USACO 教程

目录

Section 1.2　Complete Search 枚举搜索

思想：

写枚举搜索时应遵循 KISS 原则（Keep it simple stupid，译为“写最单纯愚蠢的程序”，意思是应把程序写得尽量简洁），竞赛时写程序的最终目标就是在限制时间内求出解，而不需太在意否还有更快的算法。

枚举搜索具有强大的力量，他用直接面向答案并尝试所有方案的方法发现答案。这种算法几乎总是解题时你第一个想到的方法。如果它能在规定的时间与空间限制内找出解，那么它常常很容易编写与调试。这就意味着你可以有时间去解答其他难题，即那些不能显示枚举算法强大的题目。

如果你面对一道可能状态小于两百万的题目，那么你可以考虑使用枚举搜索。尝试所有的状态，看看它们是否可行。

小心！小心！

有时，题目不会直接要求你使用枚举算法。

例题 1：派对灯 [IOI 98]

在一次 IOI 派对上有 N 个灯和 4 个灯开光，第一个开关可以使所有灯改变状态（关上开着的灯，开启关着的灯），第二个开关可以改变所有偶数位上灯的状态，第三个开关可以改变所有奇数位上灯的状态，第四个开关控制着灯 1、4、7、10……（3n+1）。

告诉你 N 的值和所有按开关的次数（小于 10,000 次）。并告诉你某些灯的状态（例如：7 号灯是关着的，10 号灯是开着的）请编程输出所有灯可能的最后状态。

很明显，每次按开关你都要偿试 4 种可能。那么总共要试 410000 次（大约 106020），那意味着你没有足够的时间去使用枚举搜索，但是我们将枚举方法改进一下，那就可以使用枚举了。因为无论有多少个灯，由于开关控制的特殊性，都会出现 6 个灯一次循环的情况，即 1 号灯的状态永远与 7 号灯，13 号灯，19 号灯……相同，2 号灯的状态也永远与 8 号灯，14 号灯，20 号灯……相同。同样，无论你按了多少次开关，按同一个开关两次就相当于没有按该开关，那么每一个开关就只需要考虑按一次或没有按，那么这题的枚举量就很小了。

例题 2: 时钟调整 [IOI 94]

有九个钟被摆放在一个 3 X 3 的矩阵中，它们各自指向 12：00，9：00，6：00，3：00 中的一种，你的目的是将它们的指针全部调向 12：00。很遗憾，每一次调整你都只能从九种调整方案中选择一种执行（九种方案已被从 1 到 9 编号），每一种方案可以改变固定钟的状态（例如：方案 1 控制钟 1，2，3，方案 2 控制钟 1，4，7，方案 3 控制钟 5，6，8，9……），即将方案指定的所有钟向前拨快 3 小时（使时针向顺时针方向旋转 90 度），请你输出一个数列，使得按该数列表示方案执行后，所有钟都指向 12：00。并且如果把整个序列看作一个数，要求该数最小。

最容易想到的方法是用递归枚举 1 到 9 的方案在该步使用。很可怕，由于递归的层数在此没有限定，所以将用掉 9k 的时间（k 为层数），那可能是相当巨大的。其实，不用紧张，细心的你一定会发现：当一个方案执行 4 次后，就相当于没有执行，又因为题目要求输出最优解，那么任意一个方案都没有必要 4 次以上执行。也就是说，只需要枚举每一个方案的 4 种情况（没执行，执行 1，2，3 次）就可以了。他仅有 49 ，约 262,072 此枚举，我们的计算机 1s 内就可以算完，应该算是极快了。

类似问题：

挤牛奶 [USACO 1996 初赛]

给出一个挤牛奶的顺序（农夫 A 在 300 秒到 1000 秒时挤牛奶, 农夫 B 从 700 秒到 1200 秒），要求输出：

最长的有人挤牛奶的时间。

最长的没人挤牛奶的时间。

完全数牛与完全数牛群 [USACO 1995 决赛]

如果一个数可以由它的某几个约数相加得到，那么我们叫它完全数，如 28 = 1 + 2 + 4 + 7 + 14。而一对完全

对数就是指两个数都可以由对方的约数相加得出。同样一个完全数组就是一个数组的第一个数可以由第二个数的约数加和得到，第二个数也可以由第三个数的约数相加得到……最后一个数可以由第一个数的约数加和得到。

现在 Farmer John 已经将它的牛儿们编好了号（1 到 32000）请找出其中所有的完全数牛与完全数牛群。

Section 1.3 Greedy Algorithm 贪心算法

样例：牛棚修理 [1999 USACO 春季公开赛]

Farmer John 有一列牛棚，在一次暴风中，牛棚的一整面墙都被吹倒了，但还好不是每一间牛棚都有牛。Farmer John 决定卖木料来修理牛棚，然而，刻薄的木材提供商却只能提供有限块的木料（木料的长度不限），现在告诉你关着牛的牛棚号，和提供的木材个数 N，你的任务是编程求出最小的木块长度和。（1 <= N <= 50）

贪心思想：

贪心思想的本质是每次都形成局部最优解，换一种方法说，就是每次都处理出一个最好的方案。例如：在样例中，若已经发现 N = 5 时的最优解，那么我们可以直接利用 N = 5 的最优解构成 N = 4 的最优解，而不用去考虑那些 N = 4 时的其他非最优解。

贪心算法的最大特点就是快。通常，二次方级的存储要浪费额外的空间，而且很不幸，那些空间经常得不出正解。但是，当使用贪心算法时，这些空间可以帮助算法更容易实现且更快执行。

贪心的难点：

贪心算法有两大难点：

① 如何贪心：

怎样才能从众多可行解中找到最优解呢？其实，大部分都是有规律的。在样例中，贪心就有很明显的规律。但你得到了 N = 5 时的最优解后，你只需要在已用上的 5 块木板中寻找最靠近的两块，然后贴上中间的几个牛棚，使两块木板变成一块。这样生成的 N = 4 的解必定最优。因为这样木板的浪费最少。同样，其他的贪心题也会有这样的性质。正因为贪心有如此性质，它才能比其他算法要快。

② 贪心的正确性：

要证明贪心性质的正确性，才是贪心算法的真正挑战，因为并不是每次局部最优解都会与整体最优解之间有联系，往往靠贪心生成的解不是最优解。这样，贪心性质的证明就成了贪心算法正确的关键。一个你想出的贪心性质也许是错的，即使它在大部分数据中都是可行的，你必须考虑到所有可能出现的特殊情况，并证明你的贪心性质在这些特殊情况中仍然正确。这样经过千锤百炼的性质才能构成一个正确的贪心。

在样例中，我们的贪心性质是正确的。如下：

假设我们的答案盖住了较大的空牛棚连续列，而不是较小的。那么我们把那部分盖空牛棚的木板锯下来，用来把较小的空牛棚连续列盖住，还会有剩余。那么锯掉它们！还给木材商！同时我们的解也变小了。也就是说，我们获得更优的解。所以，靠盖住较大空牛棚连续列的方法无法获得最优解，我们也应该尽量贪心那些距离小的木板合并。

如果仍有一个空牛棚连续列与我们的答案盖住的那个相同，我们同样使用上述的方法。会发现获得的新解与原解相同，那么不论我们选哪个，结果都将一样。

由此可见，如果我们合并的两块木板间距离最短，那么总能获得最优解。所以，在解题的每一步中，我们都只需要寻找两块距离最小的木板并合并它们。这样，我们获得的解必定最优。

结论：

如果有贪心性质存在，那么一定要采用！因为它容易编写，容易调试，速度极快，并且节约空间。几乎可以说，它是所有算法中最好的。但是应该注意，别陷入证明不正确贪心性质的泥塘中无法自拔，因为贪心算法的适用范围并不大，而且有一部分极难证明，若是没有把握，最好还是不要冒险，因为还有其他算法会比它要保险。

类似问题：

三值排序问题 [IOI 1996]

有一个由 N 个数值均为 1、2 或 3 的数构成的序列（N<= 1000），其值无序，现要求你用最少的交换次数将序列按升序顺序排列。

算法：排序后的序列分为三个部分：排序后应存储 1 的部分，排序后应存储 2 的部分和排序后应存储 3 的部分，贪心排序法应交换尽量多的交换后位置正确的（2，1）、（3，1）和（3，2）数对。当这些数对交换完毕后，再交换进行两次交换后位置正确的（1，2，3）三个数。

分析：很明显，每一次交换都可以改变两个数的位置，若经过一次交换以后，两个数的位置都由错误变为了正确，那么它必定最优。同时我们还可发现，经过两次交换后，我们可以随意改变 3 个数的位置。那么如果存在三个数恰好为 1，2 和 3，且位置都是错误的，那么进行两次交换使它们位置正确也必定最优。有由于该题具有最优子结构性质，我们的贪心算法成立。

货币系统 -- 一个反例 [已删节]

奶牛王国刚刚独立，王国中的奶牛们要求设立一个货币系统，使得这个货币系统最好。现在告诉你一个货币系统所包含的货币面额种类（假设全为硬币）以及所需要找的钱的大小，请给出用该货币系统找出该钱数，并且要求硬币数尽量少。

算法：每次都选择面额不超过剩余钱数但却最大的一枚硬币。例如：有货币系统为{1，2，5，10}，要求找出 16，那么第一次找出 10，第二次找出 5，第三次找出 1，恰好为最优解。

错误分析: 其实可以发现，这种算法并不是每一次都能构成最优解。反例如：货币系统{1，5，8，10}，同样找 16，贪心的结果是 10，5，1 三枚，但用两枚 8 的硬币才是最优解。因为这样，贪心的性质不成立，如此解题也是错的。

拓扑排序

给你一些物品的集合，然后给你一些这些物品的摆放顺序的约束，如"物品 A 应摆放在物品 B 前"，请给出一个这些物品的摆放方案，使得所有约束都可以得到满足。

算法：对于给定的物品创建一个有向图，A 到 B 的弧表示"物品 A 应摆放在物品 B 前”。以任意顺序对每个物品进行遍历。每当你找到一个物品，他的入度为 0，那么贪心地将它放到当前序列的末尾，删除它所有的出弧，然后对它的出弧指向的所有结点进行递归，用同样的算法。如果这个算法遍历了所有的物品，但却没有把所有的物品排序，那就意味着没有满足条件的解。

Section 1.3 Winning Solutions 竞赛中的策略

一个获得优势的好办法是写下你做题的策略。这会使你的思路很清晰，很清楚你做的东西是对是错。这样你是在先用你的思考时间来找出你的错误，而不是直接去想下一步应该怎么做……也就是先想好了再做。

心理准备也很重要。

竞赛中的策略

首先通读题目，然后写出它的算法、复杂度、数据规模、数据结构、程序细节……

- 想想所有可能的算法——然后选有效的中最笨的！
- 做数学计算！（时空复杂度，最坏的和期望的）
- 试着打破算法——利用特殊（让算法退化？）的测试数据（感觉是条件，但 test cases 就是测试数据）
- 做题的顺序：先做最简短的，根据你的情况（顺序（用时从短到长）：做过的、简单的、不常见的、难的）

编写程序代码——每个程序一次解决：

- 定下算法
- 想出特殊的测试数据（最狡猾的）
- 写出数据结构
- 写 input 代码，调试（写额外的输出程序去检测，笔者猜是“对拍”吧？）（后译者注：原句 write extra output routines to show data？个人认为意思是写额外的输出程序显示输入数据，估计目的为了确认输入处理正确——这个绝对有惨痛教训的。）（另一译者：另外一定要注意完成最终代码的时候删除这些输出语句！）
- 写 output 代码，调试
- 逐步完善：写注释，写出程序的思路
- 写出代码，一段一调试
- 运行&查正确性（使用特殊的数据）
- 试着去 break 代码的正确性——使用特殊的测试数据
- 不断优化——根据需求（使用极端的测试数据来测试运行时间）

时间控制 和 降低损失 scenarios

当出现错误的时候，有一个调试的计划；想想程序是什么样的，你希望它有什么样的输出？重点问题是：“什么时候你要去查错，什么时候你要放弃去做下一道题？”思考这些问题：

- 你已经用了多长时间去查它的错？
- 看起来这是什么类型的错误？
- 是你的算法错误吗？
- 是不是你的数据结构需要变化？
- 你有没有一些线索，错误在哪？
- 短时间的查错（20 分钟）比转去做别的题好；但是你可能用 45 分钟解决另一道题。
- 什么时候去回头看一些你已经放弃的问题？
- 当你已经花了很多时间去优化一道题，什么时候应该去看下一道题？
- 想到这——忘记之前，想想，从现在开始：怎样你才能在接下来的时间里得到最高的分数？

有一个 checklist，在交上你的代码之前：

- 在竞赛结束前 5 分钟停止对代码的修改？
- 停止维护。
- 停止调试性的输出。

技巧&诀窍

- 穷举，如果能这么做
- 保持简单，傻瓜 (KISS = Keep It Simple, Stupid)：简化就是聪明！

- 要点：注意限制条件（题目描述）
- 浪费内存空间吧，当它会让你的生活变得容易时。
- 不要删除你调试的额外输出，注释掉它。
- 不断优化，但是只要满足你的需求就可以了。
- 保留下所有的代码版本！
- 调试代码：
  - 空格是个好东西
  - 使用有实际意义的变量名
  - 不要重复使用变量
  - 逐步完善
  - 在代码之前写注释
- 可以的话，尽量避免指针
- 像避免灾难一样避免动态分配内存：静态分配所有变量。
- 试着不要去用浮点数 ；如果不得不用，在所有的地方去容差（不要用 == 判断相等）
- 对注释的评论：
  - 不要大段散文，只要简单的注释。
  - 解释高级的功能：++i; /* 把 i 自增 */ 写出这样的注释比没有任何注释还糟糕
  - 解释难懂的代码
  - 分割&功能的模块化
  - 让聪明的人懂你的程序，而不是代码
  - 所有的事情你都要去思考
  - 对所有你第一次看到的东西，说“我怎么把它再做一遍？”
  - 总是去说明每个数组的意义
- 记录你每一次比赛的表现：优点、错误、哪些地方可以做得更好；用这些来重写一遍，改进你的比赛计划！

复杂度

基础和命令符号 略

经验

- 当分析对于一个给定的数据，需要运行多长时间，首先一个常识是：现在（2004）计算机 1s 可以处理 100M 的内容。在一个时限 5s 的程序中，大概可以有 500M 的动作。好的优化可以让这个数字×2 甚至×4。算法的复杂度大概只能到这个数的一半。现在的竞赛常常对很大的数据给出 1s 的时限。
- 最多用 16M 内存
- 2 的 10 次方≈10 的 3 次方
- 如果你在 N 次迭代中，每次有 k 层循环，那你的程序有 O(N 的 k 次方)的复杂度。
- 如果你有 L level，每 level 有 b 层递归调用，那么复杂度是 O(b 的 L 次方)。
- 记住，N！是排列，2 的 n 次方 是 子集 或 组合。
- 对 N 的排序最好的时间复杂度是 O(N log N)。
- **做数学计算！**Plug in the numbers.

算复杂度例子： 略

解决问题例子

直接生成 VS 爆搜

（原文：Generating vs. Filtering，直译：生成 VS 筛选，怪怪的）

计算出非常多可能的解然后选择一个正确的（比如八皇后问题）方法是爆搜，一开始就计算可行解的方法是直接生成。一般，爆搜比较容易写（写得也快）但运行得慢。估算以确定题目规模允许爆搜还是不得不需要找出一定的算法。

预运算

有的时候打表或用其他数据结构能够使结果可能更快地被找出。这叫做预运算（也可以说是用空间换时间）。你可以把计算好的结果写到程序中编译，也可以在程序开始运行时计算，也可以让程序记下之前运算的结果。比如说，一个必须把大写字母转换为小写字母的程序可以打出一张对应表。竞赛中经常需要用到素数——许多时候比较常见的方法是先运算一张素数表然后在其他地方调用。

分解（在竞赛中最难做到）

虽然有少于 20 个基本算法在竞赛问题中用得到，但是解决由两个算法组合的问题是令人望而生畏的。试着把问题分割，让您可以结合循环或另一种算法来独立解决问题的不同部分。请注意，有时你可以在数据不同部分使用相同的算法两次（独立的!）来显著降低您的运行时间。

对称性

许多问题都有对称性（比如说，一对点间的距离从两种方向遍历是相同的）。对称性可以有两路、四路、八路甚至更多。试着利用对称性来降低程序运行时间。

举例来说，利用四路对称，你只需解决四分之一的问题然后利用对称性写出其余答案。（当然，要注意一些自对称方案只需输出 1 次或 2 次）

从前往后还是从后往前

令人惊奇的是，对于许多竞赛问题的测试数据中从后往前比从前往后运算要好很多。注意逆序处理数据或者构造一些常规数据以外的特别的顺序或题目中流行的测试数据。

简化

有些问题是可以改写成一个有点不同的问题，让你认为就像解决新问题，你可能对原来的问题有或很容易想到解决方案。当然，你应该解决是两个中容易的。另外，对一些问题可以用归纳法，先做点改变解决一个小问题然后找到完整解法。

Section 1.4 More Search Techniques 更多的搜索方式

样例: n 皇后问题 [经典问题]

将 n 个皇后摆放在一个 n x n 的棋盘上，使得每一个皇后都无法攻击到其他皇后。

深度优先搜索 (DFS)

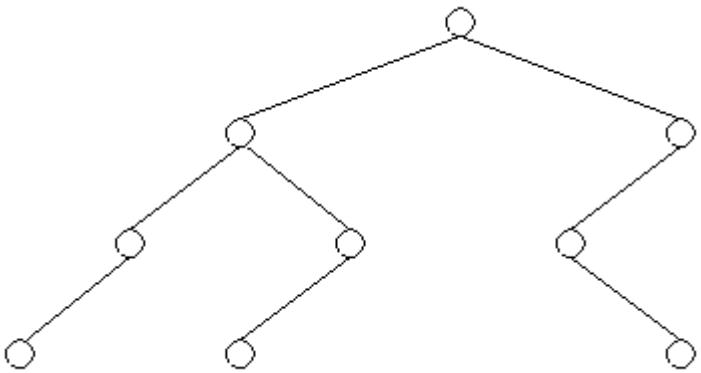
显而易见，最直接的方法就是把皇后一个一个地摆放在棋盘上的合法位置上，枚举所有可能寻找可行解。可以发现在棋盘上的每一行（或列）都存在且仅存在一个皇后，所以，在递归的每一步中，只需要在当前行(或列）中寻找合法格，选择其中一个格摆放一个皇后。

```
1 search(col)
2   if filled all columns
3       print solution and exit

4   for each row
5     if board(row, col) is not attacked
6         place queen at (row, col)
7         search(col+1)
8         remove queen at (row, col)
```

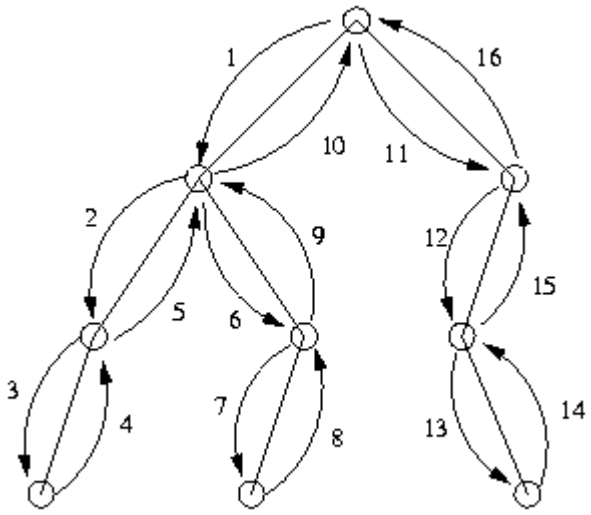
从 search(0)开始搜索，由于在每一步中可选择的节点较少，该方法可以较快地求解：当一定数量的皇后被摆放在棋盘上后那些不会被攻击到的节点的数量将迅速减少。

这是深度优先搜索的一个经典例题，该算法总是能尽可能快地抵达搜索树的底层：当 k 个皇后被摆放到棋盘上时，可以马上确定如何在棋盘上摆放下一个皇后，而不需要去考虑其他的顺序摆放皇后可能造成的影响（如当前情况是否为最优情况），该方法有时可以在找到可行解之前避免复杂的计算，这是十分值得的。

深度优先搜索具有一些特性，考虑下图的搜索树：



该算法用逐步加层的方法搜索并且适当时回溯，在每一个已被访问过的节点上标号，以便下次回溯时不会再次被搜索。绘画般地，搜索树将以如下顺序被遍历：



复杂度：

假设搜索树有 d 层（在样例中 d=n ，即棋盘的列数）。再假设每一个节点都有 c 个子节点（在样例中，同样 c=n ，即棋盘的行数，但最后一层没有子节点，除外）。那么整个搜索花去的时间将与 cd 成正比，是指数级的。但是其需要的空间较小，除了搜索树以外，仅需要用一个栈存储当前路径所经过的节点，其空间复杂度为 O(d)。

样例：骑士覆盖问题[经典问题]

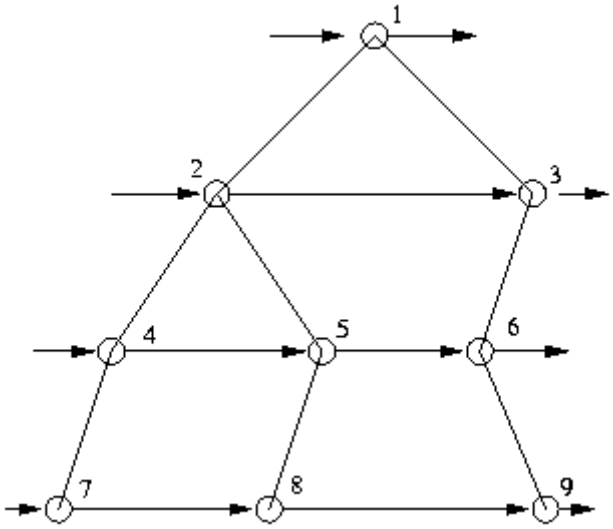
在一个 n x n 的棋盘中摆放尽量少的骑士，使得棋盘的每一格都会被至少一个骑士攻击到。但骑士无法攻击到它自己站的位置.

广度优先搜索 (BFS)

在这里，最好的方法莫过于先确定 k 个骑士能否实现后再尝试 k+1 个骑士，这就叫广度优先搜索。通常，广度优先搜索需用队列来帮助实现。

```
1 process(state)
2   for each possible next state from this one
3       enqueue next state

4 search()
5   enqueue initial state
6   while !empty(queue)
7       state = get state from queue
8       process(state)
```

广度优先搜索得名于它的实现方式：每次都先将搜索树某一层的所有节点全部访问完毕后再访问下一层，再利用先前的那颗搜索树，广度优先搜索以如下顺序遍历：



首先访问根节点，而后是搜索树第一层的所有节点，之后第二层、第三层……以此类推。

复杂度：

广度优先搜索所需的空间与深度优先搜索所需的不同（ n 皇后问题的空间复杂度为 O(n) ）,广度优先搜索的空间复杂取决于每层的节点数。如果搜索树有 k 层，每个节点有 c 个子节点，那么最后将可能有 c k 个数据被存入队列，这个复杂度无疑是巨大的。所以在使用广度优先搜索时，应小心处理空间问题。

迭代加深搜索 (ID)

广度优先搜索可以用迭代加深搜索代替。迭代加深搜索实质是限定下界的深度优先搜索，即首先允许深度优先搜索搜索 k 层搜索树，若没有发现可行解，再将 k+1 后再进行一次以上步骤，直到搜索到可行解。这个“模仿广度优先搜索”搜索法比起广搜是牺牲了时间，但节约了空间。

```
1 truncated_dfsearch(hnextpos, depth)
2   if board is covered
3       print solution and exit

4   if depth == 0
5       return

6   for i from nextpos to n*n
7       put knight at i
8       truncated_dfsearch(i+1, depth-1)
9       remove knight at i

10 dfid_search
11   for depth = 0 to max_depth
12     truncated_dfsearch(0, depth)
```

复杂度：

ID 时间复杂度与 DFS 的时间复杂度（O(n)）不同，另一方面，它要更复杂，某次 DFS 若限界 k 层，则耗时 ck 。若搜索树共有 d 层，则一个完整的 DFS-ID 将耗时 c0 + c1 + c2 + ... + cd 。如果 c = 2 ，那么式子的和是 cd+1 - 1 ，大约是同效 BFS 的两倍。当 c > 2 时（子节点的数目大于 2），差距将变小：ID 的时间消耗不可能大于同效 BFS 的两倍。 所以，但数据较大时，ID-DFS 并不比 BFS 慢，但是空间复杂度却与 DFS 相同，比 BFS 小得多。

算法选择：

当你已经知道某题是一道搜索题，那么选择正确的搜索方式是十分重要的。下面给你一些选择的依据。

简表：

搜索方式	时间	空间	适用情况
DFS	O(c k)	O(k)	必须遍历整棵树，要求出解的深度或经的过节点，或者你并不需要解的深度最小。
BFS	O(c d)	O(c d)	了解到解十分靠近根节点，或者你需要解的深度最小。
DFS+ID	O(c d)	O(d)	需要做 BFS，但没有足够的空间，时间却很充裕。

d ： 解的深度　　k ： 搜索树的深度　　d <= k

记住每一种搜索法的优势。如果要求求出最接近根节点的解，则使用 BFD 或 ID。而如果是其他的情况，DFS 是一种很好的搜索方式。如果没有足够的时间搜出所有解。那么使用的方法应最容易搜出可行解，如果答案可能离根节点较近，那么就应该用 BFS 或 ID，相反，如果答案离根节点较远，那么使用 DFS 较好。还有，请仔细小心空间的限制。如果空间不足以让你使用 BFS，那么请使用迭代加深吧！

类似问题：

超级质数肋骨[USACO 1994 决赛]

一个数，如果它从右到左的一位、两位直到 N 位（N 是）所构成的数都是质数，那么它就称为超级质数。例如：233、23、2 都是质数，所以 233 是超级质数。要求：读入一个数 N（N <= 9），编程输出所有有 N 位的超级质数。

这题应使用 DFS，因为每一个答案都有 N 层（最底层），所以 DFS 是最好的.

