十大经典算法[NEU]

Boycott 整理

第一版

搜索算法

一、 算法描述：

下面以 n 皇后问题为样例讲解

将 n 个皇后摆放在一个 $n \times n$ 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。

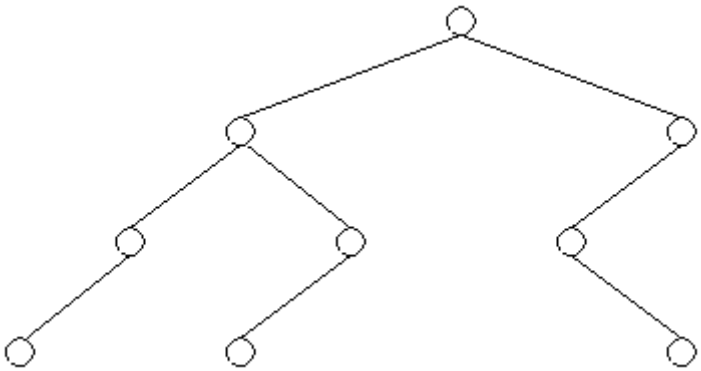
```
1  search(col)
2          if  filled  all  columns
3                  print  solution  and  exit

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)
```

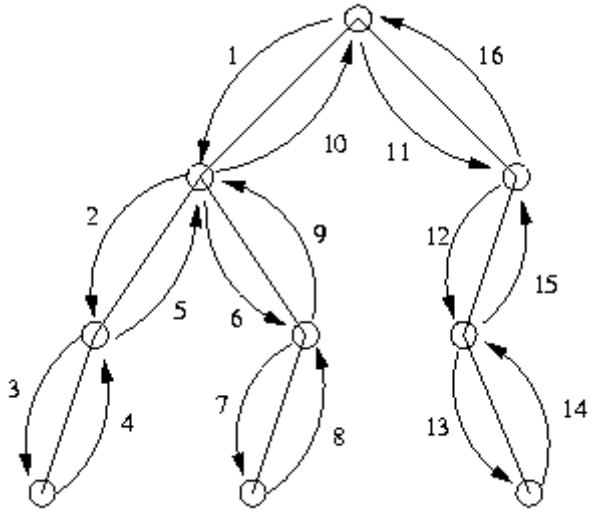
从 search(0)开始搜索，由于在每一步中可选择的节点较少，该方法可以较快地求解：当一定数量的皇后被摆放在棋盘上后那些不会被攻击到的节点的数量将迅速减少。

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 c^d 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.

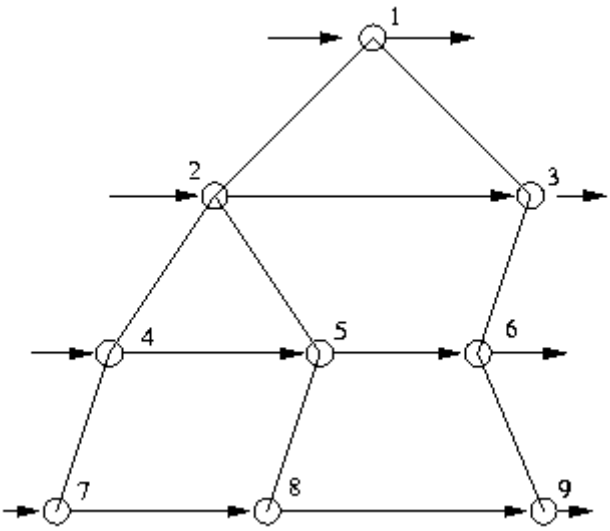
广度优先搜索 (BFS)

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、第三层厖以此类推。

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?#27169;仿广度优先搜索?#25628;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 c^k 。若搜索树共有 d 层，则一个完整的 DFS-ID 将耗时 $c^0 + c^1 + c^2 + \ldots + c^d$ 。如果 c = 2 ，那么式子的和是 $c^{d+1} - 1$ ，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。

搜索方式	时间	空间	使用情况
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。

简表：

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！

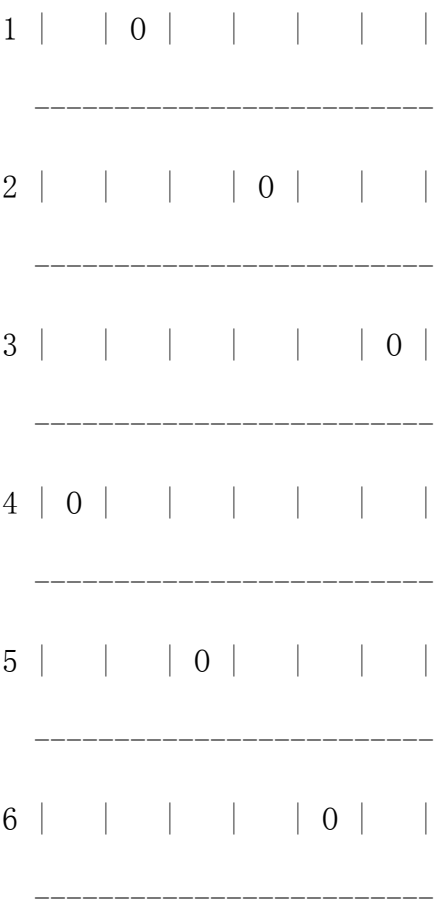
一、例题

Checker Challenge (N 皇后问题)

检查一个如下的 6 x 6 的跳棋棋盘，有六个棋子被放置在棋盘上，使得每行、每列只有一个，每条对角线(包括两条主对角线的所有平行线)上至多有一个棋子。

Column

1 2 3 4 5 6



上面的布局可以用序列 2 4 6 1 3 5 来描述，第 i 个数字表示在第 i 行的相应位置有一个棋子，如下：

行号 1 2 3 4 5 6

列号 2 4 6 1 3 5

这只是跳棋放置的一个解。请编一个程序找出所有跳棋放置的解。并把它们以上面的序列方法输出。解按字典顺序排列。请输出前 3 个解。最后一行是解的总个数。

特别注意：对于更大的 N(棋盘大小 N x N)你的程序应当改进得更有效。不要事先计算出所有解然后只输出(或是找到一个关于它的公式），这是作弊。如果你坚持作弊，那么你登陆 USACO Training 的帐号将被无警告删除

TIME LIMIT: 1s

INPUT FORMAT:

一个数字 N (6 <= N <= 13) 表示棋盘是 N x N 大小的。

OUTPUT FORMAT:

前三行为前三个解，每个解的两个数字之间用一个空格隔开。第四行只有一个数字，表示解的总数。

SAMPLE INPUT

6

SAMPLE OUTPUT

2 4 6 1 3 5

3 6 2 5 1 4

4 1 5 2 6 3

4

代码：

```
#include<stdio.h>
#define maxn 13
int n,ans,num[5][maxn+1],ta=0;
bool s[maxn+1];//标记竖行
bool xl[maxn*2+1];//标记对角线'/'
bool xr[maxn*2+1];//标记对角线'\'

void solve(int t)
{
    if(t==n+1){
        ans++;
        if(ta<3){
            ta++;
            for(t=1;t<=n;t++)
                num[ta][t]=num[ta-1][t];
        }
        return ;
    }

    int i,ts;
    if(n!=6&&t==1)//单独处理 6 这个数据，其答案数太少，不能进行对称性优化
        ts=(n+1)/2;
    else
        ts=n;
    for(i=1;i<=ts;i++){
```

```
        int temp=i-t;
        if(temp<0) temp=0-temp,temp+=n;
        if(!s[i]&&!xl[i+t]&&!xr[temp]){
            s[i]=1;    xl[i+t]=1;    xr[temp]=1;
            if(ta<3) num[ta][t]=i;
            solve(t+1);
            s[i]=0;    xl[i+t]=0;    xr[temp]=0;
        }
        if(n!=6&&t==1&&i==(n/2)) ans*=2;
    }
    return ;
}

int main()
{
    int i,j;
    scanf("%d",&n);
    ta=0;
    solve(1);
    for(i=0;i<3;i++)
       for(j=1;j<=n;j++)
         if(j!=n) printf("%d ",num[i][j]);
         else printf("%d\n",num[i][j]);
    printf("%d\n",ans);

    return 0;
}
```

