信奥编程讲义-数据结构

（8 学阶段）

陈琰宏

目　　录

第 1 讲 动态规划

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（最大值或最小值）的那个解。设计一个动态规划算法，通常可以按以下几个步骤进行：

（1）找出最优解的性质，并刻画其结构特征。

（2）递归地定义最优值。

（3）以自底向上的方式计算出最优值。

（4）根据计算最优值时得到的信息，构造一个最优解。

1.1 二维 DP

1 [3889] 长江一日游

长江游艇俱乐部在长江上设置了 n 个游艇出租站，游客可以在这些出租站租用游艇，并 在下游的任何一个游艇出租站归还游艇。游艇出租站 i 到游艇出租站 j 之间的租金为 r（i， j）。 现在要求出从游艇出租站 1 到游艇出租站 n 所需的少的租金。

输入 T，表示有 T 组数据（1<=T<=10）；输入 n，代表 n 个游艇出租站(1<=n<=20)

输入 n-1 行，第 i 行输入 a[i]----a[n-i]，代表游艇出租站 i 到游艇出租站 j 之间的租金为 r（i， j）

输出问从游艇出租站 1 到游艇出租站 n 所需的少的租金

样例输入	样例输出
1 6 2 6 9 15 20 3 5 11 18 3 6 12 5 8 6	15

分析：长江游艇俱乐部在长江上设置了 n 个游艇出租站，游客可以在这些出租站租用游艇，并 在下游的任何一个游艇出租站归还游艇。游艇出租站 i 到游艇出租站 j 之间的租金为 r（i，j）。

现在要求出从游艇出租站 1 到游艇出租站 n 所需的最少的租金。

当要租用游艇从一个站到另外一个站时，中间可能经过很多站点，不同的停靠站策略就有不同的租金。那么我们可以考虑该问题，从第 1 站到第 n 站的最优解是否一定包含前 n−1 的最优解，即是否具有最优子结构和重叠性。如果是，就可以利用动态规划进行求解。

假设第 i 个站点到第 j 个站点（i，i+1，…，j）的最 优解是 c，子问题（i，i+1，…，k）的最优解是 a，子问 题（k，k+1，…，j）的最优解是 b，那么 c=a+b，无论两个子问题的停靠策略如何都不影响 它们的结果，因此我们只需要证明如果 c 是最优的，则 a 和 b 一定是最优的（即原问题的最 优解包含子问题的最优解）

当 j=i 时，只有 1 个站点，m[i][j]=0。

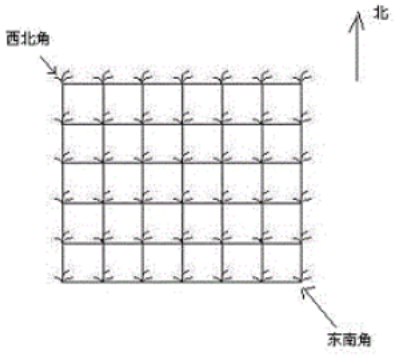
当 j=i+1 时，只有 2 个站点，m[i][j]= r[i][j]。

当 j> i+1 时，有 3 个以上站点，m[i][j]=min(m[i][k]+m[k][j],r[i][j]) j>i+1, k>i&&k<j

```
#include<bits/stdc++.h>
using namespace std;
int main()
{

   int n,t;
   int m[200][200] = { 0 };
   cin>>t;
   for(int p=1;p<=t;p++){
      cin >> n;
      memset(m,0,sizeof(m));
      for (int i = 1; i <= n - 1; i++)
      {//初始化，将所有 r(i,j)都先存在数组 m 中
         for (int j = i + 1; j <= n; j++)
         {
            cin >> m[i][j];
         }
      }

      for (int r = 2; r <= n; r++)
      {//接着从长度为 2 的开始找较优解（比如说从 1 到 2 就是长度为 2 的），直到找到长度为 n 的
         for (int i = 1; i <= n - r + 1; i++)
         {
            int j = i + r - 1;// r(i，j)的长度为 r
            for (int k = i; k <= j; k++)
            {//在 i 到 j 中找某一站 k，使得 r(i,k)+r(k,j)最小
               int t = m[i][k] + m[k][j];
               if (t < m[i][j])
               {
                  m[i][j] = t;//用较优解替换原来的 r(i,j)
               }
            }
         }
      }
      cout << m[1][n] << endl;

   }
}
```

2 [2836]摘花生

Hello Kitty 想摘点花生送给她喜欢的米老鼠。她来



到一片有网

格状道路的矩形花生地(如下图)，从西北角进去，东南角出来。地里每个道路的交叉点上都有种着一株花生苗，上面有若干颗花生，经过一株花生苗就能摘走该它上面所有的花生。Hello Kitty 只能向东或向南走，不能向西或向北走。问 Hello Kitty 最多能够摘到多少颗花生。

输入 第一行是一个整数 T，代表一共有多少组数据。1≤T≤100 接下来是 T 组数据。每组数据的第一行是两个整数，分别代表花生苗的行数 R 和列数 C(1≤R,C≤100)每组数据的接下来 R 行数据，从北向南依次描述每行花生苗的情况。每行数据有 C 个整数，按从西向东的顺序描述了该行每株花生苗上的花生数目 M(0≤M≤1000)。

输出

对每组输入数据，输出一行，内容为 Hello Kitty 能摘到得最多的花生颗数。

样例输入	样例输出
2 2 2 1 1 3 4 2 3 2 3 4 1 6 5	8 16

题解：找出每个阶段最优解，和经典数塔问题很像。每个点无非就是从它上面或者左边走过来，选择较大的一个就好。状态转移方程：

f[i][j] 从（1，1）走到（i,j）的最大花生数

f[i][j]+=max(f[i-1][j],f[i][j-1])

i-1，j 即为从上面过来；i，j-1 即为从左边过来。最后输出出口处即可。

也可以换一种考虑，每个点的最大花生数等于走东和走南两条路的最大花生数，终点的花生数就是她本身的花生数量，从终点依次往上推，就可以得出起始点的最大数量

```
dp[i][j]=max(dp[i-1][j],dp[i][j-1])+ma[i][j];
```

```
#include<algorithm>
using namespace std;
int a[105][105],f[105][105];
int c,r,maxn;
int main()
{
    int s;
    scanf("%d",&s);
    for(int i=0;i<s;i++)//因为有多组数据，所以加
    {
        scanf("%d%d",&c,&r);
        for(int i=1;i<=c;i++)
            for(int j=1;j<=r;j++)
            {
                scanf("%d",&a[i][j]);
                f[i][j]=a[i][j];//其实不用 f 数组直接用 a 数组也可以
```

```
            }
        for(int i=1;i<=c;i++)
            for(int j=1;j<=r;j++)
                f[i][j]+=max(f[i-1][j],f[i][j-1]);//递推即可
        printf("%d\n",f[c][r]);//输出
    }
}
```

3 [2839] 最低通行费

一个商人穿过一个 N×N 的正方形的网格，去参加一个非常重要的商务活动。他要从网格的左上角进，右下角出。每穿越中间 1 个小方格，都要花费 1 个单位时间。商人必须在(2N-1)个单位时间穿越出去。而在经过中间的每个小方格时，都需要缴纳一定的费用。这个商人期望在规定时间内用最少费用穿越出去。请问至少需要多少费用？

注意：不能对角穿越各个小方格（即，只能向上下左右四个方向移动且不能离开网格）.

输入 第一行是一个整数，表示正方形的宽度 N (1≤N<100)；后面 N 行，每行 N 个不大于 100 的整数，为网格上每个小方格的费用。

输出至少需要的费用。

样例输入	样例输出
5 1 4 6 8 10 2 5 7 15 17 6 8 9 18 20 10 11 12 19 21 20 23 25 29 33	109

提示

样例中，最小值为 109=1+2+5+7+9+12+19+21+33。

动态规划(二维数组左上到右下进行规划)，用一个二维数组 result[i][j]存储，到达位置 i，j 所需要的最小代价，从底(dp[0][0])向上(dp[n-1][n-1])更新数据，基于题目设定，到某位置只能从其左边过来，或者从其上面过来，每次取这两种方案中代价较小的

```
dp[i][j]=min(dp[i-1][j],dp[i][j-1])+ma[i][j];
```

```
#include<bits/stdc++.h>
using namespace std;
int main(){
  int ma[111][111];
  int dp[111][111];//dp[i][j]表示走到 i，j 点上的最优值
  int t;
  int n,m;
    cin>>n;
    memset(dp,0,sizeof(dp));
    for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++)
```

```
    cin>>ma[i][j];
    dp[1][1]=ma[1][1];
    for(int i=2;i<=n;i++)//先处理边缘上的点
    {
      dp[i][1]=dp[i-1][1]+ma[i][1];
    }
    for(int i=2;i<=n;i++)
    {
      dp[1][i]=dp[1][i-1]+ma[1][i];
    }
    for(int i=2;i<=n;i++)//从 2,2 向 n，m 处理，因为都是由左或上走来，所以通过这两个方向的格子更新到 i，
j 格子
    {
      for(int j=2;j<=n;j++)
      {
        dp[i][j]=min(dp[i-1][j],dp[i][j-1])+ma[i][j];
      }
    }
    cout<<dp[n][n]<<endl;
  return 0;
}
```

1.2 区间 DP

区间 DP 是一类在区间上进行动态规划的最优问题，一般是根据问题设计一个表示状态的 dp，然后将问题划分成两个子问题，也就是对于每段区间，将它们划分为几段更小区间直至一个元素组成的区间，枚举他们的组合 ，求合并后的全局最优解，然后得解。是分治思想的一种应用。
这类 DP 可以用常规的 for 循环来写，也可以用记忆化搜索来写。
推荐：记忆化搜索写法，因为这种写法相当易懂，需要什么值可以直接去记忆化搜索一下，随叫随到随用。

[1] 2826 合并石子

题目链接：http://acm.ocrosoft.com/problem.php?id=2826
题目描述
在一个操场上一排地摆放着 N 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。
计算出将 N 堆石子合并成一堆的最小得分。
输入第一行为一个正整数 N (2≤N≤100)；以下 N 行,每行一个正整数，小于 10000，分别表示第 i 堆石子的个数(1≤i≤N)。
输出 一个正整数，即最小得分。

样例输入	样例输出
7 13	239 3 1

7 8 16 21 4 18	

本题初看可以使用贪心法来解决，但是因为有必须相邻两堆才能合并这个条件在，用贪心法就无法保证每次都能取到所有堆中石子数最少（最多）的两堆。 下面以操场玩法为例：假设有 n=6 堆石子，每堆的石子个数分别为 3、4、6、5、4、2。

如果使用贪心法求最小花费，应该是如下的合并步骤：

第 1 次合并 3 4 6 5 4 2　　2，3 合并花费是 5

第 2 次合并 5 4 6 5 4　　5，4 合并花费是 9

第 3 次合并 9 6 5 4　　5，4 合并花费是 9

第 4 次合并 9 6 9　　9，6 合并花费是 15

第 5 次合并 15 9　　15，9 合并花费是 24

总得分＝5＋9＋9＋15＋24＝62

但是如果采用如下合并方法，却可以得到比上面花费更少的方法：

第 1 次合并 3 4 6 5 4 2　　3，4 合并花费是 7

第 2 次合并 7 6 5 4 2　　7，6 合并花费是 13

第 3 次合并 13 5 4 2　　4，2 合并花费是 6

第 4 次合并 13 5 6　　5，6 合并花费是 11

第 5 次合并 13 11　　13，11 合并花费是 24

总花费＝7＋13＋6＋11＋24＝61

显然利用贪心法来求解错误的，贪心算法在子过程中得出的解只是局部最优，而不能保证全局的值最优，因此本题不可以使用贪心法求解。

如果使用暴力穷举的办法，会有大量的子问题重复，这种做法是不可取的，那么是否可以使用动态规划呢？我们要分析该问题是否具有最优子结构性质，它是使用动态规划的必要条件。

1．路边玩法

如果 n-1 次合并的全局最优解包含了每一次合并的子问题的最优解，那么经这样的 n-1 次合并后的花费总和必然是最优的，因此我们就可以通过动态规划算法来求出最优解。

首先分析该问题是否具有最优子结构性质。

（1）分析最优解的结构特征

• 假设已经知道了在第 k 堆石子分开可以得到最优解，那么原问题就变成了两个子问题，子问题分别是{ai，a2，…，ak}和{ak+1，…，aj}，如图 4-75 所示。那么原问题的最优解是否包含子问题的最优解呢？

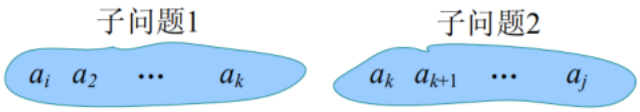


图 4-75　原问题分解为子问题

• 假设已经知道了 n 堆石子合并起来的花费是 c，子问题 1{ai，a2，…，ak}石子合并起来的花费是 a，子问题 2{ak+1，…，aj}石子合并起来的花费是 b，{ai，a2，…，aj}石子数量之和是 w（i，j），那么 c=a+b+ w（i，j）。因此我们只需要

证明如果 c 是最优的，则 a 和 b 一定是最优的（即原问题的最优解包含子问题的最优解）。

（2）建立最优值递归式

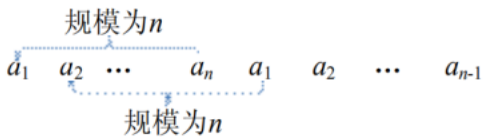
设 Min[i][j]代表从第 i 堆石子到第 j 堆石子合并的最小花费，Min[i][k]代表从第 i 堆石子 到第 k 堆石子合并的最小花费，Min[k+1][j]代表从第 k+1 堆石子到第 j 堆石子合并的最小花 费，w（i，j）代表从 i 堆到 j 堆的石子数量之和。列出递归式：

$$Min[i][j]=\begin{cases}0 & ,i=j\\ \min\limits_{i\le k<j}(Min[i][k]+Min[k+1][j]+w(i,j)) & ,i<j\end{cases}$$

Max[i][j] 代表从第 i 堆石子到第 j 堆石子合并的最大花费，Max[i][k] 代表从第 i 堆石子到第 k 堆石子合并的最大花费，Max[k+1][j] 代表从第 k+1 堆石子到第 j 堆石子合并

2．操场玩法

如果把路边玩法看作直线型石子合并问题，那么操场玩法就属于圆型石子合并问题。圆型石子合并经常转化为直线型来求。也就是说，把圆形结构看成是长度为原规模两倍的直线结构来处理。如果操场玩法原问题规模为 n，所以相当于有一排石子 a1，a2，…，an，a1，a2，…，an−1，该问题规模为 2n−1，如图 4-76 所示。然后就可以用线性的 石子合并问题的方法求解，求最大值的方法和求最小值的方法是一样的。最后，从规模是 n 的最优值找出最小值或最大值即可。



2826 合并石子，这道题就是一道典型的区间 dp，大体更新 dp 数组的思路就是对于很多堆石子，我们可以从中选取一个节点，左边先合成一堆，右边合成一堆，从这些情况中选取最优值来更新这么多堆总共合成一堆的最优值。

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<cmath>
using namespace std;
#define LL long long

int dp[111][111];//dp[i][j]表示区间[i,j]的合并最小花费
int sum[111];//sum[i]表示区间[1,i]的和，这里对整个序列求个前缀和方便后面处理区间求和
int a[111];//保存输入的 n 的数

int main()
{
  int n;//数字个数
  while (~scanf("%d", &n)) {
    sum[0] = 0;
    for (int i = 1; i <= n; i++) {
```

```
            scanf("%d", &a[i]);
            sum[i] = sum[i - 1] + a[i];  //求区间[1,i]的和
        }
        memset(dp, 0x3f3f3f3f, sizeof dp);  //先初始化 dp 数组为一个很大的值
        for (int i = 1; i <= n; i++)  //初始化长度为 1 的区间
            dp[i][i] = 0;
        for (int x = 2; x <= n; x++) {  //枚举区间长度
            for (int i = 1; i + x - 1 <= n; i++) {  //枚举区间左端点，则右端点为 i+x-1
                int j = i + x - 1;  //令右端点为 j
                int tol = sum[j] - sum[i - 1];//求出区间[i,j]的和 等于[1,j]的和减去[1,i-1]的和
                for (int k = i; k < j; k++) {//枚举断点
                    dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j] + tol);
                        //把每个断点算出来的值和原本答案作比较
                }
            }
        }
        printf("%d\n", dp[1][n]);//dp[1][n]就是答案了
    }
    return 0;
}
```

[2] 2827 乘积最大

今年是国际数学联盟确定的“2000——世界数学年”，又恰逢我国著名数学家华罗庚先生诞辰 90 周年。在华罗庚先生的家乡江苏金坛，组织了一场别开生面的数学智力竞赛的活动，你的一个好朋友 XZ 也有幸得以参加。活动中，主持人给所有参加活动的选手出了这样一道题目：

设有一个长度为 N 的数字串，要求选手使用 K 个乘号将它分成 K+1 个部分，找出一种分法，使得这 K+1 个部分的乘积最大。

同时，为了帮助选手能够正确理解题意，主持人还举了如下的一个例子：

有一个数字串：312， 当 N=3，K=1 时会有以下两种分法：

1）3*12=36

2）31*2=62

这时，符合题目要求的结果是：31*2=62。

现在，请你帮助你的好朋友 XZ 设计一个程序，求得正确的答案。

输入

第一行共有 2 个自然数 N，K（6≤N≤10，1≤K≤6）

第二行是一个长度为 N 的数字串。

输出

输出所求得的最大乘积（一个自然数）。

样例输入

4 2

1231

样例输出

62

思路

这道题就是一道典型的区间 dp，大体更新 dp 数组的思路就是对于一段连续的数字中插入一个乘号来更新数组，dp 前我们需要先处理每一个连续字符直线代表的数字是多少，然后开始枚举插入乘号的数量，然后是枚举该乘号能插在哪些位置，因为位置是连续的，那么还需要枚举这段连续的位置，然后定位乘号插入的位置后开始获取插入后的乘积去更新答案。

```
#include<iostream>
using namespace std;
int f[41][7],a[41][41];
int n,r;
char s1[255],s2[255];
int main()
{
  scanf("%d%d ",&n,&r);
  cin>>s1;
  for (int i=0;i<n;i++)
    for (int j=i;j<n;j++)
    {
      strncpy(s2,s1+i,j-i+1);//将字符串 s1 的 i 开始向后 j-i+1 个字符赋予字符串 s2
      s2[j-i+1]='\0';//将 s2 尾部封上，即弃掉 j-i+1 之后的字符
      a[i+1][j+1]=atoi(s2);//将 s2 字符串转化为数字型
    }
    //a[i][j]表示从 i 到 j 的字符串表示的数字
  for (int i=1;i<=n;i++)
      1                    //f[i][j]表示前 i 位字符里插入 j 个乘号能得到的最大值
  for (int k= 2            )   //枚举插入多少个乘号
    for (int i= 3            )//枚举输入第 i 个乘号的位置
      for (int j= 4            )
        f[i][k]= 5
  printf("%d\n",f[n][r]);
  return 0;
}
```

[3] 2818 机器分配

总公司拥有高效设备 M 台，准备分给下属的 N 个分公司。各分公司若获得这些设备，可以为国家提供一定的盈利。问：如何分配这 M 台设备才能使国家得到的盈利最大？求出最大盈利值。其中 M≤15，N≤10。分配原则：每个公司有权获得任意数目的设备，但总台数不超过设备数 M。

输入：第一行有两个数，第一个数是分公司数 N，第二个数是设备台数 M；接下来是一个 N*M 的矩阵，表明了第 I 个公司分配 J 台机器的盈利。

输出： 第一行输出最大盈利值；接下 N 行，每行有 2 个数，即分公司编号和该分公司获得设备台数。

样例输入	样例输出
3 3	70
30 40 50	1 1
20 30 50	2 1
20 25 30	3 1

思路

这道题是类似于区间 dp 的一道题，它的解题思想类似于区间 dp，f 数组的更新策略和区间 dp 相同，对于每一个 i 去更新 j 时，是通过在 0 到 j-1 中找到一个最优节点去更新 i，j 的状态，对于这道题就是对于第 i 个公司分配多少机器进行最优选择，前 i 个公司一共分配 j 台机器，那么第 i 个公司就是分配 0-j 个机器中选取最优值。

```
#include<cstdio>
#include<iostream>
using namespace std;
int n,m,max1;
int f[11][16],value[11][16];
void show(int i,int j)   //输出最优值路径函数
{
  if (i==0) return;
  for (int k=0;k<=j;k++) //根据求最大值的方法，来找到对应的每个公司需要多少台机器
    if(max1==f[i-1][k]+value[i][j-k])
    {
      max1=f[i-1][k]; //找到了之后就改变 max 相当于 max=max-value[i][j-k]
      show(i-1,k);//继续寻找前 i-1 个公司怎么分配 k 台机器
      printf("%d %d\n",i,j-k);
      return ;
    }
}
int main()
{
  scanf("%d%d",&n,&m);
  for (int i=1;i<=n;i++)
    for (int j=1;j<=m;j++)
      scanf("%d",&value[i][j]);  //输入权值矩阵
  for (int i=1;i<=n;i++)   //f[i][j]表示 i 个公司分配了 j 台机器的最优值
    for (int j=1;j<=m;j++)
    {
      max1=0;
      for (int k=0;k<=j;k++)//更新 f 数组的策略是对于 f[i][j]中的 j 进行遍历，找到某个 k 值，让第 i 个公司拿 j-k 台机器，以此来更新 f 数组
        if (f[i-1][k]+value[i][j-k]>max1) max1=f[i-1][k]+value[i][j-k];
      f[i][j]=max1;
    }
    //max 此时为最优答案
```

```
    printf("%d\n",f[n][m]);
    show(n,m);
    return 0;
}
```

课前练习

1. 读程序写结果

```
#include <iostream>
using namespace std;
int main()
{
    int s, i, j;
    int a[11];
    s = 0;
    i = 0;
    for (j = 1; j <= 10; j++)
        a[j] = 1;
    while (i < 10)
    {
        i = i + 1;
        s = s + a[i];
        for (j = i + 1; j <= 10; j++)
            a[j] = a[j] + 1;
    }
    cout << "s=" << s << endl;
    return 0;
}
```

输出：_____

2. 一元一次方程求解

算式 x=(c-b)/a 来计算出未知数 x 的值。

下列程序就是用来对形如 ax+b=c 的一元一次方程进行求解，其中，b、c 可以是任意整数，而 a 则为不等于零的整数。而且规定，从键盘输入的一元一次方程的形式都为 ax+b=c 这样的格式（也可为 ax-b=c 的形式），方程输入时以字符形式保存到字符数组 exp[ ] 中（输入时，无论 a，b 为何值，它们的值都必须原样写在对应的位置上。如，a=1，b=0 时，则方程应以“1x＋0=c”的形式输入，而不能以“x=c”的形式输入），并且以“.”作为输入结束标志。请完善程序。

```
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cstdlib>
using namespace std;
int main()
```

```
{
    string str_a, str_b, str_c;
    int a, b, c, i, j, k;
    double x;
    char op, ch;
    char exp[21] = { 0 };
    cout << "input expression:" << endl;
    i = 0;
    ch = getchar();
    while (    ①    )
    {
        i = i + 1;
        exp[i] = ch;
        ch = getchar();
    }
    j = 1;
    while (exp[j] != 'x')
    {
        str_a =    ②    ;
        j = j + 1;
    }
    j = j + 1;
    op = exp[j];
    j = j + 1;
    while (exp[j] != '=')
    {
        str_b = str_b + exp[j];
        j = j + 1;
    }
    j = j + 1;
    for (k = j;    ③    ; k++)
        str_c = str_c + exp[k];
    a = atoi(str_a.c_str());
    b = atoi(str_b.c_str());
    c = atoi(str_c.c_str());
    if (op == '-')
        b = -b;
    x =    ④    ;
    cout << x << endl;
    return 0;
}
```

习题

1 阅读程序

```
const int n = 1000000;
const int MAX = 16;
int a[1000000 + 1] = { 0 };
int main()
{
    long i, j;
    long x;
    for (i = 1; i <= MAX; i++)
    {
        cin >> x;
        a[x] = a[x] + 1;
    }
    for (j = n; j >= 1; j--)
    {
        if (a[j] != 0)
        {
            for (i = 1; i <= a[j]; i++)
                cout << j << " ";
        }
    }
    return 0;
}
```

输入：1 2 3 4 100 200 300 400 1 2 3 4 1000 2000 3000 4000

输出：_____

2.[2013]（序列重排） 全局数组变量 a 定义如下：

const int SIZE = 100;　　int a[SIZE], n;

它记录着一个长度为 n 的序列 a[1], a[2], …, a[n]。现在需要一个函数，以整数 p (1 ≤p ≤n) 为参数，实现如下功能：将序列 a 的前 p 个数与后 n –p 个数对调，且不改变这 p 个数（或 n –p 个数）之间的相对位置。例如，长度为 5 的序列 1, 2, 3, 4, 5，当 p = 2 时重排结果为3, 4, 5, 1, 2 。

有一种朴素的算法可以实现这一需求，其时间复杂度为 O(n)、空间复杂度为 O(n)：

```
void swap1(int p)
{
    int i, j, b[SIZE];
    for (i = 1; i <= p; i++)
        b[__(1)__] = a[i];
    for (i = p + 1; i <= n; i++)
        b[i - p] = __(2)__;
    for (i = 1; i <= __(3)__; i++)
        a[i] = b[i];
```

```
}
```

我们也可以用时间换空间，使用时间复杂度为 O(n2)、空间复杂度为 O(1) 的算法：

```
void swap2(int p)
{
    int i, j, temp;
    for (i = p + 1; i <= n; i++)
    {
        temp = a[i];
        for (j = i; j >=__(4)_____; j--)
            a[j] = a[j - 1];
        (5) _____= temp;
    }
}
```

3. [2015] (中位数) 给定n (n 为奇数且小于1000)个整数，整数的范围在0～m (0 < m < 2^{31}) 之间，请使用二分法求这n 个整数的中位数。所谓中位数，是指将这n 个数排序之后， 排在正中间的数。

```
const int MAXN = 1000;
int n, i, lbound, rbound, mid, m, count;
int x[MAXN];
int main(){
    cin >> n >> m;
    for (i = 0; i < n; ++i)
        cin    >> x[i];
        lbound=0;
        rbound=m;

    while (___(1)_____) {
        mid  = (lbound + rbound )/2;
        __(2)_____;
        for (i = 0; i < n; i++)
            if (_____(3)__ )
            __(4)_____;
        if (count > n / 2) lbound = mid + 1;
        else
        __(5)_____ ;
    }
    cout << rbound << endl;
    return 0;
}
```

第 2 讲 链表

当需要处理一批相关数据时，可以考虑使用数组进行存储。C++语言规定定义数组时必须确定数组元素个数，以便为其分配一块连续的存储空间，依次存放各个元素。但在实际应用中常常很难确定元素个数，例如，编写程序处理各个班级的学生成绩时，由于班级人数不同，为了能用同一程序处理不同班级的学生成绩，只能将数组定义的足够大，这样显然会造成存储空间的浪费。此外，当在数组中插入或删除元素时，都要引起大量数据的移动，而且数据量的扩充将受到数组所占用存储空间的限制。

链表是一种用于处理一批相关数据的数据结构，这些数据存放在链表的各个结点中，这些结点通过指针链接在一起。在逻辑上这些结点是连续的，但在实际存储时并不要求占据连续的存储空间，因此链表不必事先定义其结点的最大数目，在程序执行期间，可以根据需要增加或删除链表中的结点，从而有效地避免了存储空间的浪费和数据移动问题。一种处理学生成绩的单向链表如图 1 所示。

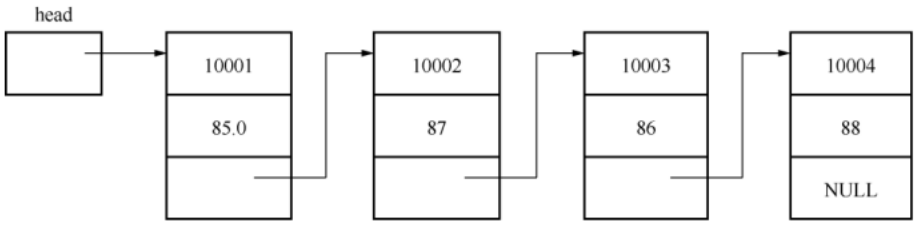


图 1 简单单向链表

那么在处理一批相关数据时，是否应该抛弃数组一律使用链表呢？答案是否定的。数组的特点是在逻辑上相邻的两个元素，在物理位置上也相邻。因此，根据数组名（首地址）和元素下标可以十分方便地确定元素的存储位置，从而实现对数组中任一元素的随机存取，这是链表无法做到的。一般而言，如果所需处理的数据数量事先确定且不发生变化，可以考虑使用数组。而链表则适合处理数据量需要动态改变的情况。

数组与链表的综合比较如表 1 所示。

表 1　数组与链表的比较

参　数	数　组	链　表
存放的数据量	固定	可变
存储空间	连续	不连续
存取数据	方便	不方便
适用场合	可事先确定所需处理的数据量	程序执行时需要动态改变数据量

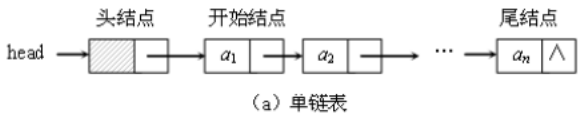
2.1 链表解析

链表是由头指针和一系列结点通过指针链接而成，单向链表的基本结构如图 1 所示。其中，头指针（head）用来存放链表中第一个结点（头结点）的地址，由头指针所指向的头结点出发，就可以依次访问链表中任何一个结点的数据成员。

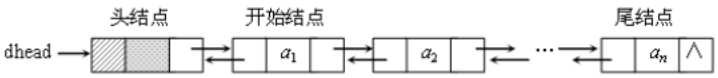
除单向链表外，还有双向链表、循环链表等，本章只讨论单向链表，有关链表的详细讨论，读者可参阅“数据结构”方面的书籍。

单向链表中每个结点一般由两大部分组成。

1）数据域，用于存放用户需要使用



（a）单链表



（b）双链表

的实际数据，可以是一个数据项，也可以是多个数据项。如图 8-4 所示，描述学生情况的数据项有两个，分别是学号和成绩，它们构成了该链表结点的数据域。

2）指针域，用于存放和该结点相链接的下一个结点的地址。如图 1 所示，第 1 个结点的指针域存放的是第 2 个结点的起始地址，第 2 个结点的指针域存放的是第 3 个结点的起始地址，以此类推。链表中最后一个结点（尾结点）因其后续无结点，其指针域不再指向其他结点，而是存放一个“NULL”（空地址），表示链表到此结束。

由于链表结点由数据域和指针域两部分组成，即一个结点中包含了多个不同类型的数据，因此，链表结点一般用结构体来描述，而头指针和结点的指针域则是同类型的结构体指针变量。图 2 所示的链表结点就可通过结构体类型定义如下。

