$$\mathrm{A}\cup\{(\mathrm{u},\ \mathrm{v})\} \subseteq \mathrm{T'}\cup\mathrm{A} = \mathrm{T}$$

即(u, v)是A的安全边，命题得证。

直接利用这个定理就可以得到求最小生成树的Kruskal算法：每次选择连接不同连通分量的权值最小的边作为安全边加入集合中即可。在具体实现上，算法需要使用并查集来实时维护图中的连通分量。

不难看出Kruskal算法的运行时间为O(ElogE)。下面是一个例子，它给出了算法执行的前六步。

Prim算法。和Kruskal算法不同，Prim算法采用了另一种确定安全边的标准。

引理。若u∈V，且(u,v)是与u相连的权最小的边，则(u,v)是空集的安全边。

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3       do MAKE-SET (v)
4  sort the edges of E by nondecreasing weight w
5  for each edge (u, v) ∈ E, in order by nondecreasing weight
6       do if FIND-SET(u) ≠ FIND-SET(v)
7             then A ← A ∪ {(u, v)}
8                  UNION (u, v)
9  return A
```

和Kruskal算法的证明类似，使用调整法即可。

由此，我们可以任意选择一个点r∈V定为生成树的根，根据上述引理，选择一条权最小的(r,u)作为安全边放入生成树中；r和u在一个连通分量内，同样使用缩点技术，将它们看作一个结点，重复这一过程|V|-1次，就得到了最小生成树。

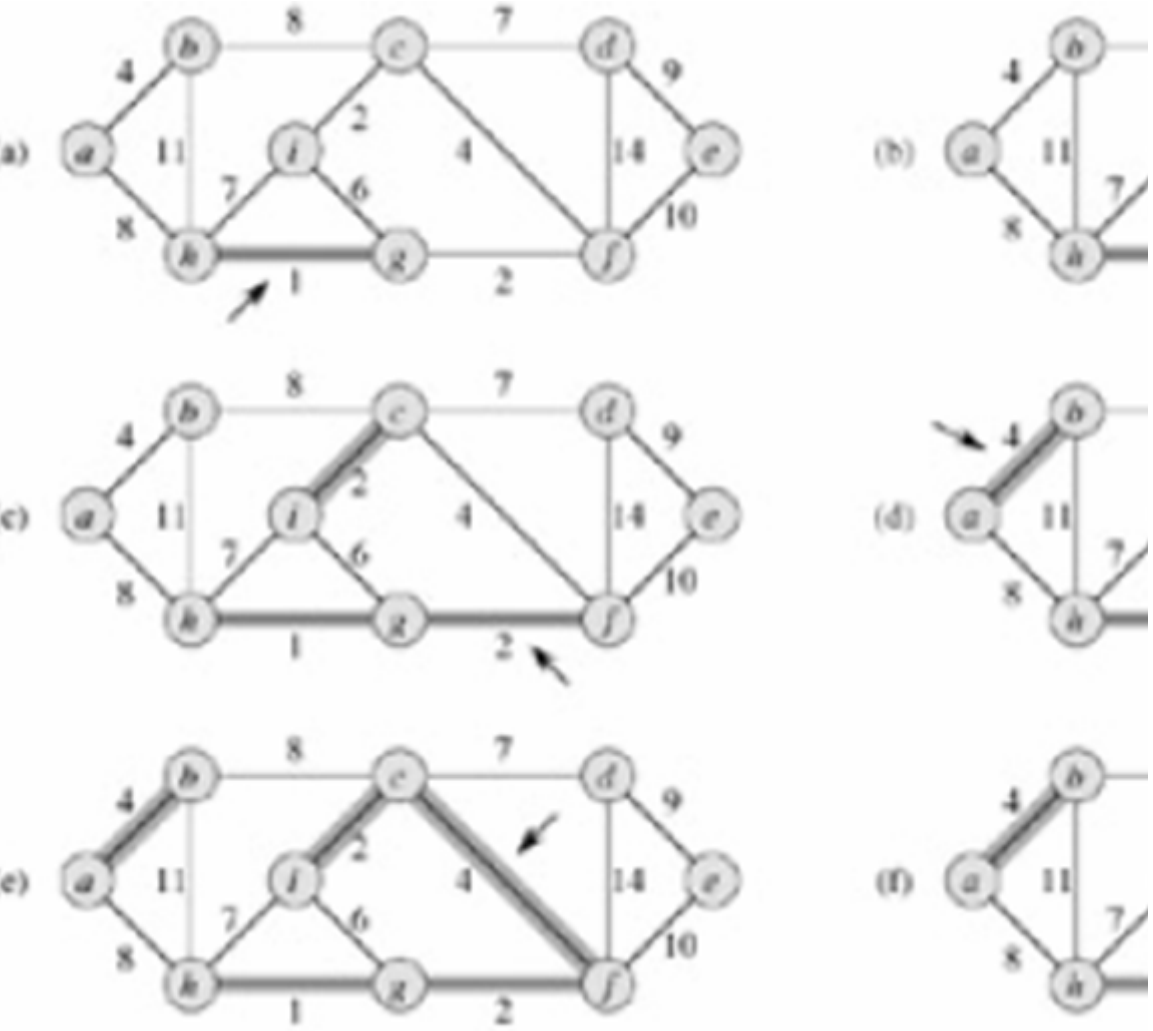
如上图所示，黑色圆圈表示算法扩展生成树时依赖的根，不难看出Prim算法的运行过程很形象：就好象从选定的根开始慢慢生长，最终长出了一颗生成树，因此我们又称这一类算法为**生长法**。

算法实现上，将不在黑色圆圈内的点放入一个优先队列Q中，每个点的优先级即为其到已经扩展结点的最短距离，则每次取出Q中最小的点u加入树中，并试图修改与u相连的不在树上点的最短距离，如下所示。

算法的时间复杂度取决于如何实现优先队列Q，若采用一般二叉堆，不难看出整个算法需要O(ElogV + VlogV)=O(ElogV)的时间；使用Fibonacci堆，可以在平摊O(1)的时间内完成11行的更新操作，此时Prim算法的时间复杂度为O(VlogV+E)。

小测验

1. 在证明两个算法正确性的时候我们都使用了调整法，什么是调整法，它还有哪些主要应用？

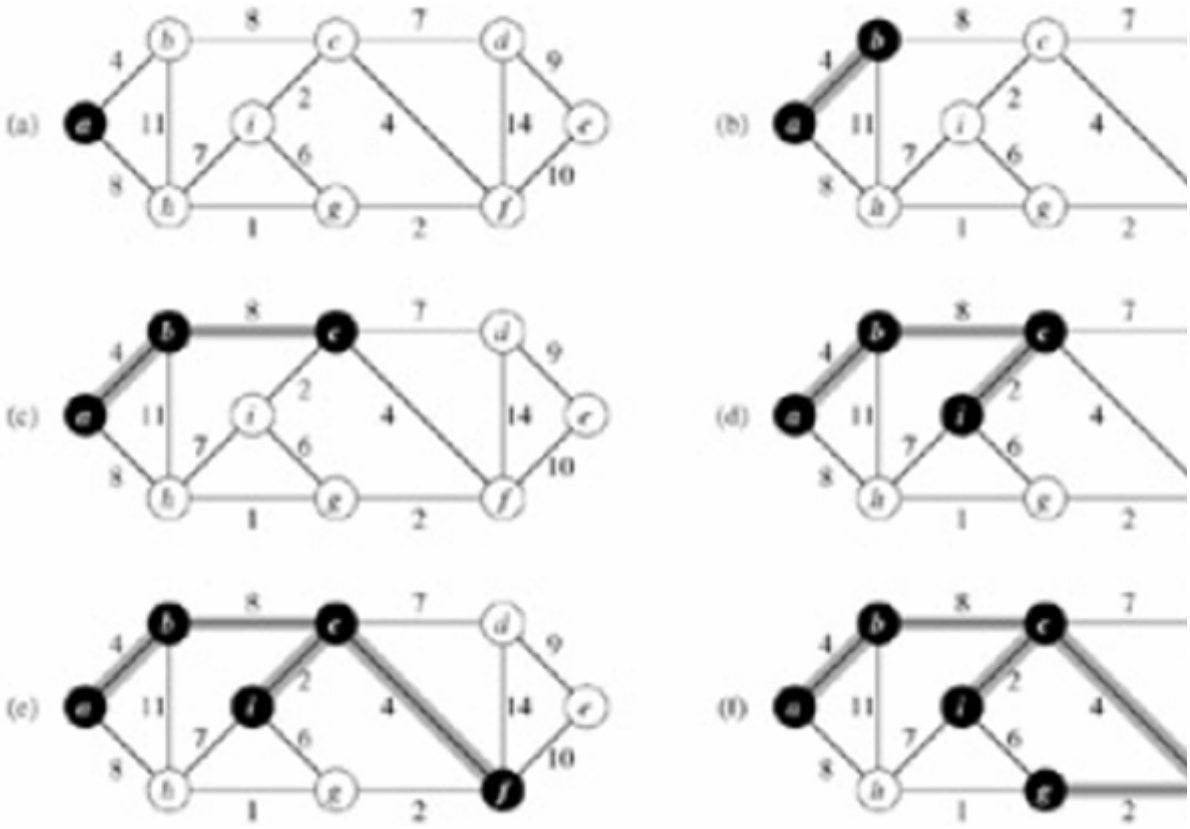


2. 缩点技术也常常用在算法设计中，请认真体会缩点的过程，并说说缩点的前提，步骤，以及它有哪些好处。

3. Kruskal算法和Prim算法确定安全边的具体方法不同，然而可以被统一成下述定理：设两个非空集合S和V-S，且A中没有边将他们相连，则G中连接S和V-S的权值最小的边一定是A的安全边。请予以证明。

7.2.2 最小度限制生成树

还是研究这样的一个无向图G=(V, E)以及其加权函数w:E◇R，$v_0 \in$V是一个确定的结点，要求一个生成树T，其中v_0的度$d_T(v_0)$=k，则称T为G的**k度限制生成树**。若同时要求w(T)最小，则称其为**最小k度限制生成树**。



这里介绍一个求最小k度限制生成树的较为简单的算法，由F. Glover和D.Klingman在1975年提出。

可行交换。T是G的一颗生成树，且存在两条边e ∈E-T，f ∈T，若T+$e-f$仍然是一颗生成树，则称这是T的一个可行交换，记作$(+e, -f)$。

引理1。设T_1和T_2是图G的两个不同的生成树，其中

$$T_2 \ / \ T_1 = \{e_1, e_2, \ldots e_m\}$$

$$T_1 \ / \ T_2 = \{f_1, f_2, \ldots f_m\}$$

则存在$\{f_i\}$集合的一个排列$f_{i1}, f_{i2}, \ldots f_{im}$，使得$(+e_j, -f_{ij})$都是$T_1$的可行交换。

这个引理的证明较为简单，请读者自行完成（习题1）。

定理1。设T是G的k度限制生成树，T是G的最小k度限制生成树当且仅当以下三条全部成立：

1. 对于G中任何两条与v_0关联的边所产生的T的可行交换都是不可改进的；

```
MST-PRIM(G, w, r)
1  Q ← V[G]
2  for each u ∈ Q
3      do key[u] ← ∞
4  key [r] ← 0
5  ∏ [r] ← NIL
6  while Q ≠ ∅
7      do u ← EXTRACT-MIN(Q)
8         for each v ∈ Adj[u]
9             do if v ∈ Q and w (u, v) < key[v]
10                then ∏ [v] ← u
11                     key[v] ← w(u, v)
```

2. 对于G中任何两条不与v_0关联的边所产生的T的可行交换都是不可改进的；

3. 对于T的任何两个可行交换$(+e_1, -f_1)$和$(+e_2, -f_2)$，若e_1、f_2关联，e_2、f_1不与v_0关联，则有$w(f_1)+w(f_2) \leq w(e_1)+w(e_2)$。

证明。首先不难证明最小k度限制生成树一定满足以上三个条件即它们的必要性。接着来证明条件的充分性：设某个k度限制生成树T满足这三个条件，若存在另一个k度限制生成树T'：w(T')¡w(T)，则不妨设

$$T' / T = \{e_1, e_2, \ldots e_m\}$$

$$T / T' = \{f_1, f_2, \ldots f_m\}$$

于是

$$\sum_{i=1}^{m} w(e_i) < \sum_{i=1}^{m} w(f_i)$$

根据引理1，存在{f_i}的一个排列，不妨仍设为$f_1, f_2, \ldots, f_m$，使得T+e_i-f_i仍然是G的生成树，则可以将上式改写为：

$$\sum_{i=1}^{m}(w(e_i)-w(f_i))<0$$

也就是说T的m个可行交换(+e_i, -f_i)中一定存在可以改进生成树权值的可行交换，显然根据条件(i)和(ii)，若e_i和f_i同时和v_0相连或是同时不相连都不可能对权产生改进，则由于T'也是k度限制生成树，则剩余的可行交换一定可分为数量相等的两类：仅e_i与v_0和仅f_i与v_0关联。将这两类任意配对，一定至少存在一对可行交换(+e_x, -f_x)和(+e_y, -f_y)可以对T产生改进，其中e_x和f_y与v_0关联，e_y和f_x 不与v_0关联，同时对T实施这两个可行交换，v_0的度数不变，且

$$w(f_x) - w(e_x) + w(f_y) - w(e_y) \text{ ¿ } 0$$

这与条件(iii)产生矛盾，假设不成立，即T是最小k度限制生成树，充分性亦满足。命题得证。

定理2。设T是G的最小k度生成树，E_0是G中与v_0关联的边的集合，$E_1=E_0\backslash E(T)$，$E_2=E(T)\backslash E_0$，可行交换集合{(+e,-f)| e∈E_1,f∈E_2}。若A为空，则G没有k+1度限制生成树。否则选取A中权增量最小的

$$w(e')-w(f')=\min\{w(e)-w(f)|\ (+e,-f)\in A\}$$

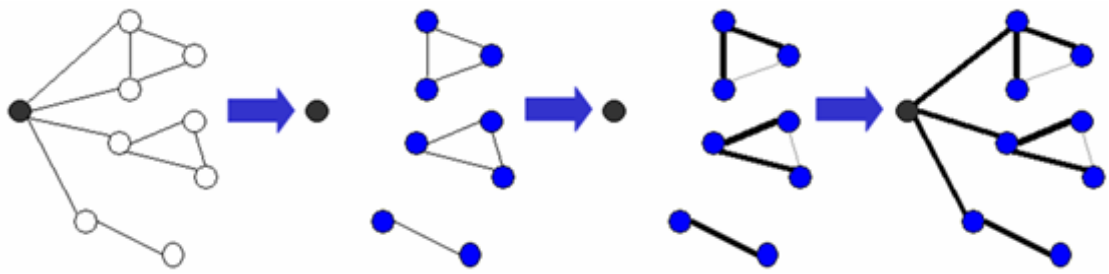
则T'=T+e'-f'是G的一个最小k+1度限制生成树。

证明。若A为空则G无k+1度限制生成树是显然的，而若A不为空，那么证明T'满足定理1中的三个条件。这一部分较为简单但叙述繁琐，因此这里略去，有兴趣的读者可以自己思考一下。

最小度限制树。如果将v_0以及与其关联的边从G图中删去，那么剩余的图可能不再连通，设有m个连通分量，那么很容易得到一个最小m度限制生成树，且m是最小的可行度限制[1]。这一点是显然的，因为把v_0和一个连通分量连接起来至少需要一条边，且每一条边不可能同时让两个连通分量都和v_0连通。

显然用来连通v_0和其他分量的边应该尽量短，且它们的选择相互独立，因此只需要对每个连通分量中选择其中和v_0关联的权最小边。每个分量内部呢？只需要各求一次最小生成树即可，不难看出这样的做法是最明智的。下面给出算法框架。

[1] 即任何小于m的度限制生成树都不存在。



```
  MIN-DEGREE-MST(G, w, v0, *m)
1         G' <- G / {v0}
2         m <- G'中连通分量数目
3         T <- 空集
4         for S = G'中每个连通分量
5           do T <- T ⋃ MST-KRUSKAL(G, w)
6         根据权w的非递减顺序对G中与v0关联的边排序
7         for 每条边(v0, u) ∈ E, 按w的非递减顺序
8           do if v0与u在T中不相连
9                then T <- T ⋃ {(v0, u)}
10        return T
```

主框架。由此，如果输入中的kjm，则可以判定无解：G中不存在k度限制生成树。否则可以根据定理2每次选择一个最优可行交换由最小m度限制生成树得到最小m+1度、m+2度，直到所需要的度数k：

```
  K-DEGREE-MST(G, w, v0, k)
1         T <- MIN-DEGREE-MST(G, w, v0, m)
2         if k < m
3           then return 无k度限制生成树
4         while k <> m
5           do T <- EXTEND-DEGREE(G, w, v0, T)
6              if T = 不存在
7                then return 无k度限制生成树
8              m <- m + 1
9         return T
```

算法的框架如此，而定理2的着重体现和效率优化的关键都在于EXTEND-DEGREE过程，它的任务就是根据定理2选取最优的可行交换，将最小m度限制生成树扩展为v_0的m+1度。

交换边的确定。如果直接枚举两条交换边，则过程的时间复杂度高达$O(E^2)$，而不难发现可行交换的+e必定与v_0相关联，枚举+e之后，T上必定出现一个环，而显然此时的-f必定是环上与v_0不关联的w权最大的边。把无根树转化为有根树后可以通过一个简单的动态规划预处理后在常数时间内取到这条边。

```
   EXTEND-DEGREE(G, w, v0, T)
 1         for u ∈ V
 2           do Best[u] <- -∞
 3              BestChoice[u] <- NULL
 4         以v0为根对T进行宽度优先搜索，定义father[v]为v在搜索树上的父亲
 5         for u ∈ V / {v0} 按宽度优先搜索先后到达的顺序
 6           do if father[u] <> v0
 7                then if Best[father[u]] > w(u, father[u])
 8                       then Best[u] <- Best[father[u]]
 9                            BestChoice[u] <- BestChoice[father[u]]
10                       else Best[u] <- w(u, father[u])
11                            BestChoice[u] <= (u, father[u])
12         Exchange <- NULL
13         for (v0, u) ∈  E / T, 且father[u] <> v0
14           do if w(v0, u) - Best[u] < Exchange.cost
15                then Exchange <- (+(v0, u), -BestChoice[u])
16         return 对T施行Exchange的可行交换
```

不难看出这个过程的时间复杂度仅为O(V)，这样综合以上三个过程，就是完整的求最小k度限制生成树算法。一般来说我们仍认为求最小生成树的复杂度[2]为O(ElogE)，因此这个算法的总时间复杂度为O(ElogE + kV)。

小测验

1. 请尝试证明引理1。（提示：可以使用关于二分图匹配的Hall定理，参见后文）

2. 本小节介绍的最小k度限制生成树的时间效率是与k有关的，请思考如果k比较大的时候有哪些简单的优化方法，并上机验证自己的想法，分析一下在哪些条件下优化的效果会更好。（提示：一种思路是着手于EXTEND-DEGREE这一过程，进行简单的动态维护）

3. 最小k度限制生成树的算法和最小生成树的算法虽然在轮廓上有很大不同，但都属于贪心算法。请读者仔细分析体会，为什么这些有关生成树的算法都可以使用贪心算法，下一小节即将讲解次小生成树乃至k小生成树，它们的算法也是另一种形式，请您尝试思考，看看能不能自己分析得出一些有效算法。

[2] 使用Fibonacci堆优化的Prim算法实现过于复杂且优化效果不明显，非极端情况不建议使用。

7.2.3 次小生成树及生成树的排序

在某些问题中，我们需要检索图G中满足某个特殊条件的代价最小的生成树，若是没有其他特殊方法，最朴素的方法就是从代价最小、第二小、第三小...的生成树开始检索，直到找到满足条件的为止。而我们即将讨论的问题就是，如何从小到大依次列出每一个生成树？

邻集。N(T)表示对生成树T进行且只进行一次可行交换后得到的集合，成为T的邻集。

次小生成树。对于一个连通的无向图G来说，其边的加权函数为w:E◇R。T_1是G的一棵最小生成树，若另有一棵树T_2，且不存在第三棵树T'满足w(T')¡w(T_2)，则T_2是G的次小生成树。

定理1。设T_1是G的最小生成树，如果另一棵最小生成树T_2满足

$$w(T_2) = \min\{w(T) \mid T \in N(T_1)\}$$

则T_2可以作为G的次小生成树。

证明。假设T_2不能作为G的次小生成树，即存在第3棵树T'，使得

$$w(T_1) \leq w(T') ¡ w(T_2)$$

显然$T' \notin N(T_1)$，那么设

$$T' \setminus T_1 = \{e_1, e_2, \ldots, e_m\}$$

$$T_1 \setminus T' = \{f_1, f_2, \ldots, f_m\}$$

其中m≥2，根据上一节的引理1可知，存在$\{f_i\}$的一个排列：不妨仍然设为$f_1, f_2, \ldots, f_m$，使得

$$T_1 + e_i - f_i$$

是G的生成树，且它们均属于$N(T_1)$，由于T_1是G的最小生成树，所以有

$$e_i - f_i \geq 0$$

根据T_2的选择方法，有$w(T_1 + e_i - f_i) \geq T_2$，因此$w(T') \geq T_2$，这与假设矛盾。因此命题得证。

根据定理1，很容易设计出求G的次小生成树的算法：首先求出最小生成树T_1，接着在T_1的邻集中寻找一个权和最小的生成树作为T_2输出即可。如何高效的完成后面一步是关键，使用与最小度限制生成树类似的预处理方法，即可将时间复杂度降为$O(V^2)$：

```
   SECOND-MST(G, w)
1         T1 <- Kruskal(G, w)
2         for u ∈ V
3           do 以u为根节点在T1上进行一次BFS
4              记录到v路径上的权最大边为Best[u, v]
5         T2 <- NULL
6         for (u, v) ∈ E \ T1
7           do if w(T1 + (u, v) - Best[u, v]) < w(T2)
8                T2 <= T1 + (u, v) - Best[u, v]
9         return T2
```

其中Best[u, v]记录T_1上u到v的唯一路径上权最大的边，试图添加每一条不在T_1上的边，并删去加边形成环上除被加入边的权最大边。枚举完所有的添加边以后选择一个权和最小的作为次小生成树输出即可。

k小生成树。我们定义了次小生成树，也就相当于这里的“2小生成树”。设已经定义了(k-1)小生成树的集合$\{T_1, T_2, \ldots, T_{k-1}\}$，有另外一棵G的生成树$T_k$，且不存在与以上所有树都不同的T’，满足$w(T_{k-1}) \geq T'$¿$T_k$，则可将$T_k$作为k小生成树。

将定理1的推广一下，不难得到有关k小生成树计算方法的定理。

定理2。设$T_1, T_2, \ldots, T_{k-1}$是已经选定的前k-1个最小生成树的序列，则如果$T_k$满足

$$w(T_k) = \min\{w(T) \mid N(T_1) \cup N(T_2) \cup \ldots \cup N(T_{k-1})\}$$

则T_k可以作为G的第k小生成树。

证明和定理1的类似，请读者自己推导（小测验1）。

首先求出G图的最小生成树之后，可以设置一个优先队列Q，记录当前求得的前几小生成树的邻集，并将它们的权和作为优先级。假设已经得到了前m小生成树，它们的邻集都在Q中，那么EXTRACT-MIN(Q)得到的就是m+1小生成树。

需要注意的是，一棵树的邻集大小约为O(EV)，由于次小生成树的特性，我们可以仅仅枚举O(E)棵，而这在求k小生成树时可能会导致错误：即需要将邻集中所有元素加入Q中，同时还需要设立一个Hash表除去有可能的重复。

```
   kTH-MST(G, w, k)
 1         T[1] <- Kruskal(G, w)
 2         Q <- NULL
 3         for i <- 2 to N
 4           do for e ∈ E \ T[i - 1]
 5                do for f ∈ T[i - 1]且f在e加入T[i - 1]后形成的环上
 6                     do if not HASH-FIND(T[i - 1] + e - f)
 7                          then HASH-ADD(T[i - 1] + e - f)
 8                               QUEUE-ADD(Q, T[i - 1] + e - f)
 9              T[i] <- EXTRACT-MIN(Q)
10         return T[k]
```

使用普通二叉堆来实现优先队列，则时间复杂度约为O(kEVlogk)，然而此时堆中有O(kEV)个元素，空间复杂度高达$O(kEV^2)$。[3]因此不妨添加一个最大堆或改用平衡二叉树实现优先队列，仅保留前k小的元素，空间复杂度仅为O(kV)。

小测验

1. 次小生成树问题是k小生成树的特例，请仔细体会定理1的证明，然后证明定理2。
2. 在求次小生成树的时候，如果G是一个稀疏图，请您试图优化SECOND-MST算法的预处理部分，能不能得到一个时间复杂度与E有关的较为高效的算法？
3. 在求k小生成树的时候，若k比较小，不难看出并不用完全枚举邻集中的所有元素，请读者设计一个优化方案，将O(kEVlogk)中的V换成一个与k有关的因子。

7.2.4 有向图的生成最小树形图

这一节主要是讨论图的生成树问题，而生成树是基于无向图定义的。此小节将此定义推广至有向图的树形图，探讨树形图的特性以及最小树形图的解法，作为本节的结束。

[3] 假设存储一棵树需要O(V)的空间。

生成树形图。已知一个有向图G=(V, E)，以及其加权函数w: E◇R。若存在G的一个无环生成子图T和一个结点v_0，沿T上的有向边从v_0出发可以到达V中所有其他的结点。则称T是G的一个生成树形图，而v_0则是这个**生成树形图的根**。

最小生成树形图。在G的所有生成树形图中，若T的边权和w(T)最小，则称T是G的一个最小生成树形图。

就无向图而言，只要其是连通的，就一定存在生成树：也就是一定存在最小生成树。而对于有向图G来说，首先可以求出G的强连通分量，如果有一个以上的强连通分量没有入度，则显然G没有生成树形图，否则一定可以找到以这个唯一的没有入度的强连通分量中的任意一个点为根的生成树形图。

这样就对生成树形图存在性问题做了解答。那么如何求得最小生成树形图呢？我们通过对下列定理的理解和证明就很容易设计出一个高效的算法，由中国科学院的朱永津、刘振宏教授在1965年提出。

对于任意的v∈V，选择v的所有入弧中权最小的一条作为v的**最小入弧**。

引理1。设对有加权函数w的有向图G的每一个顶点各选取一条最小入弧，组成一个集合T：

- 如果|T| ¡ |V| - 1，则G没有生成树形图。
- 如果|T| = |V| - 1，且T中没有环，则T是G的最小生成树形图。
- 如果|T| = |V|，则设e是T中权最大的边，且T - e中没有环，则T – e是G的最小生成树形图。

这个引理很简单，这里略去证明。

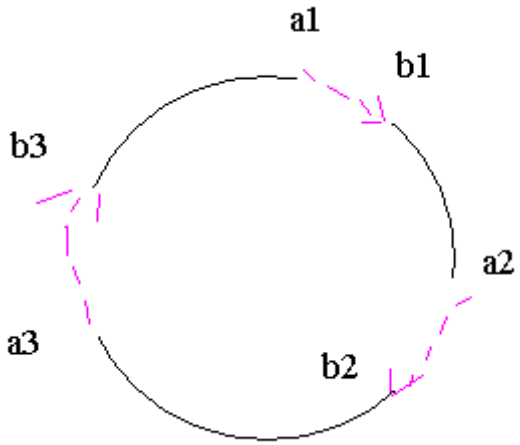
一般来说，我们构造得出的T中都是有环的：假设C是T中的一个环，则由下面的定理成立。

定理1。设C是T中的一个环，则如果G存在生成树形图，则一定存在某个最小生成树形图T_0满足$|C \setminus T_0| = 1$。

证明。由于任何生成树形图中都不会有环，那么显然$|C \setminus T_0| \geq 1$。

接着假设定理不成立，那么不妨在G的所有最小生成树形图中选择一个T_0，使得$|C \setminus T_0| = k$最小，那么k≥2。那么如下图所示，不妨设C环上有以下k条边不在T_0上：

$$(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)$$



不妨设在C环上b_1是T_0中最高（离根最近）的结点。则b_1在T_0中不会是b_2的后代，而a_2是b_1的后代，所以a_2也不会是b_2的后代。那么我们删去T_0中b_2的入弧而加上(a_2, b_2)，这显然也是G的生成树形图。而由于加上的(a_2, b_2)边是b_2的最小入弧，则显然这样的变化不会使权和更大，因而新的生成树形图不妨称为T_1也是G的最小生成树形图。而此时

$|C \setminus T_1| = |C \setminus T_0| - 1$

这与我们T_0的取法产生矛盾，进而定理得证。

定理1告诉我们，在最小入弧的集合T中，如果出现了环，那么在这个环上，最多只需要改变一条边就可以了。这里我们再次运用**“缩点技术”**[4]，将C环中的点全部用一个新的点c代替。所有从c连出的边仍完全保留，而对于每一条连入c的边e，设其连入C中的点u，而且在C中存在边(v, u)。如果我们的最小生成树形图选用了e，则必定要删去(v, u)：且根据定理1，只需要删除这样一条边。因此我们可以将e的新权值w’(e)定义为

$$w'(e) = w(e) - w(v, u)$$

以此来说明这一调整过程。

这样在这样一次收缩点之后，图的规模得到减小，递归处理这一问题，可以得到G’图的最小生成树形图T’。那么最后将T’中c还原成原图中的点即可。算法的工作流程如下：

[4] 请注意这里与第一小节中缩点法的区别，第一小节中仅仅在思考问题时，将很多个点看作一个整体，而在这里，我们真正的要在程序中进行压缩替代。

```
   MINIMUM-SPANNING-DIRECTED-TREE(G, w)
 1         求出G中每个结点的最小入弧集合T
 2         if |T| < |V| - 1
 3           then return NULL
 4         if T中没有环
 5           then return T
 6         任意找到T中的一个环C
 7         for v ∈ V
 8           do father[v] = NULL
 9         (G', w') <- SHRINK(G, w, C)
10         T <- MINIMUM-SPANNING-DIRECTED-TREE(G'(V', E'), w')
11         if T = NULL
12           return NULL
13         T <- UNWRAP(G, w, G', w', T)
14         return T
```

其中第9行的SHRINK为压缩C集合过程，而第13行的UNWRAP则展开C中的点。

```
   SHRINK(G, w, C)
 1         V' <- V \ C ∪ {c}
 2         E' <- 空集
 3         for (u, v)∈ E且u, v均不∈ C
 4           do E' <- E' ∪ {(u, v)}
 5              w'(u, v) <- w(u, v)
 6         for (u, v)∈ E, u∈ C, v不∈ C
 7           do if father[v] = NULL或w(father[v], v) > w(u, v)
 8                then E' <- E' ∪ {(c, v)}
 9                     w'(c, v) <- w(u, v)
10                     father[v] = u
11         for (u, v)∈ E, u不∈ C, v∈ C
12           do 令(v', v)为C中的边
13              E' <- E' ∪ {(u, c)}
14              w'(u, c) <- w(u, v) - w(v', v)
15         return (G'(V', E'), w')
```

UNWRAP过程可以描述为：

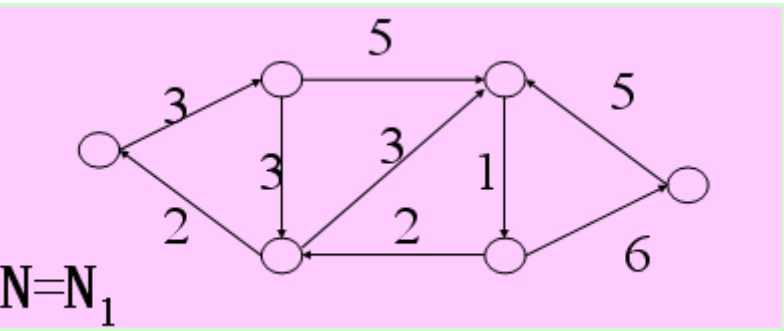
如果把每步的原始图称为N_i，最小入弧圈称为H_i，而展开之前的图$H'_m=H_m$，每次从H'_m展开得到图H'_{m-1}，直到H_1。下面是一个这样的例子。

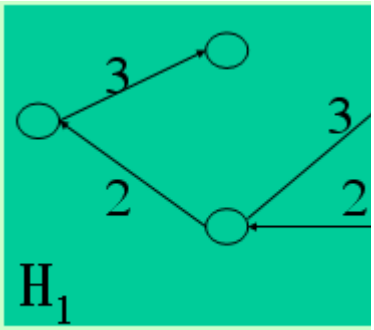
还可以画出算法框图：

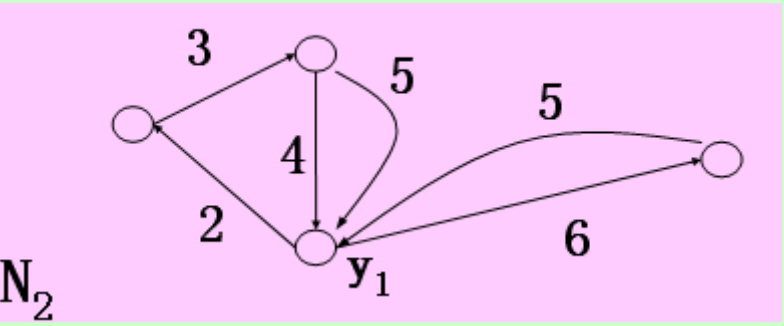
读者不难把刚才的递归过程写成迭代形式。

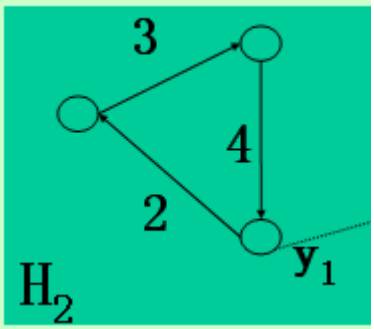
不难看出SHRINK和UNWRAP子过程以及一次执行主过程MINIMUM-SPANNING-DIRECTED-TREE的时间复杂度都是O(E)。由于每次主过程递归点的规模都有所减少，因此最多会递归O(V)次。于是这个算法的时间复杂度为O(VE)。

```
UNWRAP(G, w, G', w', T')
1         T <- 空集
2         for (u, v) ∈ T', v <> c
3           do if u = c
4                then E <- E ⋃ {(father[v], v)}
5                else E <- E ⋃ {(u, v)}
6         T <- T ⋃ C
7         if c是T的根
8           then 令(u, v)是C中权最大的边
9                T <- T \ {(u, v)} ⋃ C
10          else 令c在T中的父亲为u
11               for (u, v) ∈ E, v ∈ C
12                 do 令(v', v)为在c中的边
13                    if w'(u, c) = w(u, v) - w(v', v)
14                      then T <- T \ {(v', v)} ⋃ {(u, v)} ⋃ C
15                           break
16        return T
```







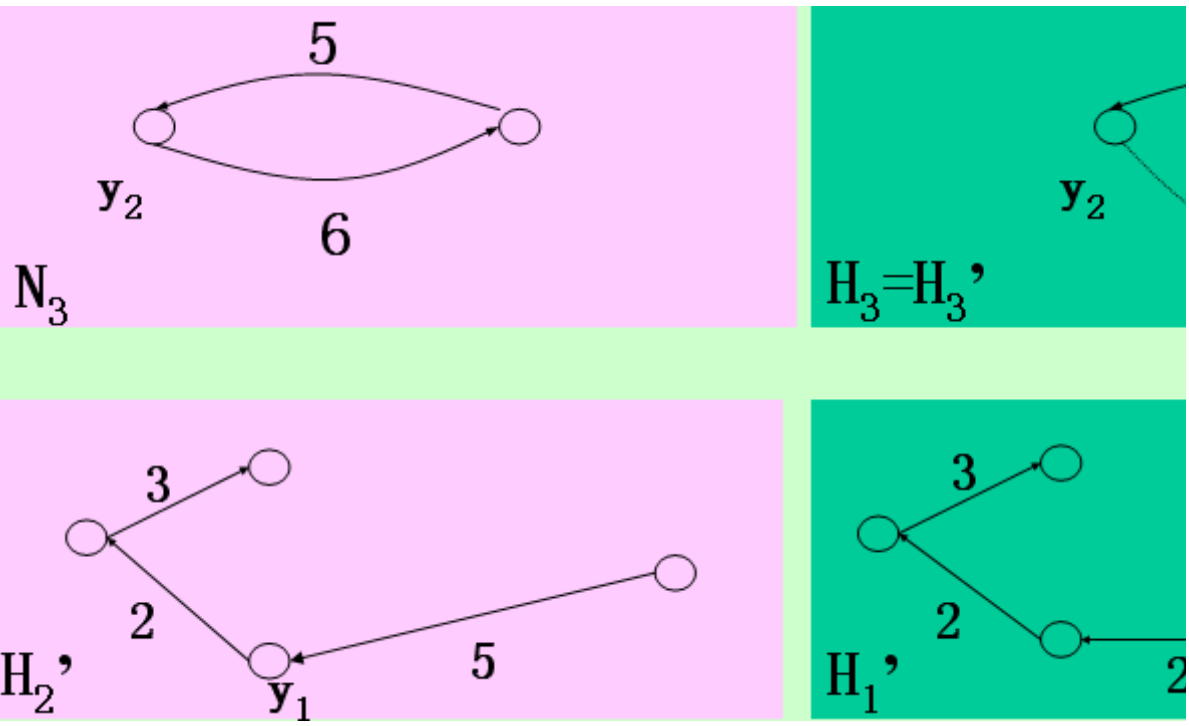


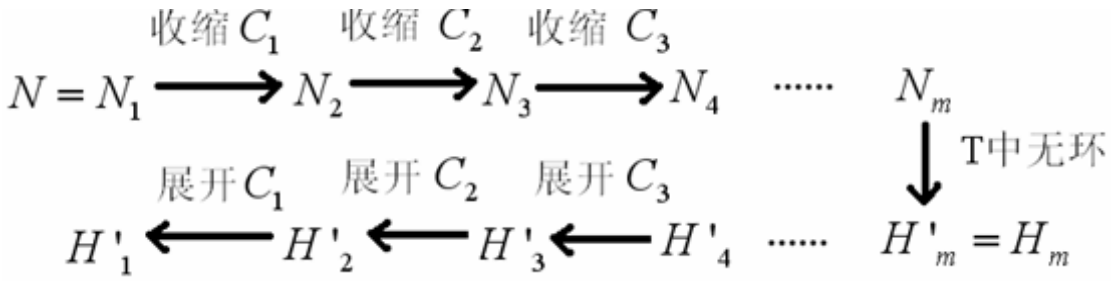
小测验

1. 本小节介绍的算法较为复杂，请仔细体会伪代码，并独自编写程序、调试。

2. 请说说引理1与定理1之间的联系，以及他们在算法中的作用。

3. 如果规定了最小生成树形图的根必须是v_0，请您根据原有的算法，想一





种最简单的方法，解决新的问题。

7.2.5 小结

本节介绍了生成树相关问题，包括最小生成树、最小度限制生成树、生成树的排序和最小树形图。这些算法都有较强的启发性，且相关定理的证明非常基本，希望读者全部掌握。本节给出的算法几乎都不是理论最优的，但是算法清楚，实现简单。有兴趣的读者可以参考相关资料。

本节的算法和程序列表如下：

算法	程序	备注
kruskal算法	kruskal.cpp	最小生成树的kruskal算法，用并查集实现。
prim算法	prim.cpp	最小生成树的Prim算法，用二叉堆实现。
klover-glingman算法	degree.cpp	最小度限制生成树的Klover-Glingman算法。
第二小生成树算法	2-mst.cpp	第二小生成树的$O(n^2)$算法。
生成树排序	k-mst.cpp	生成树排序的O(kmnlogk)时间O(kn)空间算法。
朱-刘算法	chu-liu.cpp	最小树形图的朱-刘算法的基本实现，时间复杂度为O(nm)

7.3 最短路问题

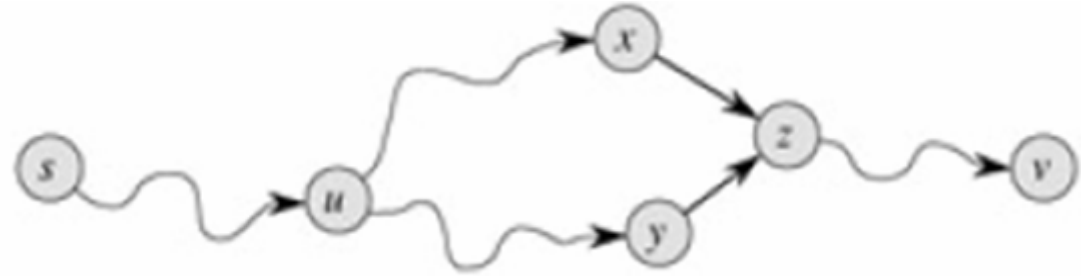
本节讨论最短路问题。我们首先给出最短路的定义和几个相关问题，然后依次讨论标号设定算法和标号修正算法，然后讨论每两点间最短路问题，给出floyd-warshall算法和Johnson算法，最后讨论k短路问题。

7.3.1 单源最短路问题

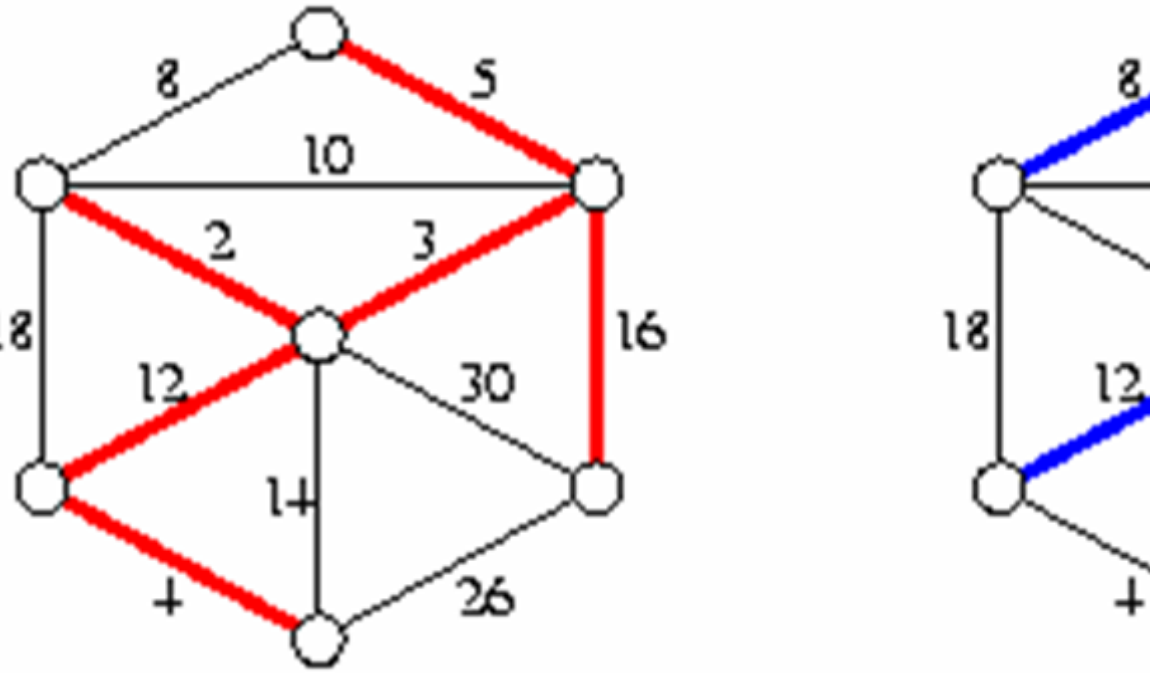
单源最短路问题(single-source shortest paths, SSSP)可以叙述如下：给加权图和一个起点s，求s到所有点的最短路。这里的最短指的是边权之后最小。关于SSSP最基本的结论是：最短路一定不存在环。如果环上的权和是正的，那么没有必要走这个环；如果权和是负的，那么没有最短路，因为可以沿这个负环兜圈子，则路径长可以无限变小。

解决单源最短路问题的基础是最优性原理：若最短路u–¿v经过中间结点w，则u–¿w和w–¿v的路径分别u到w和w到v的最短路。这个最优性原理中在动态规划中出现过，因此这里的分析和动态规划非常类似。事实上，每两点间最短路问题的floyd-warshall算法就是基于动态规划的。

一般用最短路树来表示单源最短路的结果，不管目标点如何，从s到它在树上都有唯一通路。实际代码中只需要记录每个点的父亲pred[u]，输出最短路时从终点开始倒退，直到回到原点。显然如果从s到u的最短路不唯一，最短路树只能表示其中的一条，其他的需要根据距离值找到，留给读者思考。为什么最短路形成树呢？不妨考虑下图，在这样的情况下实际上只需要取s–¿z的一条最短路而不是两条。



最短路树和最小生成树可以不一样。如下图(a)是最小生成树，图(b)是最短路树。

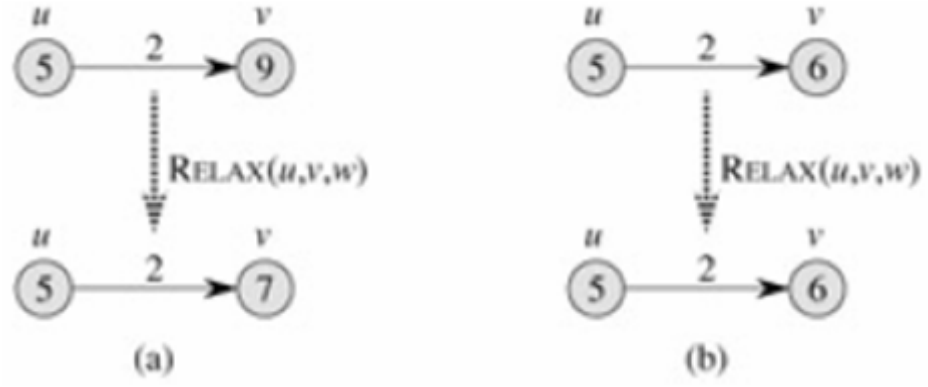


通用SSSP算法。下面我们介绍一个通用的SSSP算法，后面要介绍的dijkstra算法和bellman-ford 算法都是它的特例。首先介绍“临时最短路”的概念：它一定是存在的，即真实的最短路长度不大于此路长度，但有可能存在比它更短的，因此此路长度只是一个上界。

给定起点s，对于每个顶点v，定义：dist(v)为临时最短路树中s-¿v的长度，prev(v)为临时最短路树中s-¿v中v的前驱，初始化dist(s)=0, pred(s)=NULL，而其他点v的dist(v)为正无穷，prev(v)=NULL。这里的dist(v)称为点v的**标号(label)**，它是s-¿v最短路长度的上界。通用SSSP算法的基本思想是：让标号不断趋近，最终达到最短路的真实值。

如何让标号接近呢？有一类标号是明显可以改进的：dist(u)+w(u,v)¡dist(v)。这样的边(u,v)称为**紧的(tense)**，可以对它进行**松弛(relax)**：dist(v)=dist(u)+w(u,v)，pred(v)=u，

如下图。



紧边的概念极为重要，因为有如下的定理：

定理：临时最短路树的标号均为真实最短路长当且仅当图中没有紧边。

证明：如果存在紧边(u,v)，临时标号dist(v)显然不是真实值，因此只需要证明如果不存在紧边则所有临时标号均是真实的。

一方面，dist(v)和prev(v)是相容的（即由pred表示出的路径上所有边权和等于dist(v)），这一点归纳于松弛的次数很容易证明；另一方面，对s到v的任意路s-¿v，都有dist(v)¡=w(s-¿v)。归纳于s-¿v所含边数，假设u=pred(v)，则dist(u)¡=w(s-¿u)，两边加上w(u,v)得dist(u)+w(u,v)¡=w(s-¿v)。因为无紧边，所以dist(v)¡=dist(u)+w(u,v)¡=w(s-¿v)。

这样，我们很容易得到通用SSSP算法如下：

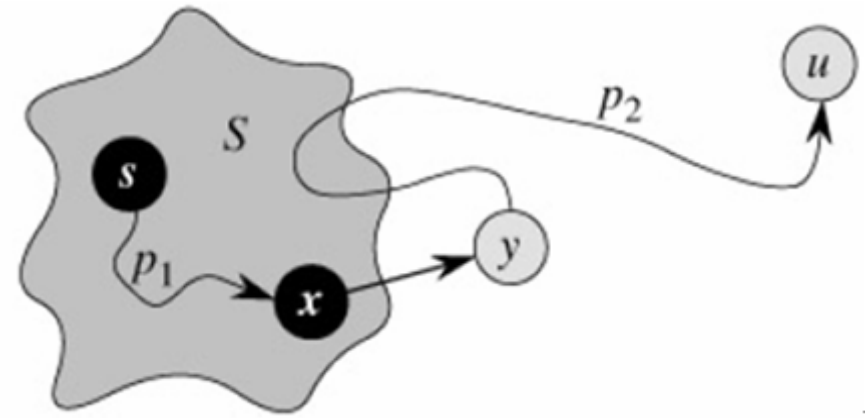
通用SSSP算法：不断找出紧边并进行松弛，更新dist和pred，直到不存在紧边。

如果算法结束，刚才已经证明了结果正确，可如果算法永远不结束呢？显然如果从s点可以到达一个负圈，则每次松弛后负圈上一定有紧边（反证法），因此算法无法结束；如果不含负圈则算法一定结束，因为每次要么减少一个无穷的dist值，要么让所有有限dist值之和至少减少一个“不太小的正值”。

通用SSSP算法并没有说明处理紧边的顺序。一般来说可以把待检查的结点放到一个集合里，每次从集合中取一个结点u，检查它的所有关联边(u,v)，如果是紧的则进行松弛操作并把v放到集合中。下面介绍的disjktra算法和bellman-ford分别用堆和队列实现这个集合，它们的特点不一样。

dijkstra算法。用堆来实现通用SSSP算法的集合，则可以得到dijkstra算法。dijkstra算法是E.W.Dijkstra于1959年在《A note on two problems in connection with graphs》中

提出的，虽然它是通用SSSP算法的特例，但是我们需要清醒的看到：它是专门为非负权网络设计的，且它最引入注目的一点是：每次dist(v)最小的一个恰好等于它的最短值，予以固定。注意在通用SSSP算法中，只有当整个算法结束时我们才说所有标号都是正确的。但在非负权网络中，dijkstra算法每次固定一个标号，保证它已经达到最优解。



由于每次固定一个标号，我们把dijkstra算法归为**标号设定算法(label setting algorithms)**这个类别中。固定一个标号后需要把它的邻居添加到堆中，每条边处理一次，因此总时间复杂度为O(mlogn)（注意松弛操作需要调用堆的decrease-key操作）。对于稠密图，每次可以用枚举的方法寻找出值最小的dist(v)然后加以固定检查和它相邻的边，每次固定一个结点需要检查所有n个点，因此总时间复杂度为$O(n^2)$。

Dial实现。dijkstra算法有一个特殊的实现方法，称为**Dial实现法(Dial's implementation)**。这个方法使用nC个桶，每个桶对应一个距离标号的取值（即在同一个桶中的距离编号相同）。这样，距离标号的三个操作对应于：

取最小值 从小到大检查所有桶，直到有一个桶不为空，则该桶里的所有距离标号均可以被固定。

固定 从桶中删除。由于桶里的距离标号都是临时的，只要一固定就可以立刻删除。

减小 一定是从一个桶移动到另一个桶，而不会保留在同一个桶中。

由于最后一个操作涉及到的桶内元素的插入和删除，所以每个桶可以用双向链表实现。

固定和减小操作都是常数级别的，因此这两个操作的总时间为O(m+n)。取最小值时需要找非空的桶，最坏情况需要遍历所有桶，时间复杂度为O(nC)。这是否意味

着n次取最小值操作一共需要O(n^2C)时间呢？不是的。注意到每次固定的距离编号是不减的（想一想，为什么），因此第一个空桶的位置也是不减的，所以每个桶都不会扫描两次或以上，总时间复杂度为O(m+nC)。

基数堆实现。Dial实现法的瓶颈在于取最小值操作时对空桶的扫描。如果桶的总数变少，那么空桶扫描将变快。基数堆实现中有K个桶1，2，...，K，初始大小（也称宽度）为1，1，2，4，8，...，2^{K-2}，运行时每个桶的大小是可变的。取$K = 2\lceil\log(C+1)\rceil$，则桶K的大小至少为C。注意我们一般保证最后一个桶的宽度为C（想一想，为什么）。

更新操作 可能需要把点从一个桶移动到另一个桶。注意在基数堆中这个点也可以留在同一个桶中。由于桶的大小是可变的，所以只能往回一个一个试，直到距离值落在该桶的范围内。每次距离更新都可能会扫描所有K个桶，是否意味着总时间为O(mK)呢？不是的。从全局看，每个点的桶编号不会增加，因此对于每个结点来说最多一共扫描所有K个桶，总时间复杂度为O(m+nK)。

取最小值操作 对于一般的桶实现，取最小值操作分为两步，一是找到编号最小的非空桶，而是在该桶中取最小值。基数堆的桶数比较少，因此扫描非空桶的时间复杂度仅为O(nK)。但由于每个桶里的元素不止一个，在桶中取最小值不再像Dial实现中一样是平凡的了。如果仍然有双向链表储存每个桶，则取最小值的时间复杂度为O(n_1)，其中n_1是桶中的元素个数。基数堆的巧妙之处在于它在O(n_1)时间找到桶B的最小值d*后进行了一个*顺便操作*，把桶B的所有元素*重新分配*到桶1，2，...，B-1，清空自身并设范围为空。这些被分配桶的起始距离标号为d*，大小仍为1，1，2，4，8，...，然后从桶1中取出最小值。注意到每个结点最多被重新插入K次，因此重分配的时间复杂度为O(nK)。注意桶B中所有结点都会被重新插入到更近的桶，因此取最小值本身的时间复杂度不超过重插入的时间复杂度，即O(nK)。

Bellman-Ford算法。这个算法由Ford于1956年、Bellman于1958年、Moore于1959年独立提出的，它是最优性原理的直接应用。这个算法是基于以下事实：

一、如果最短路存在，则每个顶点最多经过一次，因此不超过n-1条边。

二、长度为k的路由长度为k-1的路加一条边得到。

三、由最优性原理，只需依次考虑长度为1，2，...，k-1的最短路。

这样，我们只需要循环n-1次，每次依次迭代所有边即可。事实上，如果循环n-1次以前已经发现不存在紧边则可以立刻终止。设迭代次数为v（$v_¡$=n-1），则时间复杂度为O(vm)。对于完全图，如果边全在[0,1]中均匀分布，则有很大概率$v = O(\log^2 n)$。另

外用dijkstra得到一个初始标号再迭代，也会加速算法的收敛。

关于Bellman-Ford算法，Yen曾经提过一种修改方案：把G中的边(v_i,v_j)分为两类，一类是f边(i¡j)，一类是b边(i¿j)，每次迭代时i先从v_1遍历到v_n，松弛f边，再从v_n遍历回v_1，松弛b边。这样最多只需要$\lceil \frac{n}{2} \rceil$次迭代，但不降低渐进时间复杂度。

如果Bellman-Ford算法迭代了n-1次仍发现有紧边，则图中一定有负圈。由于标号始终无法固定，所以我们把Bellman-Ford归为**标号修正算法(label-correcting algorithms)**这一大类中。

寻找负圈。如果最短路不存在，那么图中存在负圈。如何寻找负圈？可以借助标号修正过程中记录的父亲指针pred(v)。迭代结束后从每个结点u出发不停往回走，沿途把所有结点都标记为u，直到走到一个已走过的点。如果这个点的标记也为u，则说明“最短路树”含圈，且u就在圈上。这样只需要O(n)的附加时间就可以检查从源点出发时候能到达一个负权圈，不增加算法的时间复杂度。

FIFO队列和双端队列实现。基本的Bellman-ford算法实际上做了不少无用操作。用一个FIFO队列可以减少无用的检查，而双端队列实现则在实际应用中表现非常好。

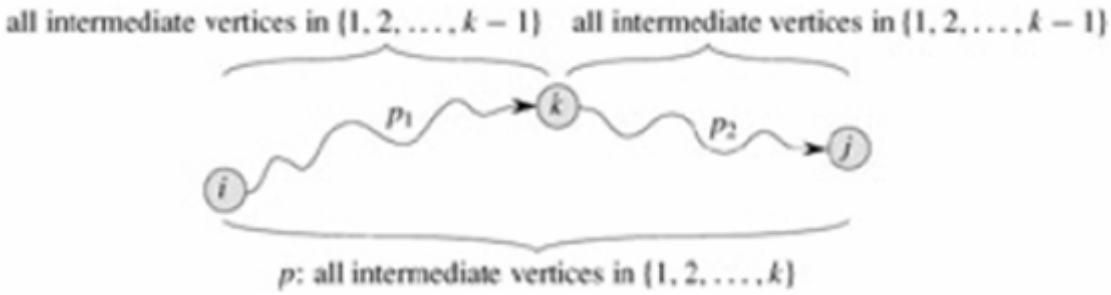
7.3.2 每对结点最短路问题

每对结点最短路问题(All-pairs shortest paths)要求任意两对结点的最短路。虽然它可以通过对每个结点出发做一次SSSP来求解，但有更简单和高效的方法。事实上，对于非负权图，可以运行n次；对于任意权图，必须运行n次bellman-ford。本节介绍的floyd-warshall算法和johnson算法不仅适合权任意的情况，而且复杂度不比n次SSSP慢：floyd-warshall算法的时间复杂度为$O(n^3)$，而johnson算法的核心是一次bellman-ford调用和n次dijkstra调用。

Floyd-warshall算法。Floyd-warshall算法可以在$O(n^3)$时间内求出每两点间最短路，对于非负权稠密图和n次dijkstra时间一致且对于任意权图比n次bellman-ford算法更快。它最引人注目的地方在于它的形式非常简单，容易证明和推广：它几乎是网络优化中最容易记忆的算法。

它的基本思想是：令d[i,j,k]为从i到j只经过结点1...k的最短路，则这样的路有两种，一种是包含点k的，一种是不包含的。显然不包含的情况等于d[i,j,k-1]，而包含的情况如下图，它等于d[i,k,k-1]+d[k,j,k-1]，它反应了最优性原理。

在程序中注意无穷大的运算，另外可以节省第三维，只用二维数组，伪代码如



下：