贪心算法

一、算法描述

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

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

一、例题

活动安排问题

- 活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合，是可以用贪心算法有效求解的很好例子。该问题要求高效地安排一系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法使得尽可能多的活动能兼容地使用公共资源。
- 设有 n 个活动的集合 E={1,2,…,n}，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。每个活动 i 都有一个要求使用该资源的起始时间 si 和一个结束时间 fi,且 si <fi 。如果选择了活动 i，则它在半开时间区间[si, fi)内占用资源。若区间[si, fi)与区间[sj, fj)不相交，则称活动 i 与活动 j 是相容的。也就是说，当 si≥fj 或 sj≥fi 时，活动 i 与活动 j 相容。
- 在下面所给出的解活动安排问题的贪心算法 greedySelector ：

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

```
count++;
}
else a[i]=false;
}
return count;
}
```

- 由于输入的活动以其完成时间的非减序排列，所以算法 greedySelector 每次总是选择具有最早完成时间的相容活动加入集合 A 中。直观上，按这种方法选择相容活动为未安排活动留下尽可能多的时间。也就是说，该算法的贪心选择的意义是使剩余的可安排时间段极大化，以便安排尽可能多的相容活动。
- 算法 greedySelector 的效率极高。当输入的活动已按结束时间的非减序排列，算法只需 O(n)的时间安排 n 个活动，使最多的活动能相容地使用公共资源。如果所给出的活动未按非减序排列，可以用 O(nlogn)的时间重排。

- 例：设待安排的 11 个活动的开始时间和结束时间按结束时间的非减序排列如下：

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

- 若被检查的活动 i 的开始时间 Si 小于最近选择的活动 j 的结束时间 fi，则不选择活动 i，否则选择活动 i 加入集合 A 中。
- 贪心算法并不总能求得问题的整体最优解。但对于活动安排问题，贪心算法 greedySelector 却总能求得的整体最优解，即它最终所确定的相容活动集合 A 的规模最大。这个结论可以用数学归纳法证明。

动态规划

一、算法描述

1、动态规划算法基本思想

动态规划算法通常用于求解具有某种最优性质的问题。在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值的解。动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。我们可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。具体的动态规划算法多种多样，但它们具有相同的填表格式。

2、动态规划适用条件

任何思想方法都有一定的局限性，超出了特定条件，它就失去了作用。同样，动态规划也并不是万能的。适用动态规划的问题必须满足最优化原理和无后效性。

1.最优化原理（最优子结构性质） 最优化原理可这样阐述：一个最优化策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。简而言之，一个最优化策略的子策略总是最优的。一个问题满足最优化原理又称其具有最优子结构性质。

2.无后效性 将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态。换句话说，每个状态都是过去历史的一个完整总结。这就是无后向性，又称为无后效性。

3.子问题的重叠性 动态规划将原来具有指数级复杂度的搜索算法改进成了具有多项式时间的算法。其中的关键在于解决冗余，这是动态规划算法的根本目的。 动态规划实质上是一种以空间换时间的技术，它在实现的过程中，不得不存储产生过程中的各种状态，所以它的空间复杂度要大于其它的算法。

二、例题

Subset Sums

对于从 1 到 N 的连续整集合合，能划分成两个子集合，且保证每个集合的数字和是相等的。

举个例子，如果 N=3，对于{1，2，3}能划分成两个子集合，他们每个的所有数字和是相等的：

{3}and {1,2}

这是唯一一种分发（交换集合位置被认为是同一种划分方案，因此不会增加划分方案总数）

如果 N=7，有四种方法能划分集合{1，2，3，4，5，6，7}，每一种分发的子集合各数字和是相等的:

{1,6,7} and {2,3,4,5} {注 1+6+7=2+3+4+5}

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

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

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

给出 N,你的程序应该输出划分方案总数,如果不存在这样的划分方案,则输出 0。程序不能预存结果直接输出。

INPUT FORMAT

输入文件只有一行，且只有一个整数 N

SAMPLE INPUT (file subset.in)

7

OUTPUT FORMAT

输出划分方案总数，如果不存在则输出 0。

SAMPLE OUTPUT (file subset.out)

4

题解：

简单的动态规划。一看题目可以看得出这是一种映射的关系。首先和是一一映射的，由于种情况是在前一个情况下加上一个不重复的 n 得到，所以情况和这情况的和是一一映射的。所以所求的状态 n 可以用状态 n 中这个集合所以的元素的和来表示。第二个映射是一个数 x，它可以由 x-n 到 x-1 这些数加上 n 得到。

所以得到状态转移方程 $$f(x)=\sum_{i=1}^{n} f(x-i)$$

参考程序如下（C++语言）：

```
#include <fstream>
using namespace std;
const unsigned int MAX_SUM = 1024;
int n;
unsigned long long int dyn[MAX_SUM];
ifstream fin ("subset.in");
ofstream fout ("subset.out");
int main() {
   fin >> n;
   fin.close();
   int s = n*(n+1);
   if (s % 4) {
      fout << 0 << endl;
      fout.close ();
      return ;
   }
   s /= 4;
   int i, j;
   dyn [0] = 1;
   for (i = 1; i <= n; i++)
      for (j = s; j >= i; j--)
         dyn[j] += dyn[j-i];
   fout << (dyn[s]/2) << endl;
   fout.close();
   return 0;
}
```

最短路径

一、算法描述

最短路径问题是图论研究中的一个经典算法问题， 旨在寻找图（由结点和路径组成的）中两结点之间的最短路径。 算法具体的形式包括：

确定起点的最短路径问题 - 即已知起始结点，求最短路径的问题。

确定终点的最短路径问题 - 与确定起点的问题相反，该问题是已知终结结点，求最短路径的问题。在无向图中该问题与确定起点的问题完全等同，在有向图中该问题等同于把所有路径方向反转的确定起点的问题。

确定起点终点的最短路径问题 - 即已知起点和终点，求两结点之间的最短路径。

全局最短路径问题 - 求图中所有的最短路径。

解决方法

综述

用于解决最短路径问题的算法被称做“最短路径算法”，有时被简称作“路径算法”。最常用的路径算法有：

Dijkstra 算法

A*算法

SPFA 算法

Bellman-Ford 算法

Floyd-Warshall 算法

Johnson 算法

所谓单源最短路径问题是指：已知图 G＝（V，E），我们希望找出从某给定的源结点 S∈V 到 V 中的每个结点的最短路径。

首先，我们可以发现有这样一个事实：如果 P 是 G 中从 vs 到 vj 的最短路，vi 是 P 中的一个点，那么，从 vs 沿 P 到 vi 的路是从 vs 到 vi 的最短路。

Dijkstra 算法

Dijkstra(迪杰斯特拉)算法是典型的最短路径路由算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。Dijkstra 算法能得出最短路径的最优解，但由于它遍历计算的节点很多，所以效率低。

Dijkstra 算法是很有代表性的最短路算法，在很多专业课程中都作为基本内容有详细的介绍，如数据结构，图论，运筹学等等。

Dijkstra 一般的表述通常有两种方式，一种用永久和临时标号方式，一种是用 OPEN, CLOSE 表方式，Drew 为了和下面要介绍的 A* 算法和 D* 算法表述一致，这里均采

用 OPEN,CLOSE 表的方式。

其采用的是贪心法的算法策略

大概过程：

创建两个表，OPEN, CLOSE。

OPEN 表保存所有已生成而未考察的节点，CLOSED 表中记录已访问过的节点。

1. 访问路网中距离起始点最近且没有被检查过的点，把这个点放入 OPEN 组中等待检查。

2. 从 OPEN 表中找出距起始点最近的点，找出这个点的所有子节点，把这个点放到 CLOSE 表中。

3. 遍历考察这个点的子节点。求出这些子节点距起始点的距离值，放子节点到 OPEN 表中。

4. 重复第 2 和第 3 步,直到 OPEN 表为空，或找到目标点。

二、例题

Silver Cow Party

Time Limit: 2000MS **Memory Limit:** 65536K

Total Submissions: 5440 **Accepted:** 2353

Description

One cow from each of N farms ($1 \le N \le 1000$) conveniently numbered 1..N is going to attend the big cow party to be held at farm #X ($1 \le X \le N$). A total of M ($1 \le M \le 100{,}000$) unidirectional (one-way roads connects pairs of farms; road i requires T_i ($1 \le T_i \le 100$) units of time to traverse.

Each cow must walk to the party and, when the party is over, return to her farm. Each cow is lazy and thus picks an optimal route with the shortest time. A cow's return route might be different from her original route to the party since roads are one-way.

Of all the cows, what is the longest amount of time a cow must spend walking to the party and back?

Input

Line 1: Three space-separated integers, respectively: N, M, and X
Lines 2..M+1: Line i+1 describes road i with three space-separated integers: A_i, B_i,

and T_i. The described road runs from farm A_i to farm B_i, requiring T_i time units to traverse.

Output

Line 1: One integer: the maximum of time any one cow must walk.