```
struct student
{    int num;                    //数据域
     double score;               //数据域
     struct student *next;       //指针域
};
```

以上定义了一个结构体 student 类型，student 类型数据包括 3 个数据成员：int 类型的 num、float 类型的 score 和指向另一个 student 类型数据的指针变量 next。

1　链表的建立

链表由一系列“结点”组成，结点可以在需要时动态生成。每个结点的数据域至少包括两部分：一是存储数据元素的“数据域”；二是存储下一个结点内存地址的“指针域”。图 8.5-1 就是一个链表的示意图，其中，head 表示头结点，数据域的值为 5 的结点为尾结点，其指针域为 NULL。

链表示意图

链表中每个结点的数据结构定义如下：

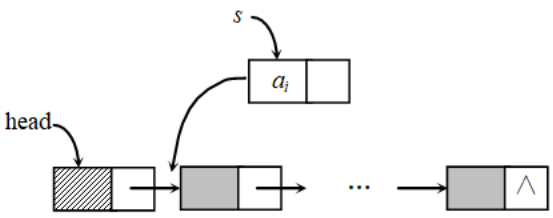
```
struct node{
     int num;
     node *next;
};
```

建立链表有“头插法”和“尾插法”两种方法，前者是把新结点插在头结点之后，后者是把新结点插在尾结点之后。本课以尾插法为例建立链表，就是不断地在链表尾部链接一个新结点。

1. 头插法

头插法从一个空表开始，读取数组 a 中的元素，生成新结点 s，将读取的数据存放到新结点的数据域中，然后将新结点 s 插入到当前链表的表头上，如图 2.13 所示。重复这一过程直到 a 数组的所有元素读完为止

```
struct LinkList      //单链表结点类型
{       int  data;       //存放数据元素
     LinkList *next;     //指向下一个结点的域
};
```

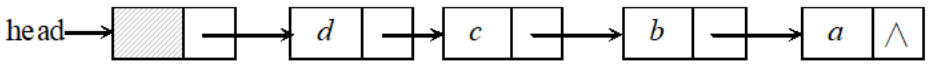


头插法建表

```
void CreateListF(int a[],int n)//头插法建立单链表
{    LinkList *s; int i;
     head->next=NULL;              //将头结点的 next 域置为 NULL
     for (i=0; i<n; i++)           //循环建立数据结点
```

```
    {    s=new LinkList();          // malloc(sizeof(struct Linklist))
         s->data=a[i];              //创建数据结点*s
         s->next=head->next; //将*s 结点插入到开始结点之前,头结点之后
         head->next=s;
    }
}
```

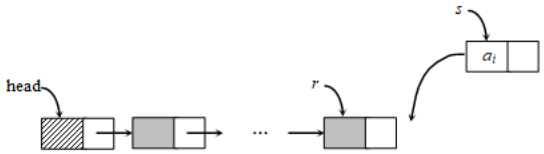
若数组 a 含 4 个元素'a'、'b'、'c'和'd'，调用 CreateListF(a,4)建立的单链表如图所示。从中看到，头插法建立的单链表中数据结点的次序与 a 数组中的次序正好相反。



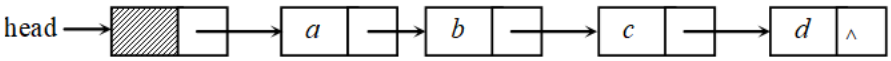
2 尾插法建表

头插法建立链表虽然算法简单，但生成的链表中结点的次序和原数组元素的顺序相反。若希望两者次序一致，可采用尾插法建立。该方法是将新结点 s 插到当前链表的表尾上，为此必须增加一个尾指针 r，使其始终指向当前链表的尾结点，如图所示。

```
void CreateListR(T a[],int n) //尾插法建立单链表
{    LinkList *s,*r; int i;
     r=head;          //r 始终指向尾结点,开始时指向
头结点
     for (i=0; i<n; i++)          //循环建立数据结点
     {    s=new LinkList ();
          s->data=a[i];           //1  创建数据结点*s
          r->next=s;              //2 将*s 结点插入*r 结点之后
          r=s;                    //3
     }
     r->next=NULL;                //将尾结点的 next 域置为 NULL
}
```

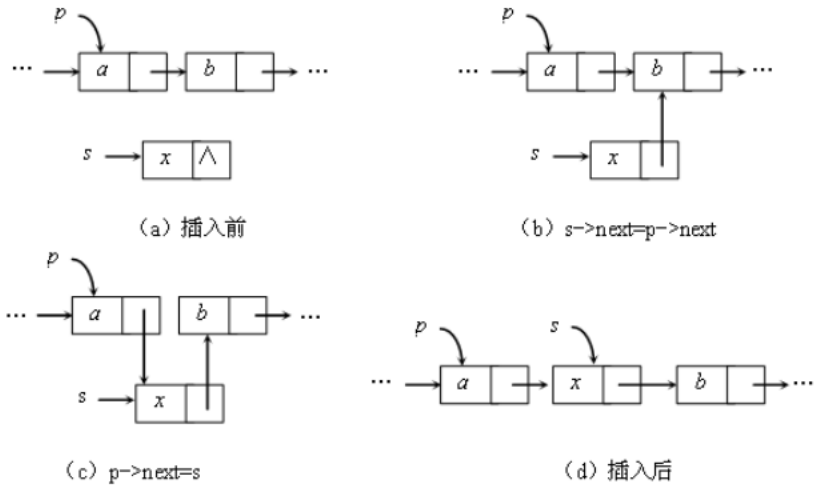


若数组 a 包含 4 个元素'a'、'b'、'c'和'd'，调用 CreateListR(a,4)建立的单链表如图所示。从中看到，尾插法建立的单链表中数据结点的次序与 a 数组中的次序正好相同。



2 结点的插入和删除

在单链表中，插入和删除结点是最常用的操作，它是建立单链表和相关基本运算算法的基础。

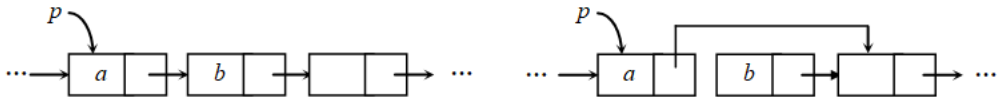


（a）插入前　　（b）s->next=p->next

（c）p->next=s　　（d）插入后

上述指针修改用 C 语句描述如下：

```
s->next=p->next; p->next=s;
```



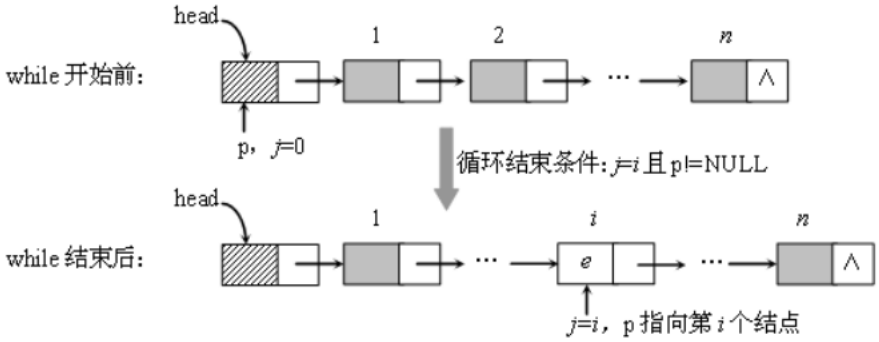
（a）删除前　　（b）删除后

上述指针修改用 C 语句描述如下：

```
p->next=p->next->next;
```

3 结点的查找

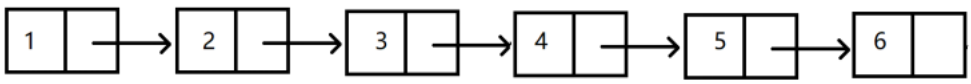
该运算在单链表 head 中从头开始找到第 i 个结点，若存在这样的数据结点，则将其 data 域值赋给变量 e，并返回 true，若没有第 i 个数据结点，返回 false。




```
bool GetElem(int i, T &e)   //求单链表中某个数据元素值
{   int j=0; LinkList *p;
    p=head;                 //p 指向头结点,j 置为 0(即头结点的序号为 0)
    while (j<i && p!=NULL)      //找第 i 个结点*p
    {   j++;
        p=p->next;
```

```
    }
    if (p==NULL)        //不存在第 i 个数据结点,返回 false
        return false;
    else                //存在第 i 个数据结点,返回 true
    {   e=p->data;
        return true;
    }
}
```

2.2 链表的数组表示

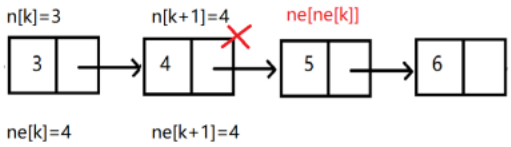


e[1]=1　　e[2]=2　　表示当前节点的值
ne[1]=2　　ne[2]=3　　表示当前节点next域指向的下一个节点编号

head 表示头节点的下标
e[i] 表示节点 i 的值
ne[i] 表示节点 i 的 next 指针是多少
idx 存储当前已经用到了哪个节点

```
//* 将 x 插到头节点
void add_to_head(int x)
{
  e[idx]  = x;     //将第一个的值 变成我们插入的值
  ne[idx] = head; //当前点指向 head 后的点
  head    = idx;  //将 head 指到 当前的头部
  idx++;          //移到下一个
}
```

```
//* 将位置 k 后面的点删除
void remove(int k)
{
  ne[k] = ne[ne[k]];
}
```



2.3 综合练习

1 [3204]约瑟夫问题

链表求解约瑟夫（Joseph）问题。有 n 个小孩围成一圈，给他们从 1 开始依次编号，从编号为 1 的小孩开始报数，数到第 m 个小孩出列，然后从出列的下一个小孩重新开始报数，数到第 m 个小孩又出列，…，如此反复直到所有的小孩全部出列为止，求整个出列序列。
如当 n=6，m=5 时的出列序列是 5，4，6，2，3，1。n,m 不等于 20
输入 n m 的值，输出 出列序列
样例输入 6 5
样例输出 5 4 6 2 3 1

本题我们可以用数组建立标志位等方法求解，但如果用上数据结构中循环链的思想，则更贴切题意，解题效率更高。n 人围成一圈，把一人看成一个结点， n 人之间的关系采用链接方式，即每一结点有一个前继结点和一个后继结点，每一个结点有一个指针指向下一个结点，最后一个结点指针指向第一个结点。这就是单循环链的数据结构。当m人出列时，将m结点的前继结点指针指向m结点的后继结点指针，即把m结点驱出循环链。

1、建立循环链表。

当用数组实现本题链式结构时，数组 a[i]作为"指针"变量来使用，a[i]存放下一个结点的位置。设立指针 j 指向当前结点，则移动结点过程为 j=a[j]，当数到 m 时，m 结点出链，则 a[j]=a[a[j]]。 当直接用链来实现时，则比较直观，每个结点有两个域：一个数值域，一个指针域，当数到 m 时，m 出链，将 m 结点的前继结点指针指向其后继结点；

2、设立指针，指向当前结点，设立计数器，计数数到多少人；

3、沿链移动指针，每移动一个结点，计数器值加 1，当计数器值为m时， 则m结点出链，计数器值置为 1 。

4、重复 3，直到 n 个结点均出链为止。

```
#include <iostream>
using namespace std;
struct Child                    //小孩结点类型
{
  int no;                       //小孩编号
  Child *next;                  //指向下一个结点指针
}*h;                            //首结点指针

  int n,m;
  void Init(int n1,int m1)          //建立有 n1 个结点的循环单链表
  {
    int i;
    Child *p,*r;                //r 指向新建循环单链表的尾结点
    n=n1; m=m1;                 //置数据成员值
    h=new Child();
    h->no=1;                    //先建立只有一个 no 为 1 结点的单链表
    r=h;
    for (i=2;i<=n;i++)
    {
      p=new Child();            //建立一个新结点*p
      p->no=i;                  //新结点存放编号 i
      r->next=p; r=p;           //将*p 结点链到末尾
    }
    r->next=h;                  //构成一个首结点为 h 的循环单链表
  }

  void Jsequence()              //求解约瑟夫序列
  {
    int i,j;
```

```
    Child *p,*q;
    for (i=1;i<=n;i++)          //共出列 n 个小孩
    {
      p=h; j=1;
      while (j<m-1)             //从*h 结点开始报数，报到第 m-1 个结点
      {
        j++;                    //报数递增
        p=p->next;              //移到下一个结点
      }
      q=p->next;                //q 指向第 m 个结点
      cout << q->no << " ";     //该结点出列
      p->next=q->next;          //删除*q 结点
      h=p->next;                //从下一个结点重新开始
    }
  }
int main()
{
  cin>>n>>m;
  Init(n,m);
  Jsequence();
  cout << endl;
  return 0;
}
```

2 [6878]记录信息

利用动态链表输出、存储若干学生信息（学号、姓名、性别、年龄、得分），再按输出顺序倒序输出。其中,学号长度不超过 20，姓名长度不超过 40，性别长度为 1，年龄和成绩为一个整数。输入包括若干行，每一行都是一个学生的信息，如：

00630018 zhouyan m 20 10.0

输入的最后以"end"结束输出将输入的内容倒序输出

每行一条记录，按照

学号 姓名 性别 年龄 得分的格式输出

样例输入 00630018 zhouyan m 20 10	样例输出
0063001 zhouyn f 21 100	10630018 zouan m 20 10
0063008 zhoyan f 20 1000	0630018 zuyan m 21 100
0063018 zhouan m 21 10000	01030018 houyan m 2010
end	00160018 zouyan f 21 100

思路：基础链表题目。

链表指向前一个数据，学习循环链表之前应该做的题目；

```
#include<iostream>
using namespace std;
struct node{
    node *nxt;
```

```
    node *pre;
    char st[100];
};

int main(){
    node *fst = new node;
    fst->pre = NULL;
    while(1){
        gets(fst->st);
        if(fst->st[0] == 'e'){
            while(fst->pre != NULL){
                fst = fst->pre;
                printf("%s\n", fst->st);
            }
            delete fst;
            return 0;
        }
        node *x = fst;
        fst->nxt = new node;
        fst = fst->nxt;
        fst->pre = x;
    }
    return 0;
}
```

3 [1892] 链表结点的物理顺序与逻辑顺序

已知一个单向链表各结点在存储器中的物理顺序，以及各结点之间的指向关系，要求输出该链表各结点的逻辑顺序。

例如有 6 个结点，按这 6 个结点在存储器中的物理顺序依次编号为 1～6。第 1 个结点的指针域为 p4，表示它指向第 4 个结点。符号“∧”表示空指针。
因此，这个链表 6 个结点的逻辑顺序为：6->1->4->2->5->3。

```
#include<iostream>
using namespace std;
struct node{//定义一个节点
        int id;
        node *next;
};

char ch[5];//输入的字符，如 p4
int n,digit[100];//digit[i]=j;表示 第 i 个节点指向第 j 个节点
int tail;//tail 表示最后一个节点
int main(){
```

```
    while(cin>>n){
        if(n==0)break;
        1_____________________
        for(int i=1;i<=n;i++){//循环 n 次，输入 n 个字符串
            cin>>ch;
            if(ch[1]=='u'){//u 即 null，查找最后一个节点
                tail=i;//记录最后一个节点的数据域
                2_____________
            }
            else
                digit[i]=ch[1]-'0';
        }//1 获取数据

        head=new node();//
        3_____________//初始头节点
        int j=1;
        while(j<n){//扫描、建立链表的过程
            for(int i=1;i<=n;i++){//反向扫描
                if( 4_____________){
            t=new node();
                    5_____________//建立 i->j
                    head=t,j++;
                }
            }
        }//2 建立链表

        while(6_____________){
            cout<<head->id<<"->";
            head=head->next;
        }//3 输出
        cout<<tail<<endl;
    }
    return 0;
}
```

4 [7287] 单链表

实现一个单链表，链表初始为空，支持三种操作：
(1) 向链表头插入一个数；
(2) 删除第 k 个插入的数后面的数；
(3) 在第 k 个插入的数后插入一个数
现在要对该链表进行 M 次操作，进行完所有操作后，从头到尾输出整个链表。
注意:题目中第 k 个插入的数并不是指当前链表的第 k 个数。例如操作过程中一共插入了 n 个数，则按照插入的时间顺序，这 n 个数依次为：第 1 个插入的数，第 2 个插入的数，…第 n 个插入的数。

输入格式
第一行包含整数 M，表示操作次数。
接下来 M 行，每行包含一个操作命令，操作命令可能为以下几种：
(1) “H x”，表示向链表头插入一个数 x。
(2) “D k”，表示删除第 k 个输入的数后面的数（当 k 为 0 时，表示删除头结点）。
(3) “I k x”，表示在第 k 个输入的数后面插入一个数 x（此操作中 k 均大于 0）。
输出格式
共一行，将整个链表从头到尾输出。
数据范围
1≤M≤100000
所有操作保证合法

```
#include <iostream>
using namespace std;
const int N = 1e5 + 10;
//* head  表示头节点的下标                    //* e[i]  表示节点 i 的值
//* ne[i]  表示节点 i 的 next 指针是多少    //* idx 存储当前已经用到了哪个节点
int head, e[N], ne[N], idx;//*  初始化
void init()
{
       head = -1;
        idx   = 0;
}
void add_to_head(int x)//*  将 x 插到头节点
{
 e[idx]   = x;    //! 将第一个的值 变成我们插入的值
 ne[idx] = head; //! 因为是插入，所以当前的 head 是插入后的 next 指针所要指的地方
 head    = idx;  //! 将 head 指到 当前的头部
 idx++;         //! 移到下一个
}
void add(int k, int x)//*将 x 这个点 插入到下标是 k 的点后面
{
       e[idx]   = x;     //! 先把 x 这个值存下来
        ne[idx] = ne[k]; //! 把新点的指针插入 k 这个点指向的下一个位置
        ne[k]    = idx;   //! 把 k 指向 要插入节点的位置
        idx++;
}
void remove(int k)//*  将位置 k 后面的点删除
{
       ne[k] = ne[ne[k]];
}
int main( )
{
 // todo  第 k 个插入的数 就是下标是 k-1 的点
```

```
 int m;
 cin >> m;
 init(); //! 初始化
 while(m--)
 {
  int  x, k;
  char op;
  cin >> op;
  if(op == 'H')
  {
   cin >> x;
   add_to_head(x);
  }
  else if(op == 'D')
  {
   cin >> k;
   if(k == 0)
    head = ne[head];//删除第一个点节点，head 连接到下一个点
   else
    remove(k - 1); // todo  下标是  k-1
  }
  else
  {
   cin >> k >> x;
   add(k - 1, x); // todo  下标是  k-1
  }
 }
  for(int i = head; i != -1; i = ne[i]) cout << e[i] << ' ';
 cout << endl;
 return 0;
}
```

课前练习

1．[2004]Joseph
原始的 Joseph 问题的描述如下：有 n 个人围坐在一个圆桌周围，把这 n 个人依次编号为 1，…，n。从编号是 1 的人开始报数，数到第 m 个人出列，然后从出列的下一个人重新开始报数，数到第 m 个人又出列，…，如此反复直到所有的人全部出列为止。比如当 n=6，m=5 的时候，出列的顺序依次是 5，4，6，2，3，1。

现在的问题是：假设有 k 个好人和 k 个坏人。好人的编号的 1 到 k，坏人的编号是 k+1 到 2k。我们希望求出 m 的最小值，使得最先出列的 k 个人都是坏人。
输入:
仅有的一个数字是 k（0 < k <14）。
输出:

使得最先出列的 k 个人都是坏人的 m 的最小值。
输入样例:
4
输出样例:
30
程序：

```
#include <stdio.h>
long k, m, begin;
int check(long remain){
        long result = ( ①                ) % remain;
        if ( ②              ){
                begin = result; return 1;
        }
        else return 0;
}
int main(){
        long i, find = 0;
        scanf("%ld", &k);
        m = k;
        while( ③                ) {
                find = 1; begin = 0;
                for (i = 0; i < k; i++)
                        if (!check( ④                ))){
                                find = 0; break;
                        }
                m++;
        }
        printf("%ld\n", ⑤                );
        return 0;
 }
```

2.[2017]（快速幂）请完善下面的程序，该程序使用分治法求 xp mod m 的值。
（输入：三个不超过 10000 的正整数 x，p，m。
输出：xp mod m 的值。
提示：若 p 为偶数，xp=(x2)p/2；若 p 为奇数，xp=x*(x2)(p-1)/2。

```
#include <iostream>
using namespace std;
int x, p, m, i, result;
int main() {
cin >> x >> p >> m;
result =          (1)          ;
while (          (2)          ) {
   if (p % 2 == 1)
   result =          (3)          ;
```

```
    p /= 2;
    x = ____(4)____ ;
}
cout << ____(5)____ << endl;
return 0;
}
```

习题

1.向一个栈顶指针为 hs 的链式栈中插入一个指针 s 指向的结点时，应执行（　　）。

A.hs->next = s;

B.s->next = hs; hs = s;

C.s->next = hs->next; hs->next = s;

D.s->next = hs; hs = hs->next;

2．双向链表中有两个指针域 llink 和 rlink，分别指向该结点的前驱及后继。设 p 指向链表中的一个结点，它的左右结点均非空。现要求删除结点 p，则下面语句序列中错误的是（　　）。

A. p->rlink->llink = p->rlink;
   p->llink->rlink = p->llink; delete p;

B. p->llink->rlink = p->rlink;
   p->rlink->llink = p->llink; delete p;

C. p->rlink->llink = p->llink;
   p->rlink->llink->rlink = p->rlink; delete p;

D. p->llink->rlink = p->rlink;
   p->llink->rlink->llink = p->llink; delete p;

3.[3204] 链表求约瑟夫环

```
#include <iostream>
using namespace std;
struct Child                    //小孩结点类型
{
   int no;
   Child *next;
} __1_____    *h;
 int n,m;
 void Init(int n1,int m1)          //建立有 n1 个结点的循环单链表
   {
      int i;
      Child *p,*r;
      n=n1; m=m1;               //置数据成员值
      h=new Child();
      h->no=1;      //先建立只有一个 no 为 1 结点的单链表
      r=h;
      for (i=2;i<=n;i++)
      {
```

```
        p=new Child();        //建立一个新结点*p
        p->no=i;
        __2_____________  r->next=p;
        r=p;_    //将*p 结点链到末尾
      }
      __3_____________________  r->next=h;  //构成一个首结点为 h 的循环单链表
   }

   void Jsequence()            //求解约瑟夫序列
   {
      int i,j;
      Child *p,*q;
      for (i=1;i<=n;i++)        //共出列 n 个小孩
      {
        p=h; j=1;
        while ( j< 4_____________  m-1)
        {
          j++;
          __5_____________________ p=p->next;        //移到下一个结点
        }
        q=p->next;            //q 指向第 m 个结点
        cout << q->no << " ";   //该结点出列
        __6_____________________        p->next=q->next;     //删除*q 结点
        delete q;           //释放其空间
        __7_____________  h=p->next;   //从下一个结点重新开始
      }
   }
int main()
{
   cin>>n>>m;
   Init(n,m);
   Jsequence();
   cout << endl;
   return 0;
}
```

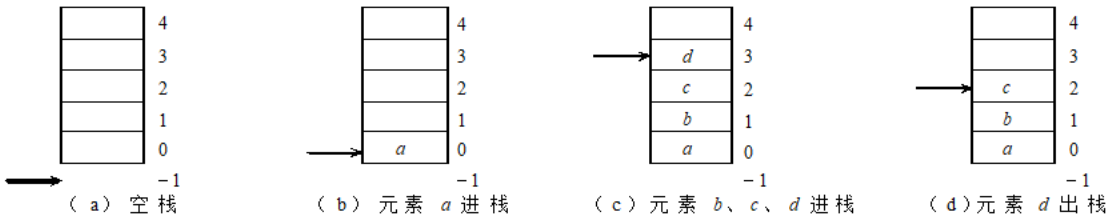
第 3 讲 栈

栈是一种只能在一端进行插入或删除操作的线性表。表中允许进行插入、删除操作的一端称为栈顶。栈顶的当前位置是动态的，由一个称为栈顶指针的位置指示器来指示。表的另一端称为栈底。当栈中没有数据元素时，称为空栈。

栈的插入操作通常称为进栈或入栈，栈的删除操作通常称为退栈或出栈。

栈的主要特点是“后进先出”，即后进栈的元素先弹出。每次进栈的数据元素都放在原当前栈顶元素之前成为新的栈顶元素，每次出栈的数据元素都是原当前栈顶元素。栈也称为后进先出表。

下图是一个栈的动态示意图，图中箭头表示当前栈顶元素位置。图（a）表示一个空栈；图（b）表示数据元素 a 进栈以后的状态；图（c）表示数据元素 b、c、d 进栈以后的状态；图（d）表示出栈一个数据元素以后的状态。



3.1 栈的操作

栈可以采用顺序存储结构，分配一块连续的存储空间 data（大小为常量 MaxSize）来存放栈中元素，并用一个变量 top（栈顶指针）指向当前的栈顶以反映栈中元素的变化，采用顺序存储的栈称为顺序栈。

栈的存储方式和实现和实现也有数组模拟和链表模拟两种，本讲义只介绍数组实现。

1 栈的定义

```
#define MaxSize 100005
    struct Stack{
        int Data[MaxSize];
        int Top;
};
```

2 栈的初始化

```
    void init(Stack st){
        st.Top=-1;
    //也可以写成 st->top=-1;
    }
```

3 判空

```
    bool empty(Stack st){
        if(st.Top==-1)return 1;
        else    return 0;
}
```

4 进栈（压栈）

①若 TOP≥ n 时，则给出溢出信息，作出错处理（进栈前首先检查栈是否已满，满则溢出；不满则作②）；

②TOP++（栈指针加 1 ，指向进栈地址）；

③S[TOP]=X，结束（X 为新进栈的元素）；

```
void push(int a,Stack st){
    st.Top++;
    st.Data[Top]=a;
}
```

5 出栈

①若 TOP≤0，则给出下溢信息，作出错处理(退栈前先检查是否已为空栈， 空则下溢；不空则作②)；

②X=S[TOP]，（退栈后的元素赋给 X）；
③TOP--，结束（栈指针减 1，指向栈顶）。

```
void pop(Stack st){
        st.Top--;
    }
```

6 取栈顶元素

```
int top(Stack st){
    return st.Data[st.Top];
}
```

练习：

初始化：_____

入栈：_____

出栈：_____

3.2 STL 中的栈

STL 是“Standard Template Library”的缩写，中文译为“标准模板库”。STL 是 C++ 标准库的一部分，不用单独安装。C++ 对模板（Template）支持得很好，STL 就是借助模板把常用的数据结构及其算法都实现了一遍，并且做到了数据结构和算法的分离。例如，vector 的底层为顺序表（数组），list 的底层为双向链表，deque 的底层为循环队列，set 的底层为红黑树，hash_set 的底层为哈希表。

包含头文件：**#include <stack>**

使用命名空间：using namespace std;

定义栈的方法：

```
stack<char> S1;     //栈中的结点为字符
stack<int> S2;      //栈中的结点为整型数据
stack<pos> S3; //栈中的结点为自定义结构体 pos 变量
```

stack 的成员函数：

push：压栈，参数为需要压入栈的结点；

pop：出栈，返回值为出栈的结点；

top：取得栈顶结点，返回值为栈顶结点，该操作并不会弹出栈顶结点；

empty：判断栈是否为空，返回值为 bool 型。

size：计算栈中结点的个数。

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    stack<int> s1;//1 默认创建
    stack<int> s2(s1);//拷贝创建
```

```
    for(int i = 0; i < 10; i++)
        s1.push(i);// 2 入栈：  void push(DT data);
    cout << s1.top() << endl;//3 访问但不弹出栈顶元素：DT top();
    s1.pop();//4 弹出但不访问栈顶元素：void pop();
    cout << s1.top() << endl;
    cout << s1.empty() << " " << s2.empty() << endl;
        // 5 判断栈是否为空：bool empty();
    cout << s1.size() << endl;// 6、获得栈中元素的个数：int size();
    return 0;
}
```

输出结果

```
9
8
0 1
9
```

练习：

1. 使用 STL 栈的头文件为 _____
2. 用 STL 定义个整数栈和一个字符栈 _____
3. 把整数 a 入栈_____
4. 判断一个栈是否为空_____

3.3 案例分析

1[1521]进制转换

把十进制到二进制的转换。

输入

234

输出

11101010

方法 1:

```
#include<iostream>
#include<stack>
using namespace std;
int main(){
        int n;
        stack<int> st;
        cin>>n;
        while(n)
        {
            st.push(n%2);
```

方法 2：

```
:#include<iostream>
using namespace std;
int main()
{
    int n,i=0;
    cin>>n;
    int a[100000];
    memset(a,0,sizeof(a));
    while(n!=0)
    {
```

```
            n/=2;
        }
        while(!st.empty())
        {
            cout<<st.top();
            st.pop();
        }
        cout<<endl;
    return 0;
}
```

```
        i++;
        a[i]=n%2;
        n=n/2;
    }
    int re=i;
    for(int i=re;i>=1;i--)
        cout<<a[i];
    cout<<endl;
    return 0;
}
```

2 [7289]模拟栈

实现一个栈，栈初始为空，支持四种操作：
(1) “push x” - 向栈顶插入一个数 x；
(2) “pop” - 从栈顶弹出一个数；
(3) “empty” - 判断栈是否为空；
(4) “query” - 查询栈顶元素。
现在要对栈进行 M 个操作，其中的每个操作 3 和操作 4 都要输出相应的结果。
输入格式：第一行包含整数 M，表示操作次数。接下来 M 行，每行包含一个操作命令，操作命令为”push x”， ”pop”，”empty”，”query”中的一种。
输出格式
对于每个”empty”和”query”操作都要输出一个查询结果，每个结果占一行。
其中，”empty”操作的查询结果为“YES”或“NO”，”query”操作的查询结果为一个整数，表示栈顶元素的值。
数据范围
1≤M≤100000 1≤x≤109 所有操作保证合法。

```
#include<iostream>
#include<stack>
using namespace std;
const int N=100010;
int n;

int main(){
  stack <int> stk;
  cin>>n;
  while(n--){
    string op;
    int x;
    cin>>op;
    if(op=="push")cin>>x,stk.push(x);
    else if(op=="pop")stk.pop();
```

```
#include<iostream>
#define R register int
using namespace std;
const int N = 1e5+5;
int x,m;
string str;
struct stack{
  int sta[N] , topp;
}s;
void init(stack &st){ st.topp=0; }//结构体内的初始化
void push(stack &st,int x){ st.sta[++st.topp] = x;}
void pop(stack &st){ st.topp--;}
int top(stack &st){ return st.sta[st.topp];}
int size(stack &st){ return st.topp;}
bool empty(stack &st){ return st.topp==0?1 : 0;}
```

```
    else if(op=="empty")
        cout<<(!stk.empty()?"NO":"YES")<<endl;
    else
     cout<<stk.top()<<endl;
  }
  return 0;
}
```

```
int main(){
  scanf("%d",&m);
  while(m--) {
    cin>>str;
    if(str=="push") { cin>>x; push(s,x);}
    else if(str=="pop") pop(s);
        else if(str=="empty") empty(s)?cout<<"YES\n" :
cout<<"NO\n";
    else cout<<top(s)<<endl;
  }
  return 0;
}
```