```
OYD-WARSHALL' (W)
 n ← rows[W]
 D ← W
 for k ← 1 to n
      do for i ← 1 to n
            do for j ← 1 to n
                  do d_ij ← mi
 return D
```

如果需要输出任意两点间的最短路，则需要记录pred[i,j]，更新时令pred[i,j]为pred[k,j]即可，时空复杂度均不变。检测负圈也是顺便的：注意到pred[i]是以i的源点的最短路树，因此可以对每个点执行一次检测操作，一共$O(n^2)$。

值得一提的是，floyd-warshall算法也属于标号修正算法，不过它是所有结点同时求解的，这一点和bellman-ford不同。

Johnson算法。Floyd-warshall算法对于稠密图是非常不错的，但是稀疏图却并不理想。这里介绍的Johnson算法适用于稀疏图，它的时间等于一次bellman-ford加上n次dijkstra。对于不同实现方式的dijkstra，Johnson算法的运行时间也略有差别，但都比直接运行n次bellman-ford [5]快很多。Johnson算法的思想是最简单的：执行n次SSSP。但直接执行dijkstra和bellman-ford都不理想：

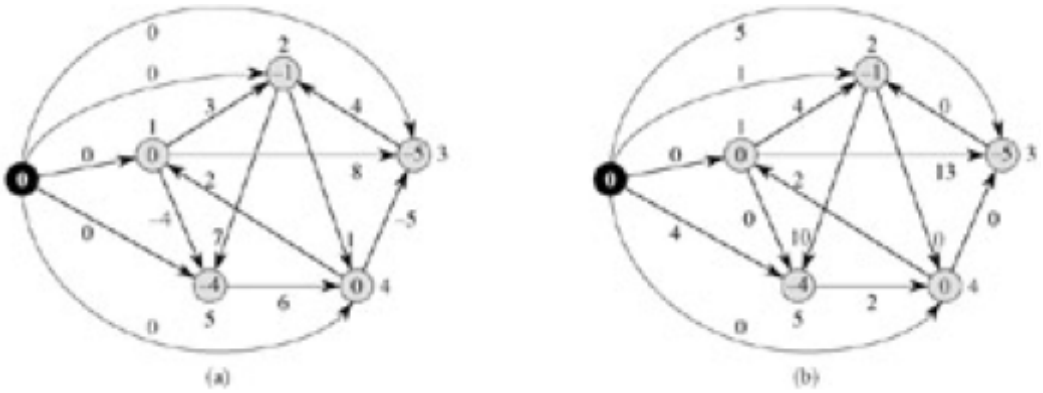
[5] 注意对于权任意的图是不能执行n次dijkstra的。

dijkstra算法速度快，但只适用于非负权图；bellman-ford算法使用任意权图，但时间慢。

Johnson算法的巧妙之处在于综合了两个算法的优点：用1次bellman-ford对图进行重加权，使得每个源点出发的最短路树都保持不变，但权变成非负，然后用n次dijkstra求出每两点间的最短路。

重加权的方法如下：增加人工结点s，直接到所有点连一条弧，权均为0，然后以s为起点运行bellman-ford，求出dist(v)。如果有负权圈则退出，否则对于原图中的每个条边(u,v)，设新权w'(u,v)=dist(u)+w(u,v)-dist(v)，则它是非负的。为什么新权是非负的？因为如果w'(u,v) ¡0，即dist(u)+w(u,v)¡dist(v)，意味着(u,v)是紧的，与bellman-ford没找到负圈矛盾。

新权的非负性只能说明可以在新图上执行dijkstra，我们还需要证明重加权后每个源点的最短路树都不变。这一步也不困难：对于给定源点s和其他任意点u，路s–¿u的权为w'(s-¿u)=w(s-¿u)+dist(s)-dist(u)[6]，是定值，因此w'和w同时取到最小值。



需要说明的是重加权技术还将在后面得到应用。

7.3.3 k短路问题

本节我们考虑求第2、3、...短路，即k短路问题。这样的问题一般有两大类，一是允许结点重复，二是必须经过不同结点。本节只讨论第一大类，即允许结点重复，且权都是不太小的正数的情况。这里讲的k短路没有去除重复长度，即如果一共有五条路，长度为1，2，2，3，3，则前3短路长度为3，3，2。

[6] 对于路上所有边权取和时相邻两个dist(u)抵消，只剩下第一个和最后一个

每对结点间的k短路。首先我们考虑最简洁的floyd-warshall算法，尝试着把它改成求出每两点间的前k短路的。设d[i,j,L]为从i到j，只经过1...L的前k短路长度。注意这是一个k维向量，第t维表示第t短路长度，因此在方程中和d有关的操作都是向量操作。定义基本的向量操作如下：

$\min_k\{S_1, S_2, \ldots, S_n\}S_1, S_2, \ldots S_n k$œck小值相成的向量。用两两归并的方法，时间复杂度为O(nk)。

$A + B$A和B每对分量之和的前k小值，即$\min_k\{A_i + B_j | 1 \le i, j \le k\}$。用堆实现，时间复杂度为O(klogk)。

$A* = \min_k\{0, A, 2A, 3A, \ldots\}$=A自己加自己再加自己.....由于这里只考虑权是不太小的正数的情况，所以权会越加越大，前k小有定义。注意到实际只需要算到kA，因此求k次加法和$\min_k$操作可，时间复杂度为$O(k^2 logk)$。

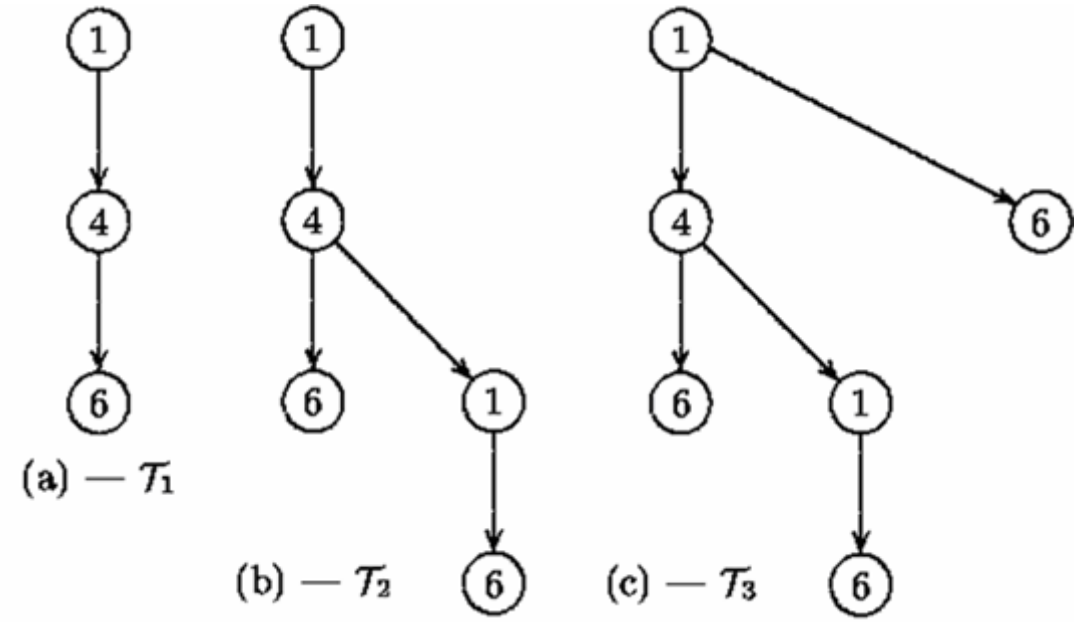
k短路和最短路有什么区别呢？除了把标量操作改成向量操作外，最重要的一点是圈的存在。从i到L后不一定立刻从L到j，有可能在L处兜圈！因此我们需要在方程中加一项表示从L出发的圈。注意我们不能用d[i,L,L]（它还没计算出来），而只能用d[i,L,L-1]，因此需要把这个圈拆成若干个只经过1...L-1的小圈之和。这正好符合上述A*的定义，因此应该是d[L,L,L-1]*。这样，状态转移方程为：

$$d[i, j, L] = \min_k\{d[i, j, L-1], d[i, L, L-1] + d[L, L, L-1] * + d[L, j, L-1]\}$$

注意到对于每个L，d[L,L,L-1]*只需要算一次，因此计算所有d[L,L,L-1]*的总时间为$O(nk^2 logk)$。另一方面，计算每个状态需要两次加法和一次$\min_k$操作，时间为O(klogk)，所有n^3个状态的总时间为$O(n^3 klogk)$，加上d[L,L,L-1]*的计算，总时间复杂度为$O(nk^2 logk + n^3 klogk)$。

两点间k短路的基本偏离算法。这里我们考虑两点间k短路，注意单源k短路并不再形成最短路树，为了简单起见我们还得指定路的终点。一类易于理解且使用范围更广的算法是**偏离算法(deviation algorithms)**，它的基本概念为**伪树(pseudo-tree)**，它是一个路径集合。把一条路径加入伪树分为两个阶段：当路径上的边在树中出现的时候顺着树边走，一旦发现路径上的边不在树中，就把剩下的路径加到树中。注意如果把路径看成字符串的话，伪树就是字符串集合的trie表示。往空树中加入三条路径{1,4,2},{1,4,6},{1,6,2}，每一步形成的伪树如下图。注意树中可以有重复结点。

每次加入新路径的时候，第一条不在原树中的弧叫做**偏离弧(deviation arc)**。最



后一个在原树中的节点叫做**偏离节点(deviation vertex)**，即“分岔点”。往树T_k加入新路径时的偏离节点通常用v_k表示，v_k后面的路径叫做偏离路径。由于在加入路径的时候，只有偏离路径才是树新增加的东西，所以它上面的点是后面将要介绍的算法最关心的。

在k短路问题中，我们用集合X来保存计算当前第k小路径可能会用到的路径，初始时X为最短路径p_1。这个思路很像用邻树法来求k小生成树，但这里的“相邻”概念来源于偏离。我们按路径长度递减的顺序把路一条一条加到X中，每次分为两个步骤：一、选偏离节点；二、找偏离路径。因此每条“邻路”由两部分组成，一是根到偏离节点的路，二是偏离路径连接而成。

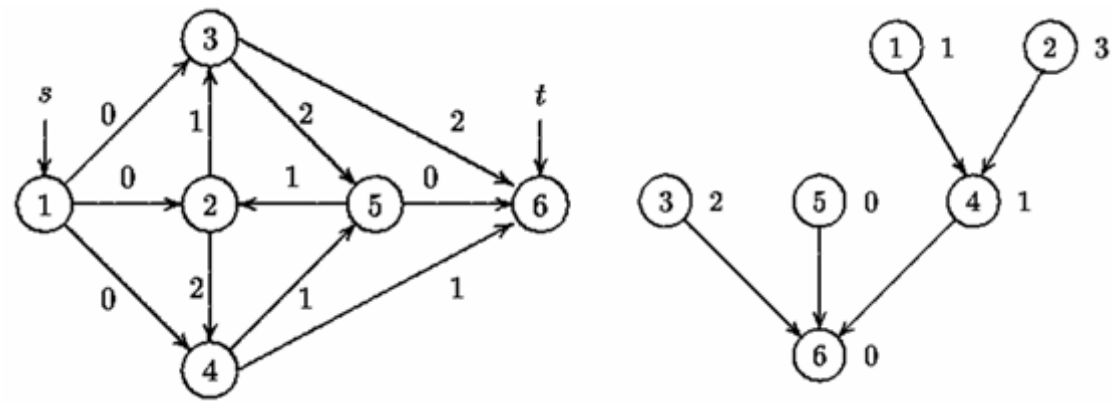
定理：用T_k表示前k条最短路形成的伪树，则T_{k+1}的偏离节点v_{k+1}的偏离路径是满足“第一条弧不是在T_k中以v_{k+1}为起点的弧”的所有路径中最小的。

证明：如果允许第一条弧是T_k中以v_{k+1}为起点的弧，那么最小的路径就是T_k中已经存在的路径。在排除了选择T_k中路径的前提下，把最小路径作为偏离路径之后理所当然的应该得到下一条最短路。

根据刚才的定理，我们只需要枚举新偏离节点v_{k+1}和第一条弧终点v(保证v不在T_k中)，则偏离路径就是v到t的最短路。另外，在枚举新的偏离节点的时候只需要从上一个偏离节点v_k的偏离路径上枚举，因为以其他顶点为偏离节点的路径早就被考虑过了。

预处理的时候，我们还需要先求出T_t*，即所有节点到t的最短路树。在算法中A(v)为以v开始的弧集合，$A_k(v)$为T_k中以v开始的弧集合，$p*(x,t)$表示在T_t*中x到t的最短路径。

下图是一个网络和对应的T_t*。



算法框架如下：

这是算法运行的前三步：

在这个算法中，需要先求一次最短路，以后每求一条的复杂度均为O(m)，因此时间复杂度为O(mlogn+km)，我们可以把求每条的时间复杂度降为O(n)，即下面将要介绍的MPS算法。

MPS算法。这里我们再次使用重加权技术，把基本偏离算法优化到O(mlogn+kn)。在刚才的算法中，选最优偏离节点的“关键步骤”耗时O(m)。要改进这个算法，我们需要优化这个瓶颈操作。为此，我们介绍**约化费用(reduced cost)**的概念。让T_t为一棵指向t的有向树，则T_t上任意节点i到t的路径唯一，我们用$d(i)$表示这条唯一路径的权和，则我们用：$c'(i,j) = d(j) - d(i) + c(i,j)$来表示弧$(i,j)$在树$T_t$的简化费用，则有如下重要定理(请读者根据定义证明这个定理)：

定理：设其中$c(p)$表示路径的原始费用和，c’(p)表示路径上所有弧的约化费用和，则

- $c(p) = c(q)$ 当且仅当c’$(p) = c$’(q)
- $c(p)$ ¡ $c(q)$ 当且仅当c’(p) ¡ c’(q)

这个定理告诉我们，按照”原始费用和”给路径排序与按照” 约化费用和”排序的结果是一致的。在计算K短路时可以用约化费用完全代替原始费用。

```
Compute T_t^*
p_1 ← shortest path from s to t                    /*  p_1 is a path of T_t^*  */
k ← 1
X ← {p_k}
T_k ← {p_k}
While k < K and X ≠ ∅
do begin
        X ← X − {p_k}
        v_k ← deviation node of p_k
        for each node v ∈ p^k_{v_k t}
        do begin
                if A(v) − A_{T_k}(v) ≠ ∅
                then begin
                        Compute the arc (v, x) such that c_vx + c(p*_xt)
                            is minimized over A(v) − A_{T_k}(v)
                            /*  p*_xt is a path of T_t^*  */
                        q ⟵ p^k_sv ◇ ⟨v, (v, x), x⟩ ◇ p*_xt
                        X ⟵ X ∪ {q}
                        q_vt ⟵ ⟨v, (v, x), x⟩ ◇ p*_xt
                        T_k ⟵ T_k ∪ {q_vt}
                           /*  In such a way that p^k_sv ◇ q_vt is a path of T_k  */
                end
        end
        k ← k + 1
        p_k ← shortest path in X
end
```

既然是等价的，那么用约化费用有什么好处呢？道理蕴涵在下面的定理中：

定理：对于最短树T_t*，图G中的任意弧(i,j)满足$c'(i,j)¿=0$；其中T_t*的弧(i,j)满足$c'(i,j)=0$。

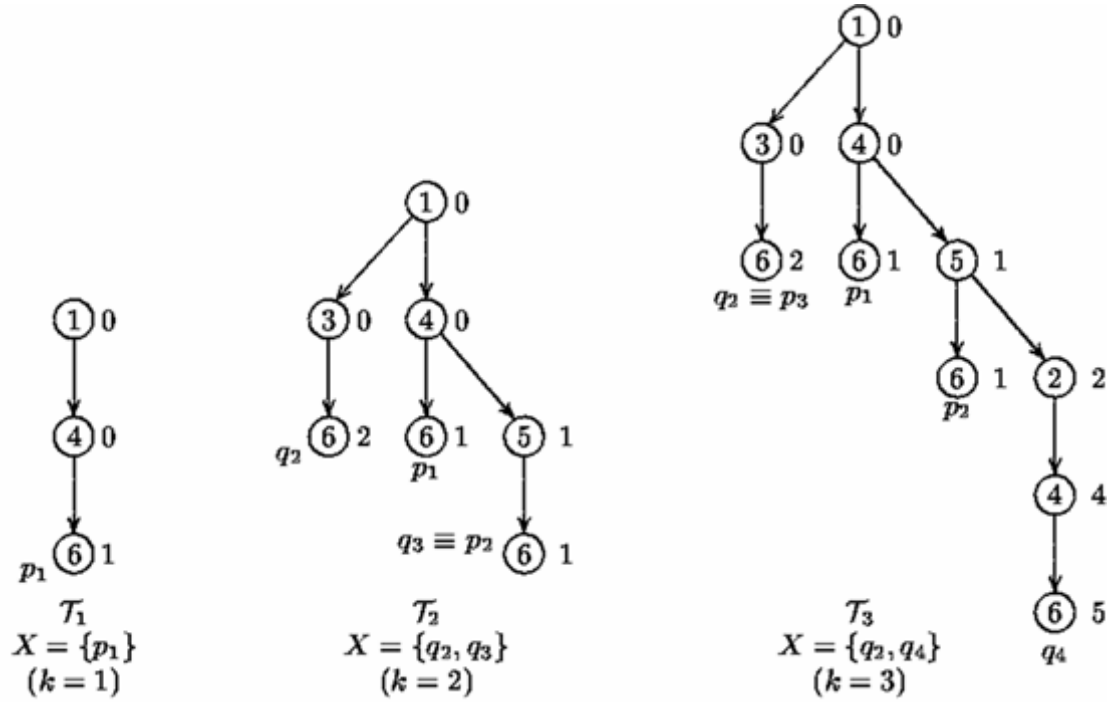
这个定理的作用可能不是很明显，我们不妨先再看刚才的算法。算法的关键步骤需要考查以v开始的所有弧，因此最坏情况下一共需要考察O(m)条弧。

根据两个定理，我们让$c(v,x)+c(p^*(x,t))$最小，等价于让$c'(v,x)+c'(p^*(x,t))$最小。由于$p^*(x,t)$上的弧都在T_t*中，因此$c'(p^*(x,t))=0$，即：我们只需要选择最小的c'(v,x)。注意约化费用值c'(v,x)已经预先算好并不会改变，因此将它们排序后可以直接找到最小值。

这样，MPS算法的核心是：

一、预处理求出T_t*和所有弧的约化费用。

二、在图的前向星表示中重排弧，使得每个点出发的所有弧按约化费用值从小到



大排序。

三、执行基本偏离算法，不同之处在于每次直接选择可以选择的第一条弧。

算法框架如下：

由于瓶颈操作从O(m)降到O(n)，算法的总时间复杂度从O(mlogn+km)降到了O(mlogn+kn)。

7.3.4 小结

本节花了不多的篇幅介绍了最短路相关知识，算法上包括标号设定算法和标号修正算法，以及偏离算法，问题上讨论了SSSP、APSP和结点可重的k短路。和上一节一样，本节的所有算法都应该熟练掌握，而相关结论的证明也是很基本的。

本节的算法和程序列表如下：

7.4 网络流问题

本节讨论最大流问题和最小费用流问题，并给出最大流问题的若干的多项式算法和最小费用流问题的经典算法。最大流问题有很多应用，本节也做简单介绍。

```
Compute T_t^*
Compute c̄_ij, for any arc (i,j) ∈ A
Rearrange the set of arcs of (N,A) in the sorted forward star form
        /*          For the computed reduced costs          */
p_1 ← shortest path from s to t                  /*   p_1 is a path of T_t^*   */
k ← 1
X ← {p_k}
T_k ← {p_k}
While k < K and X ≠ ∅
do begin
        X ← X − {p_k}
        v_k ← deviation node of p_k
        for each node v ∈ p^k_{v_k t}
        do begin
                if A(v) − A_{T_k}(v) ≠ ∅
                then begin
                        (v,x) ← first arc in the set A(v) − A_{T_k}(v)
                        q ⟵ p^k_{sv} ◇ ⟨v,(v,x),x⟩ ◇ p^*_{xt}
                        X ⟵ X ∪ {q}
                        q_{vt} ⟵ ⟨v,(v,x),x⟩ ◇ p^*_{xt}
                        T_k ⟵ T_k ∪ {q_{vt}}
                           /*  In such a way that p^k_{sv} ◇ q_{vt} is a path of T_k  */
                end
        end
        k ← k + 1
        p_k ← shortest path in X
end
```

7.4.1 最大流问题

本节讨论最大流问题的概念、特点和基本结论。一个**流网络(flow network)** G=(V,E)是一个有向图，每条边(u,v)有一个非负容量(capacity)$c(u,v) \geq 0$，对于不在E中的(u,v)，规定$c(u,v) = 0$。有两个特殊结点：**源点(source)**s和**汇点(sink)t**。假设对于任意其他点v，存在通路s-¿v-¿t。

可以把这样的流网络看作是运送物资的交通网络，我们希望从s往t运送物品。由于对于任意点v都存在s-¿v-¿t通路，因此不用担心物品“运丢”。但这个过程仍然是有限制的。定义在每条边上运送物品的数量$f(u,v)$，则：

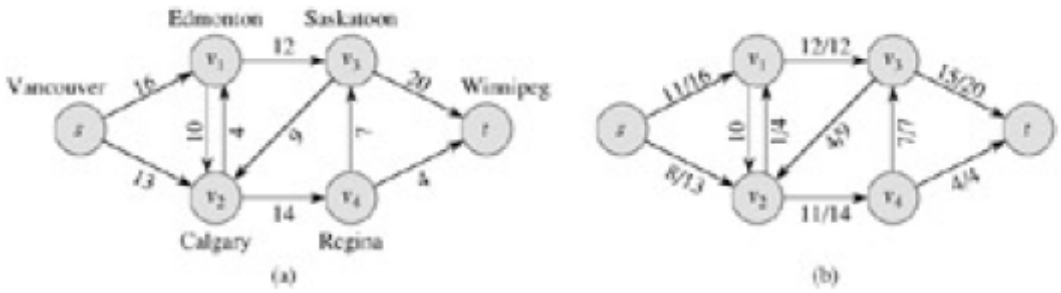
容量限制：$f(u,v) \leq c(u,v)$

对称性：$f(u,v) = -f(v,u)$

算法	程序	备注
dijkstra算法	dijkstra.cpp	基本的O(n2)实现和二叉堆的O(mlogn)实现
dial算法	dial.cpp	基本O(nC)实现和基数堆的O(m+nlogC)实现。
bellman-ford算法	bellman-ford.cpp	基本的队列实现、Yen修正算法和dequeue实现。如果失败，算法找出负圈。
floyd-warshall算法	floyd.cpp	APSP的floyd-warshall算法，时间复杂度为$O(n^3)$。
johnson算法	johnson.cpp	APSP的johnson算法，调用dijkstra算法，时间复杂度为O(nm+nS(n,m))，其中S(n,m)和dijkstra算法的具体实现有关。
每两点的结点可重k短路	k-APSP.cpp	修改的floyd-warshall算法，时间复杂度为$O(nk^2\log k+ n^3k\log k)$。
结点可重k短路偏离算法	k-deviation.cpp	基本偏离算法和MPS算法，时间复杂度分别为O(S(n,m)+km)和O(S(n,m)+kn)。

收支平衡：对于不是s或t的其他点u，有 $\sum_{(u,v)\in E} f(u,v) = 0$

定义整个网络的流量f为从s流出的流量，它也等于从t收集到的流量。本节讨论的最大流问题就是要求出流量最大的f。在物资运送问题中，它对应着把最多物体从s运送到t的方案。



为了方便，我们给两个结点集X和Y定义流量$f(X,Y) = \sum_{x\in X}\sum_{y\in Y} f(x,y)$，则以f为流函数的流网络中，以下三个等式成立：

$$f(X,X) = 0$$

$$f(X,Y) = -f(X,Y)$$

若X和Y不相交则$f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$

我们还可以定义流的加法和标量积（注意容量限制），并证明可行流集的凸性：如果f_1和f_2都是可行流，则对于所有$0 \leq \alpha \leq 1$，流$\alpha f_1 + (1-\alpha) f_2$也是可行的。

理论上弧的两个方向可以同时有流量，但实际上这是没有必要的，我们可以用所谓的**流取消(flow cancelling)**方法只保留其中的一边。这一点在理解残量网络中很有用。

残量网络(residual network)。我们常常使用残量网络来设计算法，它的基本思想是：把已有的流看作网络本身的一部分，把有流的网络转化成无流的网络。根据这个想法，定义弧(u,v)的残量容量为$c_f(u,v) = c(u,v) - f(u,v)$，它的含义是该弧的最大流增量。注意流量为负时，残量容量比原始容量还要大！可以这样理解：在取消反向容量以后还可以增加正向流量，因此流量增量（而不是终值）可以大于原始容量。这并不违反容量限制。

所有大于0的残量弧所构成的网络称为残量网络，由于每条弧最多可以从两边增加流量，因此$|E_f| \leq 2|E|$。残量网络的每条s-t路称为增广路，因为沿着它进行增广可以增加网络的流量。什么叫增广？定义增广路总残量容量最小值d为该路的残量容量，则把路上每条弧的流量增加d，反向流量减少d（由对称性）后，所得的流仍然是可行的（请读者自行验证流的三个性质），且流量增加了d。残量网络的好处是我们不用考虑流和真实容量。它是一个通用技术，本节中的所有算法都直接在残量网络中进行。

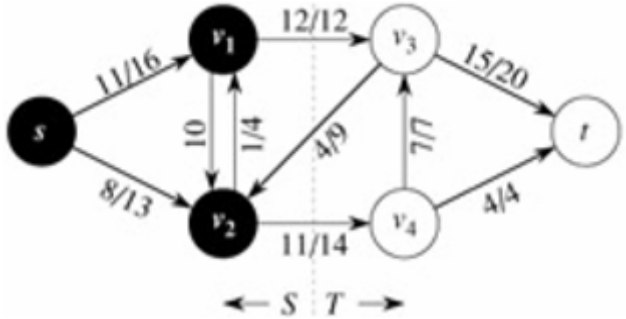
如果残量网络中有增广路，则当前流一定不是最大流。反过来呢？如果残量网络中没有没有增广路，则当前流一定是最大流吗？答案是肯定的，我们有以下的：

增广路定理：流量f是最大流当且仅当残量网络中不存在可增广路。

为了证明这个定理，我们首先引入切割的概念。这个概念本身也有重要价值，如我们即将介绍的最小切割最大流定理。

网络中的切割。网络G=(V,E)的**s-t切割(s-t cut)**是把V划分成S和T=V-S两部分，使得s在S中，t在T中。使用前面定义的隐式求和记号，可以简单的定义切割(S,T)的净流量(net flow)为f(S,T)，容量为c(S,T)。下图的割流量为19（注意有反向流），容量为26。

记最小切割(minimum cut)是使c(S,T)最小的切割，则有如下重要的：



定理：对于任意一个切割，$f(S,T)=|f|$，即切割的流量等于网络的流量。

证明：$f(S,T)=f(S,V)-f(S,S)=f(S,V)=f(s,V)+f(S-\{s\},V)=f(s,V)$，其中最后一步是根据流量平衡。也可以直观想象：既然是切割，所有流最终都是要流经切割中的某弧的，因此切割上的流量都来自s点，自然有$f(S,T)=|f|$。

由定理可知，$|f|$不会超过任意一个切割的容量（以后我们将看到，这是线性规划中的弱对偶性），因此任意流量不会超过最小切割的容量。最小切割的容量C是固定的，因此如果某流的流量等于C，则它一定是最大流。可是一定存在这样的流吗？答案是肯定的，即：

最小切割最大流定理：最大流等于最小切割的容量。

把它和前面介绍的增广路定理合在一起，我们证明如下三个条件等价：

1. f是最大流

2. 残量网络中无可增广路

3. 存在某个切割(S,T)，|f|=c(S,T)

证明：我们依次证明1–¿2，2–¿3和3–¿1。

1–¿2。反证法。若存在，可沿它增广得到更大流。

2–¿3。此时在残量网络中不存在s-t通路，即(S,T)中所有弧满载（否则残量网络中会有更多弧存在而导致s-¿t有通路）。定义S为s可达结点集，T为可达t的结点集，则$|f|=c(S,T)$。

3–¿1。由弱对偶性可证。

这样，我们得到了最大流的两个等价命题，其中从增广路定理可以直接得到求解最大流的增广路系列算法，而2–¿3的证明过程给出了从最大流构造最小切割的过程，它只需要用O(m)时间进行BFS以求出s的可达结点集S，令T=V-S即可。

下面我们考虑最大流问题的若干变形。

多源多汇问题。对于源和汇有多个的情形，可以增加附加源s'和附加汇t'，对于所有源点u，增加弧(s',u)，容量$c(s',u)=\sum_{(u,v)\in E}c(u,v)$；对于所有汇v，增加弧(v,t')，容量$c(v,t')=\sum_{(u,v)\in E}c(u,v)$，这样做的直观意义是明显的，读者可以仔细体会。

顶点容量。可以限制流经一个点u的流量上限制，称为顶点容量。只需要把点u拆成弧(u,u')，容量为u的点容量即可。

有环转化为无环。设u出发的弧容量和为sum(u)，所有弧容量和为X，则可以把每个点u拆成两个点u和u*，原始弧(u,v)变为(u,v*)，增加新弧$c(u,u*)=X+1$，附加源s'和汇t'，增加弧$c(s',u)=c(u*,t')=sum(u)$，去掉原来的所有弧。

无向图变为有向图。把每条弧拆成容量相同的两条反向弧即可。

满足供需关系的可行流。每个点u有权w(u)，正数表示供给(supply)，负数表示需求(demand)，收支平衡条件被修改为“出流-入流=w(u)”。显然所有权和为零才有可能有解。构造满足供需关系的可行流并不困难，只需要把正权点看成是源，负权点看成是汇，则问题转化为了多源多汇问题。

有下界的网络流。我们可以个每条弧增加一个流下界b(u,v)，则并不是所有网络都有可行流（对比：没有下界的网络中至少零流是可行的）。所以这类问题一般需要两步走，一是寻找可行流，二是把可行流转化为最大流。两步都可以转化为只有流量上限的传统最大流问题。

假设已经找到了可行流，则令残量容量$c_f(i,j)=(c(i,j)-f(i,j))+(f(j,i)-b(j,i))$，它既考虑了在正向弧中增加流量时不能超过容量的限制，也考虑了在反向弧中增加流量时不能小于流量下界。这样，我们得到了有下界情形的残量网络，任何一个传统最大流算法都可以在上面工作。顺便说一句，如果定义切割(S,T)的容量为C(S,T)-B(T,S)，则下界的网络中有所谓的**一般最小切割最大流定理(generalized max-flow min-cut theorem)**。

如何找到初始可行流呢？如果我们增加容量无限大的弧t-s，则问题转化为了求网络中的循环可行流。在这个网络中没有源点没有汇点，因此称为**循环流问题(circulation problem)**。在循环流中，所有结点都是收支平衡的，满足容量限制，即：

$$\sum_{(i,j)\in E} f(i,j) - \sum_{(j,i)\in E} f(j,i) = 0 \quad \forall i \in V$$
$$b(i,j) \le f(i,j) \le c(i,j) \quad \forall (i,j) \in E$$

令$f'(i,j) = f(i,j) - b(i,j)$，则上述条件变成：

$$\sum_{(i,j)\in E} f'(i,j) - \sum_{(j,i)\in E} f'(j,i) = w(i) \quad \forall i \in V$$
$$0 \le f(i,j) \le c(i,j) - b(i,j) \quad \forall (i,j) \in E$$

其中w(i)为前面所提到的供需函数，即：

$$w(i) = \sum_{(j,i)\in E} b(j,i) - \sum_{(i,j)\in E} b(i,j)$$

这样，求可行循环流的问题转化为了可行供需流。

有下界网络中最大流的独立环。用有下界网络流建模实际问题时需要特别小心。一个典型的例子是用有下界网络最大流来“求解”Hamilton圈的问题的“算法”：设每个点的流量上下界均为1（用刚才提到的方法，再用拆点法转化为边流量上下限），图中的无向边的容量均为无穷大，求网络的可行循环流。这个算法看似没有问题（上下界迫使每个点恰好走一次），但实际上肯定是错的：求最大流是有多项式算法的，如果这个算法是正确的则Hamilton圈的判定问题多项式可解。而事实上Hamilton圈的判定问题是NP完全的。问题出在哪里呢？

虽然网络流的初衷是从s到t的物资运送问题，但可行流的三个条件并没有规定有流量的弧可以从s点到达。在无下界的情况下根据已有的最大流算法得到的流一定是从s“流过去”的，但在有下界的情况下由于增加了弧t-s，计算结束拆去弧t-s后原网络中的流可能形成若干个独立环。也就是说，上述算法求出来的结果可能不是一条圈，而是多个圈的并。好在很多实际问题中并不要求所有有流弧是连通的，所以有下界网络流的应用仍然非常广泛。

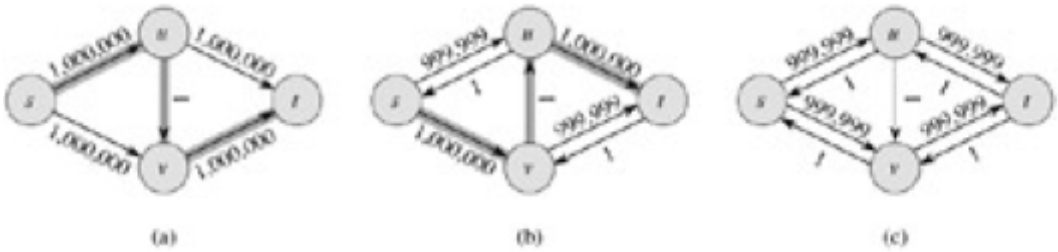
7.4.2 最大流问题的增广路算法

本节讨论最大流问题的增广路系列算法，包括Ford-Fulkerson算法、Edmonds-Karp算法等。

用增广路定理很容易证明下面的

整流定理：如果网络中所有容量是整数，则最大流也是整数。

不难发现，如果采用增广路算法，不管每次用什么增广路，流量至少增加1。设最终流量为V则任意的增广路算法最多迭代V次。这个结果并不令人满意，因为有可能V很大。这里有一个例子。两次增广后流量只增加了1。事实上最大流为1,000,000，需要增广1,000,000次才行。还有类似的例子，当容量为无理数时（本节不考虑这样的边权）可能永远无法收敛。



如果不盲目的选择增广路，情况会有所好转。

Edmonds-Karp算法。每次用BFS沿着残量网络中最短（包含弧最少）s-t路增广。

这个看起来很平常的算法，运行效率比任意增广的方法好得多。事实上，它的增广次数和边权无关，为O(nm)，如果每次都用BFS来找增广路，则总时间复杂度为$O(nm^2)$。这个算法也称为Edmonds-Karp算法。使用距离标号可以把找最短增广路的时间从O(m)减到O(n)，从而得到$O(n^2m)$的算法。虽然这个算法并不是很高效，但由于它的重要地位，这里用一点篇幅叙述一下。为此，我们首先介绍距离标号的概念。

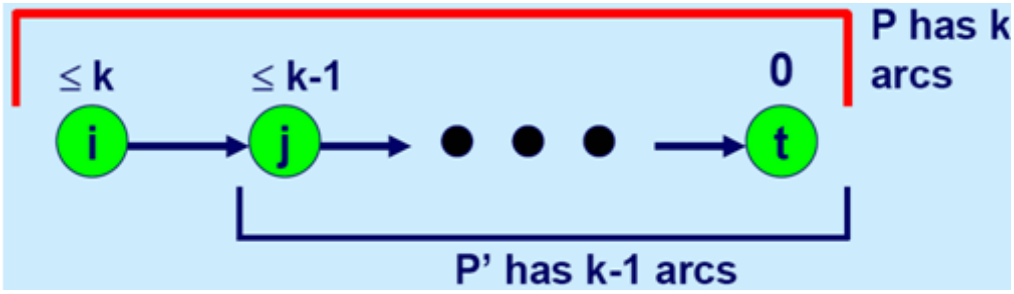
距离标号(distance label)。距离表示是专门定义在残量网络中的，从结点编号i映射到非负整数d(i)的函数。如果在残量网络G_f中d满足以下性质，我们说该距离标号是有效的：

$$\begin{aligned} &d(t) = 0 \\ &d(i) \le d(j) + 1 \quad \forall (i,j) \in G_f \end{aligned}$$

换句话说，顺着弧走，距离标号最多减小1，可以不变，也可以增加。

距离标号的重要特点是：d(i)是残量网络中i-t最短路长度的**下界(lower bound)**（考虑任意一条i-t路，用数学归纳法易证）。因此如果$d(s) \ge n$，残量网络中一定没有s-t路。特别的，如果d(i)等于i-t最短路长度，我们说该距离标号是精确的。通过反向BFS可以在O(m)时间内求出所有结点的精确距离编号。

如果一条弧(i,j)满足$d(i) = d(j) + 1$，我们说该弧是**允许弧(admissible arc)**，全由允许弧组成的s-t路径称为**允许路径(admissible path)**。允许路径的重要性质是：任



何s-t允许路径都是s-t最短路（沿着弧走每次距离标号最多减小1。每次都减小1的走法当时是最近的）。

这样，寻找最短增广路实际上是寻找残量网络中的任意一条允许路径。

最短增广路算法。利用距离标号，我们得到了另外一个最短增广路算法。

algorithm shortest_augment_path;
begin
x := 0;
obtain the exact distance labels d(i);
i := s;
while d(s) ¡ n do
begin
if i has an admissible arc (i,j) then
begin
pred(j) := i; i := j; // advance;
if i = t then augment and set i = s;
end;
else d(i) := min{d(j)+1 | (i,j)∈G_f} and i := pred(i); // retreat; note pred(s)=s
end;

其中augment过程利用pred数组从t回到s并记录下所有弧中的最小残量容量d，然后给此路增广d单位的流量。

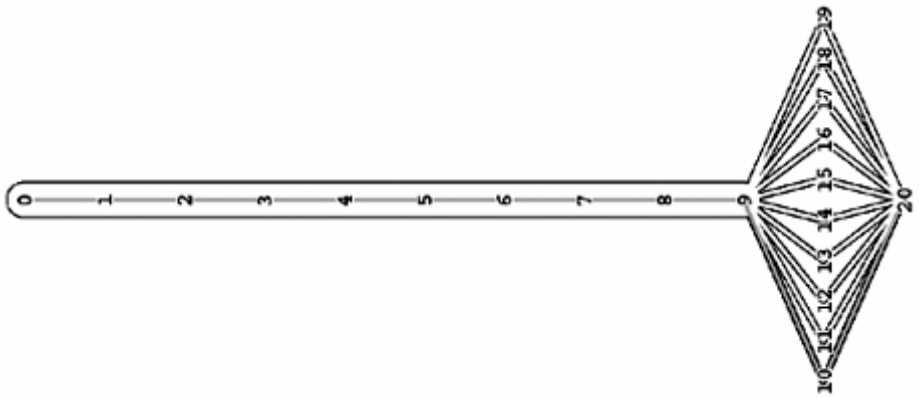
我们把每个点出发的弧按任意顺序排列，但在算法执行过程中不再改变。可以证明：如果一条弧(i,j)是非允许弧，则只要d(i)不变，它不可能再次变成允许弧。这样，当扫描到i出发的最后一条弧后还是没有发现允许弧，需要对i进行重新标号为min{d(j)+1|(i,j)∈G_f}并设当前弧为第一条弧。这样，只需要记录每个点出发检查到

的**当前弧(current arc)**，可以证明算法的总时间为O(n^2m)。

这个算法有一个很实用的改进。算法的终止条件为$d(s) \geq n$，但其实上达到这个条件之前即使已经求出了最大流，算法还会继续迭代很多次，纯粹修改距离标号而不进行任何增广。其实我们只需要记录距离标号等于k的结点个数c_k，在把结点i的距离标号从p重标号为q时把c_p减1，c_q加1，并判断是否有$c_p = 0$。如果是，则算法终止。这是因为距离标号大于p的点不可能一步到达距离标号小于p的点（因为沿着弧走距离标号最多减小1），残量网络不连通，当然也就没有增广路了。

7.4.3 最大流问题的预流推进算法

本节讨论最大流问题的预流推进算法。我们首先考虑一个例子。



感觉上增广路算法算法执行了很多重复操作，增广的方式看起来比较机械。试想如果从0点灌很多水，水会怎样流向20呢？当时是同时从0流向9，到点9以后再分岔，分别流到20。这样的想法就是预流推进法的动机。

预流推进算法(push-relabel algorithms)和增广路算法不一样, 它并不检查整个残量网络, 而是每次考虑一个结点, 在残量网络中只检查它的邻居。算法执行过程中并不始终保持流量平衡条件, 因此中间结果只是**预流(preflow)**,满足对称性和容量限制, 但只满足宽松的流量平衡式f(V,u)¿=0 (u为不是s的任意点), 即：流进每个非s点的净流非负（蓄势待发, 等着流出）。记e(u)=f(V,u), 即点u的盈余。盈余大于0的结点称为**活跃结点(active node)**。

形象的比喻有助于理解预流推进算法。想象一个自上而下的水流，有向边是管道，点是管道连接处，每个点有一个蓄水池用来保存这个点处临时储存的水（赢

余e(u)）。每个点的高度可变，随着算法进行逐渐升高。

水往低处流。水从一个结点流到比它低的其他结点，直到所有连接它的管道满载或者水流干，对应于算法中的push操作——push(i)的结果要么是i出发的弧均满载（水太多），要么i的赢余变为0（水太少）；如果有水却流不动，则需要抬高水位，让这些水可以继续往下流，对应于算法中的relabel过程——它总是增加距离标号，规则和最短路算法一样。

通用预流推进算法(generic preflow-push algorithm)。水源高度固定为n, 汇的高度固定为0, 其他点的高度初始为0, 随着时间推移慢慢增加。算法首先尽量多的从水源把水"推"出来, 即让s出发的所有弧满载，然后继续让水一步步流到t。每次当水流入某个结点u后, 首先进入该结点的蓄水池（即修改赢余e(u)），然后：

- 如果u出发有到达更低点v的非饱和弧, 把尽量多的水推入v (push过程)。
- 如果u出发有非饱和弧，但末端都与u水平甚至更高，则把u抬高(relabel过程)。

最终, 当所有能到t的水都到t了, 这时如果还有盈余结点就只能继续抬高该结点，直到抬得比s更高，把蓄水池里剩下的水倒回s。

预流推进算法使用和最短路算法几乎一样的距离函数（除了在预流推进算法中我们规定$d(s) = n$），并规定push操作只沿允许弧进行。这是很自然的，因为距离标号反映的就是离t的距离，即结点高度。下面是算法框架，算法首先求出精确距离标号，设置s出发的弧全部满载（注意同时需要设置这些弧的反向弧流量）并设$d(s) = n$，然后不停的执行push或relabel操作，直到网络中不存在活跃结点，即：

algorithm preflow-push;
begin
x := 0;
compute the exact distance labels d(i);
x_{sj} := u_{sj} for each arc (s,j)∈E;
d(s) := n;
while the network contains an active node i do
begin
if there is an admissible arc (i,j) then // push
push d = min{e(i), c_f(i,j)} units of flow from node i to node j

else d(i) = min{d(j)+1 | (i,j)∈G_f}; // relabel

end;

end;

注意在push操作中，d=c_f(i,j)时称为饱和推进（水太多），其他情况为非饱和推进（水不够）。非饱和推进总是让结点赢余变为0。如果需要relabel，则u点一定有赢余，因此它出发一定有弧（否则赢余哪里来的），大不了把这些水倒回去！所以活跃结点要么可以push要么可以relabel。另外规定s和t的赢余始终为0，它们永远都不会变成活跃结点。

正确性。算法的终止条件是无赢余结点，此时伪流已经变成了真正的可行流了。可这个可行流是最大流吗？为了证明这一点，我们需要考察一条刚才提到过的规定：规定d(s)=n。这个看似随意的规定实际上是在迫使残量网络中不含s-t路，因为若存在简单路，设长度为k，则k¡=n，在路上持续使用距离标号性质得$d(s) \leq d(t) + k - 1 = k - 1 < n$，与d(s)=n矛盾。因此一旦算法终止，残量网络一定不含s-t路，此时的流一定是最大流。

可以证明：如果活跃结点列表用双向链表实现，则算法的时间复杂度为O(n^2m)。

通用算法非常随意，活跃结点和允许弧的选取都是任意的。和最短路算法一样，把二者的选择方法确定化，将得到效率更高的算法。不管是哪种方法，我们需要引入**结点检查(node examination)**的概念。

FIFO-预流推进。用队列保存活跃结点集，每次新得到队列中不存在的活跃结点就放到队尾。则时间复杂度降为O(n^3)。

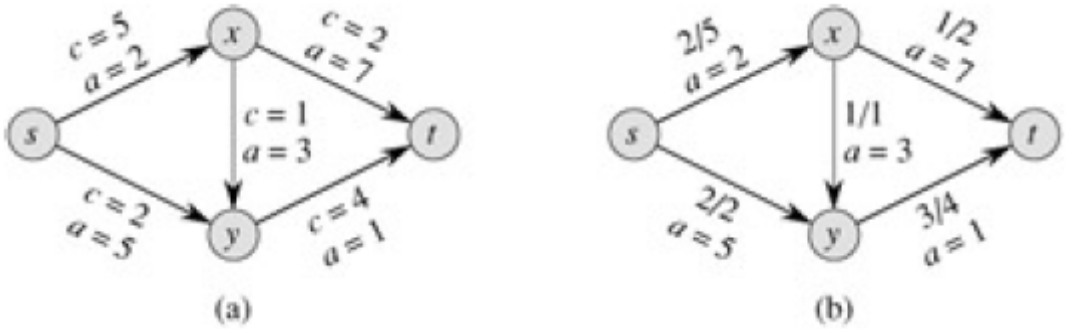
最高标号预流推进。总是选择距离标号大的结点。它的直观想法是如果先从矮的流，流完以后如果高处还有水，还得再流一遍。还不如先从高的流到矮的，然后两股水汇集到一起流。这样的算法用LIST(k)保留距离标号为k的结点列表。可以证明任意结点的距离标号不超过2n，因此一共只需要2n+1个表。算法记录一个level值，即最高标号的上限，每次增加距离标号时更新level值，然后每次只需要依次检查LIST(level)，LIST(leve-1)...即可，找到非空表LIST(p) 后更新level=p。可以证明寻找最高标号的操作不会称为瓶颈操作，算法的总时间复杂度为$O(n^2m^{1/2})$。

7.4.4 最小费用最大流问题

费用流问题是一个一般性问题，最短路和最大流问题都是它的特例。费用流问题

比最短路和最大流要困难的多，目前已知的多项式算法不仅比较复杂而且实际效果并不是特别理想。本节介绍最容易理解的最小费用路算法，而把实际效果不错的网络单纯形法留本章末。

最小费用流问题(Minimum Cost Flow Problem)。给流网络上增加费用函数$W = \sum w(e)f(e)$，其中w(e)为弧上的费用函数。假设源点流量为f，则最小费用流问题是对任意流量f求出相应的最小费用流。下图是一个费用流问题的实例，左图是流网络，右图是一个流。最小费用流问题往往需要指定流量。一个特殊形式是当流量最大时的最小费用流，称为最小费用最大流。



残量网络。由于增加了费用，我们需要对原来的残量网络的概念加以修改。设原始网络$N = (G, c, s, t)$，f为流量w的最小费用流，则可以和最大流问题类似的方法定义关于f的残量网络$N' = (G', c', s, t)$，其中正向非饱和弧e的费用仍然为w(e)，反向非零流量弧e的费用为-w(e)。

消圈算法(cycle canceling algorithms)。我们从最小费用流的充要条件开始学习本节的第一个算法。为此，我们首先证明一个更基本的定理：

定理：可行流x为最小费用流的充要条件是残量网络中不存在负费用增广圈。

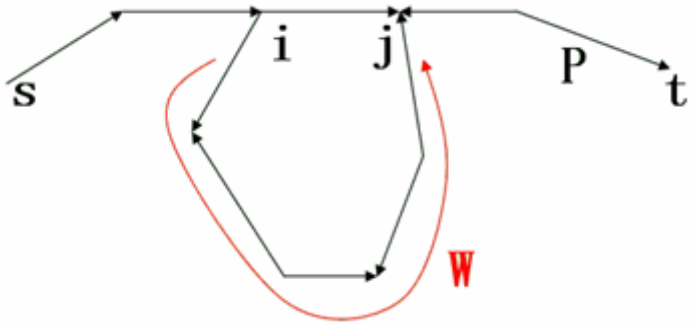
证明：必要性是显然，用反证法证明充分性。假设存在不同的可行流x_0使得它的费用比x更低，则它们的差x_1=x_0-x是网络中的负费用循环流。既然是循环流，x_1一定可以分解为至多m个非零圈流之和，则它们中至少有一个圈是负费用的（有限个非负费用之和仍是非负的），这个圈就是题目中非负增广圈，矛盾。

显然直接用这个定理可以得到经典的消圈算法，它用Klein(1967)等人提出。若利用bellman-ford找负圈则时间复杂度较高；若每次用O(nm)时间找平均费用最小的增广圈进行增广，则对于整数权来说最多迭代O(nmlog(nC))次，而对于任意实数权来说最多迭代$O(nm^2 \log n)$次。显然，虽然是多项式时间复杂度，但阶太高，仍然不实用。

连续最短路算法(successive shortest-path algorithm)。该算法由Jewell(1958), Iri(1960), Busacker和Gowen(1961)等人独立提出，是一种类似于最大流的增广路算法，但每次沿着所谓的最小费用路进行增广。算法是迭代的式的，因此当流量为v时的$m(J$就是流量为v的最小费用流。这一点有时非常有用。

最小费用增广路就是残量网络中最短的s-t路。由于残量网络中的费用有正有负，所以最短路只能用bellman-ford 算法计算，每次需要O(nm)时间。假设增广k次，则时间复杂度为$O(kn^3)$，对稀疏图为O(knm)。

算法的正确性。我们只需要证明每次增广后没有负费用圈。假设沿路径P增广后出现了负费用圈W，则由于P之外的其他弧并没有受到影响，W一定与P有公共部分，设为路径i–¿j。现在我们把增广路改一下，从S到i以后不再顺着原来的P走到j，而是顺着W走到j，然后继续沿着P从j走到t。由于W是负费用的，得到的新增广路费用一定比P小，与P是最小费用路矛盾。



改进算法。刚才算法的瓶颈是bellman-ford算法。如果网络中的费用是非负的，就可以用dijkstra算法计算s-t的最短路，每次迭代时间降为$O(n^2)$。是否可以把网络进行改造使得残量网络中总是不存在负费用呢？答案是肯定的。

Edmonds和Karp于1972年提出了一个技巧，解决了这一问题。假设增广前从s到u的距离为d(u)，增广后的费用函数为w(e)，对于弧e=uv定义一个新的权值$w^*(e) = w(e) + d(u) - d(v)$。容易知道对于任意s-x的路X有$w^*(X) = w(X) - d(x)$，因此对于权函数$w(e)$和$w^*(e)$，从s出发的单源权函数完全一样。因此可以把$w^*(e)$作为权函数进行计算，然后用$d'(x) = d^*(x) + d(x)$计算出真正的距离d'(x)。更妙的是：$w^*$是非负的！因为$w^*(e) = w(e) + d(u) - d(v) < 0 \Rightarrow d(v) > d(u) + w(e)$，这和d(u)是最短距离矛盾！

这样，我们把最小费用流问题的算法时间复杂度从$O(kn^3)$降到了$O(kn^2)$，对于稀疏图是O(kmlogn)。对于一些特殊情况（如后面讨论的二分图最大权匹配），流量k有

多项式界的，则该算法是多项式时间的。

7.4.5 小结

本节讨论了网络流相关问题，包括最大流的增广路算法和预流推进算法，最小费用流的连续最短路算法，也包括网络流的一些基本应用。这些算法并不是很容易理解，建议读者先熟悉它们的思想和高层描述，然后逐步学习细节。在这些算法中，有三个概念最为重要：一是残量网络，二是距离标号和允许网络，三是约化费用，读者应仔细体会。

本节的算法和程序列表如下：

算法	程序	备注
Edmonds算法	edmonds.cpp	每次用BFS求最短增广路，时间复杂度为$O(nm^2)$。算法顺便求出一个最小割。
最短增广路算法	augment.cpp	用距离标号实现的最短增广路算法，时间复杂度为$O(n^2m)$。
预流推进算法	preflow-push.cpp	包括一般算法、FIFO实现和最高标号实现，时间复杂度分别为$O(n^2m)$，$O(n^3)$和$O(n^2m^{1/2})$。
连续最短路算法	successive.cpp	最小费用流的连续最短路算法，包括基本实现和改进实现，时间复杂度分别为O(knm)和$O(k*\min\{n^2,m\log n\})$。

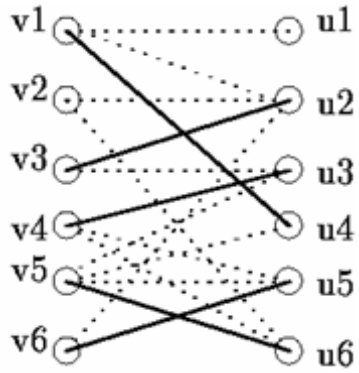
7.5 匹配问题

本节考虑二分图匹配问题，包括最大基数匹配和最大权匹配以及稳定婚姻问题。另一些应用如最小瓶颈匹配等也进行简单讨论。

7.5.1 二分图匹配

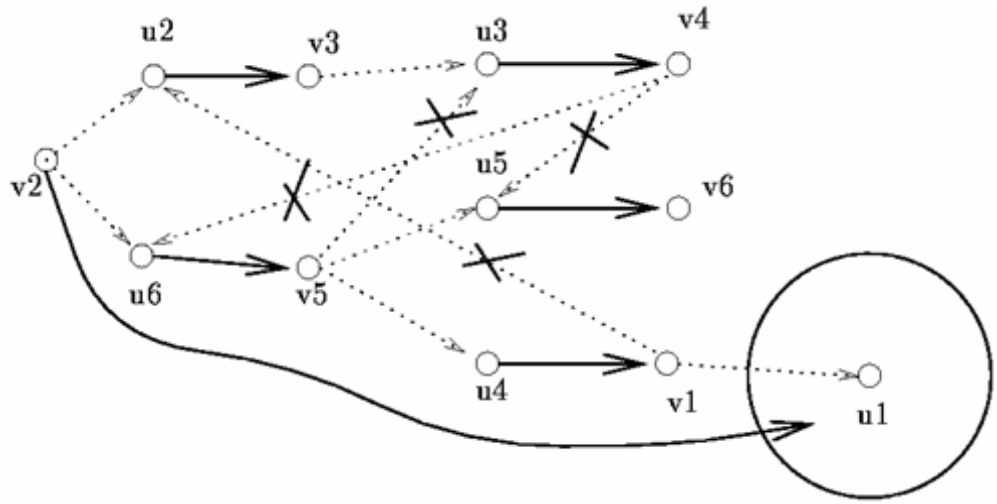
本节讨论二分图最大基数匹配：给二分图G，求最大的边集，使得任何两条边不相邻。注意到这等价于求多源多汇网络中的最大流，所以也可以用增广路算法求解。

Edmonds算法。Edmonds于1965年给出了第一个二分图最大基数匹配算法。算法非常简单：每次用BFS寻找一条增广路，则匹配数加1。由于匹配数不超过n，因此算法



最多执行n步，总时间复杂度为O(nm)。注意到它实际上对应最大流问题的Edmonds算法，它也是用BFS求最短增广路。

下面给出在上图中用BFS寻找v_2到u_1的增广路的例子，读者应习惯于把匹配边看成一个点，并忽略所有Y结点，因为到达Y结点之后v如果不是未盖点则必须走到一个固定的点，即v的匹配点。



为了叙述方便，我们把这棵BFS树称为**匈牙利树(hungarian tree)**，它是从所有未盖点出发生长出的树。在Edmonds算法中我们不需要这个概念，但在Hopcroft-Karp算法和最大权匹配的KM算法中需要用到它。

Hopcroft-Karp算法。Hopcroft算法是目前已知的对于稀疏图最有效的二分图匹配算法，它并不复杂，所以这里介绍一下。算法的每一步是寻找若干条**结点不相交(vertex disjoint)**的最短增广路，然后同时增广。可以证明如果每次寻找的最短增广路集是极大的（即无法再找出一条和它们都不相交的最短增广路），则算法一共只

需执行$O(\sqrt{n})$步。只要每步使用O(m)时间找到极大最短增广路集，则算法总时间复杂度为$O(\sqrt{n}m)$。如果寻找极大可增广路集呢？我们可以从所有未盖点开始用BFS求出距离标号，即设未盖点的距离标号为0并放入队列，每次从队列中取出元素u（它一定是X中的结点）并把u中所有没标号的邻居v标上号d(v)=d(u)+1。如果v是匹配点则设置d(pair(v))=d(v)+1并把pair(v)放到队列。注意pair(v)也是X结点。如果v是未盖点则实际上已找到最短增广路，此后应忽略更长的增广路。

设最短增广路末端对应的结点标号为t，我们称这些点为“终点”，而标号为0的点为“起点”。初始时，我们随意找一个终点，则从它出发一定可以到达起点，我们沿着这条路增广。由于我们需要保证以后找到的路和这条增广路结点不相交，需要路上的所有点。然而删除点后可能出现有的终点不再能到达起点的情况，这些终点应该一并删除，即在删除点后删除和它邻接的所有边，然后删除孤立点。这个删除过程是重复的，可以用一个所谓的删除队列实现——初始时把增广路上的所有点放在删除队列中，每次取一个出来删除它所有相邻弧并检查是否有点会因此变成孤立点。如果有，把它放在删除队列中。可以证明，在这样处理后每次增广后所有终点仍能到达起点。因此只需要重复这样的操作，在O(m)时间内就可找出一个极的最短增广路集。

下面讨论二分图最大权匹配。给加权完全二分图G，假设所有权都是非负整数，我们要找到让权和最大的完美匹配。

基本概念。我们首先定义**可行顶标(feasible vertex labeling)**，它是结点的实函数，满足对任意弧(x_i, y_i)有$l(x_i) + l(y_j) \geq w(x_i, y_j)$。**相等子图(equality subgraph)**是图G的生成子图，即包含所有顶点，但只包含满足$l(x_i) + l(y_j) = w(x_i, y_j)$的所有弧($x_i$,$y_j$)。相等子图的重要性蕴涵在下面定理中：

定理：如果相等子图有完美匹配，则该匹配是原图的最大权匹配。

证明：设M*是相等子图的完美匹配，则根据定义有

$$w(M*) = \sum_{e \in M*} w(e) = \sum_{v \in X \cup Y} l(v)$$

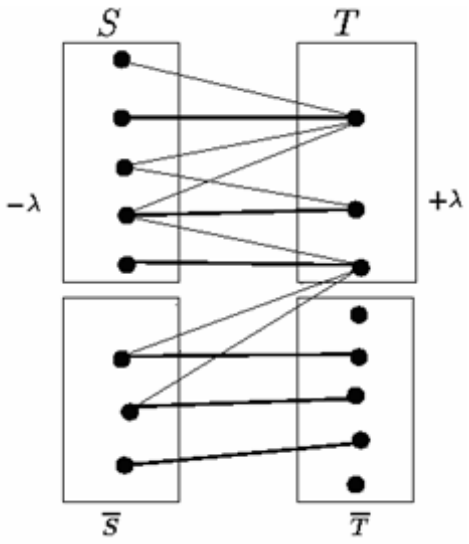
设M是原图中的任意完美匹配，则

$$w(M) = \sum_{e \in M} w(e) \leq \sum_{v \in X \cup Y} l(v) = w(M*)$$

由此可见，寻找一个“好的”可行顶标以使得相等子图存在完美匹配是问题的关

键。一个直观的想法是先随便构造一个可行顶标（如设所有Y结点的顶标为0，X结点的顶标为它出发的所有弧的最大权值），然后求相等子图的最大匹配。如果存在完美匹配存在，则算法终止，否则修改顶标使得相等子图中的边变多，有更大的机会存在完美匹配。相等子图中的边变多后，要么匹配基数变大（找到增广路），要么匈牙利树生长（增加新结点）。

Kuhn-Munkres算法。这个算法也称为**匈牙利算法(Hungarian Method)**，由Kuhn和Munkres分别于1955年和1957年提出。算法首先任意设置可行顶标并求出相等子图然后求它的最大匹配看是否为完全匹配。然后每次令S为X中的未盖点集合，用S出发生长匈牙利树，生长过程中遇到的Y结点集合设T，并把遇到的X结点也加到S中。这样，S和T实际上分别为匈牙利树中的X和Y结点集合。下面开始修改顶标，把一些新边加入到相等子图中，扩展S和T直到找到一条增广路，沿它增广并重新设置S和T，开始下一轮操作，直到相等子图中有完美匹配。



上图只画出了相等子图中的边，它描述了修改顶标的直观想法。上图中显然S到$\overline{T}$没有边（否则匈牙利树可以继续扩展）。我们把S中所有点的顶标减少一个正数，如λ，则S-$\overline{T}$中可能会有新边进入相等子图。为了防止S-T中的边离开相等子图，我们需要给T中的所有结点顶标减少λ。为了保证S-$\overline{T}$中会有新边进入相等子图，我们取

$$\lambda = \min\{l(x_i) + l(y_j) - w(x_i, y_j) | x_i \in S, y_j \in \overline{T}\}$$

则修改后取最小值的边(x_i, y_j)进入相等子图。如果$\overline{T}$中新进入相等子图的点v和$\overline{S}$中

的点匹配，把它们分别加到S和T中；如果v是未盖点，则找到增广路。由于在不扩大匹配的情况下每次匈牙利树至少增加一个元素，因此最多执行n次就会找到增广路。由于匹配数不超过n，因此最多需要找n条增广路即可，一共需要$O(n^2)$次修改顶标操作。每次修改顶标时需要扫描所有弧(x_i, y_j)以求出λ，因此修改顶标的时间复杂度为$O(n^2)$，总时间复杂度为$O(n^4)$。下面我们考虑如何把它优化到$O(n^3)$。

对于$\overline{T}$的每个元素y_j，定义松弛量$slack(y_j) = \min_{x_i \in S}\{l(x_i) + l(y_j) - w(x_i, y_j)\}$，则$\lambda = \min_{j \in \overline{T}}\{slack(y_j)\}$。每次增广后用$O(n^2)$时间计算所有点的初始slack，由于每次生长匈牙利树时所有S-$\overline{T}$弧的增量相同，因此修改每个slack值只需要常数时间，修改所有slack值需要O(n)时间。由于每次增广后最多修改n次顶标，因此每次增广后修改顶标的总时间降为$O(n^2)$，n次增广的总时间由$O(n^4)$降为$O(n^3)$。

稳定婚姻问题。稳定婚姻问题是一个很有意思的匹配问题。有n位男士和n位女士，每人都对每个异性有一个排序，代表对他们的喜欢程度。现在希望给每位男士找一位不同的女士做配偶，使得每人恰好有一个异性配偶。如果男士u和女士v不是配偶但喜欢对方的程度都大于喜欢各自当前配偶，则称他们为一个**不稳定对(unstable pair)**。如果一个配对方案存在不稳定对则此对可能会破坏原来的配对方案。稳定婚姻问题是希望找出一个不含不稳定对的方案。

稳定婚姻问题的经典算法为**求婚-拒绝算法(propose-and-reject algorithm)**，即男士按自己喜欢程度从高到低依次给每位女士主动求婚，直到有一个接受他。女士每次遇到比当前配偶更差的男士时拒绝他，遇到更喜欢的男士时就接受他，并抛弃以前的配偶。被抛弃的男士继续按照列表向剩下的女士依次求婚，直到所有人都有配偶。看起来女士更有选择权，但实际上最后得到的结果是**男士最优(man-optimal)**的。下面我们详细描述这一算法并证明相应的结论。

如果算法最后得到了一个匹配，那么它一定是稳定的。为了证明这一点，我们首先注意到随着算法的执行，每位女士的配偶越来越好，而每位男士的配偶越来越差。因此假设男士u和女士v形成不稳定对，u一定曾经向v求过婚，但被拒绝。这说明v当时的配偶比u更好，因此算法结束后的配偶一定仍比u好，和不稳定对的定义矛盾。

下面我们只需要说明算法一定成功结束，即不会存在一位男士u，使得他向所有女士求婚后仍为单身。假设存在这样的人，设其中最后一次被抛弃时刻最晚的男士为u，则他*最后一次被抛弃*时无配偶，因此当时一定存在女士v也没有配偶。由于v是单身的，所以一定没有人向她求过婚，因此v还在u的考虑范围之中，以后会向v求婚。到

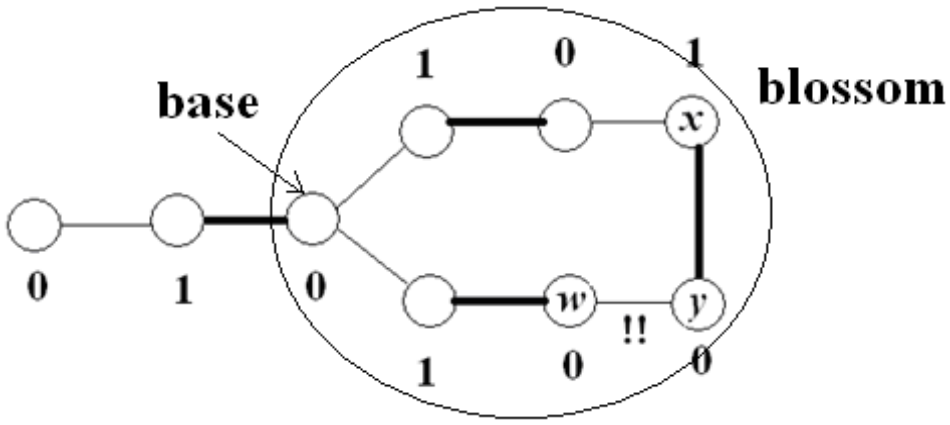
时u会再次有配偶，要么还将被抛弃一次，要么最后不会单身，无论哪种情况都将产生矛盾。

这样，我们证明了算法一定得到稳定匹配。时间复杂度显然是$O(n^2)$，因此每个男士最多考虑每个女士各一次，每次的时间复杂度均为O(1)。显然这已经是时间复杂度的下限了，因为输入是$O(n^2)$的。

如果存在一个稳定匹配使得男士i和女士j配对，则称(i,j)是稳定对。对于每个男士i，设所有稳定对(i,j)中i 最喜欢的女士为best(i)，则可以证明这里给出的算法对让每位男士i与best(i)配对。对于所有男士来说，不会有比这更好的结果了，而对于女士则恰恰相反——对于她们来说不会有比这更糟的结果了。

7.5.2 一般图的最大基数匹配

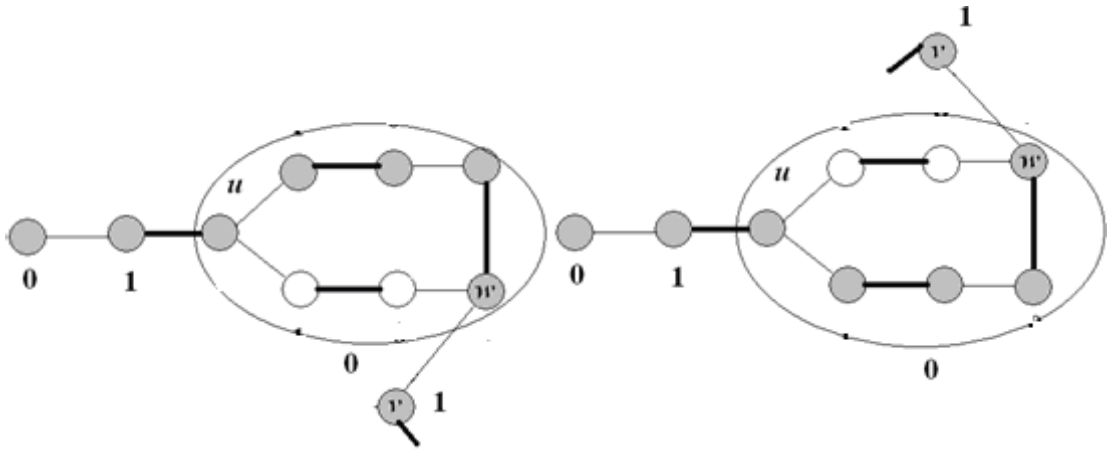
在二分图匹配中，我们总是从X中的未盖点出发生长匈牙利树，即令未盖点的标号为0执行BFS。容易证明X结点的标号（如果有的话）一定是偶数，称为偶结点，而Y结点的标号（如果有的话）一定是奇数，称为奇结点。我们把这个性质称为标号唯一性质。一般图不具备标号唯一性质：BFS可能希望给一个已经标上奇数标号的结点再编上一个偶数标号。下图就是这样的情况，在检查时w发现它的邻居y，但次时y已经被标记为了0！那么到底是重标号还是不重标号呢？都不行！不管哪种情况，如果只保留一个标号，就忽略了另外一个标号，而实际上两个标号都可能是有用的！



出现这样的情况意味着图中出现了奇圈，也成为**花朵(blossom)**。如果把花看成一个整体的话显然它“对外”生长出去的边一定是非匹配边（每个点的匹配边都在圈内），因此可以把这个圈“收缩”成一个点，称为人工结点，它一定是偶结点（想一想，为什么）。BFS中遇到的花中的第一个结点称为花的**基(base)**。

定理：如果收缩后的图有增广路，则原图也有。

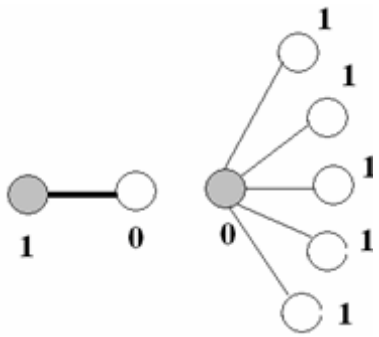
证明：设在增广路中人工结点u的下一个结点v和收缩前圈中的点w相邻。不管w点在什么位置，都可以在圈中构造出增广路（想一想，为什么），如下图。



这样，我们可以仿照二分图的情形设计出一般图上增广路算法。首先取消所有点的标号，把一个未盖点标上0并放到队列中，然后每次从队列中取出一个点i检查。

情况一 i是奇结点，则它一定有匹配边(i,j)。若j已有奇标号则执行contract(i,j)；否则如果j没有标号则给它标上偶标号，设pred(j)=i后把j加入队列。

情况二 i是偶结点，则需要检查它的所有邻边。对于每一条边(i,j)，如果j有偶标号则执行contract(i,j)；否则如果j是未盖点则找到一条增广路，否则如果j没有标号则给它奇标号，设pred(j)=i后加入队列。



收缩操作contract(i,j)无疑是算法的核心。首先需要找到i和j的最近公共祖先，它是花的基。找到基后也就找到了整个花，需要对花进行收缩。如何收缩呢？收缩是指创建一个人工结点，把和圈内点关联的所有边都连接到这个人工结点上。正式地说，设圈上的点为u_i，人工结点为w，给所有存在(u_i,v)的边的点v创建边(v,w)。

虽然收缩后原来的结点都没有了，但如果删除这些结点以后还需要恢复，时间复杂度高。更好的方法是给加一个标记，表明该结点不再活跃。注意这样会使一些点的邻接点变多（原来的邻接点不变，多了一个人工结点），但最多只增加到3n/2，不会增加算法的总时间复杂度。

我们把算法执行过程中的临时图称为**表面图(surface graph)**，则它可以看成是若干个花朵集合，每个花朵是奇数个结点构成的圈，构成花朵的结点还可以是花朵，称为**子花朵(subblossom)**。这样，每一个花朵是一个层次结构，每个叶子代表原图的结点，而其他结点代表算法生成的花朵。因此算法执行过程中的新图可以看作是一个森林，其中每棵树的根结点对应的花朵称为**表面花朵(surface blossom)**，而其他内结点称为**死花朵(head blossom)**。收缩过程实际上是给该奇圈建立一个表面花朵，让圈上所有结点成为它的儿子，并变为死花朵，而展开过程则是删除表面花朵，让它的所有儿子重新变成表面花朵。

时间复杂度。到检查奇结点只需要O(1)时间，而检查偶结点需要O(n)时间，而寻找增广路的过程中最多检查3n/2个点，因此每次增广的时间复杂度为$O(n^2)$。

每次增广前的所有收缩和展开一共只需$O(n^2)$时间（留给读者思考），因此n次增广一共需要$O(n^3)$时间，这就是算法的时间复杂度。

7.5.3 小结

本节讨论匹配问题，包括二分图的最大基数匹配、最大权匹配、稳定婚姻问题，最后叙述了一般图的最大基数匹配。

本节的算法和程序列表如下：

7.6 线性规划

为了更好的理解网络优化问题，我们学习**线性规划(linear programming)**。考虑n个变量$x_1, x_2, \ldots, x_n$，它们的**线性函数(linear function)**f被定义为：$f(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = \sum_{j=1}^{n} a_jx_j$，其中$a_1, a_2, \ldots, a_n$是$n$个实数。设$b$为实数，$f$是一个线性函数，则$f(x_1, x_2, \ldots, x_n) = b$是一个**线性等式(linear equality)**，而$f(x_1, x_2, \ldots, x_n) \leq b$和$f(x_1, x_2, \ldots, x_n) \geq b$是**线性不等式(linear inequality)**。线性等式和线性不等式统称**线性约束(linear constraint)**。注意在线性规划里，线性不等式总是可以取等号的。

算法	程序	备注
Edmonds算法	edmonds.cpp	二分图最大匹配的edmonds算法，用BFS扩展匈牙利树，时间复杂度为O(nm)。
Hopcroft-Karp算法	hopcroft.cpp	二分图最大匹配的Hopcroft-Karp算法，每次沿极大最短增广路集增广，时间复杂度为$O(n^{1/2}m)$。
Kuhn-Munkres算法	KM.cpp	二分图最大权匹配的KM算法，时间复杂度为$O(n^3)$。
求婚-拒绝算法	stable.cpp	稳定婚姻问题的求婚-拒绝算法，时间复杂度为$O(n^2)$。
Edmonds算法	general.cpp	一般图最大基数匹配的Edmonds算法，时间复杂度为$O(n^3)$。

7.6.1 标准形式和松弛形式

线性规划问题：给定n个变量的m个线性约束，让这n个变量的某个线性函数最大化或最小化。

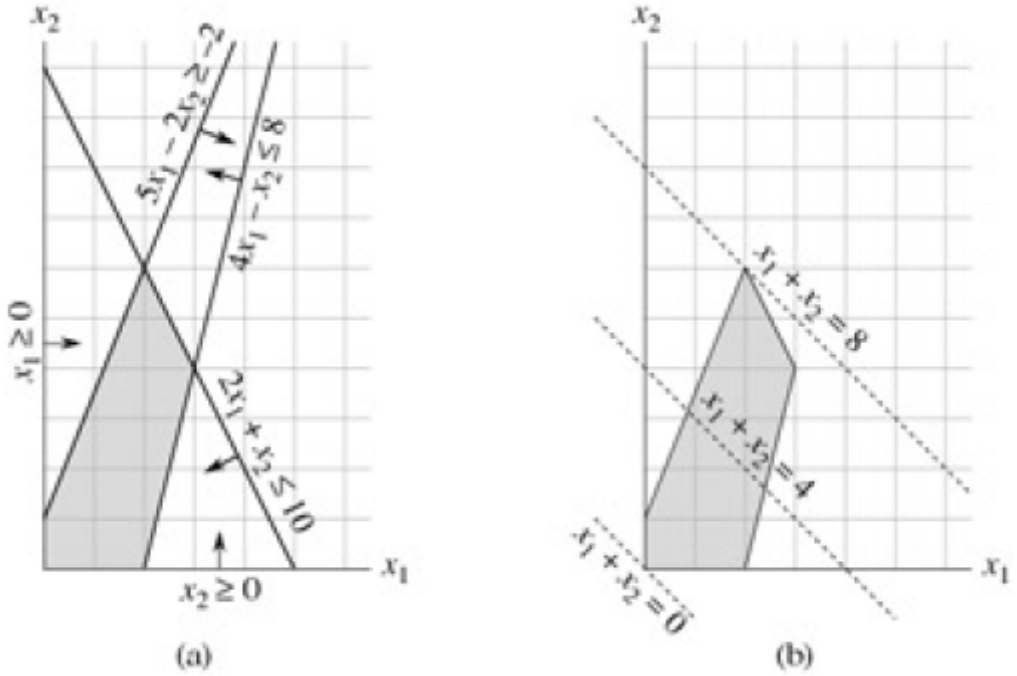
线性约束有三种，我们往往需要把它们进行规范化。线性规划问题的规范形式有两种：**标准形式(standard form)**和**松弛形式(slack form)**。非正式地说，标准形式是不等式约束下的最大化问题，而松弛形式是等式约束下的最大化问题。我们一般用标准形式描述线性规划问题，而用松弛形式描述下面我们要介绍的单纯形法。

考虑如下的线性规划问题：

$$\begin{aligned}&\max\{x_1+x_2\}\\&4x_1-x_2\le 8\\&2x_1+x_2\le 10\\&5x_1-2x_2\ge -2\\&x_1,x_2\ge 0\end{aligned}$$

满足所有4个约束的解称为**可行解(feasible solution)**，下图(a)描述了上述问题的可行解，其中每个约束对应一个半平面，所有半平面的交构成了一个凸区域，称为**可行区域(feasible region)**，它是所有可行解的集合。我们希望最大化的函数称为**目标函数(objective function)**，而可行区域里每个点都有一个**目标函数值(objective value)**。图(b)给出了可行区域关于目标函数的等值线，每条虚线上的目标函数值相同，且越往上这个值越大。存在一个值使得对应的等值线和可行区域“刚好相交”，这

就是最优解。这个“刚好相交”可以是和可行区域相交于一个点或者与它的一条边界重合，但不管怎样，一定存在可行区域的一个顶点，使得它是最优解。



三维或更高维的情形不容易像二维情形一样用直观图表示出来，但其中的思路是一致的。假设有n个变量，则：

- 每个不等式约束是n维空间中的**半空间(half-space)**。
- 可行区域是这些半空间的交，称为**单纯形(simplex)**。
- 目标函数是一个**超平面(hyperplane)**。
- 最优解仍然在此单纯形的顶点处。

线性规划的经典算法是单纯形法。它并不是多项式时间算法，但在实际应用中速度很快。它的基本思想是从可行区域构成的单纯形上的一个顶点出发进行一系列的迭代。每次迭代都沿着一条边从一个顶点走到另一个顶点，保证目标函数不减（往往是严格增加）。如果无法行走，说明已经到达局部极大值，算法终止。由于单纯形是凸的，局部极大值就是全局极大值，我们将在后面用对偶性严格证明这一结论。

虽然几何表示直观，但代数描述更容易编程。我们首先将线性规划问题重写为松弛形式，即把约束全部写成等式，这样就得到了一个方程组。这个方程组实际上是用

一些变量表示一些变量。被表示的变量称为**基变量(basic variable)**，而用来表示基变量的变量称为**非基变量(nonbasic variable)**。从一个顶点走到另一个顶点的操作对应于选一个基变量出基，再选一个非基变量入基，这个操作称为**旋转操作(pivot)**，在代数上只是把一个方程组写成另一个等价的形式，从程序上讲就是从一个松弛形式走到另一个松弛形式。由于等价的松弛形式只有C_{n+m}^{m}个，只要保证不重复生成已考虑过的松弛形式，单纯形法一定会终止且得到最优解。

线性规划算法还需要解决一些特殊情况，比如无解的情形，无有界最优解的情形，以及寻找一个初始可行解作为行走的起点，这些我们将在后面逐一讨论。

标准形式。给n个实数$c_1, c_2, \ldots, c_n$，m个实数$b_1, b_2, \ldots, b_m$和mn个实数$a_{ij}(i = 1, 2, \ldots, m; j = 1, 2, \ldots, n)$。我们希望找出n个实数$x_1, x_2, \ldots, x_n$使得$\sum_{j=1}^{n} c_j x_j$最大，且满足线性不等式约束$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (i = 1, 2, \ldots, m)$和非负约束$x_j \geq 0 \quad (j = 1, 2, \ldots, n)$，即：

$$\begin{aligned} &\max \sum_{j=1}^{n} c_j x_j \\ &\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad i = 1, 2, \ldots, m \\ &x_j \geq 0 \quad j = 1, 2, \ldots, n \end{aligned}$$

注意有些应用中并没有非负约束，但标准形式必须有。令$m \times n$矩阵$A = (a_{ij})$和m维列向量$b = (b_i)$和n维列向量$c = (c_j)$及n维列向量$x = (x_j)$，则可以把标准型写成更为紧凑的形式：

$$\begin{aligned} &\max\{c^T x\} \\ &Ax \leq b \\ &x \geq 0 \end{aligned}$$

以后我将直接用三元组(A, b, c)表示一个线性规划问题，且默认三者分别为上述大小。任何一个线性规划问题都可以转化为等价的标准形式，步骤如下：

第一步 如果是最小化问题，把目标函数所有系数取反。

第二步 如果有变量x_j没有非负约束，用$x'_j - x''_j$代替它，增加约束$x'_j \geq 0, x''_j \geq 0$。

第三步 等式约束$f(x_1, x_2, \ldots, x_n) = b$拆成两个约束$f(x_1, x_2, \ldots, x_n) \leq b$和$f(x_1, x_2, \ldots, x_n) \geq b$。

第四步 对于所有“大于等于”约束$\sum_{j=1}^{n} a_{ij} x_j \geq b_i$，改写成$\sum_{j=1}^{n} -a_{ij} x_j \leq -b_i$。

最后注意把形如x'_i和x''_i的名字改一改，让所有变量的名字都是x_i，以便程序处理。

例如对于线性规划问题：

$$\begin{aligned}&\min\{-2x_1+2x_2\}\\&x_1+x_2=7\\&x_1-2x_2\le 4\\&x_1\ge 0\end{aligned}$$

变换后的标准型为：

$$\begin{aligned}&\max\{2x_1-3x_2+3x_3\}\\&x_1+x_2-x_3\le 7\\&-x_1-x_2+x_3\le -7\\&x_1-2x_2+2x_3\le 4\\&x_1,x_2,x_3\ge 0\end{aligned}$$

松弛形式。为了使用单纯形法，我们需要把标准形式进一步转化为松弛形式，即保留非负约束的情况下给出线性方程组。在松弛形式中，非负约束是唯一的不等式约束。考虑标准型中的不等式约束$\sum_{j=1}^{n} a_{ij}x_j \le b_i$。引入变量s，把该约束变为：

$$\begin{cases} s=b_i-\sum_{j=1}^{n} a_{ij}x_j \\ s\ge 0 \end{cases}$$

其中s称为松弛变量，因为它度量了松弛度，即方程左右两边的差。为了程序方便，我们一般把第i个约束对应的松弛变量记为x_{n+i}，另外记z为目标函数，略去非负约束，则刚才那个例子的松弛形式为：

$$\begin{cases} z=2x_1-3x_2+3x_3 \\ x_4=7-x_1-x_2+x_3 \\ x_5=-7+x_1+x_2-x_3 \\ x_6=4-x_1+2x_2-2x_3 \end{cases}$$

其中方程左边的变量，即x_4，x_5和x_6称为基变量，而方程右边的变量x_1，x_2，x_3称为非基变量。从松弛形式的构造过程可以看出：任何一个变量都不可能同时出现在方程的两边。我们用N表示非基变量的下标集合，B表示基变量的下标集合，则始终有|N|=n，|B|=m，且$N\cup B=\{1,2,\ldots,n+m\}$。方程用左边基变量的下标即B中的元素来标识，而右边的变量用N中的元素来标识。有时候目标函数中还有常数v，我们把它连同其他量一起定义松弛形式的六元组(N,B,A,b,c,v)，即：

$$\begin{cases} z = v + \sum\limits_{j \in N} c_j x_j \\ x_i = b_i - \sum\limits_{j \in N} a_{ij} x_j \quad i \in B \end{cases}$$

注意松弛形式隐含了非负约束$x_i \geq 0$，且在方程中所有系数前面有个减号。因此对于松弛形式：

$$\begin{cases} z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \end{cases}$$

六元组(N, B, A, b, c, v)为：

$$A = \begin{bmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{bmatrix} = \begin{bmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{bmatrix}$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix}, c = \begin{pmatrix} c_3 & c_5 & c_6 \end{pmatrix}^T = \begin{pmatrix} -1/6 & -1/6 & -2/3 \end{pmatrix}^T, v = 28$$

关于松弛形式的一个重要结论是：基变量集合B唯一确定松弛形式。由于B里有m个元素，因此一共有C_{n+m}^m种不同的松弛形式。这一点对于理解单纯形法的迭代次数上限是很重要的。

我们把关于松弛形式的重要结论总结如下：

- 松弛形式可以用六元组(N, B, A, b, c, v)表示。
- 松弛形式有n个原始变量x_i和m个松弛变量x_{n+i}。
- 基变量和目标函数都用非基变量的线性函数表示。
- 用基变量的下标i标识各个等式。
- 非基变量的下标集合记为N，基变量的下标集合记为B。
- 松弛形式隐含了非负约束$x_i \geq 0$
- 等价的松弛形式有C_{n+m}^m种。

非基变量全部取零后可以计算出所有基变量和目标函数值，这样的解称为该松弛形式的**基解(basic solution)**。基解并不一定是可行的，但在单纯形法中，每次迭代产生的新松弛形式的基解都是可行的，这个基解对应几何描述中可行域的一个顶点，迭代结束后的松弛形式的基解就是线性规划问题的最优解。

7.6.2 单纯形法

单纯形法是解决线性规划问题最经典的方法。我们首先叙述主算法，然后对它的细节加以讨论。