Sample Input

```
4 8 2
1 2 4
1 3 2
1 4 7
2 1 1
2 3 5
3 1 2
3 4 4
4 2 3
```

Sample Output

```
10
```

Hint

Cow 4 proceeds directly to the party (3 units) and returns via farms 1 and 3 (7 units), for a total of 10 time units.

题解：

就是从各点到 X 的最短距离及从 X 到各点的最短距离和的最大值。

第一利用 dijstra 求出从 X 到各点的最短距离。

然后所有的边反向，再进行一次 dijstra 求 X 到各点的最短路径。

第二次求出的最短路径也就是各点到 X 的最短路径，因为边已经反向，对于第二次从 X 到各点的最短路径正是原图从各点到 X 的最短路径。

```
#include<iostream>
#include<queue>
using namespace std;
const int INF=0x7fffffff/100;
int map[1001][1001]={0};
int d[1001]={0},N,M,X;
int dd[1001]={0};
bool f[1001];
void dijstra()
{
	for(int i=1; i<=N; i++)
		d[i]=INF;
	d[X]=0;
	memset(f,0,sizeof f);
	for(int i=1; i<=N; i++)
	{
		int min=INF,u=0;
		for(int j=1; j<=N; j++)
			if(d[j]<min&&!f[j])
			{
				min=d[j];
				u=j;
			}
		f[u]=true;
		for(int j=1; j<=N; j++)
		{
			if(d[u]+map[u][j]<d[j])
			   d[j]=d[u]+map[u][j];
		}
	}

}

void traverse()
{
	int temp;
	for(int i=1; i<=N; i++)
	for(int j=1; j<i; j++)
	{
		temp = map[i][j];
		map[i][j]=map[j][i];
		map[j][i]=temp;
	}
}
```

```
int main()
{

    cin>>N>>M>>X;

    for(int i=1; i<=N; i++)
    for(int j=1; j<=N; j++)
            map[i][j]=INF;
    for(int i=1,s,t,value; i<=M; i++)
    {
          cin>>s>>t>>value;
          map[s][t]=value;
    }

    dijstra();
    for(int i=1; i<=N; i++)
             dd[i]=d[i];

    traverse();
    dijstra();
    int max=0;
    for(int i=1; i<=N; i++)
            if(d[i]+dd[i]>max)max=d[i]+dd[i];

    cout<<max<<endl;

    system("pause");
    return 0;
}
```

最小生成树

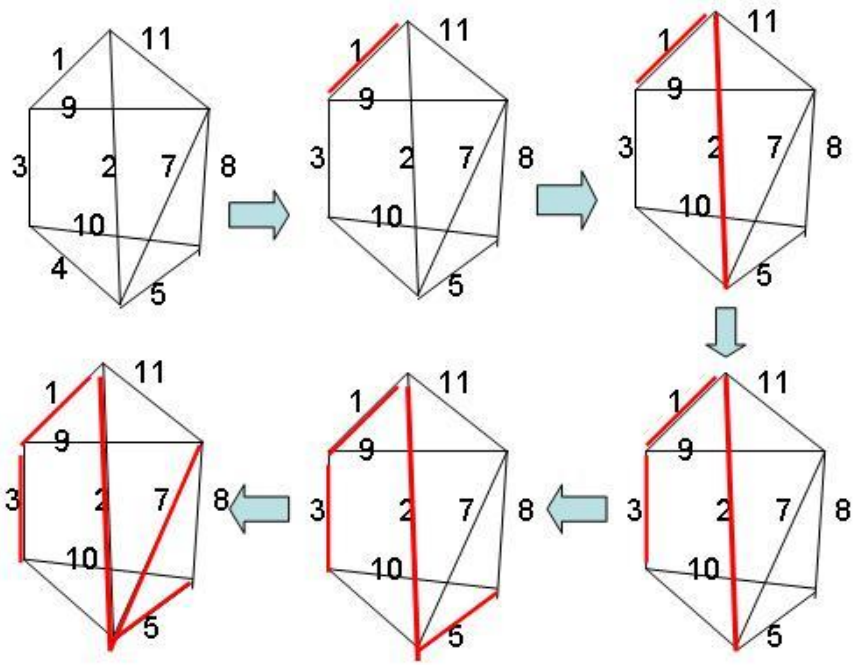
一、算法描述

定义：
1.一个连通且无回路的无向图称为树.
2.若图 G 的生成子图是一棵树,则该树称为 G 的生成树.
3.在图 G 的所有生成树中,树权最小的那棵生成树,称作最小生成树.

关于找出最小生成树的两种算法,一个称为 Kruskal(克鲁斯卡尔),另一个叫 Prim(普里姆)

(1) Kruskal 算法
step1: 选取最小权边 e1, 置边数 i=1
step2: 若 i=n-1 结束,否则转到 step3
step3: 设已选择边为 e1,e2,...ei 在 G 中选取不同于 e1,e2,...,ei 的边, 使{e1,e2,...,ei,ei+1}中无回路且 ei+1 是满足此条件的最小边.
step4: i= i+1 转 step2

一句话记住此算法: 在保证无回路前提下选 n-1 条最小权边.
如下图演示了此算法.



Kruskal算法

(2) Prim 算法(普里姆算法)

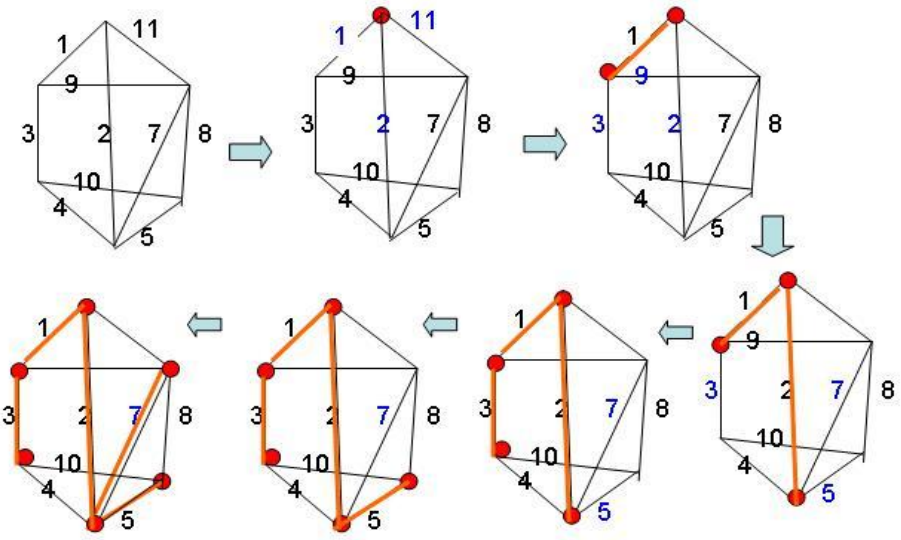
step1: 找出存在最小权边的点.

step2: 若所有顶点已经过完,则结束.否则跳 step3

step3: 通过已存在的点构成的树来计算出此树到其它未到达的点的最小权.

step4: 选取 step3 中标记的最小权边的顶点.转至 step2.

注:此步骤为自己总结的,如有错误请指正.^_^



Prim算法

二、例题

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

INPUT FORMAT

第一行： 农场的个数，N（3<=N<=100）。

第二行..结尾: 后来的行包含了一个 N*N 的矩阵,表示每个农场之间的距离。理论上，他们是 N 行，每行由 N 个用空格分隔的数组成，实际上，他们限制在 80 个字符，因此，某些行会紧接着另一些行。当然，对角线将会是 0，因为不会有线路从第 i 个农场到它本身。

SAMPLE INPUT (file agrinet.in)

```
4
0 4 9 21
4 0 8 17
9 8 0 16
21 17 16 0
```

OUTPUT FORMAT

只有一个输出，其中包含连接到每个农场的光纤的最小长度。

SAMPLE OUTPUT (file agrinet.out)

```
28
```

二分图的最大匹配匹配

一、算法描述

- 二分图又称作二部图，是图论中的一种特殊模型。
- 设 G=(V,E)是一个无向图。如顶点集 V 可分割为两个互不相交的子集，并且图中每条边依附的两个顶点都分属两个不同的子集。则称图 G 为二分图。
- 二分图又常记为(X,E,Y)，
  X,Y 是两个顶点集，
  E 是连接 X 和 Y 之间顶点
  的边的集合

- 给定一个二分图 G，在 G 的一个子图 M 中，M 的边集中的任意两条边都不依附于同一个顶点，则称 M 是一个匹配。
- 选择这样的边数最大的子集称为图的**最大匹配问题**(maximal matching problem),最大匹配的边数称为最大匹配数.
- 如果一个匹配中，图中的每个顶点都和图中某条边相关联，则称此匹配为**完全匹配**，也称作**完备匹配**。