3 [1509]超市购物车

物美超市存放购物车的轨道像一个栈，工作人员从一端推入购物车，顾客从同一端推出购物车。约定工作人员每次只推入一辆购物车，顾客也是每次只推出一辆购物车。
在购物的高峰期，经常会出现没有购物车可用的情形，物美公司很想知道一天下来究竟有多少顾客拿不到购物车。
输入
输入文件中包含多个测试数据。每个测试数据占一行，为一个字符串(最长为 100 个字符)；字符串中的字符为 p 或 q，**p 表示工作人员推入一辆购物车，q 表示有一个顾客推出一辆购物车**。约定，如果没有购物车，则顾客会放弃而不会等待。测试数据一直到文件尾。
输出
对输入文件中的每个测试数据，输出有**多少个顾客拿不到购物车**。
样例输入
pppqqqpqqqpqqpqpqpqpqpq
qpqpqpqqqpqpppqqqpppqpppq
样例输出
3
3

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    char a[10000];
    while (cin >> a)
    {
        int num = 0;
          1
        for (int i = 0; i <strlen(a); i++)
```

```
        {
            if (a[i] == 'p')
                2
            else if (a[i] == 'q'&&s.empty())
                num++;//没有购物车了，该客户就拿不到车
            else
                3
        }
        cout << num << endl;
    }
}
```

4 [2897] 括弧匹配检验

题目描述
假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，如（[ ] ()）或[（[ ] [ ])]等为正确的匹配，［（]）或（[ ] （ ）或 (( )) ）均为错误的匹配。
现在的问题是，要求检验一个给定表达式中的括弧是否正确匹配？
输入一个只包含圆括号和方括号的字符串，判断字符串中的括号是否匹配，匹配就输出“OK”，不匹配就输出“Wrong”。输入一个字符串：［（[] [] ）]，输出：OK。

输入输入仅一行字符（字符个数小于 255）。
输出匹配就输出 “OK”，不匹配就输出“Wrong”。
样例输入[(])
样例输出 Wrong

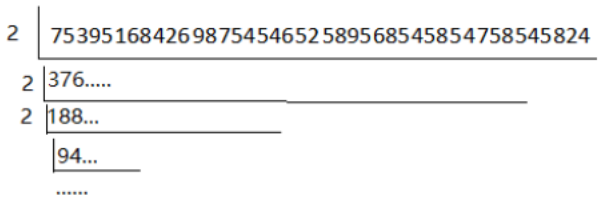
```
#include<bits/stdc++.h>
using namespace std;

int main(){
    char a[1000];
    gets(a);
    int len=strlen(a);
    for(int i=0;i<len;i++){
        if(a[i]=='('||a[i]=='[')  1
        else {
            if(  2  ){
                cout<<"Wrong";
                return 0;
            }
            else
                3
        }
    }
```

```
    if(___4_____)
    return 0;
}
```

5 [1110]使用栈实现进制转换

使用栈将一个很长（位数>30&&位数<100）的十进制数转换为二进制数
输入 若干个很长的十进制数 每行一个
输出 转换为二进制，每行输出一个
样例输入
753951684269875454652589568545854758545824
样例输出
1000101001111010101000110100100010010010000010100101101000101010100100010011110101100111000100011111001000100010110111111011011010011010000
分析：

大数做除法时，每一次上商的值都在 0 ～ 9 ， 每次求得的余数连接以后的若干位得到新的被除数，继续做除法。




```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    char a[10000];
    while(cin>>a){
        int b[10000]={0};
        int len=strlen(a);
        for(int i=0;i<len;i++){
            b[i]=a[i]-'0';
        }
        int i=0;
        stack<int>s;
        while(i<len-1||b[i]>0){
            s.push(b[len-1]%2);//模拟除法，把余数入栈(余数由最低位产生)
            int k=0;
            for(int j=i;j<len;j++){ //按位相除
                k=k*10+b[j];//
                b[j]=k/2;//把余数和被除数的次高位组合
                k=k%2;
            }
            if(b[i]==0)i++;
        }
        while(!st.empty()){
            printf("%d",s.top());
            s.pop();
        }
```

```
            printf("\n");
        }
        return 0;
}
```

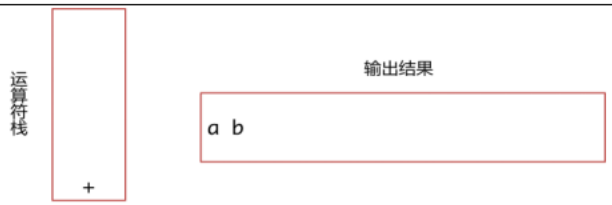
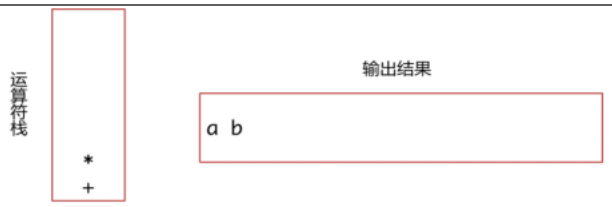
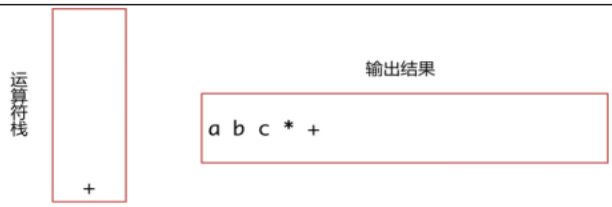
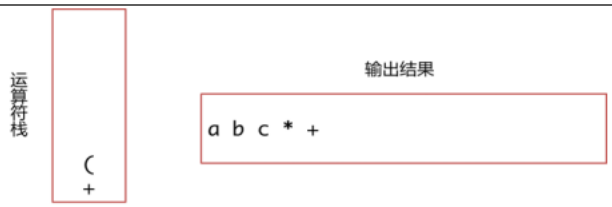
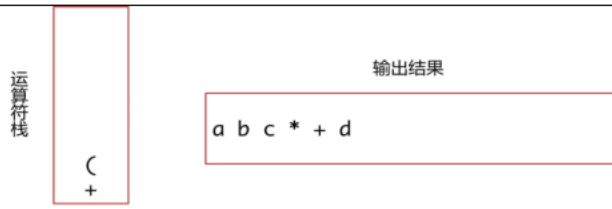
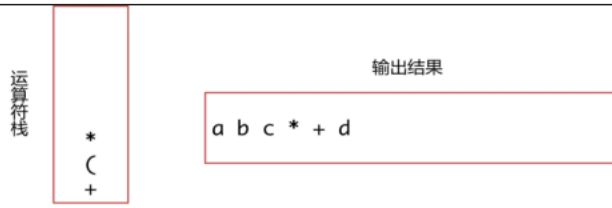
6 [2098]表达式求解

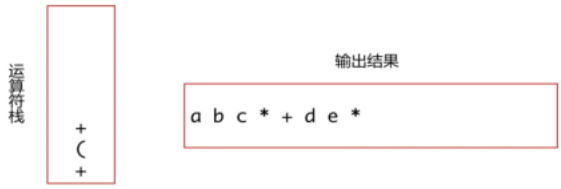
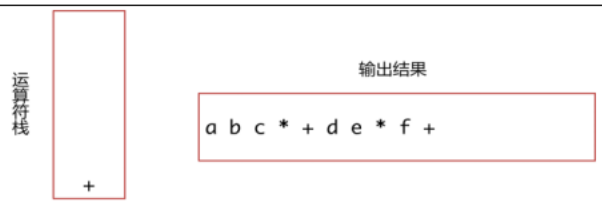
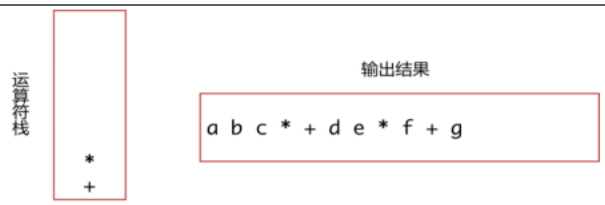
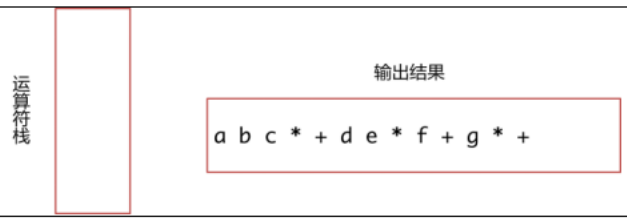
题目描述
读入一个只包含 +, -, *, / 的非负整数计算表达式，计算该表达式的值。
输入
测试输入包含若干测试用例，每个测试用例占一行，每行不超过 200 个字符，整数和运算符之间用一个空格分隔。没有非法表达式。当一行中只有 0 时输入结束，相应的结果不要输出。
输出
对每个测试用例输出 1 行，即该表达式的值，精确到小数点后 2 位。
样例输入
1 + 2
4 + 2 * 5 - 7 / 11
0
样例输出
3.00
13.36

平常我们简单的算式称为中缀表达式，通常要转换为后缀表达式进行求解。
中缀表达式转后缀表达式遵循以下原则：
1.遇到操作数，直接输出；
2.栈为空时，遇到运算符，入栈；
3.遇到左括号，将其入栈；
4.遇到右括号，执行出栈操作，并将出栈的元素输出，直到弹出栈的是左括号，左括号不输出；
5.遇到其他运算符'+''-''*''/'时，弹出所有优先级大于或等于该运算符的栈顶元素，然后将该运算符入栈；
6. 最终将栈中的元素依次出栈，输出。
经过上面的步骤，得到的输出既是转换得到的后缀表达式。

中缀表达式转后缀表达式举例：a+b*c+(d*e+f)*g ---------> abc*+de*f+g*+

1. 遇到 a，直接输出 a+b*c+(d*e+f)*g	
2.遇到+，此时栈为空，入栈 a+b*c+(d*e+f)*g	

3. 遇到 b，直接输出 a+b*c+(d*e+f)*g	
4. 遇到*,优先级大于栈顶符号优先级，入栈 a+b*c+(d*e+f)*g	
5. 遇到 c，输出 a+b*c+(d*e+f)*g	
6. 遇到+，目前站内的*与+优先级都大于或等于它，因此将栈内的*，+依次弹出并且输出，并且将遇到的这个+入栈： a+b*c+(d*e+f)*g	
7. 遇到(,将其入栈： a+b*c+(d*e+f)*g	
8. 遇到 d，直接输出： a+b*c+(d*e+f)*g	
9. 遇到*，由于*的优先级高于处在栈中的(，因此*入栈： a+b*c+(d*e+f)*g	
10. 遇到 e，直接输出： a+b*c+(d*e+f)*g	

11. 遇到+，栈顶的*优先级高于+，但是栈内的(低于+，将*出栈输出，+入栈： a+b*c+(d*e+f)*g	
12. 遇到 f，直接输出： a+b*c+(d*e+f)*g	
13. 遇到),弹出栈顶元素并且输出，直到弹出(才结束，在这里也就是弹出+输出，弹出(不输出： a+b*c+(d*e+f)*g	
14. 遇到*，优先级高于栈顶+，将*入栈： a+b*c+(d*e+f)*g	
15. 遇到 g，直接输出： a+b*c+(d*e+f)*g	
16. 此时已经没有新的字符了，依次出栈并输出操作直到栈为空： a+b*c+(d*e+f)*g	

将算术表达式转换成后缀表达式 postexp 的过程是

while (若 exp 未读完)

{　　从 exp 读取字符 ch;

ch 为数字：将后续的所有数字均依次存放到 postexp 中;

ch 为左括号'('：将'('进栈;

ch 为右括号')'：将 op 栈中'('以前的运算符依次出栈并存放到 postexp 中,再将'('退栈;

若 ch 的优先级高于栈顶运算符优先级，则将 ch 进栈；否则退栈并存入 postexp 中,再将 ch 进栈;

}

若字符串 exp 扫描完毕,则退栈 op 中的所有运算符并存放到 postexp 中。

在简单算术表达式中，只有*和/运算符的优先级高于+和-运算符的优先级。所以上述过程进一步改为：

```
while (若 exp 未读完)
{    从 exp 读取字符 ch;
     ch 为数字：将后续的所有数字均依次存放到 postexp 中;
     ch 为左括号'('：将'('进栈;
     ch 为右括号')'：将 op 栈中'('以前的运算符依次出栈并存放到 postexp 中,再将'('退栈;
     ch 为'+'或'-':将 op 栈中'('以前的运算符出栈并存放入 postexp 中,再将'+'或'-'进栈;
     ch 为'*'或'/':将 op 栈中'('以前的'*'或'/'运算符出栈并放入 postexp 中,再将'*'或'/'进栈;
}
```

明白了这个过程，现在就需要用代码实现了。对于各种运算符的优先级，可以使用整数来表示运算符的级别。可以定义一个函数来返回各种符号的优先级数字：

```
#include<bits/stdc++.h>
using namespace std;
stack<double>d_st;
stack<char>op_st;
int level(char s){//定义优先级
    if(s=='+'||s=='-')return 1;
    if(s=='*'||s=='/')return 2;
}
int main(){//40 + 21 * 100 - 7 / 11
    char str[200];
    double x;
    while(gets(str)){
        if(str[0]=='0')break;
        for(int i=0;i<strlen(str);i++){
            if(str[i]>='0'&&str[i]<='9'){//处置数值——开始
                x=str[i]-'0';//转化为数值
                while(str[i+1]>='0'&&str[i+1]<='9') {
                    x=x*10+str[i+1]-'0';
                    i++;
                }
                d_st.push(x);
            }//处理数值--结束
            else if(str[i]=='+'||str[i]=='-'||str[i]=='*'||str[i]=='/') {
                    //处理符号
                if(op_st.empty()||level(str[i])>level(op_st.top()))
                    op_st.push(str[i]); //运算符栈为空或者 当前字符优先级比栈高，入栈
                else if(level(str[i])<=level(op_st.top())) {
                    while(!op_st.empty()){//边出栈边计算
                        double b,c;
                        b=d_st.top();d_st.pop();//取得表达式栈顶元素，运算后重新入栈
                        c=d_st.top();d_st.pop();
                        switch(op_st.top()){
                            case '+':d_st.push(b+c);break;
```

```
                        case '-':d_st.push(c-b);break;
                        case '*':d_st.push(b*c);break;
                        case '/':d_st.push(c/b);break;
                    }
                    op_st.pop();
                }
                op_st.push(str[i]);
            }


        }
    }
    while(!op_st.empty()){//表达式读入完毕，栈不为空，继续计算
        double b,c;
        b=d_st.top();d_st.pop();
        c=d_st.top();d_st.pop();
        switch(op_st.top()){
            case '+':d_st.push(b+c);break;
            case '-':d_st.push(c-b);break;
            case '*':d_st.push(b*c);break;
            case '/':d_st.push(c/b);break;
        }
        op_st.pop();
    }
    printf("%.2lf\n",d_st.top());
    while(!d_st.empty())d_st.pop();
    }
    return 0;
}
```

课前练习

1.阅读程序	2. [2005]

#include <iostream>
using namespace std;
int main()
{
int i, j, n, t, f;
int a[101];
cin >> n;
for (i = 1; i <= n; i++)
cin >> a[i];
	#include<stdio.h>
	#include<string.h>
	int main()
	{
	char str[60];
	int len, i, j, chr[26];
	char mmin = 'z';
	scanf("%s", str);
	len = strlen(str);

```
    for (i = 1; i <= n - 1; i++)
    {
        f = 0;
        for (j = n; j >= 2; j--)
        {
            if (a[j - 1] > a[j])
            {
                t = a[j - 1];
                a[j - 1] = a[j];
                a[j] = t;
                f = 1;
            }
        }
        if (f == 0)
            break;
    }

    for (i = 1; i <= n; i++)
        cout << a[i] << " ";
    return 0;
}
```

输入：

8

12 1 -2 0 3 -4 7 9

输出：_____

```
    for (i = len - 1; i >= 1; i--)
        if (str[i - 1] < str[i]) break;
    if (i == 0)
    {
        printf("No result!\n"); return
0;
    }
    for (j = 0; j < i - 1; j++)
putchar(str[j]);
    memset(chr, 0, sizeof(chr));
    for (j = i; j < len; j++)
    {
        if (str[j] > str[i - 1] && str[j]
< mmin)
        mmin = str[j];
        chr[str[j] - 'a']++;
    }
    chr[mmin - 'a']--;
    chr[str[i - 1] - 'a']++;
    putchar(mmin);
    for(i = 0; i < 26; i++)
    for(j = 0; j < chr[i]; j++)
        putchar(i + 'a');
    putchar('\n');
    return 0;
}
```

输入：zzyzcccbbbaaa

输出：_____

习题

1[贪心的武松]曾经因打虎而闻名的武松在 X 年后接到了景阳岗动物园的求助信，信上说：最近我们动物园逃跑了几只老虎，请您把它们抓回来，谢谢！

武松接到信之后立刻上了山。正当他到半山腰时，突然跳出 n 只猛虎来。每只老虎都有一块虎牌，牌上写的是每一只虎最大拥有的体力，当武松与老虎 PK 时，若老虎的体力先用完，那么老虎 over，否则武松 over，求武松在 over 之前最多能干掉几只老虎？

【输入】第一行两个数字 n（老虎的只数），m（武松的体力）。第二行 n 个数字，分别表示每只老虎的体力（每只虎的体力按从小到大排列）。

【输出】一行，最多能干掉的老虎数。

【样例输入】

3 6

1 3 9

【样例输出】

2

请完善下列程序：

```
int main()
{
    int n, m, i, num;
    int a[101] = { 0 };
    cin >> n >> m;
    for (i = 1; i <= n; i++)
        cin >>  ①  ;
    num = 0;
    i = 1;
    while (m > 0 &&  ②  )
    {
        m = m - a[i];
        if (m > 0)
            num = num + 1;
         ③
    }
    cout <<  ④
        return 0;
}
```

3 [3410] 读程序写结果

```
include<stdio.h>
 int dp(int m,int n)
{
    if(!m||m==1||!n||n==1)
        return 1;
    if(m<n)
        return dp(m,m);
    return dp(m,n-1)+dp(m-n,n);
}
int main()
{
    int m,n;
     scanf("%d%d",&m,&n);
    printf("%d\n",dp(m,n));
    return 0;
}
```

输入

7 3

输出 _____

3 [合并石子]今天课间的时候，小明同学在学校的操场上发现了 n 堆大小不一的小石子，小明决定将它们合

并成一堆,但现在小明思考着这样一个问题：如何消耗最少的体力，把这 n 堆小石子合并成一堆？现已知合并所消耗的体力等于每次合并两堆小石子的重量之和,每次合并，他会把其中的两堆小石子合并到一起，n 堆小石子经过 n-1 次合并之后就只剩一堆了。

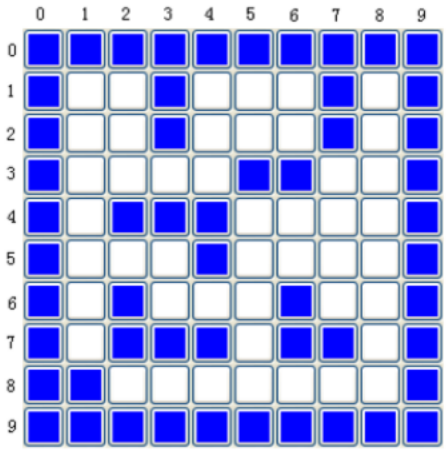
比如，n=3 时表示共有 3 堆，每堆重量分别是 2、1、9。一种合并方案是 2 和 9 合并，新堆重量是 11，耗费体力为 11;接着 11 与 1 合并，新堆重量是 12,耗费体力为 12, 因此总消耗体力是 11+12=23。另一种方案是 1 和 2 合并，新堆重量是 3,耗费体力为 3, 接着 3 和 9 合并，新堆重量是 12，耗费体力为 12，因此总消耗体力是 3+12=15。可以证明 这样合并就是最少耗费体力的方法。

【程序清单】

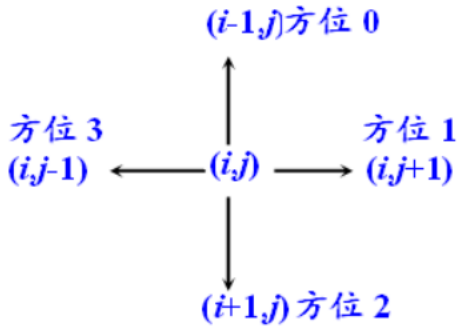
```
#include <iostream>
using namespace std;
int i,sum,n;
int a[101];
void mySort(int x)
{
    int i,j,temp;
    for(i=   ①   ; i<=n-1; i++)
    {
        for(j=n; j>=   ②   ; j--)
        {
            if(   ③   )
            {
                temp=a[j];
                a[j]=a[j-1];
                a[j-1]=temp;
            }
        }
    }
}
int main()
{
    cin>>n;
    for(i=1; i<=n; i++)
        cin>>a[i];
    sum=0;
    mySort(1);
    for(i=1; i<=n-1; i++)
    {
        a[i+1]=a[i]+a[i+1];
        sum=   ④   ;
        ___⑤___;
    }
    cout<<sum<<endl;
    return 0;
}
```

第 4 讲 栈的应用及 DFS

求迷宫问题就是在一个指定的迷宫中求出从入口到出口的路径。在求解时，通常用的是“穷举求解”的方法，即从入口出发，顺某一方向向前试探，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续试探，直至所有可能的通路都试探完为止。为了保证在任何位置上都能沿原路退回（称为回溯），需要用一个后进先出的栈来保存从入口到当前位置的路径。



对于迷宫中的每个方块，有上下左右 4 个方块相邻，如图 9 所示，第 i 行第 j 列的方块的位置记为(i,j)，规定上方方块为方位 0，并按顺时针方向递增编号。在试探过程中，假设从方位 0 到方位 3 的方向查找下一个可走的方块。



为了便于回溯，对于可走的方块都要进栈，并试探它的下一可走的方位，将这个可走的方位保存到栈中。

求解迷宫(xi,yi)到(xe,ye)路径的过程是：先将入口进栈（其初始方位设置为-1），在栈不空时循环：取栈顶方块（不退栈），若该方块是出口，则输出栈中所有方块即为路径。否则，找下一个可走的相邻方块，若不存在这样的方块，说明当前路径不可能走通，则回溯，也就是恢复当前方块为 0 后退栈。若存在这样的方块，则将其方位保存到栈顶元素中，并将这个可走的相邻方块进栈（其初始方位设置为-1）。

为了保证试探的可走相邻方块不是已走路径上的方块，如(i,j)已进栈，在试探(i+1,j)的下一可走方块时，又试探到(i,j)，这样可能会引起死循环，为此，在一个方块进栈后，将对应的 a 数组元素值改为-1（变为不可走的相邻方块），当退栈时（表示该栈顶方块没有可走相邻方块），将其恢复为 0。

4.1 [3727]迷宫问题

给定一个 M×N 的迷宫图，求从指定入口(1,1)到出口(M,N)有多少种走法。例如迷宫图如图所示(M=10，N=10），其中的方块图表示迷宫。对于图中的每个方块，用空白表示通道，用阴影表示墙。要求所求路径必须是简单路径，即在求得的路径上不能重复出现同一通道块。

如下图所示，蓝色的表示墙壁，白色的表示可走的方块，题目求解从入口位置 1（坐标（1,1））出发，到达位置 18（坐标（5,5））。

1. 确定搜索"到达目的地"的条件。假设(xe,ye)表示迷宫出口位置，则到达迷宫出口的代码可以表示成：

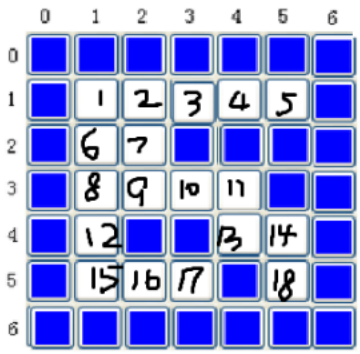
   if(xe==m&&ye==n){sum++;//根据题目做相关操作，本题到达目的题做求和操作}
2. 确定搜索方向。对于每一个方块（位置）而言，共有四种走法：上下左右。对于计算机而言所有的操作都是有序的，因此需要规定搜索过程中，寻路的顺序，如上下左右，左右上下等，本解题规定采用的搜索顺序为左上右下，定义方向数组：

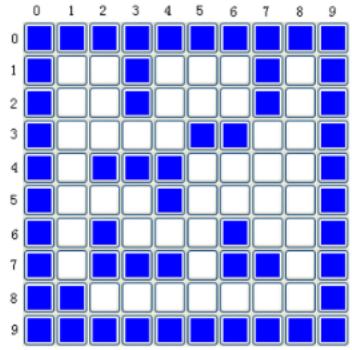
   int dir[4][2]={{-1,0},{0,1},{1,0},{0,-1}};//定义方向
3. 记录搜索过程中走过位置的状态。当一个位置被走过以后，为了避免来回重复走，需要记录当前的位置有没有被走过，如果被走过，这个位置就不能再走。假设用 vis[][]数组记录位置的状态，初始时 vis 数组

的值全部初始化为 0，vis[x][y]=0 **表示该位置没有走过，vis[i][j]=1 表示该位置已经被走过。**实现代码如下：

vis[x][y]=1;//用 1 表示该位置没有被走过

4. 确定"满足条件"。在什么状态下可以继续搜索下一个位置呢？很显然就是下一个位置是可走的并且是没有被走过的





```
#include<iostream>
#include<cstring>
using namespace std;
int a[12][12];//定义迷宫
int dir[4][2]={{-1,0},{0,1},{1,0},{0,-1}};//定义方向
int vis[12][12];//标记有没有走过
int sum=0,xe,ye;//xe,ye 终点
void dfs(int x,int y){
    if(x==xe&&y==ye){
        sum++;//到达目的地
    }
    else{
        for(int i=0;i<4;i++){
            int xx=x+dir[i][0];//下一个点
            int yy=y+dir[i][1];
            if(a[xx][yy]==0&&vis[xx][yy]==0){
                vis[xx][yy]=1;//记录走过
                dfs(xx,yy);
                vis[xx][yy]=0;//回溯
            }
        }
    }
}
int main(){
    int m,n,i,j;
    cin>>m>>n;
    xe=m,ye=n;
    memset(a,1,sizeof(a));
    for(i=1;i<=m;i++)
```

```
        for(j=1;j<=n;j++)
            cin>>a[i][j];

    vis[1][1] = 1;//记录第一点没访问过
    dfs(1,1);
    cout<<sum<<endl;
    return 0;
}
```

栈模拟

```
#include<iostream>
#include<cstring>
#include<stack>
using namespace std;
int a[12][12];//定义迷宫
int dir[4][2]={{-1,0},{0,1},{1,0},{0,-1}};//定义方向
int vis[12][12];//标记有没有走过
int sum=0,xe,ye;//xe,ye 终点

struct node{
  int x,y;
  int dir;
};

int main(){
  int m,n,i,j;
  cin>>m>>n;
  int xe=m,ye=n;
  memset(a,1,sizeof(a));
  for(i=1;i<=m;i++)
    for(j=1;j<=n;j++)
      cin>>a[i][j];

  node start,t;//初始化起点
  start.x=1,start.y=1,start.dir=0;
  stack<node>st;
  st.push(start);//起点入栈
  while(!st.empty())
  {
    node e=st.top();//获取栈点元素
    vis[e.x][e.y]=1;//记录状态为已处理
    if(e.x==xe&&e.y==ye)
    {//到达出口
```

```
            sum++;
            vis[e.x][e.y]=0;//重新可用
            st.pop();
        }
        while(!st.empty()&&st.top().dir==4)//走不通回退
            vis[st.top().x][st.top().y]=0,st.pop();
        if(st.empty())
            break;//处理完毕
        e=st.top();
        st.pop();//弹出栈点元素
        for(int i=e.dir;i<4;i++)
        {//处理下一个点
            int xx=e.x+dir[i][0];//下一个点
            int yy=e.y+dir[i][1];
            if(a[xx][yy]==0&&vis[xx][yy]==0){//可走
                node r;
                r.x=xx,r.y=yy,r.dir=0;
                e.dir=i+1;//更新 e 的方向
                st.push(e);//更新栈内 e 的值，因为修改了方向
                st.push(r);
                break;//找到一个可走点，退出继续搜索
            }
            if(i==3)
            {//该方向搜索完毕
                e.dir=i+1;
                st.push(e);
            }
        }
    }
    cout<<sum<<endl;
    return 0;
}
```

4.2 [2769]迷宫

一天 Extense 在森林里探险的时候不小心走入了一个迷宫，迷宫可以看成是由 n * n 的格点组成，每个格点只有 2 种状态，.和#，前者表示可以通行后者表示不能通行。同时当 Extense 处在某个格点时，他只能移动到东南西北(或者说上下左右)四个方向之一的相邻格点上，Extense 想要从点 A 走到点 B，问在不走出迷宫的情况下能不能办到。如果起点或者终点有一个不能通行(为#)，则看成无法办到。

输入

第 1 行是测试数据的组数 k，后面跟着 k 组输入。每组测试数据的第 1 行是一个正整数 n (1 ≤ n ≤ 100)，表示迷宫的规模是 n * n 的。接下来是一个 n * n 的矩阵，矩阵中的元素为.或者#。再接下来一行是 4 个整数 ha, la, hb, lb，描述 A 处在第 ha 行，第 la 列，B 处在第 hb 行，第 lb 列。注意到 ha, la, hb, lb 全部是从 0 开始计数的。

输出

k 行，每行输出对应一个输入。能办到则输出"YES"，否则输出"NO"。

样例输入	样例输出
2 3 .## ..# #.. 0 0 2 2 5 ..... ###.# ..#.. ###.. ...#. 0 0 4 0	YES NO

分析：

1. 搜索的目的地是什么？ _____

2. 搜索的“满足条件”时什么？ _____