```
SIMPLEX(A, b, c)
 1  (N, B, A, b, c, v) ← INITIALIZE-SIMPLEX(A, b, c)
 2  while some index j ∈ N has c_j > 0
 3      do choose an index e ∈ N for which c_e > 0
 4         for each index i ∈ B
 5             do if a_ie > 0
 6                   then Δ_i ← b_i/a_ie
 7                   else Δ_i ← ∞
 8         choose an index l ∈ B that minimizes Δ_i
 9         if Δ_l = ∞
10            then return "unbounded"
11            else (N, B, A, b, c, v) ← PIVOT(N, B, A, b, c, v, l, e)
12  for i ← 1 to n
13       do if i ∈ B
14             then x̄_i ← b_i
15             else x̄_i ← 0
16  return (x̄_1, x̄_2, ..., x̄_n)
```

上述算法就是单纯形法。首先调用INITIALIZE-SIMPLEX得到初始可行解，然后每次从非基变量中选一个c_e¿0的准备入基，并确定出基变量l，然后执行旋转操作。算法终止后设置非基变量为0，计算基变量。

这个算法框架有若干部分需要讨论：

一、如何得到初始可行解？（行1）

二、如果入基变量有多种选择，选哪个？（行3）

三、如果出基变量有多种选择，选哪个？（行8）

四、算法一定会终止吗？

五、如果终止，一定得到可行解吗？

六、如果终止，一定得到最优解吗？

其中第一个问题我们将在后面单独讨论，单纯性法本身没有对第二、三个问题作硬性规定，但通常使用**Bland规则(Bland's rule)**来解决冲突：总是选择下标最小的

出/入基变量。对于第四个问题，一般情况的单纯形法是可能产生循环的[7]，但如果使用Bland规则则循环一定不会出现。第五、六个问题的答案都是肯定的，这里暂时不加证明。这样，关于单纯形法的基本结论如下：

引理：如果INITIALIZE-SIMPLEX返回的松弛形式的基解是可行的，则算法SIMPLEX要么报告问题无界，要么最多C_{n+m}^{m}次迭代后返回最优解。

这里的迭代次数C_{n+m}^{m}可以这么理解：由于一共有C_{n+m}^{m}个可能的松弛形式。于出入基规则又是确定性的，因此若遍历过所有松弛形式后还没有终止，则一定无限循环。

旋转操作

单纯形法最核心的操作是旋转，它的作用是让一个非基变量入基，并让一个基变量出基。

```
PIVOT(N, B, A, b, c, v, l, e)
 1 ▷ Compute the coefficients of the equation for new basic variable x_e.
 2 b̂_e ← b_l / a_le
 3 for each j ∈ N - {e}
 4     do â_ej ← a_lj / a_le
 5 â_el ← 1 / a_le
 6 ▷ Compute the coefficients of the remaining constraints.
 7 for each i ∈ B - {l}
 8     do b̂_i ← b_i - a_ie b̂_e
 9        for each j ∈ N - {e}
10            do â_ij ← a_ij - a_ie â_ej
11        â_il ← -a_ie â_el
12 ▷ Compute the objective function.
13 v̂ ← v + c_e b̂_e
14 for each j ∈ N - {e}
15     do ĉ_j ← c_j - c_e â_ej
16 ĉ_l ← -c_e â_el
17 ▷ Compute new sets of basic and nonbasic variables.
18 N̂ = N - {e} ∪ {l}
19 B̂ = B - {l} ∪ {e}
20 return (N̂, B̂, Â, b̂, ĉ, v̂)
```

这个程序较长且不易理解，需要仔细推导。为了方便起见，用N_1表示除了e外的非基变量集合，即$N-\{e\}$，它也是旋转后除了l外的非基变量集合，即$\hat{N}-\{l\}$。首先考虑出基变量x_l对应的方程

[7] 原因是可能在一次迭代后目标函数不变。

$$x_l = b_l - a_{le}x_e - \sum_{j\in N_1} a_{lj}x_j$$

改写后为：

$$\begin{aligned} x_e &= \tfrac{b_l}{a_{le}} - \tfrac{1}{a_{le}}x_l - \sum_{j\in N_1}\tfrac{a_{lj}}{a_{le}}x_j \\ &= \hat{b}_e - \hat{a}_{el}x_l - \sum_{j\in N_1}\hat{a}_{ej}x_j \end{aligned} \quad (*)$$

因此定义$\hat{b}_e = \frac{b_l}{a_{le}}, \hat{a}_{el} = \frac{1}{a_{le}}, \hat{a}_{ej} = \frac{a_{lj}}{a_{le}}$，对应于第2~5行。

现在需要改写其他方程。即用x_l来代替原来的x_e，因为原来的基变量方程中含有x_e，而不含x_l，现在得“反过来”。

考虑原来的方程

$$x_i = b_i - a_{ie}x_e - \sum_{j\in N_1} a_{ij}x_j$$

旋转操作实际上是代入(*)，即

$$\begin{aligned} x_i &= b_i - (a_{ie}\hat{b}_e - a_{ie}\hat{a}_{el}x_l - \sum_{j\in N_1} a_{ie}\hat{a}_{ej}x_j) - \sum_{j\in N_1} a_{ij}x_j \\ &= (b_i - a_{ie}\hat{b}_e) - (-a_{ie}\hat{a}_{el})x_l - \sum_{j\in N_1}(a_{ij} - a_{ie}\hat{a}_{ej})x_j \\ &= \hat{b}_i - \hat{a}_{il}x_l - \sum_{j\in N_1}\hat{a}_{ij}x_j \end{aligned}$$

式子很长，但只是合并同类项。对比得：$\hat{b}_i = b_i - a_{ie}\hat{b}_e, \hat{a}_{il} = -a_{ie}\hat{a}_{el}, \hat{a}_{ij} = a_{ij} - a_{ie}\hat{a}_{ej}$。这就是行7-11的内容。

最后一步是修改目标函数$z = v + \sum_{j\in N} c_j x_j$。注意由于这里只是对非基变量进行求和，所以只需要用出基的x_l来表示入基的变量x_e，即

$$\begin{aligned} z &= v + \sum_{j\in N} c_j x_j = v + c_e x_e + \sum_{j\in N_1} c_j x_j = v + c_e(\hat{b}_e - \hat{a}_{el}x_l - \sum_{j\in N_1}\hat{a}_{ej}x_j) + \sum_{j\in N_1} c_j x_j \\ &= (v + c_e\hat{b}_e) + (-c_e\hat{a}_{el})x_l + \sum_{j\in N_1}(c_j - c_e\hat{a}_{ej})x_j \\ &= \hat{v} + \hat{c}_l x_l + \sum_{j\in N_1}\hat{c}_j x_j \end{aligned}$$

对比得：$\hat{v} = v + c_e\hat{b}_e, \hat{c}_l = -c_e\hat{a}_{el}, \hat{c}_j = c_j - c_e\hat{a}_{ej}$，这就是行13-16的内容。至此，旋转过程的全部公式推导完毕，虽然冗长但道理简单，仅仅是方程变形、代入和合并同类项，读者可以自己试一试。

对偶原理和单纯形法的最优性

前面我们不加证明的给出了单纯形法得到最优解的结论，现在我们用对偶原理把它的证明补充叙述一下。给定线性规划问题的标准形式(A,b,c)，定义它的对偶问题如下：

$$\begin{aligned}&\min\{\sum_{i=1}^{m} b_i y_i\}\\&\sum_{i=1}^{m} a_{ij} y_i \geq c_j \quad j=1,2,\ldots,n\\&y_i \geq 0 \quad j=1,2,\ldots,m\end{aligned}$$

注意我们把最大化问题换成了最小化问题，且把大于等于号换成了小于等于号，非负约束形式不变。注意变量个数变成了m，而b和c交换了位置。这可以理解为给原始问题的m个约束每个约束关联了一个对偶变量，而原始问题的变量变成了对偶问题的约束。

关于对偶问题，有一个非常有意思的结论：

弱线性规划对偶性。设$\bar{x}$和$\bar{y}$分别是原始问题和对偶问题的可行解，则

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{i=1}^{m} b_i \bar{y}_i$$

这个看起来很神奇的定理其实很容易证明。利用$\sum_{i=1}^{m} a_{ij} y_i \geq c_j$和$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$，直接可以得到：$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i\right) \bar{x}_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j\right) \bar{y}_i \leq \sum_{i=1}^{m} b_i \bar{y}_i$。

作为定理的简单推论，如果存在原始问题和对偶问题的解$\bar{x}$和$\bar{y}$满足

$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i (**)$

则二者分别是各自问题的最优解。只需注意原始问题是最大化问题和对偶问题是最小化问题，推论容易得证。

线性规划对偶定理。为了证明单纯形法得到的解是最优解，我们需要从它得到原始问题的解$\bar{x}$构造出对偶问题的可行解$\bar{y}$。如果(**)得到满足，则$\bar{x}$和$\bar{y}$分别是原始问题和对偶问题的最优解。

设单纯形法终止时的松弛形式为：

$$\begin{cases} z = v' + \sum\limits_{j \in N} c'_j x_j \\ x_i = b'_i - \sum\limits_{a'_{ij} x_j \quad i \in B} \end{cases}$$

则构造

$$\bar{y}_i = \begin{cases} -c'_{n+i} & (n+i) \in N \\ 0 & (n+i) \notin N \end{cases} \quad (***)$$

我们需要证明两个结论：原始问题和对偶问题的目标函数值相同；$\bar{y}$是对偶问题的可行解。因为非基变量都取0，而基变量根本没有在目标函数中出现过，因此原始问题的目标函数值是v'，故只需计算对偶问题的目标函数值，看它是否也为v'。

由于基变量不在目标函数中出现，因此可以令它们的系数均为0，这样对任意一组变量赋值$x = (x_1, x_2, \ldots, x_n)$（注意它不一定是原始问题的可行解），都有目标函数值

$$\sum_{j=1}^{n} c_j x_j = v' + \sum_{j=1}^{n+m} c'_j x_j$$

把这些项拆成原始变量和松弛变量两部分（而不是刚才的拆成基变量和非基变量两部分），变形得：

$$\sum_{j=1}^{n} c_j x_j = v' + \sum_{j=1}^{n+m} c'_j x_j = v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} c'_{n+i} x_{n+i} = v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} (-\bar{y}_i) x_{n+i}$$

然后在第一个松弛形式中把松弛变量x_{n+i}用原始变量表示，得

$$\sum_{j=1}^{n} c_j x_j = v' + \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} (-\bar{y}_i) \left(b_i - \sum_{j=1}^{n} a_{ij} x_j \right) = v' + \sum_{j=1}^{n} c'_j x_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{j=1}^{n} \sum_{i=1}^{m} (a_{ij} \bar{y}_i) x_j$$

按x_j进行合并同类项，得

$$\sum_{j=1}^{n} c_j x_j = \left(v' - \sum_{i=1}^{m} b_i \bar{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \right) x_j$$

这是一个非常有用的等式，它两第一个和最后一个松弛形式联系在了一起。左右两边不仅值相等，而且各变量的系数对应相等（基变量集合B唯一确定松弛形式），因此：

$$\begin{cases} v' - \sum_{i=1}^{m} b_i \bar{y}_i = 0 \\ c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i = c_j \quad j = 1, 2, \ldots, n \end{cases}$$

其中第一个等式就是我们需要的：对偶问题的目标函数值也是v'。第二个等式可以帮助我们证明$\bar{y}$的可行性：由于单纯形法结束时所有$c'_j \leq 0$，我们补充规定的基变量的系数都为0，因此$c_j = c'_j + \sum_{i=1}^{m} a_{ij}\bar{y}_i \leq \sum_{i=1}^{m} a_{ij}\bar{y}_i$，这就是对偶问题的约束。仍然由于$c'_j \leq 0$，我们在构造$\bar{y}$时得到的所有变量都是非负的，满足非负约束。

综上所述，单纯形法得到的解$\bar{x}$和按照(***)构造出的$\bar{y}$分别是原始问题和对偶问题的最优解。

最后，我们叙述对偶原理中另外一个重要定理：

互补松弛定理(Complementary slackness theorem)：设$\bar{x}$和$\bar{y}$是原始问题和对偶问题的可行解，则它们分别为两个问题的最优解当且仅当

$$\sum_{i=1}^{m} a_{ij}\overline{y_i} = c_j \ or \ \overline{x_j} = 0 \quad j = 1, 2, \ldots, n$$
$$\sum_{i=1}^{n} a_{ij}\overline{x_j} = b_i \ or \ \overline{y_i} = 0 \quad i = 1, 2, \ldots, m$$

由定理容易得到：原始问题的解$\bar{x}$为最优解当且仅当存在对偶问题可行解$\bar{y}$，使得对于$\overline{x_j} > 0$有$\sum_{i=1}^{m} a_{ij}\overline{y_i} = c_j$，对于$\sum_{j=1}^{n} a_{ij}\overline{x_j} < b_i$有$\overline{y_i} = 0$。

初始可行解

遗留下来的最后一个问题是；如何寻找基解可行的松弛形式？它甚至有可能是不存在的。显然，随意找一个松弛形式并不能保证它的基解可行，因此需要深入讨论。

构造辅助线性规划问题L_{aux}如下：

$$\max\{-x_0\}$$
$$\sum_{j=1}^{n} a_{ij}x_j - x_0 \leq b_i \quad i = 1, 2, \ldots, m$$
$$x_j \geq 0 \quad j = 0, 1, \ldots, n$$

则原始线性规划可行当且仅当L_{aux}的最优解。请注意L_{aux}显然是有可行解的，因为只要设$x_1 = x_2 = \cdots = x_n = 0$，当$\mathrm{x}_0$无限增大时所有约束都将满足。这也说明了为什么我们要“用一个线性规划问题解另一个线性规划问题”：因为辅助问题的初始可行解容

易找到——原始松弛形式的基解就是可行的。这样，寻找初始可行解的过程就不难给出了：求解L_{aux}，如果最优值为0则原问题可行，否则原问题不可行。这是因为由于非负约束，最优值一定不为正数。如果原问题可行则x_0=0时一定有解，且它一定是辅助问题的最优解。