Betsy 的旅行 [USACO 1995 资格赛]

一个正方形的小镇被分成 NxN (2 <= N <= 6）个小方格，Besty 要从左上角的方格到达左下角的方格，并且经过每一次方格都恰好经过一次。编程对于给定的 N 计算出 Besty 能采用的所有的旅行路线的数目。

这题要求求出解的数量，所以整颗搜索树都必须被遍历，这就与可行解的位置与出解速度没有关系了。所以这题可以使用 BFS 或 DFS，又因为 DFS 需要的空间较少，所以 DFS 是较好的.

奶牛运输 [USACO 1995 决赛]

奶牛运输公司拥有一辆运输卡车与牧场 A ，运输公司的任务是在 A，B，C，D，E，F 和 G 七个农场之间运输奶牛。每两个农场之间的路程（可以用 floyed 改变）已给出。每天早晨，运输公司都必须确定一条运输路线，使得运输的总距离最短。但必须遵守以下规则：

农场 A 是公司的基地。每天的运输都必须从这开始并且在这结束。

卡车任何时刻都只能最多承载一头奶牛。

给出的数据是奶牛的原先位置与运输结束后奶牛所在的位置。

而你的任务是在上述规则内寻找最短的运输路线。

在发现最优解时必须比较所有可行解，所以必须遍历整棵搜索树。所以，可以用 DFS 解题.

横越沙漠 [1992 IOI]

一群沙漠探险者正尝试着让他们中的一部分人横渡沙漠。每一个探险者可以携带一定数量的水，同时他们每天也要喝掉一定量的水。已知每个探险者可携带的水量与每天需要的水量都不同。给出每个探险者能携带的水量与需要的水量与横渡沙漠所需的天数，请编程求出最多能有几个人能横渡沙漠。所有探险者都必须存活，所以有些探险者在中途必须返回，返回时也必须携带足够的水。当然，如果一个探险者返回时有剩余的水（除去返回所需的水以外），他可以把剩余的水送给他的一个同伴，如果它的同伴可以携带的话。

这题可以分成两个小问题，一个是如何为探险者分组，另一个是某些探险者应在何处返回。所以使用 ID-DFS 是可行的。首先尝试每一个探险者能否独自横渡，然后是两个人配合，三个人配合。直到结束。

Section 1.5 Binary Numbers 二进制算法

(我狂晕，这个……明明是不知道谁直接在线翻译，根本不通顺就放上来的，奈何鄙人也不通鸟语，只能将一些我看得懂的句子捋顺一下，望哪位大虾费神放个好的翻译上来) （额……我上学期刚学过口译，试着来完成前辈未竟的事业）

---二进制数转换(基础)---

电脑以 1 和 0 为基础操作的，这些被称为'二进制位'(bit)。 1 字节（Byte）内含有 8 位，例如：00110101。一个整型数据（int）在我的计算机上是 4 个字节，32 位：10011010 11010101 10100010 10101011。其他计算机的字节大小可能不同。

如你所见，32 个 1 和 0 记录下来或阅读起来有点麻烦。 因此，按照惯例人们把数字分成几组，每组 3 或 4 位：

```
1001.1010.1101.0101.1010.0010.1010.1011
10.011.010.110.101.011.010.001.010.101.011 < - (注意，从右往左开始计数 3)
```

这些分组，映射到数字，要么每 4 位表示一个 16 进制数（这里指个位数）或每 3 位表示一个 8 进制数。 很明显，需要一些新的十六进制数字（小数点后数字只去 0 .. 9，我们还需要 6 个数字）。现在，字母'A'..'F'被用来表示这些新的数字：10 .. 15。 下面的这一张表，显而易见地表现了它们的转化方式：

替换：十六进制：

```
000 - > 0  100 - > 4  0000 - > 0  0100 - > 4  1000 - > 8  1100 - > C
001 - > 1  101 - > 5  0001 - > 1  0101 - > 5  1001 - > 9  1101 - > D
010 - > 2  110 - > 6  0010 - > 2  0110 - > 6  1010 - > A  1110 - > E
011 - > 3  111 - > 7  0011 - > 3  0111 - > 7  1011 - > B  1111 - > F
```

所以现在我们很快地用 C 语言或其他语言表示以上陈述那些十六进制和八进制整数：

```
1001.1010.1101.0101.1010.0010.1010.1011
   - >  9    A    D    5    A    2    A    B- > 0x9AD5A2AB
```

（0x 为在十六进制数前面的标志）

```
10.011.010.110.101.011.010.001.010.101.011
       2 3 2 6 5 3 2 1 2 5 3 - > 023265321253
```

（这是前一个数字'0'）八进制的优点是可以比较容易、快速地写下来，但十六进制字节的更容易分离（这是因为数字是成对的）。

---位运算(进阶)---

有时为了效率将数字储存为位数字，而不是存储为整型。 比如在数组中记录选择（每个元素只可以是一个'是'或'不'的状态），用数组对选项进行标记（相同的状态，即'真'，每个位是'是'就是'假'），或者记录一些小的连续整数（例如对位元素 0 .. 3）。 当然，有时问题其实包含'位串'。

在 C/C++和其语言中，如果你知道它的八进制或十六进制表示形式，指定一个二进制数是很容易的： i= 0x9AD5A2AB;或 i= 023265321253; 更常见的是我们会用一个整数的权来记录状态。 比如下面这个例子：

i= 0x10000 + 0x100; 直到同一位上都是 1 之前它都是符合要求的： i= 0x100 + 0x100; 在这种情况下会发生进位，然后就得到了 0x200 而不是我们所希望得到的 0x100。(在此接着前辈的工作翻译下去，修改了前面一些翻译的和原文对比不准确的地方）而 C/C++等语言中的'|'，即“按位或”操作，却能达到我们所希望的要求。“按位或”操作的规则如下：

```
0 | 0 - > 0
0 | 1 - > 1
1 | 0 - > 1
1 | 1 - > 1
```

在 C 语言里'|'的操作称为'按位或'，以免与它的表兄'||'，即所谓的'逻辑或'或'or',混淆。 '||'运算符会计算其左侧的数，如果假（在 C 语言中为 0），再判断其右侧的数。 如果任意一个不为零，那么'||'的结果为真（为 1）。这是将'|'和'+'操作区分开来的最终规则。有时候这样的操作符运算以如下真值表的形式给出：

```
| | 0  1
```

```
---+------
 0 | 0  1
 1 | 1  1
```

显而易见，'按位或'的操作方式可以用来设置记录状态的整数。当任何一方或双方都是'1'时，输出结果为'1'。
最简单的查询方法是'逻辑与'（也称为'andif'）算子，记为'&'。真值表如下：

```
 & | 0 1
---+------
 0 | 0 0
 1 | 0 1
```

只有当输入的两个值均为'1'时，输出值才为'1'。 因此，如果你想知道一个整数的 0x100 位是否为'1'，语句很简单：

```
if (a & 0x100) { printf("yes, 0x100 is on\n"); }
```

C/C++以及其他语言还包含其他的操作符，比如“异或”，用'^'表示，真值表如下：

```
 ^ | 0 1
---+------
 0 | 0 1
 1 | 1 0
```

“异或”有时表示成'xor'，为了打字时比较轻松。 当且仅当输入的两个值之一是'1'时，输出结果才为 1。这个操作符能够很方便的来控制“开关”，即将数字的某一位由'1'变成'0'，或反之亦然。 例如以下这句代码：

```
a = a ^ 0x100; /* same as a ^= 0x100; */
```

在 0x100 位处将从 0 - > 1 或从 1 - > 0，根据其当前的值。
将某个数置零等同于两个基本操作符的运算（译者按：事实上下面是在讲异或的置零功能，即一个数和它本身进行异或操作得到结果是 0，例如 xor ax,ax 是将 ax 寄存器置零）。 我们新介绍一个一元运算符，它将一个数的每一位翻转，以创造一个数的“按位补”或者简称“补码”。这个运算符称为“按位取反”或者简称为“取反”，记为波浪符'~'。 下面是一个简单的例子：

```
char a, b; /* eight bits, not 32 */
 a = 0x4A;   /* 0100.1010 */
 b = ~a;     /* flip every bit: 1011.0101 */
 printf("b is 0x%X\n", b);
```

最终得出了这样的结果：

```
b == 0xB5
```

所以，如果一个数我们只有一位是 1（例如，0x100），那么~0x100 将所有其他的'0'位置'1'而将该位置'0'，得到：0xFFFFFEFF（注意'E'处于右起第三位，和原数的'1'位相同）。
以下两个操作符将一个数完全置零：

```
a = a & (~0x100); /* swtch off the 0x100 bit */
 /* same as a &= ~0x100;
```

因为取反后，原本所有为'0'的都变成了'1'，而所有为'1'的都变成了'0'，所以每一位都保证有 0 的存在再进行按位与操作后，所有的位数都变成了'0'。
总结
总之，这些操作符能够设置，清除，转换和查找整数中的任意一位二进制位：

```
 a |= 0x20;    /* turn on bit 0x20 */
a &= ~0x20;    /* turn off bit 0x20 */
a ^= 0x20;     /* toggle bit 0x20 */
if (a & 0x20) {
                    /* then the 0x20 bit is on */
}
```

（鉴于本人水平有限，如果有译得不准确的地方，欢迎大牛们予以辅正！）

Section 2.1 Graph Theory 图论知识

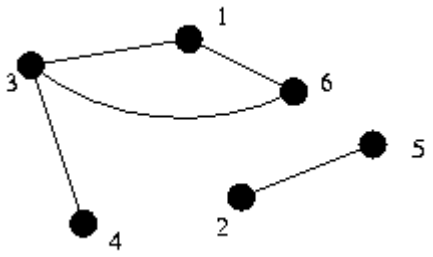
何为图?

正式地说，图 G 是由：所有结点的集合 V、所有边]的集合 E 构成的，简写成 G(V，E)。

我们可以把结点假设成一个“地点”,而结点集合就是一个所有地点的集合。同样,边可以被假设成连接两个“地点”的一条“路”,那么边的集合就可以认为是所有这样的“路”的集合。

表示方法：

图通常用如下方法表示；结点是点或者圈，而边是直线或者曲线。



在上图中，结点 V = {1, 2, 3, 4, 5, 6} ，边 E = {(1, 3), (1, 6), (2, 5), (3, 4), (3, 6)}.

每一个结点都是集合 V 中的一个数字，每条边都是集合 E 中的成员，注意并不是每个节点都有边与其他结点相连，这样没有边与其他结点相连的结点被称作***孤立结点***。

有时，边会与一些数值关联，类似表示边的长度或花费。我们把这些数字称作边的权

这样边有权的图被称为边权图。类似的，我们还定义了点权图，即每个结点都有一个权。

图论的几个例子

奶牛的电信 (USACO 锦标赛 1996)

农夫约翰的奶牛们喜欢通过电邮保持联系，于是她们建立了一个奶牛电脑网络，以便互相交流。这些机器用如下的方式发送电邮：如果存在一个由 c 台电脑组成的序列 a1,a2,...,a(c)，且 a1 与 a2 相连，a2 与 a3 相连，等等，那么电脑 a1 和 a(c)就可以互发电邮。很不幸，有时候奶牛会不小心踩到电脑上，农夫约翰的车也可能碾过电脑，这台倒霉的电脑就会坏掉。这意味着这台电脑不能再发送电邮了，于是与这台电脑相关的连接也就不可用了。有两头奶牛就想：如果我们两个不能互发电邮，至少需要坏掉多少台电脑呢？请编写一个程序为她们计算这个最小值和与之对应的坏掉的电脑集合。

图: 每个结点表示一台电脑，而边就相对应的成了连接各台电脑的缆线。

骑马修栅栏

农民 John 每年有很多栅栏要修理。他总是骑着马穿过每一个栅栏并修复它破损的地方。John 是一个与其他农民一样懒的人。他讨厌骑马，因此从来不两次经过一个一个栅栏。你必须编一个程序，读入栅栏网络的描述，并计算出一条修栅栏的路径，使每个栅栏都恰好被经过一次。John 能从任何一个顶点(即两个栅栏的交点)开始骑马，在任意一个顶点结束。每一个栅栏连接两个顶点，顶点用 1 到 500 标号(虽然有的农场并没有 500 个顶点)。一个顶点上可连接任意多(>=1)个栅栏。所有栅栏都是连通的(也就是你可以从任意一个栅栏到达另外的所有栅栏)。你的程序必须输出骑马的路径(用路上依次经过的顶点号码表示)。我们如果把输出的路径看成是一个 500 进制的数，那么当存在多组解的情况下，输出 500 进制表示法中最小的一个 (也就是输出第一个数较小的，如果还有多组解，输出第二个数较小的，等等)。输入数据保证至少有一个解。

图: 农民 John 从一个栅栏交叉点开始，经过所有栅栏一次。因而，图的结点就是栅栏交叉点，边就是栅栏。

骑士覆盖问题

在一个 n x n 的棋盘中摆放尽量少的骑士，使得棋盘的每一格都会被至少一个骑士攻击到。但骑士无法攻击到它自己站的位置.

图: 这里的图较为特殊，棋盘的每一格都是一个结点，如果骑士能从这一格跳到另一格，这就说在这两个格相对应的结点之间有一条边。

穿越栅栏 [1999 USACO 春季公开赛]

农夫 John 在外面的田野上搭建了一个巨大的用栅栏围成的迷宫。幸运的是，他在迷宫的边界上留出了两段栅栏作为迷宫的出口。更幸运的是，他所建造的迷宫是一个

完美的迷宫：即你能从迷宫中的任意一点找到一条走出迷宫的路。

这是一个 W=5,H=3 的迷宫：

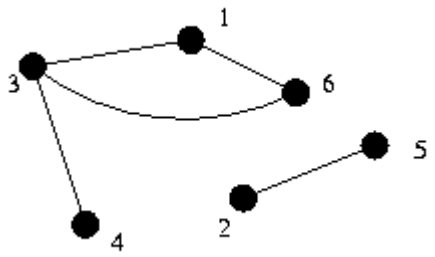
```
+-+-+-+-+-+
|       |
+-+ +-+ + +
|   | | |
+ +-+-+ + +
| |   |
+-+ +-+-+-+
```

如上图的例子，栅栏的柱子只出现在奇数行或奇数列。每个迷宫只有两个出口。

图: 如上图，图的每一个位置都是一个结点，如果两个相邻的位置之间没有被栅栏分开，则说在这两个位置相对应的结点之间有一条边。

用语：

我们再看刚才的图：



如果有一条边起点与终点都是同一个结点，我们就称它为环边, 表示为(v, v)，上图中没有环边。

简单图是指一张没有环边且边在边集 E 中不重复出现的图。与简单图相对的是复杂图。在我们的讨论中不涉及复杂图，所有的图都是简单图。

边 (u,v) 连接了结点 u 和结点 v 。例如，边 (1,3) 连接了结点 1 与 3。

结点的度是指连接该结点所有边的个数。例如，结点 3 的度数是 3，结点 4 的度数是 1。

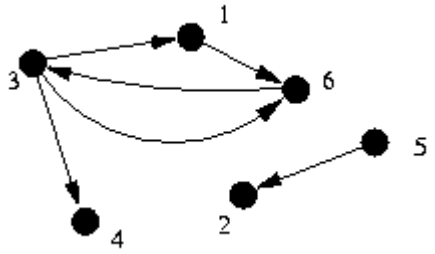
通常，如果结点 u 和 v 被用一条边连接，我们就说结点 u 与 v 相连，例如，上图结点 2 与 5 相连。

稀疏图的定义是图的边的总数小于可能边数((N x (N-1))/2)（N 为结点数），而密集图的定义相反。例如，上图就是一张稀疏图。

有向图：

在以上内容中我们介绍的图都为无向图，即每一条边都是“双向”的，如果存在一条边(1,3)，则我们可以从结点 1 到达结点 3，也可以从结点 3 到达结点 1，换句话说，如果存在边(1,3)就必定存在边(3,1)。

但有时，图也必须被定于成有向图，即每条边都有一个方向，我们只能“单向”地根据边的方向遍历图。这样的边又称为]弧。弧用带箭号的直线或弧线表示。

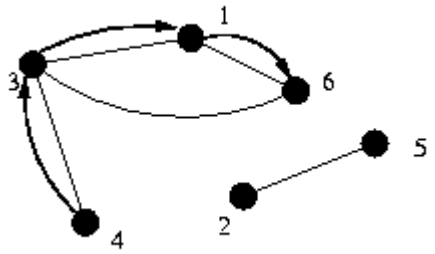


有向图结点的出度就是指从该结点“出发”的弧的个数，相反，结点的入度就是指在该结点“结束”的弧的个数。例如，上图结点 6 的入度为 2，出度为 1。

路径

如果我们说结点 u 到结点 x 有路径，就是指存在一个结点的序列 (v 0, v 1, ..., v k) 且 v 0 = u 、v k = x ， 以及(v 0, v 1) 是边集中的一条边，同样，(v 1, v 2), (v 2, v 3)……也是。

例如，上面的无向图， (4, 3, 1, 6) 就是从 4 到 6 的一条路径。



这条路径包含了边(4,3)、(3,1)、(1,6)。

如果结点 x 到 v 有一条路径，则我们从结点 x 开始通过边必定能访问到结点 v。

在一条路径序列中，如果所经过的结点都只在序列中出现一次，我们就称它为简单路径

环的定义是一条起始结点与中止结点为同一结点的路径，如果一个环除了起始结点（中止结点）以外的所有结点都只在环中出现一次，那么这种环被称为简单环。

以上定义同样适用于有向图。

图的表示法

选择一个好的方法来表示一张图是十分重要的，因为不同的图的表示方法有着不同的时间及空间需求。

通常，结点被用数字编号来表示，我们可以通过它们的编号数字来对他们进行索引。 这样，似乎所有问题都集中在如何表示边上了。

边集

边集表示法似乎是最明显的表示法，他将所有边列成一张表，用结点来表示边。

该表示法容易编写，容易调试且所需空间小。但是，它需要花费相当大的代价用于确定一条边是否存在，即确定两个结点是否相连。利用边集添加新边是很快的，但删除旧边就很麻烦了，尤其是当需要删除的边在边集中的所在位置不确定时。

对于带权图，该表示法也只需要在边集中添加一个元素，用于记录边权。同样，该表示法也适用于有向图和稀疏图。

例：

一张无向无权图可以表示成边集如下：

```
 v1   v2
```

e1	4	3
e2	1	3
e3	2	5
e4	6	1
e5	3	6

邻接矩阵

邻接矩阵式表示图的另一种方法，邻接矩阵是一个 N 乘以 N 的数组（N 为结点数）。 如果边集 E 中存在边(i, j)，则对应数组的(i,j)元素的值为一。相反，如果数组元素的值为零，就意味着图中结点 i 与 j 之间没有边。无向图的邻接矩阵总是关于左上右下对角线对称。

该表示法容易编写。但他对空间的需求和浪费较大，尤其是对于较大的稀疏图，调试起来也比较麻烦，并且寻找与某一结点相连的结点也很慢。不过该表示法在判断两结点是否相连，以及在添加、删除结点方面速度都是极快的。

对于带权图，数组的(i,j)元素的值被表示为边(i,j)的权。对于无权复杂图来说，数组的(i,j)元素的值可以用于表示结点(i,j)之间边的个数。而对于有权复杂图来说，邻接矩阵很难将其表示清楚。

例：

下面是用邻接矩阵表示一幅无权图的例子：

	V1	V2	V3	V4	V5	V6
V1	0	0	1	0	0	1
V2	0	0	0	0	1	0
V3	1	0	0	1	0	1
V4	0	0	1	0	0	0
V5	0	1	0	0	0	0
V6	1	0	1	0	0	0

邻接表

第三种表示图的方法是用一个列表列出所有与现结点之间有边存在的结点名称。该方法可以用一个长度为 N 的数组来实现（N 为结点个数），数组的每一个元素都是一个链表表头。数组的第 i 元素所连接的链表连接了所有与结点 i 之间有边的结点名称。

该表示方法较难编写，特别是对于复杂图，两结点之间边的个数不受限制的时候，链表可能要做成环，或者要动态分配内存。邻接表的调试也同样十分麻烦，特别是使用了环形链表后。但是，这种表示法所需求的空间与边集相同，寻找某结点的相邻结点也十分容易，然而，如果需要判断两结点是否相邻，就需要遍历该结点的所有相邻结点以证明相邻性，这浪费了不少时间。添加边十分容易，但删除边就十分困难了。

将该表示法用于带权图，就需要在链表的每一节上添加一个元素用于储存该节表示的两结点之间的权。有向图与复杂图同样可以用邻接表表示。对于有向图，我们只需存储单向的相邻即可。

例：邻接表表示无向图如下：

结点	邻接结点
1	3, 6
2	5
3	6, 4, 1
4	3
5	2
6	3, 1

表示法的选择

对于某些图来说，我们并不需要考虑所有结点之间的关系，例如上面的骑士覆盖和穿越栅栏两题，我们只需考虑与某结点相邻结点之间的关系，确定相连，我们就不用考虑存储两个不相邻结点之间的信息。当然，这些信息来自于题目本身的暗示。

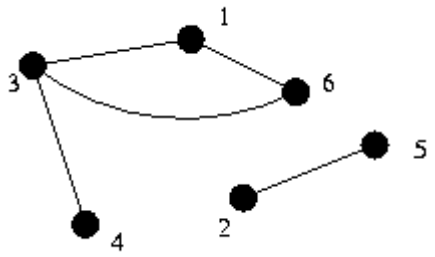
如果你发现一种表示方法可以在规定空间与时间范围内实现你的算法解决问题，并且可以使你的程序变得简洁、容易调试，那它通常就是对的。记住，编写与调试的简单是最重要的，我们不需要为一些毫无意义的加快程序速度，减少使用空间来浪费编写与调试的时间。

我们还要通过题目给我们的信息，确定我们要对图进行哪些操作，权衡后来决定表示方法。下面的表示我们为您提供的选择依据（N 为结点数，M 为边数，d max 为图中度的最大值）：

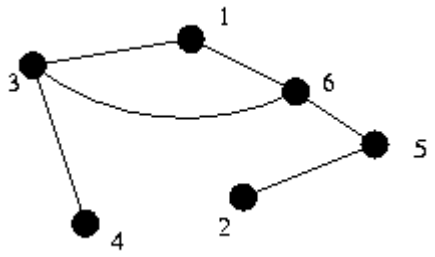
功效表	边集	邻接矩阵	邻接表
消耗空间	2xM	N2	2xM
连接判断的消耗时间	M	1	d max
检查某结点所有相邻结点的消耗时间	M	N	d max
添加边的消耗时间	1	1	1
删除边的消耗时间	M	2	2xd max

图的连通性

如果一个图的任意两个结点之间都有一条以上的路径相连，我们就称该图为连通图。例如下图就不是连通图，因为结点 2 到结点 4 之间没有路径相通。



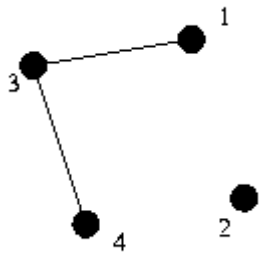
但是如果我们在该图的结点 5 和 6 之间添加一条边，该图就成为了连通图。



子图：

如果 G = (V, E)成立，且有集合 V'是 V 集合的子集，集合 E'是集合 E 的子集，那么我们就说 图 G' = (V', E') 是图 G = (V, E) 的子图。

同时，在边集合 E'中出现的边必须是连接结点集合 V'中出现的两结点的边。我们考虑下图，它是例图的一个子图，其中 V'= {1, 2, 3, 4,}，E' = {(1, 3), (1, 4)}。但是如果我们在边集 E'中添加一条边(1, 6)，E'仍然是 E 的子集，但由于在 V'中没有出现结点 6，边(1, 6)就不可能在子图 G' = (V', E')中出现。

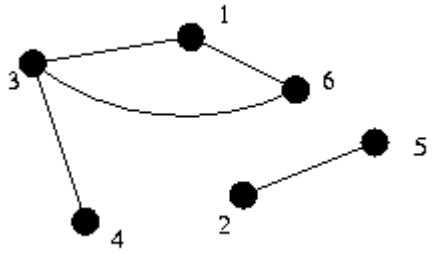


连通子图]，如同它的名称一样，是一张图的子图，且该子图就有连通性，如下图，{1, 3, 4, 6}，{2, 5}，{1, 3, 4} ，{1，3}都是该图的连通子图（原文中在此仅标注了结点，我们尚且将边默认为所有这些结点之间的边——译者）。

同时，我们还定义了极大连通子图，它的定义可以理解为将连通子图扩展的尽量大，在例图中仅有两个极大连通子图：{1, 3, 4, 6}和{2, 5}。

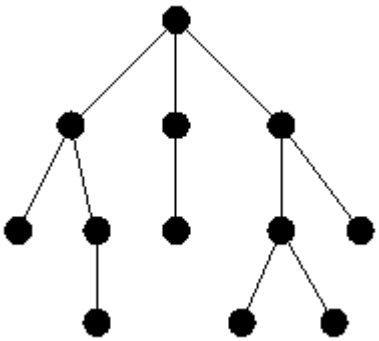
非连通图的极大连通子图被称为连通分量；有向图中的连通图叫做强连通图；非强连通图中的极大连通子图被称为]强连通分量。