```
#include <iostream>
using namespace std;
int n;
char a[100][100];
int ha,la,hb,lb;//起点、终点
int d[4][2]={1                          };//方向数组
int flag=0;//用于标记是否走到出口
void dfs(int x,int y){
        if(x>n||y>n)return ;
        if(  2                  ){
                flag=1;
                return;
        }
        else{
                for(int i=0;i<4;i++){
                        int xx=3
                        int yy=4
                        if(a[xx][yy]=='.'){
                                a[xx][yy]='#';
                                dfs(5                   );
                                 6                  ;
                        }
                }
```

```
        }
}
int main(){
        int k;
        cin>>k;
        while(k--){
                flag=0;
                cin>>n;
                for(int i=0;i<n;i++)
                        for(int j=0;j<n;j++)
                                cin>>a[i][j];
                cin>>ha>>la>>hb>>lb;
                7_____________________
                if(8_____________________) cout<<"YES"<<endl;
                else cout<<"NO"<<endl;

        }
        return 0;
}
```

4.3［3299］n 皇后问题

3299：会下国际象棋的人都很清楚：皇后可以在横、竖、斜线上不限步数地吃掉其他棋子。如何将 n 个皇后放在棋盘上（有 n × n 个方格），使它们谁也不能被吃掉！这就是著名的八皇后问题。

输入 n 输出，输出 放置的具体方法，以及总共的方法数。

分析：

1. 搜索的目的地是什么？ _____
2. 搜索的“满足条件”时什么？ _____

```
#include<iostream>//八皇后
using namespace std;
int sum=0;
void Print(int n, int array[]){
    1_____________________
  for (int j = 1; j <= n; j++)
     cout<<array[j]<<" ";  //输出最终的棋盘布局，每个数组元素值表示各行棋盘所在的该列放置皇后
  cout<<endl;
}
bool isOk(int i, int array[]) { //检查当前棋盘布局是否合理
  for (int k = i - 1; k >= 1; k--)
    if(2_____________________________________)
      return false;
  return true;
```

```
}
void search( 3                                ) {
   if(   4                  ) Print(n, array);  //如果 i 已经大于最大的行数了，则打印结果
   else {
      for(int j = 1; j <= n; j++) {
         _____________//在第 i 行第 j 列放置一个皇后
         if(isOk(i, array)) search( 5                        ); //当前布局合理，进行下一行的布局
         _6                          //移走第 i 行第 j 列放置的皇后，  回溯。
      }
   }
}
int main() {
   int n;
   cin>>n;
   int Chess[100] = {0}; //棋盘的初始状态，棋盘上无任何皇后
   search(1, n, Chess); //摆放皇后
   cout<<sum<<endl;
   return 0;
}
```

4.4 [2096]狗的诱惑

一只小狗在一个古老的迷宫里找到一根骨头，当它叼起骨头时，迷宫开始颤抖，它感觉到地面开始下沉。它才明白骨头是一个陷阱，它拼命地试着逃出迷宫。

迷宫是一个 N×M 大小的长方形，迷宫有一个门。刚开始门是关着的，并且这个门会在第 T 秒钟开启，门只会开启很短的时间（少于一秒），因此小狗必须恰好在第 T 秒达到门的位置。每秒钟，它可以向上、下、左或右移动一步到相邻的方格中。但一旦它移动到相邻的方格，这个方格开始下沉，而且会在下一秒消失。所以，它不能在一个方格中停留超过一秒，也不能回到经过的方格。

小狗能成功逃离吗？请你帮助他。

分析：

1. 搜索的目的地是什么？ _____

2 搜索的“满足条件”时什么？ _____

3 是否需要剪枝？

```
#include <stdio.h>
#include <math.h>  //用到了求绝对值的函数 abs

//迷宫地图
//X: 墙壁，小狗不能进入//S: 小狗所处的位置//D: 迷宫的门//.: 空的方格
char map1[9][9];
int n,m,t,di,dj; //(di,dj):门的位置
bool escape;
```

```
int dir[4][2]={{0,-1},{0,1},{1,0},{-1,0}}; //分别表示下、上、左、右四个方向
//表示起始位置为(si,sj)，要求在第 cnt 秒达到门的位置
void dfs( int si, int sj, int cnt )
{
    int i,temp;
    if( si>n || sj>m || si<=0 || sj<=0 ) return;
    if( si==di && sj==dj && cnt==t )
    { escape = 1; return; }
    //abs(x-ex)+abs(y-ey)表示现在所在的格子到目标格子的距离(不能走对角线)
    //t-cnt 是实际还需要的步数，将他们做差
    //如果 temp < 0 或者 temp 为奇数，那就不可能到达！
    temp = (t-cnt) - abs(si-di) - abs(sj-dj);
    if( temp<0 || temp%2 ) return;
    for( i=0; i<4; i++ )
    {
        if( map1[ si+dir[i][0] ][ sj+dir[i][1] ] != 'X')
        {
            map1[ si+dir[i][0] ][ sj+dir[i][1] ] = 'X';
            dfs(si+dir[i][0], sj+dir[i][1], cnt+1);
            if(escape) return;
            map1[ si+dir[i][0] ][ sj+dir[i][1] ] = '.';
        }
    }
    return;
}
int main()
{
    int i, j, si, sj;    //循环变量及小狗的起始位置
    while( scanf("%d%d%d", &n, &m, &t) )
    {
        if( n==0 && m==0 && t==0 )   break;   //测试数据结束
        int wall = 0;   char temp;
        scanf( "%c", &temp );    //见备注
        for( i=1; i<=n; i++ )
        {
            for( j=1; j<=m; j++ )
            {
                scanf( "%c", &map1[i][j] );
                if( map1[i][j]=='S' ) {   si=i;   sj=j;   }
                else if( map1[i][j]=='D' ) {   di=i;   dj=j;   }
                else if( map1[i][j]=='X' ) wall++;
            }
            scanf( "%c", &temp );
        }
```

```
            if( n*m-wall <= t )  //搜索前的剪枝
            {    printf( "NO\n" );   continue;   }
            escape = 0;
            map1[si][sj] = 'X';
            dfs( si, sj, 0 );
            if( escape )   printf( "YES\n" );      //成功逃脱
            else    printf( "NO\n" );
        }
        return 0;
}
```

4.5［1286］反素数

对于任何正整数 x，其约数（因子）的个数记作 g(x)。例如 g(1)=1、g(6)=4， (约数为 1,2,3,6)。 如果某个正整数 x 满足：g(x)> g(i) 0< i< x，则称 x 为反质数。例如，整数 1，2，4，6 等都是反质数。 现在给定一个数 N，你能求出不超过 N 的最大的反质数么？

输入

输入只有一行，一个数 N（1< =N< =2,000,000,000）。

输出 输出也只有一行，为不超过 N 的最大的反质数。

样例输入 1000

样例输出 840

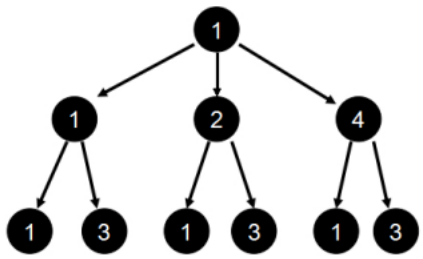
从反素数的定义中可以看出两个性质：

（1）一个反素数的所有质因子必然是从 2 开始的连续若干个质数，因为反素数是保证约数个数为 x 的这个数 n 尽量小

（2）同样的道理，如果 $n=2^{t1}*3^{t2}*5^{t3}*7^{t4}\ldots$，那么必有 t1>t2>t3>t4...

从上面的性质中可以看出，我们要求最小的 x，它的约数个数为 n，那么可以利用搜索来解。以前我们求一个数的所有因子也是用搜索，比如 $n=p_1^{s1}p_2^{s2}\ldots p_k^{sk}$，以每一个 p_i 为树的一层建立搜索树，深度为 k。

以 $12=2^2*3$ 为例进行说明，建树如下。可以看出从根节点到每一个叶子结点这条路径上的所有数字乘起来都是 12 的约数，所以 12 有 6 个约数。搜索的思路就明显了，从根节点开始进行深搜，到叶子结点。



参考：

```
#include <iostream>
using namespace std;
int p[16]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53};
long long int ans,n;
int Max=0;
void dfs(int dep,long long int now,int num)
{
  if (dep>=16) return ;
  if (num>Max)
```

```
  {
    Max=num;
    ans=now;
  }
  if (num==Max&&ans>now) ans=now;
  for (int i=1;i<=63;i++)
  {
    if (p[dep]*now>n) break;
    dfs(dep+1,now*=p[dep],num*(i+1));
  }
}
int main ()
{
  cin>>n;
  ans=0x3f3f3f3f;
  dfs(0,1,1);
  cout<<ans<<endl;
  return 0;
}
```

课前练习

1 阅读程序

```
#include <iostream>
using namespace std;
int main()
{
    int i, j, k, x, n, ans, s;
    int a[501];
    bool f;
    cin >> n >> x;
    for (i = 1; i <= n; i++)
        cin >> a[i];
    i = 1;
    j = n;
    f = false;
    s = 0;
    while (i <= j && !f)
    {
        k = (i + j) / 2;
        s = s + 1;
        if (x == a[k])
        {
```

```
            ans = k;
            f = true;
        }
        else if (x < a[k])
            j = k - 1;
        else
            i = k + 1;
    }
    if (f)
        cout << ans << " " << s;
    else
        cout << s;
    return 0;
}
```

输入 1：
4 11
7 9 11 17
输出 1：_____
输入 2：
20 256
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 101 105 109 110 138
输出 2：_____

2.[2010]（哥德巴赫猜想）哥德巴赫猜想是指，任一大于 2 的偶数都可写成两个质数之和。迄今为止，这仍然是一个著名的世界难题，被誉为数学王冠上的明珠。试编写程序，验证任一大于 2 且不超过 n 的偶数都能写成两个质数之和。

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

int main()
{
    const int SIZE = 1000;
    int n, r, p[SIZE], i, j, k, ans;
    bool tmp;
    cin>>n;
    r = 1;
    p[1] = 2;
    for (i = 3; i <= n; i++) {
        ____①_____;
        for (j = 1; j <= r; j++)
```

```
            if (i %_____②_____ == 0) {
                tmp = false;
                break;
            }
        if (tmp) {
            r++;
        _____③_____ ;
        }
    }
    ans = 0;
    for (i = 2; i <= n / 2; i++) {
        tmp = false;
        for (j = 1; j <= r; j++)
            for (k = j; k <= r; k++)
                if (i + i == ____④_____) {
                    tmp = true;
                    break;
                }
        if (tmp)
            ans++;
    }
    cout<<ans<<endl;
    return 0;
}
```

若输入 n 为 2010，则输出____⑤_____时表示验证成功，即大于 2 且不超过 2010 的偶数都满足哥德巴赫猜想。

习题

1 阅读程序

```
#include <iostream>
using namespace std;
int main()
{
    int i, j, n, s=0, x;
    int f[101]={0};
    cin>>n;
    for(i=1;i<=n;i++)
    {
        cin>>x;
        f[x]=f[x]+1;
        for(j=x+1;j<=100;j++)
            if (f[j]>0)s=s+1;
```

```
    }
    cout<<s;
    return 0;
}
```

输入：

6

12 19 14 17 16 16

输出：_____

2. [2009]**(国王放置)** 在 n*m 的棋盘上放置 k 个国王，要求 k 个国王互相不攻击，有多少种不同的放置方法。假设国王放置在第(x,y)格，国王的攻击的区域是:(x-1,y-1),(x-1,y),(x-1,y+1),(x,y-1),(x,y+1),(x+1,y-1),(x+1,y),(x+1,y+1)。读入三个数 n,m,k，输出答案。题目利用回溯法求解。棋盘行标号为 0~n-1，列标号为 0~m-1。

```
#include <iostream>
using namespace std;
int n,m,k,ans;
int hash[5][5];
void work(int x,int y,int tot){
    int i,j;
    if (tot==k){
        ans++;
        return;
    }
    do{
        while (hash[x][y]){
            y++;
            if (y==m){
                x++;
                y=_____①_____;
            }
            if (x==n)
                return;
        }
        for (i=x-1;i<=x+1;i++)
            if (i>=0&&i<n)
                for (j=y-1;j<=y+1;j++)
                    if (j>=0&&j<m)
                        _____②_____;
        _____③_____;
        for (i=x-1;i<=x+1;i++)
            if (i>=0&&i<n)
                for (j=y-1;j<=y+1;j++)
```

```
                    if (j>=0&&j<m)
                        ④      ;
        y++;
        if (y==m){
            x++;
            y=0;
        }
        if (x==n)
            return;
    }
    while (1);
}
int main(){
    cin >> n >> m >> k;
    ans=0;
    memset(hash,0,sizeof(hash));
        ⑤      ;
    cout << ans << endl;
    return 0;
}
```

3[1110]使用栈实现进制转换

使用栈将一个很长（>30）的十进制数转换为二进制数

输入 若干个很长的十进制数 每行一个输出 转换为二进制，每行输出一个

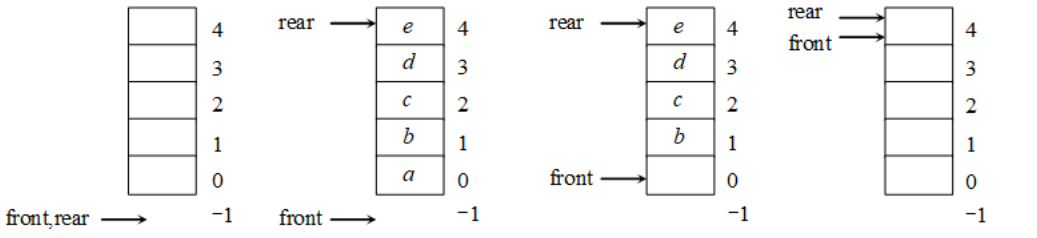
```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    char a[10000];
     int len;
    while(scanf("%s",a)!=EOF){
        int b[10000]={0};
          1
        for(int i=0;i<len;i++){
            b[i]=a[i]-'0';
        }
        int i=0;
        stack<int>s;
        while(i<len-1||b[i]>0){
            2                          //模拟除法，把余数入栈
            int k=0;
            for(int j=i;j<len;j++){ //按位相除
                3
                b[j]=k/2;//把余数和被除数的次高位组合
```

```
                k= 4
            }
            if(b[i]==0)i++;
        }
        while(s.size()){
            printf("%d",s.top());
            5
        }
        printf("\n");
    }
    return 0;
}
```

第 5 讲 队列

队列（简称为队）是一种操作受限的线性表，其限制为仅允许在表的一端进行插入，而在表的另一端进行删除。把进行插入的一端称做队尾（rear），进行删除的一端称做队头或队首（front）。向队列中插入新元素称为进队或入队，新元素进队后就成为新的队尾元素；从队列中删除元素称为出队或离队，元素出队后，其直接后继元素就成为队首元素。队列的插入和删除操作分别是在表的各自的一端进行的，元素进队只能从队尾进，不能从队头或中间位置进队，如下图所示。每个元素必然按照进入的次序出队，所以又把队列称为先进先出表。

下图是所示是一个队列的动态示意图，图中 front 指针指向队首位置(实际上是队首元素的前一个位置)，rear 指针指向队尾位置（正好是队尾元素的位置）。图（a）表示一个空队；图（b）表示插入 5 个数据元素后的状态；图（c）表示出队一次后的状态；图（d）表示再出队 4 次后的状态。



（a）空队　（b）*a,b,c,d,e* 元素进队　（c）出队一次　（d）出队 4 [例

5.1 队列的操作

队列可以采用顺序存储结构，分配一块连续的存储空间 data（大小为常量 MaxSize）来存放队列中元素，并用变量 front（队头指针）指向队头元素，用变量 rear（队尾指针）指向队头元素。队列的存储方式和实现和实现也有数组模拟和链表模拟两种，本讲义只介绍数组实现。

1 队列的定义

```
#define MaxSize 100005
 struct Stack{
      int Data[MaxSize];
      int front,rear;
};
```

2 队列的初始化

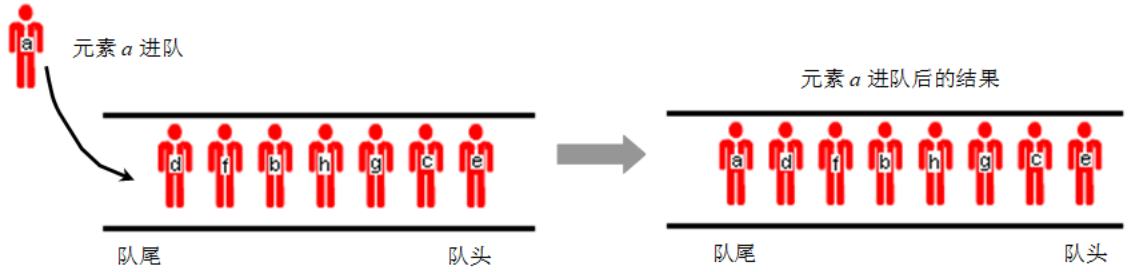
初始时置 front=rear=-1；

3 判空

若队列满足 front==rear 条件，则返回 true；否则返回 false。对应的算法如下：

```
 bool QueueEmpty()
{
     return(front==rear);
}
```

4 入队操作（压栈）



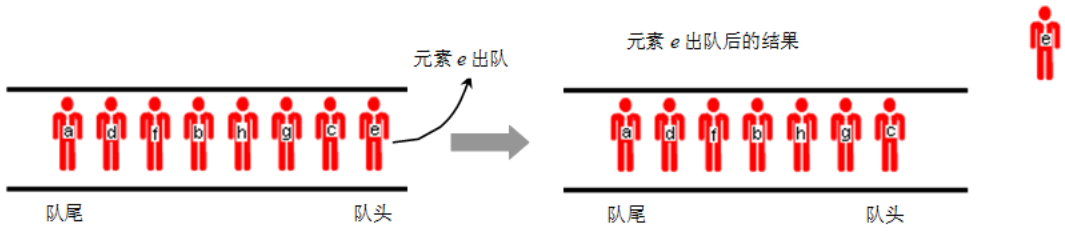
5 出栈

在进队运算中，当队列不满的条件下，先将队尾指针 rear 增 1，然后元素 e 放到该位置处。对应的算法如下：

```
bool enQueue(string e)
{     if (rear==MaxSize-1)          //队满上溢出
      return false;                 //返回 false
      rear++;
      data[rear]=e;
      return true;
}
```

6 出队操作

元素出队只能从队头出，不能从队头或中间位置出队，如下图所示。



在出队运算中，当队列不为空的条件下，将队首指针 front 增 1，并将该位置的元素值赋给 e。对应的算法如下：

```
bool deQueue( string e)
{     if (front==rear)              //队空下溢出
      return false;                 //返回 false
      front++;
      e=data[front];
      return true;
}
```

练习：

队列初始化：_____

队列为空的判断：_____

队列为满的判断：_____

队列为空的判断：_____

入队的判断：_____

出队的判断：_____

5.2 循环队列

在非循环队列中，元素进队时队尾指针 rear 增 1，元素出队时队头指针 front 增 1，当进队 MaxSize 个元素后，满足队满的条件即 rear==MaxSize-1 成立，此时即使出队若干元素，队满条件仍成立（实际上队列中有空位置），这是一种假溢出。为了能够充分地使用数组中的存储空间，把数组的前端和后端连接起来，形成一个循环的顺序表，即把存储队列元素的表从逻辑上看成一个环，称为循环队列。

循环队列首尾相连，当队首指针 front=MaxSize-1 后，再前进一个位置就自动到 0，这可以利用求余的运算(%)来实现：

队首指针进 1：front=(front+1)%MaxSize

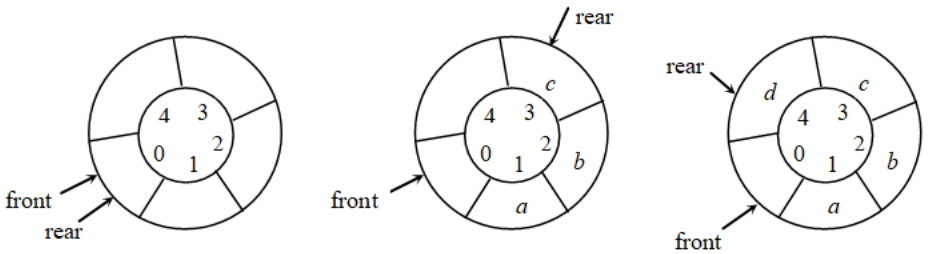
队尾指针进 1：rear=(rear+1)%MaxSize

循环队列的队头指针和队尾指针初始化时都置 0：front=rear=0。在进队元素和出队元素时，队头和队尾指针都循环前进一个位置。

那么，循环队列的队满和队空的判断条件是什么呢?显然**循环队列为空条件是 rear==front**。如果进队元素的速度快于出队元素的速度，队尾指针很快就赶上了队首指针，此时可以看出循环队列的队满条件也为 rear==front。

怎样区分这两者之间的差别呢?通常约定在进队时少用一个数据元素空间，以队尾指针加 1 等于队首指针作为队满的条件，即**队满条件为：(rear+1)% MaxSize==front。队空条件仍为 rear==front。**

下图所示说明了循环队列的几种状态，这里假设 MaxSize 等于 5。图（a）为空队，此时 front=rear=0；图（b）中有 3 个元素，当进队元素 d 后，队中有 4 个元素，此时满足队满的条件。



（a）空队　（b）a、b、c 元素进队　（c）d 元素进队⇨队满

（d）出队 2 次　（e）出队 2 次⇨队空

在这样的循环队列中，实现队列的基本运算算法如下。

（1）判断队列是否为空 QueueEmpty(q)

若队列满足 front==rear 条件，返回 true；否则返回 false。对应的算法如下：

```
bool QueueEmpty()
{
    return (front==rear);
}
```

（2）进队列 enQueue(q,e)

在队列不满的条件下，先将队尾指针 rear 循环增 1，然后将元素 e 放到该位置处。对应的算法如

下：

```
bool enQueue(string e)
{    if ((rear+1)%MaxSize==front)      //队满上溢出
     return false;                 //返回 false
     rear=(rear+1) % MaxSize;
     data[rear]=e;
     return true;
}
```

（3）出队列 deQueue(q,e)

在队列不为空的条件下，将队首指针 front 循环增 1，并将该位置的元素值赋给 e。对应的算法如下：

```
bool deQueue(ref string e)
{    if (front==rear)                 //队空下溢出
     return false;                 //返回 false
     front=(front + 1) % MaxSize;
     e=data[front];
     return true;
}
```

[例 4.1] 对于循环队列来说，如果知道队头指针和队列中元素个数，则可以计算出队尾指针。也就是说，可以用队列中元素个数代替队尾指针。设计出这种循环队列的进队、出队、判队空和求队中元素个数的算法。

解：本例的循环队列包含 data 数组、队头指针 rear 和队中元素个数 count 字段。初始时 front 和 count 均置为 0。

队空条件为 count==0；

队满的条件为 count==MaxSize；

元素 e 进队操作是，先根据队头指针和元素个数求出队尾指针 rear，将 rear 循环增 1，然后将元素 e 放置在 rear 处；

出队操作是，先将队首指针循环增 1，然后取出该位置的元素。

5.3 STL 中的队列

包含头文件：#include <queue>

使用命名空间：using namespace std;

定义栈的方法：

```
queue<char> S1;    //队列中的结点为字符
queue<int> S2;     //队列中的结点为整型数据
queue<pos> S3; //队列中的结点为自定义结构体 pos 变量
```

queue 的基本操作举例如下：

queue 入队，如例：q.push(x); 将 x 接到队列的末端。

queue 出队，如例：q.pop(); 弹出队列的第一个元素，注意，并不会返回被弹出元素的值。

访问 queue 队首元素，如例：q.front()，即最早被压入队列的元素。

访问 queue 队尾元素，如例：q.back()，即最后被压入队列的元素。

判断 queue 队列空，如例：q.empty()，当队列空时，返回 true。
访问队列中的元素个数，如例：q.size()

```
#include<queue>
#include<iostream>
using namespace std;
void clear(queue<int>&q)
{
    queue<int>empty;
    swap(empty,q);
 }
int main()
{
    queue<int>q;
    q.push(1);                    //在队列末尾依次插入 1 2 3
    q.push(2);
    q.push(3);

    int u=q.back();               //返回队列中最后一个元素
    cout<<"队列最后一个元素为："<<u<<endl;

    int v=q.front();          //返回队列中第一个元素
    cout<<"队列第一个元素为:"<<v<<endl;

    q.pop();                      //删除第一个元素
    v=q.front();
    cout<<"队列第一个元素为："<<v<<endl;

    int size=q.size();                //size 返回元素个数
    cout<<"队列中存在"<<size<<"个元素"<<endl;

    cout<<"判断队列是否为空，空输出 1  否则输出 1："<<endl;
    int flag=q.empty();               //判断队列是否为空，为空返回 1，否则返回 0
    cout<<flag<<endl;
    clear(q);   //queue 中没有 clear 操作，用函数定义 clear 函数，使用 swap
    cout<<q.empty()<<endl;
    return 0;
 }
```

输出结果：

```
    队列最后一个元素为：3
    队列第一个元素为:1
    队列第一个元素为：2
    队列中存在 2 个元素
    判断队列是否为空，空输出 1  否则输出 1：
```

0

1

5.4 案例分析

1 [2882] 周末舞会

假设在周末舞会上，男士们和女士们进入舞厅时，各自排成一队。跳舞开始时，依次从男队和女队的队头上各出一人配成舞伴。规定每个舞曲能有一对跳舞者。若两队初始人数不相同，则较长的那一队中未配对者等待下一轮舞曲。现要求写一个程序，模拟上述舞伴配对问题。

输入 第一行两队的人数;第二行舞曲的数目。

输出 配对情况。

样例输入	样例输出
4 6 7	1 1 2 2 3 3 4 4 1 5 2 6 3 1

```
#include <iostream>
#include <queue>//头文件
using namespace std;
int main()
{
  int n,m,k;
  while(~scanf("%d%d%d",&n,&m,&k))
  {
    queue<int>q1,q2;//定义一个队列
    //初始化，保持栈初始状态为空
    while(!q1.empty())q1.pop();
    while(!q2.empty())q2.pop();

    for(int i=1;i<=n;i++)
      q1.push(i);//往栈里 添加男生
    for(int i=1;i<=m;i++)
      q2.push(i);//往栈里 添加女生
    while(k--)
    {
      int x1=q1.front();//取出队头配对
      int x2=q2.front();
      q1.pop(),q2.pop();
```

```
        q1.push(x1),q2.push(x2);//配对后的男生女生重新排队
        printf("%d %d\n",x1,x2);
      }
    }
    return 0;
}
```

2 [7290] 模拟队列

实现一个队列，队列初始为空，支持四种操作：
(1) “push x” – 向队尾插入一个数 x；
(2) “pop” – 从队头弹出一个数；
(3) “empty” – 判断队列是否为空；
(4) “query” – 查询队头元素。
现在要对队列进行 M 个操作，其中的每个操作 3 和操作 4 都要输出相应的结果。
输入格式　第一行包含整数 M，表示操作次数。　接下来 M 行，每行包含一个操作命令，操作命令为”push x”，”pop”，”empty”，”query”中的一种。
输出格式　对于每个”empty”和”query”操作都要输出一个查询结果，每个结果占一行。　其中，”empty”操作的查询结果为“YES”或“NO”，”query”操作的查询结果为一个整数，表示队头元素的值。

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

int main(){
    queue <int> q;
    int m;
    cin >> m;
    while(m --){
        string op;
        int x;
        cin >> op;

        if(op =="push"){
            cin >> x;
            q.push(x);//入队
        }
        else if(op =="pop"){
            q.pop();//出队
        }
        else if(op =="query"){
            cout<<q.front()<<endl;//输出首元素
        }
```

```
include<iostream>
#include<cstring>
using namespace std;
int n,k,q=1,h=0,a[1000001];
string l;
void push(int x){
    h++;
    1_____
}
void pop(){ q++;}
bool empty(){
    if( 2_____ ) return true;
    return false;
}
int top(){ return a[q];}
int main(){
    cin>>n;
    while(n--){
        cin>>l;
        if(l=="push"){
            cin>>k;
            push(k);
        }
```

```
        else{
            if(q.empty())cout<<"YES"<<endl;
            else cout << "NO" << endl;
        }
    }
    return 0;
}
```

```
        else if(l=="empty"){
            if(!empty())cout<<"NO"<<endl;
            else cout<<"YES"<<endl;
        }
        else if(l=="pop") pop();
        else cout<<top()<<endl;
    }
    return 0;
}
```

3 [2884] 围圈报数

有 n 个人依次围成一圈，从第 1 个人开始报数，数到第 m 个人出列，然后从出列的下一个人开始报数，数到第 m 个人又出列，…，如此反复到所有的人全部出列为止。设 n 个人的编号分别为 1，2，…，n，打印出列的顺序。

输入

n 和 m。

输出

出列的顺序。

样例输入

4 17

样例输出

1 3 4 2

分析：将整个报数的过程看成是一个队列，但某个数报完后，如果不符合要去，就加入到队列，如果该数正好是所求得数就出列。

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
  1
  int n, m;
  cin >> n >> m;
  for (int i = 1; i <= n; i++)
  {
    2
  }
  while (3)
  {
    for (int i = 1; i <= m-1; i++)
    {
```

```
        int x =  4
        student. 5
        student. 6
    }
    cout << student.front() << " ";
    student.pop();
  }
}
```

4 [2883] Blah 数集

大数学家高斯小时候偶然间发现一种有趣的自然数集合 Blah，对于以 a 为基的集合 Ba 定义如下：
(1)a 是集合 Ba 的基，且 a 是 Ba 的第一个元素；
(2)如果 x 在集合 Ba 中，则 2x+1 和 3x+1 也都在集合 Ba 中；
(3)没有其他元素在集合 Ba 中了。
现在小高斯想知道如果将集合 Ba 中元素按照升序排列，第 N 个元素会是多少？

输入	输出
输入包括很多行，每行输入包括两个数字，集合的基 a(1≤a≤50))以及所求元素序号 n(1≤n≤1000000)。 样例输入 1 100 28 5437	对于每个输入，输出集合 Ba 的第 n 个元素值。 样例输出 418 900585

分析：

题目要求输出集合中第 n 小的数，我们可以按照从小到大的顺序把序列中的前 n 个数计算出来，注意数集中除了第一个数 a 以外，其余每一个数 y 一定可以表示成 2x+1 或者 3x+1 的形式，其中 x 是数集中某一个数。因此除了第一数 a 以外，可以把数集 q[]的所有数分成两个子集，一个是用 2x+1 来表示的数的集合 1，另一个是用 3x+1 来表示的数的集合 2，两个集合要保持有序非常容易，只需用两个指针 two 和 three 来记录，其中 two 表示集合 1 下一个要产生的数是由 q[two]*2+1 得到，three 表示集合 2 下一个要产生的数是由 q[three]*3+1 得到。接下来比较 q[two]*2+1 和 q[three]*3+1 的大小关系：