```
INITIALIZE-SIMPLEX(A, b, c)
 1  let l be the index of the minimum b_i
 2  if b_i ≥ 0          ▷Is the initial basic solution feasible?
 3      then return ({1, 2, ..., n}, {n + 1, n + 2, ..., n + m}, A, b, c,
0)
 4  form L_aux by adding -x_0 to the left-hand side of each equation
          and setting the objective function to -x_0
 5  let (N, B, A, b, c, v) be the resulting slack form for L_aux
 6  ▷ L_aux has n + 1 nonbasic variables and m basic variables.
 7  (N, B, A, b, c, v) ← PIVOT(N, B, A, b, c, v, l, 0)
 8  ▷ The basic solution is now feasible for L_aux.
 9  iterate the while loop of lines 2-11 of SIMPLEX until an optimal
solution
              to L_aux is found
10 if the basic solution sets x̄_0 = 0
11    then return the final slack form with x_0 removed and
                          the original objective function restored
12    else return "infeasible"
```

注意在找到辅助问题的最优解且它为0时为了得到原问题的可行解需要把x_0去掉，并把目标函数改写成只含有非基变量的形式。

至此，我们已经完整的叙述了传统单纯形法的所有实现细节，并给出了绝大部分关键定理的证明。我们用一个定理来结束本节：

线性规划基本定理。对于任意线性规划问题的标准形式，以下三种情况恰好有一个成立：

情况一 存在目标值有限的最优解

情况二 无解

情况三 无界，即对于任意大的目标函数值都能找到一组解，因此最大值不存在。

定理的正确性十分直观，且单纯形法可以在有限次迭代后准确的判断出这三种情况，是一个实际应用中表现优秀的算法。

7.6.3 网络优化问题的线性规划模型

有了基础知识，我们介绍网络优化问题的线性规划模型。了解这些模型有助于我们深入了解这些优化问题的本质，并可以理解更多基于线性规划思想的算法，解决更

多变形的网络优化问题。

单源最短路问题。给定边权w(u,v)，我们需要求出距离标号d[v]，即从s到v的最短路长度。线性规划模型为：

$$\begin{aligned}&\min\{d[t]\}\\&d[v]\le d[u]+w(u,v)\quad\forall(u,v)\in E\\&d[s]=0\end{aligned}$$

有n个变量d[v]，m+1个约束（每条边一个，再加上d[s]=0）。

最大流问题。给定边容量c(u,v)，我们需要求出边上的流量f(u,v)。线性规划模型为：

$$\begin{aligned}&\max\{\sum_{v\in V}f(s,v)\}\\&f(u,v)\le c(u,v)\quad\forall u,v\in V\\&f(u,v)=-f(v,u)\quad\forall u,v\in V\\&\sum_{v\in V}f(u,v)=0\quad u\in V-\{s,t\}\end{aligned}$$

有n^2个变量，即弧流量，$2n^2$+n-2个约束，即容量约束n^2个，对称性n^2个，收支平衡n-2个。这里把不存在的弧作为容量为0，事实上还可以做得更好（稀疏图下问题变得更小），方法留给读者思考。

最小费用流。给出每条弧上的容量c(u,v)和费用f(u,v)，希望求出每条弧上的容量。问题的线性规划模型为：

$$\begin{aligned}&\min\{\sum_{(u,v)\in E}w(u,v)f(u,v)\}\\&f(u,v)\le c(u,v)\quad\forall u,v\in V\\&f(u,v)=-f(v,u)\quad\forall u,v\in V\\&\sum_{v\in V}f(u,v)=0\quad\forall u\in V-\{s,t\}\\&\sum_{v\in V}f(s,v)=d\end{aligned}$$

其中d是指定流量。这是必须的，因为如果不指定的话最小费用流当然是0流。

多商品流问题(multicommodity-flow problem)。假设你要运送多个商品，占用同一个运输网络。也就是说，对于每条弧(u,v)来说，它们的总流量，也称**聚集流(aggregate flow)**，定义为所有商品的总流量$f(u,v)=\sum_{i=1}^{k}f_i(u,v)$，它不能超过弧容量。和最小费用流问题类似，这里定义源点处每个商品的供给量d_i，则该问题不需要做任何优化，只需要找到一个可行解即可。线性规划模型为：

$$
\begin{aligned}
&\min\{0\}\\
&\sum_{i=1}^{k} f_i(u,v) \le c(u,v) \quad \forall u,v \in V\\
&f_i(u,v) = -f_i(u,v) \quad \forall i = 1,2,\ldots,k \quad \forall u,v \in V\\
&\sum_{v\in V} f_i(u,v) = 0 \quad \forall i = 1,2,\ldots,k \quad \forall u \in V - \{s_i,t_i\}\\
&\sum_{v\in V} f_i(s_i,v) = d_i \quad \forall i = 1,2,\ldots,k
\end{aligned}
$$

类似可以定义最小费用多商品流。

7.6.4 原始-对偶算法

在对偶性原理一节中，我们曾经给出互补松弛定理。本节以最小费用流和一般图最大权匹配问题为例介绍一类基于互补松弛定理的算法，称为**原始-对偶算法(primal-dual algorithms)**。

最小费用流问题

考虑单源单汇网络的最小费用流问题：

$$
\begin{aligned}
&\min \sum_{(i,j)\in A} c_{ij}x_{ij}\\
&s.t. \quad \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = d_i = \begin{cases} v, & i = s \\ -v, & i = t \\ 0, & i \in V, i \ne s,t \end{cases}\\
&0 \le x_{ij} \le u_{ij}, \quad (i,j) \in A
\end{aligned}
$$

两类约束分别引入对偶变量π和z，则问题的对偶问题为：

$$
\begin{aligned}
&\max w(\pi,z) = \sum_{i\in V} d_i\pi_i - \sum_{(i,j)\in A} u_{ij}z_{ij}\\
&s.t. \quad \pi_i - \pi_j - z_{ij} \le c_{ij} \quad (i,j) \in A\\
&z_{ij} \ge 0 \quad (i,j) \in A
\end{aligned}
$$

由互补松弛定理有：

$$
\begin{aligned}
&x_{ij}(\pi_i - \pi_j - z_{ij} - c_{ij}) = 0 \quad (i,j) \in A\\
&z_{ij}(x_{ij} - u_{ij}) = 0 \quad (i,j) \in A
\end{aligned}
$$

容易得到：

$$
\text{当}\pi_i - \pi_j < c_{ij}\text{时，} \ x_{ij} = 0
$$

$$当\pi_i - \pi_j > c_{ij}时，\ x_{ij} = u_{ij}$$

$$当0 < x_{ij} < u_{ij}时，\ \pi_i - \pi_j = c_{ij}$$

反过来，如果满足上面三个式子，可以证明一定存在z满足互补松弛定理。事实上，只需要令$z_{ij} = \max\{\pi_i - \pi_j - c_{ij}, 0\},\quad (i,j) \in A$即可。因此我们只需要满足上述三个式子（称为最优性条件），得到的x就是最小费用流。称π_i为结点i的**势(potential)**。

原始-对偶算法从可行流x=0开始不断增广，只要随时都存在结点势，则增广过程中得到的始终是该流量的最小费用流。初始势为0，显然满足零0的最优性条件。算法的每次迭代在增广的同时修改结点势，使得最优性条件始终满足。

记$c_{ij}^{\pi} = c_{ij} - \pi_i + \pi_j$，则原始-对偶算法只能沿着满足$c_{ij}^{\pi} = 0$的弧增广。所有满足此条件的弧组成所有为允许网络(admissible network)，则原始-对偶算法每次迭代沿着允许网络的最大流进行增广。当允许网络不存在增广路时必须修改结点势，使得修改后允许网络中将增加一些弧，使得重新存在s-t路。把c_{ij}^{π}作为弧(i,j)的权，设残量网络中从s到i的最短路长度为d(i)，修改势$\pi_i' = \pi_i - d(i)$。由于最短路树上的所有弧满足

$$d(j) - d(i) = c_{ij}^{\pi} = c_{ij} - \pi_i + \pi_j$$

因此修改后满足$c_{ij}^{\pi'} = 0$，加入到允许网络中。这样，我们重复这一过程，每次求允许网络的最大流并增广，然后修改势，直到流量到达给定值。

可以看出，连续最小费用路算法是原始-对偶算法的特殊情况（虽然我们并没有显式的引入对偶变量），它每次只沿一条增广路增广，而原始-对偶算法一次沿着多条最短增广路增广。事实上，对于残量网络的任意s-t路P，都有$C(P) = \sum\limits_{(i,j)\in P} c_{ij} = \sum\limits_{(i,j)\in P} c_{ij}^{\pi} + \pi(s) - \pi(t)$。由于对于残量网络任意弧都有$c_{ij}^{\pi} \geq 0$（注意最优性条件，满足$c_{ij}^{\pi} < 0$的弧都是满载的，因此不会在残量网络中出现），所以只由满足$c_{ij}^{\pi} = 0$的弧所组成的s-t路一定是最短路。事实上，改进的连续最小费用路算法的算法步骤确实和原始-对偶算法是非常相似的，读者可以比较二者的区别。

顺便可以指出，读者可能已经看出了，二分图最大权匹配的Kuhn-Munkres算法正是原始-对偶算法，只是对于该特定应用我们用一种更容易叙述和证明的方式来阐述这个算法。

一般图的最大权匹配

一般图最大权匹配的原始-对偶算法法由Edmonds于1965年提出，时间复杂度为(n^4)，然后由Lawler 和Gabow独立改进到O(n^3)。

线性规划模型。为了使用原始-对偶算法，我们需要先把匹配问题叙述成线性规划的形式。设弧(i,j)的权为c_{ij}，x_{ij}=1表示该边是匹配边，x_{ij}=0表示该边不是匹配边，则一般图最大权匹配问题可以描述成：

$$\begin{aligned} &\max \sum_{(i,j)\in A} c_{ij}x_{ij} \\ &s.t. \quad \sum_{j:(i,j)\in A} x_{ij} \le 1 \quad \forall i \in V \\ &x_{ij} \in \{0,1\} \quad \forall (i,j) \in A \end{aligned}$$

注意这并不是一个线性规划问题，因为变量x_{ij}是0-1变量。我们把这样的问题称为**0-1整数规划问题(0-1 integer programming)**。虽然看起来和线性规划差不多，但整数规划问题是NP-难度的，包括最简单的0-1整数规划也一样。

有的读者可能试图通过求解线性规划问题来求解整数规划问题，但稍微尝试后就会发现实际上会遇到很多困难。幸运的是：有一类特殊的整数规划问题却和对应的线性规划问题等价。首先我们定义：如果一个矩阵A的任何子方阵的行列式的值都等于0，1或-1，则称A是**全幺模的(totally unimodular, TU)**。

定理：如果线性规划问题的矩阵A为全幺模矩阵，且该线性规划问题有最优解，则一定存在整数最优解。

可以证明，有向图的关联矩阵为全幺模矩阵（流网络中的整流定理是它的简单推论），而无向图的关联矩阵为全幺模矩阵当且仅当该图为二分图。因此二分图最大权匹配问题可以直接把约束$x_{ij} \in \{0,1\}$改成$x_{ij} \ge 0$，这样该问题就是一个线性规划问题了。那一般图呢？由于矩阵A不是全幺模矩阵，不能直接去掉整数约束，必须想办法把它“转化”为二分图。显然一般图的复杂之处在于奇圈。设包含奇数个（但至少包含3个）结点的奇点集为OS，考虑其中任意奇点集S_k，设它包含$2s_k$+1个结点。记i和j都在S_k中的弧集合为T(S_k)，则显然有

$$\sum_{x_{ij}} \le s_k \quad \forall S_k \in OS$$

因为一个点最多在一条匹配边上。可以证明（但很烦琐，略），加上这条约束后x_{ij}的整数约束就可以去掉，任意图的最大权匹配问题转化为了线性规划问题。

但注意到这样做会使约束的个数是指数级别的，我们甚至不能直接用单纯形法求解。但原始-对偶算法仍然有效，因为我们可以同时处理这一类特殊约束，而不是把它们等同于一般约束。

互补松弛条件。对顶点约束和奇点集分别引入对偶变量u_i和z_k，定义

$$\pi_{ij} = u_i + u_j - w_{ij} + \sum_{k:i,j\in B_k} z_k$$

则互补松弛条件为：

$$\begin{aligned}
&\pi_{ij}x_{ij} = 0, \quad \forall (i,j) \in A \\
&\left(\sum_{(i,j)\in T(S_k)} x_{ij} - s_k\right) z_k = 0 \quad \forall S_k \in OS\left(|S_k| = 2s_k + 1\right) \\
&\left(\sum_{j:(i,j)\in A} x_{ij} - 1\right) u_i = 0 \quad \forall i \in V
\end{aligned}$$

或者用更通俗的语言可以这样叙述匹配的最优性条件：

$$\begin{aligned}
&(0) u_i, \pi_{ij}, z_k \geq 0 \\
&(1) (i,j) \in M \Rightarrow \pi_{ij} = 0 \\
&(2) x_i = 0 \Rightarrow u_i = 0 \\
&(3) z_k > 0 \Rightarrow |\{(i,j) | i,j \in S_k, (i,j) \in M\}| = s_k
\end{aligned}$$

其中(2)中的x_i表示i是未盖点，而(3)的结论表示奇点集S_k是满的，即点集中的匹配边数达到了最大可能值s_k。需要特别指出的是：虽然上述条件是从互补松弛定理推导出，但其实我们还可以用更为简单方法证明如果(0)∼(3)满足，则匹配M一定是最佳匹配。事实上，只需要任意再取一个匹配N，则

$$\sum_{(i,j)\in N} w_{ij} \leq \sum_{(i,j)\in N} (u_i + u_j) + \sum_{(i,j)\in N} \sum_{k:i,j\in S_k} z_k \leq \sum_i u_i + \sum_k s_k z_k = \sum_{(i,j)\in M} w_{ij}$$

其中第一个不等式是因为$\pi_{ij} \geq 0$，第二个是因为$u_{ij}, z_k \geq 0$，且N是一个匹配，而最后的等式是根据(2)，(3)和M是一个匹配。

Edmonds算法。算法从空匹配开始，设置所有u_i都等于$\frac{1}{2} \max\limits_{i,j:(i,j)\in A} w_{ij}$。由于此时没有花朵，因此也就没有$z_k$。这时显然除了(2)的所有条件都得到满足，Edmonds算法就是在始终保持(0),(1),(3)得到满足的前提下最终使得(2)也得到满足。

为了满足(1)，只有$\pi_{ij}=0$的弧(i,j)才可以是匹配边；为了满足(3)，$z_k>0$的奇点集一定是花朵（否则匹配边不够多）。这样，虽然奇点集的个数是指数级别的，但由于我们只需要考虑O(n)个花朵，但Edmonds算法是多项式时间算法。为了让(2)得到满足，我们必须保证不存在u$_i$¿0的未盖点。这样，Edmonds算法每次从一个u$_i$¿0的点开始找增广路，就可以使得u$_i$¿0的未盖点越来越少，最终满足(2)。

这样，Edmonds算法的主框架为：每次从u$_i$¿0的未盖点开始，调用一般图最大奇数匹配算法寻找增广路。如果找到了则进行增广，否则修改对偶变量。对比最小费用流和二分图最大权匹配的原始-对偶算法可以看出，这些算法都是不断改进解（最小费用流是用最大流增广，KM算法是扩展匈牙利树以找增广路），当解无法改进时修改对偶变量，以扩大子图（最小费用流是扩大允许网络，二分图最大权匹配是扩大相等子图），使得扩大后的子图可以改进。

对偶调整(dual adjustment)。受到KM算法的启发，考虑给各个对偶变量增加或减少一个相同的常数保证互补松弛条件的同时让其中一些对偶变量变成0。这里也类似，但是由于有四类对偶变量，我们需要分别考虑。

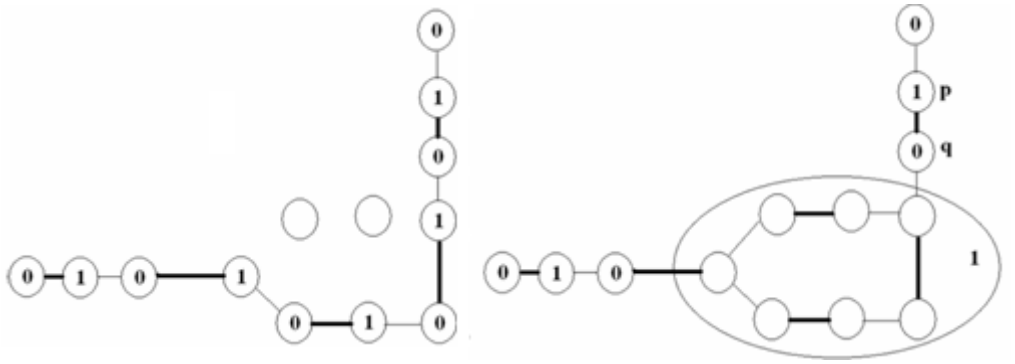
对偶变量的修改。选择一个常数$\delta>0$，每个偶点u$_i$减少δ，每个奇点u$_i$增加δ；每个偶花朵增加2δ，每个奇花朵减少2δ。不难看出，这样的修改保持了(1)和(3)，为了保持非负性(0)，我们需要保证每类对偶变量都是非负的。

取δ_1为所有偶点u$_i$的最小值，δ_2为所有满足i是偶点，j是未标号点的π_{ij}的最小值，δ_3是所有不在同一个花朵中的偶点i和j对应的π_{ij}最小值的1/2，而δ_4为所有奇花朵S$_k$的z$_k$最小值的1/2，则取δ为四个δ_i中最小值，非负条件得到满足。

情况一 如果$\delta=\delta_1$，显然修改后不存在u$_i$¿0的点了，(2)得到满足，得到最大权匹配。

情况二 如果$\delta=\delta_4$，需要展开取到最小值的所有奇花朵S$_k$（变为0后无法继续减小，如果不展开它以后就再以无法进行对偶调整了），对应的z$_k$变为0（因为花朵S$_k$已经不存在）。注意到展开花朵时需要对展开后的结点正确标号，如下图，设使得S$_k$得到偶标号的边为pq，则从花基到q有两条路。奇路径上的点除了起点和终点外全部变成未标号结点（注意它们的关联边可能需要加到队列中等待处理），而偶路径上的点交替标号，如果队列非空则继续标号过程（有希望找到新增广路），否则只能再次修改对偶变量（仍然无法增广）。

情况三 如果$\delta=\delta_2(\delta=\delta_3)$，则对于取到最小值的弧(i,j)，i是偶点，j是未标号点（和i不在同一花朵中的偶点），该弧的π_{ij}变为0，因此被加到队列中，继续寻找增广



路。

算法实现。前面说过，算法一共有O(n)个阶段，每个阶段寻找增广路并增广，而每个阶段中对偶调整的次数不超过5n/3。为了证明这一点，我们分别考虑三种情况。情况一在整个算法中最多出现一次；情况二是展开奇花朵，由于每个花朵至少有3个结点，因此在阶段开始时表面图中最多n/3个奇花朵，而本阶段中由于每次收缩的花朵都是偶花朵，而偶花朵的对偶变量总是增加，不会被展开，所以阶段内的奇花朵数递减，因此情况二最多出现n/3次。情况三每次要么给一个新结点标号（最多n次），要么收缩得到偶花朵（最多n/3），因此情况三最多出现4n/3次。每个阶段的最后我们都需要展开所有满足z_k=0的偶花朵S_k，但所有在花朵中寻找增广路的操作都延迟到花朵展开后。

用支持分裂操作的并查集可以在O(m+nlogn)内在一个阶段中维持表面图，而计算δ并进行对偶调整需要更多的技巧。现在最简单的方法是检查所有边并更新，每次调整需要O(m)时间，一个阶段共需要O(nm)时间，因此所有阶段一共需要$O(n^2m)$时间。Lawler和Gabow算法对每个未标号点v保留π值最小的关联边，并对每个奇点保存它和偶点之间的π值最小边，还对每个偶花朵保存它和其他偶花朵之间的π值最小边。这样，每次对偶调整的时间减小到O(n)，总时间减少到$O(n^3)$。

至此，我们已经粗略叙述了一般图最大权匹配的原始-对偶算法。可能有的读者可能已经看出：二分图最大权匹配的KM算法和这里算法在很多地方是类似的：它也是只考虑满足特定条件的弧，在找不到增广路时修改对偶变量，并且通过记录局部最小值降低对偶调整的开销。由于二分图的特殊性，找增广路和对偶调整要容易很多，但其中的思想在一般图最大权匹配问题中仍然适用。

7.6.5 网络单纯形法

最后，我们简单的介绍网络单纯形法。这是一个非常通用的方法，适合解决一大类网络优化问题。虽然核心是一般线性规划问题的单纯形法，但对于网络优化这类特殊的问题，我们有一些特殊的技巧可以使用。

单纯形的基本思路是从一个基可行解开始用旋转操作不断转移到其他基可行解，逐渐把目标值调整到最优，网络单纯形法也是一样，但我们需要先研究一下基在网络中的直观意义。

考虑容量无解的最小费用流问题的松弛形式：

$$\begin{aligned} &\min c^T x \\ &s.t. \quad Bx = d \\ &x \geq 0 \end{aligned}$$

则关联矩阵B构成一组基的充要条件是：B中对应的弧为生成树。这句话不是很容易懂，我们可以对比一下：单纯形法的每一步用非基变量来表示基变量，把所有非基变量都取0时求出基变量得到的所谓基解。在最小费用流问题中，生成树上T的弧流量对应基变量，其他弧流量对应非基变量，则其他弧流量取0流后能确定出T中弧的流量，得到所谓的树解。现在问题的关键是：给定生成树T，若其他弧流量均为0，T中每条弧的流量如何计算？考虑叶子i，设它的供给为d_i，则该i与父亲j的弧的流量一定为d_i（想一想，为什么），然后可以把i删除，并给j的供给增加d_i。重复上述过程，即可在O(n)时间内算出T对应的基解。

需要注意的是，这样算出的流量可能是负的，违反了容量约束，因此并不是每棵生成树T都对应可行解。我们把对应可行解的生成树T称为可行树。

旋转操作。什么时候树解是最优的呢？容量无界时，互补松弛条件为：

$$\begin{aligned} c_{ij}^{\pi} = c_{ij} - \pi_i + \pi_j \geq 0 \quad x_{ij} = 0 \\ c_{ij}^{\pi} = c_{ij} - \pi_i + \pi_j = 0 \quad x_{ij} > 0 \end{aligned}$$

由于只有树弧上的流量可能为正，所以只有树弧可能满足第二个式子。由于树弧只有n-1条而对偶变量有n个，因此在相差一个常数的意义下，由树T的弧集合可能唯一确定对偶变量，即结点势π_i。我们随便指定一个树根，另它的势为0，然后用互补松弛条件不断计算相邻结点的势，最后验证第一个式子是否成立。

如果式子不成立，则一定存在非树弧(p,q)使得$c_{pq}^{\pi} = c_{pq} - \pi_p + \pi_q < 0$，此时弧(p,q)可以进基。进基后网络中出现了一个圈W，设它的方向为弧(p,q)的方向，则圈

的总费用为

$$C(W)=\sum_{(i,j)\in W^+}c_{ij}-\sum_{(i,j)\in W^-}c_{ij}=\sum_{(i,j)\in W^+}(c_{ij}-\pi_i+\pi_j)-\sum_{(i,j)\in W^-}(c_{ij}-\pi_i+\pi_j)=c_{pq}^{\pi}<0$$

因此沿它进行增广后费用减小。为了在W中找出一条弧出基，我们应当令增广流量为W^-中所有弧流量的最小值（这样增广后才不会出现负流量），如果W^-为空，则最小费用可以趋于负无穷大。

处理退化。在单纯形法中介绍过：旋转后目标函数不变的情况称为退化，退化可能引起循环，导致单纯形法永远无法终止。在最小费用流问题中，退化对应于“找到负费用增广圈却无法增广”，因为圈W中有个反向弧流量也为0。计算测试表明网络单纯形法90%的旋转都是退化的，因此必须专门处理。

由于网络单纯形法的每一步都是一个可行树，我们只要保证每个可行树各不相同，就不会出现循环。注意这里的不相同是把树看成有标号的，实际上各个不同的可行树可能同构。注意到退化的根源是树T中的0流弧，我们引入强可行树的概念，对这些0流弧加以限制。

网络单纯形法。考虑计算结点势时选择的树根r，对于任意弧(i,j)，如果路r-j经过i，称弧(i,j) 是下行弧(downward pointing arc)。如果可行树中所有流量为0的弧都是下行弧，则称该树T是前可行树。可以证明，如果旋转操作保证产生强可行树，则这些树的势是严格递减的，因此互不相同。

如何保证这一点呢？我们需要做两件事情：

一、初始解应为强可行树。

二、旋转时产生强可行树。

为了保证第二点，出基弧不能任意选择。设入基弧为(p,q)，则所有可能的出基弧为W^-中取到最小流量的所有弧。记结点u为T中从根到p的路与圈的第一个交点，则选取出基弧(k,l)为从u出发沿W 正向前进时第一次遇到的候选出基弧。

第一点可以采用和线性规划类似的大M法解决：增加人工结点0并和所有结点连一条弧，正的供给连正向弧，负的供给连反向弧，容量全部为一个充分大的数M。设所有人工弧集为T，则T是新网络的强可行树，可以从它开始求解新网络的最小费用流。如果新网络无有界最优解，则原问题也无有界最优解；否则当所有人工弧流量为0时删除人工弧即可得到原网络的最小费用流，而若有的人工弧流量不为0，则原问题不可行。

多大的M才算“充分大”？理论指出，M¿(n-1)C/2时就可以满足上述要求，其中C为所有弧的最大费用，实际应用中也可以采取自适应的方法，先选择规模适中的M，如果求解后某些人工弧流量不为0则增大M重新计算。

容量有界的情形。刚才考虑的是容量无界的情况，现在加入容量上下界，则非树弧的流不再一定都是0，而是有的等于下界，有的等于上界，分别称为**空弧(empty arc)**和**满弧(full arc)**。界于二者之间的弧称为**部分弧(partial arc)**。这样，在任意时刻网络中的弧可以用三元组(T,L,U)表示，分别为树弧，流量等于下界的弧和流量等于上界的弧。这个三元组称为基结构。给定基结构，可以用和刚才完全类似的方法计算树弧中的流量和结点势。强可行树的定义需要修改为：L弧都是下行弧，而U弧都是上行弧，而初始可行解构造时所有原始弧均取下界，而选出基弧的规则一样：在候选弧集中选择从u出发沿W正方向行走遇到的到一条弧。

7.6.6 小结

本节介绍线性规划的基本概念、对偶原理和单纯形法，并用它分析最小费用流问题和一般图最大权匹配问题，给出了原始-对偶算法和网络单纯形法的例子。本节内容非常抽象，程序也比较复杂，供学有余力的同学参考。

本节的算法和程序列表如下：

算法	程序	备注
单纯形法	simplex.cpp	用单纯形法求解线性规划问题的标准形式，包括初始可行解的确定和旋转操作，并用Bland规则避免循环
原始-对偶算法	mincost_pd.cpp	用原始-对偶算法求解最小费用流问题。这个算法的效率并不高，只是作为例子说明该思想的应用
Lawler-Gabow算法	gabow.cpp	求解一般图最大权匹配的Lawler-Gabow算法，该算法是Edmonds的基本原始-对偶算法的改进版，时间复杂度为$O(n^3)$。
网络单纯形法	netsimplex.cpp	求解最小费用流问题的网络单纯形法，不是强多项式时间复杂度的，但实际效果很不错。

第8章　几何基本问题

本章讨论几何问题，包括经典的代数方法和Shamos计算几何的经典算法。

8.1　代数方法

在接触到具体算法之前，我们需要熟悉向量代数。向量空间(Vector space)是所有向量的集合，加上两个特别的操作：

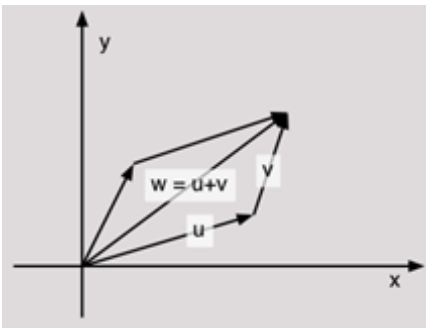
向量加法(vector addition)：满足交换律，结合律，单位元为0向量，每个向量的逆元-v。

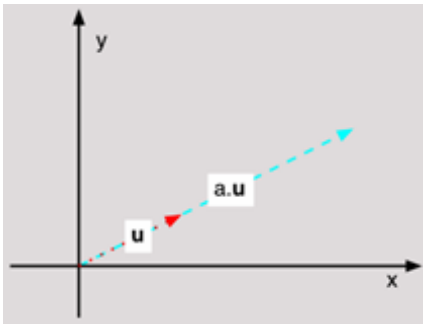
数量乘法(scalar multiplication)：满足交换律且对加法满足分配律。

把数量乘法和向量乘法合在一起构成了**线性组合(linear combination)**，即（注意向量应该加箭头或者用黑体表示）：

$$a_1\overrightarrow{v_1} + a_2\overrightarrow{v_2} + \ldots + a_n\overrightarrow{v_n}$$

如果选一个原点，那么每个向量都对应于一个点。这样可以用几何的方法表示向量加法和数量乘法，其中向量加法法则也称为**平行四边形法则（the parallelogram rule）**。





最常用的向量空间是R^n，即n维实数向量。

另一个重要的概念是**仿射空间(affine space)**，它是一个点集，且不像向量空间中那样有原点这样的“特殊点”。在向量代数中，我们需要把点和向量区分开，方法是给把坐标写成行向量，且在最后加一列，用0表示向量，1表示点。这样，很容易得到：

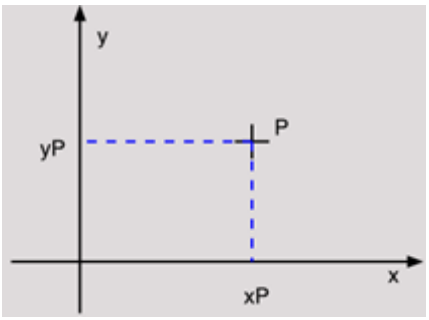
向量+ 向量= 向量（因为0+0=0）
点+ 向量= 点（因为1+0=1）
点+ 点无定义（因为1+1=2，2无定义）
点– 点= 向量（因为1-1=0）
向量* 数= 向量（因为0*任何数=0）
点* 数无定义（因为1*任何数不都有定义）

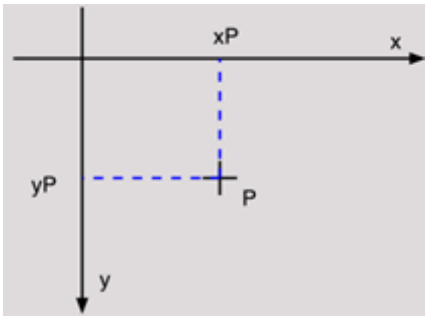
这样，我们用同样的形式把向量和点统一起来了。在结果有意义的情况下，三维空间的点和向量都满足矩阵加法，即每个分量对应相加。这样做的好处是显然的：点和向量的统一将简化很多复杂的公式，得到漂亮的形式。在这种表示法下，n维向量空间和n维仿射空间都是R^{n+1}的子空间。

在仿射空间中还有一种特殊的线性组合。在二维情形下为$P + t(Q - P)$。注意在仿射空间中两个点的加法是没有意义的，但在这里t(Q-P)是一个向量，点和向量的加法是有意义的。因此当a+b=1的时候还是可以定义一种类似向量空间里的线性组合，即$aP + bQ = (1 - b)P + bQ = P - b(Q - P)$。推广到一般情形的凸组合，即当$t_1 + t_2 + \ldots + t_n = 1$时$t_1P_1 + t_2P_2 + \ldots + t_nP_n = P_1 + t_2(P_2 - P_1) + \ldots + t_n(P_n - P_1)$。

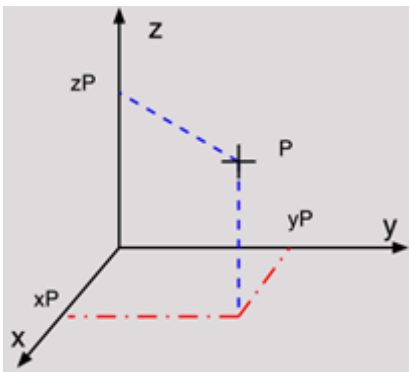
显然，在仿射空间中过两点P和Q的直线方程为(1-t)P+tQ，过三个非共线点P, Q, R的平面方程为(1-s)((1-t)P+tQ)+sR。

后面的讨论将针对向量空间和仿射空间。所有内容都在笛卡儿坐标系(Cartesian coordinate system)下（如图(a)），而不是计算机的常用坐标系（如图(b)）。





而三维情形下本书一律用左手系，如下图。



8.1.1 向量乘积与仿射变换

在本节中，我们将学习向量的点积、叉积和张量积。

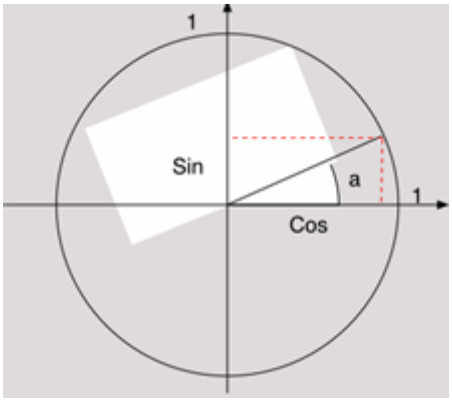
点积(dot product)。点积的定义来源于线性代数，它的定义是：

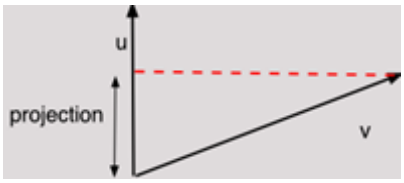
$$\overrightarrow{u} \cdot \overrightarrow{v} = [u_1\ u_2\ \cdots\ u_n] \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

在坐标表示下，$\overrightarrow{u} \cdot \overrightarrow{v} = u_1v_1 + u_2v_2 + u_3v_3$。用它很容易求出两个向量的夹角，因为

$$\cos\theta = \frac{\overrightarrow{u} \cdot \overrightarrow{v}}{\|\overrightarrow{u}\|\,\|\overrightarrow{v}\|}$$

如下图，它的几何意义是u在v投影后二者长度的乘积。角度可以用下面的圆来理解。





点积很容易写成程序：

注意：点积是标量而不是向量。后面我们将看到，它通常被用来求角度和判前后。它也被用来计算距离，因为$dist(P,Q) = \|\overrightarrow{u}\| = \sqrt{\overrightarrow{u} \cdot \overrightarrow{u}}$。这样看起来比较迂回，但是在推导公式时可能会让过程变得简单。

```
double dot(Vector3D u, Vector3D v)
{
return u[1]*v[1]+u[2]*v[2]+u[3]*v[3];
}
```

叉积(cross product)。叉积的定义由三个性质给出：长度、正交性和方向。根据这些性质可以推导出三维叉积的直接计算公式。

$$\overrightarrow{u} \times \overrightarrow{v} = \begin{bmatrix} u_2v_3 - u_3v_2 & u_3v_1 - u_1v_3 & u_1v_2 - u_2v_1 & 0 \end{bmatrix}$$

叉积和u、v都垂直，方向由**右手定则(right handed rule)**给出。长度是u和v组成的平行四边形的面积，即

$$\|\overrightarrow{u} \times \overrightarrow{v}\| = \|\overrightarrow{u}\| \cdot \|\overrightarrow{v}\| \cdot \sin\theta$$

在更复杂的情况中，需要把叉积表示成矩阵形式。当还有很多其他操作时，矩阵乘法将发挥很大作用。推导过程略，这里只给出它的计算公式，包括两个对称的形式，即

$$\overrightarrow{u} \times \overrightarrow{v} = \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix}$$

$$\overrightarrow{v} \times \overrightarrow{u} = \begin{bmatrix} 0 & v_3 & -v_2 \\ -v_3 & 0 & v_1 \\ v_2 & -v_1 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

注意对于二维向量，它们的叉积有时只看成一个有符号的数，它等于两个向量形成的三角形面积的两倍。

向量叉积通常用来求三角形的面积、判断左右，以及计算平面的**法线(normal)**。它的另一个用处是在三维空间中构造一个以给定向量N为一个轴的三维坐标系，方法如下：

：N和j不垂直，构造单位向量$\overrightarrow{u} = \frac{\overrightarrow{k} \times \overrightarrow{N}}{\|\overrightarrow{k} \times \overrightarrow{N}\|}$作为第二个坐标轴，则第三轴$\overrightarrow{v} = \frac{\overrightarrow{N} \times \overrightarrow{u}}{\|\overrightarrow{N} \times \overrightarrow{u}\|}$。

′：N和j垂直，则直接把j作为第二轴而第三轴为$\overrightarrow{u} = \frac{\overrightarrow{N} \times \overrightarrow{j}}{\|\overrightarrow{N} \times \overrightarrow{j}\|}$。

把刚才的公式写成程序就是：

```
Vector3D cross(Vector3D u, Vector3D v)
{
return Vector3D(u[2]*v[3]-u[3]-v[2], u[3]*v[1]-u[1]*v[3], u[1]*v[2]-u[2]*v[1])
}
Matrix3 GetRightCrossMat(Vector3D u)
{
return Matrix3(0, -u[3], u[2], u[3], 0, -u[1], -u[2], u[1], 0);
}
Matrix3 GetLeftCrossMat(Vector3D u)
{
return Matrix3(0, u[3], -u[2], -u[3], 0, u[1], u[2], -u[1], 0);
}
```

张量积。张量积的结果是一个矩阵，它的定义为：

$$\overrightarrow{u} \otimes \overrightarrow{v} = \begin{bmatrix} u_1v_1 & u_1v_2 & u_1v_3 \\ u_2v_1 & u_2v_2 & u_2v_3 \\ u_3v_1 & u_3v_2 & u_3v_3 \end{bmatrix}$$

显然$(\overrightarrow{u} \otimes \overrightarrow{v})^T = \overrightarrow{v} \otimes \overrightarrow{u}$。对于向量代数一种常见的表达式，可以用张量积重写成：

$$(\overrightarrow{u} \cdot \overrightarrow{v})\overrightarrow{w} = \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} (\overrightarrow{v} \otimes \overrightarrow{w})$$

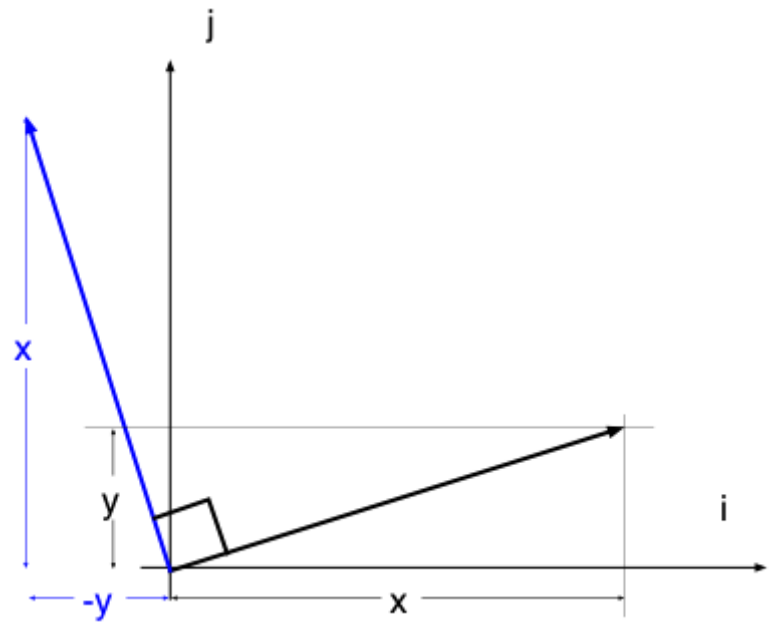
它的本质为将向量$\overrightarrow{u}$变换为一个与$\overrightarrow{w}$平行的向量。张量积的程序为：

```
Matrix3 outer(Vector3D u, Vector3D v)
{
return Matrix3(u[1]*v[1], u[1]*v[2], u[1]*v[3],
u[2]*v[1], u[2]*v[2], u[2]*v[3],
u[3]*v[1], u[3]*v[2], u[3]*v[3]);
}
```

垂直运算和垂直点积。二维向量有种特殊的运算：垂直运算，它把原来的向量逆时针旋转90度，并保持长度不变。显然只需要交换两个分量并把第一个分量取反，如下图。

写成程序即：

和点积不一样，垂直点积不仅反映向量夹角，还反映方向（回忆：-45度和+45度的cos值相同）。把其中一个向量取垂直后再进行点积，有




```
Vector2D perp(Vector2D u)
{
return vector2D(-u[2], u[1]);
}
```

$$\sin\theta = \frac{\overrightarrow{u}^{\perp} \cdot \overrightarrow{v}}{\|\overrightarrow{u}\| \, \|\overrightarrow{v}\|}$$

如果两个向量都是单位向量，则正弦值就是二者的垂直点积。有读者可能已经看出了，垂直点积也等于两向量组成三角形有向面积的两倍。事实上，正如Hill（1994）所指出，垂直点积可以看作三维叉积的二维模拟，它的值和二维叉积一样，即：

```
double perpdot(Vector2D u, Vector2D v)
{
return u[1]*v[2]-u[2]*v[1];
}
```

下面用矩阵来表示仿射变换，这比给出直接计算公式更有启发性，应用范围也更广。在仿射空间中，仿射变换T把一个点P变成另一个点T(P)。可以用一个n+1阶方阵来表示n维仿射空间中的仿射变换，即

$$\overrightarrow{v} = T(\overrightarrow{u}) = \begin{bmatrix} u_1 & u_2 & \cdots & u_n & 1 \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & 0 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & 0 \\ b_1 & b_2 & \cdots & b_n & 1 \end{bmatrix} = \overrightarrow{u}T$$

因此问题的关键是如何用一个矩阵T表示仿射变换。注意到矩阵的特殊性，我们可以把它分块成$\begin{bmatrix} A & \overrightarrow{0}^T \\ \overrightarrow{b} & 1 \end{bmatrix}$，下面的讨论都只给出矩阵A和向量b的表达式。

平移。设平移向量为$\overrightarrow{u}$，则A为单位矩阵，$\overrightarrow{b} = \overrightarrow{u}$，平移就是这么简单。读者也许会说平移不就是每个分量加上一个给定的数么，为啥还要这么麻烦？答案是矩阵乘法有结合律，可以把平移和其他仿射变化加以组合，用单个矩阵来简洁的表示一些复杂变换。

旋转。为了简单起见，我们先考虑旋转中心是原点，轴是x, y, z三个坐标轴之一的旋转。容易得到下面的三个旋转矩阵：

$$T_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & & \\ -\sin\theta & \cos\theta & & \\ & & 1 & \\ & & & 1 \end{bmatrix}, T_x(\theta) = \begin{bmatrix} 1 & & & \\ & \cos\theta & \sin\theta & \\ & -\sin\theta & \cos\theta & \\ & & & 1 \end{bmatrix}, T_y(\theta) = \begin{bmatrix} \cos\theta & & -\sin\theta & \\ & 1 & & \\ \sin\theta & & \cos\theta & \\ & & & 1 \end{bmatrix}$$

注意在二维情形下，就是饶z轴旋转，矩阵是我们熟知的：

$$T(\theta) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}, T(\frac{\pi}{2}) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

旋转90度的变换正好是前面介绍的垂直运算符。对于二维情况，一个一般旋转只需要进行一次平移，把中心移到原点，旋转后再平移一个相反向量，过程并不复杂，代码如下：

```
Vector2D Rotate2D(Vector2D u, double a)
{
return u*Matrix2(cos(a), sin(a), -sin(a), cos(a));
}
Vector2D Rotate2D(Vector 2D u, Point2D c, double a)
{
Return Trans2D(Rotate2D(Trans2D(u, -c), a), c);
}
```

这里利用了2*2矩阵来简化运算，但需要定义向量和矩阵的乘法。

三维情况要复杂得多，因为先要把沿规定轴的旋转角度分解为x、y、z三个轴的旋转分量，并不直观。下面给出一个直接计算公式，以Q为中心，单位向量$\hat{u}$为旋转轴，旋转角度为θ。这个公式有时被成为**旋转公式(rotation formula)**，或**罗德里格斯公式(Rodriguez's formula)**（Hacker 1997）：

$$T = \begin{bmatrix} T_{\overrightarrow{u},\theta} & \overrightarrow{0}^T \\ Q - QT_{\overrightarrow{u},\theta} & 1 \end{bmatrix}, T_{\overrightarrow{u},\theta} = (\cos\theta)I + (1-\cos\theta)(\overrightarrow{u} \otimes \overrightarrow{u}) + (\sin\theta)\tilde{u}$$

其中$\tilde{u}$是在叉积部分中介绍的，“左叉乘u”的矩阵$\begin{bmatrix} 0 & u_3 & -u_2 \\ -u_3 & 0 & u_1 \\ u_2 & -u_1 & 0 \end{bmatrix}$。

缩放。缩放(scale)对于几何体才看得出效果。简单的缩放以原点为中心，三个坐标轴都有一个缩放系数s_x, s_y, s_z，相等时为**均衡缩放(uniform scaling)**，否则为**非均衡缩放(non-uniform scaling)**。由于原点和坐标轴的特殊性，均衡缩放的矩阵是简单的：

$$T = \begin{bmatrix} s_x & & & \\ & s_y & & \\ & & s_z & \\ & & & 1 \end{bmatrix}$$

对于中心不在原点的情形可以类似推导。如果是均衡所放，设缩放因子为s，设T_s=sI，则$T_{s,Q} = \begin{bmatrix} T_s & \overrightarrow{0}^T \\ (1-s)Q & 1 \end{bmatrix}$。非均衡缩放还需要一个表示缩放方向的单位向量$\hat{u}$，可以得到缩放矩阵为：

$$T_{s,\hat{u}} = I - (1-s)(\hat{u} \otimes \hat{u})$$
$$T_{s,Q,\overrightarrow{u}} = \begin{bmatrix} T_{s,\hat{u}} & \overrightarrow{0}^T \\ (1-s)(Q \cdot \hat{u})\hat{u} & 1 \end{bmatrix}$$

当$\hat{u}$为x轴时，$T_{s,\hat{u}} = I - (1-s)(\hat{u} \otimes \hat{u}) = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} - (1-s)\begin{bmatrix} 1 & & \\ & & \\ & & \end{bmatrix} = \begin{bmatrix} s & & \\ & 1 & \\ & & 1 \end{bmatrix}$，符合简单缩放的情形。

反射。反射是相对于一条直线（在二维空间中）或一个平面（三维空间中）的镜像点变换。反射的一个特别重要的性质是反转方向。

我们仍然先说明简单反射。在二维空间中，简单反射是沿坐标轴的反射。沿x轴和y轴的反射矩阵分别为：

$$T_x = \begin{bmatrix} 1 & & \\ & -1 & \\ & & 1 \end{bmatrix}, T_y = \begin{bmatrix} -1 & & \\ & 1 & \\ & & 1 \end{bmatrix}$$

在三维空间中，有沿yz, xz和xy平面的三种反射，显然有如下反射矩阵：

$$T_{yz} = \begin{bmatrix} -1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}, T_{xz} = \begin{bmatrix} 1 & & & \\ & -1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}, T_{xy} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & -1 & \\ & & & 1 \end{bmatrix}$$

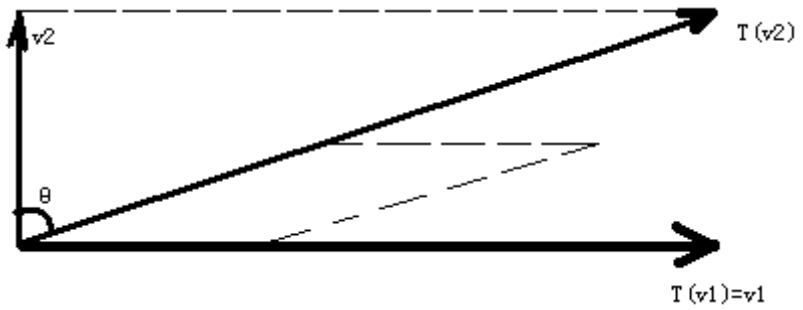
在一般反射中，需要定义一个点Q和向量。在二维空间中向量$\hat{d}$是直线的方向，三维空间中向量$\hat{n}$是平面的法向量。二维情形下的反射矩阵为：

$$T_{\hat{d}} = \left[I - 2(\hat{d}^{\perp} \otimes \hat{d}^{\perp}) \right]$$
$$T_{\hat{d},Q} = \begin{bmatrix} T_{\hat{d}} & \overrightarrow{0}^T \\ 2(Q \cdot \hat{d}^{\perp})d^{\perp} & 1 \end{bmatrix}$$

看似多余的垂直运算达到了二维和三维的统一：

$$T_{\hat{n},Q} = \begin{bmatrix} T_{\hat{n}} & \overrightarrow{0}^T \\ 2(Q \cdot \hat{n})\hat{n} & 1 \end{bmatrix}$$

剪切。剪切是一种很有意思的运算，它只是变形而面积不变（没有缩放）。“斜体字”可以近似的看成正体字的剪切。剪切仍然是沿直线进行，剪切后该向量保持不变，而它的垂直向量产生倾斜。下面指定剪切角度而另一些书籍使用剪切比例。剪切角度的几何意义如下：



二维中的两个剪切矩阵为：

$$T_{xy,\theta}=\begin{bmatrix}1 & & \\ \tan\theta & 1 & \\ & & 1\end{bmatrix}, T_{yx,\theta}=\begin{bmatrix}1 & \tan\theta & \\ & 1 & \\ & & 1\end{bmatrix}$$

在三维空间中，设$H_{\eta\gamma}=\tan\theta_{\eta\gamma}$，则剪切矩阵为：

$$\begin{bmatrix}1 & H_{yx} & H_{zx} & \\ H_{xy} & 1 & H_{zy} & \\ H_{xz} & H_{yz} & 1 & \\ & & & 1\end{bmatrix}$$

注意这些H项最多只能有一个非0元素，多个剪切的组合必须用多个矩阵相乘来实现。

一般的剪切矩阵有多种建立方法。Goldman（1991）给出了一种方法，得到结果如下：

$$T_{Q,\hat{n},\vec{v},\theta}=\begin{bmatrix}1+\tan\theta(\hat{n}\otimes\hat{n}) & \overrightarrow{0}^T \\ -(Q\cdot\hat{n})\hat{v} & 1\end{bmatrix}$$

另一种方法是记$H=\begin{bmatrix}1 & \tan\theta & \\ & 1 & \\ & & 1\end{bmatrix}, R=\begin{bmatrix}\hat{n} \\ \hat{v} \\ \hat{n}\times\hat{v}\end{bmatrix}$，则

$$T_{\hat{n},\hat{v},Q,\theta}=\begin{bmatrix}R^THR & \overrightarrow{0}^T \\ Q(I_3-R^THR) & 1\end{bmatrix}$$

平行投影。正交投影是最简单的情形，它的矩阵为：

$$\begin{aligned}T_{\hat{u}}&=I-(\hat{u}\otimes\hat{u})\\ T_{\hat{u},Q}&=\begin{bmatrix}T_{\hat{u}} & \overrightarrow{0}^T \\ (Q\cdot\hat{u})\hat{u} & 1\end{bmatrix}\end{aligned}$$

一般地，投影方向为$\hat{w}$的平行投影的矩阵为：

$$\begin{aligned}T_{\hat{u},\hat{w}}&=I-\frac{(\hat{u}\otimes\hat{w})}{\hat{w}\cdot\hat{u}}\\ T_{\hat{u},Q,\hat{w}}&=\begin{bmatrix}T_{\hat{u},\hat{w}} & \overrightarrow{0}^T \\ \frac{Q\cdot\hat{u}}{\hat{w}\cdot\hat{u}}\hat{w} & 1\end{bmatrix}\end{aligned}$$

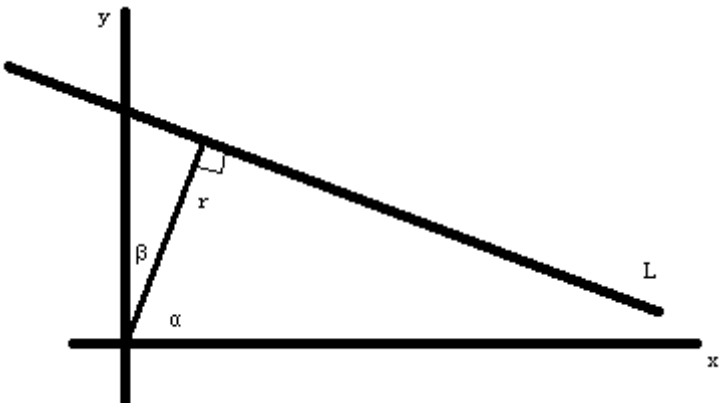
8.1.2 二维图元及其算法

本节讨论一些具体问题，包括二维图元的表示及其运算，包括距离、角度的求解等。二维图元中最简单的就是直线，它有很多

直线的表示。一条直线(line)有两种表示法：一般式和参数式。一般式为ax+by+c=0。注意到给a, b, c同时乘以一个非零数也行，所以当a和b不全为零时我们往往把它规整化，即两边同时乘以$\frac{1}{\sqrt{a^2+b^2}}$它可以写成

$$\begin{aligned} a &= \cos\alpha \\ b &= \cos\beta \\ c &= \|\overrightarrow{r}\| \end{aligned}$$

这样，a和b是垂直于直线的向量的x和y分量，而c就是直线到原点的最小（有符号）距离，如下图。



直线的**参数形式(parametric form)**为$X(t) = P + t\overrightarrow{d}, t \in \mathrm{R}$，则射线就是简单的限制$t \geq 0$，线段是限制$t \in [t_0, t_1]$。线段的另一种形式是给定端点$\mathrm{P}_0$和$\mathrm{P}_1$，则参数式是$X(t) = (1-t)P_0 + tP_1, t \in [0,1]$，它可以通过设$\overrightarrow{d} = P_1 - P_0$而转化为参数形式。

二者的转化并不复杂。对于参数形式的直线：

$$\begin{aligned} x &= P_x + td_x \\ y &= P_y + td_y \end{aligned}$$

其等价的一般式为：$-d_y x + d_x y + (P_x d_y - P_y d_x) = 0$

反过来，对于一般式ax+by+c=0，等价的参数式为：

$$\begin{aligned} P &= \begin{bmatrix} \frac{-ac}{a^2+b^2} & \frac{-bc}{a^2+b^2} \end{bmatrix} \\ \overrightarrow{d} &= \begin{bmatrix} -b & a \end{bmatrix} \end{aligned}$$

三角形(triangle)。三角形是最简单的多边形，很多多边形的问题都可以转化为三角形解决。它有三个顶点P_0, P_1和P_2，如果把P_0看作原点，则边向量$e_0 = P_1 - P_0, e_1 = P_2 - P_0$。它可以用参数形式表示：

$$X(t_0,t_1)=P_0+t_0\overrightarrow{e_0}+t_1\overrightarrow{e_1},\quad t_0,t_1\in[0,1],0\le t_0+t_1\le 1$$

也可以用重心形式表示：

$$X(c_0,c_1,c_2)=c_0P_0+c_1P_1+c_2P_2,\quad c_i\in[0,1],\quad c_0+c_1+c_2=1$$

三角形的有向面积为：

$$Area(P_0,P_1,P_2)=\frac{1}{2}\det\begin{bmatrix}1&1&1\\x_0&x_1&x_2\\y_0&y_1&y_2\end{bmatrix}=\frac{1}{2}((x_1-x_0)(y_2-y_0)-(x_2-x_0)(y_1-y_0))$$

“有向”的含义是：如果面积为正，则三个顶点逆时针排列；如果为负，则顺时针排列；如果为零，则三点共线。程序如下（结果为有向面积的两倍）：

```
Int Area2(tPointi a, tPointi b, tPointi c)
{
Return (b[x]-a[x])*(c[y]-a[y])-(c[x]-a[x])*(b[y]-a[y]);
}
```

用分割多边形的方法可以得到一般多边形的有向面积公式，即：