一个连通图的生成树是该图的极小连通子图，在有 N 个结点的情况下，生成树仅有 N-1 条边。



特殊的图

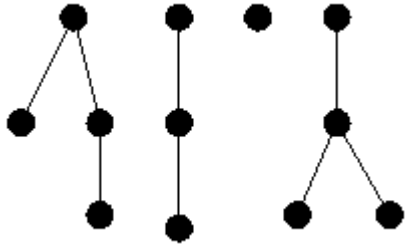
连通且无环的无向图被称作树



许多树都是“层次化”的。通常，树都有一个“最顶上的”结点，被称作根结点；相反，“最底下的”结点们被称作叶结点。除了根结点以外，所有结点都仅有一个父结点，即与它相连的那个上一层结点；除了叶结点以外，所有结点都至少有一个子女结点，即与它相连的所有下一层结点。

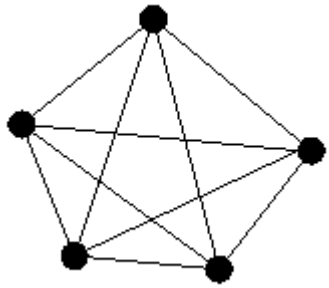
同样，类推地，还有祖先结点、后代结点。经过这样的定义后，树的样式就可以构划得像一棵倒过来生长的树了（如上图）。

不连通且无环的无向图被称为森林（如下图）。他也可以理解为是有许多树构成的图。

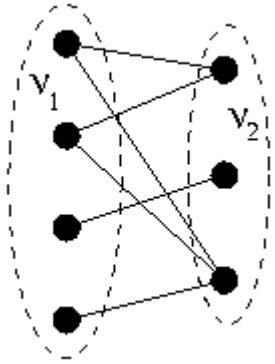


有向无环图（Directed Acyclic Graph）被称为 DAG，是他的英文缩写。

一个图的任意结点与其它所有结点有边相连的图被称为完全图



如果一个图的所有结点可以被分为两个组，同组之间的任意结点都没有边相连，即所有边连接的都是不同组的两个结点，这样的图被称为二分图



Section 2.1 Flood Fill 种子染色法

译 by Lucky Crazy & Felicia Crazy

（Flood Fill 按原意应翻译成“水流式填充法”（如果我没译错），有些中文书籍上将它称作“种子染色法”，然而大部分的书籍（包括中文书籍）都直接引用其英文原名：Flood Fill。介于此，下文所有涉及到 Flood Fill 的都直接引用英文 ——译者）

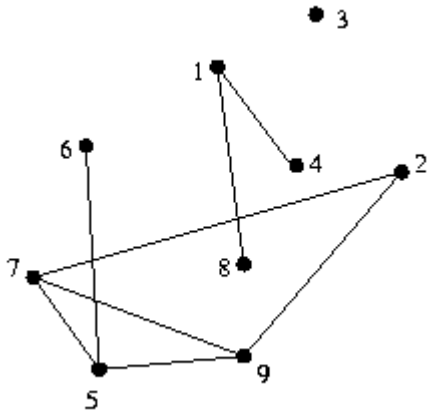
样例： 相连的农场

Farmer John 的农场被一次意外事故破坏了，有一些农场与其他的农场之间有道路相连，而有些道路却已被破坏。这使得 Farmer John 无法了解到从一个农场能否到达另一个农场。你的任务就是帮助 Farmer John 来了解哪些农场是连通的。

给出：

上题实际上就是要求寻找一张无向图的所有极大连通子图。

给出一张未知连通性的图，如下图：



可知，该图的极大连通子图是：{1,4,8}， {2,5,6,7,9} 和 {3}。

算法: Flood Fill

Flood Fill 可以用深度优先搜索，广度优先搜索或广度优先扫描来实现。他的实现方式是寻找到一个未被标记的结点对它标记后扩展，将所有由它扩展出的结点标上与它相同的标号，然后再找另一个未被标号的 结点重复该过程。这样，标号相同的结点就属于同一个连通子图。

深搜：取一个结点，对其标记，然后标记它所有的邻结点。对它的每一个邻结点这么一直递归下去完成搜索。

广搜：与深搜不同的是，广搜把结点加入队列中。

广度扫描（不常见）：每个结点有两个值，一个用来记录它属于哪个连通子图（c），一个用来标记是否已经访问（v）。算法对每一个未访问而在某个连通子图当中的结点扫描，将其标记访问，然后把它的邻结点的（c）值改为当前结点的（c）值。

深搜最容易写，但它需要一个栈。搜索显式图没问题，而对于隐式图，栈可能就存不下了。

广搜稍微好一点，不过也存在问题。搜索大的图它的队列有可能存不下。深搜和广搜时间复杂度均为 O(N+M)。其中，N 为结点数，M 为边数。

广度扫描需要的额外空间很少，或者可以说根本不要额外空间，但是它很慢。时间复杂度是 O(N^2+M)。

（实际应用中，我们一般写的是 DFS，因为快。空间不是问题，DFS 可改用非递归的栈操作完成。但为了尊重原文，我们还是译出了广度扫描的全过程。——译者）

广度扫描的伪代码

代码中用了一个小技巧，因此无须额外空间。结点若未访问，将其归入连通子图（-2），就是代码里的 component -2。这样无须额外空间来记录结点是否访问，请读者用心体会。

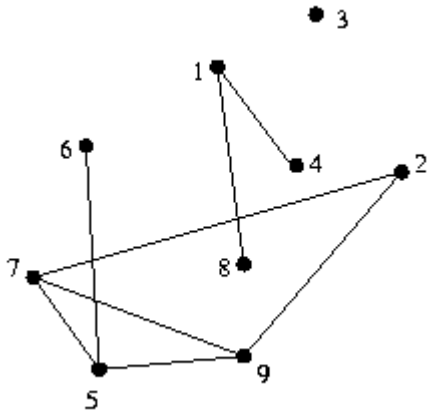
```
# component(i) denotes the
```

```
# component that node i is in
1 function flood_fill(new_component)
2 do
3   num_visited = 0
4   for all nodes i
5   if component(i) = -2
6     num_visited = num_visited + 1
7     component(i) = new_component
8     for all neighbors j of node i
9       if component(j) = nil
10       component(j) = -2
11 until num_visited = 0
12 function find_components
13 num_components = 0
14 for all nodes i
15   component(node i) = nil
16 for all nodes i
17   if component(node i) is nil
18     num_components =
          num_components + 1
19     component(i) = -2
20     flood_fill(component
                num_components)
```

算法的时间复杂度是 O(N 2)，每个结点访问一次，每条边经过两次。

实例

考虑刚才的那张图：



开始时，所有的结点都没有访问。（下例中未访问被表示为 -2)

首先从结点 1 开始，结点 1 未访问，那么先处理结点 1，将它归入连通子图 1。

结点 1

连通子图 -2

标记完成后，对它进行第一步的扩展，由结点 4 和结点 8 与结点 1 连通，故它们被扩展出来。

结点 1 4 8

连通子图 1 -2 -2

之后，先处理结点 4，将它与结点 1 归入相同的连通子图。现在它没有可扩展的结点了（结点 1 已被扩展过）

结点 1 4 8

连通子图 1 1 -2

接着处理结点 8。结果与结点 4 一样。

结点 1 4 8

连通子图 1 1 1

现在，所有与结点 1 连通的结点都已扩展，标号为 1 的连通子图产生了。那么我们将跳出扩展步骤，寻找下一个连通子图，标号为 2。

与上一步相同的顺序，找到结点 2。

结点 1 2 4 8

连通子图 1 -2 1 1

扩展结点 2，结点 7 与结点 9 出现。

结点 1 2 4 7 8 9

连通子图 1 2 1 -2 1 -2

下一步，扩展结点 7，结点 5 出现。

然后是结点 9。

扩展结点 5。结点 6 出现。

结点 1 2 4 5 6 7 8 9

连通子图 1 2 1 2 -2 2 1 2

很遗憾，结点 6 没有可供扩展的结点。至此连通子图 2 产生。

结点 1 2 4 5 6 7 8 9

连通子图 1 2 1 2 2 2 1 2

之后寻找连通子图 3，至此，仅有结点 3 未被扩展。

结点 3 没有可供扩展的结点，这样，结点 3 就构成了仅有一个结点的连通子图 3。

结点 1 2 3 4 5 6 7 8 9

连通子图 1 2 3 1 2 2 2 1 2

结点 3 处理结束后，整个图的所有 9 个结点就都被归入相应的 3 个连通子图。Flood Fill 结束。

问题提示

这类问题一般很清晰，求解关于“连通”的问题会用到 Flood Fill。它也很经常用作某些算法的预处理。

扩展与延伸

有向图的连通性比较复杂。

同样的填充算法可以找出从一个结点能够到达的所有结点。每一层递归时，若一个结点未访问，就将其标记为已访问（表示他可以从源结点到达)，然后对它所有能到达且为访问的结点进行下一层递归。

若要求出可以到达某个结点的所有结点，你可以对后向弧做相同的操作。

例题

控制公司 [有删节，IOI 93]

已知一个带权有向图，权值在 0-100 之间。

如果满足下列条件，那么结点 A“拥有”结点 B：

A = B

从 A 到 B 有一条权值大于 50 的有向弧。

存在一系列结点 C 1 到 C k 满足 A 拥有 C 1 到 C k, 每个节点都有一条弧到 B，记作 x 1 ，x 2 ...x k，并且 x 1 + x 2 + ... + x k > 50。

找出所有的（A，B）对，满足 A 拥有 B。

分析：这题可以用上面提到的“给出一个源，在有向图中找出它能够到达的结点”算法的改进版解决。要计算 A 拥有的结点，要对每个结点计算其“控股百分比”。把它们全部设为 0。现在，在递归的每一步中，将其标记为属于 A 并把它所有出弧的权加到“控股百分比”中。对于每个“控股百分比”超过 50 的结点，进行同样的操作（递归）。

街道赛跑 [IOI 95]

已知一个有向图，一个起点和一个终点。

找出所有的 p，使得从起点到终点的任何路径都必须经过 p。

分析：最简单的算法是枚举 p，然后把 p 删除，看看是否存在从起点到终点的通路。时间复杂度为 O(N (M + N))。题目的数据范围是 M

牛路 [1999 USACO 国家锦标赛，有删节]

连通图的直径定义为图中任意两点间距离的最大值，两点间距离定义为最短路的长。

已知平面上一个点集，和这些点之间的连通关系，找出不在同一个连通子图中的两个点，使得连接这两个点后产生的新图有最小的直径。

分析：找出原图的所有连通子图，然后枚举不在同一个连通子图内的每个点对，将其连接，然后找出最小直径。

笨蛋建筑公司

Farmer John 计划建造一个新谷仓。不幸的是，建筑公司把他的建造计划和其他人的建造计划混淆了。Farmer John 要求谷仓只有一个房间，但是建筑公司为他建好的是有许多房间的谷仓。已知一个谷仓平面图，告诉 Farmer John 它一共有多少个房间。

分析：随便找一个格子，遍历所有与它连通的格子，得到一个房间。然后再找一个未访问的格子，做同样的工作，直到所有的格子均已访问。虽然题目给你的不是直接的图，但你也可以很容易地对其进行 Flood Fill。

Section 2.2 Data Structures 数据结构

预备知识

- 图论

如何选择最完美的数据结构

从一些数据结构中选择一个合适的数据结构来表示一个题目中的数据。

可否工作？

如果数据结构将不能正常工作，这是完全没有用的。对于一个问题，什么样的算法可以决定什么样的数据结构，并确保数据结构可以处理算法。如果不能，那么要么必须添加一些数据结构，或者你需要找到另外的数据结构来构建这个算法。

可否编码？

如果您不知道或不记得如何编写一个给定的数据结构，选择其他的数据结构。确保你有一个清醒的认识，知道每一个操作对数据结构的影响。在此，另一个要考虑的是内存。这个数据结构能在适当大小的内存空间里运行吗？如果不能就简化它，或选择一个新的数据结构。否则，从一开始就注定了它不能正常工作。

按时完成？

由于比赛是限定时间的，5 小时内解决 3 至 5 道题。如果你在第一题上为了数据结构就花了一个半小时，那么你基本上已经悲剧了。

可否调试？

在选择数据结构的时候很容易忘掉调试这部分。请记住一点，一个程序除非它能正常工作，否则它就是无用的。不要忘记，在整个比赛的时间中，调试占的比例是很大的，因此必须把调试时间考虑到写程序的时间中。一个数据结构是否容易调试取决于下面两个属性。

- **静态的数据结构更易于检查** 通常来说，规模更小、更紧凑的表达形式更容易检查。此外，静态分配的数组比链表甚至于动态数组更容易检查。
- **静态的数据结构更容易被显示** 对于更复杂的数据结构，最简单的检验方法是写一个小例子来输出数据。可惜，由于时间的限制，你可能想限制自己的文本输出。这意味着，像树和图的结构将难以检查。

是否快速？

很奇怪，速度是在选择数据结构的时候是最后一个要考虑的。一个慢的程序很容易发现是什么导致了慢，但是一个快速的错的程序却不容易发现什么导致了错，除非运气很好。

结论

总的说来，遵循 KISS 原则: “使其简单，傻瓜化。”(Keep It Simple, Stupid.) 有时候一定的复杂度是有必要的，但请确保它值得。请牢记：在开始的时候花时间去确保你选择了一个合适的数据结构，比之后不得不用另一个数据结构去替代它，要划算得多。

避免动态内存

通常情况下，你应当避免使用动态内存，因为： **使用动态内存会很容易犯错误!!** 重写已分配的内存，忘记释放内存，忘记分配内存，只是使用动态内存时引入的一点错误。此外，这些错误的出错代码很难告诉我们发生错误的位置，因为它可能发生在一个（可能更晚）内存操作时。 **检查数据结构的含义太难了!!** 集成开发环境不能很好地操作动态内存，尤其对于 C 语言更是一塌糊涂。尝试考虑使用并行数组来实现动态内存，比如使用链表时用另一个数组来存储 next 值序列。有时你可以动态分配这些，但是因为它只需要完成一次，所以用数组来实现插入或删除操作会比分配释放内存更简单。尽管如此，有时动态内存也是一种好的办法，特别是对于未知数据范围的大型数据结构。

避免猎奇想法

不要掉进“猎奇”的陷阱。你可能刚发现了最有趣的结构，但是要记住：

- 好点子，不工作，没有用。
- 酷想法，编不出，也没用。

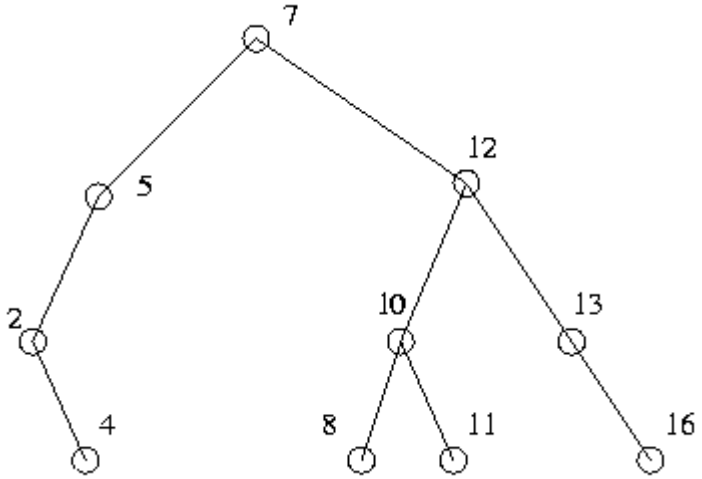
你的数据结构是有效的，比你如何改进你的数据结构更加重要。

基本结构

有五种基本数据结构：数组、链表、栈、队列和双向队列。你可能在之前已经见过这些东西；如果没有，可以问你老师。

二叉查找树

二叉查找树使您能够迅速搜索对象的集合（实型或整数）以确定给定的数值是否在集合中。基本上，二叉查找树是一个加权，有根的二叉规则树。这样的描述意味着树中的每个节点可能有一个'右‘子节点和一个`左'子节点（但双方或一方可能会丢失）。此外，每个节点都有与之相关联的对象，权值是对象在该节点的数值。二叉查找树有这样的属性：每个节点的左子树上的值小于该节点的值，每个节点的右子树的值大于或等于它。



每个节点通常由四个域构成，一个指针域指向左子节点，一个指针域指向右子节点，一个域存储权值，一个域存储对象。

为何二叉查找树可用？

给出一个具有 N 个对象的集合，二叉查找树查找一个对象只需 $O(\lg n)$ 的时间，如果它不是一棵效率较低的树（如，一棵所有节点都没有左节点的树完成查找需要 $O(n)$ 的时间）。此外，不同于存储在数组中，插入和删除对象也只需 $O(\lg n)$ 的时间。

二叉查找树的延伸结构

有时在节点上加入一个指针域指向节点的父节点也是很有用的。这里有几种变化，以确保二叉查找树不会低效率：splay tree 、红黑树、Treap、B 树、AVL 树 。这些数据结构的实现有一定难度，而且随机构造的二叉查找树通常效率已经很高了，所以一般没必要实现这些算法了。

散列表（哈希表、HASH 表）

散列表通过一种可以快速查找的方式存储数据。假设有一个集合和一种数据结构，要求快速回答：“这个对象是否在数据结构中？”（如，这个单词是否在字典中？）散列表可以用少于二叉查找所用的时间完成这个功能。可以这样想：找到一个函数把集合的所有元素映射到一个整数从 1 到 x （x 大于集合里的元素个数）。定义一个从 1 到 x 的数组，把每个元素存储在元素经函数映射后的位置。之后确定一个元素是否在所给出的集合里，只要把它代入函数，看所映射的位置是否为空。如果不确信元素在上面，那么就去看一下是否和你存储的相一致。

举个例子，假设函数定义 3 个字符的单词上，有 (首字母 + (中字母 * 3) + (尾字母 * 7)) mod 11 (定义 A=1, B=2, ... , Z=26)，单词有 'CAT' , 'CAR' , 和 'COB' 。以 ASCII 为标准，则 'CAT' 在函数上的映射是 3，'CAR' 在函数上的映射是 0，'COB' 在函数上的映射是 7，故散列表如下： 0: CAR 1 2 3: CAT 4 5 6 7: COB 8 9 10 现在来看 'BAT' 在散列表中的情况，把 'BAT' 代入散列函数得到 2，散列表在 2 的位置是空的，所以它不在集合里。另一方面，把 'ACT' 代入散列函数得到 7，所以程序必须检验条目 'COB' 和 'ACT' 是否相同。考虑如下函数：

```
#define NHASH 8999          /* 保证它是素数 */

hashnum(p)
char *p;
{
    unsigned int sum = 0;
    for ( ; *p; p++)
    sum = (sum << 3) + *p;
    return sum % NHASH;
}
```

对每个输入，函数返回 0..NHASH-1 的某个数。它的输出是均匀随机的。这个简单的函数要求 NHASH 是素数。把上面的函数与下面的主程序合在一起：

```
#include

main() {
    FILE *in;
    char line[100], *p;
    in = fopen ("/usr/share/dict/words", "r");
    while (fgets (line, 100, in)) {
    for (p = line; *p; p++)
        if (*p == '\n') { *p = '\0'; break; }
    printf("%6d %s\n", hashnum(line), line);
    }
    exit (0);
}
```

就会产生类似这样的英文单词表（当然这得在 Linux 下运行）：

```
  4645 aback
  4678 abaft
  6495 abandon
  2634 abandoned
  4810 abandoning
   142 abandonment
  7080 abandons
  4767 abase
  2240 abased
  7076 abasement
  4026 abasements
  2255 abases
  4770 abash
   222 abashed
   237 abashes
  2215 abashing
```

```
   361  abasing
  4775  abate
  2304  abated
  3848  abatement
   ... ...
```

你可以看到函数产生的数字全部是均匀随机的，并且起码在这个例子中没有重复。当然，如果你有 NHASH+1 个单词，鸽笼定理证明了至少会有两个单词返回值相同，这就叫“冲突”。实际上，散列表使用链表技术解决了这种冲突，即相同值的单词列在同一区域。看看散列表究竟该怎么用吧。首先，建立散列表的链表结构，就像这样：

```
struct hash_f {
    struct hash_f *h_next;
    char *h_string;
    int   h_value;   /* 一些和字符串相关的值*/
              /* 完全自由选择如何使用或它已经存在*/
};

struct hash_f *hashtable[NHASH];      /* 每个链表的头指针 */
                      /* 全局变量自动置空 */
```

产生如此散列表，比如用两个元素为父亲：

```
      hashtable       *hash_f          *hash_f
    +------------+
  0 |            |  +-----------+    +-----------+
    +------------+  |           *|-+  |         0|
  1 |            |  +-----------+ |  +-----------+
    +------------+  | 'string1' | |  | 'abc def' |
  2 |          * |->+-----------+ +->+-----------+
    +------------+  | val=1234  |    | val=43225 |
  3 |            |  +-----------+    +-----------+
    +------------+
          ...
8998 |          |
    +------------+
```

下面是插入操作：

```
struct hash_f *
hashinsert(p, val)
char *p;
int val;
{
    int n = hashnum(p);             /* 表的位置 */
    char *h = malloc( sizeof (struct hash_f) ); /* 新建散列元素 */

/* 链接到表头： */
    h->h_next = hashtable[n];
    hashtable[n] = h;

/* 可选值： */
    h->h_val = val;
```

```
/* 然后在链中找到适合元素放置的地方: */
    h->h_string = malloc( strlen(p) + 1 );
    strcpy (h->h_string, p);

    return h;
}
```

下面是查找操作（若找到则返回其指针）：

```
struct hash_f *
hashlookup(p) {
    struct hash_f *h;

    int n = hashnum(p);             /* 初始位置 */

    for (h = hashtable[n]; h; h=h->h_next)    /* 遍历链表 */
    if (0 == strcmp (p, h->h_string)) /* 匹配? 完成! */
        return h;
    return 0;                /* 未找到 */
}
```

现在你可以快速地插入和查找了，平均需要(链表大小/2)次字符串比较。

为何散列表可用？

散列表只占用一点点内存，而程序查找元素几乎只要常数时间。通常程序要评估函数的价值，并且可能需要在表中比较一次或若干次【这句话翻得不好】。

散列函数

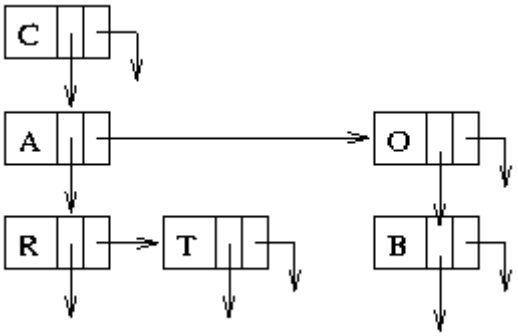
经常忘记的是，更精确点说，避免冲突的方法是找一个好的散列函数。举个例子，以单词前三个字母作为散列值显然很悲催。在此散列函数下，前缀 "CON" 会产生一个巨大的序列。你选择的函数要尽可能使不同的元素分配到不同的位置:

- 把巨大值用表的大小求模（选一个素数会干得特别好）。
- 素数是你的朋友。用它们相乘。
- 试着用小的 changes 映射到完全不同的地方。
- 不要想用两个小 changes 就能撤销映射到表外的函数（如一个变换）。
- 有一个研究的全域，要求创造一个“完美散列函数”以至于没有冲突，但是，完全产生均匀随机显然是要大量工作的；希望以后会有吧，起码现在木有啊。

散列变量

只用数值处理信息经常十分有用。比如当在一个大集合中搜寻一个小子集，用散列表处理已访问域，你可能想把它的搜索的价值定位在哈希表中. 甚至一个小散列表通过彻底减少搜寻空间都能改进运行时间。比如用首字母标识字典，你只需要找首字母就可以了。

一种特殊的树——“Trie”



这一节的图示：

简单来说，Trie 就是指有根树。它有不受限制的出度(即一个节点可能有任意多个子节点). 一个节点的子节点存储在一个链表之中, 所以一个节点有两个指针, 下一个兄弟和第一个子节点（这样实际上就把一个普通的树转化为了一棵普通的二叉树）. Tries 存储一个序列集合. 每条从根节点指向叶子节点的路径都对应集合中的一个元素. 比如, 对于图中的 trie, 指定了的集合是"CAR", "CAT", 和 "COB"假定没有其它节点存在.要测定一个序列是否在集合之中, 可以从根开始, 在它的子节点中寻找序列的起始元素. 如果没有匹配的, 这个序列就不在集合中. 否则, 再在这个节点的子节点中寻找后面的元素, 以此类推. trie 有几个不错的特点。如果一个字符串在列表中，检索的时间复杂度将不超过字符串的长度乘以一个节点的的最大子节点数。此外，比起其他的来说，这个数据结构往往可以使用较少的内存，因为前缀只出现一次（在我们的例子中，虽然'CAR'和'CAT'同时存在，但是只有一个'CA'的节点出现）。在一般情况下，对于已知前缀，查找字符序列（语句，多位数的号码，等等）的问题，trie 是很好用的。

Trie 优化

一些常见的轻微改动有：

对于节点增加标记信息。如果列表里可能包含一个是其他单词前缀的词，你必须添加一个标志在每个节点上说'一个字到此为止'。如上例中，如果'CA'也作为单词出现在你的列表中，它会被标记。在链表中保持子节点有序。这增加了时间来建立树，但减少了查询时间。建立字符串节点；如果你有很多单词前缀一致但后缀独立，有时为其建立“特殊节点”更好。例如，存储“CARTING”，“COBBLER”，“CATCHING”这三个词，用“RTING”、“BBLER”、“TCHING”这三个节点显然可以节约内存。注意这同时增加了复杂性。

堆

堆（有时称为优先级队列）是一个完全二叉树，每个节点的值小于其两个孩子的值: {{./pasted_image002.png}}

堆的表示法

如果树(tree) 是一层一层地从左到右地填入的（也就是说除了树最底层以外其他层都是完整的，最底层的元素是按从左到右填入的），那么这个堆（heap）能存储为一个数组，这个数组中元素的排列顺序是从根到底层，每层从左到右。这个例子中的堆可以表示为

3 5 9 6 12 13 10 8 11

在这个表示方法中，位于 x 的节点（node）的子节点（children）位于 2x 和 2x＋1 ，(假设变址为 1)，x 的父节点（parent）是向下取整（truncate）x/2

堆中结点的插入和移动操作

将结点置于数组末端，接着交换结点与父节点的位置，直到找到一个合适的父节点。例如，插入数字 4 的过程中，堆数组（小根堆）变化如下:

3 5 9 6 12 13 10 8 11 4

3 5 9 6 4 13 10 8 11 12

3 4 9 6 5 13 10 8 11 12

删除一个结点相对来说也很简单。用数列末尾的结点取代要删除的，再进行调整：当结点的孩子比它小时，与较小的孩子交换。例如，删除数字 3 的过程：

11 5 9 6 12 13 10 8
5 11 9 6 12 13 10 8
5 6 9 11 12 13 10 8
5 6 9 8 12 13 10 11

如果我需要修改一个变量的值呢？

要把一个变量加大，则改变它的值，然后如果需要的话不断和它的父变量交换位置。T 要把一个变量的值减小，则改变它的值，然后如果需要的话和它的子变量中较小的交换位置。

堆的适用范围

对于动态的一组数询问最小值时，用堆处理十分便利。它结构紧凑，便于调整。Dijkstra 算法的堆优化就是一个很好的例子。

Heap Variations

In this representation, just the weight was kept. Usually you want more data than that, so you can either keep that data and move it (if it's small) or keep pointers to the data. Since when you want to fiddle with values, the first thing you have to do is find the location of the value you wish to alter, it's often helpful to keep that data around. (e.g., node x is represented in location 16 of the heap).