(1) q[two]*2+1<q[three]*3+1：如果 q[two]*2+1 与 q[rear-1]不等，则把 q[two]*2+1 加到数集中，即：q[rear++]= q[two]*2+1；two++；

如果 q[two]*2+1 与 q[rear-1]相等，因为集合的唯一性，q[two]*2+1 不能加入数集，但 two 同样要加 1。

(2) q[two]*2+1>=q[three]*3+1：处理方法同上。

如此循环直到产生出数集的第 n 个数。

```
#include <iostream>
#include<queue>
using namespace std;
int a,n,k;
int main() {
    while(cin>>a>>n){
```

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

const int N=1000100;
#define long long LL
```

```
        k=1;
        __1_____________
        while(k<n){
            __2_____________
            q2.push(a*3+1);
            if(__3_____________){
                a=q1.front();
                q1.pop();
            }
            else if(__4_____________){
                a=q2.front();
                q2.pop();
            }
            else{
                __5_____________
                __6_____________
                __7_____________
            }
            k++;
        }
        cout<<a<<endl;
    }
    return 0;
}
```

```
long long q[N];
int a,n;
void work(int a,int n)
{
    int rear=2;
    q[1]=a;
    int two=1,three=1;
    while(rear<=n)
    {
        LL t1=q[two]*2+1,t2=q[three]*3+1;
        int t=min(t1,t2);
        if (t1<t2) two++;
        else three++;
        if (t==q[rear-1]) continue;
        q[rear++]=t;
    }
    cout<<q[n]<<endl;
}

int main()
{
    while (cin>>a>>n) work(a,n);
    return 0;
}
```

5[1109] 通话记录

使用 3 个队列，分别保留手机上最近 10 个，（0）未接来电、（1）已接来电、（2）已拨电话。

输入全部通话记录，每行一条记录。每条记录包含两个数字，第一个数代表记录类型，第二个数代表手机号码。

输出 分 3 列输出未接来电、已接来电、已拨电话。列之间用空格分割，后接电话在最先输出，不足 10 条用 0 占位。

样例输入

```
2    18270477699
1    10149800116
0    19906559817
1    16209018105
1    16804212234
2    19289130583
1    17982711123
0    10897630486
1    11860787674
0    15192777554
```

样例输出

15192777554 11860787674 19289130583
10897630486 17982711123 18270477699
19906559817 16804212234 0
0 16209018105 0
0 10149800116 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0

思路分析：

```
#include<bits/stdc++.h>
using namespace std;
#define ll long long
int main()
{
    1                    //未接
    2                    //已接
    3                    ;//已拨
    int kind;
    ll num;
    while(cin>>kind>>num)
    {
        if (kind == 0)
            4
        else if (kind == 1)
            5
        else
            yibo.push(num);
    }
    for (int i = 1; i <= 10; i++)
    {
        if (6                )
        {
            cout << weijie.top() << " ";
            7
        }
        else
            cout << "0 ";
```

```
        if (!yijie.empty())
        {
            cout << yijie.top() << " ";
            yijie.pop();
        }
        else
            cout << "0 ";

        if (!yibo.empty())
        {
            cout << yibo.top() << " ";
            yibo.pop();
        }
        else
            cout << "0 ";
        cout << endl;
    }
}
```

6 [7027] 细胞个数

一矩形阵列由数字 0 到 9 组成,数字 1 到 9 代表细胞,细胞的定义为沿细胞数字上下左右还是细胞数字则为同一细胞,求给定矩形阵列的细胞个数。

输入

第一行为两整数 m,n(m 行，n 列)（0<m,n<=100)

从第行开始是一个 m*n 的矩阵

输出

只有一行为矩阵中的细胞个数。

样例输入

4 10

0234500067

1034560500

2045600671

0000000089

样例输出

4

#include<iostream> #include<queue> using namespace std; const int gx[4]={0,0,-1,1}; const int gy[4]={1,-1,0,0}; int m,n,tot=0;	#include<iostream> #include<algorithm> using namespace std; int dx[4]={1,-1,0,0}; int dy[4]={0,0,-1,1}; int fs=4;

```
bool a[110][110];
void init();
void work(int,int);
int main()
{
    init();
    for(int i=1;i<=m;i++)
      for(int j=1;j<=n;j++)
        if(a[i][j])work(i,j);
    cout<<tot<<endl;
    return 0;
}
void work(int x,int y)
{
    queue<int> qx,qy;
    int x1,y1;
    tot++;
    qx.push(x);
    qy.push(y);
    a[x][y]=0;
    while(!qx.empty())
    {
        for(int i=0;i<4;i++)
        {
            x1=qx.front()+gx[i];
            y1=qy.front()+gy[i];
            if(x1>0&&x1<=m&&y1>0&&y1<=n&&a[x1][y1])
            {
                qx.push(x1);
                qy.push(y1);
                a[x1][y1]=false;
            }
        }
        qx.pop();
        qy.pop();
    }
}
void init()
{
    string s;
    cin>>m>>n;
    for(int i=1;i<=m;++i)
    {
        cin>>s;
```

```
int a[90][90];
void dfs(int x,int y)
{
  if(!a[x][y]||x<0||y<0)
    return ;
  a[x][y]=0;
  for(int i=0;i<fs;i++)
  dfs(x+dx[i],y+dy[i]);
}
int main()
{
  int n,m;
  int ans=0;
  cin>>n>>m;
  char s[90];
  for(int i=0;i<n;i++)
  {
    scanf("%s",s);
    for(int j=0;j<m;j++)
      a[i][j]=s[j]-'0';
  }
  for(int i=0;i<n;i++)
    for(int j=0;j<m;j++)
  {
    if(a[i][j])
    {
      ans++;
      dfs(i,j);
    }
  }
  cout<<ans;
  return 0;
}
```

```
            for(int j=0;j<n;++j)
                if(s[j]=='0')a[i][j+1]=0;
                else a[i][j+1]=1;
        }
}
```

课前练习

1.
```
#include <iostream>
using namespace std;
int main()
{
    bool f[10001]={0};
    int n, k, i, j, s=0;
    cin>>n>>k;
    for(i=2;i<=n;i++)
        if(f[i]==false)
            for(j=1;j<=n /i;j++)
                if (f[i*j]==false)
                {
                    s=s+1;
                    f[i*j]=true;
                    if(s==k)
                    {
                      cout<<i*j;
                      return 0;
                    }
                }
    return 0;
}
```
输入：
40 31
输出：_____

2.[2014](数字删除)下面程序的功能是将字符串中的数字字符删除后输出。请填空。
```
 #include <iostream>
using namespace std;
int delnum(char *s)
 {
    int i, j;
   j = 0;
    for(i = 0; s[i] != '\0'; i++)
      if(s[i] < '0' ___①___ s[i] > '9')
      {
           s[j] = s[i];
        ____②_____;
       }
   return ___③____;
}
const int SIZE = 30;
int main()
{
    char s[SIZE];
  int len, i;
    cin.getline(s, sizeof(s));
  len = delnum(s);
    for(i = 0; i < len; i++)
  cout << ___④____;
  cout << endl;
  return 0;
 }
```

习题

1 阅读程序
```
#include <iostream>
#include <string>
using namespace std;
```

```
int main()
{
    int i, s, m;
    int a[11];
    for (i = 1; i <= 10; i++)
        cin >> a[i];
    m = 0;
    s = 0;
    for (i = 1; i <= 10; i++)
    {
        if (a[i] < 0)
        {
            if (s > m)
                m = s;
            s = 0;
        }
        else
            s = s + a[i];
    }
    if (s > m)
        m = s;
    cout << "m=" << m << endl;
    return 0;
}
```

输入：

-5 13 -1 4 7 8 -1 -18 24 6

输出：_____

2. [2014](最大子矩阵和)给出 m 行 n 列的整数矩阵，求最大的子矩阵和(子矩阵不能为空)。

输入第一行包含两个整数 m 和 n，即矩阵的行数和列数。之后 m 行，每行 n 个整数，描述整个矩阵。程序最终输出最大的子矩阵和。(最后一空 4 分，其余 3 分，共 16 分)

比如在如下这个矩阵中： 4 4

0 -2 -7 0

9 2 -6 2

-4 1 -4 1

-1 8 0 -2

拥有最大和的子矩阵为：

9 2

-4 1

-1 8

其和为 15

3 3

-2 10 20

-1 100 -2

0 -2 -3
最大子矩阵和为 128
4 4
0 -2 -9 -9
-9 11 5 7
-4 -3 -7 -6
-1 7 7 5
最大子矩阵和为 26

```
#include <iostream>
using namespace std;
const int SIZE = 100;
int matrix[SIZE + 1][SIZE + 1];
int rowsum[SIZE + 1][SIZE + 1]; //rowsum[i][j]记录第 i 行前 j 个数的和
int m, n, i, j, first, last, area, ans;
int main()
 {
    cin >> m >> n;
    for(i = 1; i <= m; i++)
      for(j = 1; j <= n; j++)
          cin >> matrix[i][j];
  ans = matrix[ ① ];
    for(i = 1; i <= m; i ++)
      [ ② ]
  for(i = 1; i <= m; i++)
      for(j = 1; j <= n; j++)
               rowsum[i][j] = [ ③ ];
   for(first = 1; first <= n; first++)
      for(last = first; last <= n; last++)
      {
               [ ④ ];
               for(i = 1; i <= m; i++)
              {
                      area += [ ⑤ ];
                   if(area > ans)
                         ans = area;
                  if(area < 0)
                      area = 0;
               }
       }
    cout << ans << endl;
  return 0;
  }
```

3 [2882]假设在周末舞会上，男士们和女士们进入舞厅时，各自排成一队。跳舞开始时，依次从男队和女队的队头上各出一人配成舞伴。规定每个舞曲能有一对跳舞者。若两队初始人数不相同，则较长的那一队中

未配对者等待下一轮舞曲。现要求写一个程序，模拟上述舞伴配对问题。
输入第一行两队的人数;第二行舞曲的数目。
输出配对情况。

```
#include <iostream>
#include <1_____>        //头文件
#include <algorithm>
using namespace std;
int main()
{
   int n,m,k;
   while(~scanf("%d%d%d",&n,&m,&k))
   {
      queue<int>q1,q2;//定义一个队列
      for(int i=1;i<=n;i++)
          ___2_____//往栈里 添加男生
      for(int i=1;i<=m;i++)
          __3_____//往栈里 添加女生
      while(k--)
      {
         int x1=__4_____//取出队头配对
         int x2=___5_____
         q1.pop(),q2.pop();
         q1.push(x1),q2.push(x2);//配对后的男生女生重新排队
         printf("%d %d\n",x1,x2);
      }
   }
   return 0;
}
```

第 6 讲 队列应用及 BFS

广度优先搜索算法（又称宽度优先搜索）是最简便的图的搜索算法之一，这一算法也是很多重要的图的算法的原型。广度优先算法的核心思想是：从初始节点开始，应用算符生成第一层节点，检查目标节点是否在这些后继节点中，若没有，再用产生式规则将所有第一层的节点逐一扩展，得到第二层节点，并逐一检查第二层节点中是否包含目标节点。若没有，再用算符逐一扩展第二层的所有节点……，如此依次扩展，检查下去，直到发现目标节点为止。即

1.从图中的某一顶点 V0 开始，先访问 V0；

2.访问所有与 V0 相邻接的顶点 V1，V2，……，Vt；

3.依次访问与 V1，V2，……，Vt 相邻接的所有未曾访问过的顶点；

4.循此以往，直至所有的顶点都被访问过为止。

这种搜索的次序体现沿层次向横向扩展的趋势，所以称之为广度优先搜索。

广度优先搜索算法描述：

```
int bfs()
{
    初始化，初始状态存入队列；
    队列首指针 head=0; 尾指针 tail=1；
    while(head<tail)          //队列为空
     {
        指针 head 后移一位，指向待扩展结点；
        for (int i=1;i<=max;++i)   //max 为产生子结点的规则数
         {
           if (子结点符合条件)
             {
               tail 指针增 1，把新结点存入列尾；
               if (新结点与原已产生结点重复) 删去该结点（取消入队，tail 减 1）；
               else if (新结点是目标结点) 输出并退出；
             }
         }
     }
}
```

6.1[2806] 走出迷宫

当你站在一个迷宫里的时候，往往会被错综复杂的道路弄得失去方向感，如果你能得到迷宫地图，事情就会变得非常简单。假设你已经得到了一个 n*m 的迷宫的图纸，请你找出从起点到出口的最短路。

输入 第一行是两个整数 n 和 m(1≤n,m≤100)，表示迷宫的行数和列数。接下来 n 行，每行一个长为 m 的字符串，表示整个迷宫的布局。字符'.'表示空地，'#'表示墙，'S'表示起点,'T'表示出口。

输出从起点到出口最少需要走的步数。

样例输入	样例输出

3 3 S#T .#. ...	6

分析：

搜索从(xi,yi)到(xe,ye)路径的过程是：首先将(xi,yi)进队，在队列 que 不为空时循环：出队一次（由于不是循环队列，该出队元素仍在队列中），称该出队的方块为当前方块，que.front 为该方块在队列中的下标位置。如果当前方块是出口，则按入口到出口的次序输出该路径并结束。

否则，按顺时针方向找出当前方块的 4 个方位中可走的相邻方块（对应的迷宫 a 数组值为 0），将这些可走的相邻方块均插入到队列 que 中，其 pre 设置为本搜索路径中上一方块在 que 中的下标值，也就是当前方块的 qu.front 值，并将相邻方块对应的 a 数组元素值置为-1，以避免回过来重复搜索。如此队列为空，表示未找到出口，即不存在路径。

```
#include<iostream>
#include<queue>
using namespace std;
int n,m,vis[110][110];////用来标记有没有走过（有没有在队列中)
int dir[4][2]={{-1,0},{0,1},{1,0},{0,-1}};//定义方向 上右下左
char a[110][110];
struct node{
  int r,c,step;//row ,column,step
};
void bfs(int sr,int sc,int er,int ec){
  int k;
  memset(vis,0,sizeof(vis));
  queue<node> qe;//定义队列
  node q,t;//q 表示当前队头位置，t 表示下一个位置
  q.r=sr,q.c=sc,q.step=0;//把起点入队
  vis[sr][sc]=1;//记录已经走过
  qe.push(q);//入队 ……1 初始化，首元素入队
  while(!qe.empty()){
    q=qe.front();//取出队头元素
    qe.pop();
    if(er==q.r&&ec==q.c){//到出口了 ….. ……2 取出队头，判断是否是终点
      printf("%d",q.step);
      break;
    }
    for(k=0;k<4;k++){//继续搜索 4 个方向 ……3 没到终点，继续搜索
      t.r=q.r+dir[k][0],t.c=q.c+dir[k][1];
       //如果没有出迷宫、并且该位置可走、没走过
      if(0<=t.r&&t.r<n&&0<=t.c&&t.c<m&&vis[t.r][t.c]==0&&a[t.r][t.c]=='.'){
        vis[t.r][t.c]=1;//走到这个位置
        t.step=q.step+1;
```

```
            qe.push(t);.....4 把满足条件的点入队
          }
        }
      }
    }
}
int main(){
    int i,j,sr,sc,er,ec;
    scanf("%d%d",&n,&m);
    for(i=0;i<n;i++)scanf("%s",a[i]);//读入字符数组 a[i]表示每行的首地址
    for(i=0;i<n;i++)
       for(j=0;j<m;j++)//求出入口和出口的位置
          if(a[i][j]=='S')sr=i,sc=j;
          else if(a[i][j]=='T')er=i,ec=j,a[i][j]='.';
    bfs(sr,sc,er,ec);//把出口和入口作为参数
    return 0;
}
```

6.2［2804］走迷宫

一个迷宫由 R 行 C 列格子组成，有的格子里有障碍物，不能走；有的格子是空地，可以走。
给定一个迷宫，求从左上角走到右下角最少需要走多少步(数据保证一定能走到)。只能在水平方向或垂直方向走，不能斜着走。
输入
第一行是两个整数，R和C，代表迷宫的长和宽。（ 1≤ R，C ≤ 40)
接下来是R行，每行C个字符，代表整个迷宫。
空地格子用‘.’表示，有障碍物的格子用‘#’表示。
迷宫左上角和右下角都是‘.’。
输出
输出从左上角走到右下角至少要经过多少步（即至少要经过多少个空地格子）。计算步数要包括起点和终点。

样例输入	样例输出
5 5 ..### #.... #.#.# #.#.# #.#..	9

方法 1：广搜

```
#include<iostream>
#include<queue>
```

```
#include<cstring>
using namespace std;
struct node{
        int r,c,s;//row, column,step
};
int dir[4][2]={ 1                              };
int vis[45][45],n,m;
char a[45][45];

void bfs(int sr,int sc, int er, int ec){
        2
        node q,t;
        3
        vis[q.r][q.c]=1;
        4
        while(!qe.empty()){
                5
                6
                if( 7                  ){
                        cout<<q.s<<endl;
                        break;
                }
                for(int k=0;k<4;k++){
                        8
          if(a[t.r][t.c]=='.'&&vis[t.r][t.c]==0&&t.r>=0&&t.r<n&&t.c>=0&&t.c<m){
                                9
                                10
                    qe.push(t);
                        }
                }
        }
}
int main(){
        cin>>n>>m;
        memset(vis,0,sizeof(vis));
        for(int i=0;i<n;i++)scanf("%s",a[i]);
        bfs(0,0,n-1,m-1);
        return 0;
}
```

方法 2：深搜

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

```
char a[12][12];//定义迷宫
int dir[4][2]={{-1,0},{0,1},{1,0},{0,-1}};//定义方向
int vis[12][12];//标记有没有走过
int minstep=10000,xe,ye;//xe,ye 终点
void dfs(int x,int y,int step){
   if(x==xe&&y==ye){
      if(step<minstep)minstep=step;
   }
   else{
      for(int i=0;i<4;i++){
         int xx=x+dir[i][0];//下一个点
         int yy=y+dir[i][1];
         if(a[xx][yy]=='.'&&vis[xx][yy]==0){
            vis[xx][yy]=1;//记录走过
            dfs(xx,yy,step+1);
            vis[xx][yy]=0;//退格，下一次还可以走
         }
      }
   }
}
int main(){
   int m,n,i,j;
   cin>>m>>n;
   xe=m,ye=n;
   memset(a,'#',sizeof(a));
   memset(vis,0,sizeof(a));
   for(i=1;i<=m;i++)
      for(j=1;j<=n;j++)
         cin>>a[i][j];
   dfs(1,1,1);
   cout<<minstep<<endl;
   return 0;
}
```

方法 3：手写队列

```
#include<iostream>
using namespace std;
int sx[4]={1,-1,0,0};
int sy[4]={0,0,1,-1};
int n,m,head,tail,nowx,nowy,x,y,step;
struct hp{
   int x,y,step;
}queue[2005];
char s[50],a[50][50];
```

```
bool b[50][50];
int main(){
  scanf("%d%d\n",&n,&m);
  for (int i=1;i<=n;++i){
    gets(s);
    for (int j=1;j<=m;++j)
     a[i][j]=s[j-1];
  }
  head=0; tail=1;
  queue[tail].x=1,queue[tail].y=1,queue[tail].step=1;
  while (head<tail){
    ++head;
    nowx=queue[head].x, nowy=queue[head].y;
     step=queue[head].step;
    for (int i=0;i<4;++i){
      x=sx[i]+nowx,y=sy[i]+nowy;
      if (x>0&&x<=n&&y>0&&y<=m&&a[x][y]!='#'&&!b[x][y]){
        tail++;
        b[x][y]=true;
        queue[tail].x=x,queue[tail].y=y,queue[tail].step=step+1;
        if (x==n&&y==m) {printf("%d",step+1); return 0;}
      }
    }
  }
}
```

6.3［2805］抓住那头牛

农夫知道一头牛的位置，想要抓住它。农夫和牛都位于数轴上，农夫起始位于点 N(0≤N≤100000)，牛位于点 K(0≤K≤100000)。农夫有两种移动方式：

1、从 X 移动到 X-1 或 X+1，每次移动花费一分钟

2、从 X 移动到 2*X，每次移动花费一分钟

假设牛没有意识到农夫的行动，站在原地不动。农夫最少要花多少时间才能抓住牛？

输入两个整数，N 和 K。

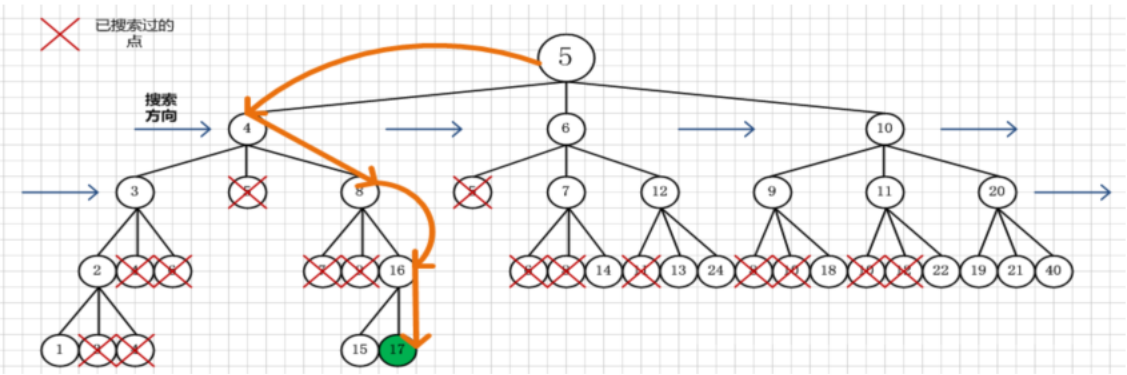
输出一个整数，农夫抓到牛所要花费的最小分钟数。

样例输入 5 17

样例输出 4

分析：

从 5 扩展出去，如下图所示



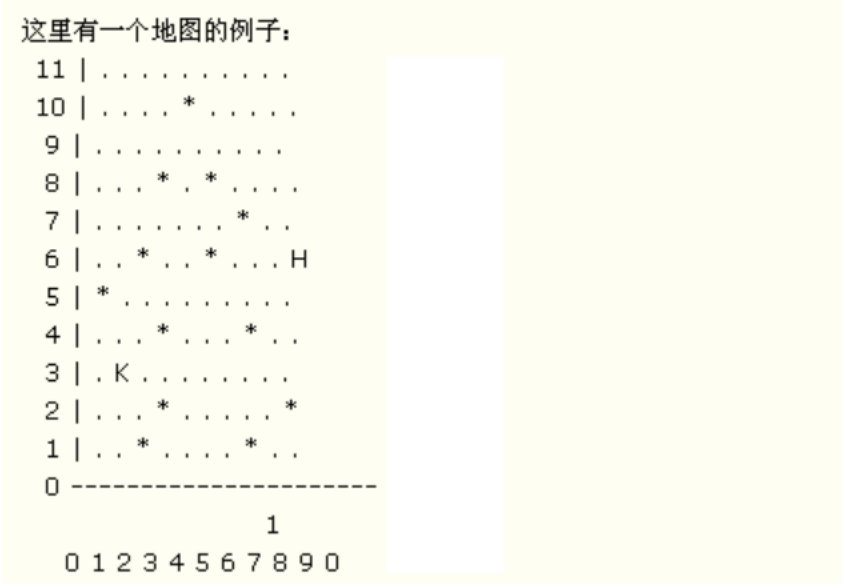

```
#include <iostream>
#include <queue>
#include <algorithm>
using namespace std;
#define max_n 100001

int n,k,cur,res;
int closed[max_n],d[3];//closed 数组储存最小分钟数，且标记已经访问
__1_____________________________________
void bfs(){
  //初始化 closed 数组
  memset(closed,-1,sizeof(closed));
  __2_____________________;//将起点加入 open 队列
  closed[n] = 0;//
  while (__3_____________________){
    //若当前值为终点，则退出循环
    cur = __4_____________
    open.pop();
    if (cur == k) break;
    //三种选择
    d[0] = cur - 1;
    d[1] = __5_____________
    d[2] = __6_____________

    //对三种选择 bfs
    for (int i = 0;i < 3;i++){
      //注意数据不能越界且不能被访问
      if (7_____________________________){
        open.push(d[i]);
        __8_____________
      }
    }
  }
```

```
    res = closed[k];
    cout << res << endl;
}
 int main() {
    cin >> n >> k;
    bfs();
    return 0;
}
```

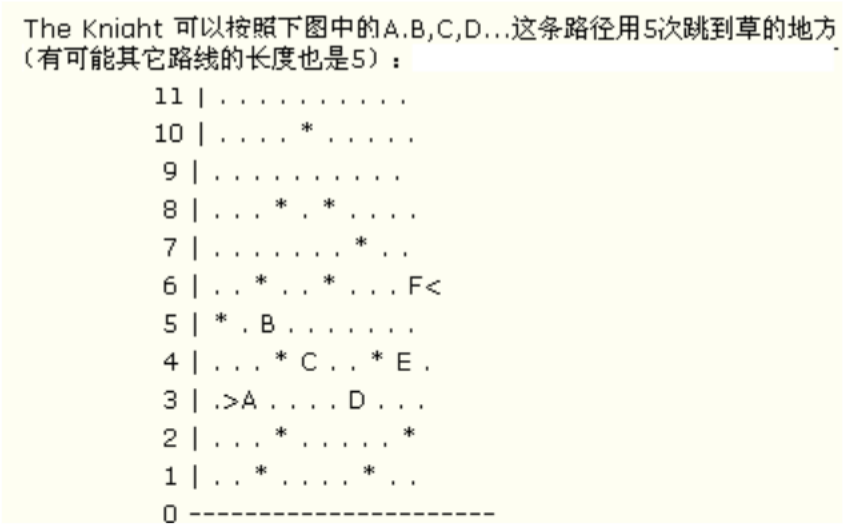
6.4［1190］武士风度的牛

这头神奇的牛像其它牛一样喜欢吃草，给你一张地图，上面标注了 The Knight 的开始位置，树、灌木、石头以及其它障碍的位置，除此之外还有一捆草。现在你的任务是，确定 The Knight 要想吃到草，至少需要跳多少次。The Knight 的位置用'K'来标记，障碍的位置用'*'来标记，草的位置用'H'来标记。这里有一个地图的例

这里有一个地图的例子：



The Knight 可以按照下图中的A,B,C,D...这条路径用5次跳到草的地方（有可能其它路线的长度也是5）：



输入
第一行： 两个数，表示农场的列数(< =150)和行数(< =150) 第二行..结尾: 如题目描述的图。
输出一个数，表示跳跃的最小次数。
样例输入
10 11
....
....*.....
....
...*.*....
.......*..
..*..*...H
*.........
...*...*..
.K........
...*.....*
..*....*..
样例输出
5

```
#include<bits/stdc++.h>
#define N 155
using namespace std;
const int dir[8][2]={{2,1},{-2,1},{2,-1},{-2,-1},{-1,2},{1,2},{-1,-2},{1,-2}};struct node{
   int x,y;
};
int n,m,sx,sy,ex,ey;
char Map[N][N];
int dis[N][N];
inline bool check(int x,int y)//判断当前状态是否合法
{
   if(x<1||x>n||y<1||y>m||dis[x][y]||Map[x][y]=='*')
      return false;
   return true;
}
inline int BFS()
{
   queue<node> q;
   q.push(node{sx,sy});
   while(!q.empty()){
      node now=q.front();
      q.pop();
      for(int i=0;i<8;i++){
         int dx=now.x+dir[i][0],dy=now.y+dir[i][1];
         if(!check(dx,dy))
```

```
            continue;
        if(dx==ex&&dy==ey)//到达终点
            return dis[now.x][now.y]+1;
        q.push(node{dx,dy});
        dis[dx][dy]=dis[now.x][now.y]+1;//距离加一
      }
    }
}
int main()
{
   memset(dis,0,sizeof(dis));
   scanf("%d %d",&m,&n);
   for(int i=1;i<=n;i++)
     for(int j=1;j<=m;j++){
        cin>>Map[i][j];
        if(Map[i][j]=='K'){//记录起点
           sx=i,sy=j;
           dis[i][j]=0;
        }
        if(Map[i][j]=='H')//记录终点
           ex=i,ey=j;
     }
   printf("%d\n",BFS());
}
```

课前练习

1. 阅读程序

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    int i, j, k, n;
    bool a[101];
    cin >> n;
    for (i = 1; i <= n; i++)
        a[i] = true;
    for (i = 1; i <= n; i++)
    {
        j = i;
        while (j <= n)
```

```
        {
            a[j] = !a[j];
            j = j + i;
        }
    }
    for (i = 1; i <= n; i++)
    {
        if (a[i] == true)
            cout << "0" << " ";
        else
            cout << "1" << " ";
    }
    return 0;
}
```

输入：8
输出:_____

2. [2012]（排列数）输入两个正整数 n，m（1<n<20,1<m<n），在 1~n 中任取 m 个数，按字典序从小到大输出所有这样的排列。例如：
输入：3 2
输出：
1 2
1 3
2 1
2 3
3 1
3 2

```
#include <iostream>
#include <cstring>
using namespace std;
const int SIZE =25;
bool used[SIZE];
int data[SIZE];
int n,m,i,j,k;
bool flag;
int main()
{
    cin>>n>>m;
    memset(used,false,sizeof(used));
    for(i=1;i<=m;i++)
    {
        data[i]=i;
```

```
            used[i]=true;
        }
        flag=true;
        while(flag)
        {
            for(i=1;i<=m-1;i++)  cout<<data[i]<<" ";
            cout<<data[m]<<endl;
            flag=  ____①____  ;
            for(i=m;i>=1;i--)
            {
                ____②____;
                for(j=data[i]+1;j<=n;j++)
                    if(!used[j])
                    {
                        used[j]=true;
                        data[i]=____③____;
                        flag=true;
                        break;
                    }
                if(flag)
                {
                    for(k=i+1;k<=m;k++)
                        for(j=1;j<=____④____ ;j++)
                        if(!used[j])
                        {
                            data[k]=j;
                            used[j]=true;
                            break;
                        }
                    ____⑤____ ;
                }
            }
        }
        return 0;
}
```

习题

1．[2012]（坐标统计）输入 n 个整点在平面上的坐标。对于每个点，可以控制所有位于它左下方的点（即 x、y 坐标都比它小），它可以控制的点的数目称为“战斗力”。依次输出每个点的战斗力，最后输出战斗力最高的点的编号（如果若干个点的战斗力并列最高，输出其中最大的编号）。

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

```
const  int  SIZE  =100;
int  x[SIZE],y[SIZE],f[SIZE];
int  n,i,j,max_f,ans;
int  main()
{
      cin>>n;
      for(i=1;i<=n;i++)  cin>>x[i]>>y[i];
      max_f=0;
      for(i=1;i<=n;i++)
      {
            f[i]=___①___;
            for(j=1;j<=n;j++)
            {
                  if(x[j]<x[i]  &&___②___ )
                  _____③_____;
            }
            if( _____④_____)
            {
                  max_f=f[i];
                  ____⑤____;
            }
      }
      for(i=1;i<=n;i++)  cout<<f[i]<<endl;
      cout<<ans<<endl;
      return  0;
}
```