二分图匹配的匈牙利算法

图采用矩阵存储，下标从 1 开始。

匈牙利匹配算法的理论和证明不复杂，就是不断寻找两个端点都是未浸润点的交替路径，把路径上的边匹配状态全部取反。每次操作，图的匹配数都增加 1。为方便描述，将二部图划分为 X 和 Y 两个部集。
此算法有几个性质：
1.算法只需以某一个部集（可以是 X 或 Y）中的所有未浸润点为交替路径的起始点，展开寻找。
2.X 中的某个点，只有在以它为起始点进行过交替路径搜索后，才有可能变为浸润点。
3.以 X 中的某个点为起始点，如果无法找到交替路径，那么以后不论何时以它为起始点，都不可能找到交替路径。
4.据 1,选择处理 X 集，由 2,3 知 X 集中的所有点最多只需处理一次。

伪代码：

```
SEARCH(Vi):
    SEARCH AUGMENT PATH STARTING FROM Vi
    IF FOUND THEN RETURN TRUE
    ELSE RETURN FALSE
MATCHING(G(X,Y)):
    ANS:=0
```

```
7    FOR EACH VERTEX Vi IN SET X
8       T:=SEARCH(Vi)
9       IF T=TRUE
10          ANS:=ANS+1
11      END IF
12   END FOR
```

寻找交替路径这个过程有 **BFS** 和 **DFS** 两种方式。

DFS:

```
1 int w[NX][NY]; //X 中的点到 Y 中的点的连接矩阵,w[i][j]是边<Vxi,Vyj>的权
2 int m[NY]; //Vyi 的匹配对象
3 int v[NY]; //在一次 DFS 中，Vyi 是否被访问过
4 bool dfs(int k){ //以 Vxk 为起点找交替路径
5    int i;
6    for(i=1;i<=N;i++){
7       if(!v[i] && w[k][i]){ //如果 Vyi 未访问过，而且 Vxk,Vyi 有边连接
8          v[i]=1;
9          if(!m[i] || dfs(m[i])){ //如果 Vyi 是未浸润点,或者以 Vyi 原来的匹配点为起始，
有扩张路径
10            m[i]=k;
11            return true; //扩张成功
12         }
13      }
14   }
15   return false;
16 }
```

这个算法也可以从类似"贪心"的角度理解：一个 X 中的点 Vx0，如果找到某个能到达的 Y 点 Vy0，就暂时把它"据为己有"，然后看 Y 的"原配 X 点"还能不能找到别的 Y 点配对，如果能，那么 Vx0 和 Vy0 就配对成功，否则不成功，Vx0 继续寻找别的 Vy。
Boycott 的理解：反正每次尽量添加一个匹配，不管这次匹配用的是 y 集合的哪个点，对下一次添加匹配没有影响，因为只要还没有达到最大匹配，就一定有一条增广路径（不管这条增广路径再长，也一定存在）。

BFS:
这是我在某些算法书上看到的 BFS 版本，需要用变量（或变量数组）tag 记录扩展目标。代码中，我改为用 que[i]的 bit1 记录：

```
1 int w[NX],ma[NY];
2 int que[NX+NY],pq,sq; //广搜队列
3 int pax[NX],pay[NY]; //记录交替路径
4 bool bfs(int r){
5    int i,j,k,tag; //tag，记录交替路径的下一步要扩展 X 中的点(tag==1 时)，还是 Y 中的
```

```
点(tag==0 时)
    memset(pax,0,sizeof(pax));
    memset(pay,0,sizeof(pay));
    que[0]=(r<<1);
    pq=0; sq=1;
    while(pq<sq){
        k=que[pq]>>1; tag=que[pq]&1;
        if(tag==0){ //process set X, look for unvisited vex in Y
            for(i=1;i<=N;i++){
                if(!pay[i] && w[k][i]){
                    pay[i]=k;
                    if(!ma[i]){ //是未浸润点，扩展路径搜索完毕
                        for(j=i;j;j=pax[pay[j]]){
                            ma[j]=pay[j];
                        }
                        return true;
                    }
                    else{ //这个 Y 点不是未浸润点,入队
                        que[sq++]=(i<<1)|1;
                    }
                }
            }
        }
        else{ //Vyk 不是未浸润点，路径必须沿着它所在的匹配边扩展
            que[sq++]=(ma[k]<<1);
            pax[ma[k]]=k;
        }
        pq++;
    }
    return false;
}
```

其实，在遇到浸润的 Vyi 时，由于下一步只能沿着它的匹配点 Vxj 走，即排队轮到 Vyi 时，必定是 Vxj 被加入队列。因此，只要令队列 que 仅存放 X 集的点，每次遇到浸润的 Vyi，把它的匹配点 Vxj 加入队列。这样就省去了分支，减小了代码量。

实现：

```
int w[NX],ma[NY];
int que[NX+NY],pq,sq;
int pax[NX],pay[NY];
bool bfs(int r){
    int i,j,k;
    memset(pax,0,sizeof(pax));
    memset(pay,0,sizeof(pay));
```

```
 8    que[0]=r;
 9    pq=0; sq=1;
10    while(pq<sq){
11       k=que[pq++];
12       for(i=1;i<=N;i++){
13          if(!pay[i] && w[k][i]){
14             pay[i]=k;
15             if(!ma[i]){ //free vex, augment path found
16                for(j=i;j;j=pax[pay[j]]){
17                   ma[j]=pay[j];
18                }
19                return true;
20             }
21             else{ //add Y's matched vex to que
22                pax[ma[i]]=i;
23                que[sq++]=ma[i];
24             }
25          }
26       }
27    }
28    return false;
29 }
```

显然，单纯的匹配问题，BFS 和 DFS 复杂度都是 O(mn)，但 DFS 编码难度小很多。
还有一种 Hopcroft-Karp 算法，复杂度是 O(msqrt(n))，只能用 BFS 实现，暂时还没深入研究。
然而，解决带权匹配问题时，DFS 只能做到 O(n^4)，而 BFS 可以做到 O(n^3)。所以掌握 BFS 的匈牙利依然有“实用价值”（其实真正搞学术，不应该用这种功利的标准去衡量价值）。

二、例题

完美的牛栏

农夫约翰上个星期刚刚建好了他的新牛棚，他使用了最新的挤奶技术。不幸的是，由于工程问题，每个牛栏都不一样。第一个星期，农夫约翰随便地让奶牛们进入牛栏，但是问题很快地显露出来：每头奶牛都只愿意在她们喜欢的那些牛栏中产奶。上个星期，农夫约翰刚刚收集到了奶牛们的爱好的信息（每头奶牛喜欢在哪些牛栏产奶）。一个牛栏只能容纳一头奶牛，当然，一头奶牛只能在一个牛栏中产奶。

给出奶牛们的爱好的信息，计算最大分配方案。

PROGRAM NAME: stall4
INPUT FORMAT
第一行　　两个整数，N (0 <= N <= 200) 和 M (0 <= M <= 200) 。N 是农夫约翰的奶牛数量，M 是新牛棚的牛栏数量。
第二行到第 N+1 行　　一共 N 行，每行对应一只奶牛。第一个数字 (Si) 是这头奶牛愿意在其中产奶的牛栏的数目 (0 <= Si <= M) 。后面的 Si 个数表示这些牛栏的编号。牛栏的编号限定在区间 (1..M) 中，在同一行，一个牛栏不会被列出两次。

SAMPLE INPUT (file stall4.in)
5 5
2 2 5
3 2 3 4
2 1 5
3 1 2 5
1 2

OUTPUT FORMAT
只有一行。输出一个整数，表示最多能分配到的牛栏的数量。

SAMPLE OUTPUT (file stall4.out)
4

题解：

给定两个顶点集合 X 与 Y 以及一些 X 中某顶点与 Y 中某顶点“匹配”的关系，求最多能找到多少个互不相交的匹配的顶点对，这就是二分匹配问题。USACO 中的 stall4 就是单纯的二分匹配问题。

解决二分匹配问题，可以将它转换成最大流问题求解，方法是：添加源点 S 和汇点 T，对于 X 集合中的每个顶点，添加从 S 出发的一条边，对于 X 集合中的每个顶点，添加指向 T 的一条边，对于 X 与 Y 的每个匹配关系，添加一条由 X 中顶点指向 Y 中顶点的边，以上边的权值均为 1。在这个图中求出的最大流的数值即与原最
大匹配的数值相等。