Section 2.2 Dynamic Programming 动态规划

译 by 铭(特别感谢,这是她在 N 次死机中译出来的)

作为显著减少算法运行时间（从指数的到多项式的）的一种编程技巧，动态规划是一个令人困惑的名字。它的基本思想是努力避免重复解决相同问题或子问题。如下面的问题证明了它的威力：

已知 10，000 个整数的序列（整数大于 0 小于 100，000），求最长递减子序列。注意，子序列不一定非得连续。

递归下降解法

解决此问题显然的方法是递归下降。只需找到循环和一个终止条件。看看下面的解法：

```
1 #include
2 long n, sequence[10000];
3 main () {
4 FILE *in, *out;
5 int i;
6 in = fopen ("input.txt", "r");
7 out = fopen ("output.txt", "w");
8 fscanf(in, "%ld", &n); &a mp;n bsp;
9 for (i = 0; i best) best = better;
19 }
20 }
21 return best;
22 }
```

1-9 行和 11-12 行有疑义。它们创建了一些标准变量控制输入。秘密就在于第 10 行的递归程序‘check’。‘check’程序知道它应该从哪开始查寻较小的整数，当前最长序列的长度及最小整数。由于额外调用的开销，当“开始”不超过其合适的范围，它就会“自动地”终止。‘check’程序本身很简单。它沿着列表寻找比当前‘smallest’更小的整数。如果找到了，‘check’递归调用它本身来寻找更多的整数。

同递归解法随之而来的问题是运行时间：

```
N Seconds
60 0.130
70 0.360
80 2.390
90 13.190
```

由于序列的最大长度可能会接近 6 位数，这个解法是受限的。

从末端开始

当利用向前或从前面开始的方法无法运作时，从相反的方面开始处理问题却可以收获良多。像这个例子可以先从序列的末端开始。

另外，用一点儿存贮空间来交换执行效率也会获利很多。另一种程序可能会从序列的末尾开始，用一个辅助变量来记录当前最长递减（子）序列。

考虑长度为 1 的序列从末端开始的情况。任意长度为 1 的序列都满足最长序列的所有标准。本例将‘bestsofar’数组标记为‘1’。

考虑序列的最后两个元素。如果倒数第二个数字比最后那个大，那么‘bestsofar’为 2（1+最后那个数字的‘bestsofar’）。否则为 1。

考虑在最后两个元素之前的任一元素。它总是比接近末尾的那个元素大。它的‘bestsofar’元素至少比找到的较小元素的大 1。最后‘bestsofar’的最大值就是最长递减子序列的长度。

这很明显是一个 O(N 2)算法。代码如下：

```
1 #include
2 #define MAXN 10000
3 main () {
```

```
4 long num[MAXN], bestsofar[MAXN];
5 FILE *in, *out;
6 long n, i, j, longest = 0;
7 in = fopen ("input.txt", "r");
8 out = fopen ("output.txt", "w");
9 fscanf(in, "%ld", &n);
10 for (i = 0; i = 0; i--) {
13 bestsofar[i] = 1;
14 for (j = i+1; j = bestsofar[i]) {
16 bestsofar[i] = bestsofar[j] + 1;
17 if (bestsofar[i] > longest) longest = bestsofar[i];
18 }
19 }
20 }
21 fprintf(out, "bestsofar is %d\n", longest);
22 exit(0);
23 }
```

同样地，11 行建立了末尾条件。12-20 行以‘i’循环从后计数和‘j’循环从前计数的巧妙直接的方法运行 O（N2）算法。尽管比前面的那个算法多 1 行，运行时间却更佳：

```
N Secs
1000 0.080
2000 0.240
3000 0.550
4000 0.950
5000 1.450
6000 2.080
7000 2.990
8000 3.700
9000 4.700
10000 6.330
11000 7.350
```

当 N=9000 时由于竞争，算法依就很慢。

内部循环（“寻找任意较小的数”）乞求用一些存储空间来交换时间。

一组不同的值最好存在辅助数组中。数组‘bestrun’的指针是最长子序列的长度，其值是率领子序列的第一个（也被证明是最佳的）整数。执行该数组。当遇到一个整数比在这个数组中的值大时就意味着可能会产生一个新的更长的序列。新整数可能是一个新的“序列最佳首部”，也可能不是。看下面这个序列：

10 8 9 4 6 3

从右向左浏览（从后向前），遇到 3 之后‘bestrun’数组有且只有一个元素：

0：3

继续浏览，6 比 3 大，‘bestrun’数组变成：

0:3

1:6

4 不比 6 大，但它比 3 大，所以数组变为：

0:3

1:4

9 使数组增大为：

0:3

1:4

2:9

8 同 4 使数组变成：

0:3

1:4

2:8

l0 再次扩增数组为：

0:3

1:4

2:8

3:10

于是结果出来了：4（数组中有 4 个元素）。

因为‘bestrun’数组可能比处理过的序列长度增长得慢，这个算法可能比上一个运行得要快。实际上是增速很多。代码如下：

```
1 #include
2 #define MAXN 200000
3 main () {
4 FILE *in, *out;
5 long num[MAXN], bestrun[MAXN];
6 long n, i, j, highestrun = 0;
7 in = fopen ("input.txt", "r");
8 out = fopen ("output.txt", "w");
9 fscanf(in, "%ld", &n);
10 for (i = 0; i = 0; i--) {
14 if (num[i] = 0; j--) {
19 if (num[i] > bestrun[j]) {
20 if (j == highestrun - 1 || num[i]
#define SIZE 200000
#define MAX(x,y) ((x)>(y)?(x):(y))
int best[SIZE]; // best[] holds values of the optimal sub-sequence
int
main (void) {
FILE *in = fopen ("input.txt", "r");
FILE *out = fopen ("output.txt", "w");
int i, n, k, x, sol = -1;
fscanf (in, "%d", &n); // N = how many integers to read in
for (i = 0; i x; k++)
;
best[k] = x;
sol = MAX (sol, k + 1);
}
printf ("best is %d\n", sol);
return 0;
}
```

为了不溢出，Tyler Lu 指出：

目前存在的算法使用线性查找来寻找合适的位置在‘bestrun’数组中插入一个整数。但是，因为辅助数组已分好类，我们可以用二叉查找（O（logN））。这样对于大型序列我们的运行时间就降下来了。下面是将上面代码修改后的解法：

```
#include
```

```
#define SIZE 200000
#define MAX(x,y) ((x)>(y)?(x):(y))
int best[SIZE]; // best[] holds values of the optimal sub-sequence
int
main (void) {
FILE *in = fopen ("input.txt", "r");
int i, n, k, x, sol;
int low, high;
fscanf (in, "%d", &n); // N = how many integers to read in
// read in the first integer
fscanf (in, "%d", &best[0]);
sol = 1;
for (i = 1; i = best[0]) {
k = 0;
best[0] = x;
}
else {
// use binary search instead
low = 0;
high = sol-1;
for(;;) {
k = (int) (low + high) / 2;
// go lower in the array
if(x > best[k] && x > best[k-1]) {
high = k - 1;
continue;
}
// go higher in the array
if(x best[k] && x best[k+1])
best[++k] = x;
break;
}
}
sol = MAX (sol, k + 1);
}
printf ("best is %d\n", sol);
fclose(in);
return 0;
}
```

摘要

这些程序证明了动态规划背后的主要构想：在先前找到的算法的基础上建立更大的算法。这种逐步建立的解法通常能产生速度非常快的程序。

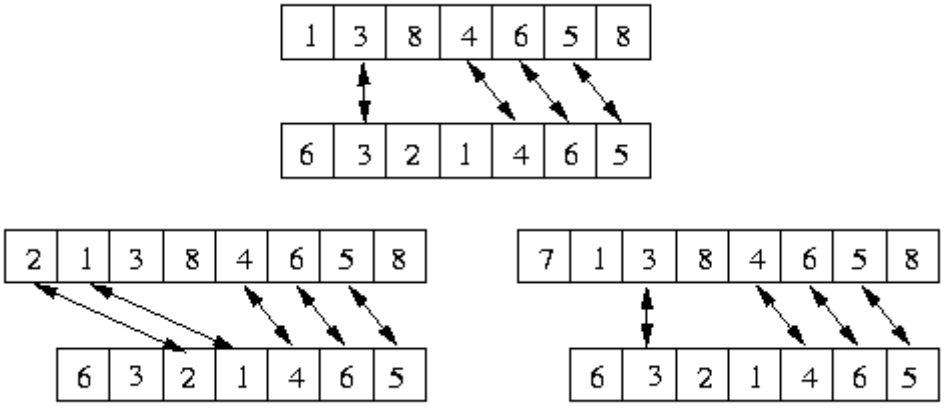
由于前面编程的挑战，主要的子问题是：在已知元素的右侧数字中找到最大的递减子序列（及其第一个值）。这种方法可解决被称为“一维的”一类问题。

二维动态规划

有时可能会创建如此的多维问题：已知两个整数序列，求二者的最长公共子序列。

这里，子问题就是较小序列（原来子序列末端部分）的最长公共子序列。首先，如果两序列之中的一个序列只含有一个元素，那解法就不重要了（或者所求元素在另一个序列中或者不在）。

考虑在第一个序列最后 i 个元素和在第二个元素最后 j 个元素中找到最长公共子序列的问题。有两种可能。第一个末尾部分的首元素可能在最大公共子序列中也可能不在。不包含第一个末尾部分首元素的最长公共序列就是第一个序列最后 i-1 个元素和第二个序列最后 j 个无素的最长公共子序列。第二个序列的末尾部分中某元素同第一个序列末尾部分的首元素一样且能找到在那些匹配元素之后元素的最长公共子序列，这导致了第二种可能。



伪代码

下面是本算法的伪代码：

```
# the tail of the second sequence is empty
1 for element = 1 to length1
2 length[element, length2+1] = 0
# the tail of the first sequence has one element
3 matchelem = 0
4 for element = length2 to 1
5 if list1[length1] = list2[element]
6 matchelem = 1
7 if matchelem = 0 then
8 length[length1,element] = 0
9 else
10 length[length1,e lement] = 1
# loop over the beginning of the tail of the first sequence
11 for loc = length1-1 to 1
12 maxlen = 0
13 for element = length2 to 1
# longest common subsequence doesn't include first element
14 if length[loc+1,element] > maxlen
15 maxlen = length[loc+1,element]
# longest common subsequence includes first element
16 if list1[loc] = list2[element]
17 &nbs p; ; length[loc+1,element+1]+1 > maxlen
18 maxlen = length[loc,element+1] + 1
19 length[loc,element] = maxlen
```

这个程序的运行时间是 O（NxM），N 和 M 分别为序列的长度。

本算法并不直接计算最长公共子序列。但是，如果给出长度矩阵，你就可以很快地确定子序列了：

```
1 location1 = 1
2 location2 = 1
```

```
3 while (length[location1,location2] != 0)
4 flag = False
5 for element = location2 to length2
6 if (list1[location1] = list2[element] AND
7 length[location1+1,element+1]+1 = length[location1,location2]
8 output (list1[location1],list2[element])
9 location2 = element + 1
10 flag = True
11 break for
12 location1 = location1 + 1
```

动态规划的技巧就是找到子问题来解决它。有时它需要多个参数：

振荡（？？？）序列是指第一部分递增且第二部分递减的序列。求一整数列的最长振荡序列（振荡可以是先增再减也可以是先减再增，但本问题只考虑先增加再减少的情况）。

这个问题的子问题是最长振荡序列和序列前缀的最长递减序列。

有时子问题被很好的隐藏起来了：

你刚赢得一场比赛，奖励是一份免费到加拿大的旅游。你必须乘飞机游玩，城市从东到西依次排列。另外，根据规则，你必须从最西侧的城市开始一直向东游览，直到你到达最东边的城市，再一直向西飞回到起点。另外每个城市参观一次（当然起点城市除外）。

已知城市顺序和乘坐的航班（你只能在某些城市间飞，因为你可以从 A 城飞到 B 城不意味着你可以向其它方向下），求你最多能参观多少个城市。

若使用动态规划应注意的是你的位置和你的方向，但重要的是你采用的路径（因为在回程时你不能再次参观某个已参观过的城市），而且路径数不能太多，否则将无法解出（并存贮）所有子问题的答案。

但是，如果不是寻找上面描述的路径，而是用另一不同的方式，那么状态数将大大减少。想像一下，有两个旅行者从最西侧的城市出发。两旅行者轮流向东走，而旅行者走的下一站总是最西侧的城市，但是两旅行者不能在同一个城市，除非他们在第一个或最后一个城市。但是，其中一个旅行者只允许使用“相反的班次”，即他可以从 A 城飞到 B 城当且仅当有一架从 B 城飞往 A 城的航班飞机。

从两个旅行者的路径中找到可以组合成一个环程旅行（用一个旅行者的正常路径飞到最东边的城市，然后再用另一个旅行者相反的路径返回最西侧的城市）并不太困难。而当旅行者 x 走了，你知道旅行者 y 除了他现在所在的城市外没参观过任何一个旅行者 x 东边的城市，否则 x 在 y 西侧时旅行者 y 已经走过一次了。于是两旅行者的路径就无法接在一起了。此算法能产生参观城市最大值的原因留作练习。

通过动态规划将问题求解

一般地，动态规划解法应用在那些可能会花费指数级时间的算法上，因此如果有界限问题在任意但输入较少的情况下太大以至于不能以指数时间级运行时，可考虑使用动态规划法。基本上，任何问题你只要考虑到用递归下降的方法，如果输入不超过 30 就可以试试是否有动态规划的解法存在。

寻找子问题

如上所述，找到子问题是进行动态规划的重要一步。你的目的是用少量的数据，如一个整数、一对整数、一个布尔数和一个整数等等，完整地描述出算法的状态。

若不失败，子问题会是一个问题的“尾端”。也就是说，有一个进行递归下降的方法这样每一步你只能处理一小部分数据。举个例子，在飞行旅游的那道题中，你可从使用递归下降的方法找到完整的路径，那也意味着你得记住你的位置和你已参观过的城市（记城列表或布尔数组的形式）。这样动态规划要做的就太多了。但是在你向东飞的一对城市问使用递归给已知常数是需要递归的非常小的一部分数据。

如果路线很重要，除非路径很短，你不要使用动态规划。但是正如飞行旅行问题，路线也许不重要，就看你怎么看了。

示例题目

多边形游戏《1998 IOI》

想像一个均匀的 N 边形。结点处放入数字，边放入操作符‘+’或‘x’。第一次去掉一条边。然后，合并边，即计算出操作符两边结点的值，将其代替原来的边和结点，举例如下：

```
...-- 3 --+-- 5 --*-- 7 --...
...----- 8 ----*----- 7 --...
...-------- 56 -----------...
```

已知一作好标记的 N 边形，求最后计算出的最大值。

子集和《春季 98 USACO》

从 1 到 N（N 在 1 到 39 之间）的连续整数的集合，可将其分成和相等的两个集合，例如，若 N=3，将集合《1，2，3》分成《3》和《1，2》两个子集合且其和相等。这计为单独划分（如，反序计为同样的划分，并不增加划分数）。

若 N=7，有 4 种方法将集合{1, 2, 3, ... 7} 分成和相等的两部分：

{1,6,7} - {2,3,4,5}

{2,5,7} - {1,3,4,6}

{3,4,7} - {1,2,5,6}

{1,2,4,7} - {3,5,6}

已知 N，打印将从 1 到 N 个整数的集合分成和相等的两个集合的各种方法。若没有则打印 0。

数字游戏《IOI96 ， maybe》

已知不超过 100 个整数（-32000_32000）列，两个对手依次轮流从数列的最左边或最后边去掉一个数。游戏到最后每个玩家的分是他去掉的数的和。已知一数列，假设第二位玩家玩得非常好，求第一位玩家赢时的最高分。

Section 2.4 Shortest Paths 最短路径

译 by tim green

Sample Problem: Overfencing [Kolstad & Schrijvers, Spring 1999 USACO Open]

农夫 John 在外面的田野上搭建了一个巨大的用栅栏围成的迷宫。幸运的是，他在迷宫的边界上留出了两段栅栏作为迷宫的出口。更幸运的是，他所建造的迷宫是一个“完美的”迷宫：即你能从迷宫中的任意一点找到一条走出迷宫的路。

给定迷宫的宽 W(1

```
+-+-+-+-+-+
|       |
+-+ +-+ + +
|   | | |
+ +-+-+ + +
| |     |
+-+ +-+-+-+
```

如上图的例子，栅栏的柱子只出现在奇数行或奇数列。每个迷宫只有两个出口。

The Abstraction

给出:一个边的权为非负的有向图

两个顶点间的一条路径是由图中相邻的边以任意的顺序连接而成的

两个顶点之间的最短路径是指图中连接这两个顶点的路径中代价最小的一条，一条路径的代价是指路径上所有边的权的和

题目常常只要求出最短路径的代价而不需要求出路径本身。在这个例子中只用求出迷宫内部的点和出口之间的最短距离(代价)。最别的，它需在求出所有代价中最大的一个。

用 Dijkstra 算法来求加权图中的最短路径

给出:所有的顶点、边、边的权，这个算法按离源点的距离从近到远的顺序来"访问"每个顶点。

开始时把所有不相邻顶点间的距离(边的权)标为无穷大,自己到自己的距离标为 0。

每次,找出一个到源点距离最近并且没有访问过的顶点 u。现在源点到这个顶点的距离就被确定为源点到这个顶点的最短距离。

考虑和 u 相邻的每个顶点 v 通过 u 到源点的距离。考查顶点 v,看这条路径是不是更优于已知的 v 到源点的路径,如果是,更新 v 的最佳路径。

[In determining the shortest path to a particular vertex, this algorithm determines all shorter paths from the sou rce vertex as well since no more work is required to calculate all shortest paths from a single source to vertic es in a graph.]

参考: [Cormen, Leiserson, Rivest]的第 25 章

Pseudocode:

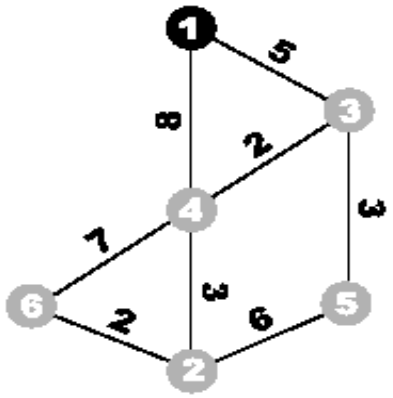
```
# distance(j) is distance from source vertex to vertex j
# parent(j) is the vertex that precedes vertex j in any shortest path
# (to reconstruct the path subsequently)
1 For all nodes i
2 distance(i) = infinity # not reachable yet
3 visited(i) = False
4 parent(i) = nil # no path to vertex yet
5 distance(source) = 0 # source -> source is start of all paths
6 parent(source) = nil
7 8 while (nodesvisited
```

这个算法的时间效率为 O(V2)。你可以使用堆来确定下一个要访问的顶点来获得 O(E log V)的效率(这里的 E 是指边数 V 是指顶点数)，但这样做会使程序变得复杂，并且在一个大而稀疏的图中效率只能提高一点点。

Sample Algorithm Execution

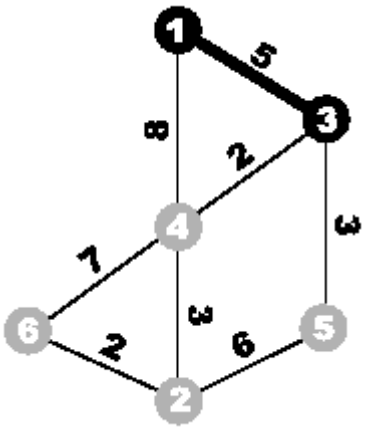
考虑下面的图

这是问题的初始状态:



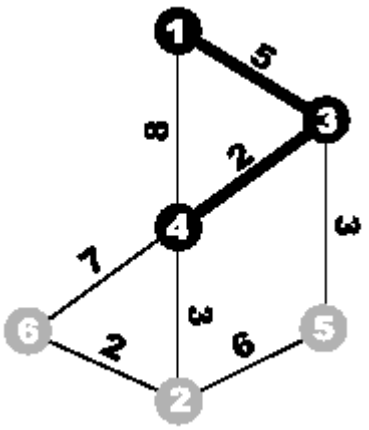
和顶点 1 相邻的有顶点 3、4

现在还没被访问的顶点中顶点 3 到源点的距离最小，所以它是下一个要被访问的点:

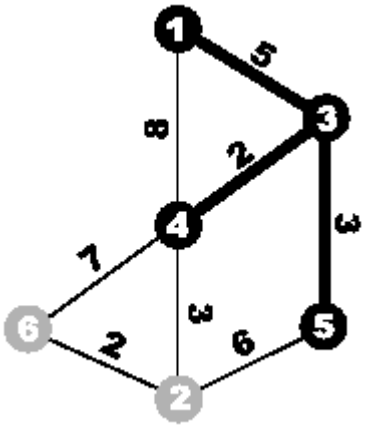


和顶点 3 相邻的顶点有 4、5 更新这些顶点:

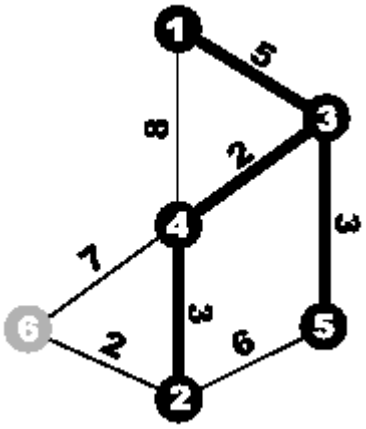
现在还没被访问的顶点中顶点 4 到源点的距离最小，它的邻点有 1、2、3、6, 因为别的点都已经被访问过了，所以只有顶点 2、6 需要被更新:



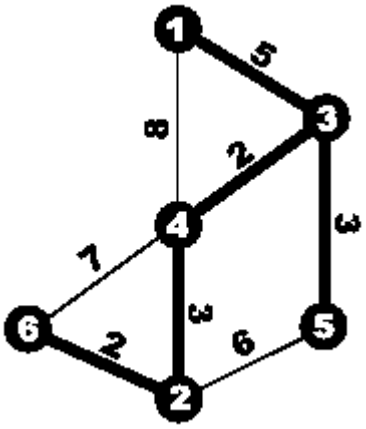
在剩下的三个顶点(2, 5, 和 6)中, 顶点 5 最接近源点，所以要被访问。它的邻点有 2、3,只有顶点 2 没有被访问过。经顶点 5 再到顶点 2 的距离是 14,比经顶点 4 的路径长(那条长为 10),所以顶点 2 不需要更新。



在剩下的两个顶点中最接近的是顶点 2 ，它的邻点是顶点 4、5、6,只有顶点 6 没被访问过。另外顶点 6 要被更新:



最后，只有顶点 6 没被访问过，它所有的邻点都被访问过了:



Sample Problem: Package Delivery 投递包裹

给出一个地点的集合，和连接它们的道路的长，和一个按排好的包裹要投递的地点的清单。找出按顺序投递所有包裹的最短路程。

分析:对于每一次投递，使用 Dijkstra 算法来确定两个点间的最短路程。如果要投递的次数超过 N，我们就用计算每对点之间的最短距离来代替计算每投递的最短路程，最后只用简单把每一次投递一路程相加就是答案。