2.[2013]

```
#include <iostream>
using namespace std;
int main()
{
      const int SIZE = 100;
      int n, f, i, left, right, middle, a[SIZE];
      cin>>n>>f;
      for (i = 1; i <= n; i++)
      cin>>a[i]; left = 1;
      right = n;
      do {
            middle = (left + right) / 2;
```

```
            if (f <= a[middle])
                right = middle;
            else
                left = middle + 1;
        } while (left < right);
        cout<<left<<endl;
        return 0;
}
```
输入：

12 17

2 4 6 9 11 15 17 18 19 20 21 25

输出：_____

3. 抓住那头牛

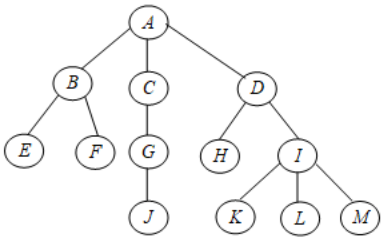
```
void bfs(){
  //初始化 closed 数组
  memset(closed,-1,sizeof(closed));
  __2_____________________;//将起点加入 open 队列
  closed[n] = 0;//
  while (__3_____________________){
    //若当前值为终点，则退出循环
    cur = __4_____________
    open.pop();
    if (cur == k) break;
    //三种选择
    d[0] = cur - 1;
    d[1] = __5_____________
    d[2] = __6_____________

    //对三种选择 bfs
    for (int i = 0;i < 3;i++){
      //注意数据不能越界且不能被访问
      if (7_____________________________________){
        open.push(d[i]);
        __8_____________
      }
    }
  }

  res = closed[k];
  cout << res << endl;
}
```

第 7 讲 二叉树的概述

树是由 n（n≥0）个结点组成的有限集合（记为 T）。如果 n=0，它是一棵空树，这是树的特例；如果 n>0，这 n 个结点中存在（有仅存在）一个结点作为树的根结点（root），其余结点可分为 m（m≥0）个互不相交的有限集 T1、T2、…、Tm，其中每个子集本身又是一棵符合本定义的树，称为根结点的子树。
树是一种非线性数据结构，具有以下特点：

它的每一结点可以有零个或多个后继结点，但有且只有一个前趋结点（根结点除外）；这些数据结点按分支关系组织起来，清晰地反映了数据元素之间的层次关系。可以看出，数据元素之间存在的关系是一对多的关系。

树的逻辑结构表示方法



（a）树形表示法

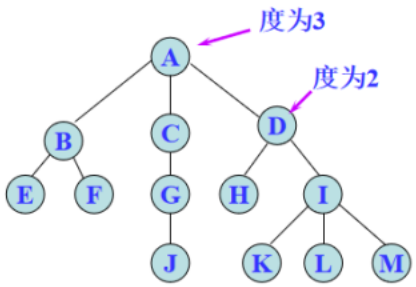
$A(B(E,F),C(G(J)),D(H,I(K,L,M)))$

（d）括号表示法

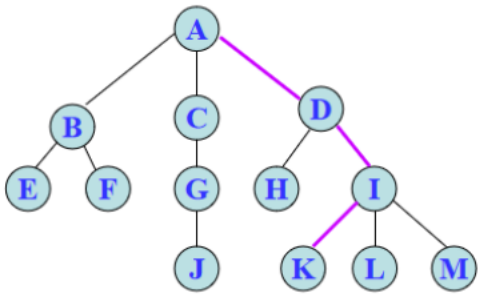
7.1 树的基本术语

1. **结点的度与树的度**：树中某个结点的子树的个数称为该结点的度。树中各结点的度的最大值称为树的度，通常将度为 m 的树称为 m 次树。

2. **分支结点与叶结点**：度不为零的结点称为非终端结点,又叫分支结点。度为零的结点称为终端结点或叶结点。在分支结点中，每个结点的分支数就是该结点的度。如对于度为 1 的结点,其分支数为 1，被称为单分支结点；对于度为 2 的结点，其分支数为 2，被称为双分支结点，其余类推。



3. **路径与路径长度**：对于任意两个结点 di 和 dj，若树中存在一个结点序列 di,di1,di2,…,din,dj，使得序列中除 di 外的任一结点都是其在序列中的前一个结点的后继，则称该结点序列为由 di 到 dj 的一条路径，用路径所通过的结点序列(di,di1,di2,…,dj)表示这条路径。



A到K的路径为A,D,I,K，
其长度为3

路径长度等于路径所通过的结点数目减 1（即路径上分支数目）。

4. **孩子结点、双亲结点和兄弟结点**：在一棵树中，每个结点的后继，被称作该结点的孩子结点（或子女结点）。相应地，该结点被称作孩子结点的双亲结点（或父母结点）。

具有同一双亲的孩子结点互为兄弟结点。进一步推广这些关系，可以把每个结点的所有子树中的结点称为该结点的子孙结点。

从树根结点到达该结点的路径上经过的所有结点被称作该结点的祖先结点。

5. **结点的层次和树的高度**：树中的每个结点都处在一定的层次上。结点的层次从树根开始定义，根结点为第 1 层，它的孩子结点为第 2 层,以此类推,一个结点所在的层次为其双亲结点所在的层次加 1。树中结点的最大层次称为树的高度（或树的深度）。

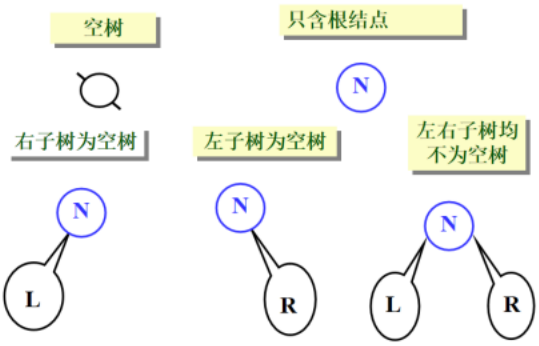
6. **有序树和无序树**：若树中各结点的子树是按照一定的次序从左向右安排的，且相对次序是不能随意变换的，则称为有序树，否则称为无序树。

7. **森林**：n（n＞0）个互不相交的树的集合称为森林。森林的概念与树的概念十分相近，因为只要把树的根结点删去就成了森林。反之，只要给 n 棵独立的树加上一个结点，并把这 n 棵树作为该结点的子树，则森林就变成了树。

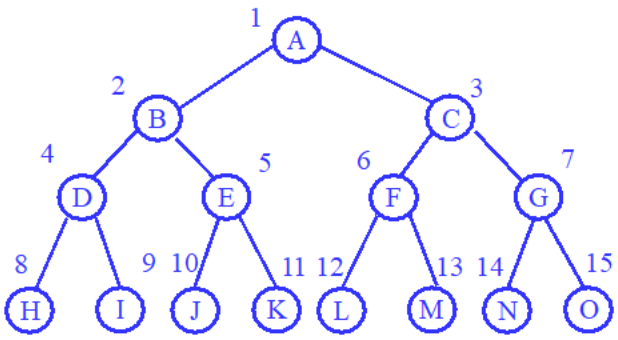
7.2 二叉树的概念

7.2.1 概念

二叉树是有限的结点集合。这个集合或者是空。 或者由一个根结点和两棵互不相交的称为左子树和右子树的二叉树组成。二叉树有五种基本形态：

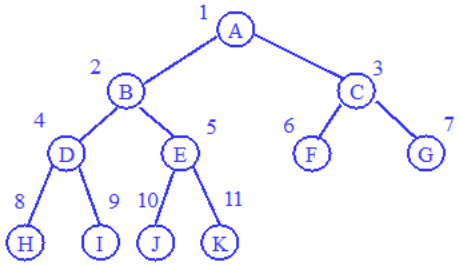


在一棵二叉树中，如果所有分支结点都有左孩子结点和右孩子结点，并且叶结点都集中在二叉树的最下一层，这样的二叉树称为**满二叉树**。下图所示就是一棵满二叉树。可以对满二叉树的结点进行连续编号，约定编号从树根为 1 开始，按照层数从小到大、同一层从左到右的次序进行。图中每个结点外边的数字为对该结点的编号。



满二叉树

若二叉树中最多只有最下面两层的结点的度数可以小于 2，并且最下面一层的叶结点都依次排列在该层最左边的位置上，则这样的二叉树称为完全二叉树。如下图所示为一棵完全二叉树。同样可以对完全二叉树中每个结点进行连续编号，编号的方法同满二叉树相同。图中每个结点外边的数字为对该结点的编号。



完全二叉树

若二叉树中最多只有最下面两层的结点的度数可以小于 2，并且最下面一层的叶结点都依次排列在该层最左边的位置上，则这样的二叉树称为**完全二叉树**。

如下图所示为一棵完全二叉树。同样可以对完全二叉树中每个结点进行连续编号，编号的方法同满二叉树相同。图中每个结点外边的数字为对该结点的编号。若二叉树中最多只有最下面两层的结点的度数可以小于 2，并且最下面一层的叶结点都依次排列在该层最左边的位置上，则这样的二叉树称为完全二叉树。

7.2.2 性质

性质 1 **非空二叉树上叶结点数等于双分支结点数加 1。**

证明：设二叉树上叶结点数为 n0，单分支结点数为 n1，双分支结点数为 n2，则总结点数 n=n0+n1+n2。在一棵二叉树中，所有结点的分支数（即度数）应等于单分支结点数加上双分支结点数的 2 倍，即总的分支数=n1+2n2。

由于二叉树中除根结点以外，每个结点都有唯一的一个分支指向它，因此二叉树中有：总的分支数=总结点数-1。由上述三个等式可得：n1+2n2=n0+n1+n2-1

即：n0=n2+1

性质 2 **非空二叉树上第 i 层上至多有 2^{i-1} 个结点**，这里应有 i≥1。

性质 3 **高度为 h 的二叉树至多有 2^h-1 个结点**（h≥1）。

性质 4 对完全二叉树中编号为 i 的结点（1≤i≤n，n≥1，n 为结点数）有：

（1）若 i≤ n/2 ，即 2i≤n，则编号为 i 的结点为分支结点，否则为叶子结点。

（2）若 n 为奇数，则每个分支结点都既有左孩子结点，也有右孩子结点；若 n 为偶数，则编号最大的分支结点只有左孩子结点，没有右孩子结点，其余分支结点都有左、右孩子结点。

（3）若编号为 i 的结点有左孩子结点，则左孩子结点的编号为 2i；若编号为 i 的结点有右孩子结点，则右孩子结点的编号为(2i+1)。

（4）除树根结点外,若一个结点的编号为 i，则它的双亲结点的编号为 i/2 ，也就是说，当 i 为偶数时，其双亲结点的编号为 i/2,它是双亲结点的左孩子结点，当 i 为奇数时，其双亲结点的编号为(i-1)/2，它是双亲结点的右孩子结点。

7.3 树的基本操作

树的运算主要分为三大类：

第一类，寻找满足某种特定关系的结点，如寻找当前结点的双亲结点等；

第二类，插入或删除某个结点，如在树的当前结点上插入一个新结点或删除当前结点的第 i 个孩子结点等；

第三类，遍历树中每个结点。

7.3.1 树的遍历

树的遍历运算是指按某种方式访问树中的每一个结点且每一个结点只被访问一次。

有以下三种遍历方法：

先根遍历：若树不空，则先访问根结点，然后依次先根遍历各棵子树。

后根遍历： 若树不空，则先依次后根遍历各棵子树，然后访问根结点。

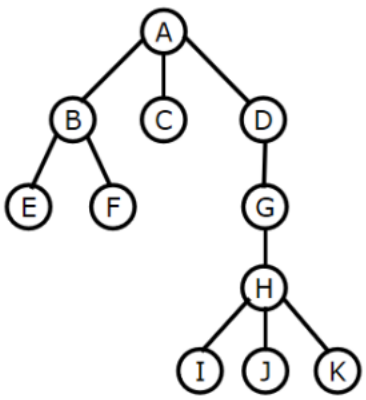
层次遍历：若树不空，则自上而下自左至右访问树中每个结点。

如图所示，遍历的顺序为：

先根遍历的顶点访问次序：A B E F C D G H I J K

后根遍历的顶点访问次序：E F B C I J K H G D A

层次遍历的顶点访问次序：A B C D E F G H I J K

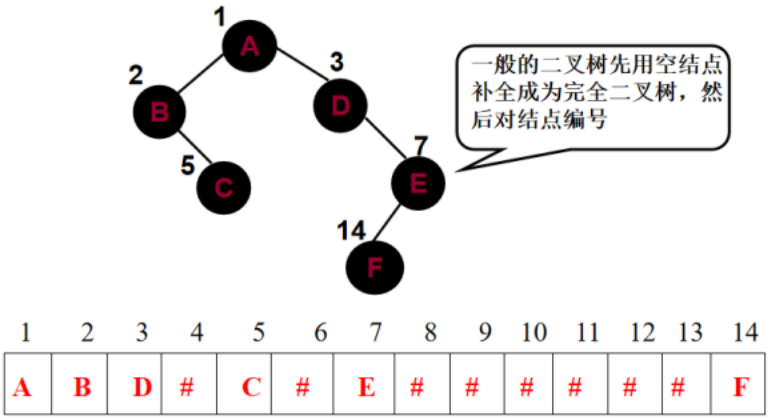


7.3.2 树的建立

1. 二叉树的顺序存储结构

二叉树的顺序存储结构中结点的存放次序是：对该树中每个结点进行编号，其编号从小到大的顺序就是结点存放在连续存储单元的先后次序。

若把二叉树存储到一维数组中,则该编号就是下标值加 1（注意 C/C++语言中数组的起始下标为 0）。树中各结点的编号与等高度的完全二叉树中对应位置上结点的编号相同。

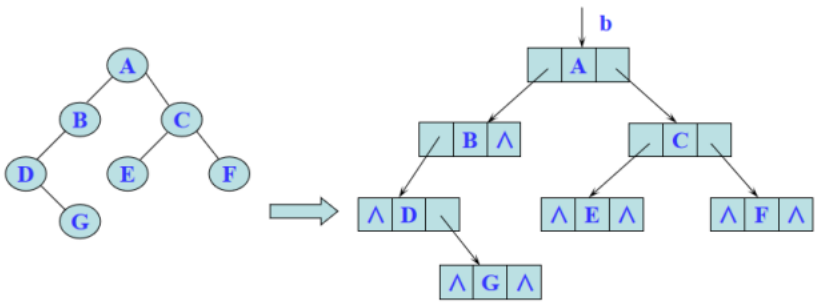


2. 二叉树的链式存储结构

在二叉树的链接存储中，结点的结构如下:

```
struct BTNode       //二叉链中结点类型
{    int data;              //数据元素
     BTNode *lchild;        //指向左孩子结点
     BTNode *rchild;        //指向右孩子结点
};
```

其中,data 表示值域,用于存储对应的数据元素，lchild 和 rchild 分别表示左指针域和右指针域，用于分别存储左孩子结点和右孩子结点（即左、右子树的根结点）的存储位置。



二叉树及其链式存储结构：二叉链

7.3.3 树的遍历

二叉树遍历是指按照一定次序访问二叉树中所有结点，并且每个结点仅被访问一次的过程。通过遍历

得到二叉树中某种结点的线性序列，即将非线性结构线性化，这里的“访问”的含义可以很多，如输出结点值或对结点值实施某种运算等。二叉树的遍历可以分为先序遍历、中序遍历、后续遍历、层次遍历。

二叉树遍历是树的最基本运算，是二叉树中所有其他运算的基础。

先序遍历二叉树的过程是：

① 访问根结点；

② 先序遍历左子树；

③ 先序遍历右子树。

实现的过程如下：

```
struct node{
    int left,right,id,father;//左右孩子、编号、父节点
    char ch;//数据域
}t[100];
void inorder(int root){
    cout<<t[root].ch;
    if(t[root].left)inorder(t[root].left);
    if(t[root].right)inorder(t[root].right);
}
```

7.4 案例分析

1 [1530] 完全二叉树的高度

已知完全二叉树的结点数，求其高度。

输入文件中包含多个测试数据。每个测试数据占 1 行，为一个整数 n(1≤n≤100)，表示完全二叉树的结点数。

输入文件中最后一行为 0，表示测试数据结束。

对输入文件中的每个测试数据，输出完全二叉树的高度。

样例输入	样例输出
10 20 0	4 5

分析：根据二叉树的性质可知，**二叉树的高度 h=log2(n)+1.**

```
#include<iostream>
using namespace std;
int main(){
  int n,h;
  while(cin>>n&&n!=0){
    int h=0;
    while(n!=0){
      n/=2;
      h++;
    }
```

```
    cout<<h<<endl;// h=log2(n)+1;
  }
  return 0;
}
```

2 [1909] 二叉树 1

满二叉树：第 1 层有 2^0=1 个节点（即根节点）、第 2 层有 2^1=2 个节点、...、第 i+1 层有 2^i 个节点的二叉树。也就是说，在完全二叉树中，最后一层的节点都是叶节点，其他层次的节点都是非叶节点、且都有两个子节点。
完全二叉树：如果有一棵具有 n 个节点、高度为 k 的二叉树，它的每一个节点都与高度为 k 的完全二叉树中编号为 1～n 的节点一一对应，则称为完全二叉。

在本题中，给定完全二叉树的节点树 n，要求输出完全二叉树的高度 k、叶子节点个数 n1 和非叶节点个数 n2。

输入文件中包含多个测试数据。每个测试数据占一行，为一个自然数 n，n≤1000。输入文件最后一行为 0，表示输入结束。对输入文件中的每个测试数据，计算 k、n1 和 n2 并输出。

样例输入	样例输出
12 0	4 6 6

分析：因为 n0=n2+1，同时**对于完全二叉树来说，度为 1 的点的个数 n1=1 或 0，所以 n0=(n+1)/2,**

```
#include<bits/stdc++.h>
#include<iostream>
using namespace std;
int main(){
  int n,n1,n2;
  while(scanf("%d",&n)!=EOF){
    int k=0;
    if(n==0) return 0;
    for(int i=n;i>=1;i=i/2){
      k++;
    }
    n1=(n+1)/2;
    n2=n-n1;
    printf("%d %d %d\n",k,n1,n2);//cout<<k;
  }
  return 0;
}
```

3 [2886] 找树根和孩子

给定一棵树，输出树的根 root，孩子最多的结点 max 以及他的孩子。
输入

第一行：n（结点个数≤100），m（边数≤200）。
以下 m 行：每行两个结点 x 和 y，表示 **y 是 x 的孩子**(x,y≤1000)。
输出
第一行：树根：root；第二行：孩子最多的结点 max；第三行：max 的孩子。

样例输入	样例输出
8 7 4 1 4 2 1 3 1 5 2 6 2 7 2 8	4 2 6 7 8

思考：题目给出节点间的父子关系，这样就可以找到根节点。
1.
1. 如何表示父子关系？_____
2. 具有什么特点的节点是根节点？_____
3. 孩子最多的结点具有什么特点？_____

```
#include<iostream>
using namespace std;
int main(){
        int parent[101]={0};
        int m,n,x,y,maxnode,sum=0,root;
        int i,j,k;
        cin>>n>>m;
        for(i=1;i<=m;i++){
                cin>>x>>y;
                __1_____________________//记录父亲节点

        }
        for(i=1;i<=n;i++)
                if(__2_____________________){
                        root=i;//寻找父亲节点
                        break;
                }
        int max=0;
        for(i=1;i<=n;i++){//寻找最大
                sum=0;
                for(j=1;j<=n;j++)
                        if(__3_____________)sum++;
                if(_4_____________){//
                        max=sum;
                        maxnode=i;
```

```
                }

        }
        cout<<root<<endl;
        cout<<maxnode<<endl;
        for(i=1;i<=n;i++){//寻找孩子
                if(parent[i]==maxnode)
                        cout<<i<<" ";
        }
        return 0;
}
```

4 [2915] 查找二叉树(tree_a)

已知一棵二叉树用邻接表结构存储，中序查找二叉树中值为 x 的结点，并指出是第几个结点(通过中序遍历访问的 第几个节点)。例：如图二叉树的数据文件的数据格式如下:

输入

第一行 n 为二叉树的结点个树，n≤100；第二行 x 表示要查找的结点的值；以下第一列数据是各结点的值，第二列数据是左儿子结点编号，第三列数据是右儿子结点编号。

输出

一个数即查找的结点编号。（中序遍历的顺序为 29 12 8 15 23 5 10）

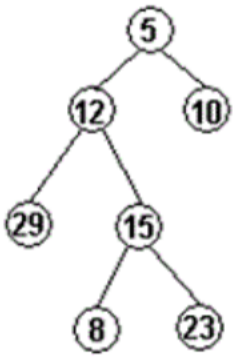
样例输入

```
7
15
5 2 3
12 4 5
10 0 0
29 0 0
15 6 7
8 0 0
23 0 0
```

样例输出

```
4
```

```
7
15
5 2 3
12 4 5
10 0 0
29 0 0
15 6 7
8 0 0
23 0 0
```




```
#include <stdio.h>
#include <stdlib.h>
struct node
{
  int d, left, right;
}q[110];
int n, ans = 0, a;
void in_order(int k)
{
  if (__1_____)in_order(q[k].left);
```

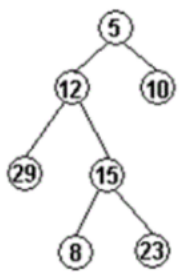
```
        ans++;
    if (q[k].d == a)
    {
        printf("%d", ans);
        exit(0);
    }
    if ( __2_____)
        in_order(q[k].right);
}
int main()
{
    scanf("%d%d", &n, &a);
    for (int i = 1; i <= n; i++)
        scanf("%d%d%d", &q[i].d, &q[i].left, &q[i].right);
    in_order(1);
    return 0;
}
```

5 [7301] 二叉树求节点和

已知一棵二叉树用邻接表结构存储，求出所有节点的权值和。
例：如图二叉树的数据文件的数据格式如下:

输入
第一行 n 为二叉树的结点个树，n≤100；以下第一列数据是各结点的值，第二列数据是左儿子结点编号，第三列数据是右儿子结点编号。
输出
权值和。
样例输入

```
7
5 2 3
12 4 5
10 0 0
29 0 0
15 6 7
8 0 0
23 0 0
```



样例输出

```
102
```

```
#include<bits/stdc++.h>
using namespace std;
struct node{
```

```
        int data;
        __1_____________________
}q[105];
int sum=0,x;
int sumbt(int root){
      if(_2_____________________)
              return q[root].data;
    else
        return _3_____________________________________________
}
int main(){
        int n;
        cin>>n;
        for(int i=1;i<=n;i++)
                cin>>q[i].data>>q[i].left>>q[i].right;
        cout<<sumbt(1);
        return 0;
}
```

写法 2：

```
#include<bits/stdc++.h>
using namespace std;
struct node{
  int data;
  int left,right;
}q[105];
int sum=0;
void sumbt(int root){
  sum+=q[root].data;
  if(q[root].left)   sumbt(q[root].left);  //访问左子树
  if(q[root].right)   sumbt(q[root].right);//访问右子树
}
int main(){
  int n;
  cin>>n;
  for(int i=1;i<=n;i++)
    cin>>q[i].data>>q[i].left>>q[i].right;
  sumbt(1);
  cout<<sum;
  return 0;
}
```

课前练习

1.[2013]

```
#include <iostream>
using namespace std;
int main()
{
    const int SIZE = 100;
    int height[SIZE], num[SIZE], n, ans;
    cin>>n;
    for (int i = 0; i < n; i++)
    {
        cin>>height[i]; num[i] = 1;
        for (int j = 0; j < i; j++)
        {
            if ((height[j] < height[i]) && (num[j] >= num[i]))
            num[i] = num[j]+1;
        }
    }
    ans = 0;
    for (int i = 0; i < n; i++)
    {
      if (num[i] > ans) ans = num[i];
    }
    cout<<ans<<endl;
}
```

输入：

6

2 5 3 11 12 4　　输出：_____

2．[2011]（**子矩阵**）给输入一个 n1*m1 的矩阵 a，和 n2*m2 的矩阵 b，问 a 中是否存在子矩阵和 b 相等。若存在，输出所有子矩阵左上角的坐标；若不存在输出“There isno answer”。

```
#include<iostream>
using namespace std;
const int SIZE = 50;
int n1,m1,n2,m2,a[SIZE][SIZE],b[SIZE][SIZE];
int main()
{
    int i,j,k1,k2;
    bool good ,haveAns;
    cin>>n1>>m1;
    for(i=1;i<=n1;i++)
```

```
        for(j=1;j<=m1;j++)
            cin>>a[i][j];
    cin>>n2>>m2;
    for(i=1;i<=n2;i++)
        for(j=1;j<=m2;j++)
            ____①____;
    haveAns=false;
    for(i=1;i<=n1-n2+1;i++)
        for(j=1;j<=____②____;j++){
            ____③____;
            for(k1=1;k1<=n2;k1++)
                for(k2=1;k2<=____④____;k2++){
                    if(a[i+k1-1][j+k2-1]!=b[k1][k2])
                        good=false;
                }
            if(good){
                cout<<i<<' '<<j<<endl;
                ____⑤____;
            }
        }
    if(!haveAns)
        cout<<"There is no answer"<<endl;
    return 0;
}
```

习题

1．[2007]（求字符串的逆序）下面的程序的功能是输入若干行字符串，每输入一行，就按逆序输出该行，最后键入-1 终止程序。请将程序补充完整。

```
#include <iostream.h>
#include <string.h>
int maxline=200,kz;
int reverse(char s[])
{int i,j,t;
for(i=0,j=strlen(s)-1; i<j;____①____ , ____②____ )
{t=s[i]; s[i]=s[j]; s[j]=t;}
return 0;
}
void main()
{ char line[100];
cout<<"continue? -1 for end."<<endl;
```

```
cin>>kz;
while( ③                    )
{ cin>>line;
   ④                              ;
cout<<line<<endl;
cout<<"continue? -1 for end."<<endl;
cin>>kz;
}
}
```

2 【味子圣诞礼物】

圣诞节到了，妈妈在味子的圣诞树上挂满了礼物（一共有 n 个），妈妈挂礼物的方法很独特，那就是每个礼物都是挂在某个礼物的下面（最上面的礼物除外），这样，所有的礼物挂在圣诞树上就形成了一个树型结构（如下图所示）。

味子当然很高兴，可是高兴之余，味子又担心了，因为妈妈告诉她“我给每个礼物都编了号，你要从上往下逐层往下数，最后要告诉妈妈，你按照这个顺序数礼物，得到的礼物的编号序列是什么。如果你答对了，今年春节我会在圣诞树上给你挂更多的礼物，否则，春节的礼物就会比圣诞节礼物要少。”

可是味子很聪明，她略加思索就说出了所有礼物从上下排序的顺序是如下序列：

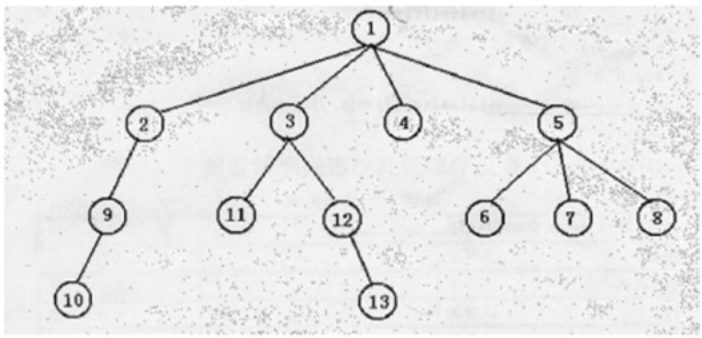
1 2 3 4 5 9 11 12 6 7 8 10 13

但味子不满足于现在的成绩，她想到了用程序来解决这个问题的所有情况。就是告诉程序一共有多少礼物，以及礼物挂在圣诞树上的顺序，程序应能输出从上往下逐层（每层中的礼物从左到右排列）排列的结果。

味子的程序首先能读入一个整数 n，表示所有圣诞礼物的总数，然后能读入 n 行（该 n 行数据表示礼物挂在圣诞树上的位置顺序），每行的第一个编号表示某个礼物的本身编号 i，该行后面的编号依次表示挂在礼物 i 下面的其他礼物，（按从左到右的顺序给出）。比如，输入样例第二行中的“1 2 3 4 5 0”，2、3、4、5 分别表示挂在礼物 1 下面的其他礼物编 号分别是 2、3、4、5，最后的 0 表示挂在礼物 1 下面没有其他礼物了。味子编写了以下程序来达到上面的目的，该程序运行后能用一行输出数据来表示礼物从上入下逐层排列的顺序序列，每个编号之间用一个空格分隔，请帮助味子完成程序。

输入样例：

13
1 2 3 4 5 0
2 9 0
3 11 12 0
4 0
5 6 7 8 0
6 0
7 0
8 0
9 10 0
10 0
11 0
12 13 0
13 0



输出样例：

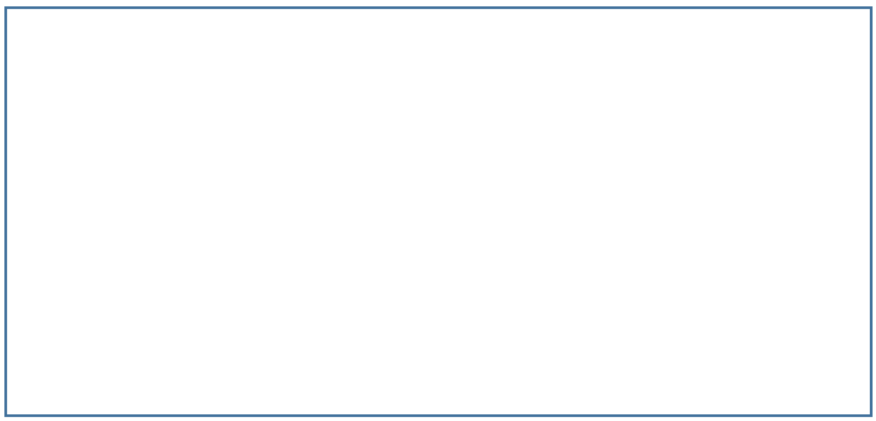
1 2 3 4 5 9 11 12 6 7 8 10 13

完善下列程序

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
int main()
{
    int first, tail, i, j, n;
    int a[21][21];
    int list[1001];
    for (i = 1; i <= 20; i++)
        for (j = 1; j <= 20; j++)
            a[i][j] = 0;
    ①          ;
    for (i = 1; i <= n; i++)
    {
        j = 1;
        cin >> a[i][j];
        while (a[i][j] != 0)
        {
            ②          ;
            cin >> a[i][j];
        }
    }
    first = 1;
    tail = 1;
    list[1] = a[1][1];
    while (first <= tail)
    {
        ③          ;
        i = list[first];
        while (a[i][j] != 0)
        {
            ④          ;
            list[tail] = a[i][j];
            j = j + 1;
        }
        first = first + 1;
    }
    for (i = 1; i <= n; i++)
        cout << ⑤          << " ";
    return 0;
}
```

3 二叉树球高度