然而，使用一般的最大流算法求解二分匹配问题的效率是比较低的，由于最大流的代码一般都不短，前述方法在实际编程中的“性价比”不高。事实上，还有一种易于编程且效率很高的专门求解二分匹配问题的算法：Hungary 算法（也就是“匈牙利算法” ） 。它的基本思想是通过 DFS 在二分图中找“交错轨” 。

但事实上，我认为掌握这个算法甚至根本不需要理解“交错轨”这个概念，它似乎和“决策树” “隐式图”类似，只是为了理解算法的本质而抽象出来的一种东西，在代码中不会出现，
也不影响对代码的浅层理解和记忆。

算法的核心是 bool find(int v)这个函数，它的作用是：寻找顶点 v 可能匹配的顶点。对于每个可以与 v 匹配的顶点 j,假如它未被匹配,可以直接使用 j 与 v 匹配；假如 j 已与某顶点 x 匹配，那么只需调用 find(x)来求证 x 是否可以与其它顶点匹配,如果返回 true 的话,仍可以使 j 与 v 匹配;这就是一次 DFS。每次 DFS 时，要标记访问到的顶点（vis[j]=true） ，以防死循环和重复计算；每次 DFS 开始前所有的顶点都是未标记的。主过程只需对每个左侧的顶点调用 find，如果返回一次 true，就对最大匹配数加一；一个简单的循环就求出了最大匹配的数目。

```
{
PROG: stall4
LANG: PASCAL
}
var n,m,i,j,total:longint;//n 牛数 m 牛栏
cow:array[1..200,0..200]of longint;//cow[i,j] 第 i 只牛第 j 号喜欢的牛栏
link:array[1..200]of longint;//第 i 只牛栏住的牛
vis:array[1..200]of boolean;
function find(k:longint):boolean;
var i:longint;
begin
       for i:=1 to cow[k,0] do
           if not vis[cow[k,i]] then
           begin
                 vis[cow[k,i]]:=true;
                 if (link[cow[k,i]]=0)or(find(link[cow[k,i]])) then
                 begin
                         link[cow[k,i]]:=k;
                         exit(true);
                 end;
           end;
       exit(false);
end;
begin
       assign(input,'stall4.in');reset(input);
       assign(output,'stall4.out');rewrite(output);
```

```
        read(n,m);
        for i:=1 to n do
        begin
                read(cow[i,0]);
                for j:=1 to cow[i,0] do
                    read(cow[i,j]);
        end;
        close(input);
        total:=0;
        for i:=1 to n do
        begin
                fillchar(vis,sizeof(vis),false);
                if find(i) then inc(total);
        end;
        writeln(total);
    close(output);
end.
```

网络最大流问题

一、算法描述

所谓网络或容量网络指的是一个连通的赋权有向图 D＝（V、E、C），其中 V 是该图的顶点集，E 是有向边(即弧)集，C 是弧上的容量。此外顶点集中包括一个起点和一个终点。网络上的流就是由起点流向终点的可行流，这是定义在网络上的非负函数，它一方面受到容量的限制，另一方面除去起点和终点以外，在所有中途点要求保持流入量和流出量是平衡的。如果把下图看作一个公路网，顶点 v1…v6 表示 6 座城镇，每条边上的权数表示两城镇间的公路长度。现在要问 ：若从起点 v1 将物资运送到终点 v6 去 ，应选择那条路线才能使总运输距离最短？ 这样一类问题称为最短路问题 。如果把上图看作一个输油管道网 ， v1 表示发送点，v6 表示接收点，其他点表示中转站 ，各边的权数表示该段管道的最大输送量。现在要问怎样安排输油线路才能使从 v1 到 v6 的总运输量为最大。这样的问题称为最大流问题。

最大流理论是由福特和富尔克森于 1956 年创立的 ，他们指出最大流的流值等于最小割(截集)的容量这个重要的事实，并根据这一原理设计了用标号法求最大流的方法，后来又有人加以改进，使得求解最大流的方法更加丰富和完善 。最大流问题的研究密切了图论和运筹学，特别是与线性规划的联系，开辟了图论应用的新途径。

假设 $G(V,E)$ 是一个有限的有向图，它的每条边 (u,v)∈E 都有一个非负值实数的容量 $c(u,v)$ 。如果(u,v)不属于 E，我们假设 $c(u,v)=0$ 。我们区别两个顶点：一个源点 s 和一个汇点 t。一道网络流是一个对于所有结点 u 和 v 都有以下特性的实数函数 f:V×V→R

容量限制 (Capacity Constraints):	f(u,v)≤c(u,v)一条边的流不能超过它的容量。
斜对称 (Skew Symmetry):	f(u,v)=-f(v,u)由 u 到 v 的净流必须是由 v 到 u 的净流的相反（参考例子）。
流守恒 (Flow Conservation):	除非 $u=s$ 或 $u=t$，否则 Σ (w∈V) f(u,w)=0 一结点的净流是零，除了“制造”流的源点和“消耗”流的汇点。

注意 $f(u,v)$ 是由 u 到 v 的*净流*。如果该图代表一个实质的网络，由 u 到 v 有 4 单位的实际流及由 v 到 u 有 3 单位的实际流，那么 $f(u,v)=1$ 及 $f(v,u)=-1$。

边的**剩余容量（Residual Capacity）**是 $cf(u,v) = c(u,v) - f(u,v)$。这定义了以 $Gf(V,Ef)$表示的**剩余网络（Residual Network）**，它显示*可用的*容量的多少。留意就算在原网络中由 u 到 v 没有边，在剩余网络乃可能有由 u 到 v 的边。因为相反方向的流抵消，*减少*由 v 到 u 的流等于*增加*由 u 到 v 的流。**扩张路径（Augmenting Path）**是一条路径 (u1,u2...uk)，而 $u1 = s$、$uk = t$ 及 $cf(ui,ui + 1) > 0$，这表示沿这条路径传送更多流是可能的。

编辑本段

求最大流算法

1、augment path，直译为“增广路”，其思想大致如下：

原有网络为 G，设有一辅助图 G'，其定义为 V(G') = V(G)，E(G')初始值（也就是容量）与 E(G)相同。每次操作时从 Source 点搜索出一条到 Sink 点的路径，然后将该路径上所有的容量减去该路径上容量的最小值，然后对路径上每一条边<u,v>添加或扩大反方向的容量，大小就是刚才减去的容量。一直到没有路为止。此时辅助图上的正向流就是最大流。

我们很容易觉得这个算法会陷入死循环，但事实上不是这样的。我们只需要注意到每次网络中由 Source 到 Sink 的流都增加了，若容量都是整数，则这个算法必然会结束。

寻找通路的时候可以用 DFS，BFS 最短路等算法。就这两者来说,BFS 要比 DFS 快得多，但是编码量也会相应上一个数量级。

增广路方法可以解决最大流问题，然而它有一个不可避免的缺陷，就是在极端情况下每次只能将流扩大 1（假设容量、流为整数），这样会造成性能上的很大问题，解决这个问题有一个复杂得多的算法，就是预推进算法。

2、push label，直译为“预推进”算法。

3、压入与重标记(Push-Relabel)算法

除了用各种方法在剩余网络中不断找增广路(augmenting)的 Ford-Fulkerson 系的算法外，还有一种求最大流的算法被称为压入与重标记(Push-Relabel)算法。它的基本操作有：压入，作用于一条边，将边的始点的预流尽可能多的压向终点；重标记，作用于一个点，将它的高度（也就是 label）设为所有邻接点的高度的最小值加一。Push-Relabel 系的算法普遍要比 Ford-Fulkerson 系的算法快，但是缺点是相对难以理解。

Relabel-to-Front 使用一个链表保存溢出顶点，用 Discharge 操作不断使溢出顶点不再溢出。Discharge 的操作过程是：若找不到可被压入的临边，则重标记，否则对临边压入，直至点不再溢出。算法的主过程是：首先将源点出发的所有边充满，然后将除源和汇外的所有顶点保存在一个链

表里，从链表头开始进行 Discharge，如果完成后顶点的高度有所增加，则将这个顶点置于链表的头部，对下一个顶点开始 Discharge。

Relabel-to-Front 算法的时间复杂度是 O(V^3)，还有一个叫 Highest Label Preflow Push 的算法复杂度据说是 O(V^2*E^0.5)。我研究了一下 HLPP，感觉它和 Relabel-to-Front 本质上没有区别，因为 Relabel-to-Front 每次前移的都是高度最高的顶点，所以也相当于每次选择最高的标号进行更新。还有一个感觉也会很好实现的算法是使用队列维护溢出顶点，每次对 pop 出来的顶点 discharge，出现了新的溢出顶点时入队。

Push-Relabel 类的算法有一个名为 gap heuristic 的优化，就是当存在一个整数 0<k<V，没有任何顶点满足 h[v]=k 时，对所有 h[v]>k 的顶点 v 做更新，若它小于 V+1 就置为 V+1。

除此之外,还有：

最小费用最大流问题

有上下界的最大流问题

等等

二、例题

Drainage Ditches

Description

Every time it rains on Farmer John's fields, a pond forms over Bessie's favorite clover patch. This means that the clover is covered by water for awhile and takes quite a long time to regrow. Thus, Farmer John has built a set of drainage ditches so that Bessie's clover patch is never covered in water. Instead, the water is drained to a nearby stream. Being an ace engineer, Farmer John has also installed regulators at the beginning of each ditch, so he can control at what rate water flows into that ditch.