Extended Problem: All Pairs, Shortest Paths 每对点之间的最短路程

这个扩展问题就是确定一个表格 a：

这里 ai,j = 顶点 i 和 j 之间的最短距离，或者无穷大如果顶点 i 和 j 无法连通。

这个问题常常是其它大的问题的子问题，如投递包裹。

Dijkstra 算法确定单源最短路径的效率为 O(N2)。我们在每个顶点上执行一次的效率为 O(N3)。

如果不要求输出路径，还有一个更简单的算法效率也是 O(N3)

The Floyd-Warshall Algorithm

Floyd-Warshall 算法用来找出每对点之间的最短距离。它需要用邻接矩阵来储存边，这个算法通过考虑最佳子路径来得到最佳路径。

注意单独一条边的路径也不一定是最佳路径。

从任意一条单边路径开始。一个两点之间的距离是边的权，或者无穷大如呆两点之间没有国相连。
对于每一对顶点 u 和 v，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比已知的路径更短。如果是更新它。

不可思议的是，只要按排适当，就能得到结果。

想知道更多关于这个算法是如何工作的，请参考[Cormen, Leiserson, Rivest]第 26 章。

```
Pseudocode:
# dist(i,j) is "best" distance so far from vertex i to vertex j
# Start with all single edge paths.
For i = 1 to n do
  For j = 1 to n do
    dist(i,j) = weight(i,j)
For k = 1 to n do # k is the `intermediate' vertex
  For i = 1 to n do
    For j = 1 to n do
        if (dist(i,k) + dist(k,j)
```

这个算法的效率是 O(V3)。它需要邻接矩阵来储存图。

这个算法很容易实现，只要几行。

即使问题是求单源最短路径，还是推荐使用这个算法，如果时间和空间允许(只要有放的下邻接矩阵的空间，时间上就没问题)。

Problem Cues

如果问题要求最佳路径或最短路程，这当然是一个最短路径问题。即使问题中没有给出一个图，如果问题要求最小的代价并且这里没有很多的状态，这时常常可以构造一个图。最大的一点要注意的是:最短路径 = 搜索做某件事的最小代价。

Extensions

如果图是不加权的，最短路径包括最少的边。这种情况下用宽优搜索(BFS)就可以了。如果图中有很多顶点而边很少，这样会比 Dijkstra 算法要快(看例子 Amazing Barn)

如果充许负权边，那么 Dijkstra 算法就错了。幸运的是只要图中没有负权回路 Floyd-Warshall 算法就不受影响(如果有负权回路，就可以多走几圈得到更小的代价)。所以在执行算法之前必须要检查一下图。

可以再增加一些条件来确定最短路径(如，边少的路径比较短)。只要用从距离函数中得到比较函数，问题就是一样的。在上面的例子中距离函数包括两个值 代价和边数。两个值都要比较如果必要的话。

Sample Problems

Graph diameter 图的直径

给出:一个无向无权的连通图,找出两点距离最远的顶点。

分析:找出所有点之间的最短距离，再找出最大的一个。

Knight moves 爵士的移动

给出:两个 N*N 的棋盘.确定一个爵士从一个棋盘移动到另一个所需的最少步数。

分析:把棋盘变成一个 64 个顶点的图。每个顶点有 8 条边表示爵士的移动。

Amazing Barn (abridged) [USACO Competition Round 1996]

考虑一个非常大的谷仓由 N(N "最中心的畜栏",一个畜栏到其它的平均距离最小则被认为是"最中心的畜栏"的。

分析:计算每对顶点间的最短距离来确定平均距离。任何 O(N3)的算法在 N=2500 的情况下都是不行的。注意到图中的边比较少(一个 4 条边)，用 BFS 加上队列是可行的，一次 BFS 的较率为 O(E)，所以 2500 x 10,000 = 2.5 x 106 比 25003 = 1.56 x 1010 要好的多。

Section 3.1 Minimal Spanning Trees 最小生成树(MST)

译 by Lucky Crazy & Felicia Crazy

例题：最短网络 [Russ Cox, Winter 1999 USACO Open] 农民约翰被选为他们镇的镇长！他其中一个竞选承诺就是在镇上建立起互联网，并连接到所有的农场。当然，他需要你的帮助。约翰已经给他的农场安排了一条高速的网络线路，他想把这条线路共享给其他农场。为了用最小的消费，他想铺设最短的光纤去连接所有的农场。你将得到一份各农场之间连接费用的列表，你必须找出能连接所有农场并所用光纤最短的方案。每两个农场间的距离不会超过 100000 数学模型：

给出: 一幅连通的有权无向图。

一棵生成树的图是一张无向的连通图(也就是图中任意一对节点都是连通的).

而一棵最小生成树是一棵权和最小的生成树(即树所有的权值和最小). 利用 Prim 算法建造最小生成树：

给出: 所有节点和权值的列表。

利用贪心算法建造最小生成树，即在每次都将已有的生成树相连的一个权值最小的节点连入树中。

从某一个节点开始生成。

确定所有不在树中结点连接到树中所需的最小权值（如下例中的 distance），并记录下与之相连的结点（如下例中的 source)。 如果有结点无法连入树中，那么假设它的权值为无穷大 （比图中所有的权值都大）.

在每一步中,找出一个和生成树相连的权值最小的节点（即在 distance 栏中）并且用一条边将它连入树中。（由于最小生成树的性质，它必定是不含圈的。否则断开这个圈仍然可以保证每个节点都在树中——译者）

如果你想知道详细的分析或该算法的正确性证明，请参考《Cormen, Leiserson, Rivest》的第 24 章。

伪代码:

```
 # distance(j) is distance from tree to node j
 # source(j) is which node of so-far connected MST
 #                   is closest to node j
1   For all nodes i
2     distance(i) = infinity       # no connections
3     intree(i) = False           # no nodes in tree
4     source(i) = nil
5   treesize = 1                 # add node 1 to tree
6   treecost = 0
7   intree(1) = True
8   For all neighbors j of node 1   # update distances
9      distance(j) = weight(1,j)
10 source(j) = 1
11 while (treesize < graphsize)
12 find node with minimum distance to tree; call it node i
13 assert (distance(i) != infinity, "Graph Is Not Connected")
   # add edge source(i),i to MST
14 treesize = treesize + 1
15 treecost = treecost + distance(i)
16 intree(i) = True # mark node i as in tree
   # update distance after node i added
17 for all neighbors j of node i
18 if (distance(j) > weight(i,j))
19 distance(j) = weight(i,j)
20 source(j) = i
```

该算法的时间复杂度为 O(N2)。使用堆可以得到 O(N log N)的复杂度。图例：

考虑下图的权，边情况:

目标：生成最小生成树。 该算法将在 1）节点开始，它与节点 2），6），3）相连，权值情况如下：

Node distance intree source 1 infinity True nil 2 30 False 1 3 20 False 1 6 25 False 1

已知不存在权为无穷大的情况。（intree=False & source=nil）

可连接的最小权为 20, 所以节点 3）被连入树中：

Node distance intree source 1 infinity True nil 2 9 False 3 3 20 True 1 6 25 False 1 7 7 False 3

注意：节点 3）现在已经“在树中”了，节点 2）的权已经从 20 变成的 9，且 source 也变成的 3。

可连接的最小权为 7, 所以节点 3）和 7）被连接:

Node distance intree source 1 infinity True nil 2 9 False 3 3 20 True 1 6 10 False 7 7 7 True 3

可连接的最小权为节点 2）的 9。 连接节点 3)和 2)：

Node distance intree source 1 infinity True nil 2 9 True 3 3 20 True 1 4 21 False 2 5 9 False 2 6 10 False 7 7 7 True 3

连接节点 2）和 5)：

Node distance intree source 1 infinity True nil 2 9 True 3 3 20 True 1 4 8 False 5 5 9 True 2 6 10 False 7 7 7 True 3 8 12 False 5

下一步连接节点 5) 和 4）:

Node distance intree source 1 infinity True nil 2 9 True 3 3 20 True 1 4 8 True 5 5 9 True 2 6 10 False 7 7 7 True 3 8 12 False 5 连接节点 6）和 7）:

Node distance intree source 1 infinity True nil 2 9 True 3 3 20 True 1 4 8 True 5 5 9 True 2 6 10 True 7 7 7 True 3 8 11 False 6

最后，连接节点 6）和 8)：

Node distance intree source 1 infinity True nil 2 9 True 3 3 20 True 1 4 8 True 5 5 9 True 2 6 10 True 7 7 7 True 3 8 11 True 6

最小生成树完成。注意：

必须知道某些情况下某些结点无法连入树中，应避免重复计算这些结点。（即 intree=False & source=nil，在例中没有该情况）。题型提示：

如果某些问题需要一张最优的连通图，并且需要用以一个最小的花费来连接该系统或该系统的任意两个部分，这样的问题就极类似最小生成树问题。

拓展：

如果你的生成树有任何约束条件的话（任意两个结点可能离得非常远，或者平均距离必须最小），这个算法就玩完了，而且让程序适应这样的约束条件非常困难。

显而易见，任意两个结点之间不能有多边（你就留下权值最小的边，忽略其余的边）。

Prim 算法无法扩展到有向图（如果你想要的是强连通图的话）。例题：包裹寄送

给出：一些城市的位置，和轮船公司连接每对城市的航线的花费。找出使得一个包裹能够从任意一座城市送到任意的另外一座城市的花费最小。高速公路建设

当然，为了经济效益，他们想要花最少的钱来做这件事。高速公路的花费正比于它的长度。给出 L.S. 州的所有城市的 x,y 坐标，设计使得所有城市互相连通的最便宜的建造方案。 Bovile 电话(已删节) [USACO Training Camp 1998, Contest 2]

给出：一些奶牛和田野中的干草堆（奶牛和干草堆在一起），连接任意的位置需要一定的花费。只用干草堆和奶牛，计算哪些干草堆应该包含在干草堆网络中，并且使总花费最小。

分析：对于每一组可能的干草堆（也就是，共有 2 n 组），计算这组干草堆和奶牛的最小生成树。计算最小生成树的组合，使得花费最小。

Section 3.2 Knapsack Problems 背包问题

译 by 铭

（在中国，背包问题一般是这样描述的：设 n 个重量为（W1,W2,...Wn）的物品和一个载重为 S 的背包，将物品的一部分 xi 放进背包中的利润是 Pixi，问如何选择物品的种类和数量，使得背包装满而获得最大的利润？另有一简化版本说：设有一个背包可以放入的物品重量为 S，现有 n 件物品，重量分别为 W1,W2,...Wn。问能否从这 n 件物品中选择若干件放入此背包，使得放入的重量之和正好为 S。--译者加，不知道有没有班门弄斧之嫌）

前提

●贪心法（它是一种多步决策法，它总是作出在当前看来是最好的选择，它的考虑不是从整体出发，而只是某种意义上的局部最优，这样贪心法不能对所有问题达到整体最优解，但是对相当范围的许多问题都能够产生整体最优解。--译者）

●动态规划（它是将问题进行逐步的划分来缩小问题的规模，直到可以求出子问题的解为止。分划子问题后，对应的子问题中含有大量的重复，这样就将重复地求解；在第一次遇到重复时把它解决，并将解保存起来，以备后面引用。动态规划法常用来求一个问题在某种意义下的最优解。--译者）

●递归下降

示例问题：用录音带录音

农场主约翰最喜欢的爱好是制作一个 Bessie 喜欢的音乐合集磁带以便它在产奶时听。Bessie 的产奶量取决于它产奶时所听的歌曲。

已知一组歌曲（每首歌都由一对整数－－此曲的长度（以秒计），听该首歌时的产奶量来表示）以及给挤奶的总时间。找到这样一组歌曲的集合，使得歌曲的总长度不超过给 Bessie 挤奶的总时间且使 Bessie 的产奶量达到最大。

抽象描述

已知一组物品--每个都有其尺寸和值（比如，重量），以及可用的总空间。找到这样一个集合，使得该集合的值的和最大，且其尺寸的和受某些限制所约束。集合中任何一个特定的项目的总数目/尺寸不能超过它的可利用率。

解题想法

视其为背包问题的一般方法是一个容量受限的背包使得放入其中的物品的值达到最大。

以上述问题为例，Bessie 产奶时听的音乐带就是“背包”，而那些歌就是“放入背包中的物品”。

三个背包问题

背包问题有三种形式：

●小数背包问题

允许将小数表示的物品放入背包中的是小数背包问题。举例来说，如果物品是原油、飞机燃料、煤油而你的背包是一只水桶，取 0.473 升的原油，0,263 升的飞机燃料和 0,264 升的煤油就是有意义的。这是形式最简单的要解决的背包问题。

●整数背包问题

整数背包问题中，只有完整的物品能放入背包里。此形式的一个例子就是：部分的曲子不允许放入包中。

●多重背包问题

在多重背包问题中，需被填充的背包多于一个。如果允许有小数的物品放入，也就等于有一个大的背包，其容量相当于所有可用背包的和。因此，此术语只用来指多重整数背包的情况。

小数背包问题

小数背包问题是三者中最简单的，其贪婪解法如下：

●找到“值密度”（物品值/尺寸）最大的物品

●如果总容量仍就超过物品的可利用率，把所有满足条件的物品放入背包中，然后反复执行。

●如果总容量少于物品的可利用率，尽可能多的使用可用空间，然后终止。

●由于这个算法必须先按照值密度把物品分类，然后以降序将它们放入背包，直至容量用完，该算法以 N l og N 级运行。通常简单些的方法不是将它们分类，而是不停地找每次不用的最大值密度，这种算法的时间复杂度是 O(N 2) 。

注意：对于这类问题，因为你可以做一个微小的变换使得所有的物品尺寸大小为一，且原始尺寸大小和可利用率（当然用原始尺寸大小除值）的乘积就是总容量，同时有尺寸和可利用率是很少见的。
延伸：在这种情况下，物品的值和可利用率可以是实数。用这种算法处理有小数的尺寸大小也不是问题。

整数背包问题

这个问题有点难度，但是如果背包足够小，使用动态规划，它还是可解的。

●依据背包大小的最大值设计动态的程序。

●刷新用来表示大小为 S 的物品的数组，颠倒其次序，看将当前物品放入大小为 K 的背包中所产生的集合是否比当前最好的大小为 K+S 的背包更符合条件。

●这个算法运行 K*N 次，其中 K 是背包的大小，N 是物品的可利用率义之和。

●如果背包太大了以至于无法分配此数组，递归下降是一种选择，即这个问题是 NP 完全的（给定 I 上的一个语言 L，如果有一架非确定图灵机 M 和一个多项式 P(n)，对任何 I 上的长度为 n 的串 w，M 都可以在 P(n)步内确定是否接受 w，则称 L 是非确定图灵机下多项式时间复杂性问题，简记为 NP 问题/语言。若 L 是属于 NP 的，且对 NP 中的每一个语言 L'，都存在一个从 L'至 L 的多项式时间转化，我们说 L 是 NP 完整的。--译者）。当然，递归下降在以小的物品填充的大背包情况下可以运行相当长的一段时间。

延伸：

●小数的值不是问题；数组可以用实数数组来代替整数数组。小数的可利用率并不影响什么，在没有大量损失的条件下，缩短数字（如果你有 3,5 个物品，你可以仅用 3）。

●小数的尺寸是个讨厌的东西，它使得问题递归下降。

●如果尺寸都相同，问题就能贪婪地解开，在下降的值排序中选择物品，直到背包满为止。

●如果值都是 1.0，同样地使用贪心法，在上升的尺寸大小排序中选择物品，直到背包满为止。

多重背包问题

对于任何大小的多重背包，状态空间太大了以至于无法使用从整数背包算法中来的 DP 解法。于是递归下降是解决这个问题的方法。

延伸：

●用递归下降，通常扩展就简单了。小数的尺寸和值就不是问题了，同样地值的计算功能也不是问题。

●如果值都是同一个，那么如果能被放入所有背包中的物品的最大值是 n，则存在使用 n 个最小物品的解法。它能大大减少查找时间。

示例问题

分数膨胀[1998 USACO National Championship]

你正试图设计一个有最高分数(<10,000)的比赛。已知比赛长度，一组问题，问题的长度以及每个问题的分值，计算满足长度约束的最高分数的比赛。

分析：这是一个整数背包问题，比赛是背包，尺寸是问题的长度，值是分数值。背包（比赛）尺寸的限制是其足够小使得解法在存储器中运行。

篱笆栏[1999 USACO Spring Open]

农场主约翰准备在他的领地建一圈篱笆。他已装好了柱子，所以他知道所要的围栏长度。当地的木材店有各种长度的木板（至多 50 个）。已知木材店木板的长度，约翰要的围栏长度，计算约翰建篱笆所用的围栏最大值。

分析：这是个多重背包问题，木材店的木板是背包，物品是约翰用的围栏。物品的尺寸就是长度，值是一。由于值都是一，如果存在用任意 K 个围栏的解法，则有用 K 个最小围栏的解法。

装满你的油箱

你在 Beaver 郡中部一百英里有一个加油站的城市中，想将你的油箱装满好能到达 Rita Blanca。幸运地是，这个小镇有两三个加油站，但它们的油都好像要用光了。已知每个加油站的油价，每个加油站的油量，计算为了花最少的钱，应该从每个加油站买多少汽油。

分析：这是一个小数背包问题，背包是油箱，物品是汽油。

Section 3.3 Eulerian Tour 欧拉通路

by 多维数组

例题：Riding the Fences

农夫 John 拥有许多篱笆，他需要定期检查他们是否完整。农夫 John 有一个表，上面记有所有交叉点与各个篱笆的端点，他用这个表来得到它的篱笆的地图。每个篱笆有两个端点，每个端点都在交叉点上。然而有的交叉点可能只与一个篱笆相连。当然，有两个以上的篱笆可能共有同一个端点。现在给你农夫 John 的表，计算是否存在一条路，能使农夫 John 骑马经过他所有的篱笆且每个篱笆只经过一次。农夫 John 可在任何位置出发或结束，但他不能穿越他的农场。请问是否存在这样的一条路径。若存在，请找出它。

抽象模型

现在给你一副无向图。寻找一条包含所有边的路径，其中每一条边只经过一次。这被叫做欧拉通路。若这条路径的起点与终点为同一点，则为欧拉回路。

算法

判定一个图是否存在欧拉通路或欧拉回路比较容易，这里提供两种不同的判定法则。定理 1：一个图有欧拉回路当且仅当它是连通的（即不包括 0 度的结点）且每个结点都有偶数度。定理 2：一个图有欧拉通路当且仅当它是连通的且除两个结点外，其他结点都有偶数度。定理 3：在定理 2 的条件下，含奇数度的两个结点中，一个必为欧拉通路的起点，另一个必为终点。

此算法的基本思想是从图中的某个结点出发求出一个能回到该结点的路径。现在，该路径已被加入（根据它的结果，逆序存放）。检验在该路径中的所有结点的每条边是否已被应用。若某个结点存在未被应用的边，则算法进一步找出一条从此结点的这条边开始的新回路，并将这个新的回路连入原路径中。算法一直运行到最初的路径中的所有结点的每一条边都已被应用。既然改图是连通的，这也就表明图中所有的边都已经被应用，所以最终结果也就是欧拉回路。

更加正式的说，寻找图的欧拉回路的方法是选取一个起点，并对它进行递归，递归的步骤为：

- 选取一个起点并在此结点上，按以下步骤递归：
  - 若此结点无邻结点，则添加该结点到回路中并退出。
  - 若此结点有邻结点，则生成一个邻结点表并对表中的结点进行操作（遍历一个点，删除一个点）直到列表为空。最后将处理的结点加入回路中。
- 对表中结点的操作为：删去当前处理的结点（指的是当前递归下的结点）与表中的结点相连的边，然后对这个结点（指表中的结点，我觉得好难表述，建议直接看伪代码.....）进行递归。

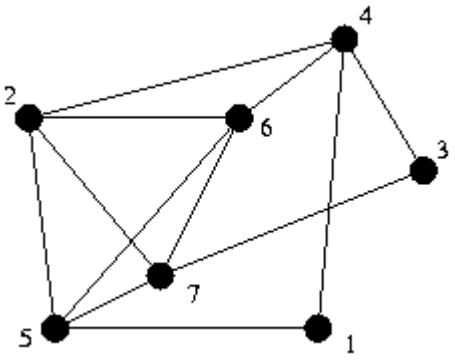
以下是伪代码：

```
# circuit is a global array
   find_euler_circuit
     circuitpos = 0
     find_circuit(node 1)

# nextnode and visited is a local array
# the path will be found in reverse order
  find_circuit(node i)

    if node i has no neighbors then
      circuit(circuitpos) = node i
      circuitpos = circuitpos + 1
    else
      while (node i has neighbors)
          pick a random neighbor node j of node i
          delete_edges (node j, node i)
```

```
        find_circuit (node j)
    circuit(circuitpos) = node i
    circuitpos = circuitpos + 1
```

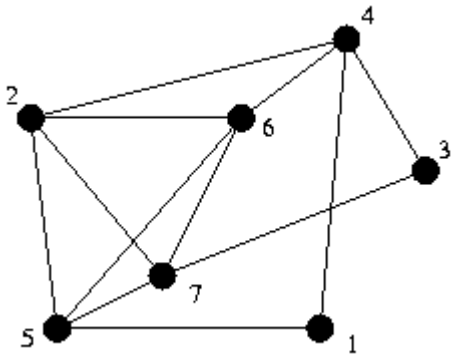
寻找欧拉通路的方法是先找到一个含有奇数度的节点，然后以它为参数调用 find_circuit。如果你用邻接表存储图，这两个算法的时间复杂度均为 O(m+n)，其中 m 为边的个数、n 为结点个数。若图非常的大，则很容易栈溢出。所以你应该自己建栈。

运行实例

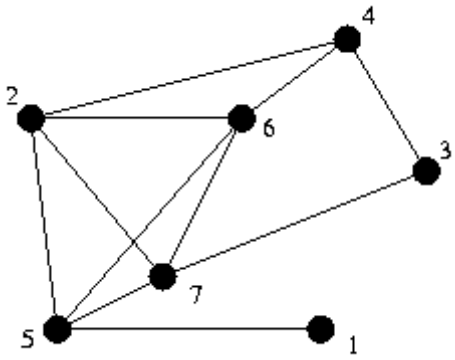


考虑以下的图

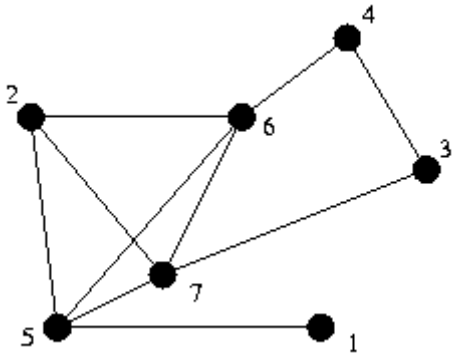
假设先选择编号最小的邻结点：



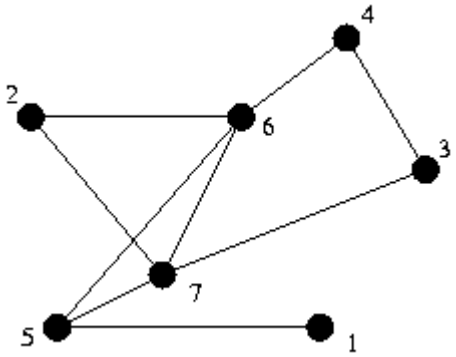
Stack: Location: 1 Circuit:



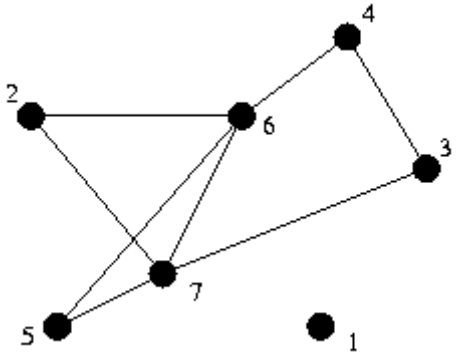
Stack: 1 Location: 4 Circuit:



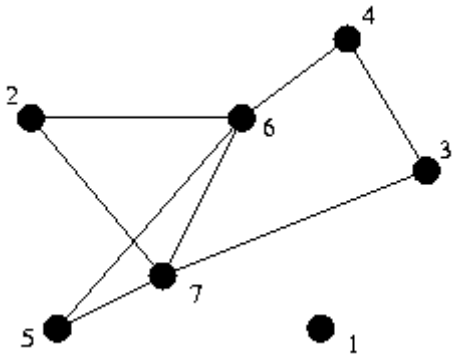
Stack: 1 4 Location: 2 Circuit:



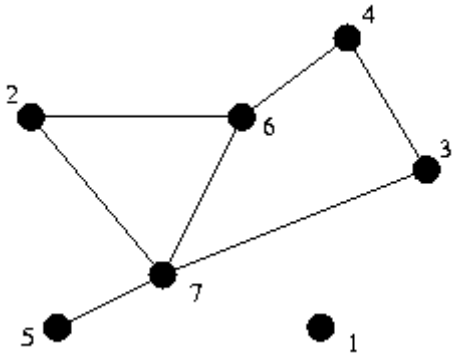
Stack: 1 4 2 Location: 5 Circuit:



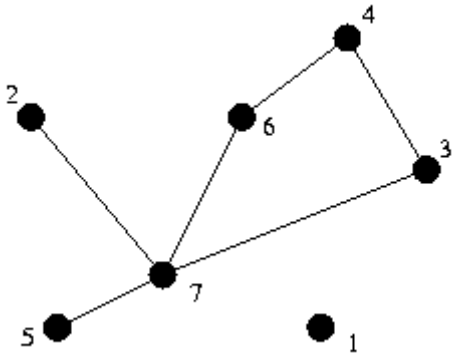
Stack: 1 4 2 5 Location: 1 Circuit:



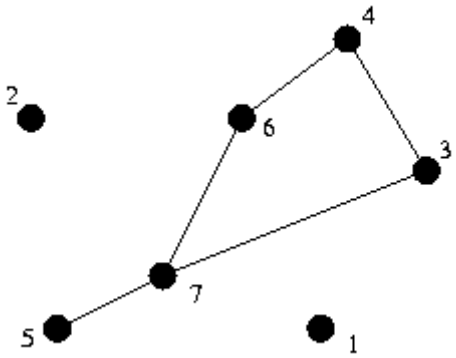
Stack: 1 4 2 Location: 5 Circuit: 1



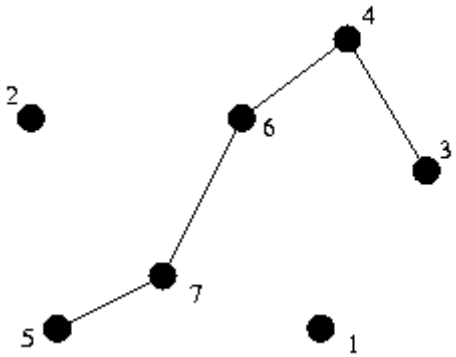
Stack: 1 4 2 5 Location: 6 Circuit: 1



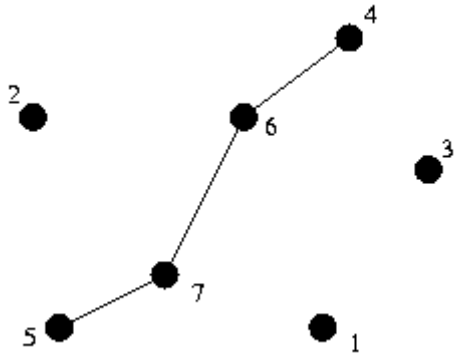
Stack: 1 4 2 5 6 Location: 2 Circuit: 1



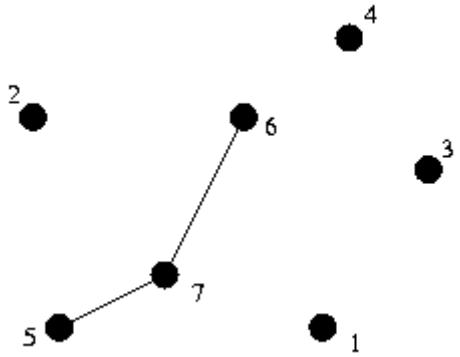
Stack: 1 4 2 5 6 2 Location: 7 Circuit: 1



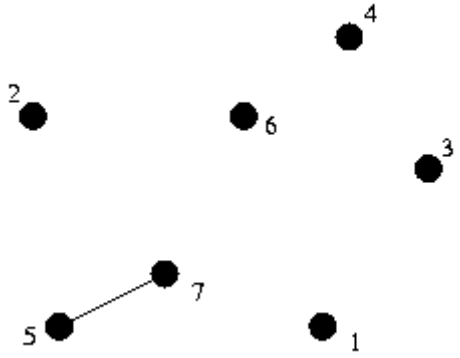
Stack: 1 4 2 5 6 2 7 Location: 3 Circuit: 1



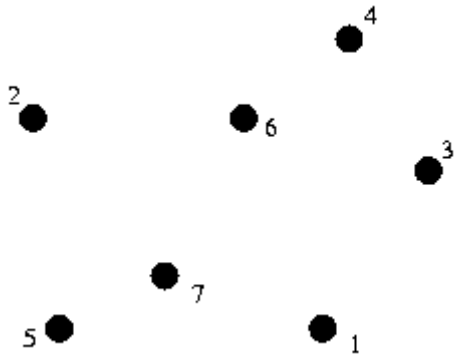
Stack: 1 4 2 5 6 2 7 3 Location: 4 Circuit: 1



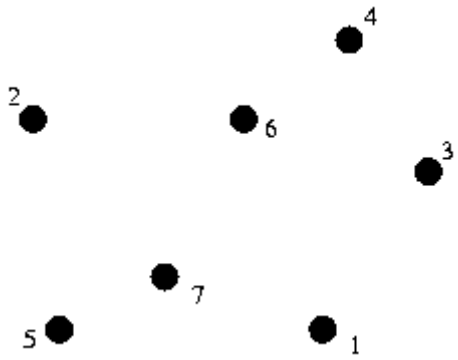
Stack: 1 4 2 5 6 2 7 3 4 Location: 6 Circuit: 1



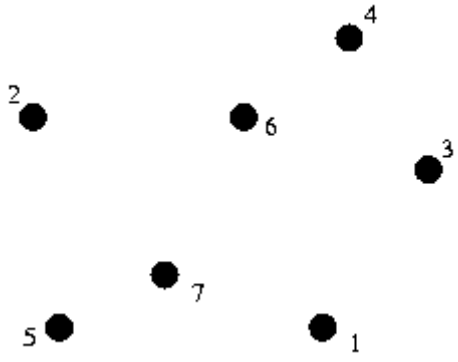
Stack: 1 4 2 5 6 2 7 3 4 6 Location: 7 Circuit: 1



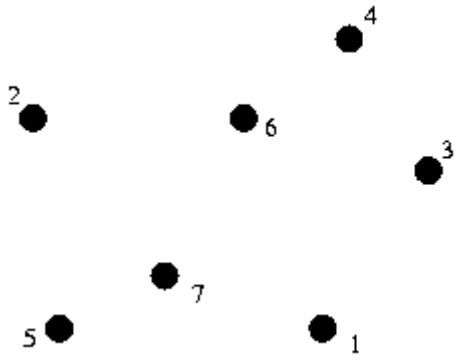
Stack: 1 4 2 5 6 2 7 3 4 6 7 Location: 5 Circuit: 1



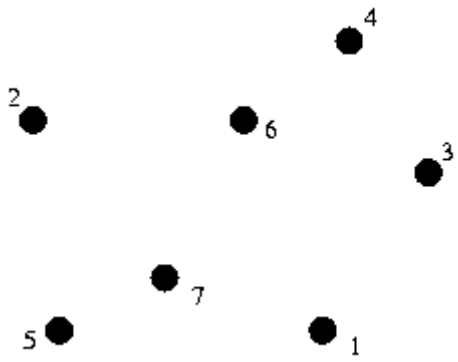
Stack: 1 4 2 5 6 2 7 3 4 6 Location: 7 Circuit: 1 5



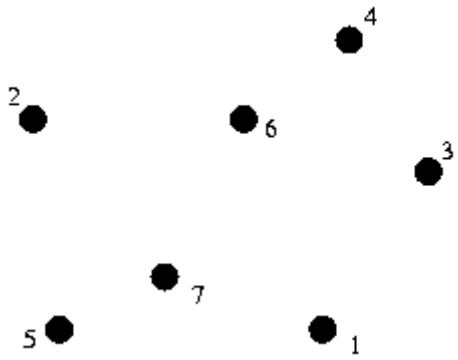
Stack: 1 4 2 5 6 2 7 3 4 Location: 6 Circuit: 1 5 7



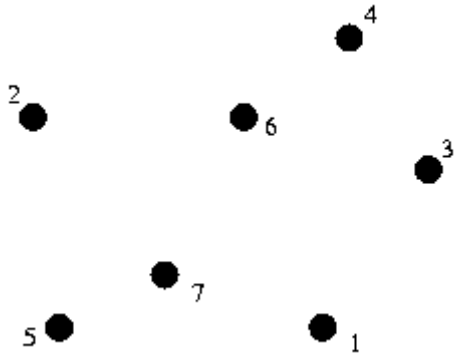
Stack: 1 4 2 5 6 2 7 3 Location: 4 Circuit: 1 5 7 6



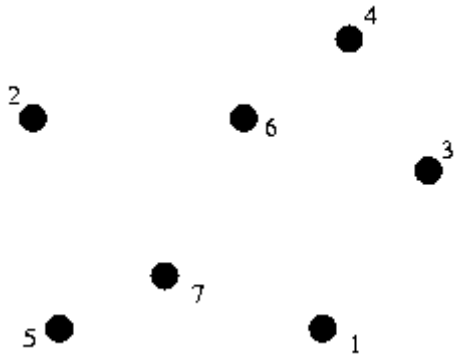
Stack: 1 4 2 5 6 2 7 Location: 3 Circuit: 1 5 7 6 4



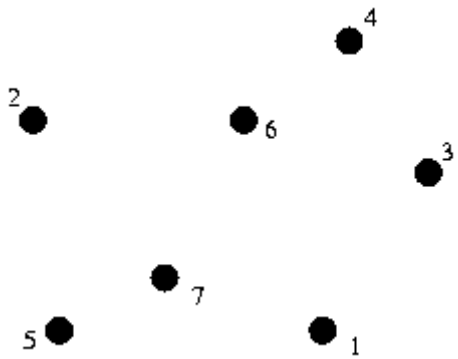
Stack: 1 4 2 5 6 2 Location: 7 Circuit: 1 5 7 6 4 3



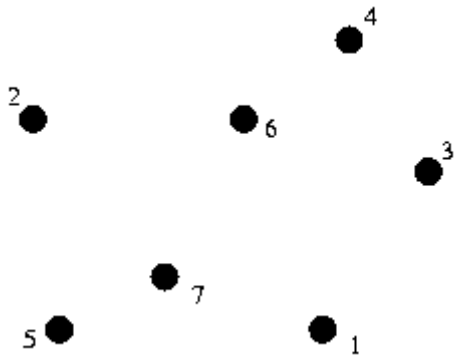
Stack: 1 4 2 5 6 Location: 2 Circuit: 1 5 7 6 4 3 7



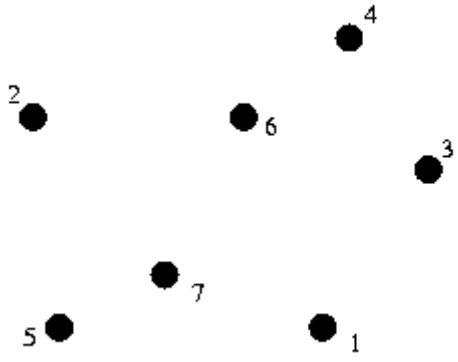
Stack: 1 4 2 5 Location: 6 Circuit: 1 5 7 6 4 3 7 2



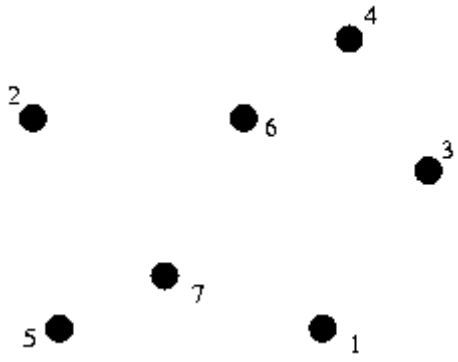
Stack: 1 4 2 Location: 5 Circuit: 1 5 7 6 4 3 7 2 6



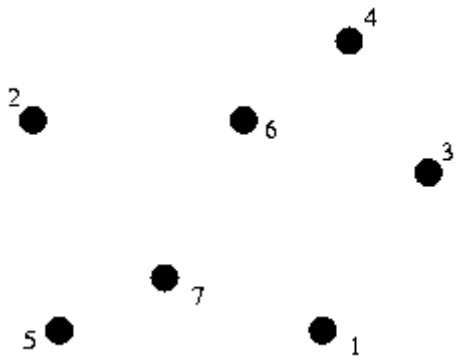
Stack: 1 4 Location: 2 Circuit: 1 5 7 6 4 3 7 2 6 5



Stack: 1 Location: 4 Circuit: 1 5 7 6 4 3 7 2 6 5 2



Stack: Location: 1 Circuit: 1 5 7 6 4 3 7 2 6 5 2 4



Stack: Location: Circuit: 1 5 7 6 4 3 7 2 6 5 2 4 1

扩展知识

结点间的重边能用相同的算法解决。自环也能被同样的算法解决。把它们看成给结点加上 2（一进一出）的度。

当一个有向图是强连通图（除了入度和出度均为 0 的结点），并且每个点的入度和出度相等时，其中存在一条欧拉回路。算法也是一样的，只是由于它是逆序找到的，你需要反向遍历它。在有向图中寻找一条欧拉回路更难一些。如果你感兴趣，可以查阅 Sedgewick [(其实就是《Algorithms in ……》那一系列书)]的相关资料。

练习 Airplane Hopping 给出一系列城市，以及城市间的航班。请找出一条路线能搭乘所有的航班并再次回到出发城市。
解析：这等价于在有向图中寻找欧拉回路。

(以下待翻译) Analysis: This is equivalent to finding a Eulerian circuit in a directed graph. Cows on Parade Farmer John has two types of cows: black Angus and white Jerseys. While marching 19 of their cows to market the other day, John's wife Farmeress Joanne, noticed that all 16 possibilities of four successive black and white cows (e.g., bbbb, bbbw, bbwb, bbww, ..., wwww) were present. Of course, some of the combinations overlapped others.
Given N (2 <= N <= 15), find the minimum length sequence of cows such that every combination of N successive black and white cows occurs in that sequence.
Analysis: The vertices of the graph are the possibilities of N-1 cows. Being at a node corresponds to the last N-1 cows matching the node in color. That is, for N = 4, if the last 3 cows were wbw, then you are at the wbw node. Each node has out-degree of 2, corresponding to adding a black or white cow to the end of the sequence. In addition, each node has in-degree of 2, corresponding to whether the cow just before the last N-1 cows is black or white.
The graph is strongly connected, and the in-degree of each node equals its out-degree, so the graph has a Eulerian circuit.
The sequence corresponding to the Eulerian circuit is the sequence of N-1 cows of the first node in the circuit, followed by cows corresponding to the color of the edge.

Section 3.4 Computational Geometry 计算几何

译 By SuperBrother

知识准备

- 图论
- 最短路

操作

这个模块讨论几个计算某些几何问题的算法，这些算法大多基于下列两个操作：叉积和反正切。

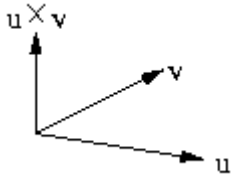
叉积

u 和 v 的叉积被表示成 u x v。两个**三维的向量** u,v 的叉积是下列行列式（**i,j,k 为 x,y,z 轴上单位向量**）：

```
| i  j  k  |
| Ux Uy Uz |
| Vx Vy Vz |
```

即:

```
(Uy*Vz-Vy*Uz)i + (Uz*Vx-Ux*Vz)j + (Ux*Vy-Uy*Vx)k
```



z 分量为 0 时，上面式子就化为 2 维的两向量叉积了。结果只有 z 分量。

即：

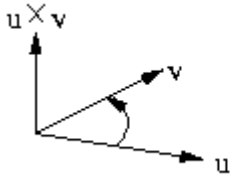
```
| Ux Uy |
| Vx Vy |
Ux*Vy-Uy*Vx
```

（注意！二维的叉积较为猥琐。。其实叉积最早就是定义在三维空间内的，当 z=0 时的特殊情形才是二维向量积。“二维向量”表达式求出的是一个超出平面的向量。。这个向量的方向我们不关心，但**有时又要利用它**。。。（例如求多边形面积）故上式不甚严格）

叉积有 3 个特点：

- 两个向量的叉积是一个与这两个向量同时垂直的向量。
- 叉积的大小等于下面 3 个量的乘积:
  - u 的大小
  - v 的大小
  - u,v 夹角的正弦。

当然与 u,v 同时垂直的向量有两个方向，叉积的方向取决于 u 在 v 的右边还是在 v 的左边。



点积

点积的值由以下三个值确定：

- u 的大小
- v 的大小

- u,v 夹角的余弦。

在 u,v 非零的前提下，点积如果为负，则 u,v 形成的角大于 90 度；如果为零，那么 u,v 垂直；如果为正，那么 u,v 形成的角为锐角。

反正切

反正切函数对于一个给定的正切值，返回一个在-pi/2 到 pi/2 之间的角（即-90 度至+90 度）。C 中另外有一个函数 atan2，给出 y 分量和 x 分量（注意顺序!），计算向量与 x 正半轴的夹角，在-pi 到 pi 之间。它的优点就是不需担心被 0 除，也不需为了处理 x 为负的情况而写代码修改角。atan2 函数几乎比普通的 atan 函数更简单，因为只需调用一次。

全面考虑问题

这些几何问题大多都产生**很多**特殊情况。注意这些特殊情况并且要**保证自己的程序能处理所有的情况**。

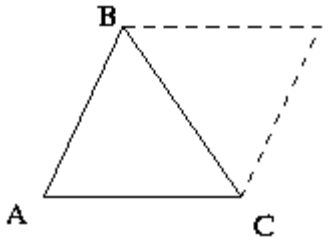
浮点运算也会带来新问题。浮点运算很难精确，因为注意：计算机只能计算有限的精度。特别地，要通过判断两个值的差是否在一个很小的范围内来判断是否相等。

计算几何算法

这里是一些能帮助你计算几何问题的东西。

三角形面积

为了计算由点(A,B,C)构成的三角形的面积，先选取一个顶点（例如 A），再向剩余两个顶点作向量（令 u=b-a, v=c-a）。三角形(A,B,C)的面积即为 u,v 叉积长度的一半。



另一个求三角形面积的变通方法就是用海伦公式。如果三角形三边长为 a,b,c，令 s=(a+b+c)/2，那么三角形面积就是:

```
sqrt(s*(s-a)*(s-b)*(s-c))
```

两条线段平行吗？

为了判断两条线段是否平行，分别沿两条线段建立向量，判断叉积是否（几乎为）零。

多边形面积

由点(x1,y1)...(xn,yn)组成的多边形的面积等于下列行列式的值：

```
 1   | x1 x2 ... xn |
---  |              |
 2   | y1 y2 ... yn |
```

也等于下面的式子的值:

```
x1y2 + x2y3 + ... + xny1 - y1x2 - y2x3 - ... - ynx1
```

就是选取原点作标准 连接原点和个顶点 并且两两个地计算叉积并加起来

点到直线的距离

点 P 到直线 AB 的距离也可以由叉积给出，准确的说,d(P,AB) = |(P - A) x (B - A)| / | B - A| 。

点 P 到由点 A,B 和 C 确定的平面的距离，令 n = (B - A) x (C - A)，那么 d(P,ABC) = (P-A) • n / |n|。

点在直线上

如果点到直线的距离是 0，那么点在直线上。

点都在直线的同侧

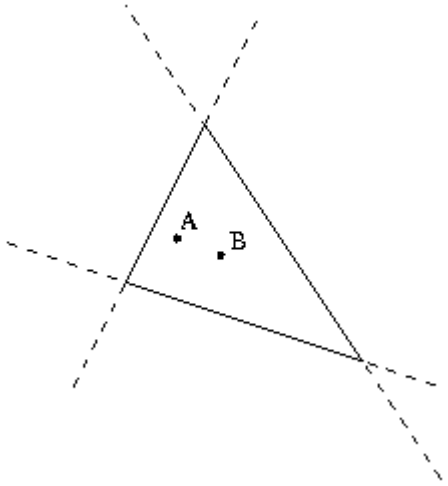
只讲两个点的情况。如果要确定点 C 和 D 是否在直线 AB 同侧，计算(B - A) x (C - A)和(B - A) x (D - A)的 z 分量。如果同号（或如果积为正），那么点 C,D 在直线 AB 同侧。

点在线段上

为了求出点 C 是否在线段 AB 上，先判断点 C 是否在直线 AB 上，再判断线段 AB 的长度是否等于线段 AC 长度与线段 BC 长度之和。

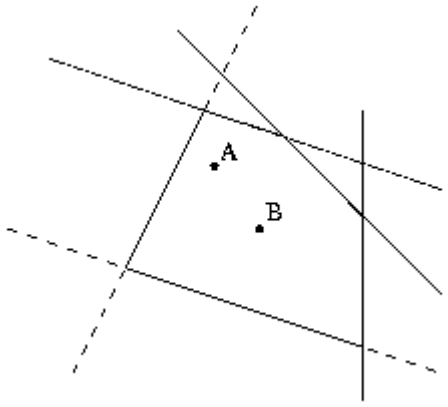
点在三角形内

要确定点 A 是否在三角形内，首先选择一个三角形内部的点 B(重心就很不错)。接下来，判断点 A,B 是否都在三边所在的三条直线的同侧。



点在凸多边形内

方法同上



四点（或更多）共面

如果要确定一组点是否共面，任选 3 个点。如果对于任意点 D，有((B - A) x (C - A)) • (D - A) = ~0，那么这些点共面。

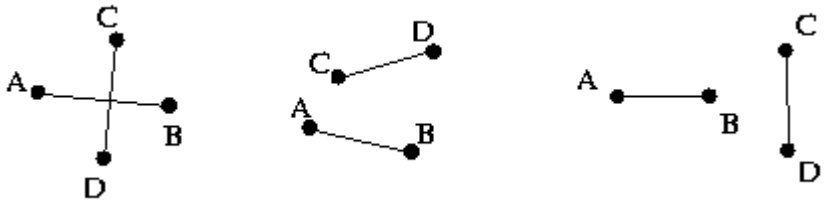
两条直线相交

平面内两条直线相交当且仅当直线不平行。

空间内，两直线 AB,CD 相交则 AB,CD 不平行，且点 A,B,C,D 共面。

两条线段相交

平面内，两条线段相交当且仅当 A,B 在直线 CD 异侧且 C,D 在直线 AB 异侧。



注意两个判断都是必须的，例如第三种情况第一个判断为 true，但第二个判断说明线段 AB,CD 不相交。在空间中，计算下面方程组，其中 i,j 未知：

```
Ax + (Bx - Ax) i = Cx + (Dx - Cx) j
Ay + (By - Ay) i = Cy + (Dy - Cy) j
Az + (Bz - Az) i = Cz + (Dz - Cz) j
```

如果方程组有解(i,j)满足 0<=i<=1，0<=j<=1，那么两线段相交于点(Ax + (Bx - Ax)i, Ay + (By - Ay)i, Az + (Bz - Az) i)

两直线的交点

在平面内的两条直线 AB,CD，求交点最直接的方法就是解下列的二元一次方程组：

```
Ax + (Bx - Ax)i = Cx + (Dx - Cx) j
Ay + (By - Ay)i = Cy + (Dy - Cy) j
```

交点是：

```
(Ax + (Bx - Ax) i, Ay + (By - Ay) i)
```

空间内，解同样的方程组来判断交点，交点是：

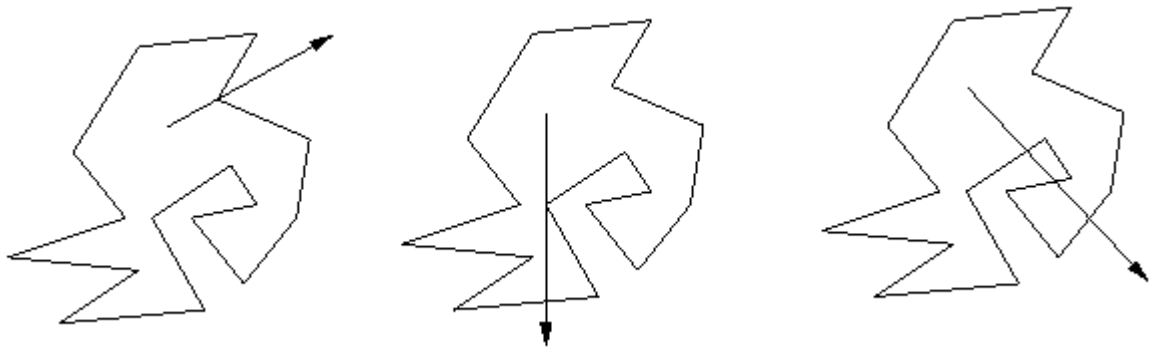
```
(Ax + (Bx - Ax)i, Ay + (By - Ay)i, Az + (Bz - Az)i)
```

判断平面内多边形的凹凸性

要判断平面内一多边形是否为凸，沿着顺时针方向扫一遍。对于每三个点(A,B,C)，计算叉积(B - A) x (C - A)。如果叉积的 z 分量均为负，多边形则为凸多边形。

点在凹多边形内

要确定点是否在凹多边形内，任选一点作射线，计算相交次数。如果与多边形相交于一点或一边，则换一个方向，否则，点在多边形内当且仅当射线与点相交次数为奇数。



这个方法也可以扩展到三维（或更高），但此时应相交于面（再判断），不是在点上或边上。

几何方法

这些几何方法介绍了一些技巧，可以用来优化时间和求得更精确的解。

蒙特卡洛方法(Monte Carlo)

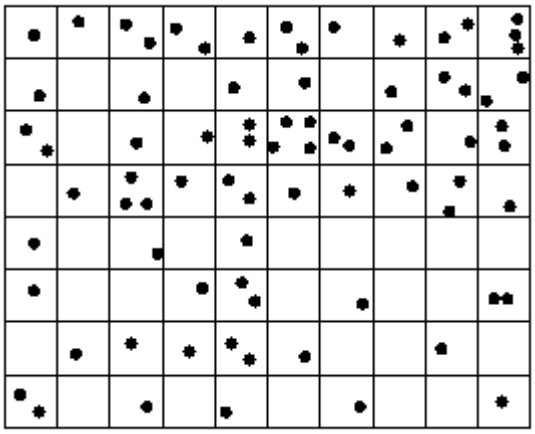
第一个方法是建立在随机化之上的。它不去直接计算某件事的概率，而是以一个随机事件来模拟，求得事件发生的频率。如果次数足够，频率和概率的差别会很小。

这对于确定某些东西来说是很有用的，例如计算某个图形的面积。它不去直接计算面积，而是建立一个已知大小的区域，每次向该区域扔一个“飞镖”，并计算击中区域内图形的频率。如果计算足够精确，就可以得到实际面积的一个足够好的近似值。

这个方法要得到一个足够小的相对误差（误差于实际值之商），需要大量成功发生的事件。如果发生的概率很小，结果就不好了。

分割技术

分割技术可用来优化时间。这需要把平面分割成小块（通常是分成小格，但有时也会用角度或其它方法），并将元素放入合适的区域中。当检查图案的某部分时，只有有该图案的格子会被检查到。所以，时间可以大大地优化，比如，要确定到定点的距离小于某个值的元素（此时图案是个圆），或求是否相交（此时图案为直线）。



转化为图

有时看起来像几何问题的问题实际上却是一个图的问题。因为输入是平面内的点，并不代表问题是几何问题。

例子

Point Moving

给定一组线段和点 A,B，能否在不穿过线段的情况下从 A 移动到 B?