```
#include<bits/stdc++.h>
using namespace std;
struct node{
   int data;
   int l,r;
}nd[100];
const int noChild=0;
int getD(int root){
```



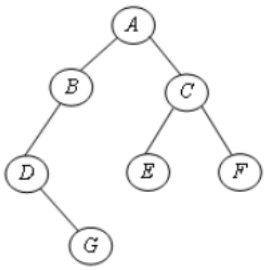
```
}
int main(){
   int n;
   cin>>n;
   for(int i=1;i<=n;i++)
      cin>>nd[i].data>>nd[i].l>>nd[i].r;
   cout<<getD(1);

   return 0;
}
```

第 8 讲 树的遍历

二叉树遍历是指按照一定次序访问二叉树中所有结点，并且每个结点仅被访问一次的过程。通过遍历得到二叉树中某种结点的线性序列，即将非线性结构线性化，这里的“访问”的含义可以很多，如输出结点值或对结点值实施某种运算等。

二叉树遍历是最基本的运算，是二叉树中所有其他运算的基础。

8.1 先序遍历



先序遍历二叉树的过程是：

① 访问根结点；

② 先序遍历左子树；

③ 先序遍历右子树。

例如，下图的二叉树的先序序列为：ABDGCEF。显然，在一棵二叉树的先序序列中，第一个元素即为根结点对应的结点值。

```
void DispBTNode1(BTNode t)
//递归先序遍历
{
  if(t!=NULL)
  {
    printf("%c",t.data);
    DispBTNode1(t.lchild);
    DispBTNode1(t.rchild);
  }
}
```

```
void PreOrder31(BTNode *&t)      //先序遍历的非递归算法 2
{
  BTNode *st[MaxSize];          //定义一个顺序栈
  int top=-1;                   //栈顶指针初始化
  BTNode *p=t;
  while (top!=-1 || p!=NULL)
  {
    while (p!=NULL)             //访问*p 及其所有左下结点并进栈
    {
      cout << p->data;          //访问*p 结点
      top++;  st[top]=p;
      p=p->lchild;
    }
    if (top!=-1)                //若栈不空
    {
      p=st[top];top--;          //出栈*p 结点
      p=p->rchild;              //转向处理其右子树
    }
  }
}
```

8.2 中序遍历

中序遍历二叉树的过程是：

① 中序遍历左子树；② 访问根结点；③ 中序遍历右子树。

例如，下图的二叉树的中序序列为：DGBAECF。显然，在一棵二叉树的中序序列中，根结点值将其

序列分为前后两部分，前部分为左子树的中序序列，后部分为右子树的中序序列。

```
void DispBTNode2(BTNode *t)
//递归中序遍历
{
  if(t!=NULL)
  {
    DispBTNode2(t->lchild);
    printf("%c",t->data);
    DispBTNode2(t->rchild);
  }
}
```

```
void InOrder21(BTNode *&t)  //中序遍历的非递归算法
{
  BTNode *st[MaxSize];      //定义一个顺序栈
  int top=-1;               //栈顶指针初始化
  BTNode *p=t;
  while (top!=-1 || p!=NULL)  //栈不空或者 p 不空时循环
  {
    while (p!=NULL)         //扫描*p 的所有左结点并进栈
    {
      top++; st[top]=p;
      p=p->lchild;
    }
    if (top>-1)             //若栈不空
    {
      p=st[top]; top--;   //出栈*p 结点
      cout << p->data;   //访问*p 结点
      p=p->rchild;    //转向处理右子树
    }
  }
}
```

8.3 后序遍历

后序遍历二叉树的过程是：

① 后序遍历左子树；② 后序遍历右子树；③ 访问根结点。

例如，下图的二叉树的后序序列为：GDBEFCA。显然，在一棵二叉树的后序序列中，最后一个元素即为根结点对应的结点值

```
void DispBTNode3(BTNode *t)
//递归后序遍历
{
  if(t!=NULL)
  {
    DispBTNode3(t->lchild);
    DispBTNode3(t->rchild);
    printf("%c",t->data);
  }
}
```

```
void PostOrder21(BTNode *&t)  //后序遍历的非递归算法
{
  BTNode *st[MaxSize];     //定义一个顺序栈
  int top=-1;              //栈指针置初值
  BTNode *p=t,*q;
bool flag;
//若当前结点的左子树已处理则为 true,否则为 false
  do
  {
    while (p!=NULL) //将*p 结点及其所有左下结点进栈
    {
      top++; st[top]=p;
      p=p->lchild;
    }
```

```
    q=NULL;
//q 指向栈顶结点的前一个已访问的结点或为 NULL
    flag=true; //表示*p 结点的左子树已遍历或为空
    while (top!=-1 && flag==true)
    {
      p=st[top]; //取出当前的栈顶结点
      if (p->rchild==q)//若*p 结点右子树已访问或为空
      {
        cout << p->data;   //访问*p 结点
        top--;             //结点访问后退栈
        q=p;   //让 q 指向则被访问的结点
      }
      else      //若*p 结点右子树没有遍历
      {
        p=p->rchild;     //转向处理其右子树
        flag=false; //此时*p 结点的左子树未遍历
      }
    }
  } while (top!=-1);
}
```

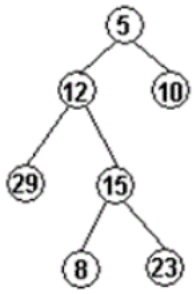
8.4 案例讲解

1 [7275] 二叉树求子树值

已知一棵二叉树用邻接表结构存储，查找二叉树中值为 x 的结点（每个节点的值都不相同），求出以该点为根的子树的节点个数及权值和。

点)。例：如图二叉树的数据文件的数据格式如下:

```
7
15
5 2 3
12 4 5
10 0 0
29 0 0
15 6 7
8 0 0
23 0 0
```



输入

第一行 n 为二叉树的结点个树，n≤100；第二行 x 表示要查找的结点的值；以下第一列数据是各结点的值，第二列数据是左儿子结点编号，第三列数据是右儿子结点编号。

输出

以该点为根的子树的节点个数及权值和。

样例输入	样例输出
7 15 5 2 3 12 4 5 10 0 0 29 0 0	3 46

15 6 7 8 0 0 23 0 0	

```
#include<bits/stdc++.h>
using namespace std;
struct node{
        int data;
        int left,right;
}a[105];
int ans=0;
int s1=0,s2=0;
int n,x;
void dfs(int i){
        s1++,s2=s2+a[i].data;
        if(a[i].left)dfs(a[i].left);
        if(a[i].right)dfs(a[i].right);
}
void mid_order(int root){
        if(a[root].left)mid_order(a[root].left);
        ans++;
        if(a[root].data==x){
                dfs(root);
                return;
        }
        if(a[root].right)mid_order(a[root].right);

}
int main(){
        cin>>n>>x;
        for(int i=1;i<=n;i++){
                cin>>a[i].data>>a[i].left>>a[i].right;
        }

        mid_order(1);
        cout<<s1<<" "<<s2<<endl;
        return 0;
}
```

2[3298] 二叉树建树

括号表示法表示的二叉树字符串 A(B(D,F(E,)),C(G(,H),I))，建立这棵树，并用先序遍历输出结果。

分析：建树主要有两种方法，一种是用栈模拟

```
#include<bits/stdc++.h>
using namespace std;
struct node{
    int left,right,id,father;//左右孩子、编号、父节点
    char ch;//数据域
}t[100];//A(B(D,F(E,)),C(G(,H),I))
string str;
stack<int>st;
void inorder(int root){
    cout<<t[root].ch;
    if(t[root].left)inorder(t[root].left);
    if(t[root].right)inorder(t[root].right);
}
int main(){
    int k;
    cin>>str;
    int len=str.length();
    int id=0;
    for(int i=0;i<len;i++){
        if(str[i]=='('){
            1                                   //k 表示下一点是左还是右
        }//A(B(D,F(E,)),C(G(,H),I))
        else if(str[i]==')')  2                 //栈顶元素出栈
        else if(str[i]==',')k=2;//转到右孩子
        else{
            if(id==0){//处理根节点
                id++;
                t[id].id=id,t[id].father=-1;
                t[id].ch=str[i]; //结点赋值
            }
            else{//不是根结点
                if(k==1){
                    id++;//t[id].father=st.top();
                    3
                    4
                }
                else if(k==2){
                    id++;//t[id].father=st.top();
                    5
                    6
                }
            }
        }
```

```
    }
    inorder(1);cout<<endl;
    return 0;
}
```

第二种递归模拟：

```
#include<bits/stdc++.h>
using namespace std;//A(B(D,F(E,)),C(G(,H),I))
char s[10000];
void dfs(int l,int r)
{
    int x,start;
    for(int i=l;i<=r;i++)
    {
        if(s[i]=='(')
        {
            x=1,start=i;
            while(x!=0)
            {
                i++;
                if(s[i]=='(')
                    x++;
                else if(s[i]==')')
                    x--;
            }
            dfs(start+1,i-1);
        }
        if(s[i]!=','&&s[i]!=')')
            cout<<s[i];
    }
}
int main()
{
    scanf("%s",s);
    dfs(0,strlen(s)-1);
    return 0;
}
```

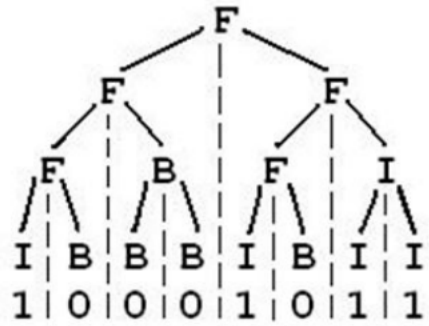
3[3081]]FBI 树-数据结构

[2004_p4 我们可以把由“0”和“1”组成的字符串分为三类：全“0”串称为 B 串，全“1”串称为 I 串，既含“0”又含“1”的串则称为 F 串。

FBI 树是一种二叉树（如下图），它的结点类型也包括 F 结点，B 结点和 I 结点三种。由一个长度为 2N 的“01”串 S 可以构造出一棵 FBI 树 T，递归的构造方法如下：

1) T 的根结点为 R，其类型与串 S 的类型相同；
2) 若串 S 的长度大于 1，将串 S 从中间分开，分为等长的左右子串 S1 和 S2；由左子串 S1 构造 R 的左子树 T1，由右子串 S2 构造 R 的右子树 T2。
现在给定一个长度为 2N 的“01”串，请用上述构造方法构造出一棵 FBI 树，并输出它的后序遍历序列。
输入每组输入数据的第一行是一个整数 N（0<=N<=10），第二行是一个长度为 2N 的“01”串。

数据规模：
对于 40%的数据，N<=2；
对于全部的数据，N<=10。
输出 每组输出包括一行，这一行只包含一个字符串，即 FBI 树的后序遍历序列。



样例输入
3
10001011
样例输出
IBFBBBFIBFIIIFF
本题主要是建树、遍历树。

1.建树。按照题意是在递归过程中建立树，建树的方法实际上就是树的先序遍历（先根节点，再左右子树）。当本节点长度大于 1 时递归建立子树。

2.输出。而输出过程是对树的后序遍历（先左右子树，再根节点），这里有个技巧就是可以和建树过程集成在一起。只需将代码放在递归调用之后就可以了。

3.判断。最后是判断当前节点的 FBI 树类型，可以用 B（初始值为 1）保存全是‘0’的情况，如果遇到‘1’就将 B 置为 0，用 I（初始值为 1）保存全是‘1’的情况，如果遇到‘0’就将 I 置为 0。最后判断 B 和 I 中的值，如果两个都为 0 则输出 F（不全为‘0’，不全为‘1’）。

方法 1：

```
#include<bits/stdc++.h>
using namespace std;
int node[3000];
int len=1;

void bulidTree(int root){//建树
    if(root>=len)return;//表示遍历结束，最后一个点的编号为 len-1
    bulidTree(2*root);
    bulidTree(2*root+1);//类似于后续遍历，先访问做，再访问右，在访问自己
    if(node[2*root]==0&&node[2*root+1]==0)node[root]=0;
    else if(node[2*root]==1&&node[2*root+1]==1)node[root]=1;
    else node[root]=2;
}
void lastTree(int root){//根据自己的理解写，可以不写满
```

```
        ____________________________________
        ____________________________________
        ____________________________________
        ____________________________________
}

int main(){
        int n;
        string s;
        cin>>n>>s;
//    len=n<<1;
        for(int i=1;i<=n;i++)len=2*len;
        for(int i=len;i<2*len;i++)//初始化树节点，最下面这一层
                node[i]=s[i-len]-'0';
        bulidTree(1);
        lastTree(1);
        return 0;
}
```

方法 2：

```
#include<bits/stdc++.h>
using namespace std;
string s;
int ll,n=0;
void fbi(int l,int r){
   int B=1,I=1;//初始化标记
if(r>l){//如果不是叶子节点
// r==l 时是第一个叶子节点，即样例数据的第 8 个点
      fbi(l,(l+r)/2);//查找左子树
      fbi((l+r)/2+1,r);//查找右子树
   }
   for(int i=0;i<=r-l;i++){//遍历区间中 0,1 的情况，根据情况标记
      if(s[i+l]=='1') B=0;
      else if(s[i+l]=='0')I=0;
   }
   if(B!=0)    cout<<"B";//如果不存在 1 则为 B 串
   else if(I!=0)    cout<<"I";//如果不存在 0 则为 I 串
   else cout<<"F";//否则为 F 串
}
int main(){
   cin>>n>>s;
   fbi(0,(1<<n)-1);//遍历字符串
   return 0;
}
```

写法 3：

```
#include <bits/stdc++.h>
using namespace std;
string str;
void fbi(int left,int right)
{
        if(left>right) return ;
        int mid=(left+right)/2,B=0,I=0;
        if(left!=right)
        {
        fbi(left,mid);
        fbi(mid+1,right);
        }
        while(left<=right) if(str[left++]=='0') B++; else I++;
        if(B!=0&&I!=0) printf("F");
        else if(I!=0&&B==0) printf("I");
        else printf("B");
}
int main()
{
        int n;
        scanf("%d",&n);
        cin>>str;getchar();
        fbi(0,str.size()-1);
        return 0;
}
```

4 [2909] 二叉树遍历(flist)

树和二叉树基本上都有先序、中序、后序、按层遍历等遍历顺序，给定中序和其它一种遍历的序列就可以确定一棵二叉树的结构。

假定一棵二叉树一个结点用一个字符描述，现在给出**中序和按层遍历的字符串**，求该树的先序遍历字符串。

输入

两行，每行是由字母组成的字符串（一行的每个字符都是唯一的），分别表示二叉树的中序遍历和按层遍历的序列。

输出　一行，表示二叉树的先序序列。

样例输入	样例输出
DBEAC ABCDE	ABDEC

```
#include <bits/stdc++.h>
using namespace std;
string s1,s2;
void slove(int l1,int r1, int l2, int r2){
```

```
    //DBEAC 中序遍历 //ABCDE 层次遍历
    int pos,flag=0;
    //1 在层次遍历中取得根节点 s2[i]
    for(int i=l2;i<=r2;i++){
        for(int j=l1;j<=r1;j++){
            if(s1[j]==s2[i]){//2 中序遍历中找到 s2[i]及位置
                cout<<s1[j];//找到后输出
                pos=j,flag=1;
                break;
            }
        }
        if(flag==1)break;
    }
    if(l1<pos)___1_____________________//3 遍历搜索左子树
    if(pos<r1)___2_____________________//4 遍历搜索右子树

}
int main()
{
    cin>>s1>>s2;
    int len1=s1.length();
    int len2=s2.length();
    slove(0,len1-1,0,len2-1);
    return 0;
}
```

3 [3009] 加分二叉树

NOIP2003TGT3 设一个 n 个节点的二叉树 tree 的中序遍历为（l,2,3,…,n），其中数字 1,2,3,…,n 为节点编号。每个节点都有一个分数（均为正整数），记第 j 个节点的分数为 di，tree 及它的每个子树都有一个加分，任一棵子树 subtree（也包含 tree 本身）的加分计算方法如下：

subtree 的左子树的加分× subtree 的右子树的加分＋subtree 的根的分数

若某个子树为主，规定其加分为 1，叶子的加分就是叶节点本身的分数。不考虑它的空子树。

试求一棵符合中序遍历为（1,2,3,…,n）且加分最高的二叉树 tree。要求输出；

（1）tree 的最高加分

（2）tree 的前序遍历

[输入格式]	[输出格式]
第 1 行：一个整数 n（n＜30），为节点个数。 第 2 行：n 个用空格隔开的整数，为每个节点的分数（分数＜100） [输入样例]	第 1 行：一个整数，为最高加分（结果不会超过 4,000,000,000）。 第 2 行：n 个用空格隔开的整数，为该树的前序遍历。

5 5 7 1 2 10	[输出样例] 145 3 1 2 4 5

这是一道比较简单的树形 DP。我们可以分成 3 个任务分别解决，

1. 设 f[i, j]为顶点 i 、 顶点 j 所组成的子树的最大分值。若 f[i, j] = -1，则表明最大分值尚未计算出。

f(i,j)={1 (i>j) ; 顶点 i 的分数 (i=j) ; max(f{i,k-1}*f{k+1,j}+顶点 i 的分数 (i<j) 『k 取 i～j』)

root[i, j]——顶点 i..顶点 j 所组成的子树达到最大分值时的根编号。当 i = j 时，root[i, i] = i。

由于问题没有明显的阶段特征，而是呈现为非线性的树形结构，因此，我们采用后序遍历的顺序来计算状态转移方程。计算过程如下：

```
long long search(int L, int r)       // 递归计算 f[L][r]
{
      int    k;
      long long    now, ans;       // 当前分值
      if (L > r) return 1;if (f[L][r]== -1)        // 若尚未计算出顶点 L..顶点 r 对应子树的最高分值
           for(k=L; k<=r; k++) {         // 穷举每一个可能的子根 k
                now = search(L, k-1) * search(k+1, r) + f[k][k];   // 计算以 k 为根的子树的分值
                if(now > f[L][r])   {//若该分值为目前最高，则记入状态转移方程，并记下子根}
                     f[L][r] = now;
      root[L][r] = k;
                }
      }
      return    f[L][r];     {返回顶点 L..顶点 r 对应子树的最高分值}
}
```

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 33;
int a[maxn];
int dp[maxn][maxn];//dp[i][j]表示从第 i 到 j 个节点的最高加分
int tree[maxn][maxn];//tree[i][j]表示从 i 到 j 个节点组成的树的根节点
 int dfs(int l,int r)
{
   if(r<l) return 1;
   if(l==r)//叶子节点
   {
      tree[l][r]=l;
      return a[l];
   }
   if(dp[l][r]) return dp[l][r];
   int Max=0;
   for(int i =l;i<=r;i++)//枚举从 l 到 r 节点组成的树里的根节点
   {
```

```
        if(Max<dfs(l,i-1)*dfs(i+1,r)+a[i])
        {
            Max=dfs(l,i-1)*dfs(i+1,r)+a[i];
            tree[l][r]=i;
        }
    }
    return dp[l][r]=Max;

}
int ans[maxn];//前序遍历的答案
int cnt=0;
void pre(int l,int r)
{
    if(tree[l][r]==0) return;
    ans[++cnt]=tree[l][r];
    pre(l,tree[l][r]-1);//遍历左子树
    pre(tree[l][r]+1,r);//遍历右子树
}

int main()
{

    int n;
    scanf("%d",&n);
    for(int i=1;i<=n;i++) scanf("%d",&a[i]);
   memset(dp,0,sizeof(dp));
    dp[1][n]=dfs(1,n);
    pre(1,n);
    printf("%d\n",dp[1][n]);
    for(int i=1;i<=n;i++){
        printf("%d%c",ans[i],i==n?'\n':' ');
    }
}
```

课前练习

```
1.  #include<iostream>
using namespace std;
int solve(int n,int m)
{
        int i,sum;
        if(m==1) return 1;
        sum=0;
        for(i=1;i<n;i++)
```

```
        sum+= solve(i,m-1);
    return sum;
}
int main()
{
    int n,m;
    cin>>n>>m;
    cout<<solve(n,m)<<endl;
    return 0;
}
```

输入：6 4
输出：_____

2．[2010]**（过河问题）**在一个月黑风高的夜晚，有一群人在河的右岸，想通过唯一的一根独木桥走到河的左岸。在这伸手不见五指的黑夜里，过桥时必须借助灯光来照明，很不幸的是，他们只有一盏灯。另外，独木桥上最多承受两个人同时经过，否则将会坍塌。每个人单独过桥都需要一定的时间，不同的人需要的时间可能不同。两个人一起过桥时，由于只有一盏灯，所以需要的时间是较慢的那个人单独过桥时所花的时间。现输入 n（2≤n<100）和这 n 个人单独过桥时需要的时间，请计算总共最少需要多少时间，他们才能全部到达河的左岸。
例如，有 3 个人甲、乙、丙，他们单独过桥的时间分别为 1、2、4，则总共最少需要的时间为 7。具体方法是：甲、乙一起过桥到河的左岸，甲单独回到河的右岸将灯带回，然后甲、丙再一起过桥到河的左岸，总时间为 2+1+4=7。

```
#include <iostream>
using namespace std;
const int SIZE = 100;
const int INFINITY = 10000;
const bool LEFT = true;
const bool RIGHT = false;
const bool LEFT_TO_RIGHT = true;
const bool RIGHT_TO_LEFT = false;
int n, hour[SIZE];
bool pos[SIZE];
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
int go(bool stage)
{
    int i, j, num, tmp, ans;
    if (stage == RIGHT_TO_LEFT) {
```

```
        num = 0;
        ans = 0;
        for (i = 1; i <= n; i++)
            if (pos[i] == RIGHT) {
                num++;
                if (hour[i] > ans)
                    ans = hour[i];
            }
        if ( ____①____ )
            return ans;
        ans = INFINITY;
        for (i = 1; i <= n - 1; i++)
            if (pos[i] == RIGHT)
                for (j = i + 1; j <= n; j++)
                    if (pos[j] == RIGHT) {
                        pos[i] = LEFT;
                        pos[j] = LEFT;
                        tmp = max(hour[i], hour[j]) +____②____ ;
                        if (tmp < ans)
                            ans = tmp;
                        pos[i] = RIGHT;
                        pos[j] = RIGHT;
                    }
        return ans;
    }
    if (stage == LEFT_TO_RIGHT) {
        ans = INFINITY;
        for (i = 1; i <= n; i++)
            if (____③____ ) {
                pos[i] = RIGHT;
                tmp =____④____ ;
                if (tmp < ans)
                    ans = tmp;
                ____⑤____;
            }
        return ans;
    }
    return 0;
}

int main()
{
    int i;
    cin>>n;
```

```
    for (i = 1; i <=n; i++) {
        cin>>hour[i];
        pos[i] = RIGHT;
    }
    cout<<go(RIGHT_TO_LEFT)<<endl;
    return 0;
}
```

习题

1.[2008]

```
#include<iostream>
#include<cstring>
using namespace std;
#define MAX 100
void solve(char first[], int spos_f, int epos_f, char mid[], int spos_m, int epos_m)
{
    int i, root_m;
    if(spos_f > epos_f)
        return;
    for(i = spos_m; i <= epos_m; i++)
        if(first[spos_f] == mid[i])
        {
            root_m = i;
            break;
        }
    solve(first, spos_f + 1, spos_f + (root_m - spos_m), mid, spos_m, root_m - 1);
    solve(first, spos_f + (root_m - spos_m) + 1, epos_f, mid, root_m + 1, epos_m);
    cout << first[spos_f];
}

int main()
{
    char first[MAX], mid[MAX];
    int len;
    cin >> len;
    cin >> first >> mid;
    solve(first, 0, len - 1, mid , 0, len - 1);
    cout << endl;
    return 0;
}
```

输入：　7

ABDCEGF

BDAGECF　　输出：_____

2.void InOrder21(BTNode *&t) //中序遍历的非递归算法

```
{
  BTNode *st[MaxSize];      //定义一个顺序栈
   1_____________________          //栈顶指针初始化
  BTNode *p=t;
  while (_2_____________________)  //栈不空或者 p 不空时循环
  {
    while (p!=NULL)       //扫描*p 的所有左结点并进栈
    {
      top++;
       3_____________________
      p=p->lchild;
    }
    if (top>-1)          //若栈不空
    {
      p=st[top]; top--;   //出栈*p 结点
      cout << p->data;    //访问*p 结点
      4_____________________   //转向处理右子树
    }
  }
}
```

3.给出一棵二叉树的中序与后序排列。求出它的先序排列。（约定树结点用不同的大写字母表示，长度<=10）

```
#include<bits/stdc++.h>
using namespace std;
char s1[100],s2[100];
/*DBEFAGHCI                DEFBHGICA                    ABDFECGHI
//       中序的左右端点  后续的左右端点
void solve(int l1,int r1,int l2,int r2){
  int i,j,flag=0;
  for(_1_____________________){ //后续第一个字符为树根
      for(j=l1;j<=r1;j++){
      if(_2_____________________){//在中序中找到根的位置
        cout<<s1[j];
        flag=1;
        break;
      }
    }
    if(flag==1)break;
  }
  if(j>l1)_3_____________________//继续递归求解
```

```
    if(j<r1)__4_____________________
}
int main(){
    cin>>s1>>s2;
    int len1=(strlen(s1)),len2=strlen(s2);
    solve(0,len1-1,0,len2-1);
    return 0;
}
```

写法 2：

```
#include<iostream>
#include<cstring>
using namespace std;
void change(string begin,string end){
    if(begin.empty())return;
    char ch=end[end.size()-1];
    int k=begin.find(ch);
    cout<<ch;
    change(begin.substr(0,k),end.substr(0,k));
    change(begin.substr(k+1,begin.size()-k-1),end.substr(k,begin.size()-k-1));
}

int main(){
    string middle,last;
    cin>>middle>>last;
    change(middle,last) ;
    return 0;
}
```

第 9 讲 优先队列与哈夫曼树

9.1 概念介绍

“优先队列”（Priority Queue）是特殊的“队列”，就是将队列中的元素赋予优先级，在访问优先队列中的元素时，具有最高优先级的元素先被访问。队列是先进先出，而优先队列是优先级最高的先出。采用完全二叉树存储的优先队列 称为堆（Heap）。

初始化：需要 头文件 #include<queue>
priority_queue<Type, Container, Functional>
Type 就是数据类型，Container 就是容器类型（Container 必须是用数组实现的容器，比如 vector 等等，但不能用 list。STL 里面默认用的是 vector，当需要用自定义的数据类型时才需要传入这三个参数，使用基本数据类型时，只需要传入数据类型，默认是大顶堆
一般是：

```
priority_queue <int,vector<int>,greater<int> > q; //升序队列
priority_queue <int,vector<int>,less<int> >q;      //降序队列
Priority_queue<node>q;          //     node 为结构体，可以自定义优先级
```

基本操作：
empty( ) //判断一个队列是否为空
pop( ) //删除队顶元素
push( ) //加入一个元素
size( ) //返回优先队列中拥有的元素个数
top( ) //返回优先队列的队顶元素

例 1：阅读如下程序，体会输出结果：

```
#include<iostream>
#include<queue>
using namespace std;
int main()
{
    priority_queue<int,vector<int>,greater<int> >q1;          //升序队列
    priority_queue<int,vector<int>,less<int> >q2;             //降序队列
    int a[5]={4,5,2,1,3};
    for(int i=0;i<5;i++)
    {
        q1.push(a[i]);
        q2.push(a[i]);
    }
    cout<<"升序队列"<<endl;
    while(!q1.empty())
    {
        cout<<q1.top()<<' ';
```

```
        q1.pop();
    }
    cout<<endl;
    cout<<"降序队列"<<endl;
    while(!q2.empty())
    {
        cout<<q2.top()<<' ';
        q2.pop();
    }
    return 0;
}
```

输出结果：

升序队列

1 2 3 4 5

降序队列

5 4 3 2 1

结构体优先队列

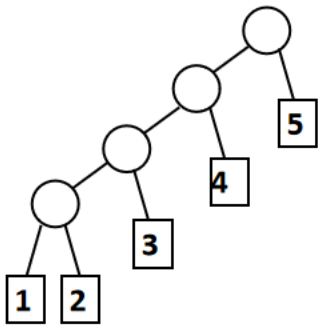
```
struct node
{
    int dis, num;
};
priority_queue<node>que;
bool operator<(const node &a, const node &b)
{
    return a.dis > b.dis;
}//按照 dis 从小到大排序
```

9.2 哈夫曼树

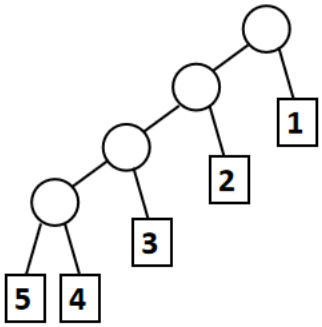
设一棵二叉树有 n 个叶子结点，每个叶子结点带有权值 wk，从根结点到每个叶子结点的长度为 lk，则每个叶子结点的带权路径长度之和就是这棵树的“带权路径长度（Weighted Path Length，简称 WPL）”

假设有 n 个权值{w1 ,w2 , …… , wn} ，构造有 n 个叶子的二叉树，每个叶子的权值是 n 个权值之一。这样的二叉树也许可以构造多个，其中必有一个（或几个）是带权路径长度 WPL 最小的。达到 WPL 最小的二叉树就称为最优二叉树或哈夫曼树。

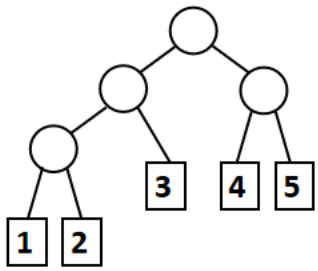
〖例〗有五个叶子结点，它们的权值为{1,2,3,4,5}，用此权值序列可以构造出形状不同的多个二叉树。