```
Int AreaPoly(void)
{
Int sum = 0;
tVertex p, a;
p = vertices; // fixed
a = p-¿next;
do{
sum += Area2(p-¿v, a-¿v, a-¿next-¿v);
a = a-¿next;
}while(a-¿next != vertices);
}
```

有一个很有趣的细节：基点是多边形的其中一个顶点而不是原点的，否则在多边形本身很小但坐标值很大时不会乘法溢出。

二次曲线(quadratic curves)。有时候会接触到二次曲线，其中圆是最常见的一种。一般二次曲线由两个变量的二次方程所隐含确定，即：

$$a_{00}x_0^2 + 2a_{01}x_0x_1 + a_{11}x_1^2 + b_0x_0 + b_1x_1 + c = 0$$

设

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}, B = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}, X = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

则二次曲线方程的矩阵形式为：

$$X^TAX + B^TX + c = 0$$

对A进行特征值分解得$A = R^TDR$，其中R为旋转矩阵，D是对角阵。定义E=RB，Y=RX，则方程变为：

$$Y^TDY + E^TY + c = d_0y_0^2 + d_1y_1^2 + e_0y_0 + e_1y_1 + c = 0$$

可以根据D的对角元素d_0和d_1来进行分类。例如若$d_0 \neq 0$，则有

$$d_0y_0^2+e_0y_0 = d_0\left(y_0^2 + \frac{e_0}{d_0}y_0\right) = d_0\left(y_0^2 + \frac{e_0}{d_0}y_0 + \frac{e_0^2}{4d_0^2} - \frac{e_0^2}{4d_0^2}\right) = d_0\left(y_0 + \frac{e_0}{2d_0}\right)^2 - \left(\frac{e_0}{2d_0}\right)^2 \quad d_0 \neq 0, d_1 \neq$$

$$d_0\left(y_0 + \frac{e_0}{2d_0}\right)^2 + d_1\left(y_1 + \frac{e_1}{2d_1}\right)^2 = \frac{e_0^2}{4d_0} + \frac{e_1^2}{4d_1} - c = r$$

当$d_0d_1 > 0$时有四种情况：

情况	$d_0r < 0$	$d_0r = 0$	$d_0r > 0$	
			$d_0 = d_1$	$d_0 \neq d_1$
形状	无解	点	圆	椭圆

当$d_0d_1 < 0$时$r \neq 0$为双曲线，$r = 0$是两条相交的线。

情况二：$d_0 \neq 0$且$d_1 = 0$，分解为：

$$d_0\left(y_0 + \frac{e_0}{2d_0}\right)^2 + e_1y_1 = \frac{e_0^2}{4d_0} - c = r$$

当$e_1 \neq 0$时解为抛物线，否则有三种情形，如下表：

情况三：$d_0 \neq 0$且$d_1 = 0$，和情况二对称一致。

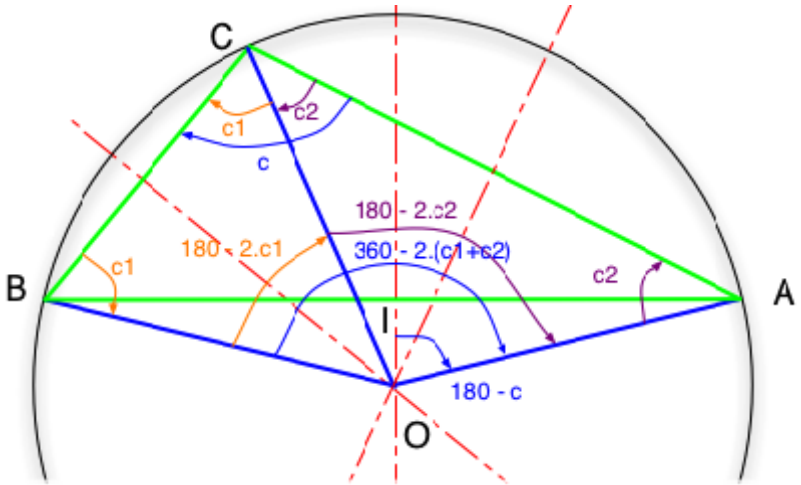
情况	$d_0r < 0$	$d_0r = 0$	$d_0r > 0$
形状	无解	一条直线	两平行直线

情况四：$d_0 = 0$且$d_1 = 0$。方程为：

$$e_0y_0 + e_1y_1 + c = 0$$

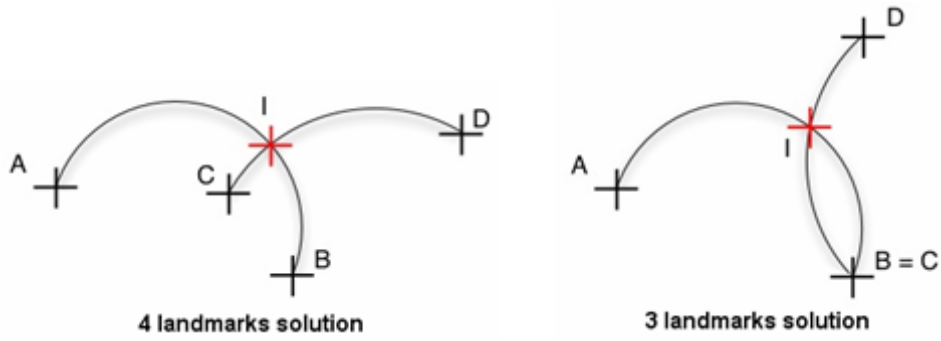
如果$e_0 = e_1 = 0$，则当$c \neq 0$时，无解；当$c = 0$时，方程变为$0 = 0$。如果$e_0 \neq 0$或$e_1 \neq 0$，则解为一条直线。

圆(circle)。圆是一个很有意思的几何元，它通常和角度相关。考虑定位问题：如果我看到两个标志(landmark)，即已知位置的点，我能知道我自己的位置吗？注意只能看到二个点和研究所成的角度，而无法确定距离。答案是否定的，因为满足条件的点有很多。我们把这样的点的集合称为**轨迹(locus)**。



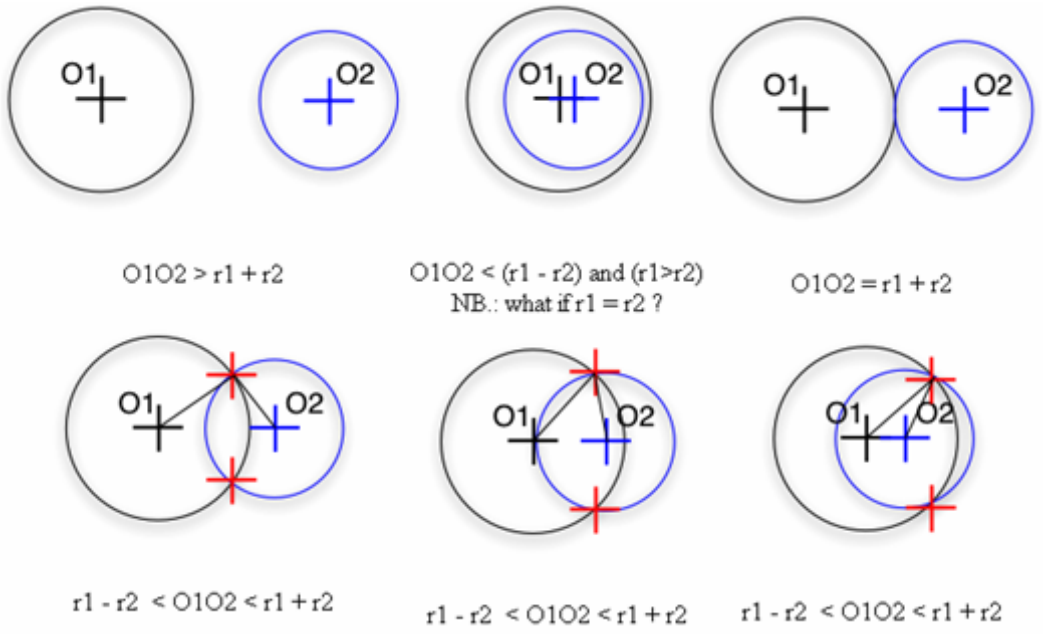
上图是一个例子，当视角不等于90度时轨迹是弧AB。注意AB之间的弧有两段，需要根据视角和90度的大小关系进行选择。比如当视角为100度时应选择劣弧。由AB和视角很容易算出O的位置，因为$\angle C + \frac{1}{2}\angle O = 180°$。设AB的中点为I，则由$\Delta AIO$的三角关系容易算出OI的值，从而得到O的位置。如果需要得到确切位置，需要有3~4的位置，如下图：

不管哪种情况，都需要计算圆弧交点。设两个圆心为$O_1(x_1, y_1)$和$O_2(x_2, y_2)$，半径分别为r_1和r_2，设交点为M(x, y)，则写成方程就是：



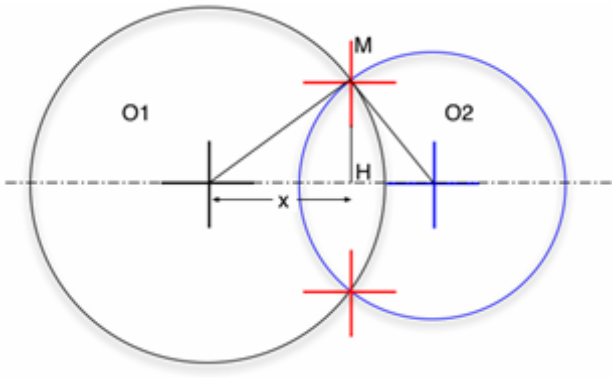
$$\begin{cases} (x-x_1)^2+(y-y_1)^2=r_1^2 \Leftrightarrow x^2-2xx_1+x_1^2+y^2-2yy_1+y_1^2=r_1^2 \\ (x-x_2)^2+(y-y_2)^2=r_2^2 \Leftrightarrow x^2-2xx_2+x_2^2+y^2-2yy_2+y_2^2=r_1^2 \end{cases}$$

两式相减得到关于x和y的一元二次方程，比较复杂。另一种方法是用几何性质直接计算。注意到两个圆的位置关系有如下四种（后三中的区别是双圆的圆心在不在另一个圆内）。



前三种情况都很容易计算（注意情况二中两圆半径相等的情形），关键是后三种。以第四种情况为例：

设$O_1H=x$，MH为h，$O_1O_2=d$，则$HO_2=d-x$，我们有：



$$\begin{cases} R_1^2 = h^2 + x^2 \\ R_2^2 = h^2 + (d-x)^2 = h^2 + d^2 - 2dx + x^2 \end{cases} \Rightarrow R_2^2 - R_1^2 = d^2 - 2dx$$

解这个新方程并代入第一个方程，得：

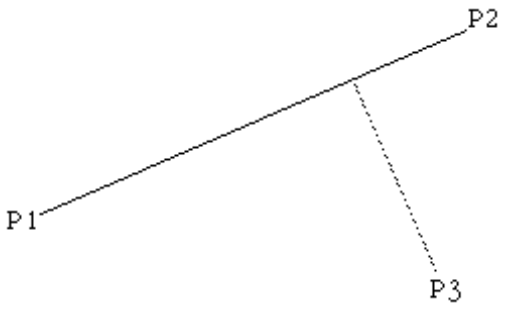
$$\begin{cases} x = (d^2 - R_2^2 + R_1^2)/2d \\ h = \pm\sqrt{R_1^2 - x^2} \end{cases}$$

因此所求结果为：

$$M = O_1 + O_1H \pm HM$$

比代数法简单了许多。这个方法很容易推广到三维情形，即两个球的相交。把上图看成两个球的投影，则x和h的表达式不变，仍可以在线段O_1O_2上求出H点。与刚才不同的是两个球相交得到的是垂直于O_1O_2平面上的圆，圆心为H，半径为MH。

点到直线的距离。从现在开始我们考虑距离问题。最简单的是点到点的距离，这已被大家所熟知。第一个并显然的例子是点到直线的距离。



考虑直线的参数形式，则PP_3应该和P_1P_2垂直，即$(P_3 - P) \cdot (P_2 - P_1) = 0$，把P代入直线方程$P = P_1 + u(P_2 - P_1)$，解得：

$$u = \frac{(x_3 - x_1)(x_2 - x_1) + (y_3 - y_1)(y_2 - y_1)}{\|P_2 - P_1\|^2}$$

注意在程序实现时需要保证P_1和P_2不重合，否则将发生除0错误。有了u，我们很容易得到P，进而求出距离。如果是线段，还应该检查u是否在0和1之间。这个方法很容易推广到高维情形。

线段交点。仍然用参数法，考虑直线P_1P_2和P_3P_4，则相对于两条直线的参数u_a和u_b分别为：

$$\begin{cases} u_a = \frac{(x_4-x_3)(y_1-y_3)-(y_4-y_3)(x_1-x_3)}{(y_4-y_3)(x_2-x_1)-(x_4-x_3)(y_2-y_1)} \\ u_b = \frac{(x_2-x_1)(y_1-y_3)-(y_2-y_1)(x_1-x_3)}{(y_4-y_3)(x_2-x_1)-(x_4-x_3)(y_2-y_1)} \end{cases}$$

两个表达式的分母相同，如果为0则两条直线平行；如果分子也为0，则两直线重合。

把任何一个参数代入直线方程即可求出交点。这样做有一个附加的好处是：不管关心的是直线还是射线还是线段都可以，只需要最后检查参数范围即可。

直线和圆的交点。仍然用参数式表示直线P_1P_2，中心在P_3的圆用标准方程，则联立可得一个复杂的二次方程：$au^2 + bu + c = 0$，其中

$$\begin{cases} a = (x_2 - x_1)^2 + (y_2 - y_1)^2 \\ b = 2((x_2 - x_1)(x_1 - x_3) + (y_2 - y_1)(y_1 - y_3)) \\ c = x_3^2 + y_3^2 + x_1^2 + y_1^2 - 2(x_3x_1 + y_3y_1) - r^2 \end{cases}$$

可以看出，直线和圆的交点仅仅是上述方程去掉含有z的所有项。

过三点的圆。利用点积可以得到三个距离等式，解出即可。假设三个顶点按逆时针排列。设$X_i = x_i - x_0, Y_i = y_i - y_0$，则三角形的有向面积

$$A = \frac{1}{2}\det\begin{bmatrix} X_1 & Y_1 \\ X_2 & Y_2 \end{bmatrix}$$

令$L_{ij} = \|V_i - V_j\|$，则圆心为：

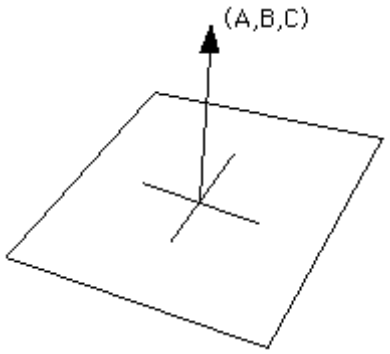
$$\begin{cases} x = x_0 + \frac{1}{4A}(Y_2L_{10}^2 - Y_1L_{20}^2) \\ y = y_0 + \frac{1}{4A}(X_1L_{20}^2 - X_2L_{10}^2) \end{cases}$$

有了圆心很容易算出半径，这里不再赘述。

8.1.3 三维图元及其算法

本节把上节的讨论扩展在三维空间中，讨论多面体、球等图元及其距离的计算。

平面(plane)。和直线一样，平面有多种表示法。第一种是一般式$Ax+By+Cz+D=0$。这个平面的法向量是(A, B, C)，如下图。注意$s=Ax+By+Cz+D$描述了点(x,y,z)在平面的“哪边”。s¿0则和法线(A, B, C)同侧，s¡0为异侧，s=0为平面上。



第二种表示法称为点法式，即给平面上的点P_1和平面的法向量N，则平面上的任意点P和P_1得到的向量都在平面上，因此和N垂直，即$\overrightarrow{N}\cdot(\overrightarrow{P}-\overrightarrow{P_1})=0$。这个方程本质上和一般式一样，但由于没涉及到坐标，有时候会显得很简洁。这个式子还可以进一步变形为$\overrightarrow{N}\cdot(\overrightarrow{P}-\overrightarrow{P_1})=\overrightarrow{N}\cdot\overrightarrow{P}-\overrightarrow{N}\cdot\overrightarrow{P_1}=\overrightarrow{N}\cdot\overrightarrow{P}-d=0\Leftrightarrow\overrightarrow{N}\cdot\overrightarrow{P}=d$。这个式子也经常使用。

三维空间中的方向。在二维平面中，方向可以用一个实数表示，即从x轴逆时针旋转的角度。在三维空间中一般用三个数表示，即方向余弦：

$$\cos\alpha=\frac{P_x}{\left\|\overrightarrow{P}\right\|},\cos\beta=\frac{P_y}{\left\|\overrightarrow{P}\right\|},\cos\gamma=\frac{P_z}{\left\|\overrightarrow{P}\right\|}$$

这三个角度为到每个轴的角度，满足$\cos^2\alpha+\cos^2\beta+\cos^2\gamma=1$。

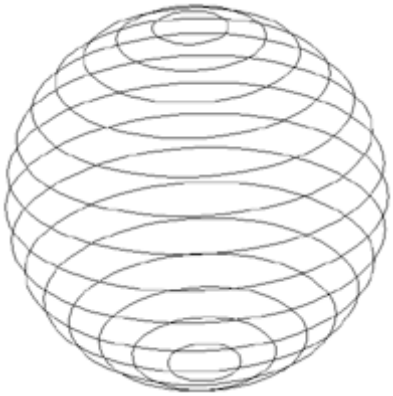
球(sphere)。球是一个重要的几何体。中心在(x_o,y_o,z_o)，半径为r的球方程为：

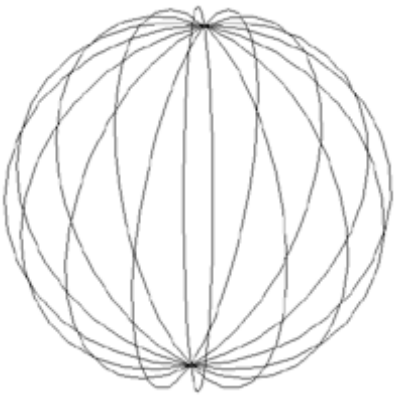
$$(x-x_o)^2+(y-y_o)^2+(z-z_o)^2=r^2$$

左边小于r^2时在球内。球还有一种参数表示法，即：

$$\begin{cases}x=x_o+r\cos\theta\cos\varphi\\ y=y_o+r\cos\theta\\ z=z_o+r\sin\theta\end{cases},0\le\theta<2\pi,-\pi/2\le\varphi\le\pi/2$$

如果把这个球看作地球的话，θ为常数时我们将得到一条条经线（如图(b)），而φ为常数时我们将得到纬线（如图(a)）。比如$\varphi = 0$为赤道。



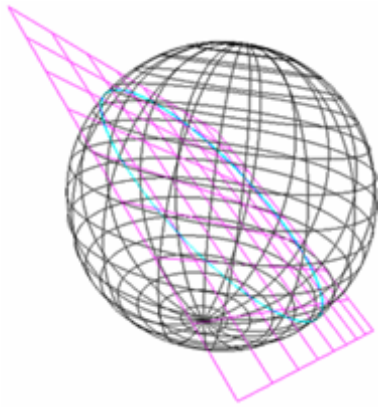


直线和球可能相离、相切或者相交。一个穿过球心的直线和球相交在两点，这两点称为**对踵点(antipodal point)**。平面和球可能相离、相切或相交。一个穿过球心的平面和球相交成一个圆，称为**大圆(great circle)**，如下图。球面上两点的最短距离是大圆的一段弧。

三个大圆相交后形成一个球面三角形。球面三角形的内角和大于180度，面积为

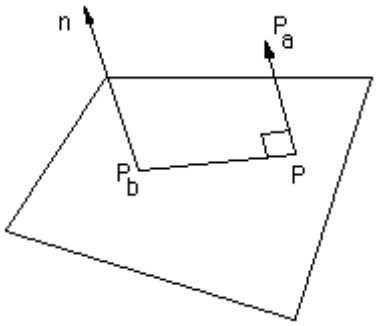
$$area = r^2(A + B + C - \pi)$$

这个公式可以简单的推广到四边形。还用三条边长a, b, c直接计算面积，即：



$$area = 4\arctan\sqrt{\tan\frac{m}{2}\tan\frac{m-a}{2}\tan\frac{m-b}{2}\tan\frac{m-c}{2}}, m = \frac{a+b+c}{2}$$

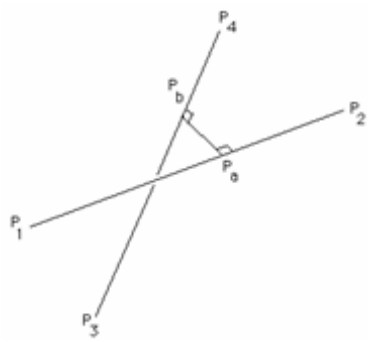
点和平面的距离。设点为$P_a = (x_a, y_a, z_a)$，平面的法向量$n = (A, B, C)$，以及平面上任意点$P_b = (x_b, y_b, z_b)$。只需要把线段$\mathrm{P}_a\mathrm{P}_b$投影到平面法向量n上，则投影距离为所求，即$dist = \frac{\overrightarrow{P_aP_b}\cdot\vec{n}}{\|\vec{n}\|}$，展开得$dist = \frac{A(x_a-x_b)+B(y_a-y_b)+C(z_a-z_b)}{\sqrt{A^2+B^2+C^2}}$。



注意到如果n是单位向量则除法可以省略，因此在不特别说明的情况下我们总是认为法向量是单位向量。如果有平面方程$Ax + By + Cz + D = 0$则结果更简单：$dist = \frac{Ax_a+By_a+Cz_a+D}{\sqrt{A^2+B^2+C^2}}$，当法向量为单位向量时答案简单等于把$\mathrm{P}_a$代入方程左边后的值。注意这里的dist是有向距离，这个符号有时是很有用的。

异面直线的最近线段。在二维平面中任两条不平行的直线相交，但在三维空间中更多情况是异面。在异面的情况下，有两种方法可以求出最近线段。一是根据定义，是连接两条直线的最短线段，即

即最小化$\|P_b - P_a\|^2$，代入两点的直线方程得$\|P_1 - P_3 + u_a(P_2 - P_1) - u_b(P_4 - P_3)\|^2$，可以把它展开成x, y, z的形式，并让$\mathrm{u}_a$和$\mathrm{u}_b$的偏导数等于0。另一个方法是利用最短直线



和两直线都垂直的特点，即

$$\begin{cases} P_1 - P_3 + u_a(P_2 - P_1) - u_b(P_4 - P_3)) \cdot (P_2 - P_1) = 0 \\ P_1 - P_3 + u_a(P_2 - P_1) - u_b(P_4 - P_3)) \cdot (P_4 - P_3) = 0 \end{cases}$$

耐心展开可得这样两个方程：

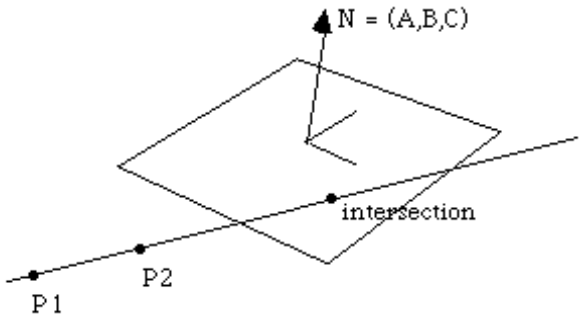
$$\begin{cases} d_{1321} + u_a d_{2121} - u_b d_{4321} = 0 \\ d_{1343} + u_a d_{4321} - u_b d_{4343} = 0 \end{cases}$$

其中$d_{mnop} = (x_m - x_n)(x_o - x_p) + (y_m - y_n)(y_o - y_p) + (z_m - z_n)(z_o - z_p)$。注意$d_{mnop} = d_{opmn}$。解这个方程组可得

$$\begin{cases} u_a = (d_{1343}d_{4321} - d_{1321}d_{4343})/(d_{2121}d_{4343} - d_{4321}d_{4321}) \\ u_b = (d_{1343} + u_a d_{4321})/d_{4343} \end{cases}$$

注意一共只有5个系数，所以可以直接计算出来而不用另写函数。

直线和平面的交点。直线我们一般都用参数式，但平面则一般式和点法式都用得比较多，因此这里给出两种公式。



过P_3且法线为N的平面的点法式为$\overrightarrow{N} \cdot (\overrightarrow{P} - \overrightarrow{P_3}) = 0$，过点$P_1(x_1,y_1,z_1)$和$P_2(x_2,y_2,z_2)$的直线方程为$P = P_1 + u(P_2 - P_1)$，联立解得$u = \frac{\overrightarrow{N} \cdot (P_3 - P_1)}{\overrightarrow{N} \cdot (P_2 - P_1)}$。注意分母为0代表直线和平面

法线平行，因此要么和平面平行（无解）要么在平面内（无穷多解）。

若平面用一般式$Ax+By+Cz+D=0$，联立$P=P_1+u(P_2-P_1)$解得

$$u=\frac{Ax_1+By_1+Cz_1+D}{A(x_1-x_2)+B(y_1-y_2)+C(z_1-z_2)}$$

分母为0的解释和刚才一样。

过三点的平面。用一般式$Ax+By+Cz+D=0$来表示过三个不共线点(x_1,y_1,z_1)，(x_2,y_2,z_2)和$(x_3$平面，则

$$A=\begin{vmatrix}1&y_1&z_1\\1&y_2&z_2\\1&y_3&z_3\end{vmatrix},B=\begin{vmatrix}x_1&1&z_1\\x_2&1&z_2\\x_3&1&z_3\end{vmatrix},C=\begin{vmatrix}x_1&y_1&1\\x_2&y_2&1\\x_3&y_3&1\end{vmatrix},D=-\begin{vmatrix}x_1&y_1&z_1\\x_2&y_2&z_3\\x_3&y_3&z_3\end{vmatrix}$$

展开可得：

$$\begin{cases}A=y_1(z_2-z_3)+y_2(z_3-z_1)+y_3(z_1-z_2)\\B=z_1(x_2-x_3)+z_2(x_3-x_1)+z_3(x_1-x_2)\\C=x_1(y_2-y_3)+x_2(y_3-y_1)+x_3(y_1-y_2)\\-D=x_1(y_2z_3-y_3z_2)+x_2(y_3z_1-y_1z_3)+x_3(y_1z_2-y_2z_1)\end{cases}$$

其中D也可以当计算出A，B，C后任意代入一个点求出。注意若三点共线则(A, B, C)=(0, 0, 0)。

两平面的交线。不平行的两个平面相交得到线。设两个平面为$\begin{cases}\overrightarrow{N_1}\cdot\overrightarrow{P}=d_1\\\overrightarrow{N_2}\cdot\overrightarrow{P}=d_2\end{cases}$，则交线可以写成参数形式$P=c_1\overrightarrow{N_1}+c_2\overrightarrow{N_2}+u(\overrightarrow{N_1}\times\overrightarrow{N_2})$。这是因为平面不平行时两个法向量不平行，因此它们所张成的平面和交线必有交点（交线同时垂直于两个法线，因此不可能和它们所张成的平面平行，故必有交点）。设此交点为$c_1\overrightarrow{N_1}+c_2\overrightarrow{N_2}$，而方向和和两法线都垂直因此可以写成$u(\overrightarrow{N_1}\times\overrightarrow{N_2})$。把P的表达式代入两平面得：

$$\begin{cases}\overrightarrow{N_1}\cdot\overrightarrow{P}=d_1=c_1\overrightarrow{N_1}\cdot\overrightarrow{N_1}+c_2\overrightarrow{N_1}\cdot\overrightarrow{N_2}\\\overrightarrow{N_2}\cdot\overrightarrow{P}=d_2=c_1\overrightarrow{N_1}\cdot\overrightarrow{N_2}+c_2\overrightarrow{N_2}\cdot\overrightarrow{N_2}\end{cases}$$

解得：

$$\begin{cases}c_1=(d_1\overrightarrow{N_2}\cdot\overrightarrow{N_2}-d_2\overrightarrow{N_1}\cdot\overrightarrow{N_2})/\Delta\\c_2=(d_2\overrightarrow{N_1}\cdot\overrightarrow{N_1}-d_2\overrightarrow{N_1}\cdot\overrightarrow{N_2})/\Delta\\\Delta=(\overrightarrow{N_1}\cdot\overrightarrow{N_1})(\overrightarrow{N_2}\cdot\overrightarrow{N_2})-(\overrightarrow{N_1}\cdot\overrightarrow{N_2})^2\end{cases}$$

注意事先应该判断二平面是否平行，即判断法向量是否平行，即$\overrightarrow{N_1} \times \overrightarrow{N_2} = 0$。

三平面的交点。在通常情况下三平面交于一点，除非有两者平行。可以下求出两平面的交点，但还有更优美的结论：

$$P = \frac{d_1(N_2 \times N_3) + d_2(N_3 \times N_1) + d_3(N_1 \times N_2)}{N_1 \cdot (N_2 \times N_3)}$$

分母为0时有两平面平行，因此无交点。

直线和三角形相交。直线和三角形相交是一个很基础的问题，因为多边形可以先三角剖分再判断。显然，我们可以先求出平面和直线的交点，再判断交点是否在直线上，最后判断交点是否在三角形中。判断点是否在三角形中有三种方法。

方法一：判断三个角度和是否为2π。

方法二：采取重心坐标系，即前面提到的三个参数的表示法。这个方法数学上很漂亮，可惜计算精度有问题。

方法三：投影法。把三角形投影到xy平面上，问题变成二维的。注意如果三角形是竖直的，投影xy平面上后三角形退化成线段。为了解决这个问题，我们规定法线绝对值最大的分量所对应的轴为投影平面的法向量。竖直平面法向量z分量为0，因此投影平面的发现要么为x轴要么为y轴（取决于这两个分量的相对大小），即投影在xz或yz平面，而不是xy平面。

投影到平面后，我们有更简单的判断方法：计算有向面积。计算出三个有向面积后，如果三个面积同号，则内部；两个为0，则和顶点重合；仅一个为0时在一条边上。

这个思想可以扩展到判断两个三角形是否相交，这个问题留给读者思考。需要说明的是在进行仔细计算前有两个快速实验判断可以先做：一是包围盒测试，二是判断一个三角形的三个顶点是否都在另一个的同侧。

直线和球的交点。仍然用参数式表示直线P_1P_2，中心在P_3的球用标准方程，则联立可得一个复杂的二次方程：$au^2 + bu + c = 0$，其中

$$\begin{cases} a = (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2 \\ b = 2((x_2 - x_1)(x_1 - x_3) + (y_2 - y_1)(y_1 - y_3) + (z_2 - z_1)(z_1 - z_3)) \\ c = x_3^2 + y_3^2 + z_3^2 + x_1^2 + y_1^2 + z_1^2 - 2(x_3x_1 + y_3y_1 + z_3z_1) - r^2 \end{cases}$$

可以看出，直线和圆的交点仅仅是上述方程去掉含有z的所有项。

过四点的球。利用点积可以得到四个距离等式，解出即可。假设四面体$\langle V_0, V_1, V_2, V_3\rangle$的次序与规范次序$\langle(0,0,0),(1,0,0),(0,1,0),(0,0,1)\rangle$是同构的。

定义$X_i = x_i - x_0, Y_i = y_i - y_0, Z_i = z_i - z_0$，则四面体的体积

$$V = \frac{1}{6}\det\begin{bmatrix} X_1 & Y_1 & Z_1 \\ X_2 & Y_2 & Z_2 \\ X_3 & Y_3 & Z_3 \end{bmatrix}$$

令$L_{ij} = \|V_i - V_j\|$，则球心为：

$$\begin{cases} x = x_0 + \frac{1}{12V}(+(Y_2Z_3 - Y_3Z_2)L_{10}^2 - (Y_1Z_3 - Y_3Z_1)L_{20}^2 + (Y_1Z_2 - Y_2Z_1)L_{30}^2) \\ y = y_0 + \frac{1}{12V}(-(X_2Z_3 - X_3Z_2)L_{10}^2 + (X_1Z_3 - X_3Z_1)L_{20}^2 - (X_1Z_2 - X_2Z_1)L_{30}^2) \\ z = z_0 + \frac{1}{12V}(+(X_2Y_3 - X_3Y_2)L_{10}^2 - (X_1Y_3 - X_3Y_1)L_{20}^2 + (X_1Y_2 - X_2Y_1)L_{30}^2) \end{cases}$$

有了圆心很容易算出半径。

8.1.4 小结与应用举例

在本节中，我们介绍了计算机图形学中的数学基础。第一小节介绍了向量空间、仿射空间的基本概念，读者应该学会向量加法、数乘、点积、叉积和张量积，仿射变换的矩阵表示法并能把平移、旋转、缩放、反射、剪切和平行投影转换成程序。读者应能体会仿射变换的矩阵表示法的好处：变换可以用矩阵乘法进行组合。仿射变换中也许最有意思的一种是旋转：它有三种常用的表示法：矩阵表示法、轴-角表示法和四元数表示法。由于矩阵法难以插值，轴-角法难以组合，四元数的应用越来越广泛[1]。由于本小节的目的只是为读者提供一个计算方法，并没有过分关注时空开销和一些特定的应用场合，因此不再介绍四元数。

第二小节介绍了二维几何图元及其基本算法。最重要的是直线，它有一般式和参数式两种表示法，但我们几乎只用参数式。参数式的好处是可以很容易的把直线和射线、线段统一起来，且参数具有明显的几何意义。另一个好处时我们往往是根据两点确定一条直线，而并不是直接根据倾斜角。在给定点为整点的时候这个好处尤其明显。最简单的多边形是三角形，读者应记住它的有向面积公式，特别是方向。这个公式很容易根据叉积的几何意义得到，并可以用来推导出本节给出的任意平面多边形的面积公式。对二次曲线的一般讨论比较复杂，读者不必记住这些公式但需要了解推导过程。其中变换的第一步：矩阵的特征值分解并没有包含在本节中，有兴趣的读者可

[1] 但四元数变换向量的时间比较长，即使用特殊的四元数乘法

以阅读相关书籍。需要指出的是，矩阵分解是数值计算中的重要问题。本节接下来讨论了圆及其交点，读者应该熟练掌握文中的例子和圆相交算法。点到直线的距离、线段交点、直线和圆的交点和三角形外接圆都是基本问题，读者不必死记这些结论，但应熟悉推导过程。

第三小节把问题转向三维，首先介绍了平面、方向的表示和球。平面的一般式、点法式都有广泛应用，读者都应熟悉；球的一般方程和参数方程都很重要，尤其是参数方程，读者应了解各参数的几何含义。大圆在球面几何中起到了重要作用。相对于二维，三维中的基本计算问题变得更多和更复杂。点面距离、直线和平面的交点以及二平面交线、过三点的平面都是最基础的问题。本节还介绍了另外一些问题，读者可以根据需要阅读。需要特别指出的是：考虑和平面相关的问题时往往需要同时从一般式和点法式出发讨论。一般式适合代数推导，而点法式适合几何推导（尤其是和角度相关的）。读者应重视三维空间中的法线。

一个不错的参考资料在：http://escience.anu.edu.au/lecture/cg/Maths/index.en.html

该网站介绍了向量空间、仿射空间的基本知识、点积和叉积以及圆的相交等，主要是数学但通俗易懂。

另一个有用的网址是：http://astronomy.swin.edu.au/~pbourke/geometry/，它包含一些基本计算问题的材料。

重要算法和程序列表：

类别	程序	备注
向量运算	vector.cpp	包括二维和三维点积、叉积、张量积及其应用
二维仿射变换	affine2d.cpp	平面上的仿射变换，包括平移、旋转、缩放、反射、剪切和点在直线上的投影
三维仿射变换	affine3d.cpp	空间中的仿射变换，包括包括平移、旋转、缩放、反射、剪切和平行投影
二维图元算法	geom2d.cpp	包括直线和线段表示，圆和圆弧的表示，点到直线距离，圆和直线交点，圆和圆的位置关系，三角形的平面和外接圆
三维图元算法	geom3d.cpp	平面的点法式和一般式，方向的欧拉角表示，球的参数表示和球面距离、球面三角形面积，点到平面的距离，异面直线的最近线段，直线和平面的交点，过三点的平面，两平面的交线，三平面的交点，直线和三角形的交点，直线和球的交点，球和球的位置关系，四面体的外接球。

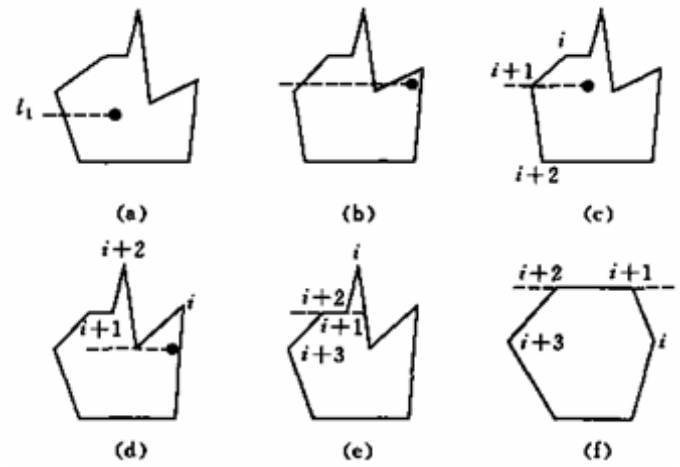
8.2 基本几何问题

本节讨论各种常见的几何问题和算法，如定位、线段相交、多边形和多面体的经典问题和算法。

8.2.1 定位问题

本节讨论查找问题，它主要分为两部分，一是点定位，二是范围查找。点定位问题有两种形式，一是判断它是否在一个几何体内，二是判断它在哪个区域内。其中第二个问题我们将留在平面剖分中讨论。

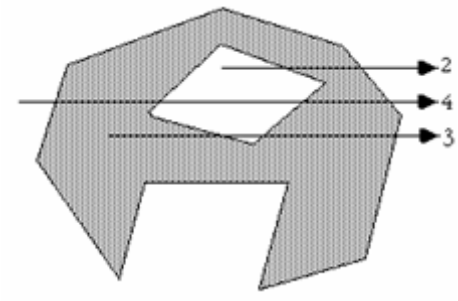
点在多边形内。一般有两种方法：射线法和转角法，如下图。

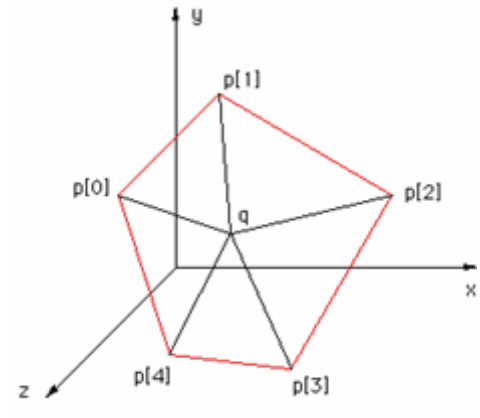


在相交次数为奇数时在内部，偶数时在外部。需要注意端点：如果把每条边看作是闭区间，那么每个端点恰好属于一条边。如果交点在端点处时将被重复计算！正确的方法是把边看作左闭右开区间。另一个问题是和某边重合。这个方法在遇到(d)时出了问题：这个交点应不应该算？对于(e)和(f)这种和边重合的情况又应如何算？

区分(c)和(d)的方法是当判断到端点相交时判断相邻两个点p_i和p_{i+2}在射线的同侧还是异侧。如果判断到p_{i+2}仍在线段上，应进一步判断p_i和p_{i+3}在射线的哪一侧。注意这个算法对有洞多边形仍然适用，如下图。

一个很有意思的情形是在三维空间中的多边形。在这种情况下最容易想到的方法是先判断点是否在平面内，然后利用坐标变换把该平面变成xy平面再用平面上的算法。但实际上有一个更方便的算法可以同时完成两步判断：判断内角和是否为2π，如下图：

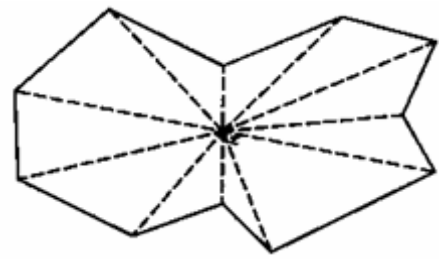




注意尽量先单独判断点是否和多边形顶点重合，或在一条边上。

点在星形多边形内。星形多边形内的情形比较简单：它可以通过二分查找完成，首先随便在形内找一个点p，连接每个顶点得到若干楔形，然后根据角度区间进行二分查找，注意区间$[t_1, t_2]$应是闭的，按照逆时针顺序，初始时$t_1=t_2$代表所有角度。

凸多边形是星形多边形的特殊情况。中心点可以简单的取顶点坐标平均数；但一般星形多边形不能取坐标平均数（它可能在形外！），而需要下求出多边形的核（它是一个凸多边形），然后取核的顶点坐标平均数。

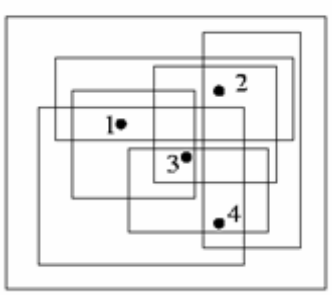


范围查找问题(range search)。范围查找问题需要给定一个点集S和范围D，考查有哪些点在D中。它的两个基本形式是：

报告问题：列举出范围内所有点构成的S的子集S'

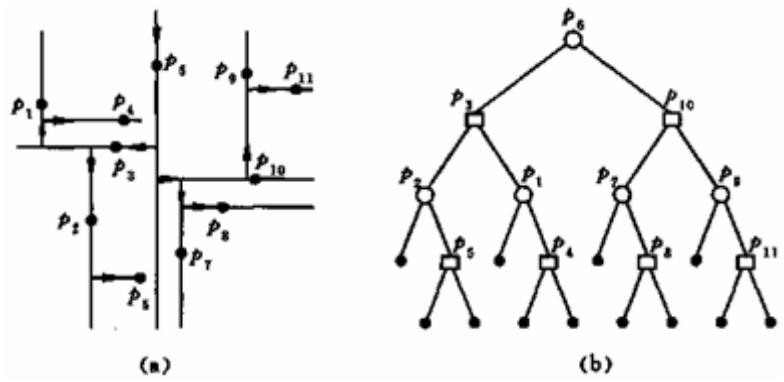
计数问题：只就算出S'的基数，即在范围内点的个数

正交查询(orthogonal query)。正交查询是最简单的范围查询，它的查询范围D总是正交区域。



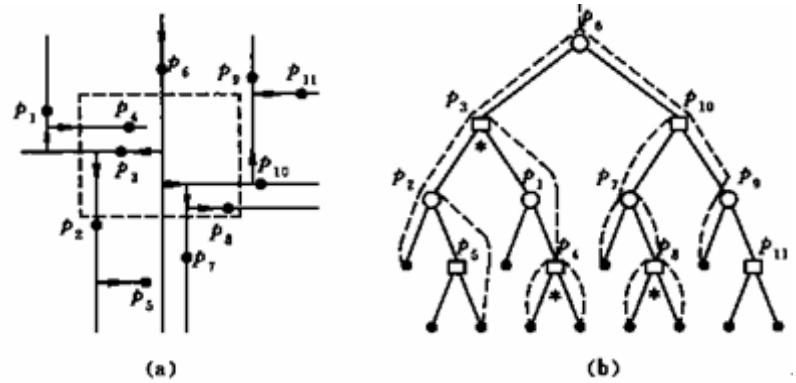
R = { { },
{1}, {2}, {3}, {4},
{1,2}, {1,3}, {2,3}, {2,4}, {3,4},
{1,2,3}, {1,3,4}, {2,3,4} ,
{1,2,3,4} }

k-D树方法。多维二叉树也称为k-D树(k-D tree)，它是二分思想在范围查找问题中的直接应用。它交替使用横线和竖线将平面分成两个矩形。这些直线过S中的点，并且使得直线两侧S内:œ8近似相等。注意这里不是按面积平分，而是按点数平分，因为规定直线必须经过S内的点。



用k-D树进行范围查找时，每次需要测试当前矩形区域和当前结点v对应的矩形区域R(v)的交。记R_1和R_2分别是v的两个儿子对应的区域，则如果它完全包含在R_1或R_2中，只需继续查找该儿子。如果和R_1和R_2均相交，则v有可能属于D，应测试，然后两边同时递归查找。作为特殊情况，如果D完全包含R_1和R_2，可以直接列举出R_1或R_2中的所有元素，不用继续查找。

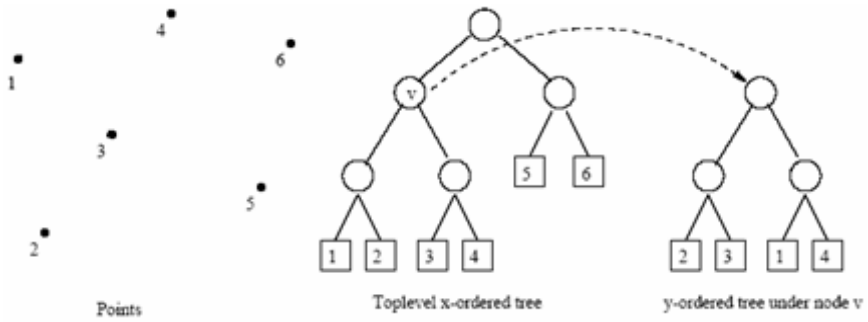
下面是执行过程的例子，*表示成功测试，而虚线表示访问路径。



每次求中值和分割点集需要O(n)时间，因此建树的时间复杂度T(n)满足T(n)=2T(n/2)+O(n)，解得T(n)=O(nlogn)。查询时间和访问结点树成正比。若p(v)在范围中，那么在结点v中|ˆΩ此时间（称v为生成结点），否则浪费了时间（称v为非生成结点）。可以证明：非生成结点数最坏为O($\sqrt{n}$)的，因此查询时间为O($\sqrt{n}+k$)。

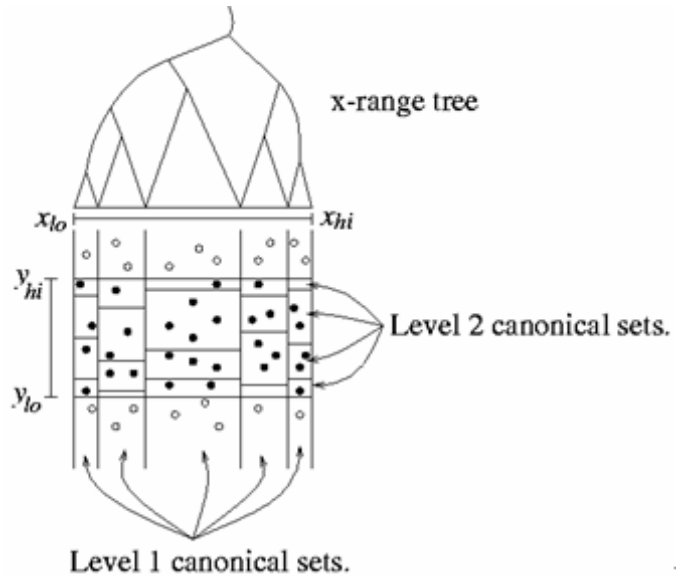
推广到n维，k-D树方法耗费O(dnlogn)预处理时间和O(dn)存储，询问时间为O($dn^{1-1/d}$+k)，可以完成一个d维n个点集合的范围查找。

范围树(range tree)。范围树方法和前面提到的线段树非常类似。对于二维情形，首先根据x坐标构造一棵线段树，然后每个线段（x区间）中把该区间中所有点组织成一棵线段树。



我们知道，一维线段树包含2n个结点，而每个结点对应的x区间内的点恰好是该结点子树的所有叶子，因此每个点恰好被logn个结点的y范围树所包含，总空间复杂度为O(nlogn)。从**标准集(canonical set)**的角度讲还可以把范围树画成下图。

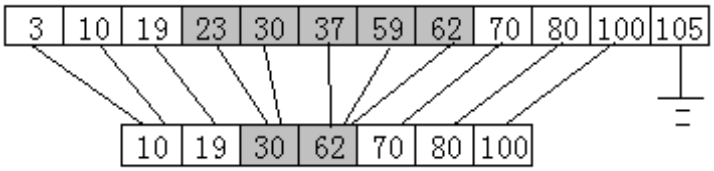
建二维范围树的时间是O(nlogn)。如果按照最简单的思路，每一步都进行排序的话时间复杂度为O(nlog^{2}n)，并不是最优的。必须反过来按照从下到上的方式进行合并，或者说先构造子树，然后用线性时间把两个子结点中的数组合并，则时间复杂度降



为O(nlogn)。

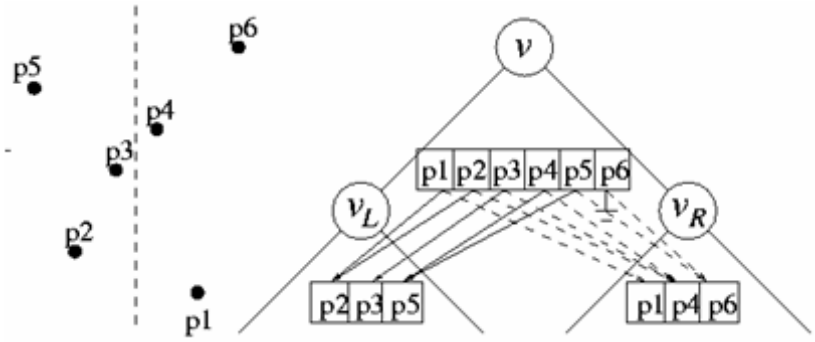
查询时，首先把x坐标范围分解成logn个标准集，然后在每个标准集中执行一次一维查找。每个一维查找时间为O(n)，因此总的为$O(\log^2 n)$。另外，由于一维查找的特殊性，我们可以简单的把第一层范围树的每个结点表示成数组而不是一棵一维范围树。有一个称为**分块层叠(fractional cascading)**的技术可以把查询时间降到O(logn)。

这个技术的突破口是：在x范围的每个标准集里，查找的y范围是完全一样的！分块层叠技术把这些标准集合并在一起形成一个大表，只进行一次查找，然后利用查找得到的信息对于每个区域只用O(1)时间便能得到想要的结果。



这里举一个两个序列的例子。上图中A_2是A_1的子集，两个序列都是有序的。我们事先给A_1的每个元素x设置一个指向A_2的指针，表示不超过x的最大值。这样，如果搜索范围是[20, 65]，则找到A_1的范围如阴影部分所示。通过指针很容易得到A_2中的范围，而不需要重新进行二分查找。

上图是修改后的二维范围树，树中每个结点都是按y轴排序好的，每个元素都有两个指针，一个对应左儿子，一个对应右儿子。这样在O(1)时间内可以往下走一层，因



此总时间复杂度为O(logn+k)。扩展到d维，只有最后一层可以用这个技巧，因此询问时间为O($\log^{d-1}$n+k)。

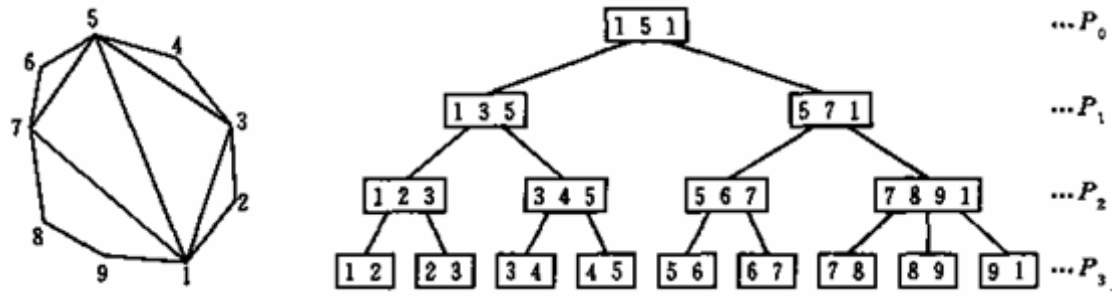
8.2.2 多边形及其剖分

多边形是一个很有意思的话题，它包含很多内容，这里只列举一些最有代表性的。

凸多边形(convex polygon)。什么是凸？凸就是任何两点的连线在形内；直观的说，如果从最小方的点沿着极角最小的边走，则每次都向**左转(left turn)**。这样，我们得到了凸多边形的判定算法：先求出第一条，然后判定是否每条边相对于上一条边都是左转。如何判断左转呢？可以借助于三角形的有向面积。

```
Bool left(tPointi a, tPointi b, tPointi c)
{
Return Area2(a, b, c) ¿ 0;
}
Bool LeftOn(tPointi a, tPointi b, tPointi c)
{
Return Area2(a, b, c) ¿= 0;
}
Bool Collinear(tPointi a, tPointi b, tPointi c)
{
Return Area2(a, b, c) == 0;
}
```

凸多边形有一种所谓的均衡分层表示，每个结点表示一个顶点编号连续的多边形和该序列的钟点，用以把多边形尽量均衡的分成两部分。

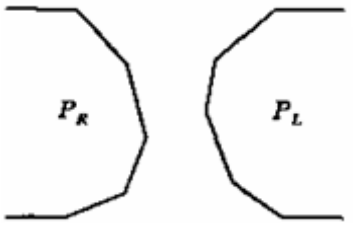


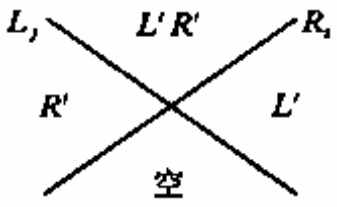
显然这样的显示表示不仅允许我们对凸多边形进行二分查找，还允许进行修改。显然可以在O(n)时间内建立这样的分层表示，且深度为logn。

问题一：给定凸多边形P的均衡分层表示和内点x，可以在O(logn)时间内判断任意点是否在P内。

问题二：给出凸多边形P的均衡分层表示和内点x，可以在O(logn)时间内判断任意直线l是否和P相交并求出交。仍然是二分查找，不过变成判断每个结点（对应一个三角形）是否和l相交。如果不交，l在该三角形的哪侧，以便走到正确的儿子。

问题三：给出两个凸多边形P和Q的均衡分层表示，可以在O(log(n+m))时间内判断出它们是否相交。





如上图，只需把每个多边形分割成两个单调链P_R和P_L（顶部和底部是水平射线），如果P的右链和Q的左链交或者P的左链和Q的右链，则P和Q相交。具体方法是取P_R的“中间线段”R_i（其中i=(1+m)/2）和P_L的“中间线段”L_j（其中j=(1+n)/2）。假设R_i和L_j相交（如果不是，则给i或j加1或减1，而相邻线段不共线，调整后二者一定相交）

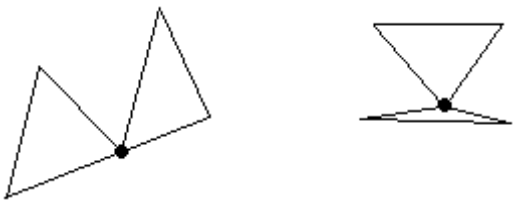
上图给出了一种可能的情况。对于直线R_i，链P_R要么完全在直线上方要么完全在下方。

简单多边形(simple polygon)。简单多边形的情况比凸边多边形复杂得多。我们还是先进行简单多边形判定。判定的方法是检测边相交情况，但很容易设计出错误的判定算法。

错误法则一：如果两条不同的边规范相交，则不是简单多边形。反例如图(a)

错误法则二：如果两条不同边规范相交或不规范相交，则不是简单多边形。这个显然会把所有简单多边形判断为非简单多边形，因为相邻边总是非规范相交的

错误法则三：如果两条没有公共端点的不同边相交，则不是简单多边形。反例如图(b)



因此正确的说法是：如果两条非相邻边规范相交或非规范相交，则不是简单多边形。

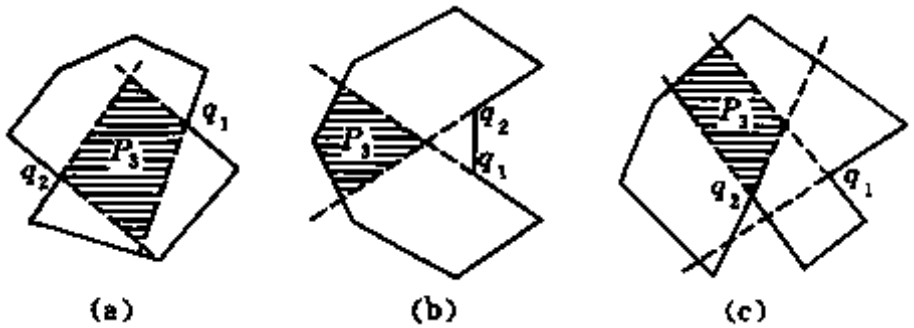
如何判定规范相交呢？注意我们需要用xor函数而不是面积乘积，以避免乘法溢出。代码如下：

```
bool IntersectProp(tPointi a, tPointi b, tPointi c, tPointi d)
{
// Eliminate improper cases
If(Colinear(a, b, c) || Colinear(a, b, d) || Colinear (c,d,a) || Colinear(c, d, b)) return
false;
Return xor(Left(a, b, c), Left(a, b, d)) && xor(Left(c, d, a), Left(c, d, b));
}
bool xor(bool x, bool y)
{
return !x  !y; // negated to ensure that they are 0/1 values
}
```

非规范相交类似，需要写一个辅助函数between。

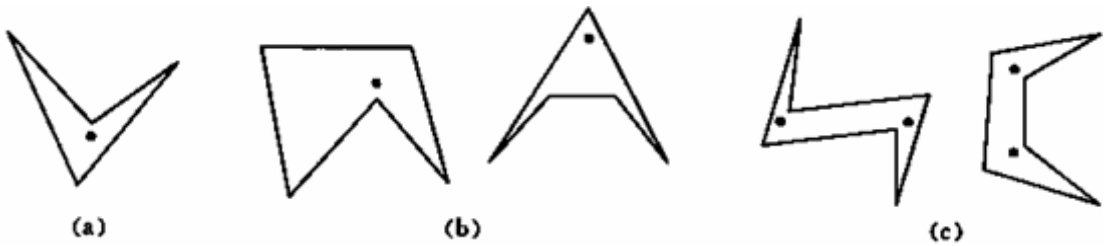
一个重要的问题是求简单多边形的**核(kernel)**，即能“看到”多边形每个顶点的点集。显然凸多边形的核是自身，而凹点将导致核缩小。下图是凹点数为2的几种情况。

```
Bool Intersect(tPointi a, tPointi b, tPointi c, tPointi d)
{
If(IntersectProp(a, b, c, d)) return true;
Else if(Between(a, b, c) || between(a, b, d) || between(c, d, a) || between(c, d, b)
Return true;
Return false;
}
Bool between(tPointi a, tPointi b, tPointi c)
{
If(!Colinear(a, b, c)) return false;
If(a[x] != b[x]) return (a[x]¡=c[x]&&c[x]¡=b[x]) || (a[x]¿=c[x]&&c[x]¿=b[x]));
Else return (a[y]¡=c[y]&&c[y]¡=b[y]) || (a[y]¿=c[y]&&c[y]¿=b[y]));
}
```

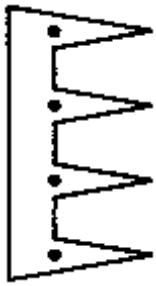


求核的算法很多，其中比较易于理解的方法是半平面交。从左下方点开始沿着极角最小的边走，则多边形的内部始终在左侧。这样把所有边所对应的半平面相交，得到的就是核。半平面的交可以在O(nlogn)时间内求出，因此求核的时间复杂度为O(nlogn)。

和“看到”关系有关的另一个有趣问题是画廊问题：最少用一个点可以“看到”整个多边形。容易看出，看到所有点即可看到整个多边形，但看到所有边不能保证（想一想，为什么）。显然凸多边形随便放一个点就行，有凹点的情况只要核非空也只需要一个；但如果核为空，也许需要两个、三个或者更多。



一个自然的问题是：在所有n边形中，怎么样的多边形需要最多的点？设这个最坏的多边形至少需要F(n)个点。

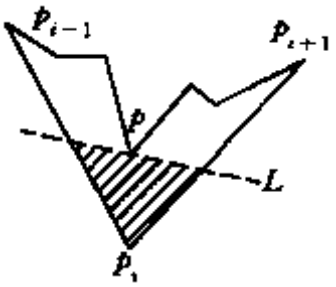


由上图，很容易构造出需要n/3个点的例子，又可以构造出n/3个点覆盖任意n边形的方案，因此我们有：$F(n) = \lfloor \frac{n}{3} \rfloor$，例如F(3)=F(4)=F(5)=1，F(6)=2。

三角剖分(triangulation)。多边形的三角剖分是一个很重要的问题，它是很多复杂算法的第一步。多边形的**对角线(diagonal)**被定义为连接两个端点，不和其他边相交且处于多边形内部的线段。多边形并不是“显然存在对角线”的，对角线的存在需要证明。

定理1：任意n(n¿=4)多边形都有对角线

首先可以证明多边形必有一个凸顶点，因为最下方的点就是凸的。设p_i是凸的，考虑p_{i-1}和p_{i+1}组成的线段。如果它是对角线，则定理成立，否则它要么完全在多边形外要么和P的边界相交。在两种情况下三角形$p_i p_{i+1} p_{i-1}$一定包含另外一个顶点，设p为其中离p_i最近的一个，则阴影部分内一定没有其他点，因此pp_i不可能和其他边相交。因此p是一条对角线。

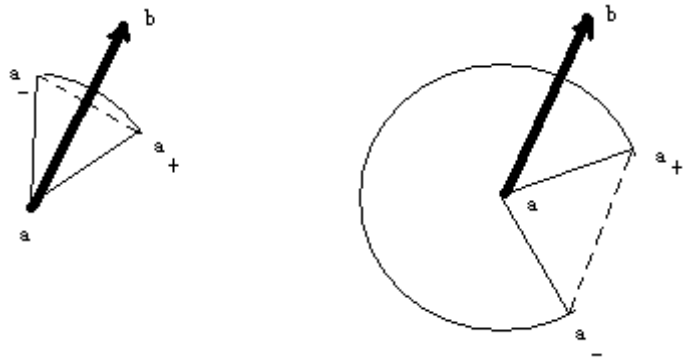


下面的程序在$O(n)$时间内判断一条边ab是否为“对角线候选”，即不和与a和b邻接的边相交。

然而这一点是不够的，因为它可能完全位于多边形外部，我们需要排除这一

```
Bool Diagonalie(tVertex a, tVertex b)
{
tVertex c, c1;
// for each edge (c,c1) of P
c = vertices;
do{
c1 = c -¿ next;
if(c!=a && c1!=a && c!=b && c1!=b && Intersect(a-¿v,b-¿v,c-¿v,c1-¿v))
return false;
c = c -¿ next;
}while(c != vertices);
Return true;
}
```

情况。考虑a为**凸(convex)**点和**凹(reflex)**点的情况。总之和和由a_-,a, a_+组成的**圆锥(cone)**有关系。a为凸时必须在里面，a为凹时必须在外面。



下面的代码处理了两种情况。注意凸性的判定是通过Left函数完成的。

虽然InCone函数很简单，但是容易写错。读者可以体会为什么凸时（注意共线也算凸）用Left而凹时用LeftOn。其实Diagonal函数中两个InCone只需要保留一个即可，而InCone函数的另一种实现法是（由Andy Mirzian提出）InCone为真当且仅当下面三个值中最多一个为假：Left(a, a_+, b), Left(a, b, a_-), Left(a, a_-, a_+)。

定理2：任意n(n¿=4)边形都有三角剖分。

不断的加对角线，可以把多边形分割为不相交（没有公共内点）三角形的并。用这个方法可以顺便证明n边形的三角剖分包含n-3条对角线，n-2个三角形，内角和为$(n-2)\pi$。

用这个定理可以得到一个简单的三角剖分算法：枚举对角线，再递归分割。最坏