线段将平面分割成不同部分,检查 A,B 是否在一个部分。

Bicycle Routing

给出有起点和终点的一组建筑物，找出从建筑物 A 到 B 不穿过任何建筑物的最短路线。

题解：图论问题。以建筑物的起点和*重*终点为点，两点之间在不径直穿过任何建筑物时连一条边，边权为两点之间的距离。这样就把原问题转化成了最短路问题。

Maximizing Line Intersections

给出平面内一组点，找到能被一条直线能相交到的最多线段。

题解：很显然，直线必须经过两个交点。这样，枚举每对交点，计算此时直线的交点数。可用分割法优化。

Polygon Classification

给出一组直线所确定的多边形，判断多边形是否是简单的（任意两条不相邻的线段不会相交）和凸的.

Section 4.1 Optimization 最优化

什么是最优化？

让你的工作代码更快

注意你的工作。在你的程序工作之前，不要试图让它变得更快，因为调试查错可能将更为复杂。在为了优化（而不是调试查错）而对程序作出任何更改之前，给你的代码做一下备份。当你作出了 3 种深层次的优化之后，发现第一次的优化破坏了程序的正确性，而此时你又没有任何备份可以恢复之前的正确程序，没有再比这更为痛苦的事了。

这块内容将注意几种能加速程序而不更改算法的方法。当然如果有更快的算法，则你的思考方向就应该是思考那种算法，而不是让一种本来就很慢的算法快那么一小点，这就像在猪身上使用香水一样。（USACO 这个比喻有点.....）

尤其值得注意的是，本模块内容仅将讨论使递归下降程序变得更快的方法，尝试避免搜索整棵树。这种方法通常被归为搜索的*剪枝*。

一个问题：链

这是一个在编译器中的现实问题，虽然在那个世界中只有乘法是可用的。

假设一个程序需要精确地得到 x^n 的值，而标准操作只有乘和除（没有 log/exp）。这些操作最少需要多少次？按次序输出为得到 x^n 所需要计算的 x 的幂值。

为了说明这个问题，以下给出的计算过程将仅表现指数，乘法将被表示为加法（两个指数的和），除法将被表示为减法（两个指数的差）。在合法的链中，每个指数都将是两个已得出的指数的和或差。所以计算 x^8 的序列将表示为：1 2 4 8；计算 x^15 的序列可以表示为：1 2 4 8 16 15

Test Set #1

考虑计算集合[1..50]内的数据。所有时间将在 233MHz 的 Pentium II 处理器上得到。

一个基本的算法

从 1 开始，进行深度优先搜索。

出现了什么问题？

这个程序将永远不会终止。

所以需要改变算法以减少生成比答案更长的数列的次数。假设最长链的长度为 32，为实现例子所需的运行时间达到了 4658.34 秒。

优化的基础

更早地剪枝，更频繁地剪枝

思考：在一棵输出 2 的搜索树中，避免搜索出超过 4 个节点。

两个能使搜索更快的基本思想：

- **不要做任何愚蠢的事**
- **不要把事情做两遍**

问题在于如何找到进行了两遍的事,什么是愚蠢的事。这里所有的优化操作都建立在计算正确的前提下，你要保证你的程序是正确的。

链问题的优化

Observation #1

有一种简单的方法可以先确定出一个数，或者纯粹是先得一个搜索的最大值。计算 n 的二进制表示形式，首先不断生成 2 的幂值，再将这些生成的数相加得到 n。

举例说：n=43，则它的二进制形式为 101011。因此，首先的到的数列为：1 2(10) 4(100) 8(1000) 16(10000) 32(100000)，所以计算 1+2=3(11)，3+8=11(1011)，11+32=43(101011)。故输出数列：1 2 4 8 16 32 3 11 43。至少可以先得到一个“最优解”。

而这个程序的运行时间为 4609.81 秒。

评价：这只是勉强可以算是优化。虽然这避免了很多不必要的计算，但在一棵如此巨大的搜索树中确实不能起到太大作用。

Observation #2

出现负数显然是愚蠢的，不要让它们出现在数列中。（证明过程略...）

现在新的运行时间为 387.34 秒。

出现 0 甚至更愚蠢。（证明过程略...）

新的运行时间：43.24 秒。

评价：两个简单的操作减少了 100 倍数量级地运算时间，并且它们的代码非常容易写，这非常好地体现了优秀的优化。

Observation #3

最优解显然要小于前面出现过的最大值。记录最大值，当搜索结点超过最大值时，即使得到了解也不会是最优解，所以果断退出。

运行时间：0.15 秒。

增加数据规模

目前程序很快，我们应该增加数据规模了。新的数据中幂不大于 300，运行时间是 93.21 秒。

注意 4

为什么不使用可变下界搜索呢？结点的增长随着层数增长，所以结点的质量越来越小。

运行时间: 93.05 秒。

评价：既然这样,从数据看这是一个不强的优化(慢慢会好起来)。它使程序代码产生了很大的改变，出错的机会增加了，还没有带来任何好处。这令我们有点惊讶，它一向有很大作用。

注意 5

最近的操作必须使用 next-to-last 数。如果不用，生成它干什么?（这句话是 The most recent operation must use the next-to-last number created. If it didn't, why bother generating that number，我翻译不通） (现在就呈现出可变下界深度优先搜索算法)。 [next-to-last 意为倒数第二的]

运行时间: 7.47 秒 。

注意 6

不要复制已有的数。

运行时间:：3.40 秒

检查

这样, 除了可变下深度优先搜索以外，只有“Don't Do Anything Stupid”这一原则被体现了，执行时间减少了约 842,510 倍。这是好的，但也许可以再提高。

增加数据规模

新的数据中幂不大于 500，运行时间是 206.71 秒。

注意 7

如果一个数列的前缀没有得到解，在后面加什么数也没有用。例如，如果 1 2 4 8 7 得不到解，就不用尝试 1 2 4 8 16 7 了。

运行时间： 53.70 秒

注意 8

如果选择了 i,数列中最大的数是 j,而且 i+j 小于下一个需要的数,那就不要选择 i。

运行时间: 44.52 秒。

注意 9

如果一个数列生成 x (x 不是目标)用了 j 项，而且存在另外一个数列用了少于 j 项，我们应该用这个短的数列把它替换掉，获得一个“更好的”数列，这是最佳的。

错误!

第一个反例: 10,127.用这种方法得到的解是 17 项，但是存在一个 16 项的方法。

这就是优化的风险：有时候，会得到错误结论。确保你不会掉入陷阱.

总结

8 个优化使运行时间缩小 400 万倍。可以使不大于 500 的数据在 44.52 秒内出解。有的程序可以在 640k 内存限制的情况下 1.91 秒内出解(无限制时 0.85 秒)。看看你的程序有多快。

Section 4.2 Network Flow 网络流

译 By Thunder

问题预备

最短路

问题

给出一个有向连通带权图，包含源点和汇点。

每一条弧的权表示那条弧的容量。一个图的流通过分配整数量给每条边来达到

- 通过每条弧的流不大于这条弧的容量
- 除了源点和汇点的每个点，流入量等于流出量

使源点的流出总量减去流入总量（或汇点的流入总量减去流出总量)最大

例

给出水管的设计和每条水管的容量。水管里的水必须从上往下流，所以每条水管里，水只能向一个方向流。

计算能从头（自来水厂）流到尾（您的农场）的水量。

算法

算法（贪心）通过迭代增加从源到汇的流来建立网络流。

从每条弧的权等于初始的权开始（弧的权符合那条弧里还未使用的容量）。

给出当前的图，找到从源到汇的一条路径，这条路径上的弧都是非 0 权。计算这条路径的最大流，叫它路径容量。

//增广路径 + 路径容量 = 瓶颈容量

对于每一条路径上的弧，给弧的权减去路径容量。另外，给路径中的反向弧（在相同两点之间的弧，不过方向相反）加上等于和路径容量相等的权（只要反向弧存在，就增加它的权）。//调整增广路径

继续调整路径直到没有增广路径存在。

这能保证停止，因为您加上每次至少一单位流（权总是整数），并且流严格单调递增。增加反向弧的权相当于减少路径上的流量。

如果您对算法的细节分析感兴趣，请向 Sedgewick 请教（注：Robert Sedgewick 普林斯顿大学的计算机科学教授）.

这是算法的伪代码

```
1   IF (源点 = 汇点)
2       总流 = 无限大
3       结束

4   总流 = 0

5   WHILE (TRUE)
6 # 找到从源到汇的最大的容量的路径
7 # 用修改过的 Dijkstra 算法 (在 Chapter 4 的 ditches 就要用 Bellman-Ford 了)
8     FOR 所有的顶点 i
9       前驱(i) = nil
10      flow(i) = 0
11      visited(i) = FALSE
12    flow(源点) = 无限大

13    WHILE (TRUE)
14      最大流 = 0
15      最大流节点 = nil
16      #   找到最大容量的未访问节点
17      FOR 每个顶点 i
18        IF ((flow(i)>最大流) AND (NOT visited(i)))
```

```
19            最大流 = flow(i)
20            最大流节点 = i
21        IF (最大流节点 = Nil)
22            退出内层 While 循环
23        IF (最大流节点 = 汇点)
24            退出内层 While 循环
24        visited(最大流节点) = TRUE
25          #   调整相邻节点
26        FOR 最大流节点的所有相邻节点 i
27          IF (flow(i)<min(最大流, 最大流节点到 i 的容量))
28            前驱(i) = 最大流节点
29            flow(i) = min(最大流, 最大流节点到 i 的容量))
30      IF (最大流节点 = Nil)    #   没有路径
31        退出外层 While 循环

32      路径容量 = flow(汇点)
33      总流 = 总流 + 路径容量

# 把那个流加到网络里 适当调整容量
35      当前节点 = 汇点
#     对于增广路径上每条弧, 前驱(当前节点), 当前节点
36      WHILE (当前节点不为源点)
38        下一个节点 = 前驱(当前节点)
39        下一个节点到当前节点的容量减去路径容量
40        当前节点到下一个节点的容量加上路径容量
41        当前节点 = 下个节点
```

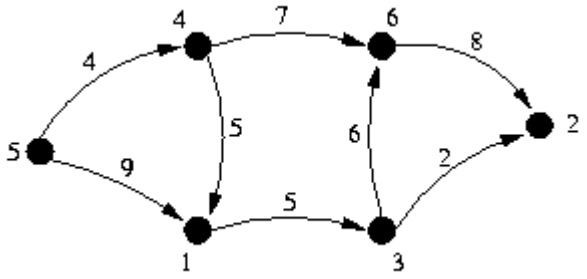
这个方法的运行时间为 O(FM)，F 是最大流 M 是弧的数目. 运行时间通常更短, 因为算法每次总是尽可能大的提升流。

为了确定每条弧上的流，比较开始的容量和最后的容量。如果最后的容量变少了，区别就在于通过这条弧的流。

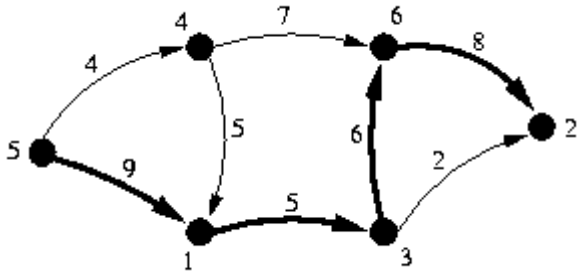
当有回路的时候，这个算法可能产生死循环。

实例

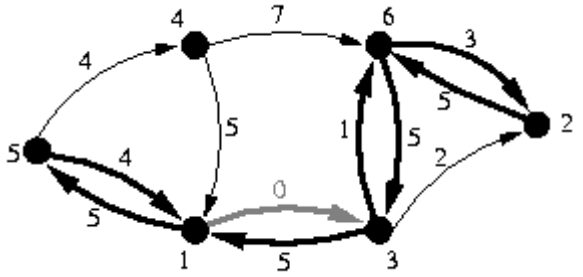
考虑下面那个网络，源点为 5，汇点为 2



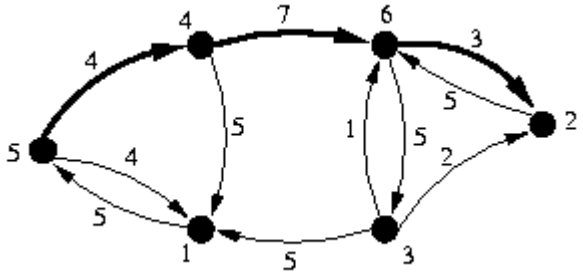
最大增广路径为(5,1,3,6,2)



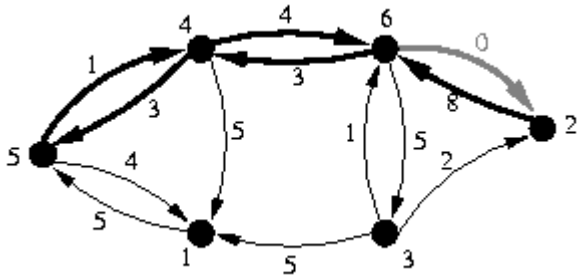
瓶颈弧是 1->3，容量为 5。因此，把路径上所有的弧减去 5，把反向弧上加上 5（如果需要，添加弧）。这就是得到的图：



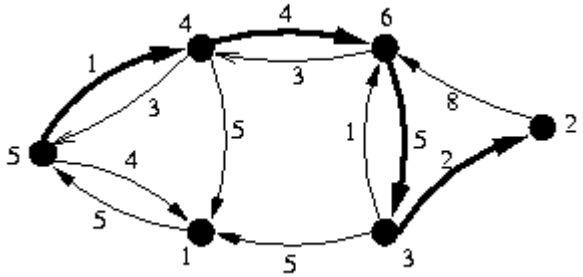
在新的图里，最大增广路径为(5,4,6,2)



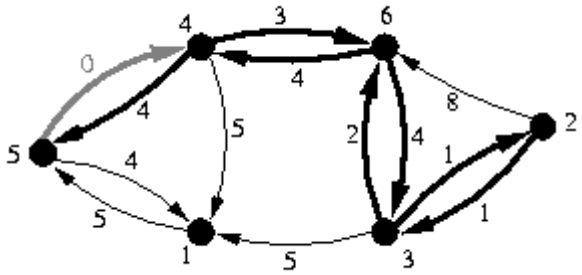
瓶颈容量为 3，再来一次。



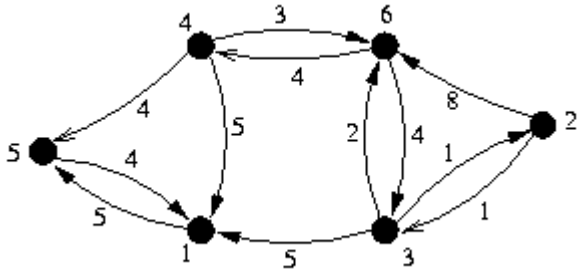
现在网络的最大容量路径为(5,4,6,3,2)



瓶颈为 1，再来一次



结果图从源到汇就没有增广路径了。唯一可以到达的源点的就是它自己和 1 号节点



算法加了三次流，第一次是 5，第二次是 3，最后一次是 1。因此，最大流是 9。

扩展

网络流问题是很广泛的，通常和图结合在一起。

扩展到无向图，只需要简单的把无向边扩展为两条相反方向的弧就可以了。

如果您想要限制任意一个节点的通过量，把一个节点拆成两个，一个流入节点，一个流出节点，把流入的弧连到流入节点里，流出弧连到流出节点里，在流入节点和流出节点之间加上一条弧，容量为限制。

如果您有多源点汇点，建立一个虚拟的源点和虚拟的汇点，虚拟的源点射出弧到每个源点，汇点射出弧到虚拟汇点。容量无限。

如果您有实数型的权，这个问题不再保证能出解，尽管答案能逐渐逼近。

可选问题

网络流能用来解决其他的一些不明显的问题。

最大匹配

给出两个对象集合（叫他们 A 和 B），你想要把 A 和 B 中的元素尽可能多的匹配起来，约束条件是只可以两两配对。这就是最大匹配问题。

把这个问题用网络流的形式表现出来，建立一个源点，射出容量为 1 的弧给 A 的元素，建立一个汇点，从 B 中的元素射出容量为 1 的弧到此汇点。另外，如果 A 中某个元素可 以和 B 中某个元素匹配，从 A 的这个元素向 B 的那个元素射出一条容量为 1 的弧。用网络流算法确定哪些 A 元素到 B 元素的弧被用到。

最小割

给出一个带权无向图，最小割就是一个权和最小的，能够把两个给出点分开的边的集合。

最小的权和就是两个节点之间的流。

要确定路径，按照权递增的顺序尝试删除每条边，看它是否减少了网络流(如果减少了，减少量应为边的容量)。第一个这样的边就是最小割的元素，去掉这条边继续在图上操作。

用相同的技巧，把节点用容量限制，这能够扩展到求节点的割。有向图也是一样。然而，它不能够解决所谓的“最佳匹配”问题，每一对都有一个“佳值”，而且您想要得到最高“佳值”和的匹配。

例题

如果问题讨论取流或者其他从一个地方到另一个地方东西的移动的最优值，就可以几乎确定是最大流。如果它讨论的是将两个物料项目以最小代价分隔开的问题，就可能是最小割。如果它讨论的任何一类东西的最大配对，就可能是最大匹配。

病毒流

您有一个通过网线连接的计算机网络。数据可能从任何一个方向流经网线。不幸的是，一台网络里的机器染上病毒，您需要从中心服务器隔离这台机器来防止病毒传染。给出关闭一对机器连接的代价，计算控制病毒传染到服务器的最小的代价。

这就是最小割问题。

伐木工人的计划

不同种类的树需要雇佣拥有不同的技巧的伐木工人来正确的砍树。不管哪种树或伐木工人，砍一棵树需要 30 分钟。给出伐木工人的信息，和哪种树需要哪位才能正确的砍倒，以及树的信息，计算在半小时内砍倒的最多的树的数目。对于每一个伐木工人，都有一种对应的他（她）能砍倒的树。所以，这个问题能用最大匹配算法来解决。

牛的电话联系（USACO 锦标赛 1996）

给出特定区域里的一组电脑，和电脑之间连着的网线，要想阻止给定两台电脑的联系 最少需要关掉多少网络中的电脑。假定两台给出的电脑没有关掉。

这相当于最少节点割的问题。两台给出的电脑标上源点和汇点。电缆是双向的。把每个点拆成流入点和流出点，这样我们就能够用 1 限制任何一台电脑。现在，网络里的最大流就相当于节点最小割了。

为了具体确定割，反复删除节点，直到您找到一个能降低网络流量的节点。

科学展览审定

一个科学展览由 N 个学科，和 M 个裁判。每个裁判愿意审定某些学科，每个学科需要一些裁判。每个裁判只能够审定在给出的科学展览中的一个学科。您在这些条件下总共能分配给多少裁判任务？

这是一个很类似最大匹配的问题，除了每个学科需要可能多于一个裁判。解决这个问题的最简单的方法是把从科目到汇点的弧的容量赋予需要裁判数的值。

石油管道设计

给出阿拉斯加管道线的设计（每条管道的容量，和管道如何连接），以及每个交点的位置，您想提高朱诺和费尔班柯斯之间最大流量，但您的钱只够添加一条容量为 X 的管道。此外，管道只能 10 英里长。这条管道添加在哪两个交点能最大程度的提高流量？

要解决这个问题，对于距离 10 英里以内的每一对交点，添加一条管道，再计算从朱诺到费尔班柯斯的流量增加值。每一个子问题都是最大流。

Section 4.3 Big Number 高精度

样例问题：阶乘（Factorial）

给出 N (1 <= N <= 200)，计算出 N!的值.

题目

所有编程语言默认定义的整型变量都是有取值限制的，有时，问题会要求计算一个超出了所有可用的数据类型存储上限的数据。比如说，200!就有 375 个数位（十进制下），1246 位（如果选择使用二进制来存储的话）的空间来存储它，显然是不可能用任何比赛环境下默认定义的数据类型来搞定的。所以，我们需要一个方法来储存和处理很大的数据。

结构

首先一个方法事实上是很直接的：一个数字列表和一个指针。如果上限给出了的话可以用数组来储存它，如果没有给出也可以用链表。

另一个思路来考虑这个存储方案是用多进制的方式储存。（大概就这个意思，不好直译）

比方说采用 b 进制，那么 a0, a1, ...就是被储存的每一位数字。那么，这个数字就被表示为（-1）^正负符号×（(a 0+ b^1 a1+ b^2 a2+ ... + b^n an)）。需要留意的是，a0, a1, ...的取值必须在[0，b-1]里。

通常情况下，我们使用的是十进制，毕竟这样让数字的展示变得容易（不要忘记了前导零哦，亲）

注意在这种表达方式下存储的数字的顺序是 a0，然后 a1，之后 a2，以此类推，刚好是我们通常的书写方式的逆序，对于链表，有时更值得去构建一个双端队列，以照顾部分倒序处理的算法。

数据处理

对于这种数据结构，想法完成各种操作需要我们回想一下怎样手动完成这些操作。最主要的问题是溢出，一定切记检查确保所有的加法和乘法什么的不会造成算术上溢，否则整个操作只会得到错误的结果（Pascal 中直接程序报错哦，亲）

方便起见，这里说明的算法都将基于数组进行说明，所以如果一个数字是 a0, a 1, ..., ak，那么对于所有的 i > k, ai = 0。对于链表，相关的算法会比较复杂一点（译者是 SB：别被吓到了，难不到哪里去）。附加说明一下，用于存储结果的数组每一项一般来说都应将被初始化为零。

数据的比较

要比较两个数据 a0, a1, ..., an 和 b0, b1, ..., bk 以及它们的符号位（用于标记正数和负数）signA（SA）、signB（SB）的步骤如下：

```
# 标注符号位
# sizeA = A 的数位个数
# signA = A 的符号
# (0 => 正, 1 => 负)
 1  if (signA < signB)
 2    return A 要小一些
 3  else if (signA > signB)
 4    return A 要大一些
 5  else
 6    for i = max(sizeA,sizeB) to 0
 7      if (a(i) > b(i))
 8        if (signA = 0)
 9          return A 要大一些
10        else
11          return A 要小一些
12    return A = B
```

加法

给出两个数 a0, a1, ..., an 和 b0, b1, ..., bk，请计算它们的和并存储至数组 C。为了让加法正常进行，2 x b 必须要小于能最大的能被表达的数。

如果两个数符号不相同，那么：计算哪一个的绝对值更大，然后用大的减去小的，最后再加上和绝对值大的那个数一样的符号位；否则，直接从 0 加到 max(n,k)，注意进位。

注意如果已经知道两个数都是正数，那就没必要写减法（absolute_subtract）的算法了。

下面是加法的伪代码。

```
 1 absolute_subtract(bignum A, bignum B,bignum C)
 2   borrow = 0
 3   for pos = 0 to max(sizeA, sizeB)
 4     C(pos) = A(pos)-B(pos)-borrow
 5     if (C(pos) < 0)
 6       C(pos) = C(pos) + base
 7       borrow = 1
 8     else
 9       borrow = 0
    #必须要这样做
    #否则没法完成部分数据
    #比如两个差不多的数相减
    # (比如：7658493 - 7658492)
10     if C(pos) != 0
11       sizeC = pos
12   assert (borrow == 0,
       "|B| > |A|; can't handle this")
13 absolute_add(bignum A,
                bignum B, bignum C)
14   carry = 0
15   for pos = 0 to max(sizeA,sizeB)
16     C(pos) = A(pos)+B(pos)+carry
17     carry = C(pos) / base
18     C(pos) = C(pos) mod base
19   if carry != 0
20     CHECK FOR OVERFLOW
21     C(max(sizeA,sizeB)+1) = carry
22     sizeC = max(sizeA,sizeB)+1
23   else
24     sizeC = max(sizeA, sizeB)

25 add (bignum A, bignum B, bignum C)
26   if signA == signB
27     absolution_add(A,B)
28     signC = signA
29   else
30     if (Compare(A,B) = A is larger)
       then
31       absolute_subtract(A,B)
32       signC = signA
33     else
34       absolute_subtract(B,A)
35       signC = signB
```

减法

有了刚才的加法做铺垫，减法就很简单了。要计算 A - B, 反转 B 的符号位再加上 A 和 B.