Farmer John knows not only how many gallons of water each ditch can transport per minute but also the exact layout of the ditches, which feed out of the pond and into each other and stream in a potentially complex network.

Given all this information, determine the maximum rate at which water can be transported out of the pond and into the stream. For any given ditch, water flows in only one direction, but there might be a way that water can flow in a circle.

Input

The input includes several cases. For each case, the first line contains two space-separated integers, N (0 <= N <= 200) and M (2 <= M <= 200). N is the number of ditches that Farmer John has dug. M is the number of intersections points for those ditches. Intersection 1 is the pond. Intersection point M is the stream. Each of the following N lines contains three integers, Si, Ei, and Ci. Si and Ei (1 <= Si, Ei <= M) designate the intersections between which this ditch flows. Water will flow through this ditch from Si to Ei. Ci (0 <= Ci <= 10,000,000) is the maximum rate at which water will flow through the ditch.

Output

For each case, output a single integer, the maximum rate at which water may emptied from the pond.

Sample Input

```
5 4
1 2 40
1 4 20
2 4 20
2 3 30
3 4 10
```

Sample Output

```
50
```

题解：

//dfs 法求增广路径的最大流算法,使用深度优先搜索策略实现最简单，算法复杂度上，若为整数网络，每进行一次搜索，至少可以使得答案增加 1(即网络中可能存在的最小的容量为 1 的边)，每次搜索的复杂度为 O(|E|),故若最大流为 c,总复杂度为 O(|E|c),此算法即为 Ford-Fulkerson 算法。

```
#include<stdio.h>
#include<string.h>
int i,s,e,vis[200],c[200][200],tp,sink,source,m,n;
int dfs(int x,int low)//x 为增广路径上的待扩展点，low 表示目前路径上的最小流，
                      //dfs 最终返回成功寻找到的增广路径上的最小流。
{
    int i,flow;
```

```
    if(x==sink)return low;
    for(i=0;i<m;i++)
        if(!vis[i]&&c[x][i]){//如果 i 点没有被访问，并且 x 与 i 点之间还有残余流，就执行下面过程
            vis[i]=1;//访问过就先占住，而且在递归后不把 vis[i]=0，
                    //因为这个点访问了以后就表示在一条增广路径上了,如果没有找到能扩展的下一个点就表示该条增广路径废了,
                    //如果最终能成功访问到汇点,表示找到增广路径,就可以返回此条路上一路比较下来得到的最小流(即这句:if(x==sink)return low;).
            if(flow=dfs(i,low<c[x][i]?low:c[x][i])){
                c[x][i]-=flow;c[i][x]+=flow;
                return flow;//成功找到增广路径以后,以上两行利用回溯的过程，从后往前把增广路径上的边都更新。
                            //更新完成后剩下的就是减去一条增广路径的残余网络.
                            //可以非常直观的想象当已经没法找到空隙继续增大流量的时候,整个流就是最大流了
            }
        }
    return 0;
}
int maxflow()
{
    int flow,ans=0;//vis 用来记录每次求增广路径时是否访问过某点
    do{
        memset(vis,0,sizeof(vis));
        vis[source]=1;
        flow=dfs(source,2147483647);
        ans+=flow;
    }while(flow);
    return ans;
}
int main()
{
    while(scanf("%d%d",&n,&m)!=EOF){
        memset(c,0,sizeof(c));
        for(i=0;i<n;++i){
            scanf("%d%d%d",&s,&e,&tp);
            c[s-1][e-1]+=tp; //c 是残留网络，初始残留网络就是读入的网络
        }
        source=0;sink=m-1;
        printf("%d\n",maxflow());
    }
    return 0;
}
```

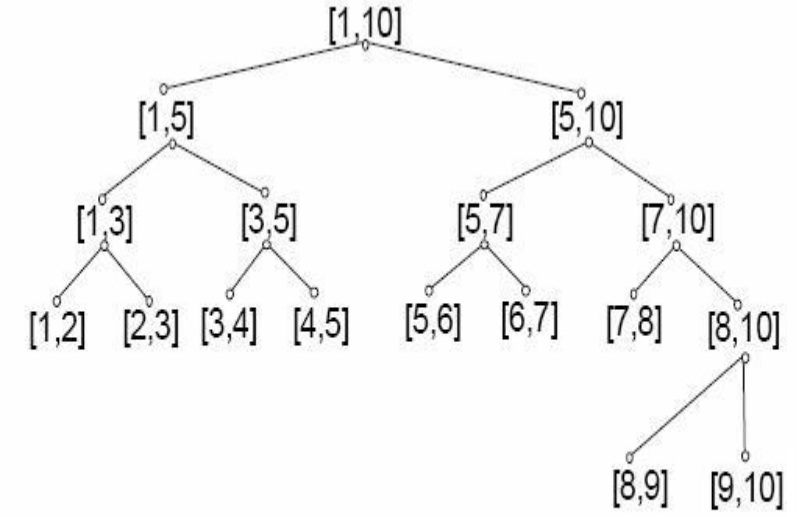
线段树

一、算法描述

线段树的定义

从简单说起，线段树其实可以理解成一种特殊的二叉树。但是这种二叉树较为平衡，和静态二叉树一样，都是提前已经建立好的树形结构。针对性强，所以效率要高。这里又想到了一句题外话：动态和静态的差别。动态结构较为灵活，但是速度较慢；静态结构节省内存，速度较快。

接着回到线段树上来，线段树是建立在线段的基础上，每个结点都代表了一条线段[a , b]。长度为 1 的线段成为元线段。非元线段都有两个子结点，左结点代表的线段为[a , (a + b) / 2]，右结点代表的线段为[(a + b) / 2 , b]。



图一

图一就是一棵长度范围为[1 , 10]的线段树。

长度范围为[1 , L] 的一棵线段树的深度为 log (L - 1) + 1。这个显然，而且存储一棵线段树的空间复杂度为 0（L）。

线段树支持最基本的操作为插入和删除一条线段。下面已插入为例，详细叙述，删除类似。

将一条线段[a , b] 插入到代表线段[l , r]的结点 p 中，如果 p 不是元线段，那么令 mid＝（l＋r）/2。如果 a<mid，那么将线段[a , b] 也插入到 p 的左儿子结点中，如果 b>mid，那么将线段[a , b] 也插入到 p 的右儿子结点中。

插入（删除）操作的时间复杂度为 0 (Log n)。

线段树的第一种变化

基本的线段树代表的是线段，如果我们把线段离散成点来看，那么线段树可以变化成一种离散类型线段树。

这里可以有两种理解。一种离散关系可以是连续的线段的点，比方说在一条直线上放置的连续小球的着色问题；另一种则是完全将线段离散化分成若干小段，对每一段小段做为元线段来建立线段树，这种线段树可以支持实数划分类型的线段。

线段树的第二种变化　（树状数组）

在结构上对线段树进行改变，可以得到线段树的另一种变化。

用 O（n）的一维数组构造出线段树，无其他附加空间。比方说，一棵从[0,L]的线段树表示为 TNode　Tree[L];

这里应用二进制将树进行划分。将整数 R 的二进制表示中的最后一个 1 换成 0，得到数 L。Tree[R]代表的线段就是[L,R]。例如：6 的二进制表示为（110）$_2$ 将最后一个 1 换成 0 即为（100）$_2$=4，所以 Tree[6]代表的线段就是[4,6]。

析出数 R 的最后一位 1 的方法是：LowBit（R）=R^~R。

包含点 L 的一系列数为 x1,x2,……，这里 x1=R，x2=x1+LowBit　(x1)，x3=x2+LowBit(x2)，……

这种线段树的优点在于：

1. 节省空间。完全线段长度的空间，无需左右指针，无需左右范围。
2. 线段树查找严格 log(R)，因为二进制的每位查找一遍。
3. 状态转移快，操作简单。
4. 扩展到多维较为容易。

缺点：

1. 随意表示线段[a,b]较为困难。

这种线段树适用于：

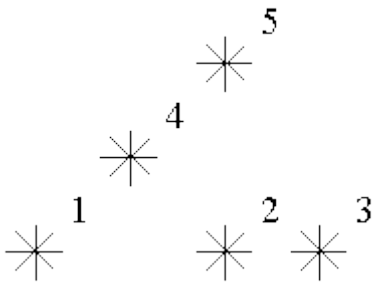
1. 查找线段[0,L]的信息。
2. 求线段[a,b]的和（应用部分和做差技术）。

二、例题

星星

给定星星的坐标(i,j)，每个星星的 level 定义为： 坐标为(i’,j’)的星星的个数，满足 i’<=i 且 j’<=j，不算自己.