```
Bool InCone(tVertex a, tVertex b)
{
tVertex a0, a1;
a1 = a-¿next;
a0 = a-¿prev;
// convex
If(LeftOn(a-¿v,a1-¿v,a0-¿v))
Return Left(a-¿v,b-¿v,a0-¿v)&&Left(b-¿v,a-¿v,a1-¿v);
Else
Return LeftOn(a-¿v,b-¿v,a1-¿v)&&LeftOn(b-¿v,a-¿v,a0-¿v);
}
Bool Diagonal(tVertex a, tVertex b)
{
Return InCone(a, b) && InCone(b, a) && Diagonalie(a, b);
}
```

情况每次需要枚举n^2条连线，检查它们每个是否合法对角线需要O(n)时间，因此每找一条对角线的时间复杂度为$O(n^3)$，找n-3条对角线需要$O(n^4)$的时间。

可以用直观的方式定义三角剖分的对偶。无孔多边形的三角剖分对偶应该是一棵树，且每个点的度数最多为3。如果把一个度数为1或2的点看成根，则这棵树是二叉树。

如果连续三个顶点$p_{i-1}p_ip_{i+1}$中$p_{i-1}p_{i+1}$是一条对角线，则这三个顶点构成一个**耳(ear)**，其中p_i是耳尖。

定理3：任意n(n¿=4)边形至少有两个不重叠的耳。

考虑它的一个三角剖分树T，则T至少有两个叶子。每个叶子对应于一个耳，定理得证。

这样，只需要枚举n个顶点，花O(n)时间可以判断每个点是否为耳尖，花$O(n^2)$时间可以找到一条对角线，总时间$O(n^3)$。

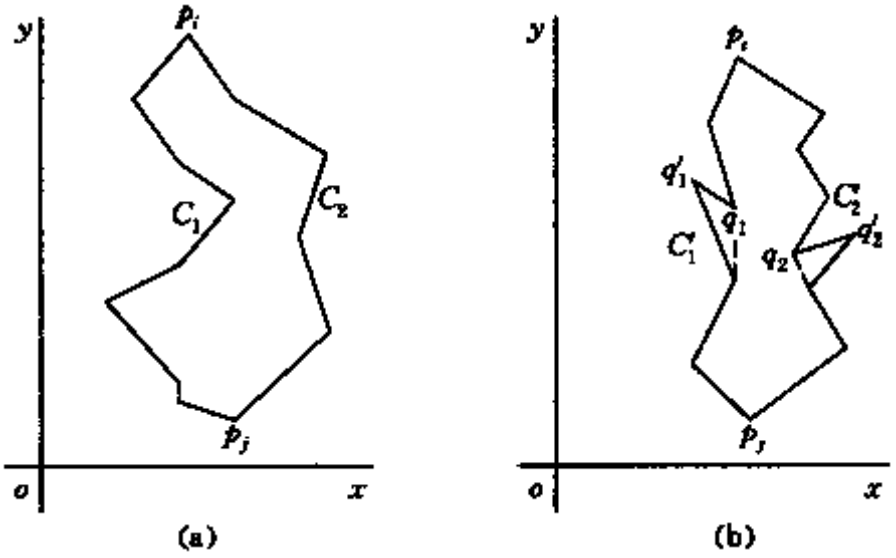
同样是用定理3，动态维护耳信息可以把时间复杂度降为O(n^2)。具体方法是第一次计算出每个顶点是否耳尖，花费O(n^2)时间，然后每割一个耳更新旁边两个点的耳尖信息，则每找一条对角线只花费O(n)时间，总时间O(n^2)。

基于梯形剖分的三角剖分快速算法。任意简单多边形的三角剖分问题可以在O(n)时间内解决，但算法比较复杂。这里介绍一种比较实用且启发性强的O(nlogn)时间算法。

首先介绍单调多边形链。我们总是找到多边形的最上方和最下方的点，然后把它分解为两条链，并规定y轴为参考线。如下图，(a)中多边形的两条链都关于y轴单调，但(b)中多边形的两条链都不单调。对于相邻三个点p_{i-1}, p_i, p_{i+1}，如果p_{i-1}和p_{i+1}的坐标都大于（小于）等于p_i的y坐标，且p_i本身是严格凹的，则则p_i为P的**歧点(interior cusp)**。(b)中的q'$_1$，q'$_2$都是歧点。

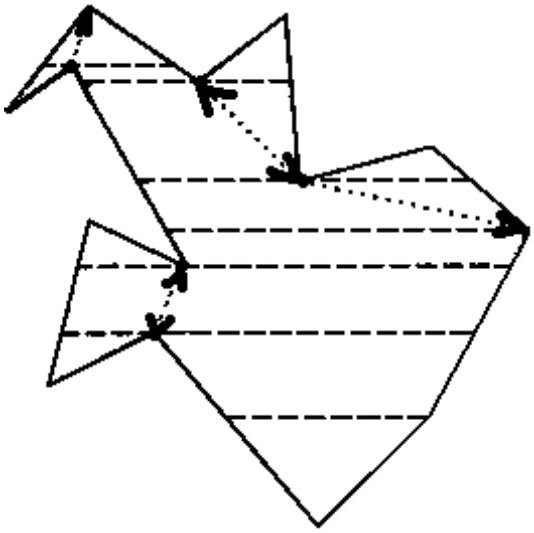
定理：无歧点的链是单调链。

这个定理很直观，但并不容易证明，且不能把单调链改为严格单调链，因为歧点要求两个相邻点的y坐标必须不同（否则三点共线，不能算凹），因此C'$_1$和C'$_2$不是单调链。

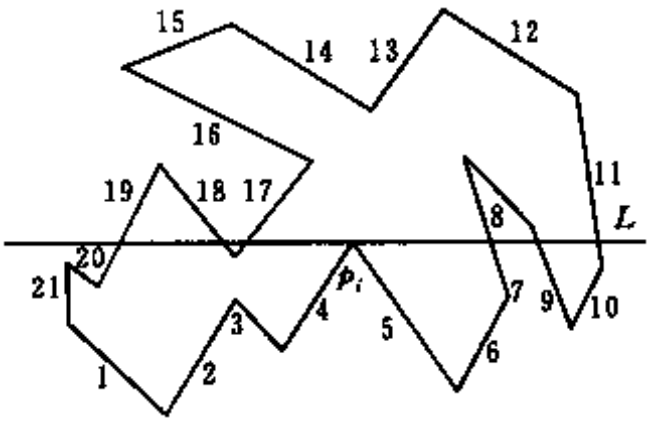


过每个顶点处往左往右尽量延伸一条水平线，直到和边界相交，得到的划分成为**水平梯形剖分(horizontal trapezoidalization)**。每个点称为它所画出先的**支撑点(supporting vertex)**。如果多边形P中任意两个点的y坐标都不相同，则每个梯形恰好有两个支撑点：一个在顶边，一个在底边。一个有趣的性质是：如果一个支撑点在

一个梯形顶边或底边的中间，则它是一个歧点。如果把每个歧点和与它在同一个梯形中的“相对支撑点”(即下歧点往下连，上歧点往上连)，则所有歧点消失，多边形变为两条单调链。



梯形剖分本身应用平面扫描技术，每个不同的y坐标都是一个事件，例如下图中维护的当前边列表为(19, 18, 17, 7, 9, 11)。利用平衡二叉树可以在O(nlogn)时间内得到梯形剖分。



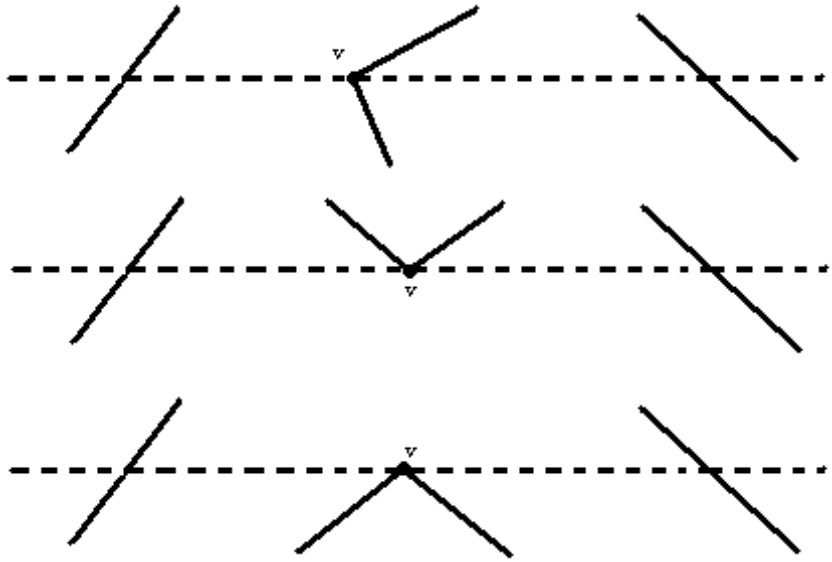
具体来说，设v是事件点，且在扫描线L上落于边a和b之间，且v是边c和d的公共点，则：

情况一：c在L上方，d在下方。用d来替换c。

情况二：c和d都在L上方。删除c和d。

情况三：c和d都在L上方。在a和bm插入c和d。

如下图。



最后一个问题是：如何剖分单调多边形？从上到下扫描所有点，并将每个点v连接到相对链上，在v上方的可见点上。显然不能进行可见性测试，所以必须指定一些简单的规则。

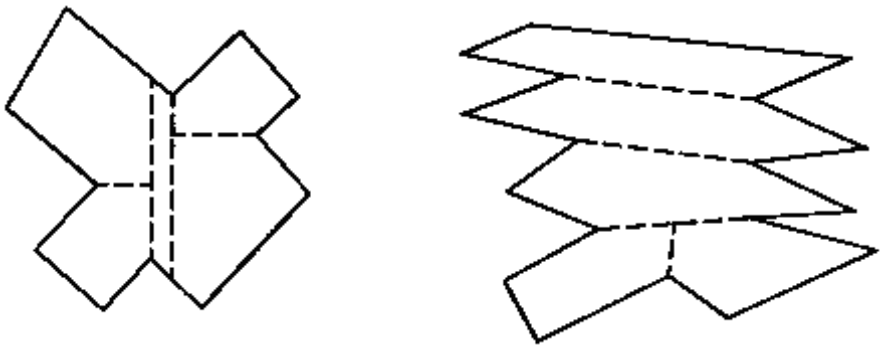
下图(a)是一个凹链，它对链上的点可以直接连接凹链上的各个顶点，完成三角剖分。对凸点p_{i+1}的出现，可以让p_i连接p_{i+2}，p_{i+3}，...，直到右转为止。在图(b)中，从p_ip_{i+3}到p_ip_{i+4}是右转，此时应停止，因为p_ip_4肯定不是对角线（容易看出它和边界相交）

这个算法可以用一个栈来实现：首先规定最顶部两个点所在的链为凹链并把它们放到栈中（栈底是最高点），然后按y坐标从大到小（从上到下）扫描。如果遇到另一条链上点，直接把它和栈里所有点相连并让所有元素出栈；如果遇到凹链上的点p_i，则

检查它的上一个点p_{i+1}的凹凸性。如果是凹点，则让p_i入栈，否则连接p_ip_{i+2}并让p_{i+1}出栈。这相当于是割除了一只耳。重复这一过程直到栈顶为凹点为止，让p_i入栈。显然这个算法在预处理（按y坐标排序）后是线性的。

多边形的凸划分。划分成三角形时，三角形的数目太多，有时候只需要分解成凸多边形即可。凸划分出的块数会少得多，因为我们有：

定理：设M是划分多边形成凸多边形的最少数目，对于有r个凹点的多边形，有$\lceil \frac{r}{2} \rceil + 1 \leq M \leq r + 1$。



证明：对多边形的每个凹点画一条角平分线，可以得到凸划分，数目为r+1，如图(a)；另外一条对角线至多可以分解两个凹点，因此至少$\lceil \frac{r}{2} \rceil + 1$个凸多边形。

Hertel-Mehlhorn算法。如果只能用对角线进行划分，一个简单的思路是从三角剖分开始不断删除“不必要”的对角线，即删除后不会产生凹点的对角线。显然算法可以在O(n)时间内完成。可以证明H-M算法得到的凸多边形数目不超过4M。

在只能使用对角线的情况下，是有保证最优解（数目最少）的多项式算法的。Greene于1983年设计出$O(n^4)$的算法，并由Keil于1985年改进到$O(n^3 \log n)$。

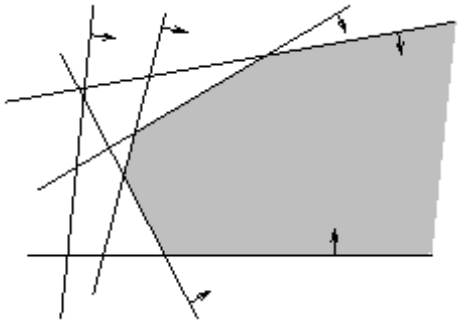
如果可以用其他线段进行划分，则问题变得更加困难，但仍有多项式算法求最优解，如1980年Chazelle设计出的$O(n^3)$算法。

8.2.3 半平面交和低维线性规划

一个半平面是一个线性不等式的解所构成的区域。

$$ax + by \leq c$$

这个半平面用系数(a, b, c)表示。显然n个半平面的交是凸的，但可能是空的甚至可能是无界的。



显然在半平面交中，每条边最多只出现一次，因此组合复杂度是O(n)的。容易证明这个问题的下界也是$\Omega(n\log n)$的。

分治法可以达到这个下限，方法是：

划分：把n个半平面划分成两半，分别有$\lfloor n/2\rfloor$和$\lceil n/2\rceil$个半平面。

求解：求出两个半平面集各自的交，得到两个可能无界的凸多边形C_1和C_2。

合并：求C_1和C_2的交C。

求凸多边形交最容易想到的方法就是利用线段相交的扫描算法。

线段相交问题。给出平面上的n条线段，报告出所有交点。显然$O(n^2)$时间内可以解决该问题，但我们可以做得更好。设有k个交点，则显然至少需要$O(k)$时间。事实上，在第二章介绍的代数决策树模型下，线段相交问题的时间下界为$\Omega(n\log n+k)$，因为可以用它解决元素唯一性问题。假定读者已经熟悉上节介绍的线段相交的判断方法，这里只介绍算法本身。

这个算法是基于扫描法的，这里给出扫描法的三个基本要素：

扫描线(sweep line)：用一条竖直线从左扫描到右，记录扫描线和线段的交点，从上到下储存起来。

事件(event)。虽然我们可以认为扫描线是连续移动的，但实际上我们只需要考虑它经过一些特殊位置的时刻，称为事件。事件一般有两类，静态事件是事先知道的，而动态时间是扫描过程中动态增加或删除的。在线段相交问题中，静态事件是线段端点，而动态事件是线段交点。这些交点是事先不知道的（知道了就不用算了），只能在扫描过程中动态生成和处理。在线段相交问题中，交点事件一旦产生就不会被删除，但在后面我们将看到更复杂的例子，动态事件有可能需要被取消。动态事件最重要的一点是：事件必须在发生之前被检测到。因为我们按从左到右的顺序处理事件，

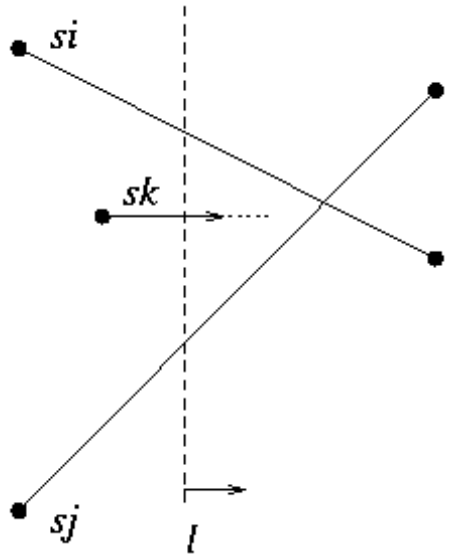
如果事件发生以后才检测到，就没有办法处理了。

事件更新(event update)。在遇到事件时，我们一般需要更新数据结构。设计数据结构时一定要注意它维持的不变量(invariant)是啥。比如在线段相交问题中，需要维持各线段自上而下的顺序，因此在经过交点时要交换两条线段的位置。

为了简化问题，我们先进行如下三个假设：

1. 没有竖直线段
2. 没有两条线段部分重叠。即任何两条线段最多只有一个公共点
3. 没有三线共点

对于线段相交问题来说，事件更新过程并不复杂：检查每对相邻线段是否相交。为了说明这个方法是正确的，我们只需证明所有的交点都能被这种方法检测到，即对于任意一个交点，在它左边总能找到一个位置，扫描线经过此位置时这两条线段相邻。容易证明：在该交点前的第一个事件就是这样一个位置。如下图。



假设si和sj中间还有一条sk，则要么sk的一个端点在扫描线右边，要么和si或sj相交且在si和sj的交点之前，均与扫描线在交点前第一个事件矛盾。

事件队列显然可以用堆来实现，但这里有一个问题：堆不能检测重复事件。一个解决方法是用更复杂的数据结构如平衡树，另一个方法是允许多个相同时间同时在队列里。对于x坐标相同的事件，可以按照字典序排序，即给二元组(x, y)排序。这相当于把扫描线逆时针旋转了一个无穷小量。

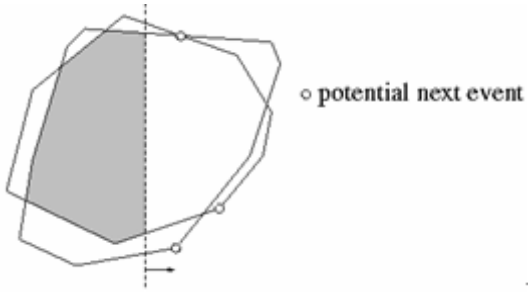
扫描线数据结构需要支持线段的插入、删除、交换相邻线段和取前驱、后继。这些操作均可以用平衡树在O(logn)时间内完成。

现在我们可以完整叙述线段相交算法了：首先把所有端点放入事件队列中，扫描线上线段集为空，然后每次取下一个事件。对于每个事件，根据类别进行处理：

线段左端点：插入线段并分别判断新线段与前驱和后继是否相交。
线段右端点：删除线段并检测被删除线段与它的前驱和后继是否相交。
交点：交换两条线段。对于新的上方线段，判断它是否和前驱相交；对于新的下方线段，判断它是否和后继相交。

算法的时间复杂度显然为$O((n+k)\log n)$，并不是最好，但很实用。

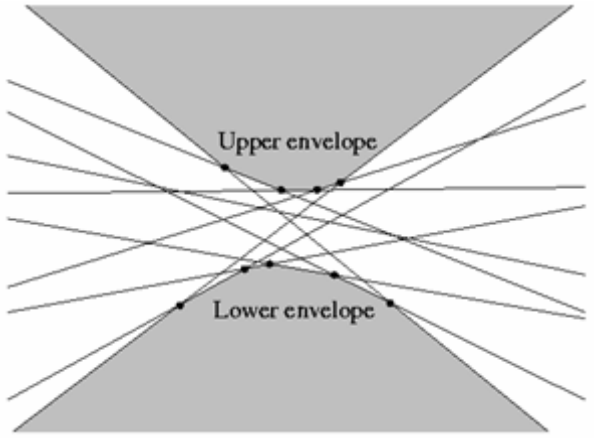
回到凸多边形相交问题。注意到每条边最多和另一个多边形的两条边相交，因此交点最多O(n)个，因此扫描算法的时间复杂度为O(nlogn)，这样整个算法的时间复杂度为$O(n\log^2 n)$，不是最优的。如果我们可以在O(n)时间内算出凸多边形（可能无界）的交，则由主定理可得整个算法的时间复杂度为O(nlogn)。



事实上，我们只需要把扫描算法修改一下。扫描线在任何时刻和每个凸多边形最多交于两点，因此在任意时刻扫描线上最多只有4个点，不需要用平衡树来保存；另外事件队列也不需要用堆，每次比较4个当前线段的交点和右端点，以及C_1和C_2的最左点（它们可能退化成$-\infty$）就可以知道哪个是下一个事件了。由于事件队列和扫描线数据结构的更新都是O(1)的，扫描算法的总时间复杂度为O(n)。

半平面的下包络线。平面上有n条直线$L_1, L_2, ..., L_n$，每条在直线$L_i : a_i x - b_i$。考虑每条直线下面的半平面$y \le a_i x - b_i$，它们交的边界称为直线集的**下包络线(lower envelope)**。同样可以定义**上包络线(upper envelope)**。

有趣的是，在没有竖直线的情况下，一般的半平面交问题总是可以转化为包络线问题，即求出所有下半平面的下包络和上半平面的上包络，然后把二者加以合并。



事实上，半平面的下包络线问题和点集的上凸包问题完全等价：它们只是同一个问题的不同表现方式。为了看清这一点，我们首先学习对偶。

对偶性(duality)。为了进行对偶变换，我们把直线表示成$y = ax - b$。注意这样的表示法无法表示竖直线，我们只能暂时忽略它们。我们把(a, b)看成一个点在新平面的坐标，因此$y = 7x - 4$对应新平面上的点(7, 4)。我们把原来的平面称为**主平面(primal plane)**，而系数构成的点(a, b)所在的新平面称为**对偶平面(dual plane)**。

考虑主平面上的点$p = (p_x, p_y)$。所有经过该点的非竖直线都满足$p_y = ap_x - b$。这些直线集在对偶平面上是点集$L = \{(a, b)|p_y = ap_x - b\} = \{(a, b)|b = p_x a - p_y\}$，它恰好是一条直线！因此不仅主平面上的直线对应对偶平面上的点，主平面上的点也（从某种意义下）对应对偶平面上的直线！正式地，我们用星号(*)来表示这种对偶变换。考虑主平面上的点$p = (p_x, p_y)$和直线$l : (y = ax - b)$，它们的对偶直线和点分别为：

$$\begin{cases} l^* = (a, b) \\ p^* : (b = p_x a - p_y) \end{cases}$$

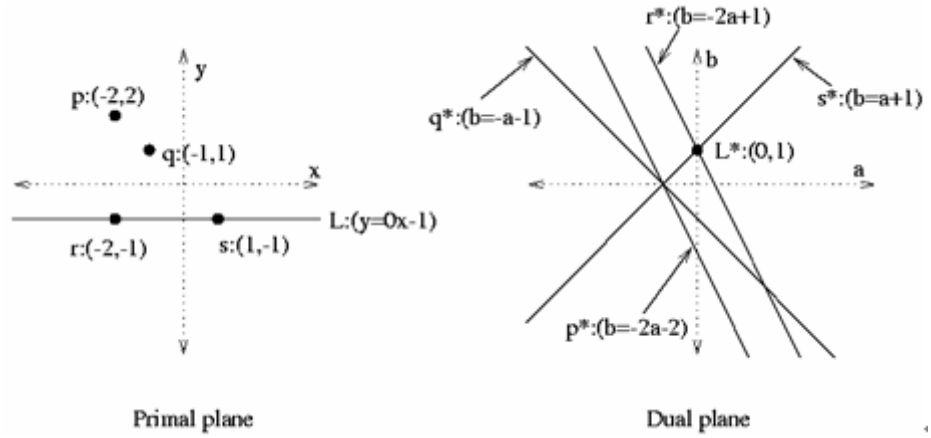
从对偶平面到主平面上的变换完全一致。

对偶变换有以下性质，它们都容易用简单的代数变形加以验证：

自反性*(self inverse)*：$(p^*)^* = p$

保序性*(reversing)*：主平面上点p在直线l的上方、线上、下方当且仅当在对偶平面中点l*在直线p*的上方、线上、下方。

保交性*(intersection preserving)*：主平面上直线l_1和l_2相交于点p当且仅当对偶平面上p^*同时经过点l_1^*和l_2^*。



共线(collinearity)/共点(coincidence)：在主平面上三点共线当且仅当在对偶平面上它们的三条对偶直线共点。

有了对偶的基本知识，我们可以来叙述下包络问题和上凸包的联系了。

定理：设P为平面上的点集。则P的上（下）凸包中点的逆时针顺序等于对偶平面上直线集P*的下（上）包络的从左到右顺序。

这里有一个看起来不太好的差异：上下凸包是连在一起的，但上下包络线却是分开的。如何理解这一点呢？这需要从**射影几何(projective geometry)**的观点解释主平面和对偶平面，这里不再讨论。

低维线性规划。线性规划问题我们已经在图论部分简单介绍过了。我们已经看到：很多运筹学问题都可以转化成特殊的线性规划问题。在几何中，也有一些有趣的优化问题可以转化为低维线性规划问题。这里我们给出一个O(d!n)的随机增量算法。

考虑二维情形。所有的约束都是二维不等式，即半平面：

$$h_i : a_{i,x}x + a_{i,y}y \leq b_i$$

而目标函数也可以用向量$c = (c_x, c_y)$表示。则线性规划问题是找一个点$p = (p_x, p_y)$似的满足所有不等式约束（即在这些半平面交中）且让点积$c_x p_x + c_y p_y$最大。在我们的例子中假设$c = (0, -1)$，即y越小目标值越大。

假设**可行域(feasible region)**是有界的，因此可以找到两个平面的交点对于目标函数来说是有限的。设这两个平面为h_1和h_2，下面我们需要依次增加平面h_3，h_4，...，每次迭代更新目标最优值。设加入第i个面后的可行域为C_i，此时的最优值点为v_i，则：

1：v_{i-1}符合约束h_i，则最优值不变，即v_i=v_{i-1}

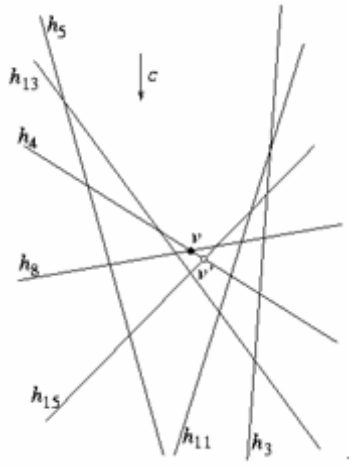
2：v_{i-1}违反约束h_i，必须寻找新的最优值。如果可性域非空，则新的最优点一定会在h_i的边界l_i上。直观上，如果v_i不在l_i上，则可以从v_{i-1}到v_i连一条线段。注意到从v_{i-1}走到v_i时目标函数值是单调递减的，且这条线段一定和l_i相交（因为从非可性域走到可行域），因此走到l_i上就停下来是最明智的，因为越走解越差。

现在问题变为：l_i上哪个点最优？这是一个一维线性规划问题，只需要考虑每个半平面，每个半平面约束对应l_i的一条射线。这样，只需要简单的不断更新可性域的两个边界（它在l_i上是一个区间），最后挑选出两个端点中目标函数较大的左右新最优值点v_i。

现在扩展到d维。我们仍然需要找到d个半空间，它们的交集定义一个有界解，然后把d个半平面重排使得它们是前d个平面。设v_d为初始最优值，然后每次增加一个半空间。我们每次仍然需要检查v_{i-1}是否在半空间h_i里，如果在则v_i=v_{i-1}否则v_i在h_i边界上。我们仍需要把其他半空间和此边界相交，问题转化为一个d-1维问题。这样，我们通过解决d-1维线性规划问题解决了d维线性规划问题。

对于二维问题，显然最坏情况时间复杂度为$O(n^2)$，因为每次最多需要计算一条直线和n个半平面的交，而这需要O(n)时间。如果在增量过程之前随机打乱一下各个半平面的顺序（注意前两个半平面应该不变），结果又会如何呢？我们需要考虑所有n!个可能排列下的平均时间复杂度。注意我们并没有假设输入数据遵守某种随机分布，而只考虑了可能的排列（即：没有最坏输入，只有最坏排列）。

我们用**反向分析(backward analysis)**技巧来记性复杂度分析。考虑下面这个图（不考虑平面编号），平面数i=7。这个图是怎么得到的呢？如果刚加入的半平面是h_5，则上次的开销是O(1)；但如果是h_4或h_8则上次开销为O(i-1)。

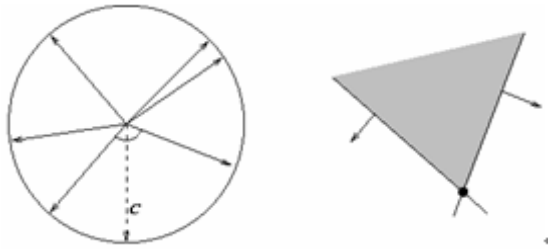


这个分析和下图的具体结构无关，即一定恰好两个平面对应于耗时操作。这样，有2/i的概率时间复杂度为O(i-1)而其他情况都为O(1)，因此每一步的期望为：

$$\frac{2}{i}O(i-1)+\frac{i-2}{i}O(1)\leq O(1)$$

由期望的线性性质立刻得：总时间复杂度的期望为O(n)。

前面遗留下一个问题：如何找到初始点？在二维情况下有一种简单的处理方法：把所有法线画到圆中，则包含c的那个区间端点对应的半平面即为所求，如下图。

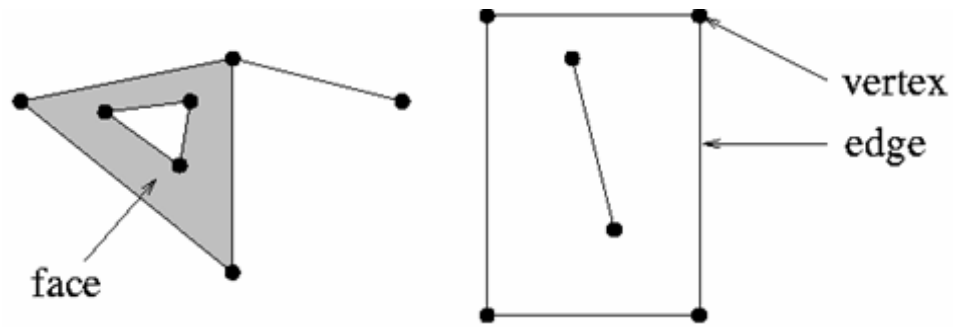


在多维情形下，每次通过一个高斯消元法可以求出半空间和超平面的交，从而把问题转化为d-1维线性规划，这里不再叙述。

8.2.4 平面剖分

本节讨论平面剖分。最典型的应用是地理信息系统：我们不仅关心组成地图的线段，也关心它们围成的区域，甚至对区域进行重叠，或者定位等。所有算法首先依赖于下面的线段相交问题。

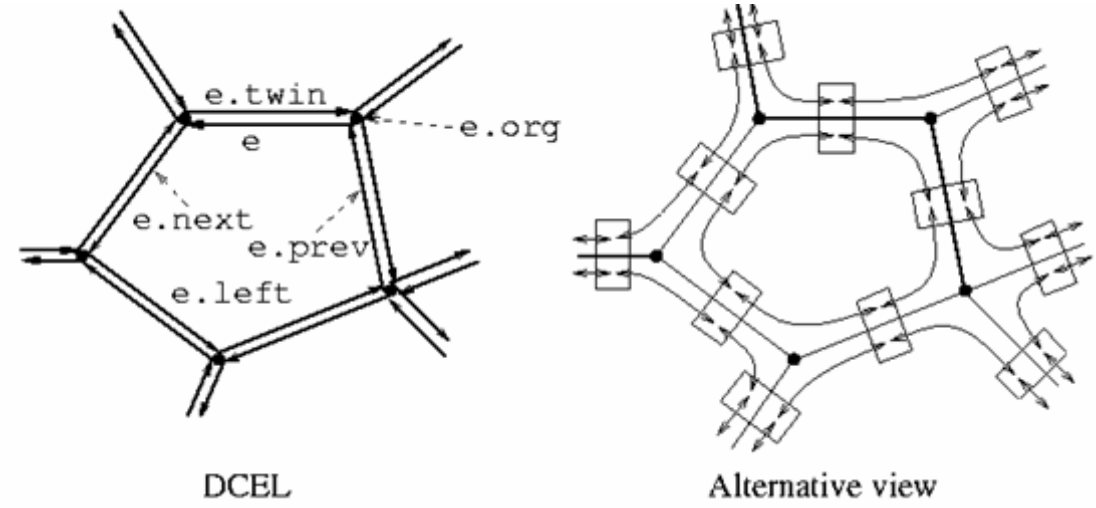
平面剖分(planar subdivision)。除了线段，我们往往需要考虑更复杂的几何结构。平面剖分就是一个典型的例子。



对于连通分量为C的平面图，有欧拉公式$V-E+F-C=1$，因此很容易得到平

面图的边数和点数都是O(n)的。有很多方法表示一个平面剖分，这里介绍**双向连通边列表（doubly-connected edge list, DCEL）**。

DCEL主要是关于边的信息，但也储存其他几何信息。一般来说一个DCEL包含三个记录：顶点表，边表和面表。每条边被表示为两条半边(half edge)。通常我们假设区域里是没有洞的，而对于一般情况下通过引入虚拟边(dummy edge)的方法把洞和面边界连通。在这个假设下，每个区域的边界边构成一个循环链表。



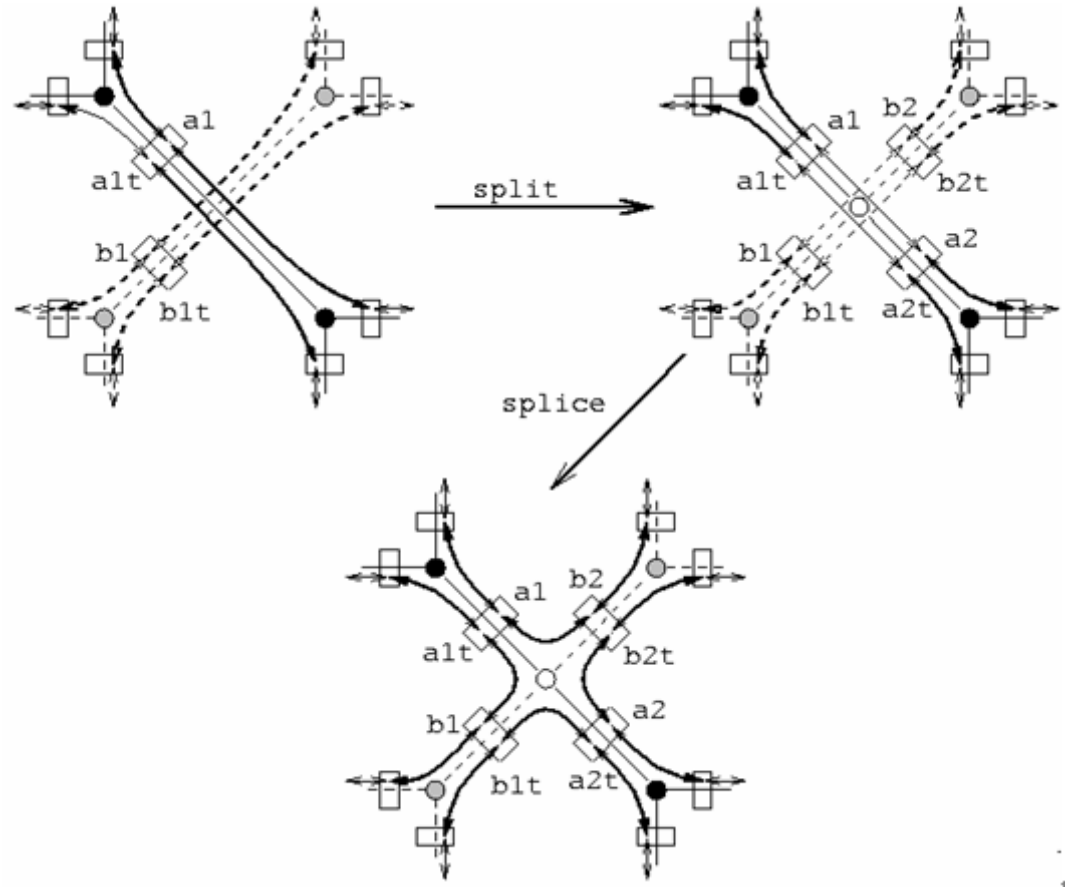
顶点表：每个顶点储存坐标和任一条从它出发的半边v.inc_edge。

边表：同一条边对应的两条半边互为twin，每条半边有一个origin和一个destination。每条半边和两个面相邻，一个左面一个右面。我们储存每条边的origin，即e.org和左面e.left也称为**关联面(incident face)**，以及循环链表中的下条边e.next和上条边e.prev。边的destination和右面可以通过访问twin的origin和左面完成。

面表：储存以该面为关联面的任一条半边，即f.inc_edge。

有了DCEL，我们很容易枚举一个面f上所有顶点。只需要从f.inc_edge开始沿着循环链表走一圈（每次令e = e.next即可）。

平面剖分的合并。有了DCEL，我们可以叙述平面区域的合并问题：给出两个平面剖分S_1，S_2的DCEL，求出合并后的DCEL。显然合并后的平面剖分中顶点是两个剖分中的顶点加上一些新的点。这些新的点都是某两条线段的交点，一条在S_1中，一条在S_2中。



如上图。我们可以仍然用平面扫描算法来计算。每次遇到S_1和S_2的线段相交时需要先把两条线段各分成两份(split操作)，然后把四个线段合在一起(splice操作），代码如下：

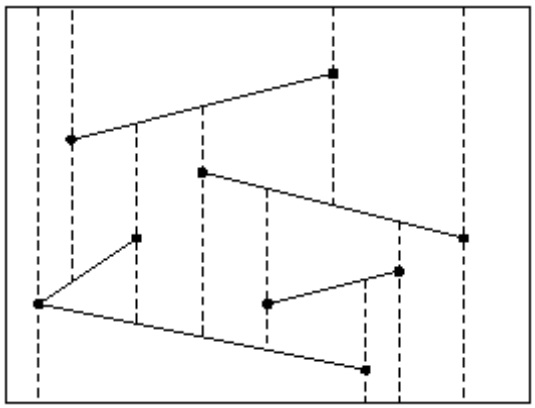
```
Split(edge &a1, edge &a2) {                  // a2 is returned
    a2 = new edge(v, a1.dest());             // create edge (v,a1.dest)
    a2.next  = a1.next;   a1.next.prev  = a2;
    a1.next  = a2;        a2.prev  = a1;
    a1t = a1.twin;        a2t = a2.twin; // the twins
    a2t.prev = a1t.prev;  a1t.prev.next = a2t;
    a1t.prev = a2t;       a2t.next  = a1t;
}
```

点在平面剖分中的位置。给一个平面图G的嵌入，它把平面划分成若干区域。如果G每个点的度不小于2，则各个域都是简单多边形。假设G是连通的。显然可以逐区域检查（调用刚才介绍的多边形内判定算法），但这样做速度太慢。梯

```
Splice(edge &a1, edge &a2, edge &b1, edge &b2) {
    a1t = a1.twin;   a2t = a2.twin;      // find the twins
    b1t = b1.twin;   b2t = b2.twin;
    a1.next  = b2;   b2.prev  = a1;      // link the edges together
    b2t.next = a2;   a2.prev  = b2t;
    a2t.next = b1t;  b1t.prev = a2t;
    b1.next  = a1t;  a1t.prev = b1;
}
```

形剖分(Trapezoidal Map)可以用来加速点定位，它的大小是O(n)的，期望构造时间为O(nlogn)且用它做平面点定位的期望询问时间仅为O(logn)，其中期望不取决于区域和询问点，而取决于对象的插入顺序。

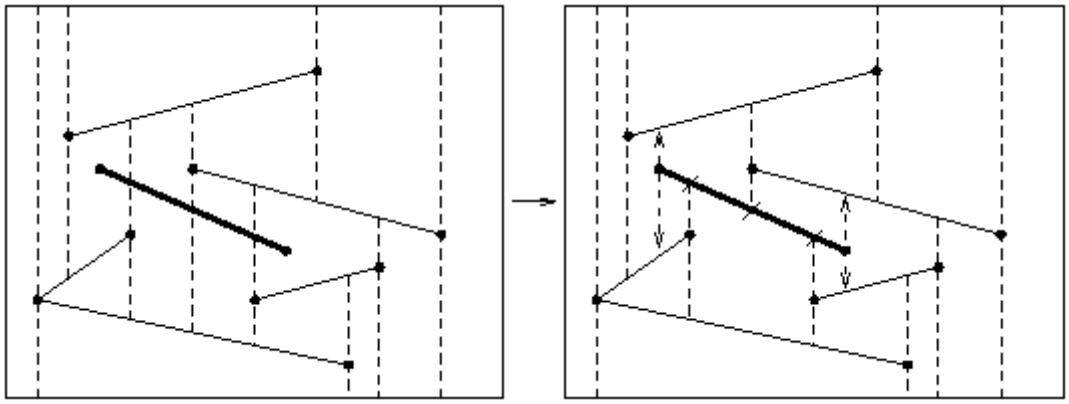
假设输入是一个线段序列，没有两个端点的x坐标相同，且没有竖直线，很容易定义如下的梯形剖分图：从每个顶点出发向上向下各发射一颗子弹。子弹撞到线段后消失，留下的痕迹和原始序列一起构成梯形图。一般来说我们把整个图放在一个很大的包围盒中，如下图所示。



增加了线段以后图的组合复杂度如何呢？最多6n+4个顶点和3n+1个梯形。这一点从发射子弹的过程很容易得到。每个梯形由恰好四个实体构成：上线段，下线段，左支撑点和右支撑点。梯形剖分可以用扫描的方法得到，但这里采用一种随机增量的方法，因为后面的点定位部分依赖于这种方法。

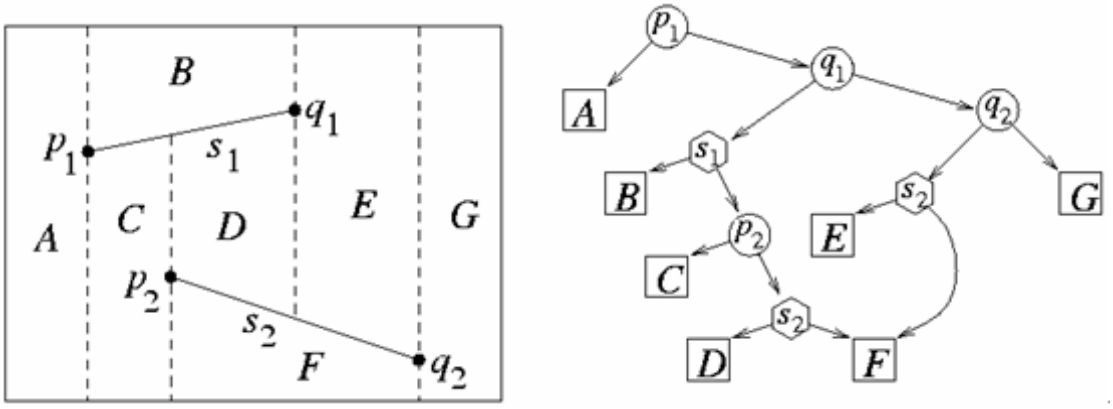
如下图，增量构造的三个步骤分别为：

步骤一：找到新线段两个端点所在的梯形步骤二：从两个端点出发各发射一颗子弹步骤三：修正与该线段相交的所有梯形，即把不应有的子弹轨迹截断。



第一步就是点定位问题，它的方法稍后叙述，时间复杂度为O(logn)；定位之后步二是常数级别的，而步三的时间复杂度显然为O(k)，k为新增梯形数目。用和二维线性规划的分析同样技巧可以得出（这里略）：构造过程的时间复杂度期望为O(nlogn)。

点定位问题。下面我们叙述在梯形剖分中的点定位问题。为了快速定位，我们因为有向无环图来表示梯形剖分，如下图。



图中有两种顶点：x顶点和y顶点。其中x顶点代表一个端点，它的左右“子树”的x坐标分别小于和大于该端点的x坐标，而y顶点代表一条线段，它的左右“子树”分别在该线段的上方和下方。注意：仅当我们知道查询点的x坐标在一条线段两端点中间时我们才会访问该线段对应的y顶点。

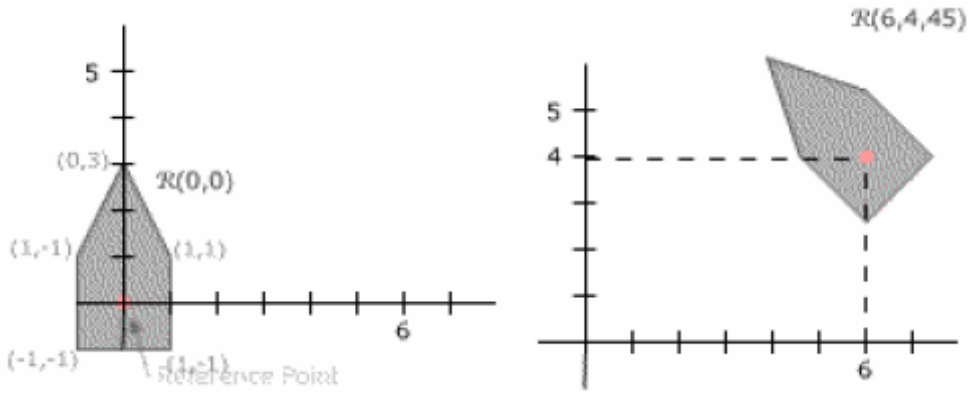
数据结构的修改是直观的，下面分两种情况，即两端点在同一个区域和不同区域。

(缺图)

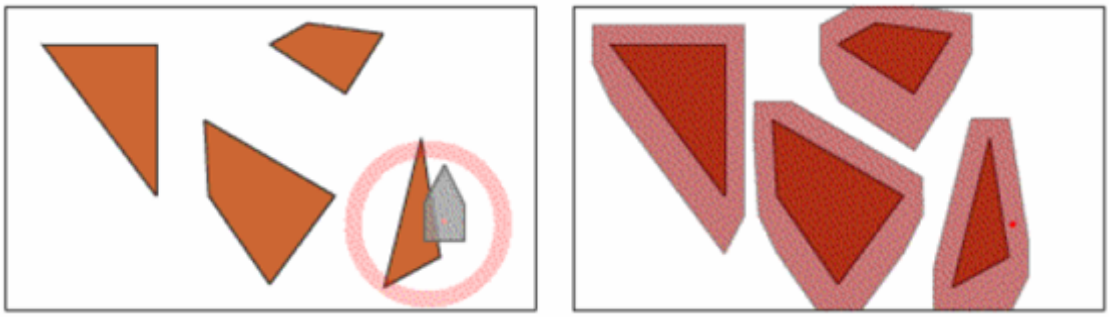
8.2.5 运动规划初步

本节讨论运动规划。这是一个有趣但较难的话题。它和机器人学有关，目标是让自主机器人规划自己的运动。一般的运动规划问题可以简单的表示为从起点到终点，不和任何墙（用事先规划）和人（用传感器）碰撞。本节进一步把问题做如下假设：只考虑2D平面区域，障碍物都是多边形的，机器人也是多边形的，场景是静态的（没有人），且没有运动限制（可以平移和旋转，而不像汽车那样受限制）。

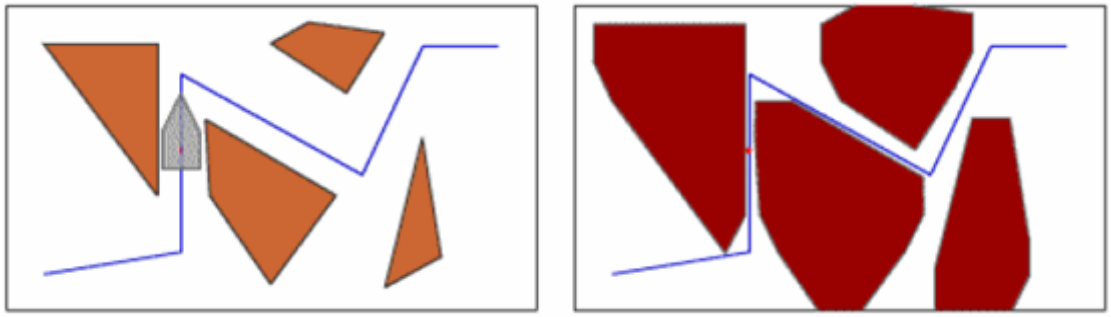
点机器人(point robot)。由于机器人是个多边形，我们需要规定一个“在原点”时的机器人多边形R(0,0)，用$R(x, y, \Phi)$表示中心在(x, y)且偏移角（以R(0,0)为基准逆时针旋转的角度）为Φ时的机器人状态。R(x, y, 0)简写为R(x, y)。



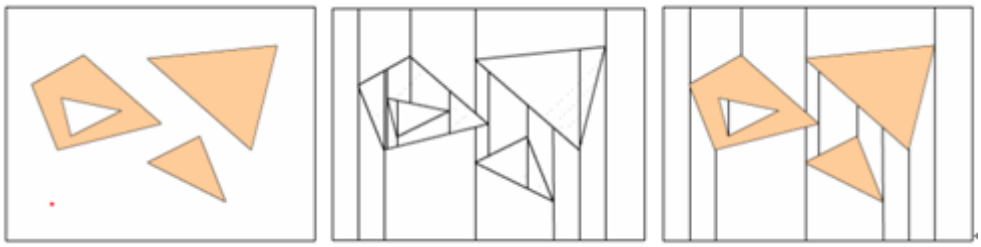
我们区分两种空间。一种是**工作空间(work space)**，即真实的障碍物地图。另一个是**配置空间(configuration space)**，即机器人的中心不能到达的位置集。下图是工作空间和配置空间的例子。



如下图，在假设机器人不能旋转的情况下，只要给工作空间的障碍物外面加一圈障碍，就可以把机器人看作是点状的，问题转化为在配置空间寻找**点机器人(point robot)**的路径。配置图中的无障碍部分称为C_{free}。



考虑点机器人的路径规划。首先对障碍图外边加一个边框，然后对每个障碍物多边形进行梯形划分（即在每个结点处向上/向下延伸竖直线，直到它们碰到其他障碍物或边界），最后删除障碍中间的梯形区域，得到剖分后的配置空间。这一步是期望O(nlogn)的。

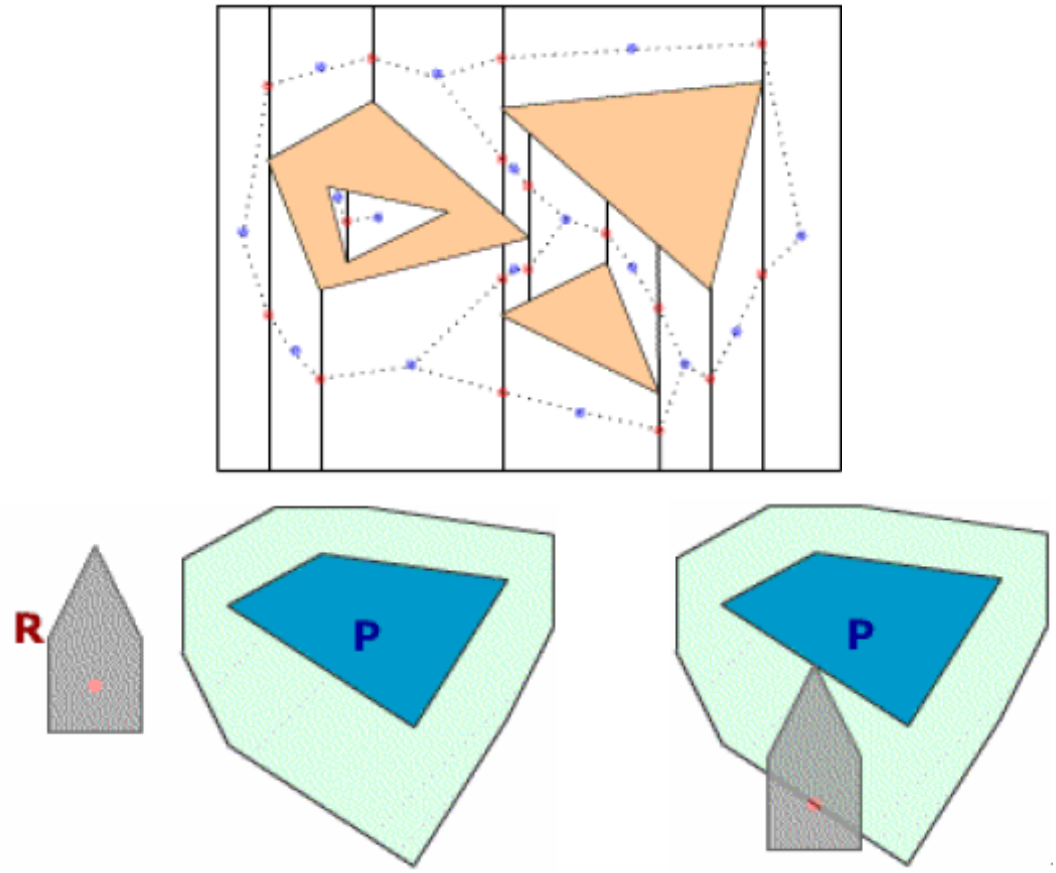


下面给每个梯形的中间创建一个梯形结点，再给每条竖直延伸线（注意对每个结点最多有上先两条线）创建一个线段结点，通过遍历梯形剖分图T(C_{free})的双向连通边表(doubly connected edge list)，可以在O(n)得到下面的图G_{road}。

首先找到起点和终点所在的梯形，如果是同一个，则直接走过去；否则在此图上进行宽度优先遍历，找到一条可行路径。这样，我们在O(nlogn)预处理时间和O(n)询问时间解决了点机器人的运动规划问题。

凸多边形的情形。如果机器人不是点状的，我们需要变换障碍物。本节介绍的Minkowski和可以用来进行障碍物变换，从而解决不带旋转时的多边形机器人运动规划问题。

我们把变换后的障碍称为C-obstacle，则下图为一个障碍对于机器人R的C-obstacle。形式上地，C-obstacle中的点$CP = \{(x, y) : R(x, y) \cap P \neq \emptyset\}$，直观地，CP的边界是R贴着P的边界“走一圈”时R中心的轨迹。

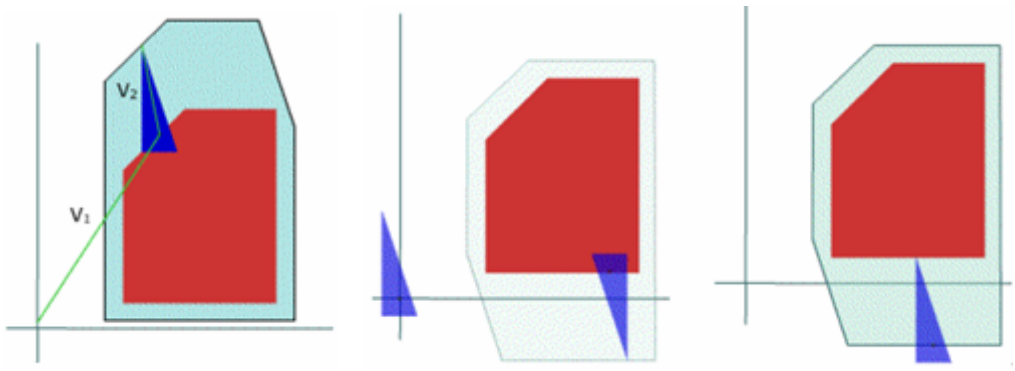


Minkowski和。两个集合S_1和S_2的**Minkowsi和(Minkowsi Sum)**被定义为

$$S_1 \oplus S_2 = \{p + q : p \in S_1, q \in S_2\}$$

其中$p + q = (p_x + q_x, p_y + q_y)$。因此两个集合的Minkowsi和是所有点对的向量和。如下图，对于多边形点集来说，Minkowsi和的直观含义是其中一个多边形（在下图中为三角形）的参考点（即在该多边形坐标系下的原点）沿着另一个多边形（在下图中五边形）的边界移动时得到的图形。

这个定义看起来和CP的定义有点不一样：Minkowsi和是参考点沿着多边形移动，结果取决于边界的轨迹；CP是多边形边界沿着多边形移动，结果取决于参考点的轨迹。



对于点$p=(p_x,p_y)$，定义$-p=(-p_x,-p_y)$，对点集S，定义$-S=\{-p:p\in S\}$，则对于多边形A，-A是它的翻转。下面的定理说明了二者的紧密联系。

定理1：对于机器人R和障碍物P，P的C-obstacle为$P\oplus(-R(0,0))$

换句话说：先把R翻转，然后求与P的Minkowski和，就可以得到P的C-obstacle。

证明：只需证明R(x, y)和P相交当且仅当$(x,y)\in P\oplus(-R(0,0))$。

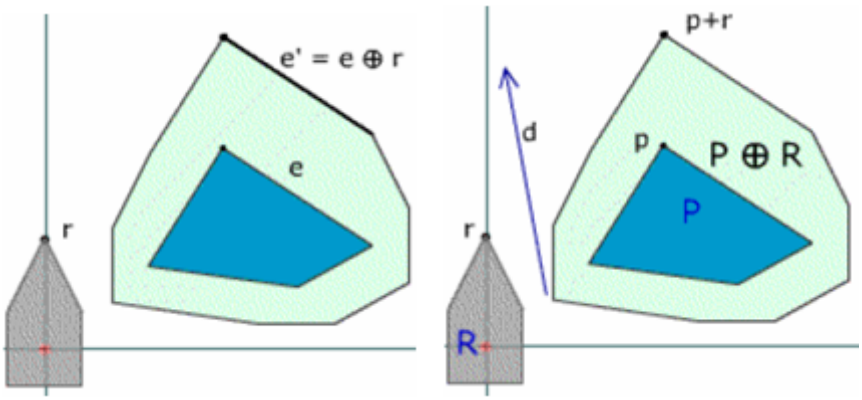
第一步：假设R(x, y)和P相交，q为其中一个交点。因为$q\in R(x,y)$，$(q_x-x,q_y-y)\in R(0,0)$，因此$(-q_x+x,-q_y+y)\in -R(0,0)$。由于$q\in P$，$(x,y)\in P\oplus(-R(0,0))$。

第二步：假设$(x,y)\in P\oplus(-R(0,0))$，则存在点$(r_x,r_y)\in R(0,0)$和$(p_x,p_y)\in P$使得$(x,y)=(p_x-r_x,p_y-r_y)$。变形得$p_x=r_x+x,p_y=r_y+y$，因此R(x,y)和P相交。

极点。对于凸多边形，每一个方向都有一个或多个**极点(extreme point)**，它在该方向是最远的。如果P和R都是凸多边形，可以得出两个有意思的性质。

性质一：$P\oplus R$对方向d的极点恰好是P和R各自在此方向的极点之和。

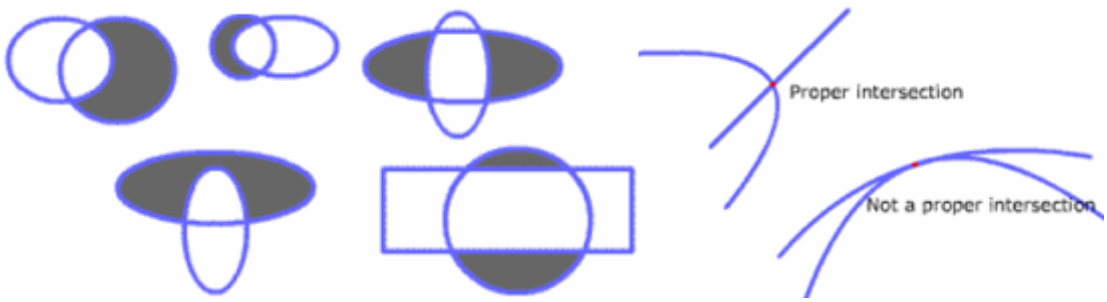
性质二：$P\oplus R$的每条边都来自于P或R的一条边，且P和R的每条边最多贡献一次。



根据性质二，我们立即得到：

定理2：若P和R都是凸多边形，边数分别不超过n和m，则$P \oplus R$也是凸多边形，边数最多为n+m。

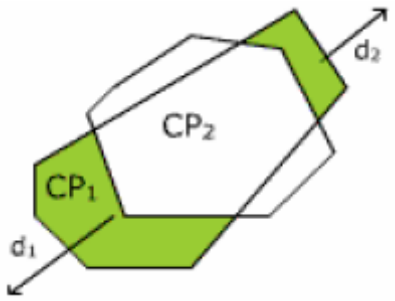
伪盘片对。一个平面物体对o_1, o_2被称为**伪盘片对(pseudodiscs pair)**当且仅当o_1-o_2和o_2-o_1都是连通的，如下图(a)。图(b)是两个反例，因为阴影部分不连通。伪盘片对最重要的性质是：最多有两个规范交点。规范交点是不在切点或端点上的交点，如图(c)。

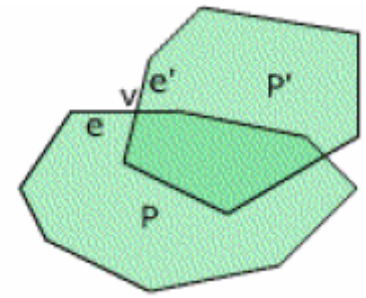


方向。考虑两个凸多边形P_1和P_2。对于一个方向d，如果P_1在该方向的极点比P_2的更远，说在此方向上P_1比P_2更极端。图(a)中的两个方向d_1和d_2（对应于二者的两条正切线，用虚线画出）上P_1和P_2的极端程度一样。这个关系可以用图(b)的圆来表示，比如从方向d_1到d_2顺时针旋转的过程中，P_2始终更极端。

定理3：如果P_1和P_2都是凸多边形，且没有公共内点，则它们和同一个凸多边形R的Minkowski和$P_1 \oplus R$和$P_2 \oplus R$形成一个伪盘片对。

证明：用反证法。由对称性，只需要假设CP_1-CP_2不连通，如下图。



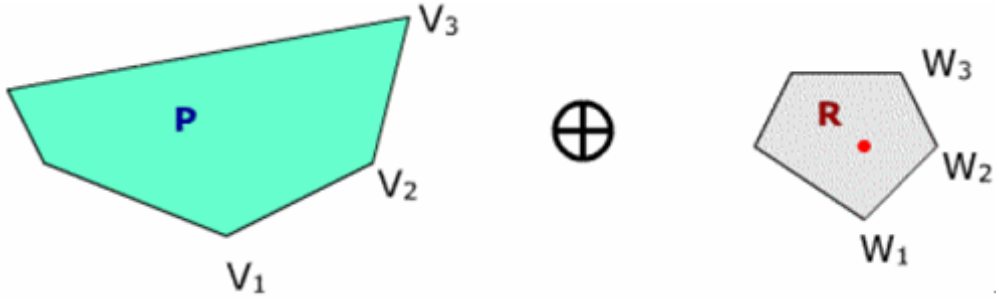


则在d_1和d_2方向CP_1都比CP_2更极端，且从d_1到d_2和从d_2到d_1都至少存在一个方向使得CP_2更极端（这样CP_1才会不连通）。由于CP_1和CP_2是P_1和P_2与凸多边形R得到的Minkowski和，因此：在d_1和d_2处P_1比P_2极端，但从d_1到d_2方向和d_2到d_1方向都至少存在一个方向使得P_2比P_1极端，与方向圆矛盾。

定理4：设S为一共包含n条边的多边形伪盘片对，则它们并的复杂度为O(n)。

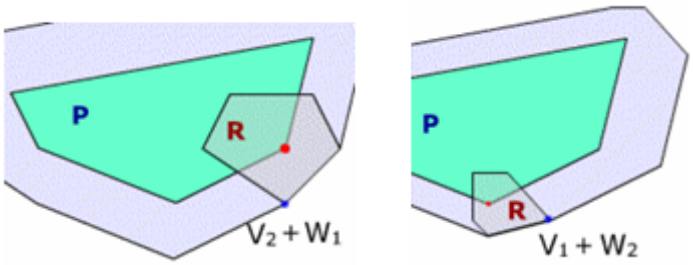
证明：并的边界有两种点，一是原来的点，二是交点。对于每个原来的点，算到自身；对于每个交点，算到它的“下一个点”，则每个点最多被算两次，因此复杂度不超过2n。

凸多边形的Minkowski和。把凸多边形P和R的顶点按逆时针顺序排列，从y坐标最小的点算起，如下图。我们希望按逆时针的顺序给出$P \oplus R$的边界。



首先计算V_1+W_1，它是$P \oplus R$的最低点。下一个点一定是V_2+W_1么（即R的参考点滑到P的第二个点）？在图(a)中，R的确眼着边(V_1, V_2)滑到V_2，“刻”出边(V_1+W_1, V_2+W_1)但是在图(b)中，刻出的边应该是(V_1+W_2, V_2+W_2)！问题的关键在于边(V_1, V_2)和(W_1, W_2)的角度哪个大。

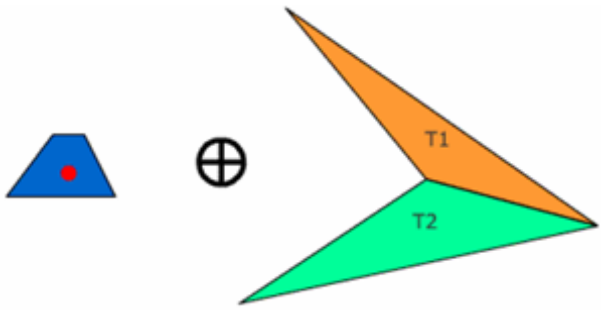
理解了这一问题，我们很容易叙述出完整的Minkowski和算法：它简单的把R沿



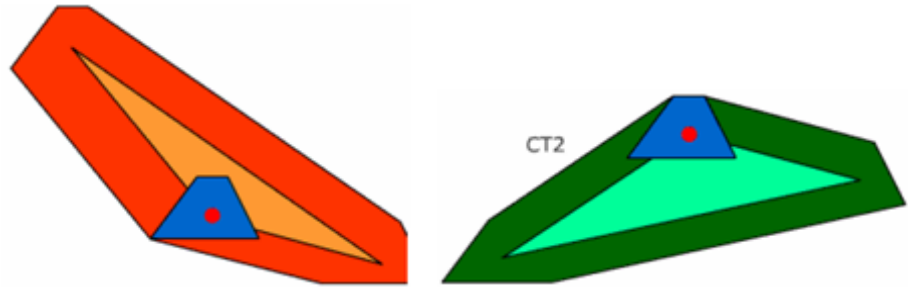
着P饶一圈，每次判断两个多边形Φc>的角度，以确定$P \oplus R$的下一个顶点。算法如下。时间复杂度显然为$O(n)$。