与普通正整数相乘

确保在结果不超过最大表示范围的情况下进行和普通正整数的乘法，可以用和在纸上手动计算的类似方法。

```
 1  if (s < 0)
 2    signB = 1 - signA
 3    s = -s
 4  else
 5    signB = signA
 6  carry = 0
 7  for pos = 0 to sizeA（*pos 是数位指针）
 8    B(pos) = A(pos) * s + carry
 9    carry = B(pos) / base
10    B(pos) = B(pos) mod base
11  pos = sizeA+1
12  while carry != 0
13    CHECK OVERFLOW
14    B(pos) = carry mod base
15    carry = carry / base
16    pos = pos + 1
17   sizeB = pos - 1
```

高精度乘法

高精度乘法就是多次普通乘法然后在正确的数位相加。

一般来说，用另一个数的各个数位来直接乘，然后加起来（在正确的数位）和我们手算乘法是差不多的方法

```
 1 multiply_and_add(bignum A, int s,
              int offset, bignum C)
 2  carry = 0
 3  for pos = 0 to sizeA
 4    C(pos+offset) = C(pos+offset) +
                A(pos) * s + carry
 5    carry = C(pos+offset) / base
 6    C(pos+offset) =
           C(pos+offset) mod base
 7  pos = sizeA + offset + 1
 8  while carry != 0
 9    CHECK OVERFLOW
10    C(pos) = C(pos) + carry
11    carry = C(pos) / base
12    C(pos) = C(pos) mod base
13    pos = pos + 1
14  if (sizeC < pos - 1)
15    sizeC = pos - 1
16 multiply(bignum A,
          bignum B, bignum C)
17  for pos = 0 to sizeB
18    multiply_and_add(A,
                B(pos), pos, C)
19  signC = (signA + signB) mod 2
```

大数据被小数据除

和手算没两样。

```
 1 divide_by_scalar (bignum A,
                int s, bignum C)
 2   rem = 0
 3   sizeC = 0
 4   for pos = sizeA to 0
 5     rem = (rem * b) + A(pos)
 6     C(pos) = rem / s
 7     if C(pos) > 0 and
              pos > sizeC then
 8       sizeC = pos
 9     rem = rem mod s
 # 记得退位
```

高精度除法

和手算也差不多。。。各种细心。。。注意如果 b 太大，这个算法会很耗时间

```
 1 divide_by_bignum (bignum A,
             bignum B, bignum C)
 2   bignum rem = 0
 3   for pos = sizeA to 0
 4     mult_by_scalar_in_place(rem, b)
 5     add_scalar_in_place(rem,
                           A(pos))
 6     C(pos) = 0
 7     while (greater(rem, B))
 8       C(pos) = C(pos) + 1
 9       subtract_in_place(rem, B)
10     if C(pos) > 0 and pos > sizeC
       then
11       sizeC = pos
```

二分查找

总体而言二分查找是很有用的，可是对于高精度运算来说，代价也很高。给出上下限，然后检查答案的取值范围是否在这个区间内，然后减小一般的区间再继续递归运行下去。

比如说，b 大了，那么用刚才的方法直接做除法就会很慢，可是下面的二分查找就会快得多：

```
 1 divide_by_bignum2 (bignum A,
               bignum B, bignum C)
 2   bignum rem = 0
 3   for pos = sizeA to 0
 4     mult_by_scalar_in_place(rem, b)
 5     add_scalar_in_place(rem,
                         A(pos))

 6     lower = 0
 7     upper = s-1
 8     while upper > lower
 9       mid = (upper + lower)/2 + 1
10       mult_by_scalar(B, mid, D)
11       subtract(rem, D, E)
12       if signE = 0
```

```
13        lower = mid
14      else
15        upper = mid - 1

16    C(pos) = lower
17    mult_by_scalar(B, lower, D)
18    subtract_in_place(rem, D)

19    if C(pos) > 0 and
                   pos > sizeC
20      sizeC = pos
```

Section 5.1 Two Dimensional Convex Hull 二维凸包

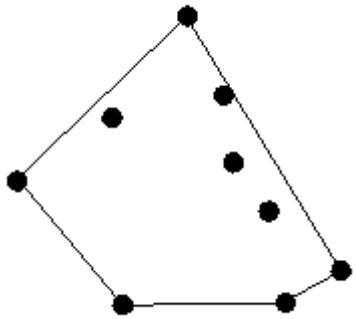
译 by Felicia Crazy

准备知识

几何学

数学模型

给出平面内的点集，找出面积最小的凸多边形，使得这些点在这个多边形之内（或者在它的边上）。
可以看出，多边形的顶点必须是给定点集中的点。

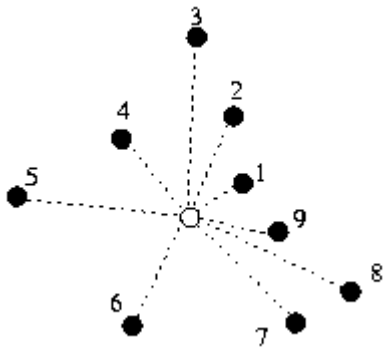


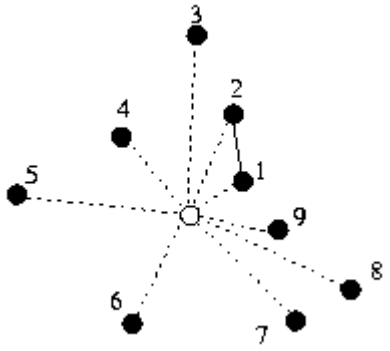
例题：放牧奶牛

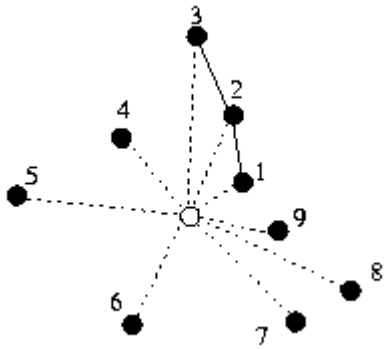
农夫约翰想要修建一个栅栏，来防止讨厌的当地大学生在他的奶牛们睡觉的时候把它们掀翻。他的每头奶牛都有一个最喜欢的吃草点，农夫约翰想要把这些点都围在栅栏内。农夫约翰要围出一个凸的形状，因为这样更容易把奶牛赶进牧场。
帮助农夫约翰确定面积最小的而且包括所有奶牛喜爱的吃草点的栅栏形状。

格拉汉扫描法

解决二维凸包问题有好几种算法。这里，我们只介绍比较容易编码和记忆的“卷包裹”算法（其实就是我们所说的格拉汉扫描法——译者注!）。
算 法的基本思想是在一个肯定会在凸包内的点周围不断地由顺时针或逆时针方向增加顶点，并确保每个内角都小于 180 度（保证最终答案是凸的）。如果三个连续的顶点构成的角度大于 180 度，删掉中间的点。可以用两个沿着多边形边的连续向量的叉积来判断角度是否大于 180 度。







凸包算法流程

- 找出一个必定会在凸包内的中点
- 计算每个点和中点的连线与 x 轴的夹角（在 0——360 度的范围内）
- 根据这些夹角对顶点排序
- 加入最初的两个顶点
- 对于除最后一个顶点以外的其余顶点
- 让其成为凸包上的下一个顶点
- 检查它和前面两个顶点组成的角是否大于 180 度
    - 如果它和前面两个顶点组成的角大于 180 度，那么把它前面那个顶点删掉
- 加入最后一个顶点
    - 完成上述的删除任务
    - 检查最后一个顶点和它的前一个顶点和第一个顶点所组成的角是否大于 180 度，或者最后一个顶点和第一、第二个顶点组成的角是否大于 180 度。
    - 如果第一种情况为真，删除最后一个顶点，并且检查倒数第二个顶点。
    - 如果第二种情况为真，删除第一个顶点，继续检查。
    - 当两种情况都不为真时，停止。
- 由于角度的计算方式，增加顶点的时间复杂度是线性的（就是我们所说的 O(n) ）。因此，运行的时间复杂度决定于排序的时间复杂度，所以“卷包裹法”的时间复杂度是 O(n log n)，这是最优的。

伪代码

这里是凸包算法的伪代码：

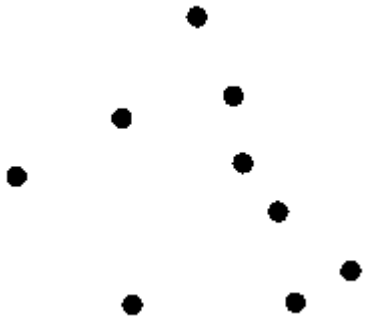
```
0    # x(i), y(i) 是第 i 个顶点的 x,y 坐标
     # atan2 在 Pascal 中是 arctan2，-pi<=结果<=pi
     # zcrossprod(v1,v2) 为 v1,v2 的叉积的 z 分量
     #  如果 zcrossprod(v1,v2) < 0,
     #     那么 v2 在 v1 的“右边”(好像指顺时针方向)
     #   因为我们逆时针加入点
     #     如果 zcrossprod(v1,v2) < 0
     #        则 angle > 180°
```

```
 1  (midx, midy) = (0, 0)
 2  For all points i
 3    (midx, midy) = (midx, midy) +
          (x(i)/npoints, y(i)/npoints)
 4  For all points i
 5    angle(i) = atan2(y(i) - midy,
                       x(i) - midx)
 6    perm(i) = i

 7  把 perm 基于 angle() 排序
    # 其中 angle(perm(0)) <= angle(perm(i)) for all i

    # 开始构造凸包
 8  hull(0) = perm(0)
 9  hull(1) = perm(1)
10  hullpos = 2
11  for 对所有点 p, 根据 perm() 的次序,
              除 perm(npoints - 1) 以外
12    while (hullpos > 1 and
         zcrossprod(hull(hullpos-2) -
13       hull(hullpos-1),
         hull(hullpos-1) - p) < 0)
14          hullpos = hullpos - 1
15       hull(hullpos) = p
16       hullpos = hullpos + 1

    # 加入最后一个点
17  p = perm(npoints - 1)
18  while (hullpos > 1 and
         zcrossprod(hull(hullpos-2) -
19       hull(hullpos-1),
         hull(hullpos-1) - p) < 0)
20    hullpos = hullpos - 1

21  hullstart = 0
22  do
23    flag = false
24    if (hullpos - hullstart >= 2 and
           zcrossprod(p -
25         hull(hullpos-1),
           hull(hullstart) - p) < 0)
26      p = hull(hullpos-1)
27      hullpos = hullpos - 1
28      flag = true
29    if (hullpos - hullstart >= 2 and
          zcrossprod(hull(hullstart) - p,
          hull(hullstart+1) -
          hull(hullstart)) < 0)
```

```
30        hullstart = hullstart + 1
31        flag = true
32    while flag
33    hull(hullpos) = p
34    hullpos = hullpos + 1
```

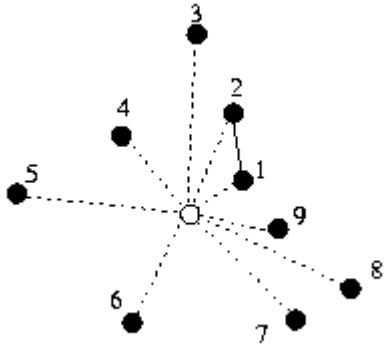
跟踪

对于下面的跟踪，我们使用这些顶点：

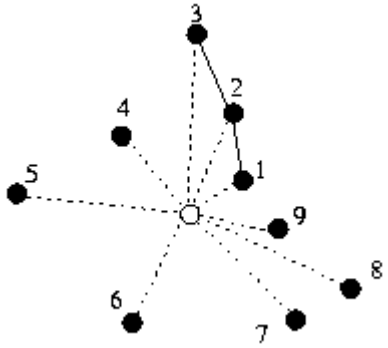


选择一个中心，计算夹角，并排序。

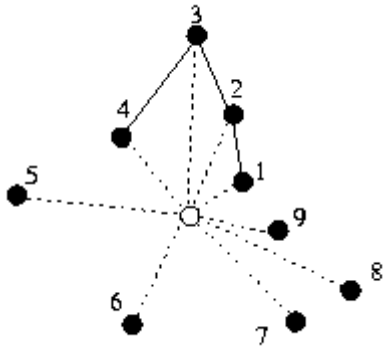
现在，从加入最初的两个顶点开始。



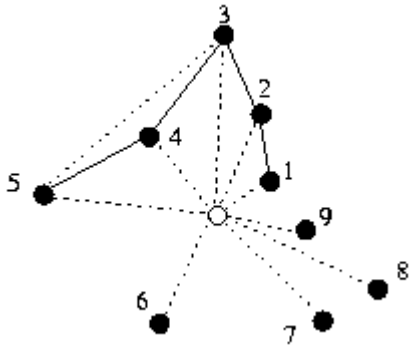
现在，加入第三个顶点。由于这步操作没有和前两个顶点构成一个大于 180 度的角，我们只需加入这个顶点。



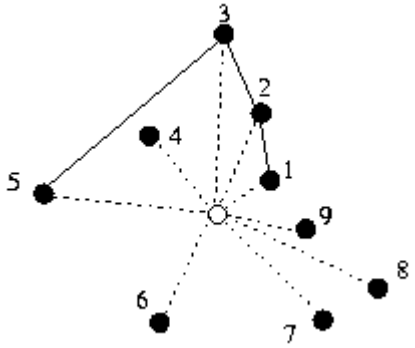
加入第四个顶点。这一次没有构成大于 180 度的角，所以不必做更多的工作。



加入第五个顶点。

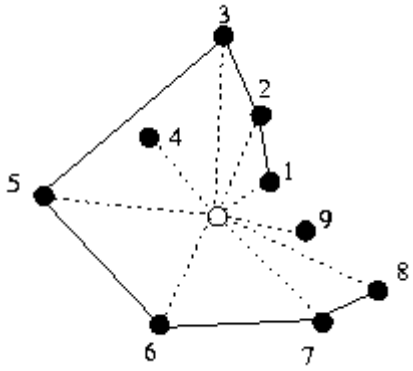


由于第三个，第四个，和第五个顶点构成了一个大于 180 度的角（一个“向右的”转弯），我们删去第四个顶点。

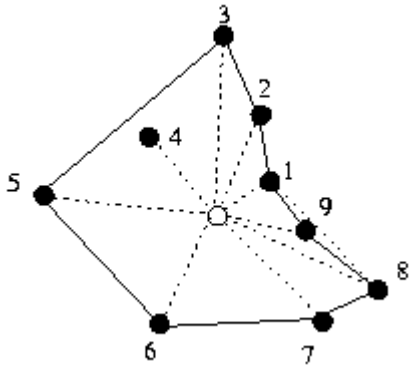


第二个，第三个，和第四个顶点没有构成一个大于 180 度的角，所以我们完成了加入第五个顶点的任务。

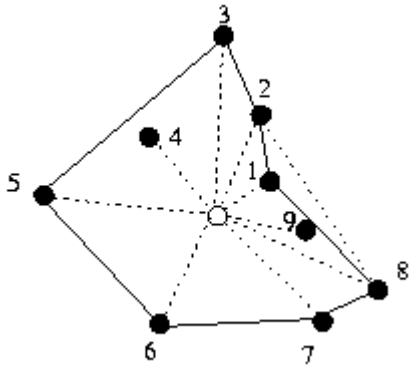
加入第六个，第七个，和第八 8 个顶点。这个过程中没有附加的工作。



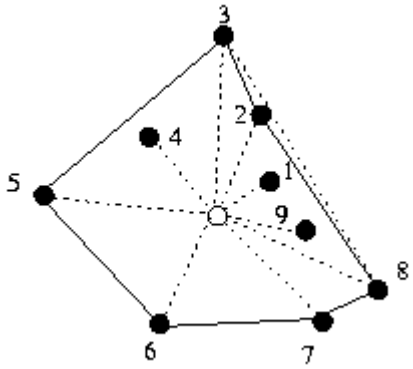
接着，加入最后一个顶点。



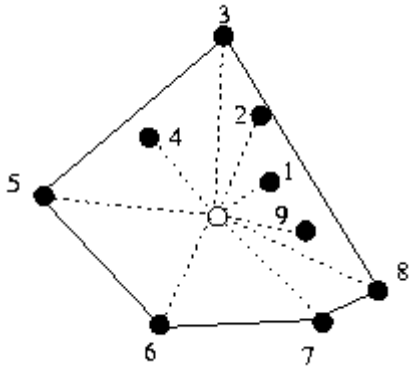
第八个，第九个，和第一个顶点构成“右转”；删除第九个顶点。



第七个，第八个，和第一个顶点已经完成了，可是第八个，第一个，和第二个顶点构成“右转”，所以我们必须删除第一个顶点。



现在，第八个，第二个，和第三个顶点构成“右转”，所以我们又要删除第二个顶点。



剩下的顶点并不违反“左转”原则，所以我们已经完成了任务，并且建立了给出顶点的凸包。

问题提示

类似把顶点放入多边形的题目通常是求凸包。如果题目要求一个面积最小的凸多边形，或者周长最小的凸多边形，那么我们几乎可以确定是要求凸包了。

推广

不幸的是，这个算法不能简单地推广到三维的情形。幸运的是，三维凸包算法全都超级复杂（四维以上的更恶心），所以题目不太可能要你去求。

如果你给多边形加上任何限制条件时，这个算法就玩完了（例如，多边形的顶点不多于 n 个，或者必须是矩形）。

例题

树的难题 [IOI 1991, problem 2]

已知：有一些树，你必须用铁丝围绕这些树，使得你用的铁丝最短。计算哪些树会在多边形的顶点上和铁丝的长度，还有农夫的小屋是在多边形内，还是在它之外，还是跨过多边形的边？

分析：多边形的顶点和铁丝的长度可以由问题直接得到。农夫的小屋是坐标轴上的一个矩形，你需要一点几何学知识来确定矩形中的所有点都在凸包中，或者都在凸包外，或者一些在凸包内，一些在凸包外。这样你就可以得到你想要的答案。查查几何手册，找一下这类问题的解法。

吝啬的护城河建设

已知：一些多边形房屋，计算包含这些多边形除去一个多边形的护城河的最小长度。

分析：计算已知多边形的凸包相当于计算它的所有顶点的凸包。这是一个组合问题，需要一个循环和一个凸包生成程序。对于每座房屋，删去这座房 屋并计算剩下顶点的凸包。选择使得凸包最小的那座房屋。注意，只要考虑那些顶点和整个凸包重合的房屋即可，考虑整个凸包内的房屋对于问题没有任何帮助。

Section 5.3 Heuristic Search 启发式搜索

译 By SuperBrother

知识准备

递归思想

主要思想

启发式搜索的主要思想是通过评价一个状态有"多好"来改进对于解的搜索.

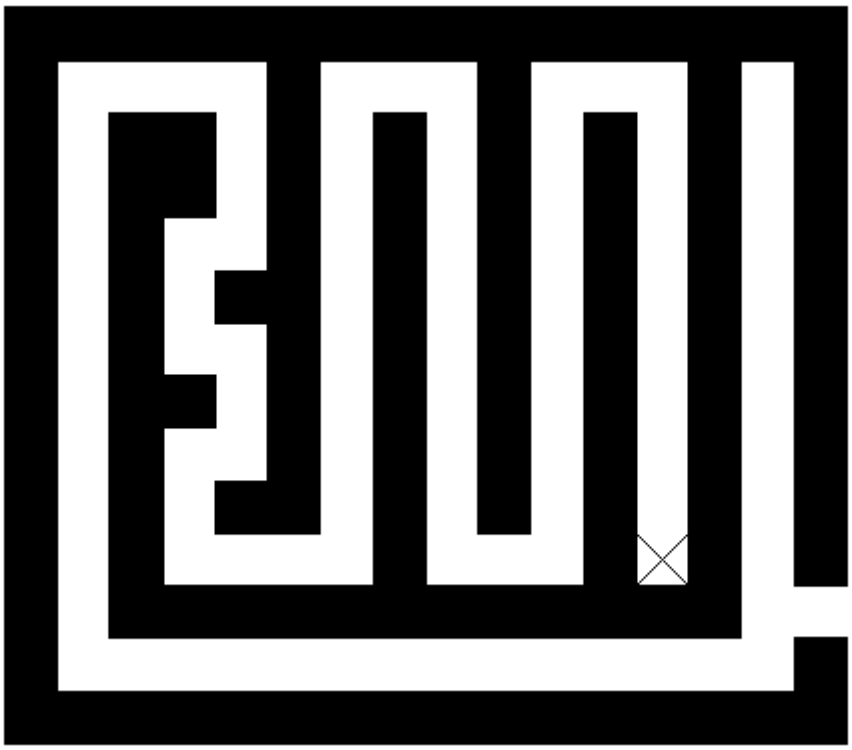
通常,我们用关于状态的一个函数来表示一个状态有多好.这种函数被称为估价函数.下面有估价函数的几个例子:

- 在迷宫中当前点至出口的欧氏距离;
- 在跳棋中你的跳棋数与对手的跳棋数之差;
- 空当接龙中放入空当的纸牌数加上空当数的 5 倍

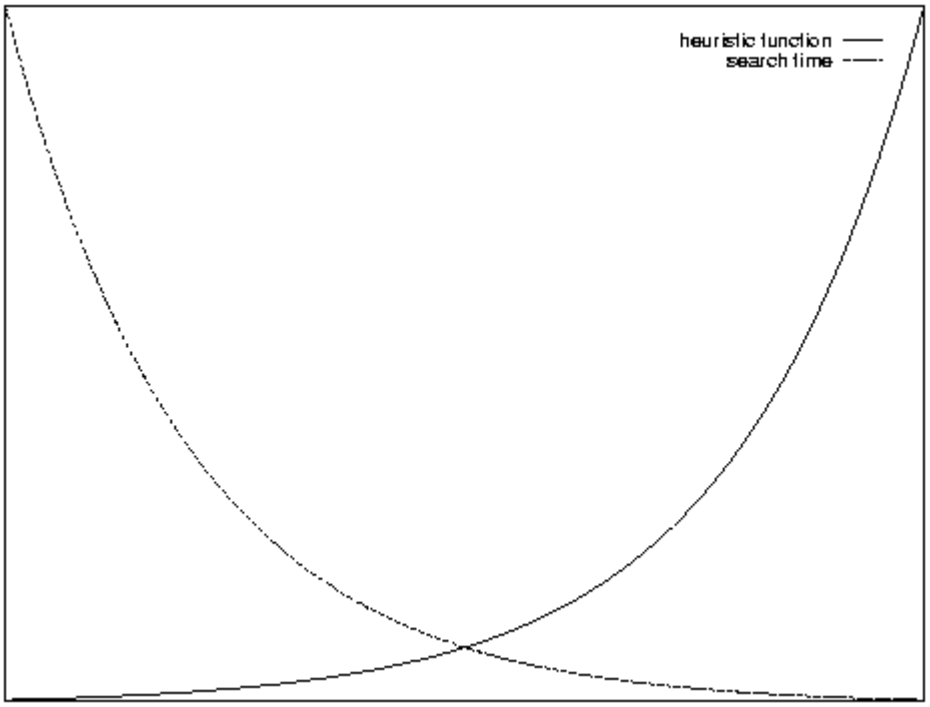
设计估价函数

直观上,强的估价函数一般使搜索更快.问题在于,如何评判一个估价函数的优劣?

评价估价函数的优劣,在于这个函数能够多好地表达状态的代价.例如,在迷宫中,如何估计到出口的距离?欧式距离很不好,甚至对于一些简单的迷宫,相同的解也会离的很远:



通常,估价函数越优,搜索越快,搜索时间和估价函数准确性有如下关系:



需要注意的是即使是很差的估价函数也能加快搜索速度(如果用的正确的话)
另外很重要的一点是设计估价函数时,考虑它的可行性.说一个估价函数有可行性,就是说这个函数低估了状态的代价,并且对于所有状态,是非负的.例如,在迷宫中欧式距离是不可行的,理由如上图所示.

方法#1:启发式剪枝

估价函数最简单最普通的用法是进行剪枝.假设有一个求最小代价的一个搜索,使用一个可行的估价函数.如果搜到当前状态时代价为 A,这个状态的估价函数是 B,那么从这个状态开始搜所能得到的最小代价是 A+B.如果当前最优解是 C 满足 C<A+B,那么就不需要从这个状态开始搜索.
把这个剪枝加入到一个朴素的搜索中,编程简单,易于调试,但可以带来时间效率的巨大提升.

方法#2:最佳优先搜索

最佳搜索可以看成贪心的深度优先搜索.
与一般搜索随意扩展后继节点不同,最优优先搜索按照估价函数所给的他们的"好坏"的顺序扩展节点.与只扩展最优节点的贪心不同,最佳优先搜索先扩展较"优"的后继,然后再扩展较"差"的后继.与上面的启发式剪枝组合,它可以很好地提升效率.

方法#3:A*搜索

A*搜索和"贪心"的广度优先搜索相似.
广度优先搜索总是扩展已达到的代价最小节点.但是,A*搜索,扩展最"有可能"的节点(就是说,到达 1 节点的代价和这个节点的估价函数之和最小)
状态被保存在优先队列中,优先级是状态的代价与估价函数之和.每一步,算法移除优先级最小的节点,将它的后继加入优先队列中.
如果估价函数可行,A*搜索到的第一个结束状态可以保证是最优的.

例题

骑士覆盖[经典问题]

在 n*n 的棋盘中放入尽可能少的骑士,使得所有的格子都能被攻击到.骑士所在的格子不会被自己攻击到.
算法:一个可行的估价函数是骑士攻击到的格子总数除以 8(8 即一个骑士能攻击的格子)(原句:the number of squares a knight can attack).这个可以被任意方式使用,虽然 A*由于队列的原因,空间需求量大.

8-方块[经典问题]

一个 3*3 的棋盘,一个格子为空,空格临近的数可以被移动到空格中.

5 _ 4---->5 1 4
6 1 8---->6 _ 8

7 3 2---->7 3 2

达到下面的状态所需最小步数是多少?(保证有解)

1 2 3

4 5 6

7 8 _

估价函数:起始状态到结束状态中所有数的位置的曼哈顿距离之和.

算法:状态空间很小(只有 362,880),A*就很好,不过要保证每个状态只能被访问一次.