问题：求每个 level 有多少个星星。



输入格式：第一行 n，表示星星个数，以下 n 行表示每个星星的坐标
输出格式：输出 n 行，level 分别为 0 到 n-1 的星星的个数

样例输入：
5
1 1
5 1
7 1
3 3
5 5

样例输出：
1
2
1
1
0

题解：

题中输入是按先 y 坐标升序，再按照 x 坐标排好的，就相当时一维了数据了。只需要在 x 轴上用线段树即可。
比如插入一个 x 坐标为 7 的星星,则要吧线段树中包含 7 的线段的节点值都加 1，这样，每个星星的 level 就是组成线段[1,x]的线段的节点值的和。

代码：
```
#include <iostream>
using namespace std;
#include <cstring>
#define MAXN 15000
#define MAX 32001

int n;
int x[MAXN], level[MAXN];
int tree[(MAX+1)*3];//*3 保险，此题*2 会 wa

int insert(int l,int r,const int &k,int pos)
{
    tree[pos] ++;
    if (l==r) return tree[pos];
    int mid = (l+r)>>1;
```

```
    if (k <= mid) insert(l,mid,k,(pos<<1));
    else return tree[pos<<1]+insert(mid+1,r,k,(pos<<1)+1);
}

int main(){
    int x, y;
    scanf("%d",&n);
    memset(level,0,sizeof(level));
    memset(tree,0,sizeof(tree));
    for (int i=0;i<n ;i++){
        scanf("%d%d",&x,&y);
        level[insert(1,MAX,x+1,1)-1]++;
    }
    for (int i=0 ;i<n ;i++) printf("%d\n",level[i]);
    system("pause");
    return 0;
}
```

字符串匹配

一、算法描述（KMP）

我们这里说的 KMP 不是拿来放电影的（虽然我很喜欢这个软件），而是一种算法。KMP 算法是拿来处理字符串匹配的。换句话说，给你两个字符串，你需要回答,B 串是否是 A 串的子串(A 串是否包含 B 串)。比如,字符串 A="I'm matrix67"，字符串 B="matrix"，我们就说 B 是 A 的子串。你可以委婉地问你的 MM：“假如你要向你喜欢的人表白的话，我的名字是你的告白语中的子串吗？”

解决这类问题，通常我们的方法是枚举从 A 串的什么位置起开始与 B 匹配，然后验证是否匹配。假如 A 串长度为 n，B 串长度为 m，那么这种方法的复杂度是 O (mn)的。虽然很多时候复杂度达不到 mn（验证时只看头一两个字母就发现不匹配了），但我们有许多“最坏情况”，比如，A="aaaaaaaaaaaaaaaaaaaaaaaaaaab"，B="aaaaaaaab"。我们将介绍的是一种最坏情况下 O(n)的算法（这里假设 m<=n），即传说中的 KMP 算法。

之所以叫做 KMP，是因为这个算法是由 Knuth、Morris、Pratt 三个提出来的，取了这三个人的名字的头一个字母。这时，或许你突然明白了 AVL 树为什么叫 AVL，或者 Bellman-Ford 为什么中间是一杠不是一个点。有时一个东西有七八个人研究过，那怎么命名呢？通常这个东西干脆就不用人名字命名了，免得发生争议，比如“3x+1 问题”。扯远了。

个人认为 KMP 是最没有必要讲的东西，因为这个东西网上能找到很多资料。但网上的讲法基本上都涉及到“移动(shift)”、“Next 函数”等概念，这非常容易产生误解（至少一年半前我看这些资料学习 KMP 时就没搞清楚）。在这里，我换一种方法来解释 KMP 算法。

假如，A="abababaababacb"，B="ababacb"，我们来看看 KMP 是怎么工作的。我们用两个指针 i 和 j 分别表示，A[i-j+ 1..i]与 B[1..j]完全相等。也就是说，i 是不断增加的，随着 i 的增加 j 相应地变化，且 j 满足以 A[i]结尾的长度为 j 的字符串正好匹配 B 串的前 j 个字符（j 当然越大越好），现在需要检验 A[i+1]和 B[j+1]的关系。当 A[i+1]=B[j+1]时，i 和 j 各加一；什么时候 j=m 了，我们就说 B 是 A 的子串（B 串已经整完了），并且可以根据这时的 i 值算出匹配的位置。当 A[i+1]<>B[j+1]，KMP 的策略是调整 j 的位置（减小 j 值）使得 A[i-j+1..i]与 B[1..j]保持匹配且新的 B[j+1]恰好与 A[i+1]匹配（从而使得 i 和 j 能继续增加）。我们看一看当 i=j=5 时的情况。

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B = a b a b a c b
j = 1 2 3 4 5 6 7
```

此时，A[6]<>B[6]。这表明，此时 j 不能等于 5 了，我们要把 j 改成比它小的值 j'。j'可能是多少呢？仔细想一下，我们发现，j'必须要使得 B[1..j]中的头 j'个字母和末 j'个字母完全相等（这样 j 变成了 j'后才能继续保持 i 和 j 的性质）。这个 j'当然要越大越好。在这里，B [1..5]="ababa"，头 3 个字母

和末 3 个字母都是"aba"。而当新的 j 为 3 时，A[6]恰好和 B[4]相等。于是，i 变成了 6，而 j 则变成了 4：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =         a b a b a c b
j =         1 2 3 4 5 6 7
```

从上面的这个例子，我们可以看到，新的 j 可以取多少与 i 无关，只与 B 串有关。我们完全可以预处理出这样一个数组 P[j]，表示当匹配到 B 数组的第 j 个字母而第 j+1 个字母不能匹配了时，新的 j 最大是多少。P[j]应该是所有满足 B[1..P[j]]=B[j-P[j]+1..j]的最大值。

再后来，A[7]=B[5]，i 和 j 又各增加 1。这时，又出现了 A[i+1]<>B[j+1]的情况：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =         a b a b a c b
j =         1 2 3 4 5 6 7
```

由于 P[5]=3，因此新的 j=3：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =               a b a b a c b
j =               1 2 3 4 5 6 7
```

这时，新的 j=3 仍然不能满足 A[i+1]=B[j+1]，此时我们再次减小 j 值，将 j 再次更新为 P[3]：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =                     a b a b a c b
j =                     1 2 3 4 5 6 7
```

现在，i 还是 7，j 已经变成 1 了。而此时 A[8]居然仍然不等于 B[j+1]。这样，j 必须减小到 P[1]，即 0：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =                       a b a b a c b
j =                     0 1 2 3 4 5 6 7
```

终于，A[8]=B[1]，i 变为 8，j 为 1。事实上，有可能 j 到了 0 仍然不

能满足 A[i+1]=B[j+1]（比如 A[8]="d"时）。因此，准确的说法是，当 j=0 了时，我们增加 i 值但忽略 j 直到出现 A[i]=B[1]为止。

这个过程的代码很短（真的很短），我们在这里给出：

```
j:=0;
for i:=1 to n do
begin
     while (j>0) and (B[j+1]<>A[i]) do j:=P[j];
     if B[j+1]=A[i] then j:=j+1;
     if j=m then
     begin
            writeln('Pattern occurs with shift ',i-m);
            j:=P[j];
     end;
end;
```

最后的 j:=P[j]是为了让程序继续做下去，因为我们有可能找到多处匹配。

这个程序或许比想像中的要简单，因为对于 i 值的不断增加，代码用的是 for 循环。因此，这个代码可以这样形象地理解：扫描字符串 A，并更新可以匹配到 B 的什么位置。