```
Repeat until iterated around both polygons {
   Add Vi + Wj
   if angle(ViVi+1) < angle (WjWj+1) {
      //Next robot edge falls inside
      j++
   } else if angle(ViVi+1) > angle(WjWj+1) {
      //Next robot edge extends out
      i++
   } else {
      //Both edges parallel
      i++, j++
   }
}
```

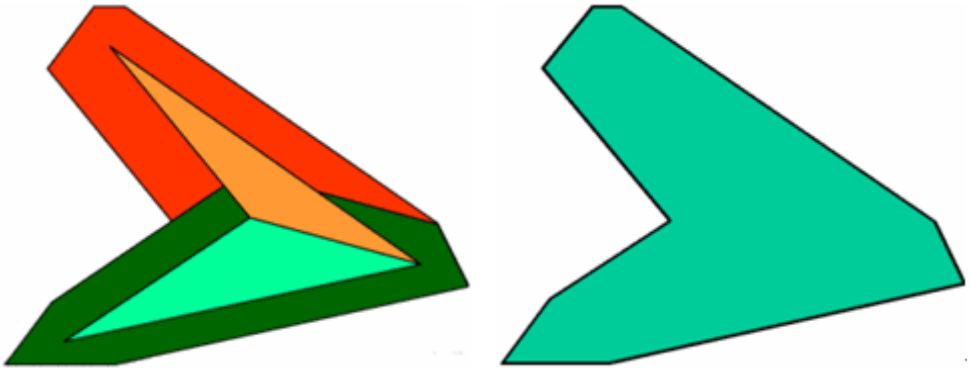
一般多边形的情形。刚才的算法只适用于凸多边形。对于一般多边形，需要先进行三角剖分，然后计算每一对三角形的Minkowski和，最后求并。



下图是$CT_1 = T_1 \oplus R$和$CT_2 = T_2 \oplus R$的结果：



合并后的结果为：

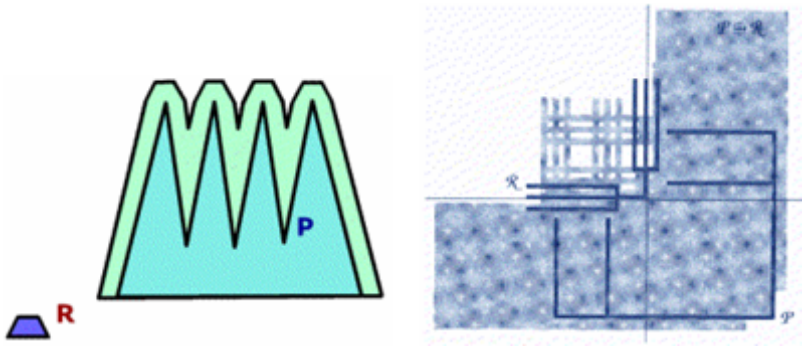


Minkowski和的复杂度分析。分为三种情况。

情况一：P和R分别为n顶点和m顶点的凸多边形。由刚才的算法，复杂度为O(n+m)。

情况二：P为n顶点一般多边形，R为m顶点凸多边形。则P的三角剖分结果为n-2个三角形，求Minkowsi和后得到n-2个最多m+3边形。由于n-2个三角形是两两没有公共内点的，根据定理3，每两个m+3边形都形成伪盘片对，由定理4得并的复杂度为O(nm)。

情况三：P和R分别为n顶点和m顶点的一般多边形，则分别进行三角剖分，再两两求Minkowski和，得到(n-2)(m-2)个常数复杂度的多边形，因此并的复杂度为$O(n^2m^2)$。



有了Minkowski和这一工具，我们现在讨论无旋转时凸机器人的运动规划。一般的机器人都是常数复杂度的凸多边形，对于每个障碍都可以用O(n)时间把它进行三角剖分，用O(n)时间算出所有C-obstacle，然后在O(n)时间内求出并，得到C-obstacle。有了各个障碍的C-obstacle，可以用分治的方法求出它们的并：

第一步：三角剖分。设障碍P_i的复杂度为m_i，则单独剖分的时间复杂度为$O(m_i \log m_i)$，总时间复杂度为：

$$\sum_{i=1}^{t} m_i \log m_i \leq \sum_{i=1}^{t} m_i \log n = n \log n$$

第二步：计算C-obstacle。对于常数复杂度的凸多边形机器人，与O(n)个三角形的Minkowski和可以在O(n)时间内得到。

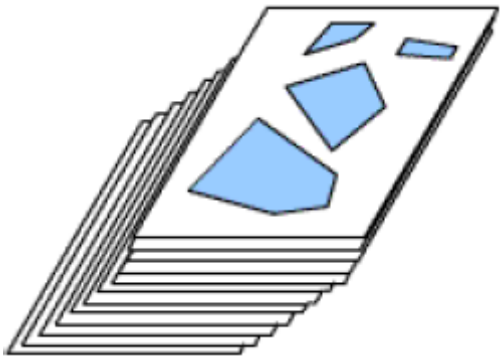
第三步：合并。假设需要合并C_{forb1}和C_{forb2}，它们的复杂度分别为n_1, n_2，而并的复杂度为k，则合并的时间复杂度为$O((n_1+n_2+k)\log(n_1+n_2))$。由于总的禁止空间复杂度为O(n)，因此合并步骤的复杂度为O(nlogn)。分治法的递归式为：$T(n)=2T(n/2)+O(n\log n)$，解得$T(n)=O(n\log^2 n)$。

这样，预处理的总时间复杂度为$O(n\log^2 n)$。得到图后，任意一个询问可以在O(n)时间内回答。

有旋转的情形。不进行旋转，机器人往往不能进入一些狭窄的区域。前面的讨论都只针对一个固定角度进行，而在角度变化时，每个障碍在不同角度时对应于不同的C-obstacle。不同的方向对应于不同的配置空间，这样得到一个多层结构：

我们需要把这个多层结构合并成一个完整的图。在这个图上有两种操作：

平移操作：在同一个层次上移动。旋转操作：从一个层次移动到另一个层次（相邻的下一个或上一个）。



在相邻两个层次可以求出覆盖(overlay)，以确定两个配置空间的公共部分。

8.2.6 小结与应用举例

本节介绍经典的几何问题，在问题中介绍常用的数据结构。

第一节介绍定位问题，包括点定位和范围查找。本节介绍的基本问题有：点在二维空间中的多边形内、点在三维空间中的平面多边形内。点在星形多边形内和凸边形内可以用二分查找做得更快。范围查找问题可以分为报告问题和计数问题，很多情况下二个问题可以用类似的方法解决。本文介绍了K-D树和范围树，它们只能解决正交查找问题。其中范围树中的分块层叠技术值得仔细体会。

第二节讨论多边形。首先是凸多边形，介绍了判定算法和均衡分层表示，并提出了三个经典问题：判断点在凸多边形内、确定凸多边形和直线的交、判断两个凸多边形是否相交。接下来是简单多边形。首先仍然是判定问题，并从中引出了规范相交和非规范相交的区别和它们的判定方法。接下来是和“看见”有关的问题，包括星形多边形的核和画廊问题。其中核的求解要用到半平面交，它在第三节中讨论。本节的重点是多边形的三角剖分。我们首先给出三角剖分相关的一些性质和简单证明，然后用割耳法把最容易想到的$O(n^4)$算法优化到$O(n^2)$。三角剖分可以在线性时间内完成，但方法比较复杂且系数较大。这里给出一个基于梯形剖分的算法，它首先用扫描法在O(nlogn)时间内把多边形梯形剖分，同时用连接相对支撑点的方法删除歧点，得到单调多边形，最后用一个栈在线性时间内将单调多边形剖分。本节最后介绍凸划分的简单结果和H-M近似算法。

第三节讨论半平面交和低维线性规划并第一次正式介绍几何中常用的随机增量算法。本节首先介绍线段相交的扫描算法，然后修改后得到了凸多边形相交的线性时间算法，并通过分治法得到了半平面交的O(nlogn)算法。本节的后半部分讨论低维线性规

划问题。算法本身并不复杂，但反向分析法应当引起读者的注意。由于这里的算法对于d来说并不是多项式的（系数为$c^d d!$，其中c为某常数），因此只对很小的d实用。本节还介绍了对偶变换，把半平面交分解为上下包络线后和凸包的上下链等同起来。

第四节主要应用在地理信息系统中，讨论平面剖分中点定位和地图覆盖问题。本节的重点是DCEL及其基本分裂、合并操作，基于线段相交扫描算法的地图覆盖算法以及平面剖分的梯形图的随机增量构造法。

第五节介绍运动规划中的基本问题，包括工作空间和配置空间的概念、梯形剖分图上的路径查找，以及Minkowski和的计算方法，其中凸多边形的Minkowski和是重点内容。有旋转的情形复杂很多，但有求精确解的多项式算法，有兴趣的读者可以参考相关书籍。

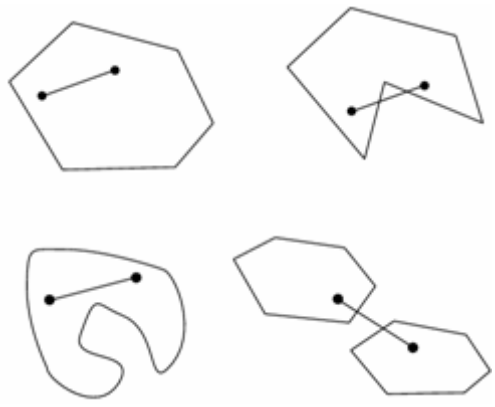
重要算法和程序列表：

8.3 几何结构及其应用

从本节开始，我们讨论三大几何结构：凸包、Voronoi图和直线的排列。凸包是相对而言是最直观的，应用也最广泛。

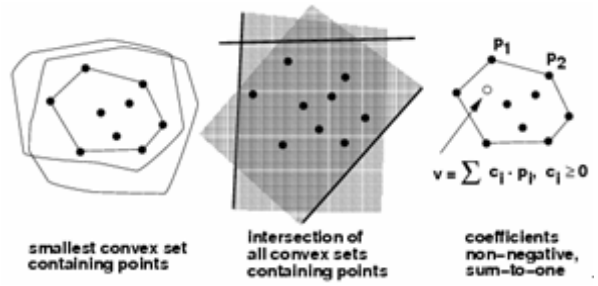
8.3.1 二维凸包与三维凸包

最重要的概念之一是**凸性(convexity)**。一个点集是凸的当且仅当连接任何两个点的线段上每个点都在点集中。



凸包有三个等价定义：包含所有点的最小凸图形，所有包含它的凸图形并，以及所有能被点集凸表出的点集，如下图。但三个定义都无法直接设计算法。

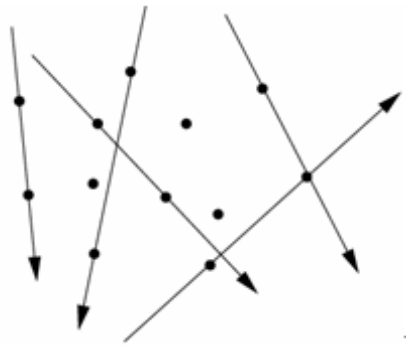
算法	程序	备注
点在多边形内判定	pos_poly.cpp	射线法和转角法，包括平面和空间中的多边形，时间复杂度均为O(n)
凸多边形和点、直线的位置关系	pos_convex.cpp	用二分法在O(logn)时间内判定点是否在凸多边形内，求出直线和凸多边形的交点
二维k-D树	kd2d.cpp	二维k-D树基本操作：建立和查询。建立时间为O(nlogn)，查询时间为$O(\sqrt{n}+k)$
二维范围树	rangetree2d.cpp	利用分块层叠技术的二维范围树基本操作：建立和查询。建立时间为O(nlogn)，查询时间为O(logn+k)。
三角剖分的基本算法	tri_basic.cpp	基本三角剖分算法，即原始的$O(n^4)$算法，基本的$O(n^3)$割耳算法和优化的$O(n^2)$割耳算法。
三角剖分的快速算法	tri_quick.cpp	利用扫描法得到单调多边形的剖分算法，时间复杂度为O(nlogn)。
凸划分的H-M算法	h_m.cpp	限定用对角线的凸划分问题的Hertel-Mehlhorn算法，输入为三角剖分，时间复杂度为O(n)，凸多边形数目不超过4M，其中M为最优值。
线段相交问题	seg_inter.cpp	线段相交问题的扫描算法，时间复杂度为O((n+k)logn)。
凸多边形相交	convex_inter.cpp	可能无界的凸多边形相交算法，时间复杂度为O(n+m)。
半平面交	halfplane_inter.cpp	半平面交算法，时间复杂度为(nlogn)。
二维线性规划	lp2d.cpp	二维线性规划的随机增量算法，时间复杂度的期望为O(n)。
地图覆盖问题	map_overlap.cpp	给出两个地图的DCEL，求合并后的DCEL。利用扫描法，时间复杂度为O((n+k)logn)，其中k为新增点数。
点在平面剖分中的位置	inside.cpp	利用梯形图的随机增量算法得到。期望建立复杂度为O(nlogn)，查询O(logn)。
障碍物的梯形图	groad.cpp	把多边形障碍物集合组织成梯形图供路径查询。建立的时间复杂度的期望为O(nlogn)，查询是O(n)的。
Minkowski和	minkowski.cpp	凸多边形的Minkowski和，一般多边形和凸多边形的Minkowski和及一般多边形的Minkowski和，时间复杂度分别为O(n+m)，O(nm)和$O(n^2m^2)$。
平移多边形的路径规划	motion.cpp	给常数复杂度的多边形机器人，在$O(n\log^2 n)$时间内可以构造出障碍图，支持O(n)时间的路径询问。



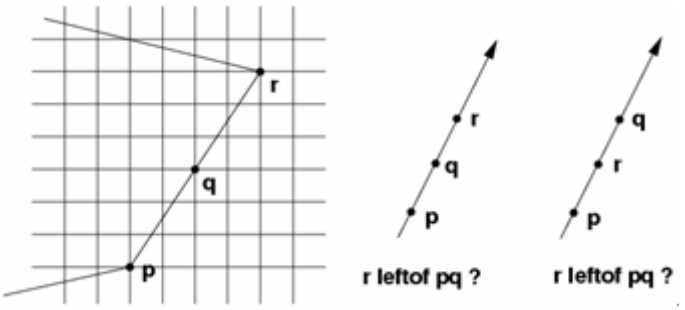
首先澄清一个问题：凸包的输出是什么？一般按逆时针输出凸包边界，要求相邻三点不共线。由于CH(S)本身的复杂度为O(n)，因此有希望设计出快速算法。

需要关注的地方：正确性、时间复杂度、鲁棒性、输入退化情况、对输出的敏感程度。

枚举算法：判断每条边是否为**极边(extreme edge)**，时间复杂度为$O(n^3)$。一条有向边(p, r)为极边当且仅当所有其他点都在它的左测。



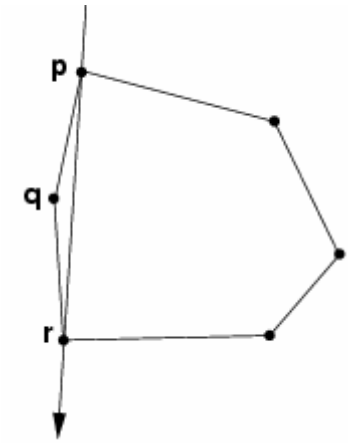
退化(Degeneracy)的情形。LEFTOF谓词一定返回真或假，那么三点共线的时候应如何处理呢？



如果情况(a)为真，则按p, q, r的顺序处理，凸包上会同时出现p, q, r；如果情况(b)为真，按p, q, r的顺序处理，凸包上会同时出现p, q, r。这个问题说明：只

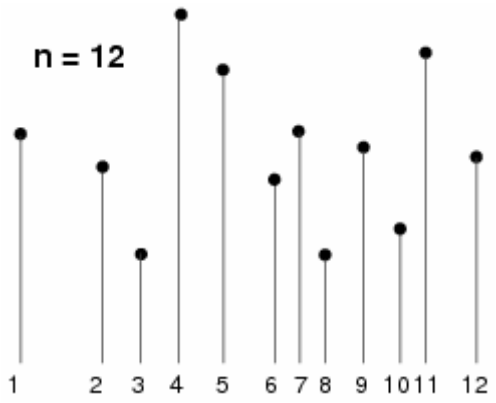
靠LEFTOF函数是不够的。

鲁棒性(robustness)的考虑。考虑浮点误差，下面的情况有可能为真！



R在pq左侧，p在qr的左侧，且q是pr的左侧。

修改的Graham算法。Andrew在1979做出了Granham扫描算法（1972）的一个修改，它从左到右给点排序，然后一个一个加到凸包中，它简单，能处理退化，并且鲁棒。



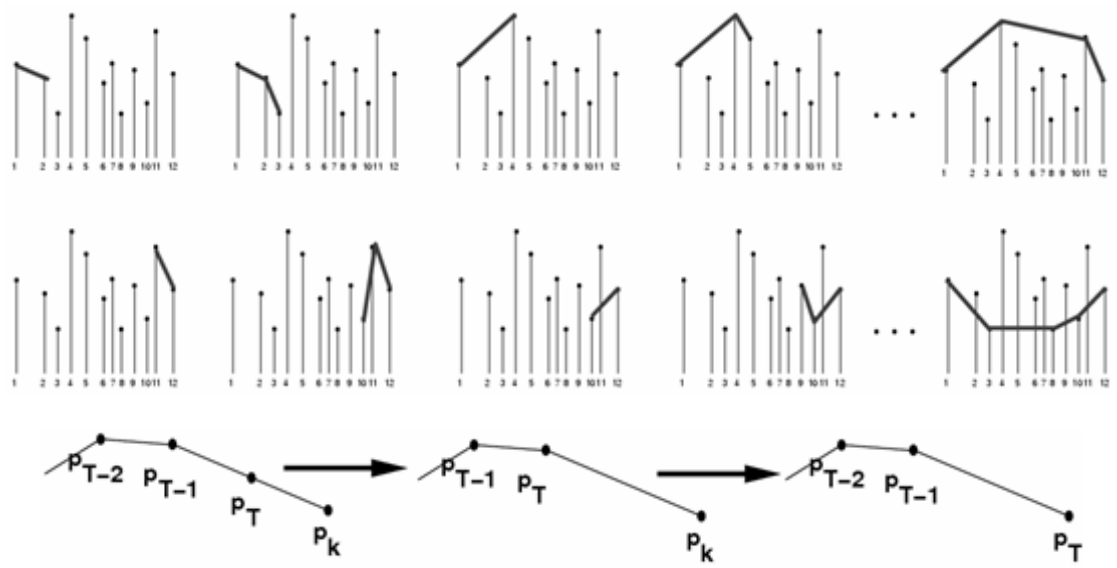
主过程：每次判断新点p_k和有向线段P_{T-1}-P_T。如果p_k在左侧，则P_T出栈，否则P_k入栈，如下：

下面是刚才例子的执行过程。

退化情形。稍微修改LEFTOF使得三点共线时返回true，则“按x排序”这一特性使得处理总是正确的。

对于x坐标相同的情况下，必须按照字典序排序，如下图。

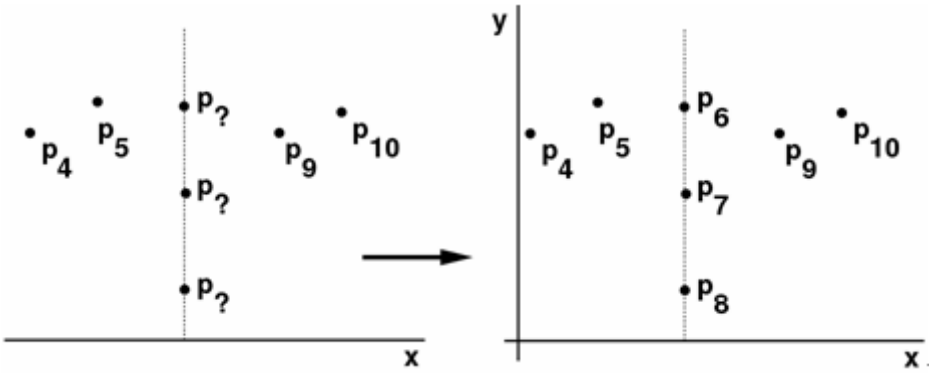
```
CONVEXIFY(S, p)
    /* S is a stack of points; TOS is t */
    while (S.len ≥ 2) and (p LEFTOF (p_{T-1}, p_T))
        POP(S)
    PUSH(S, p)

SWEEP-HULL( Array p_i )
    Sort p_i in place, by x coordinate
    STACK UpperHull = { p_1 }
    For i = 2 to n
        CONVEXIFY ( UpperHull, p_i )
    STACK LowerHull = { p_n }
    For i = n − 1 downto 1
        CONVEXIFY ( LowerHull, p_i )
    Remove first, last points of LowerHull
    Output UpperHull concat LowerHull
```



算法保证输出封闭多边形链，但误差可能会引起多点或少点的情况，即鲁棒但不保证结果正确。

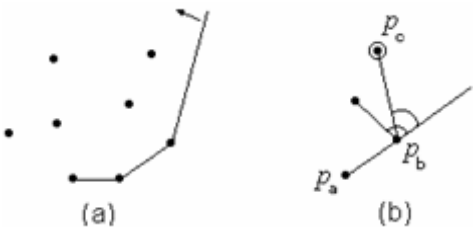
思考题：一、对于简单多边形，给一个O(n)算法计算凸包。二、给出凸包上边的无序列表，用O(n)的时间把它们按逆时针顺序连接起来。

三维凸包。和二维凸包类似，三维凸包输入n个点，输出包含它们的最小凸多面



体。二维凸包由极边组成，而三维凸包由极三角形组成，因此最显然的算法仍然是枚举算法：枚举每个有向三角形(i, j, k)(i¡j,k, j!=k)，如果其他点都在它的左侧，则它是一个极三角形。注意四点共面的情况，因此枚举法在计算完毕后还需要进行后处理，合并共面三角形。

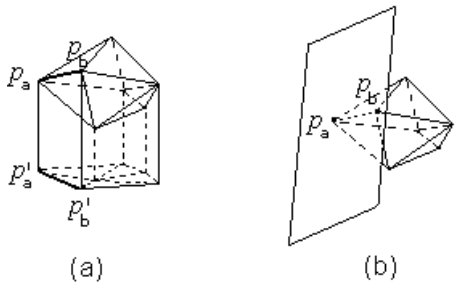
卷包裹法。首先我们来看二维凸包的另一个算法。它比Graham算法慢，但思路清晰，有启发性。它的基本思想是在平面中的那些点上敲上钉子，用一根绳子，一头绑住最外侧的一个点(起始点)，然后逆时针旋转，当绳子转回起始点时就裹出了凸包，如下图(a)。



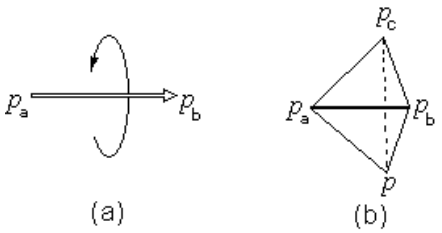
说得规范一些：开始时，以y坐标最小的点作为起始点p_O。假设当前求得的最后一

条凸包上的边为p_ap_b，依次考虑其他的点p，找出使得p_ap_b转到p_bp转角最小的点p_c（就是“最顺时针方向”的点），则p_bp_c为凸包上的下一条边。然后以p_bp_c作为p_ap_b，继续这一过程，直到$p_c = p_O$。

仿造求二维凸包的过程，我们可以这样求三维凸包。首先找一条肯定在$\mathrm{CH}^{3D}(P)$上的一条边。我们可以将所有点投影到$x\mathrm{O}y$平面，求投影点的二维凸包$\mathrm{CH}^{2D}(P)$，从中任取一条边$p'_ap'_b \in \mathrm{CH}^{2D}(P)$，则$p'_ap'_b$投影前的边$p_ap_b$可以作为起始边，如下图(a)。



下面的工作可形象地看成将一张纸贴紧p_ap_b，朝各个方向包裹所有顶点（上图(b)）。给出一个不同于p_a和p_b的点p，也就给出了一个平面p_ap_bp。所有这些平面都是绕p_ap_b的三角形。我们绕p_ap_b的旋转定义一个方向：如下图(a)所示，规定p_ap_b的右手螺旋方向为正方向，即大拇指朝向量$\overrightarrow{p_ap_b}$的方向，其余四指的方向就是环绕p_ap_b的正方向。于是p_c就是正方向上第一个三角形$p_ap_bp_c$的对应顶点。



如上图(b)所示，判断$\Delta p_ap_bp_c$是否在Δp_ap_bp的正方向上，可以求四面体$pp_ap_bp_c$的带符号的体积$V(\mathrm{T}(pp_ap_bp_c))$，若体积小于0，则不在正方向上，否则(大于0)在正方向上。

这样我们就得到了组成$\mathrm{CH}^{3D}(P)$的下一个三角形$p_ap_bp_c$。接下来分别以p_ap_c, p_cp_b作为p_ap_b，继续找下一个三角形，直到找到所有的三角形。

与求二维凸包不同的是，求二维凸包时下一个点只有一个，而以上的过程中，下一条边有两个选择(p_ap_c和p_cp_b)。显然两个都要继续考虑(如果还问考虑过的话)，为了对付这种不断扩展的过程，我们用一个队列Q存放需要考虑的边。每次从队首head[Q]取出一条边p_ap_b，找第一个转到的三角形$p_ap_bp_c$，扩展出两条边p_ap_c, p_cp_b，如

果它们尚未被考虑过，则加入队列Q。这样就能保证我们毫无遗漏地枚举出凸包上所有的三角形了。

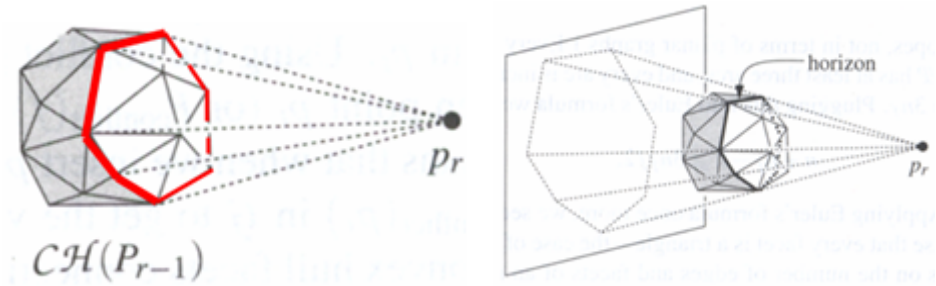
随机增量算法。下面介绍一种计算凸包的实用算法。它的期望时间复杂度为O(nlogn)，且编程相对比较简单。算法首先进行初始化，然后递增的加入各个点，每次修改凸包。因此问题的关键是：如何把CH(P_{r-1})变成CH(P_r)。

初始时需要一个四面体。可以先找两个不同点p_1, p_2，寻找和它们不共线的第三个点p_3，再找不共面的第四个点p_4。如果找不到，则只需要调用二维凸包算法！

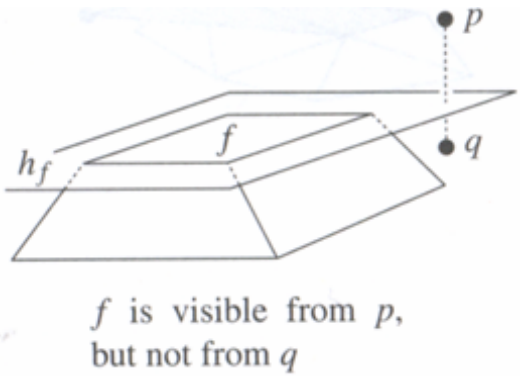
接下来计算剩下点的随机排列p_5, p_6, ..., p_n。每次加一个点，有两种情况：

情况一：新点在当前凸包内部，只需简单的忽略该点

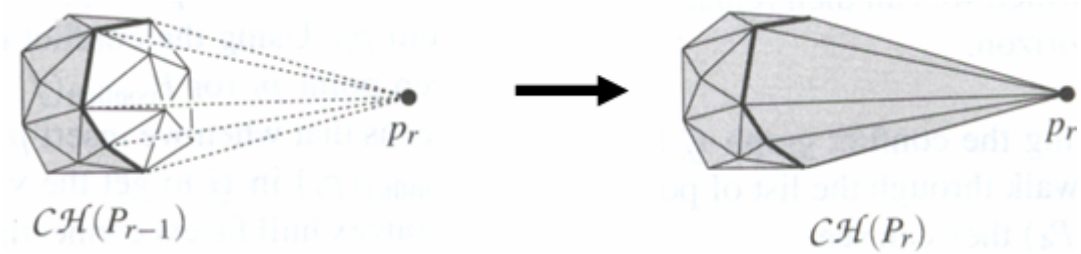
情况二：新点在当前凸包外部，需要计算并的凸包。在这种情况下，我们需要首先计算原凸包相对于p_r的**水平面(horizon)**，即p_r可以“看到”的封闭区域。



受到二维凸包的启发，可以把可见性转化为半平面的存在性。例如在下图中，f对于p来说是可见的，因为p和半平面h_f的“另一侧”。而q不可见，因为q和凸包在h_f的同侧。



这样，我们得到了情况二的计算方法：首先计算p_r的可见面并删除，然后求出对p_r的水平面，并把每条水平面上的边与p_r相连组成面。

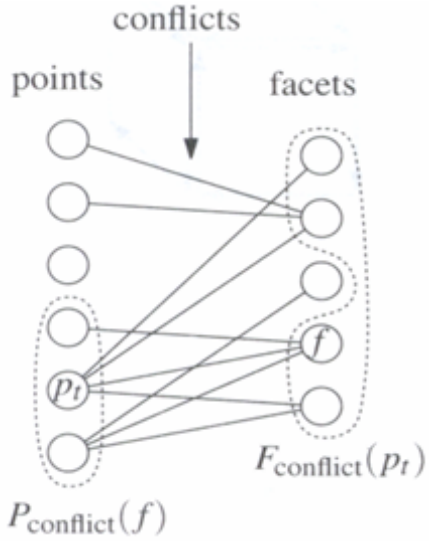


计算可见区域。由此可见，随机增量算法的关键在于计算新点的可见区域。最简单的方法是考虑所有有向面，判断p_r是否在其右侧，如果是，则该面可见。这样的方法最终将导致$O(n^2)$的总时间复杂度，不令人满意。一个更好的方法是维护一些信息，帮助更快的找到可见区域，这个信息就是**冲突集(conflict list)**。

当前面的冲突集：对于每个面f，它的冲突集$P_{conflict}(f)$是未来点集$\{p_{r+1}, p_{r+2}, \ldots, p_n\}$中能看见面f的点集。

未来点的冲突集：对于未来的每个点p_t(t¿r)，冲突集$F_{conflict}(p_t)$包含当前凸包$CH(p_r)$中对p_t可见的面集。

冲突集的含义是：p和f是冲突的，因为它们不能同时存在于新的凸包上！这样，我们很容易用一个称为**冲突图(conflict graph)**的二分图来表示冲突集，其中点都是还未插入的，而面是当前凸包$CH(p_r)$上的。每个元素（点或面）的冲突元素可以在线性时间内得到，如下图。

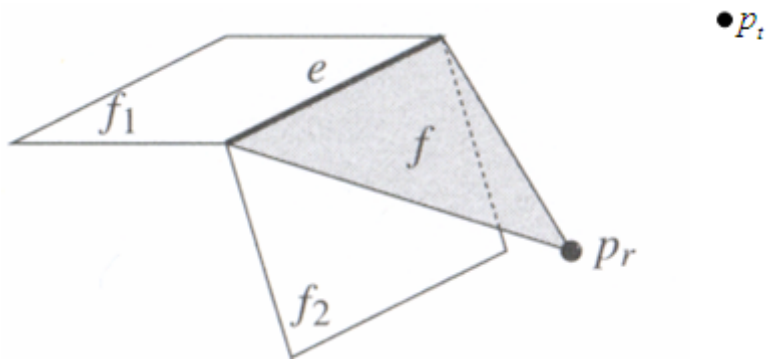


冲突图的初始化。首先计算出$CH(P_4)$，然后对于所有剩下的点，依次判断它们可以看到当前凸包仅有的四个面中的哪些。这一步是线性的。

冲突图的更新。每加入一个新点，需要更新冲突图，步骤如下：

第一步：删除p_r可见的面（即冲突图中p_r的邻居）。
第二步：删除p_r本身。
第三步：检测新的冲突。对于原来的面，冲突集显然不变（除了可能少了p_r之外）。对于新的面，需要计算未来的点中哪些能看见它。

最简单的方法仍然是依次判断所有未来点，但有更好的方法。



如上图，新面f一定和两个原来的面f_1和f_2邻接（其中一个还在而另一个已经被删除）。如果p_t可以看到f，则p_t一定可以看到e，因此一定能看到f_1或f_2。因此计算f的冲突集只需枚举f_1和f_2的冲突集中的点，而不需要考虑所有点。

这样，修改后的算法框架为：

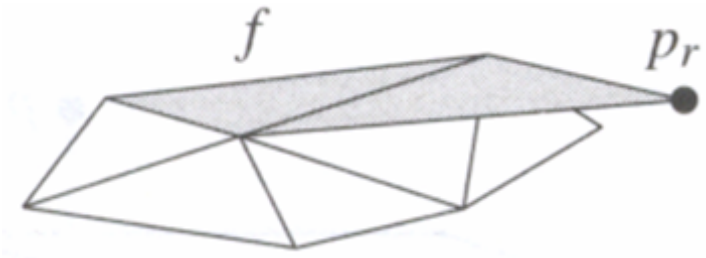
第一步：初始化CH(P_4)和冲突图G。
第二步：对于每个点p_r，

2.1 在凸包中删除与p_r冲突的面（根据G），同时找出水平面（即只有一个邻接面可见的边）并构造新面。

2.2 对于每个新面f，检查它所对应的水平面的边e邻接的两个面f_1和f_2的冲突集，确定f的冲突集并更新G。

2.3 在G中删除p_r的冲突集和p_r本身。

共面(coplanar)的情况。应作为f不对p_r可见，从而合并两个三角形，如下图。新面的冲突集和原来相同。



时间复杂度分析。首先，CH本身的复杂度不高。根据欧拉定理，面数是O(n)的。下面证明创建的面数期望值不超过6n-20。

初始化，面数为4。下面进行反向分析，从最后的完整凸包开始，统计被移除的面有多少。记$CH(p_r)$中和p_r邻接的边数为$\deg(p_r, CH(p_r))$。由于r结点的凸多面体边数不超过3r-6，因此度数的期望为

$$\begin{aligned} \mathrm{E}[\deg(p_r, \mathcal{CH}(P_r))] &= \frac{1}{r-4}\sum_{i=5}^{r}\deg(p_i, \mathcal{CH}(P_i \\ &\leqslant \frac{1}{r-4}\left(\left\{\sum_{i=1}^{r}\deg(p_i, \mathcal{C}\mathcal{H} \right.\right. \\ &\leqslant \frac{6r-12-12}{r-4} = 6. \end{aligned}$$

因此所有被创建的面的总和的期望为

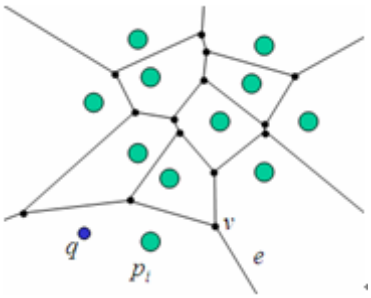
$$4+\sum_{r=5}^{n}\mathrm{E}[\deg(p_r, \mathcal{CH}(P_r))] \leqslant 4+6(n-4) = 6n-$$

问题的关键是寻找新冲突的时间。对于水平面上的每条边e，计算时间为O(|P(e)|)，其中$P(e) = P_{conflict}(f_1) \cup P_{conflict}(f_2)$，因此总时间复杂度为所有阶段中所有水平面边e的|P(e)|总和。可以证明，这个总和是O(nlogn)的，因此总时间复杂度为O(nlogn)。

8.3.2 二维Voronoi图

Voronoi图是三大几何结构之一，应用十分广泛，但构造算法相对比较复杂。

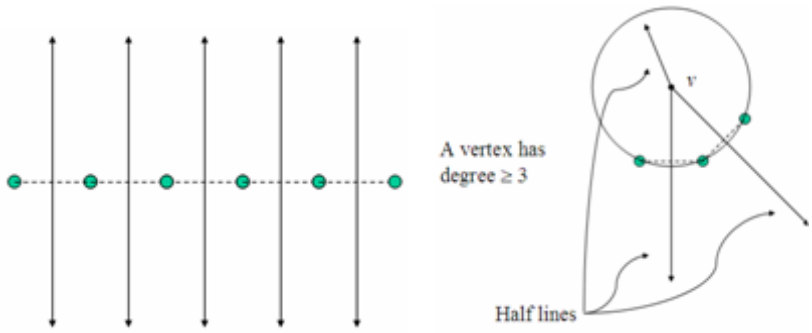
假设平面上有n个邮局。假设每个邮局都是一样的且服务能力无限大，每个人显然会去离他最近的邮局。每个邮局的服务范围是怎样的呢？如下图：



假设绿色点为邮局，则每个邮局的服务范围都是一个凸多边形（可能有些边是无界的）。任给一个点q，只需要确定它是哪个凸多边形内，就可以知道它离哪个邮局最近了。

Voronoi图。平面n个点的**Voronoi图**（**Voronoi diagram**）是平面的一个划分(subdivision)，它把整个平面分成n个区域(cell)，每个点有一个属于它的区域。点p_i所拥有的区域里的点q满足对于原点集的任意其他点p_j(i!=j)，有dist(q, p_i) ¡ dist(q, p_j)，其中dist表示欧几里得距离。

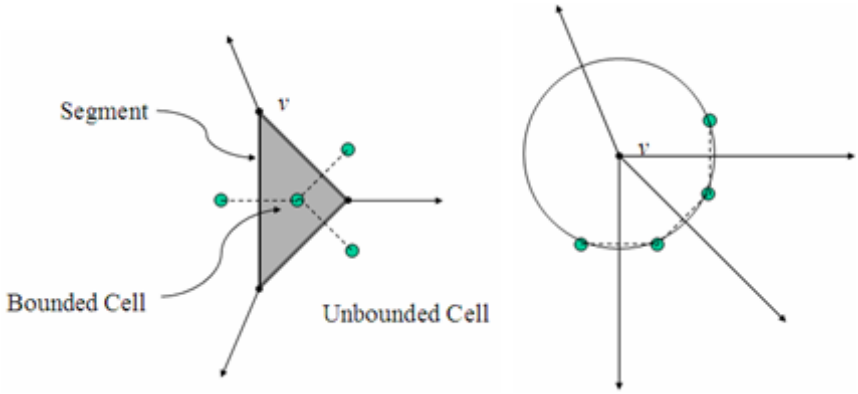
下面是共线点的Voronoi图。每条Voronoi边都是两个相邻边的中垂线。



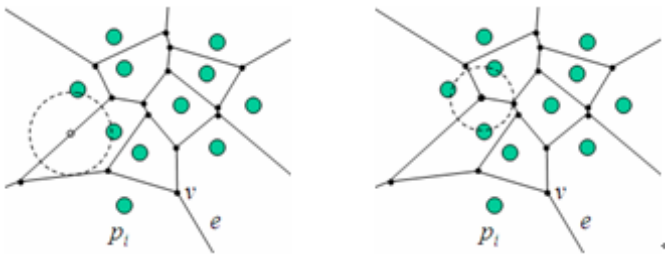
下面是三个不共线点的Voronoi图。其中**Voronoi结点(Voronoi vertex)**是过三点的圆的圆心，因为它到三个点的距离相等（定义）。

下图(a)有三个Voronoi结点和三条Voronoi边，围成了一个Voronoi三角形。容易看出：至少需要四个点才能得到一个有界的Voronoi多边形，而如果所有点共圆，再多的

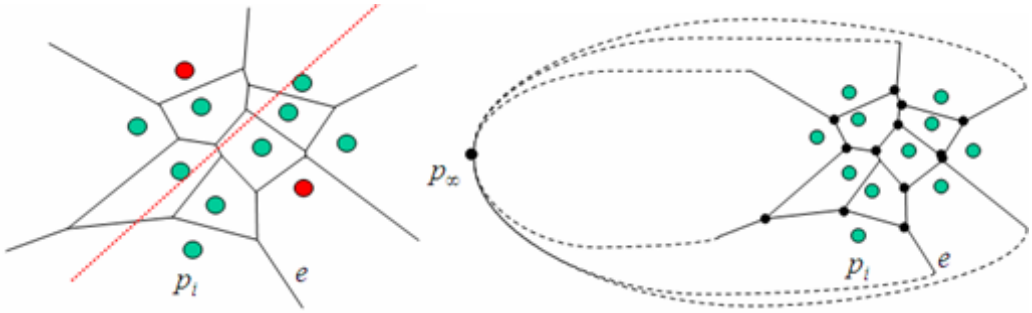
点也得不到有界Voronoi多边形，如图(b)



显然，每条Voronoi边到某两个原点的距离相等；每个Voronoi结点到某三个原点距离相等，如下图。



这是否意味着Voronoi边的个数应该是$C_n^2 = \Theta(n^2)$么？当然不是。虽然每条Voronoi边都到某两个原点的距离相等，但并不是每一对原点都是可以产生一条Voronoi边的，如下图。事实上，我们有Voronoi图是线性的，即|v|,|e|=O(n)。



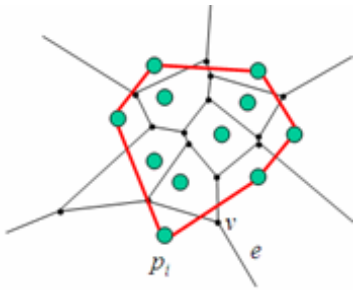
由于Voronoi图是一个平面图，我们想到用欧拉定理研究它的复杂度。为了用欧拉定理，我们需要创建一个虚拟结点p_∞，并把所有无限Voronoi边连接到此结点上。由于$|v| - |e| + |f| = 2$，且$f = n$，代入得（注意我们加了一个新结点）：

$$(|v|+1)-|e|+n=2$$

由于 $\sum_{v\in Vor(P)} \deg(v)=2\cdot|e|$且$\forall v\in Vor(P),\ \deg(v)\geq 3$（想一想，为什么），因此$2\cdot|e|\geq 3(|v|+1)$。代入上式得：

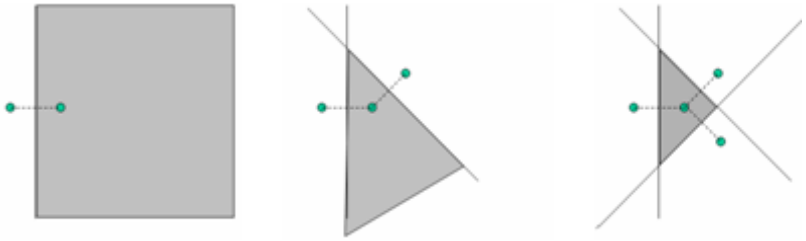
$$|v|\leq 2n-5$$
$$|e|\leq 3n-6$$

Voronoi图的一个有趣性质是：和无限边邻接的点组成了这n个点的凸包，如下图。



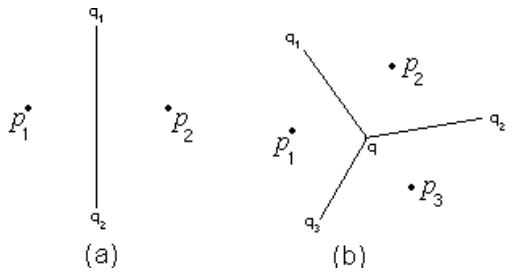
构造二维Voronoi图的算法很多，下面介绍三个。

增量算法。根据定义，很容易得到一个基于半平面交的算法计算Voronoi图。对于每个site，我们有如下的增量算法。



随着点的增多，该点的区域越来越小。每次相交是直线和凸多边形交，时间复杂度为O(n)，则n个半平面交总时间复杂度为$O(n^2)$，构造Voronoi图的总时间复杂度高达$O(n^3)$。前面讲过，n个平面的半平面交可以用分治思想在O(nlogn)内解决，这样构造Voronoi图的总时间复杂度降为了$O(n^2logn)$。

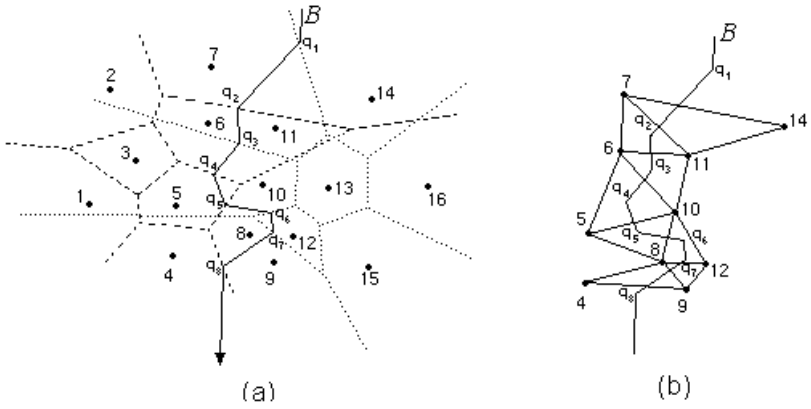
分治算法。如果n=2，则Voronoi图只是一条中垂线；如果n=3，则Voronoi图是三条与三点组成的三角形的外心相连的中垂线；如果n¿3，则将点分成尽量等量的两部分，分别求Voronoi图，然后合并。可以看出，算法的关键是合并过程。



假设现在要合并Vor(S_1)和Vor(S_2)，合并的关键是要构造一条折线B，下图(a)所示，折线将S_1, S_2分开，且折线恰为Vor($S_1 \cup S_2$)的边。删去Vor(S_1)中位于B右侧的边，删去Vor(S_2)中位于B左侧的边，就得到$S = S_1 \cup S_2$的Voronoi图。

由Voronoi图的性质，知道了Vor(S_1)和Vor(S_2)，就知道了CH(S_1)和CH(S_2)。在线性时间内可以求出CH(S_1)和CH(S_2)的正切线（正切线pq满足$p \in S_1$, $q \in S_2$，且pq∈CH($S_1 \cup S_2$)）。假设正切线为p_1p_2，$p_1 \in S_1$, $p_2 \in S_2$。

作p_1p_2的中垂线。设想由上向下沿该中垂线下移的z点遇到Vor(S_1)或Vor(S_2)的一条边，比如遇到Vor(S_2)的一条边，如下图(a)所示。图中点p_7和p_{14}分别属于S_1={p_1,p_2,...,p_8}和S_2={p_9,p中垂线首先与Vor(S_2)的边相交，即与$p_{11}p_{14}$的中垂线交于q_1，留下折线B的第一段(射线)。点q_1是$p_{14}p_{11}$的中垂线和$p_{14}p_7$的中垂线的交点，因此q_1是三角形$p_{14}p_7p_{11}$的外心，所以下一段折线为p_7p_{11}的中垂线上的一条线段，此时寻找p_7p_{11}的中垂线与p_7关联的Voronoi多边形的哪条边相交，图中示出p_7p_{11}的中垂线与p_7p_6的中垂线交于q_2。这样不断进行下去，直到找到q_8为p_8p_9的中垂线与p_8p_4的中垂线的交点。q_8向下的射线是p_9p_4的中垂线上的一条射线（注意p_9p_4是CH(S_1)和CH(S_2)的另一条正切线）。



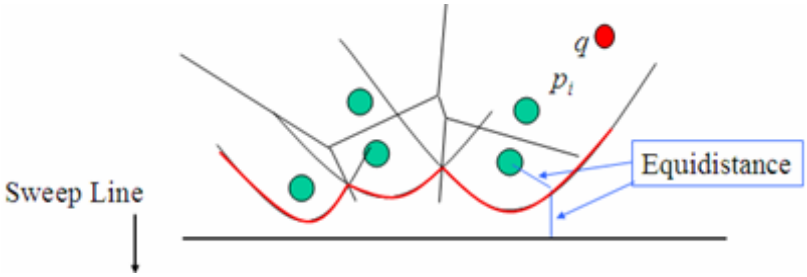
整个过程可以看成是三角形序列的演变过程，也就是$p_{14}p_7p_{11}$ → $p_7p_{11}p_6$ → $p_{11}p_6p_{10}$ → $p_6p_{10}p_5$ → $p_5p_{10}p_8$ → $p_{10}p_8p_{12}$ → $p_{12}p_8p_9$ → $p_8p_9p_4$，称为**三角形顶**

点转移法。

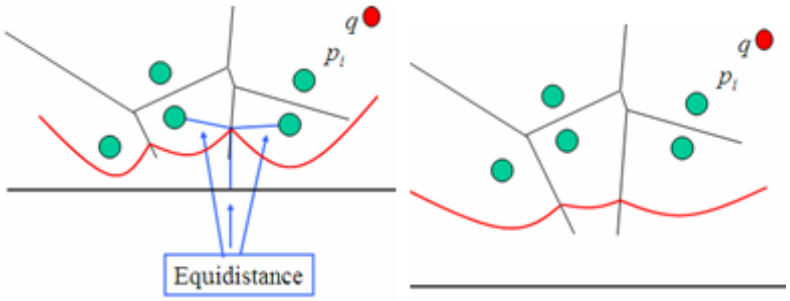
求正切线需要O(n)时间。折线B穿过Vor(S_1)与Vor(S_2)时，组成B的线段数目不超过$|S_1|+|S_2|=n$，与B相交的Voronoi边出超过O(n)，所以合并过程只要O(n)时间。求所有点的Voronoi图的分治算法的时间复杂度为O($n$log$n$)。

Fortune算法。和前两种算法不一样，Fortune算法是一种扫描算法，它在O(nlogn)的时间内构造出了Voronoi图。它的基本过程是把一条水平线从上到下移动，维护“不会被未来site所影响”的部分，并处理新遇到的site和Voronoi结点所带来的影响。

直观地讲，下图中红色线以上部分均是“不会再改变”的，因为其中的点到某个site的距离比扫描线近，因此也比扫描线下没有处理过的site近。



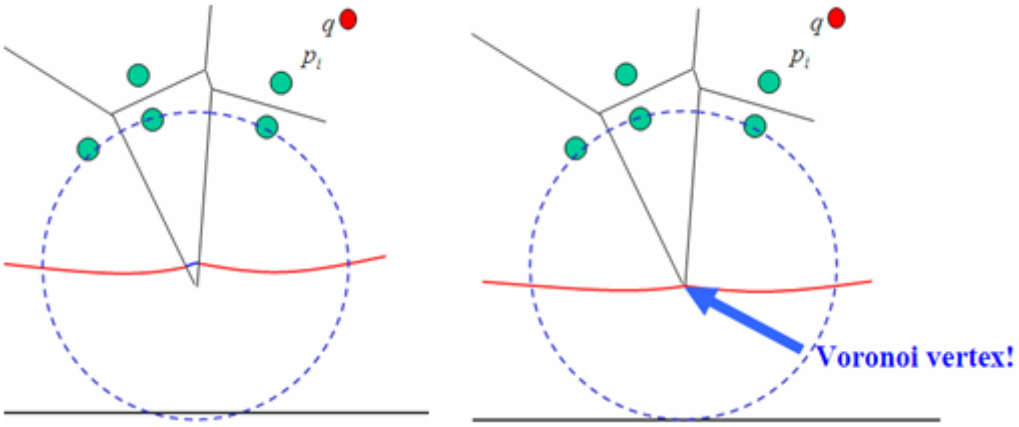
所有这样的抛物线弧形成了一个**海岸线(beach-line)**，它是不变部分的边界。抛物线之间的**断点(break points)**的轨迹就是Voronoi边（如图(a)），且这些弧越来越平滑（如图(b)）。



最后，海岸线的中间弧消失，算法找到了一个过三个（或更多点）的**空圆(empty circle)**（如图c），即找到了一个Voronoi结点。

海岸线的性质。我们不难得出海岸线和Voronoi边、结点之间的重要联系。

> *Voronoi*边：它们是扫描线下移时断点的轨迹。一个新断点的出现意味着一条新Voronoi边的出线，这可能是因为产生了一条新弧（site 事件）或合并



了两个已有的断点（circle事件）。

*Voronoi*结点：它们是当两个断点融合（即它们对应的两条Voronoi边相交）时的产物，也可以看作是一条旧的弧退化而成。

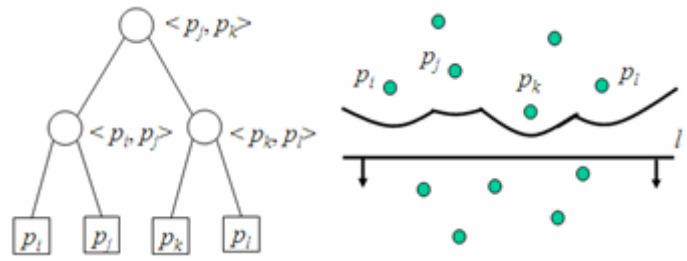
Fortune算法的数据结构。Fortune算法需要维护三大类的信息。

当前的*Voronoi*图：用DCEL储存，称为D。前面已经介绍过，这里不再叙述。

当前的海岸线：用一棵平衡二叉树T表示，内结点对应于断点，用二元组标识；叶子对应于一条和弧，用产生此弧的site标识。

当前扫描线：一个事件的优先队列Q，按y坐标递减顺序排列。

平衡二叉树T。如下图，内结点表示两条弧之间的断点，另外还需要包含一个指针指向此断点对应Voronoi边的D表项。叶结点表示弧。每条弧用产生它的site表示，还包含一个指针指向潜在的circle事件。



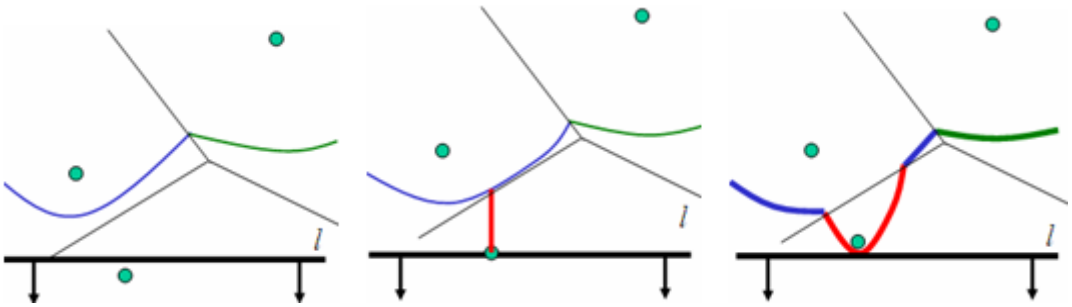
事件队列Q。事件(event)是扫描线从上到下扫描过程中遇到的特殊结点。扫描线只在这些特殊结点上停下来，因此它是离散的移动，而非连续移动。事件的优先级是y坐

标递减的顺序，因为扫描线总是先遇到y坐标比较大的事件，应当先被处理。事件有两种：

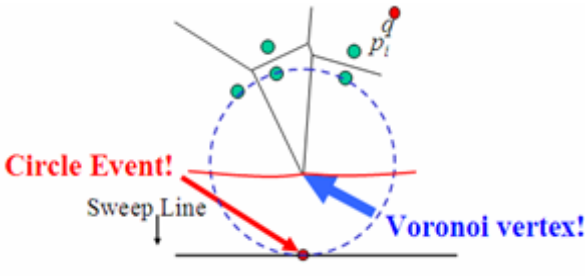
*Site*事件：扫描线碰到了一个新site。这些事件是静态的，不会修改。

*Circle*事件：扫描线到达一个穿过3个或更多site的空圆的.。这些事件的动态生成的，且可能被中途取消。

Site事件。如下图，新增一个site后此site正上方的弧会分裂。



Circle事件。当一个穿过3个或多个site的圆与扫描线相切时，一条弧会消失。扫描线可以帮助我们判断此圆是不是空圆。



Fortune算法框架。根据我们的讨论，Fortune算法框架如下图。其中问题的核心在于事件处理过程HandleEvent(e, T, D)，它处理事件e并更新数据结构T和D。

Site事件的处理。根据刚才的直观感觉，site事件的处理分为四个步骤。

步骤一：确定新site上方的弧（如果有的话）。

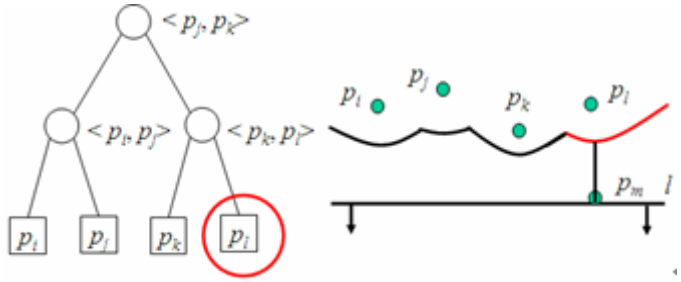
步骤二：分裂此弧，即把它对应的叶结点用一棵表示新弧及其断点的子树(sub tree)代替。

步骤三：在D中加入两条半边记录。

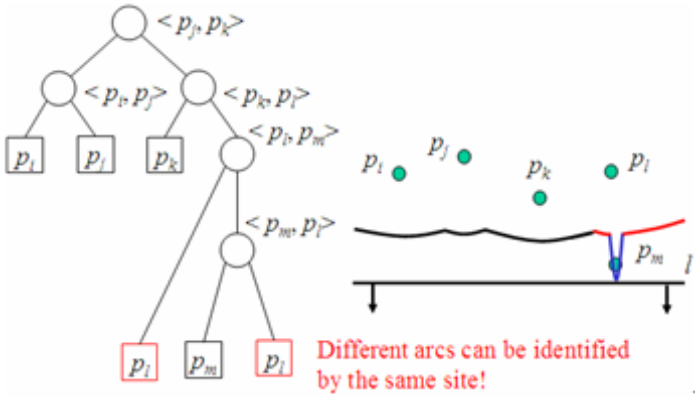
步骤四：检查可能的circle事件。如果存在，加入到事件队列Q中。

1. Initialize
   - Event queue Q ← all site events
   - Binary search tree T ← ∅
   - Doubly linked list D ← ∅
2. While Q not ∅,
   - Remove event (e) from Q with largest y-coordinate
     - HandleEvent(e, T, D)

步骤一可以通过二分查找完成。其中新结点的x坐标用来作为查找依据，而每个断点的x坐标总是现用现算。

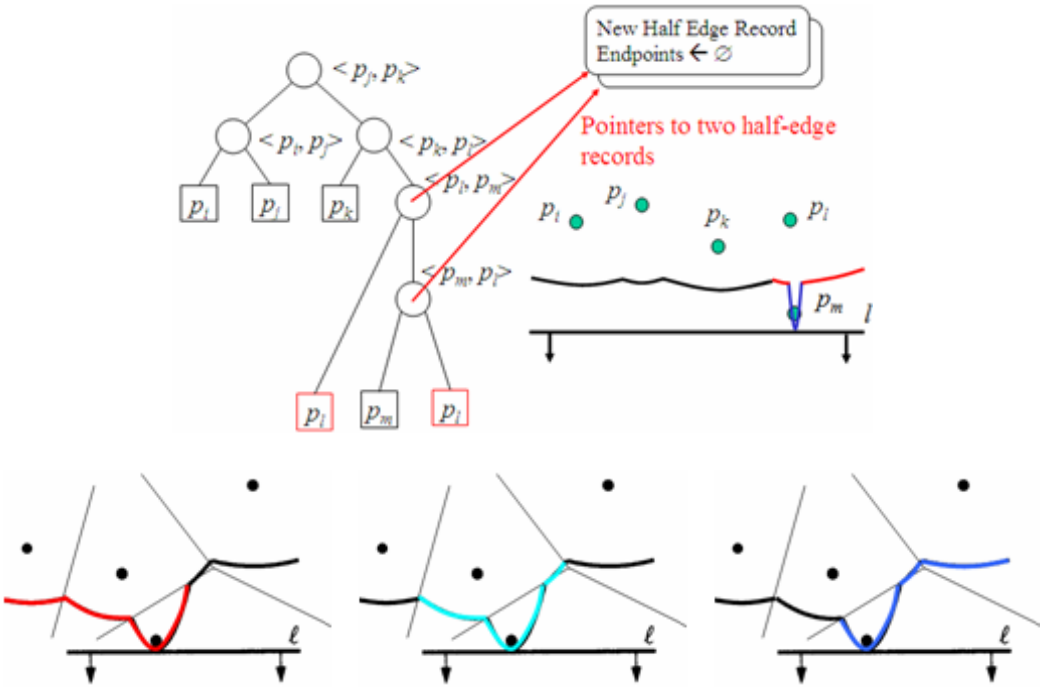


步骤二需要替换子树。注意到下图中p_m把p_l产生的弧分裂成了两断，因此p_l在T中出现了两次。

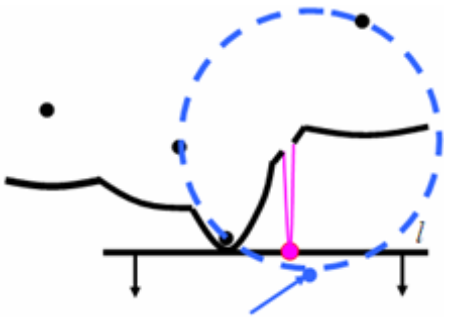


步骤三也并不困难，但需要注意此时半边的端点还不清楚。

步骤四需要依次考虑连续的三条弧，并检查断点是否汇聚。注意以新弧为中间弧一定不会出现汇聚。



但需要注意的是断点的汇聚并不一定引起circle事件而只能是“可能事件”，如下图，新site位于一个circle事件对于圆的内部，必须取消该事件。这种情况的处理可以在步骤一中“顺便进行”，即找到新site上方的弧时顺便把失效的circle事件删除。这要求在得到可能circle事件的同时给对应的弧所在的叶结点加一个指针指向该circle事件。



Circle事件的处理。Circle事件的处理类似，但理解起来困难一些。

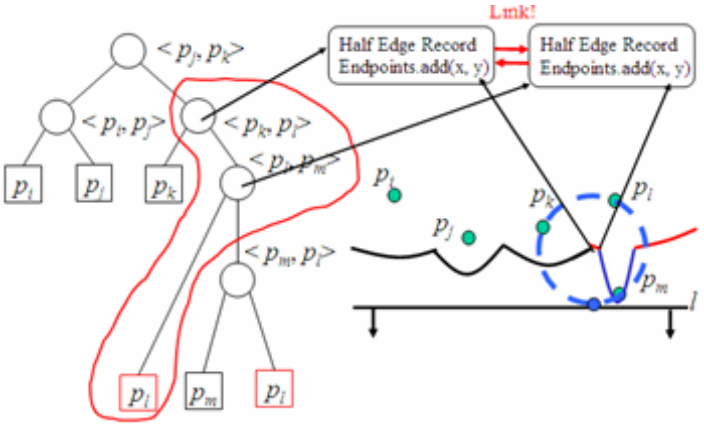
步骤一：给D中对应边增加结点。

步骤二：在T中删除消失弧对应的叶子，并删除它对应的circle事件。

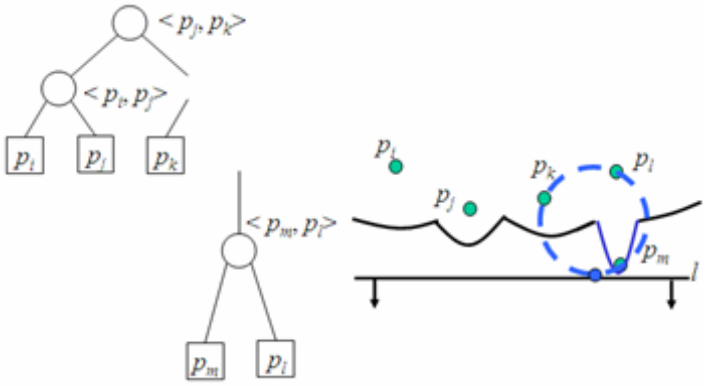
步骤三：在D中创建新的边记录。

步骤四：检查新的连续弧三元组以得到可能的新Circle事件。

步骤一如下图：



步骤二只是简单的删除子树。



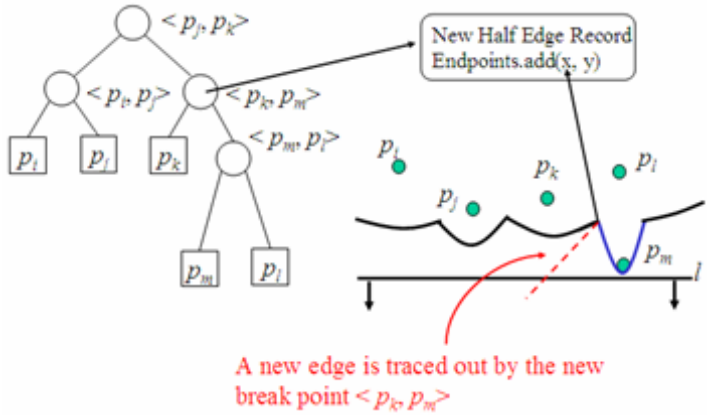
步骤三在刚才的基础上增加边记录。

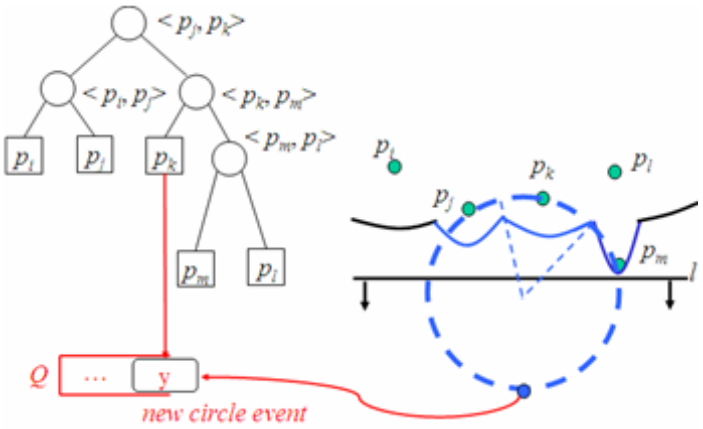
步骤四是检查新的三元组以发现新的circle事件。

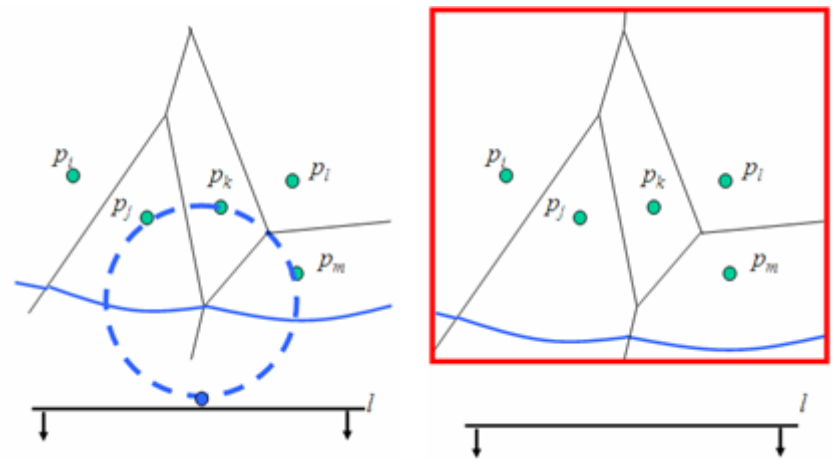
算法的终止。刚才的算法框架有一个小问题：Q为空时算法不应该终止，因为此时的海岸线将继续延伸出Voronoi边，需要用一个包围盒来终止这些“半无穷边”，如下图：

时间复杂度分析。下表列出了处理site事件和circle事件的四个步骤的时间复杂度。

Circle事件的四个步骤的时间复杂度为：





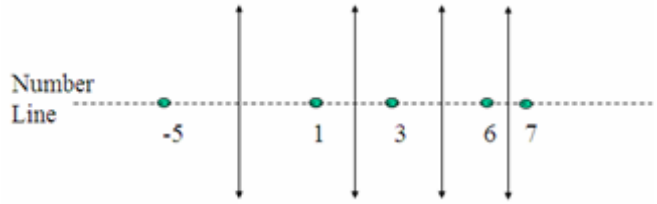


一个新site最多可以建立两条弧，因此海岸线最多包含2n-1条弧，因此Q中最多有O(n)个site事件和circle事件，而从Q中取时间的时间复杂度为O(logn)，因此总时间复杂度为O(nlogn)。

这个时间复杂度是最优的，因为可以用构造Voronoi图的方法给坐标进行排序。

	Site事件	时间复杂度
步骤一	确定新site上方的弧（如果有的话），并删除对应的circle事件	O(logn)
步骤二	分裂此弧，即把它对应的叶结点用一棵表示新弧及其断点的子树代替。	O(1)
步骤三	在D中加入两条半边记录。	O(1)
步骤四	检查可能的circle事件。如果存在，加入到事件队列Q中，并给对应的弧所在的叶结点加一个指针指向该circle事件。	O(1)

	Circle事件	时间复杂度
步骤一	给D中对应边增加结点	O(1)
步骤二	在T中删除消失弧对应的叶子，并删除它对应的circle事件	O(logn)
步骤三	在D中创建新的边记录	O(1)
步骤四	检查新的连续弧三元组以得到可能的新Circle事件。	O(1)



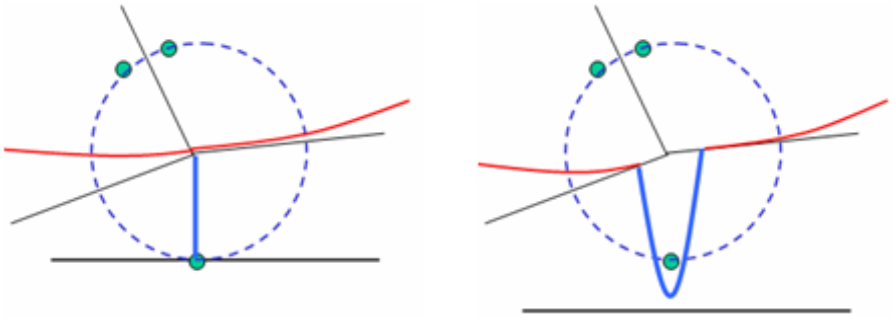
退化的情形。主要有以下情况的退化：

1. 不同事件的y坐标相同。这个可以再按照x坐标排序。
2. 过超过三个点的circle事件。当前的算法将生成多个重合的3度voronoi点，用零长弧连接。这可以通过后处理加以修正。
3. 共线的site。这将导致断点既不发散也不汇聚。最后的包围盒可以很好的处理这一情况。
4. site和circle事件重合。什么都不用做。

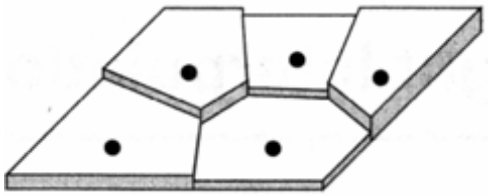
这样，我们在O(nlogn)时间内计算出了Voronoi图。

8.3.3 Delaunay三角剖分

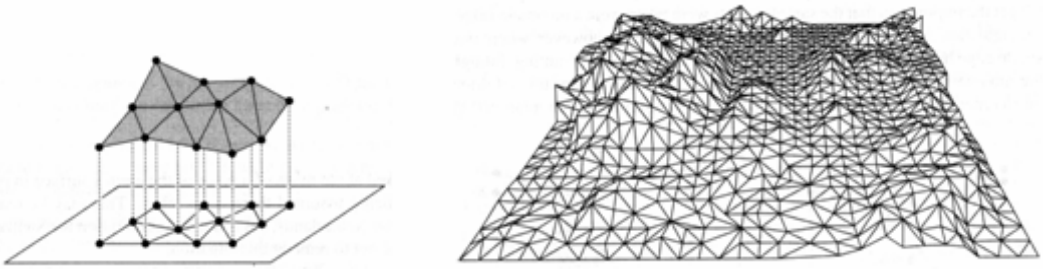
考虑如下的**领土问题(terrain problem)**：已知平面上n个点的高度f(p)，如何求出其他点的高度？



一个最直观的方法是先求Voronoi图，每个点离哪个点近就和它算作同一高度，如下图：



这个方法看起来不太对劲，应该真实的山应该比较平滑而不会这样突变。可以先求一个三角剖分，然后假设每个三角形都是平面，从而得到任意点的高度，如下图。



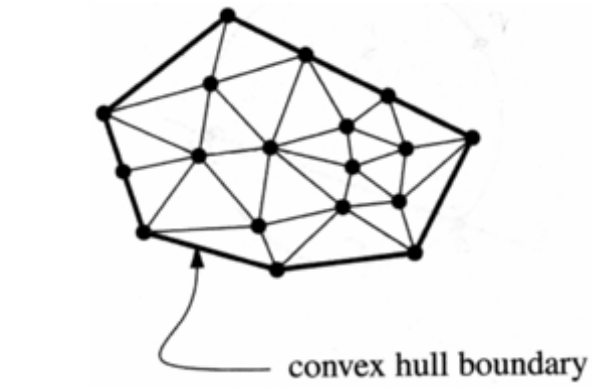
三角剖分。平面点集的**三角剖分(triangulation)**是以这些点为顶点的极大平面划分(maximal planar subdivision)，即任意连接两个顶点都将破坏划分的平面性。容易证明：

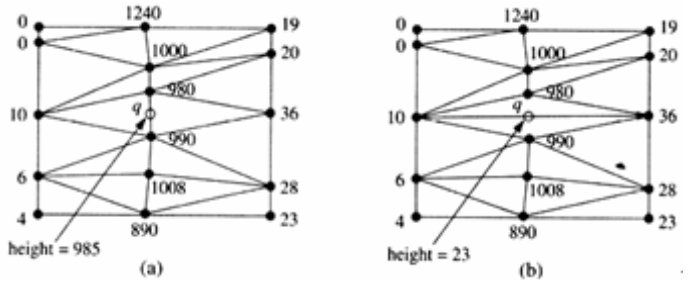
性质一、最外面的多边形是点集的凸包。

性质二、内部的面都是三角形。

性质三、设点集有n个点，凸包上有k个点，则任意三角剖分都有2n-2-k个三角形，3n-3-k条边。

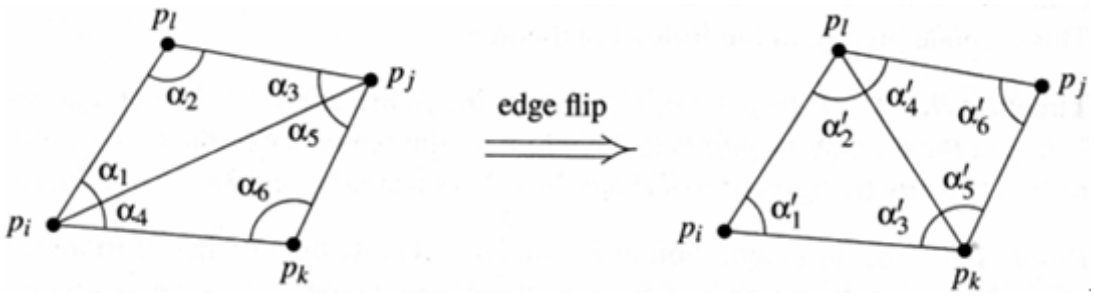
角度最优三角剖分。回到领土问题。有些三角剖分明显比其他的要“合理”，如下图。





因此我们应该避免“特别瘦小”的三角形，而应让最小角最大化。严格地说，我们对每个三角剖分T定义**角度向量(angle vector)**：A(T)=(α_1, α_2, α_3, ... α_{3m})，所有角度从小到大排序。角度向量可以按字典序排序，则最大的角度向量是最优的（最小角最大）。

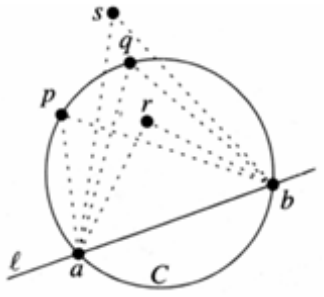
非法边和非法三角剖分。我们先从局部考察三角剖分。考虑一个凸四边形的三角剖分，定义一种**边翻转(edge flip)**操作，如下图：

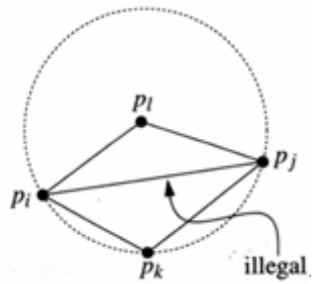


如果某个剖分的最小角小于另一个，称此对角线为**非法边(illegal edge)**。含有非法边的三角剖分很容易改进，称为**非法三角剖分(illegal triangulation)**。

用Thale定理可以在不计算角度的情况下判断一条边是否为非法边：对于一个圆C和圆上两点a, b，与圆外点（如s）形成的角度最小，与圆内点（如r）形成的角度

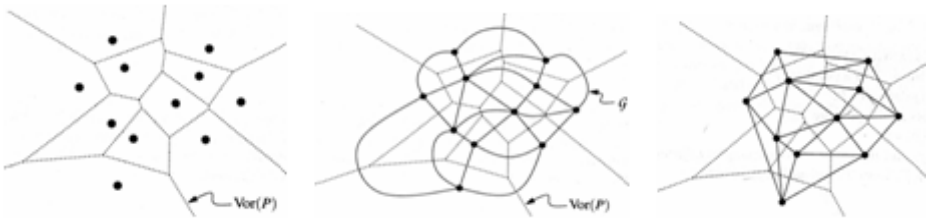
最大。在下图中有$\angle arb > \angle apb = \angle aqb > \angle asb$。





考虑不共圆的四边形$p_ip_jp_kp_l$，p_ip_j和p_lp_k中有且仅有一条是非法边。如图，p_ip_j是非法边的充要条件是p_l在圆C中。

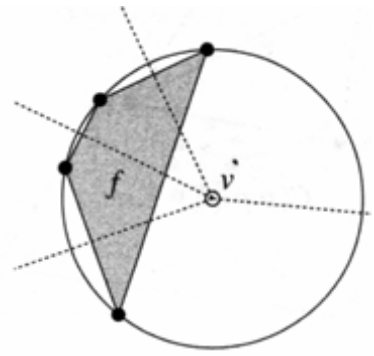
Delaunay图。点集P的Voronoi图Vor(P)的对偶图称为**Delaunay图**（**Delaunay Graph**）DG(P)，如下图。先给每个Voronoi多边形域创建一个结点，然后把有Δ>œ多边形域所对应的结点连起来，最后把这些连接弧改为直线段。



可以证明，Delaunay图是平面图。如果点集有多点共圆，那么将在Delaunay图上形成凸多边形而不是三角形，如下图。

这样的Delaunay图还可以进一步进行三角剖分，最终成为**Delaunay三角剖分**（**Delaunay Triangulation**）。如果这个Delaunay图本来就没有多点共圆的情况，则此图本身就是Delaunay三角剖分。

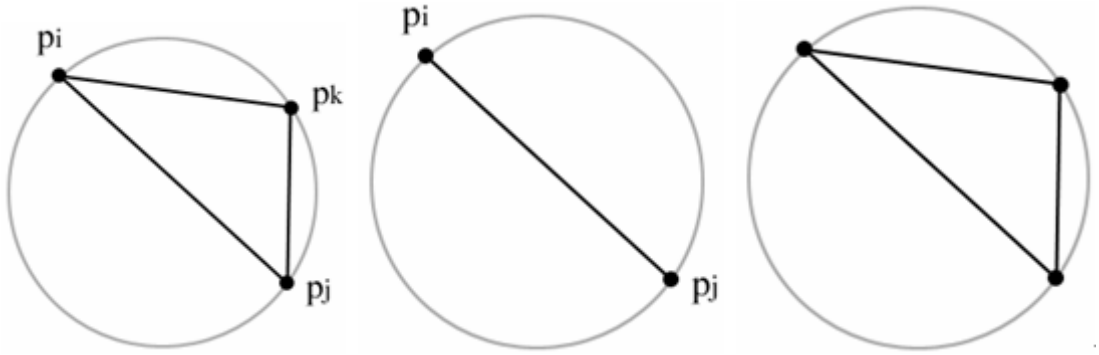
Delaunay三角形的性质。根据Voronoi图的性质，容易知道Delaunay三角形具有以下三个性质。



性质一：三个点$p_ip_jp_k$组成Delaunay三角形当且仅当过$p_ip_jp_k$的圆内不包含其他点。

性质二：两个点p_ip_j组成一条Delaunay边当且仅当¨3一个过p_ip_j的圆，不包含其他点。

性质三：P的三角剖分T是Delaunay三角剖分当且仅当它的每个三角形的外接圆均不包含其他点。



定理1：一个三角剖分是合法的当且仅当它是Delaunay三角剖分。

证明：用空圆性质和Thale定义可以得出所有DT都是合法的，而任意合法三角剖分都是DT比较复杂，这里略。

定理2：角度最优三角剖分一定是DT。

如果没有四点共圆，则DT唯一，它一定是角度最优的。如果DT不唯一，并不是所有DT都是角度最优的，但根据Thale定理，它们的最小角同时达到最大。对于领土问题来说，任意DT都是最优解。

Delaunay三角剖分的构造可以通过先求Voronoi图再取对偶的方法，但比较复杂，且不直接。事实上，可以直接采用随机增量法。

随机增量算法。本节我们讨论Delaunay三角剖分的随机增量算法。算法框架如下：

步骤一：初始化，构造一个“足够大”的包围三角形，包含P中的所有点。

步骤二：随机选取P中的一个点p_r。

步骤三：寻找包含p_r的三角形T。

步骤四：把T进一步剖分。

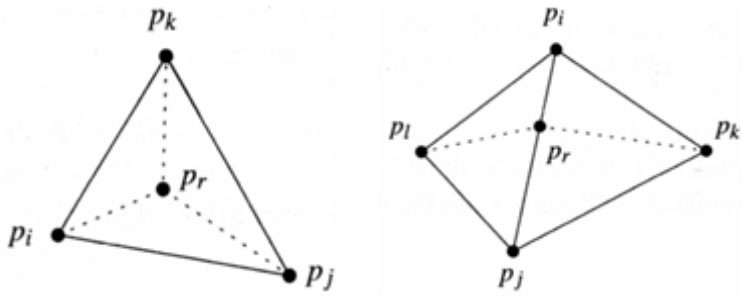
步骤五：翻转非法边，直到所有边均为合法。

步骤六：重复步骤二到步骤五，直到所有点均被加入。

剖分加细。考虑算法的步骤四，假设我们已经找到包含该点的三角形，则有两种情况：

情况一：p_r在某三角形的内部，如图(a)

情况二：p_r在两个三角形的公共边上。

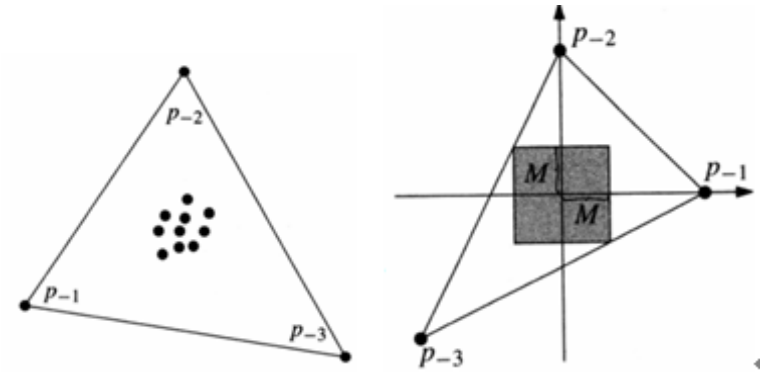


在划分加细前所有的边都是合法的，那么加细后呢？有哪些边可能变得不合法？一开始显然只有新点所在三角形的外边有可能变得非法，即图(a)中的p_ip_j, p_jp_k和p_kp_i，图(b)中的p_ip_l，p_lp_j，p_jp_k，p_kp_i。反过来，新增的边一定是合法的（想一想，为什么）。翻转非法边后可能会出现新的非法边，因此翻转过程应该是递归的，即：

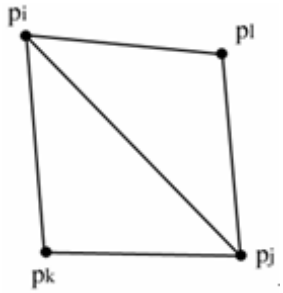
其中p_r是新增点，p_ip_j是待考虑的边（可能合法，也可能非法）。

初始三角形。注意到算法步骤的第一步需要一个“足够大的三角形”，如下图。但这样做可能需要很大的坐标。相反，我们修改对非法边的检测过程，让算法表现得好比真的存在这个足够大的三角形$p_{-1}p_{-2}p_{-3}$。

```
LEGALIZEEDGE(p_r, p_ip_j, T)
1.  if p_ip_j is illegal
2.    then Let p_ip_jp_l be the triangle adjacent to
      p_rp_ip_j along p_ip_j
3.         Replace p_ip_j with p_rp_l
4.         LEGALIZEEDGE(p_r, p_ip_l, T)
5.         LEGALIZEEDGE(p_r, p_lp_j, T)
```

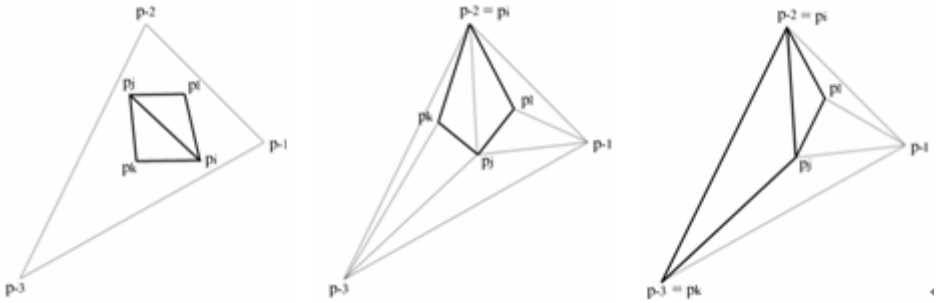


假设所有坐标的绝对值最大为M，令三个点的坐标为p_{-1}=(3M, 0), p_{-2}=(0, 3M), p_{-3}=(-3M, -3M)。假设待判断的边为p_ip_j，而另两个点为p_k和p_l，需要分四种情况讨论。



情况一：i和j都是负数。因此p_ip_j是包围边，它应该被保留，所以是合法的。

情况二：i, j, k, l都是正数，即通常情况。p_ip_j非法当且仅当p_l在$p_ip_jp_k$的外接圆内部。

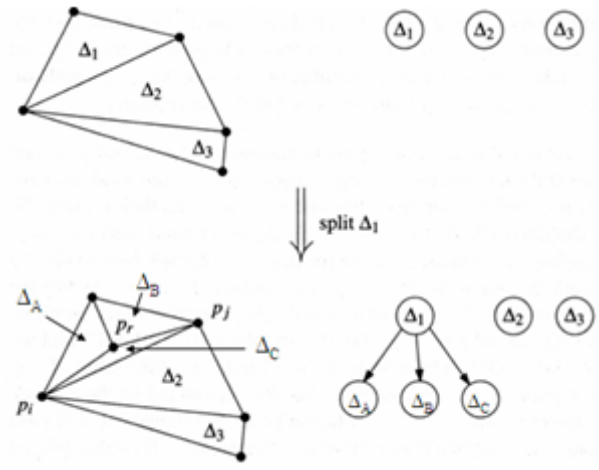


情况三：i, j, k, l中恰好有一个负数。由于我们不希望包围三角形破坏Delaunay三角剖分的边（即去掉包围三角形后剩下的Delaunay三角剖分应是合法的），因此当且仅当i或j为负时$\mathrm{p}_i\mathrm{p}_j$是非法的。

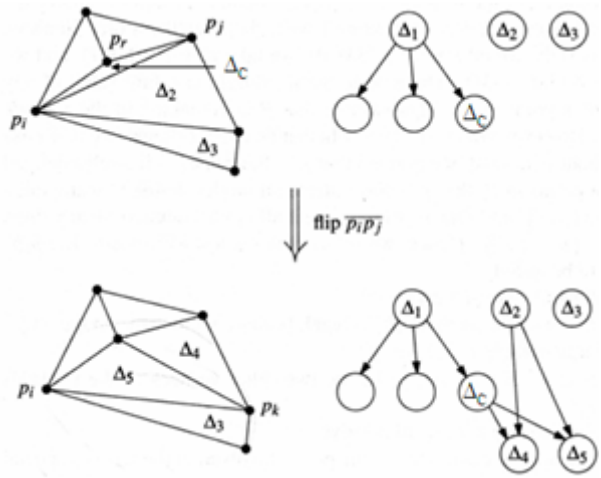
情况四：i, j, k, l中恰好有两个负数。这是最复杂的一种情况，首先应注意到k和l不能均为负（因为p_k和p_l中必须有一个p_r）。i和j也不能同时为负，因此i和j中恰好有一个为负，且k和l中恰好有一个为负。如果前者的负下标比后者小，则$\mathrm{p}_i\mathrm{p}_j$为合法，否则为非法。

点定位。算法的第三步需要找到新点所在的三角形。这一步需要一个复杂数据结构D来实现。D是一个有向无环图，D对应于当前三角剖分的三角形，维护D的叶子和三角剖分之间的交叉指针，初始化时只有一个叶子，代表包围三角形$\mathrm{p}_{-1}\mathrm{p}_{-2}\mathrm{p}_{-3}$。

当把三角形Δ_1分裂成两个（点在边上）或三个（点在三角形内）时，把这些子三角形作为Δ_1的叶子。



除了分裂操作外，边翻转时也需要更新D，即给两个新三角形创建叶子（Δ_4和Δ_5），并把它们作为被破坏的两个老三角形（Δ_C和Δ_2）的儿子。



在D中进行点定位非常方便：先设当前结点为根，然后每次走到包含它的儿子，指导到达叶子。复杂的分析指出所有查找操作的总时间复杂度的期望为O(nlogn)。下面是完整的算法框架。

1. Initialize triangulation T with helper bounding triangle. Initialize $\mathcal{D}$.
2. Randomly choose a point p_r from P.
3. Find the triangle Δ that p_r lies in using $\mathcal{D}$.
4. Subdivide Δ into smaller triangles that have p_r as a vertex. Update $\mathcal{D}$ accordingly.
5. Call LEGALIZEEDGE on all possibly illegal edges, using the modified test for illegal edges. Update $\mathcal{D}$ accordingly.
6. Repeat steps 2-5 until all points have been added to T.

由于用了随机，期望的时间复杂度为O(nlogn)，空间复杂度为O(n)。

8.3.4 直线的排列

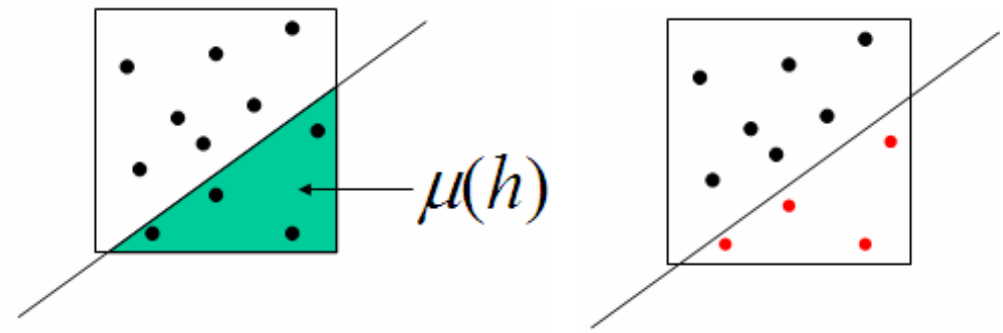
直线的排列是第三大几何结构，在计算几何中起着重要作用。

采样问题。考虑在一个单位正方形U上撒n个点，另有一个半平面h。则下图阴影部分的面积是h和单位正方形的交，即$\mu(h) = area(h \cap U)$。另一个计算方法是基于采样的：它等于区域内的点数与总点数的比，即$\mu_S(h) = card(S \cap h)/card(S)$。定义这个点分布对于h的Discrepancy为$\Delta_S(h) = |\mu(h) - \mu_S(h)|$，而对于所有半平面集合

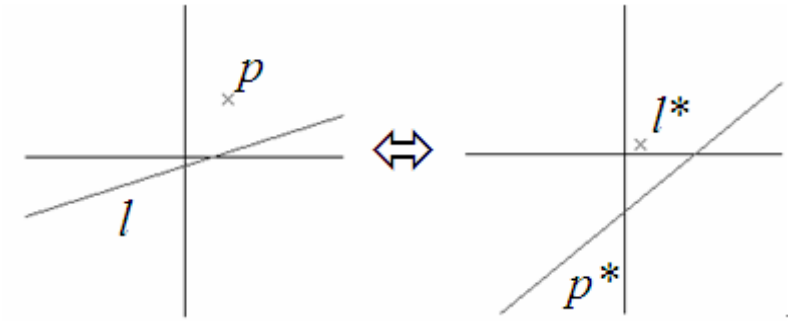
的Discrepancy为：

$$\Delta_H(S) = \max_{all\ h} \Delta_S(h)$$

如何求出点集的Discrepancy？显然我们无法枚举所有半平面，但容易发现使得最大值成立的半平面一定过其中一个点，因此只需要枚举O(n²)个半平面。问题转化为：给定一个半平面，有多少点在它的内部？

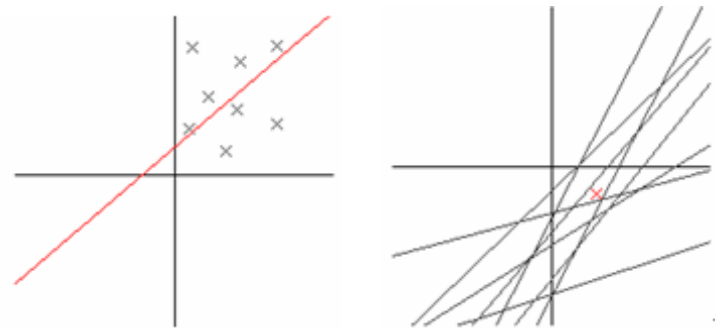


回忆前面介绍过的对偶变换。对偶变换是保序的，即若在主平面上点p在直线l上方，则在对偶平面上点l*在直线p*的上方。

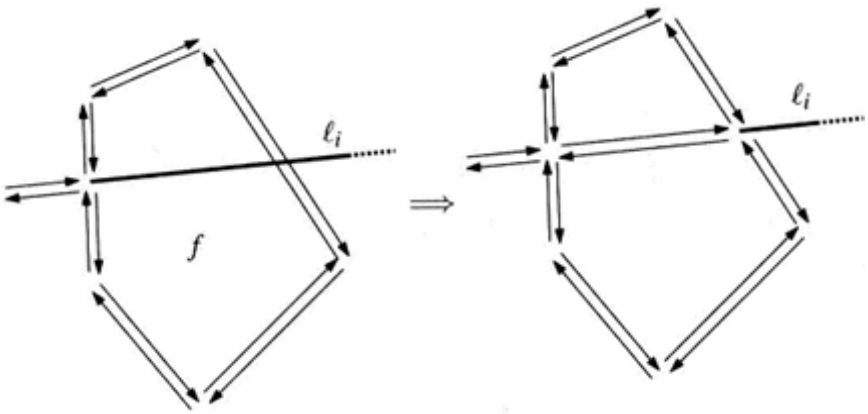


这样，原问题转化为了：判断对偶平面中给定点的上方有多少条线。对偶变换后，问题的难度没有改变，但更加直观，便于思考，如下图。

直线的排列。对偶变换把点集变成了直线集合，我们把这个直线集合对平面做的划分称为**直线的排列(arrangements of lines)**。我们用A(L)表示由直线集L导出的排列。容易证明顶点数不超过$\binom{n}{2}$，边不超过n^2，面不超过$\frac{n^2}{2}+\frac{n}{2}+1$，因此直线排列的组合复杂度为O(n²)。



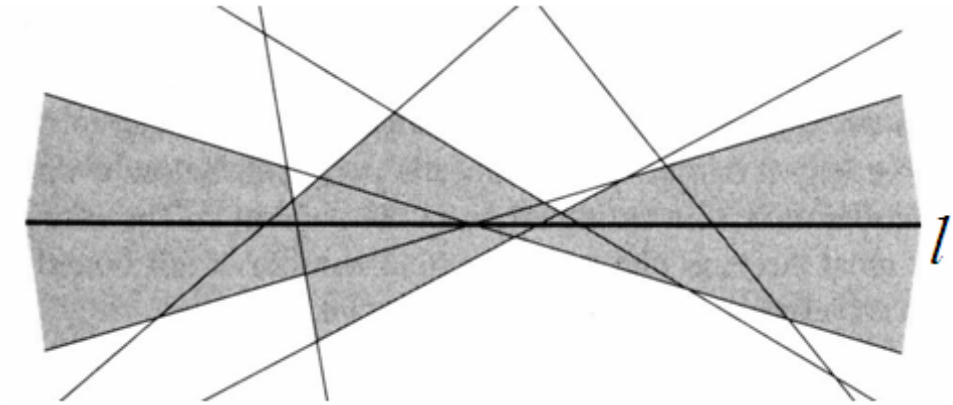
直线排列的增量算法。我们用前面介绍过的DCEL来表示直线的排列，则可以用增量算法构造直线的排列，每次需要计算所有和新直线相交的边，并做分裂，然后转移到twin(e)上，如下图。



下面给出构造直线排列的算法。

Input: A set L of n lines in the plane
Output: DCEL for the subdivision induced by the part of $A(L)$ inside a bounding box
1. Compute a bounding box $B(L)$ that contains all vertices of $A(L)$ in its interior
2. Construct the DCEL for the subdivision induced by $B(L)$
3. **for** i=1 to n **do**
4. Find the edge e on $B(L)$ that contains the leftmost intersection point of l_i and A_i
5. f = the bounded face incident to e
6. **while** f is not the face outside $B(L)$ **do**
7. Split f, and set f to be the next intersected face

区域复杂性。在一个直线的排列中，一条直线l的**区域(zone)**被定义为它穿过的所有面的并，如下图。



一个区域的**复杂性(complexity)**被定义为该区域中所有面的边数与点数之和。显然插入直线l_i的时间复杂度和$A(\{l_1,\ldots,l_{i-1}\})$中$l_i$的复杂度成正比。

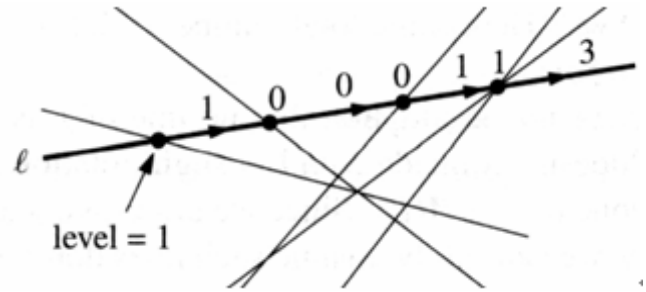
区域定理(Zone Theorm)：在一个由m条直线组成的排列中，任一条直线的区域复杂度为O(m)。

用数学归纳法加上细心的讨论，读者不难证明这个定理，这里略。作为定理的直接推论，刚刚给出的直线排列构造算法的时间复杂度为$O(n^2)$。

回到原问题。现在的任务是给出对偶平面中某两条直线的交点，统计该点上方上多少条直线。首先我们构造出S*的排列A(S*)，并称每个结点上方直线的数目为该结点的**层次(level)**。

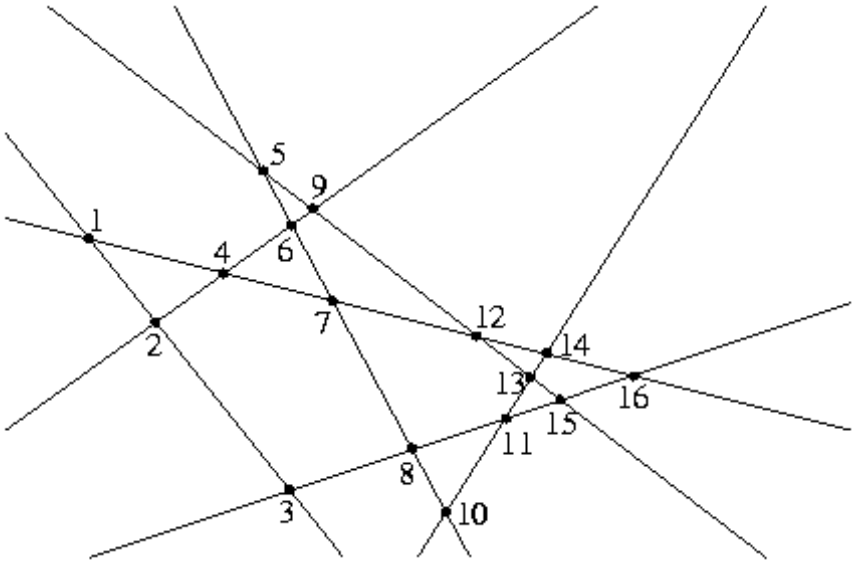
计算层次可以在增量式构造直线排列的过程中ˆB?1。对于每条新直线l，首先计算出它最左边的交点的层次。这一步可以在O(n)的时间内进行，只需依次判断其他直线是否在其上方即可。

下一步是根据DCEL从左到右沿着该直线走，遇到交点时修改计数器，如下图：

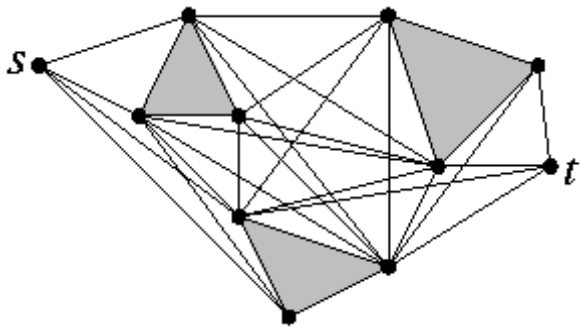


容易看出，处理每条直线的时间复杂度为仍为O(1)，没有为构造算法增加额外的时间。

扫描。有了直线的排列，一个重要的操作就是扫描。如果按照严格从左到右的顺序进行，时间复杂度和线段相交一样是$O(n^2 \log n)$的。但一般来说只需要保证每条线上的交点从左到右访问即可，不需要在整体上按顺序。在这种情况下，时间复杂度可以降到$O(n^2)$，它只需要简单的顺着排列走即可。



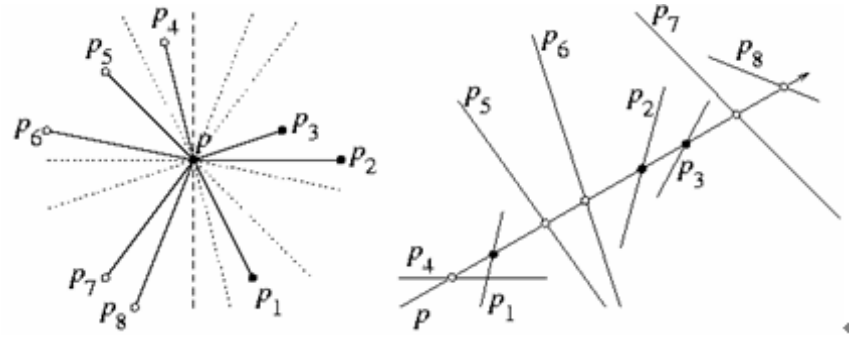
可视图的构造。利用排列还可以加速可视图(visibility)的构造。给出平面n条线段，可视图连接了所有可见的端点对，它在最坏情况下显然是$\Omega(n^2)$的。



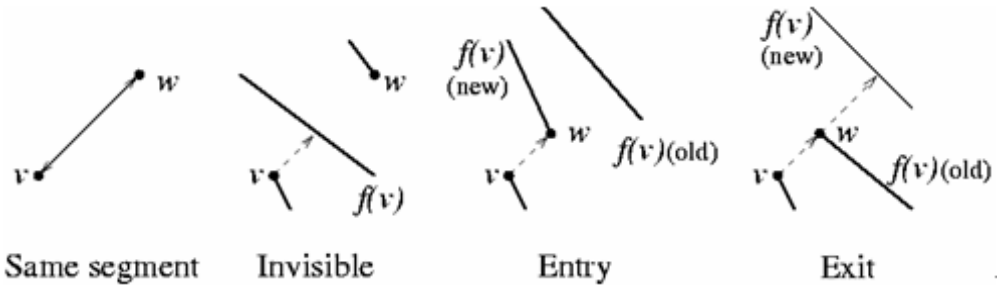
一个最简单的方法是依次判断每个端点对是否可见，时间复杂度为$O(n^3)$。一个较好的方法是依次求出每个端点能可视点集：把所有端点按照极角排序，然后按

照极角序扫描。极角排序过程是O(nlogn)的，扫描也是O(nlogn)的，因此时间复杂度为$O(n^2logn)$。

利用排列可以把时间复杂度进一步降到$O(n^2)$。考虑点p和其他端点$p_1, p_2, ..., p_8$，它们的对偶线如图(b)所示。则p^*与其他线的交点按照斜率排序。如果把按斜率正负分成两个集合，则可以由p^*上交点的顺序直接得到两个集合中点的顺序，合并即可得到完整的顺序。只要把所有点的对偶直线组成的排列求出，就可以在$O(n^2)$时间内求出围绕每个点的极角序。



我们同时围绕所有点进行扫描，并给每个点设置当前子弹射到的线段编号f(v)。所有事件一共只有n^2个（每个端点对一个事件），可以通过遍历排列而一次性求出。



每个事件有四种情况。

相同线段*(Same segment)*：设w为真。

不可见*(Invisible)*：如果f(v)档在前面，设w不可见。

进入*(Entry)*：w可见且是右端点，置f(v)=w。

离开*(Exit)*：w可见且是左端点，重设f(v)。注意到我们同时也在扫描w，容易知道f(v)=f(w)。

有读者可能会怀疑到这个算法的正确性：排列上的遍历只能保证每条直线上的点按顺序走过，但并不保证整体的顺序。在Exit事件中v处子弹的角度并不等于w处的角度！的确如此，但这个角度差异对算法的正确性没有影响，有兴趣的读者可以自己证明。

8.3.5 小结与应用举例

本节介绍几何结构及其应用。

第一节介绍凸包，包括凸性、二维凸包和三维凸包的组合复杂性与构造算法。二维凸包的Graham算法是最广为流传的计算几何算法之一，这里我们给出的是Andrew于1979年提出的修改版本，概念和实现都更为简单，且对退化的处理更好，算法也更为鲁棒。三维凸包的随机增量算法是本节的重点。在介绍这个算法之前，我们首先给出了三维凸包的两个简单算法：即$O(n^4)$的枚举算法和$O(n^2)$的卷包裹法。增量算法的核心是计算不在旧凸包的新点的可见区域，我们提出了冲突集方法，即分别考虑当前面和未来点的冲突集，并用二分图来表示。冲突集的快速更新需要一个结论：即新面的冲突集只包括两个老面中的元素。如果各个点按随机顺序加入，可以证明时间复杂度的期望是O(nlogn)的。

第二节介绍Voronoi图，包括图的组合复杂性和基本几何性质。二维凸包是一个凸多边形，可以简单的用数组或链表表示，而Voronoi图是一个平面剖分，因此需要用DCEL储存。构造Voronoi图的方法有很多，本节首先介绍了基于半平面交的$O(n^2logn)$时间算法，然后简要介绍了分治算法。虽然分治算法的时间复杂度为O(nlogn)，但是难以理解，且容易编错。本节的重点是Fortune算法，它的概念清晰，且很有启发性。作为平面扫描算法，它的特别之处是Circle事件不仅动态生成，而且可能提前取消。简单的说，Fortune算法除了保存扫描线状态外，还需要维持一个称为海岸线的特殊数据结构，它由一些抛物线片段组成。这些断点的轨迹形成Voronoi边，而两条Voronoi边相交时（同时一段弧消失）形成一个Voronoi点。算法的核心是Site事件和Circle事件的处理，往往涉及到海岸线平衡树上的操作及新事件的检测。这部分算法比较罗嗦但很直观，最后需要注意算法的终止部分。

第三节讨论Delaunay三角剖分。本节首先从领土问题中引入点集三角剖分问题，并且直观阐述了为什么需要让极小角最大化并据此定义了边翻转操作和非法边、非法剖分的概念。这些概念是Delaunay三角剖分的增量算法的核心。Delaunay三角剖分

是从Delaunay图而来，但由于点集可能存在四点共圆的情况，所以Delaunay图还需要进行进一步的三角剖分。接下来给出了Delaunay三角剖分的一些几何性质，并证明了DT和角度最优剖分的关系。本节的重点是Delaunay三角剖分的增量算法。算法有三个关键点，一是初始三角形的选取（比较罗嗦但代码简单），二是点定位问题（很像梯形剖分中的数据结构），三是剖分加细及非法边的翻转（递归翻转）。

第四节讨论直线的排列。本节的重点有两个，一是直线排列的增量算法，二是对偶法。本节从采样问题开始讨论，用对偶法把问题转化为了点层次的计算问题。直线排列的构造算法十分简单，其复杂度证明需要用到区域定理。由于排列的组合复杂度是平方的，因此本节介绍的直线排列构造法是渐进最优的。本节最后给出了一个可视图构造的例子，利用对偶法和直线排列把时间复杂度从原始的$O(n^3)$降到$O(n^2)$。由于可视图的组合复杂度是平方的，因此在输出不敏感的算法中，本节给出的算法渐进最优。

重要算法和程序列表：

算法	程序	备注
修改的Graham算法	andrew.cpp	Andew提出的基于水平序的Graham算法，时间复杂度为O(nlogn)
三维凸包的基本算法	3dhull_basic.cpp	$O(n^4)$枚举算法和$O(n^2)$卷包裹算法。
三维凸包的随机增量算法	3dhull_incr.cpp	期望O(nlogn)的随机增量算法
Voronoi图的基本算法	voronoi_basic.cpp	基本$O(n^3)$算法和基于快速半平面交的$O(n^2 \log n)$算法。
Voronoi图的Fortune算法	fortune.cpp	基于平面扫描的Fortune算法。
DT的随机增量算法	dt_incr.cpp	二维Delaunay三角剖分的O(nlogn)随机增量算法。
直线排列的增量算法	arrangement.cpp	直线排列的$O(n^2)$算法。
可视图构造的基本算法	vis_basic.cpp	可视图构造的枚举算法$O(n^3)$和角度扫描算法$O(n^2 \log n)$
可视图构造的快速算法	vis_quick.cpp	利用对偶和直线排列的可视图构造算法，时间复杂度$O(n^2)$。