现在，我们还遗留了两个重要的问题：一，为什么这个程序是线性的；二，如何快速预处理 P 数组。

为什么这个程序是 O(n)的？其实，主要的争议在于，while 循环使得执行次数出现了不确定因素。我们将用到时间复杂度的摊还分析中的主要策略，简单地说就是通过观察某一个变量或函数值的变化来对零散的、杂乱的、不规则的执行次数进行累计。KMP 的时间复杂度分析可谓摊还分析的典型。我们从上述程序的 j 值入手。每一次执行 while 循环都会使 j 减小（但不能减成负的），而另外的改变 j 值的地方只有第五行。每次执行了这一行，j 都只能加 1；因此，整个过程中 j 最多加了 n 个 1。于是，j 最多只有 n 次减小的机会（j 值减小的次数当然不能超过 n，因为 j 永远是非负整数）。这告诉我们，while 循环总共最多执行了 n 次。按照摊还分析的说法，平摊到每次 for 循环中后，一次 for 循环的复杂度为 O(1)。整个过程显然是 O(n)的。这样的分析对于后面 P 数组预处理的过程同样有效，同样可以得到预处理过程的复杂度为 O(m)。

预处理不需要按照 P 的定义写成 O(m^2)甚至 O(m^3)的。我们可以通过 P[1],P[2],...,P[j-1]的值来获得 P[j]的值。对于刚才的 B="ababacb"，假如我们已经求出了 P[1],P[2],P[3]和 P[4]，看看我们应该怎么求出 P[5]和 P[6]。P[4]=2,那么 P [5]显然等于 P[4]+1,因为由 P[4]可以知道,B[1,2]已经和 B[3,4]相等了，现在又有 B[3]=B[5]，所以 P[5]可以由 P[4] 后面加一个字符得到。P[6]也等于 P[5]+1 吗？显然不是，因为 B[P[5]+1]<>B[6]。那么，我们要考虑“退一步”了。我们考虑 P[6]是否有可能由 P[5]的情况所包含的子串得到，即是否 P[6]=P[P[5]]+1。这里想不通的话可以仔细看一下：

```
          1 2 3 4 5 6 7
B = a b a b a c b
P = 0 0 1 2 3 ?
```

P[5]=3 是因为 B[1..3]和 B[3..5]都是"aba"；而 P[3]=1 则告诉我们，B[1]、B[3]和 B[5]都是"a"。既然 P[6]不能由 P[5]得到，或许可以由 P[3]得到（如果 B[2]恰好和 B[6]相等的话，P[6]就等于 P[3]+1 了）。显然，P[6]也不能通过 P[3]得到，因为 B[2]<>B[6]。事实上，这样一直推到 P[1]也不行，最后，我们得到，P[6]=0。

怎么这个预处理过程跟前面的 KMP 主程序这么像呢？其实，KMP 的预处理本身就是一个 B 串“自我匹配”的过程。它的代码和上面的代码神似：

```
P[1]:=0;
j:=0;
for i:=2 to m do
begin
     while (j>0) and (B[j+1]<>B[i]) do j:=P[j];
     if B[j+1]=B[i] then j:=j+1;
     P[i]:=j;
end;
```

最后补充一点：由于 KMP 算法只预处理 B 串，因此这种算法很适合这样的问题：给定一个 B 串和一群不同的 A 串，问 B 是哪些 A 串的子串。

串匹配是一个很有研究价值的问题。事实上，我们还有后缀树，自动机等很多方法，这些算法都巧妙地运用了预处理，从而可以在线性的时间里解决字符串的匹配。我们以后来说。

Matrix67 原创

二、例题

Power Strings

Time Limit: 3000MS **Memory Limit:** 65536K

Total Submissions: 14390 **Accepted:** 6029

Description

Given two strings a and b we define a*b to be their concatenation. For example, if a = "abc" and b = "def" then a*b = "abcdef". If we think of concatenation as

multiplication, exponentiation by a non-negative integer is defined in the normal way: a^0 = "" (the empty string) and a^(n+1) = a*(a^n).

Input

Each test case is a line of input representing s, a string of printable characters. The length of s will be at least 1 and will not exceed 1 million characters. A line containing a period follows the last test case.

Output

For each s you should print the largest n such that s = a^n for some string a.

Sample Input

```
abcd
aaaa
abababab
.
```

Sample Output

```
1
4
3
```

Hint

This problem has huge input, use scanf instead of cin to avoid time limit exceed.

题解：

题意： a 是一个字符串，记 s=a^n 为 a 重复 n 次所形成的字符串。比如说 a 是 abcd，那么当 n=3 时，a^3 就是 abcdabcdabcd。现在给出字符串 s，求出最大的重复次数 n。

解法： 为了求最短子串的长度，考虑到 KMP 中若匹配失败也会进行最短位移，因此可以利用 KMP 的 next 数组求得最短子串的长度。

而求最短位移，最简单的办法就是若第 n+1 个匹配失败会跳转到 next[n+1]，n+1-next[n+1]即为解。

由于下表从 0 开始，结果为 d = len-next[len]。

然后 mod 一下看看能否整除，能救输出 len/d，否则输出 1.

by Boycott

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

char S[1100000];
int next[1100000];

void get_next(char* T,int* next){
    int i=0, j=-1, lenT=strlen(T);
    next[0]=-1;
    while(i<lenT) {
        if(j==-1 || T[i]==T[j]) { ++i; ++j; next[i]=j; }
        else j=next[j];
    }
}

int main(){
    while(scanf("%s", S)!=EOF){
        if(strcmp(S, ".")==0) break;
        get_next(S, next);
        int slen = strlen(S);
        int alen = slen - next[slen];
        if(slen%alen == 0) printf("%d\n", slen/alen);
        else printf("1\n");
    }
    return 0;
}
```

数论/数学相关

一、算法描述

（1）欧几里得/扩展欧几里得算法（经典）
（2）求解同余方程（组）（经典）
（3）费马小定理
（4）欧拉函数
（5）高斯消元
（6）置换群

欧几里德算法概述：

欧几里德算法又称辗转相除法，用于计算两个整数 a,b 的最大公约数。其计算原理依赖于下面的定理：

gcd 函数就是用来求(a,b)的最大公约数的。

gcd 函数的基本性质：

gcd(a,b)=gcd(b,a)=gcd(-a,b)=gcd(|a|,|b|)

欧几里得算法的公式表述

gcd(a,b)=gcd(b,a mod b)

证明：a 可以表示成 a = kb + r，则 r = a mod b

假设 d 是 a,b 的一个公约数，则有

d|a, d|b，而 r = a - kb，因此 d|r

因此 d 是(b,a mod b)的公约数

假设 d 是(b,a mod b)的公约数，则

d | b , d |r ，但是 a = kb +r

因此 d 也是(a,b)的公约数

因此(a,b)和(b,a mod b)的公约数是一样的，其最大公约数也必然相等，得证。

欧几里德算法代码：

```
int Gcd(int a, int b)
{
    if(b == 0)    return a;
    return Gcd(b, a % b);
}
```

扩展的欧几里德算法:

给定的两个正整数 a, b，计算它们的最大公因子 d 和两个整数 a 和 b，使得 ax+by=d

要证明这个定理，首先需给出一个定理

带余除法：

若 a, b 是两个整数，其中 b>0, 则存在两个整数 q, r，使得

$$a = b*q + r \quad 0<=r<b$$

, 并且 q 和 r 是唯一的。

这个定理证明是简单的，有了这个定义，我们就来证明欧几里德算法

```
a = q0*b + r0
b = q1*r0 + r1
r0= q2*r1 + r2
  ...
rn-3=qn-1*rn-2 + rn-1
rn-2 = qn * rn-1 + rn(rn = 0)
```

这里 rn-1 就是最大公约数。

现在就需要反推回去，可以得到 r 可以用 a,b 线性表示

```
Ri=(Xi-2-Qi*Xi)*a-(Yi-2-Qi*Yi-1)*b
```

下一步就是求 x,y 的值。

```
Xi=Xi-2-Qi*Xi-1
Yi=Yi-2-Qi*Yi-1
```

而我们很容易得到 x0,y0,x1,y1 的值，那么递推，我们就可以得到 x,y 的值

算法描述

```
E1 x0=0, y0=1;x1=1, y0=0, c=a, d=b
E2 q<-c/d r<-c%d
E3 if r=0
```

```
        return
      else E4
E4
        x=x0-q*x1
        x0=x1;x1=x
        y=y0-q*y1
        y0=y1;y1=y
```

扩展的欧几里德代码

```
int exGcd(int a, int b, int &x, int &y)
{
      if(b == 0)
      {
              x = 1;
              y = 0;
              return a;
      }
      int r = exGcd(b, a % b, x, y);
      int t = x;
      x = y;
      y = t - a / b * y;

      return r;
}
```