WPL = 34



WPL = 50



WPL = 33

注意：哈夫曼树并不唯一，但带权路径长度一定是相同的。

构造过程如下：

假设有 n 个权值，则构造出的哈夫曼树有 n 个叶子结点。 n 个权值分别设为 w1、w2、…、wn，则哈夫曼树的构造规则为：

(1) 将 w1、w2、…，wn 看成是有 n 棵树的森林(每棵树仅有一个结点)；

(2) 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；

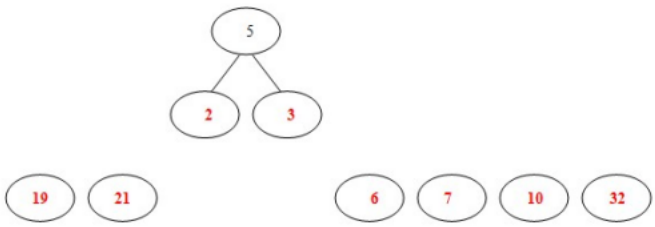
(3)从森林中删除选取的两棵树，并将新树加入森林；

(4)重复(2)、(3)步，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。

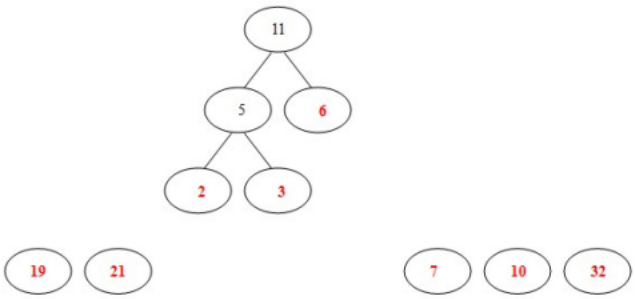
具体过程如下：

（1）8 个结点的权值大小如下：

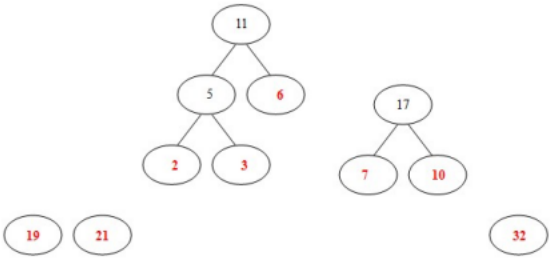
（2）从 19，21，2，3，6，7，10，32 中选择两个权小结点。选中 2，3。同时算出这两个结点的和 5。



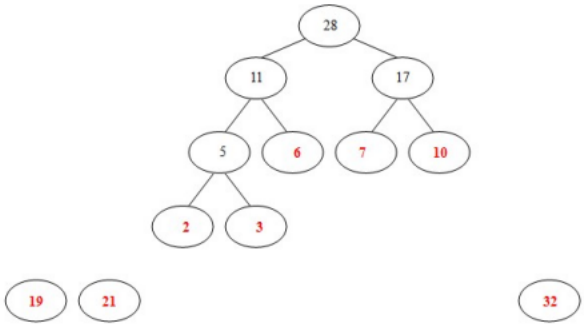
（3）从 19，21，6，7，10，32，5 中选出两个权小结点。选中 5，6。同时计算出它们的和 11。



（4）从 19，21，7，10，32，11 中选出两个权小结点。选中 7，10。同时计算出它们的和 17。

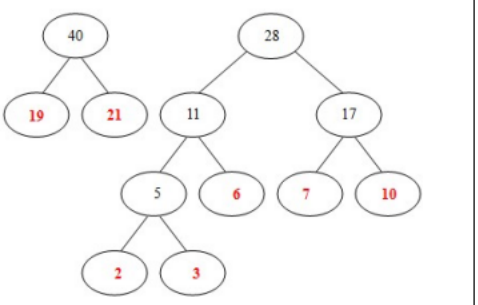


（4）

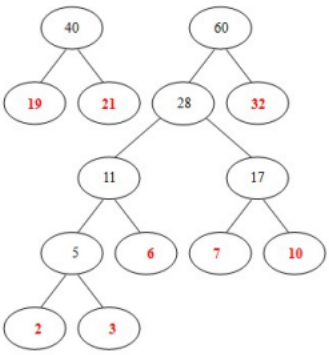


（5）

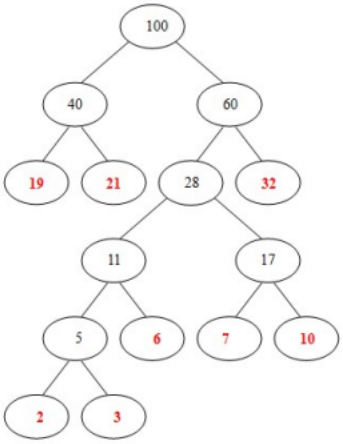
（5）从 19，21，32，11，17 中选出两个权小结点。选中 11，17。同时计算出它们的和 28。
（6）从 19，21，32，28 中选出两个权小结点。选中 19，21。同时计算出它们的和 40。另起一颗二叉树。
（7）从 32，28， 40 中选出两个权小结点。选中 28，32。同时计算出它们的和 60。
（8）从 40， 60 中选出两个权小结点。选中 40，60。同时计算出它们的和 100。 好了，此时哈夫曼树已经构建好了。



（6）



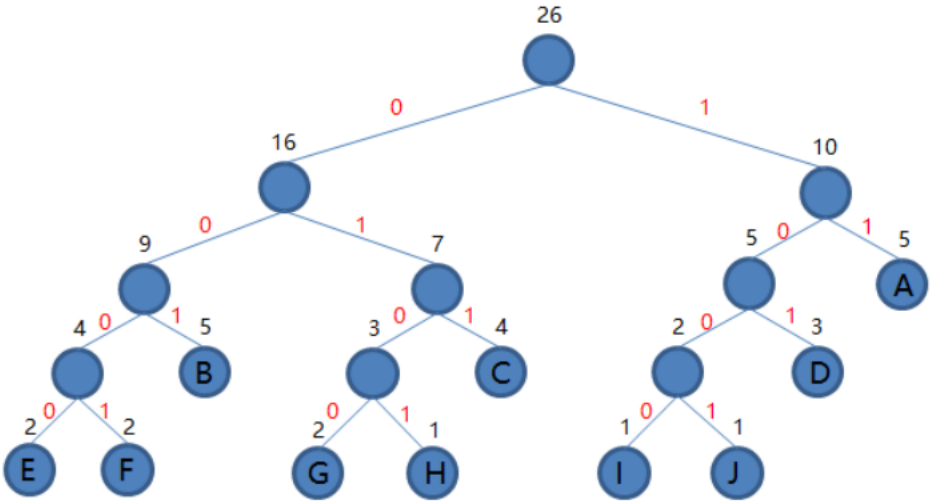
（7）



（8）

哈夫曼编码

首先我们来看这棵构造好的哈夫曼树：（经过左边路径为 0，经过右边路径为 1）
则可直接写出编码，例如：
A:11　B:001　C:011　D　E：0000　F：0001　G：0100　H：0101　I：1000　J：1001



9.3 案例分析

1[6887] 看遍排队

病要排队这个是地球人都知道的常识。
不过经过细心的 0068 的观察，他发现了医院里排队还是有讲究的。0068 所去的医院有三个医生（汗，这么少）同时看病。而看病的人病情有轻重，所以不能根据简单的先来先服务的原则。所以医院对每种病情规定了 10 种不同的优先级。级别为 10 的优先权最高，级别为 1 的优先权最低。医生在看病时，则会在他的队伍里面选择一个优先权最高的人进行诊治。如果遇到两个优先权一样的病人的话，则选择最早来排队的病人。

现在就请你帮助医院模拟这个看病过程。
Input
输入数据包含多组测试，请处理到文件结束。
每组数据第一行有一个正整数 N(0<N<2000)N(0<N<2000)表示发生事件的数目。
接下来有 N 行分别表示发生的事件。
一共有两种事件：
1:”IN A B”,表示有一个拥有优先级 B 的病人要求医生 A 诊治。(0<A<=3,0<B<=10)(0<A<=3,0<B<=10)
2:”OUT A”,表示医生 A 进行了一次诊治，诊治完毕后，病人出院。(0<A<=3)(0<A<=3)
Output
对于每个”OUT A”事件，请在一行里面输出被诊治人的编号 ID。如果该事件时无病人需要诊治，则输出”EMPTY”。
诊治人的编号 ID 的定义为：在一组测试中，”IN A B”事件发生第 K 次时，进来的病人 ID 即为 K。从 1 开始编号。

Sample Input	Sample Output

7 IN 1 1 IN 1 2 OUT 1 OUT 2 IN 2 1 OUT 2 OUT 1 2 IN 1 1 OUT 1	2 EMPTY 3 1 1

```
#include<iostream>
#include <queue>
using namespace std;
struct node{
        int no;
        int grade;
};
//先定义结构体，再写重载函数定义优先级比较简单明了
bool operator < (const node &x, const node &y)//重载 "<" 操作符定义优先级
{
        if(x.grade==y.grade)
            return x.no>y.no;
        else
            return x.grade<y.grade;
}
int main()
{
        char str[10];
        int n,do_num,grad;
        while(cin>>n){
              priority_queue<node> q[4];//3个医生，3个队列
              int cnt=0;//病人的排队序号
              for(int i=0;i<n;i++){
                      cin>>str;
                      if(str[0]=='I'){
                              cin >> do_num >> grad;
                              cnt++;
                              node tmp;
                              tmp.grade=grad,tmp.no=cnt;
                              q[do_num].push(tmp);
                      }
                      else if(str[0]=='O'){
```

```
                cin >> do_num;
                if(!q[do_num].empty()){
                        node tmp=q[do_num].top();
                        q[do_num].pop();
                        cout<<tmp.no<<endl;
                }
                else cout<<"EMPTY"<<endl;
            }

        }
    }

    return 0;
}
```

2 [1891] Windows 操作系统的消息队列

消息队列是 Windows 操作系统的最基本要素。对每个进程，系统维护一个消息队列，如果进程执行某种操作，例如单击鼠标等，系统向队列添加一个相应的消息。同时如果队列非空，则进程执行消息循环：根据消息的优先级取出一个消息，并执行相应的操作。注意，优先级数值小表示优先级高。在本题中，要求你模拟消息队列，向消息队列中添加消息和取出消息。

输入

输入中只有一个测试数据，每行为一个命令：GET 或者 PUT，分别表示取消息和添加消息。如果是 PUT 命令，则命令后面紧跟一个字符串，代表消息名称(最多 10 个字符)，然后是 2 个整数代表参数和优先级(优先级为小于 20 的正整数)。至多有 60000 个命令。注意，每个消息可能会出现 2 次甚至更多次，测试数据保证任一时刻队列中各消息的优先级不一样。处理一直到文件末尾。

输出

对每个 GET 命令，输出从消息队列中取出的消息的名称和参数，如果消息队列为空，则输出 EMPTY QUEUE!，对 PUT 命令，没有输出内容。

样例输入	样例输出
GET PUT msg1 10 5 PUT msg2 10 4 GET GET GET	EMPTY QUEUE! msg2 10 msg1 10 EMPTY QUEUE!

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

struct node {  //定义信息点结构
```

```
    int n, p, num;  //n 为参数，p 为优先级，num 为输入顺序
    string s;
}an;

int  1                          { //定义信息点的大小函数
    if (a.p == b.p)   //优先级相等，就比较输入顺序
        return a.n > b.n;
    return  2                  //优先级不等，就比较优先级
}
int main() {
    string ss;
     3
    int k = 1;
    while (cin>>ss) {
        if (ss[0] == 'G') { //为 get 则拿信息
            if ( 4              ) {//判断队列是否为空
                cout<<"EMPTY QUEUE!"<<endl;
            }
            else {
                5
                cout << an.s << " " << an.num << endl;
                6
            }
        }
        else { //存信息
            cin >> an.s>> an.num>> an.p;
            an.n = k;
            k++;
            7                    }
    }
    return 0;
}
```

3 [3307] 哈夫曼

熵编码器是一种数据编码方法，通过将“浪费的”或“额外的”信息删除，从而实现无损数据压缩。换句话说，熵编码消除了最初不必要的信息，以便准确地对消息进行编码。高熵意味着消息有大量的浪费信息；用 ASCII 编码的英文文本是具有高熵的消息类型的示例。已经压缩的消息，例如 JPEG 图形或 ZIP 归档，具有非常小的熵，并且不能从熵编码的进一步尝试中受益。

在计算机中，一个英文字符用八个比特表示，如 00110111,11100111 等，如 AAAAABCD 每个字符占八个比特，一共八个字符占 64 个比特。但是若用 00 表示 A,用 01 表示 B,用 10 表示 C,用 11 表示 D。则原字符串只占 16 个比特。

若用 0 表示 A, 10 表示 B，110 表示 C, 111 表示 D, 则 A 出现了五次，占 5 个比特，B 出现了 1 次,占 2

个比特，C 出现了一次，占 3 个比特，D 出现了一次，占 3 个比特。一共占 13 个比特。
现在给一个只包含 26 个大写字母和下划线 "_"的字符串，使每个出现的字符都对应一个二进制串，且不能存在一个二进制串是另一个二进制串的前缀。如若用 001 表示 A,0010 表示 B 则不合法，因为 001 是 0010 的前缀。要求在符合要求的前提下最少要多少个比特可以存得下这个字符串。

输入 输入数据有多组，每组一个字符串，包含大写字母和下划线，以"END"为结束符。
输出 对于每个字符串输出一行，包含三个数字，a,b,c。 a 为每个字符都占八个比特时的所占用的比特数，b 为最少占用多少比特数，c 为 a/b, c 结果保留一位小数。

样例输入	样例输出
AAAAABCD	64 13 4.9
THE_CAT_IN_THE_HAT	144 51 2.8
END	

```
#include<bits/stdc++.h>
using namespace std;
struct Node{
        int x;
        friend bool operator <(Node a,Node b){
                return a.x>b.x;
        }
};
int main(){
        int cnt[30];
        char str[1001];
        while(cin>>str){
                if(!strcmp(str,"END")) break;
                Node n1,n2;
                int sum=0,len=strlen(str);
                memset(cnt,0,sizeof(cnt));
                for(int i=0;i<len;i++){
                        if(str[i]=='_') cnt[26]++;
                        else cnt[str[i]-'A']++;
                }
                priority_queue<Node>q;
                for(int i=0;i<=26;i++)
                        if(cnt[i]) n1.x=cnt[i],q.push(n1);
                while(q.size()>1){
                        n1=q.top();q.pop();
                        n2=q.top();q.pop();
                        int t;
                        t=n1.x+n2.x;
                        sum+=t;
                        n1.x+=n2.x;
                        q.push(n1);
```

```
            }
            if(sum==0) sum=len;
            printf("%d %d %.1lf\n",len*8,sum,(len*8.0)/(sum*1.0));
        }
        return 0;
}
```

4 [7538] 哈夫曼编码

哈夫曼编码是一种编码方式， 是可变字长编码的一种， 由 Huffman 于 1952 年提出。该方法完全依据字符出现概率来构造异字头的平均长度最短的码字， 有时称之为最佳编码，一般就叫 Huffman 编码。简单地来说，就是出现概率高的字符使用较短的编码， 反之出现概率低的则使用较长的编码，这便使编码之后的字符串的平均期望长度降低， 从而达到无损压缩数据的目的。
现在请你模拟这样的原则对给定的一个字符串进行字母统计。
输入
只有一行，是一个字符串，由小写英文字母组成，长度不超过 255 个字符。
输出
有若干行，每行有两部分组成：一个字母和该字母出现的频率，中间用一个空格分隔，并按频率高低排列，频率相同时则按字母的 ASC 码的先后顺序排列。
样例输入
soon
样例输出
o 2
n 1
s 1

```
#include <bits/stdc++.h>
using namespace std;
const int M=300;
char s[M];
struct node{
   int cnt,id;
}f[35];
int cmp(node x,node y){
   if(x.cnt==y.cnt) return x.id<y.id;
   else return x.cnt>y.cnt;
}
int main() {
   cin>>s+1;
   int l=strlen(s+1);
   for(int i=1;i<=l;i++){
   int idx=s[i]-'a'+1;
   f[idx].cnt++;
   f[idx].id=idx;
```

```
    }
    sort(f+1,f+1+26,cmp);
    for(int i=1;i<=26;i++)
    if(f[i].cnt!=0) printf("%c %d\n",f[i].id+'a'-1,f[i].cnt);
    return 0;
}
```

课前练习

1．[2009]**（最大连续子段和）**给出一个数列（元素个数不多于 100），数列元素均为负整数、正整数、0。请找出数列中的一个连续子数列，使得这个子数列中包含的所有元素之和最大，在和最大的前提下还要求该子数列包含的元素个数最多，并输出这个最大和以及该连续子数列中元素的个数。例如数列为 4，-5，3，2，4 时，输出 9 和 3；数列为 1 2 3 -5 0 7 8 时，输出 16 和 7。

```
#include <iostream>
using namespace std;
int a[101];
int n,i,ans,len,tmp,beg;
int main(){
    cin >> n;
    for (i=1;i<=n;i++)
        cin >> a[i];
    tmp=0;
    ans=0;
    len=0;
    beg= ____①____;
    for (i=1;i<=n;i++){
        if (tmp+a[i]>ans){
            ans=tmp+a[i];
            len=i-beg;
        }
        else if (_____②_____ &&i-beg>len)
            len=i-beg;
        if (tmp+a[i]____③____){
            beg=____④____;
            tmp=0;
        }
        else
        _____⑤_____;
    }
    cout << ans << " " << len << endl;
    return 0;
}
```

2.[2013] （二叉查找树） 二叉查找树具有如下性质： 每个节点的值都大于其左子树上所有节点的 值、小

于其右子树上所有节点的值。试判断一棵树是否为二叉查找树。
输入的第一行包含一个整数 n，表示这棵树有 n 个顶点， 编号分别为 1, 2, … , n，其 中编号为 1 的为根结点。之后的第 i 行有三个数 value, left_child , right_child ，分别表示该节点关键字的值、左子节点的编号、右子节点的编号；如果不存在左子节点或右子节点，则用 0 代替。输出 1 表示这棵树是二叉查找树，输出 0 则表示不是。

```
#include <iostream>
 using namespace std;
const int SIZE = 100;
const int INFINITE = 1000000;
struct node
{
    int left_child, right_child, value;
};node a[SIZE];
int is_bst(int root, int lower_bound, int upper_bound)
{
    int cur;
    if (root == 0)
        return 1;
    cur = a[root].value;
    if ((cur > lower_bound) && (  (1)                ) && (is_bst(a[root].left_child, lower_bound, cur) == 1) &&
    (is_bst( (2)        , (3)          , (4)           ) == 1))
        return 1;
    return 0;
}
int main()
{
    int i, n; cin>>n;
    for (i = 1; i <= n; i++)
        cin>>a[i].value>>a[i].left_child>>a[i].right_child;
    cout<<is_bst( (5) , -INFINITE, INFINITE)<<endl;
    return 0;
}
```

习题

```
1.#include<cstring>
#include <iostream>
using namespace std;
int n,m,z,i,j;
bool a[5001];
int main() {
    z=1;
    cin>>n>>m;
    memset(a,0,sizeof(a));
```

```
    for(i=1; i<=m; i++)
        for(j=1,j<=n; j++)
            if(j%i==0)a[j]=!a[j];
    for(i=1; i<=n; i++)
        if(a[i]) {
            if(z)z=0;
            else cout<<",";
            cout<<i;
        }
    cout>>endl;
    return 0;
}
```

输入：

10 10

输出：_____

2. [2008](找第 k 大的数) 给定一个长度为 1,000,000 的无序正整数序列，以及另一个数 n (1<=n<=1000000)，然后以类似快速排序的方法找到序列中第 n 大的数（关于第 n 大的数：例如序列{1，2，3，4，5，6}中第 3 大的数是 4）。

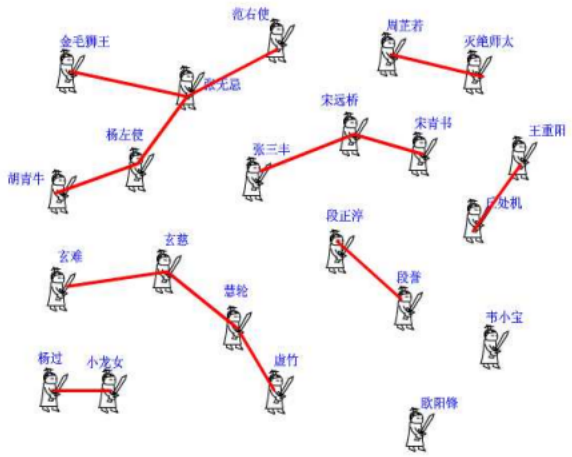
```
#include <iostream>
using namespace std;
int a[1000001],n,ans = -1;
void swap(int &a,int &b)
{
    int c;
    c = a; a = b;     b = c;
}

int FindKth(int left, int right, int n)
{
    int tmp,value,i,j;
    if (left == right) return left;
    tmp = rand()% (right - left) + left;
    swap(a[tmp],a[left]);
    value = _____①_____
    i = left;
    j = right;
    while (i < j)
    {
        while (i < j && _____②_____) j --;
        if (i < j) {a[i] = a[j]; i ++;} else break;
        while (i < j &&  _____③_____ ) i ++;
        if (i < j) {a[j] = a[i]; j - -;} else break;
```

```
    }
        ④
    if (i < n) return    FindKth(          ⑤          );
    if (i > n) return                 ⑥
    return i;
}

int main()
{
    int i;
    int m = 1000000;
    for (i = 1;i <= m;i ++)
        cin >> a[i];
    cin >> n;
    ans = FindKth(1,m,n);
    cout << a[ans];
    return 0;
}
```

第 10 讲 并查集

在一些有 N 个元素的集合应用问题中，我们通常是在开始时让每个元素构成一个单元素的集合，然后按一定顺序将属于同一组的元素所在的集合合并，其间要反复查找一个元素在哪个集合中。这一类问题看似并不复杂，但数据量极大，若用正常的数据结构来描述的话，往往超过了空间的限制，计算机无法承受；即使在空间上能勉强通过，运行的时间复杂度也极高，根本不可能在规定的运行时间内计算出试题需要的结果，只能采用一种特殊数据结构——并查集来描述。

10.1 并查集的操作



FIND(x)：返回一个指向包含 x 的集合的代表。

```
int find(int x)     //查找用非递归的实现
{
        while (father[x] != x) x = father[x];
        return x;
}
int find(int x) //查找用递归的实现
{
        if (father[x] != x) return find(father[x]);
       else return x;
}
```

UNIONN(x,y)：将包含 x 和 y 的动态集合（例如 Sx 和 Sy）合并为一个新的集合，假定在此操作前这两个集合是分离的。结果的集合代表是 Sx∪Sy 的某个成员。一般来说，在不同的实现中通常都以 Sx 或者 Sy 的代表作为新集合的代表。此后，由新的集合 S 代替了原来的 Sx 和 Sy。

```
void unionn(int r1,int r2)//合并
{
      father[r2] = r1;
}
```

10.2 案例讲解

1 [2903] 亲戚(relation)

题目描述

或许你并不知道，你的某个朋友是你的亲戚。他可能是你的曾祖父的外公的女婿的外甥女的表姐的孙子。如果能得到完整的家谱，判断两个人是否是亲戚应该是可行的，但如果两个人的最近公共祖先与他们相隔好几代，使得家谱十分庞大，那么检验亲戚关系实非人力所能及。在这种情况下，最好的帮手就是计算机。为了将问题简化，你将得到一些亲戚关系的信息，如 Marry 和 Tom 是亲戚，Tom 和 Ben 是亲戚，等等。从这些信息中，你可以推出 Marry 和 Ben 是亲戚。请写一个程序，对于我们的关于亲戚关系的提问，以最快的速度给出答案。

输入由两部分组成。第一部分以 N，M 开始。N 为问题涉及的人的个数(1≤N≤20000)。这些人的编号为 1,2,3,…, N。下面有 M 行(1≤M≤1000000)，每行有两个数 ai,biai,bi ，表示已知 aiai 和 bibi 是亲戚。
第二部分以 Q 开始。以下 Q 行有 Q 个询问(1≤ Q ≤1000000)，每行为 ci,dici,di ，表示询问 cici 和 didi 是否为亲戚。

输出 对于每个询问 ci,dici,di ，输出一行：若 cici 和 didi 为亲戚，则输出“Yes”，否则输出“No”。

样例输入	样例输出
10 7 2 4 5 7 1 3 8 9 1 2 5 6 2 3 3 3 4 7 10 8 9	Yes No Yes

[算法分析]

将每个人抽象成为一个点，数据给出 M 个边的关系，两个人是亲戚的时候两点间有一条边。很自然的就得到了一个 N 个顶点 M 条边的图论模型，注意到传递关系，在图中一个连通块中的任意点之间都是亲戚。对于最后的 Q 个提问，即判断所提问的两个顶点是否在同一个连通块中。

用传统的思路，可以马上反应过来，对于输入的 N 点 M 条边，找出连通块，然后进行判断。但这种实现思首先必须保存 M 条边，然后再进行普通的遍历算法，效显然不高。再进一步考虑，如果把题目的要求改一改，于边和提问相间输入，即把题目改成：

第一行是 N，M。N 为问题涉及的人的个数(1≤N≤20000)。这些人的编号为 1,2,3,⋯, N。下面有 M 行(1≤M 000 000)，每行有三个数 ki,ai, bi。ai, bi 表示两个元素，0 或 1，ki 为 1 时表示这是一条边的信息，即 ai, bi 是亲关系；ki 为 0 时表示是一个提问，根据此行以前所得到信息，判断 ai, bi 是否亲戚，对于每条提问回答 Yes 或者 这个问题比原问题更复杂些，需要在任何时候回答提问

输入关系	分离集合
初始状态	{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}
(2,4)	{1}{2,4}{3}{5}{6}{7}{8}{9}{10}
(5,7)	{1}{2,4}{3}{5,7}{6}{8}{9}{10}
(1,3)	{1,3}{2,4}{5,7}{6}{8}{9}{10}
(8,9)	{1,3}{2,4}{5,7}{6}{8,9}{10}
(1,2)	{1,2,3,4}{5,7}{6}{8,9}{10}
(5,6)	{1,2,3,4}{5,6,7}{8,9}{10}
(2,3)	{1,2,3,4}{5,6,7}{8,9}{10}

个
路
率
对

≤ 2
ki 为
戚
的
No。
的

两个人的关系，并且对于信息提示还要能立即合并两个连通块。采用连通图思想显然在实现上就有所困难，因为要表示人与人之间的关系。

用集合的思路，对于每个人建立一个集合，开始的时候集合元素是这个人本身，表示开始时不知道任何人是他的亲戚。以后每次给出一个亲戚关系时，就将两个集合合并。这样实时地得到了在当前状态下的集合关系。如果有提问，即在当前得到的结果中看两元素是否属于同一集合。

对于该问题，我们可以运用并查集简单地进行如下做法：

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

```
#define maxn 20001
int father[maxn];
int m,n,i,x,y,q;
/*
int find(int x)              //用非递归的实现
{
  while (father[x] != x) x = father[x];
  return x;
}
*/
int find(int x)              //用递归的实现
{
  if (father[x] != x) return find(father[x]);
  else return x;
}
void unionn(int r1,int r2)
{
  father[r2] = r1;
}
int main()
{
  cin >> n >> m;
  for (i = 1; i <= n; i++)
    father[i] = i;          //建立新的集合，其仅有的成员是 i
  for (i = 1; i <= m; i++)
  {
    scanf("%d%d",&x,&y);
    int r1 = find(x);
    int r2 = find(y);
    if (r1 != r2) unionn(r1,r2);
  }
  cin >> q;
  for (i = 1; i <= q; i++)
  {
    scanf("%d%d",&x,&y);
    if (find(x) == find(y)) printf("Yes\n");
      else printf("No\n");
  }
  return 0;
}
```

下面有一种优化的方法：

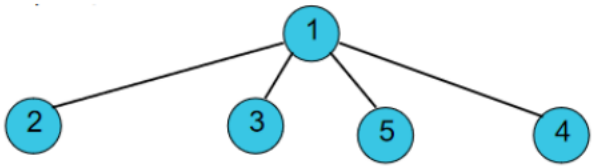
并查集的路径压缩

此种做法就是将元素的父亲结点指来指去地指，当这棵树是链的时候，可见判断两个元素是否属于同

一集合需要 O(n)的时间，于是路径压缩产生了作用。

路径压缩实际上是在找完根结点之后，在递归回来的时候顺便把路径上元素的父亲指针都指向根结点。

这就是说，我们在“合并 5 和 3”的时候，不是简单地将 5 的父亲指向 3，而是直接指向根节点 1，如图：



[程序清单]

（1）初始化：

```
for (i = 1; i <= n; i++) father[i] = i;
```

因为每个元素属于单独的一个集合，所以每个元素以自己作为根结点。

（2）寻找根结点编号并压缩路径：

```
int find (int x)
{
    if (father[x] != x) father[x] = find (father[x]);
    return father[x];
}
```

（3）合并两个集合：

```
void unionn(int x,int y)
{
    x = find(x);y = find(y);
    father[y] = x;
}
```

（4）判断元素是否属于同一集合：

```
bool   judge(int x,int y)
{
    x = find(x);
    y = find(y);
    if   (x == y)   return true;
            else return false;
}
```

这个的引题已经完全阐述了并查集的基本操作和作用。

//优化的具体程序如下:

```
#include<iostream>
#include<cstdio>
using namespace std;
#define maxn 20001
int father[maxn];
```

```
int m,n,i,x,y,q;
int find(int x)   //用递归的实现
{
    1                                    //路径压缩
    return   2
}
void unionn(int r1,int r2)
{
    3
}
int main()
{
    cin >> n >> m;
    for (i = 1; i <= n; i++)
        father[i] = i;                  //建立新的集合，其仅有的成员是 i
    for (i = 1; i <= m; i++)
    {
        scanf("%d%d",&x,&y);
        int r1 =  4
        int r2 =  5
        if (r1 != r2) 6
    }
    cin >> q;
    for (i = 1; i <= q; i++)
    {
        scanf("%d%d",&x,&y);;
        if ( 7                                ) printf("Yes\n");
        else printf("No\n");
    }
    return 0;
}
```

2[2935]：食物链

[NOI2001]动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B， B 吃 C，C 吃 A。现有 N 个动物，以 1－N 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。
有人用两种说法对这 N 个动物所构成的食物链关系进行描述：
第一种说法是"1 X Y"，表示 X 和 Y 是同类。
第二种说法是"2 X Y"，表示 X 吃 Y。
此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
1）当前的话与前面的某些真的话冲突，就是假话；
2）当前的话中 X 或 Y 比 N 大，就是假话；
3）当前的话表示 X 吃 X，就是假话。

你的任务是根据给定的 N（1≤ N ≤50,000）和 K 句话（0≤K≤100,000），输出假话的总数。

输入	输出
第一行是两个整数 N 和 K，以一个空格分隔。 以下 K 行每行是三个正整数 D，X，Y，两数之间用一个空格隔开，其中 D 表示说法的种类。 若 D=1，则表示 X 和 Y 是同类。 若 D=2，则表示 X 吃 Y。 样例输入 100 7 1 101 1 2 1 2 2 2 3 2 3 3 1 1 3 2 3 1 1 5 5	只有一个整数，表示假话的数目。 样例输出 3

对于动物 x 和 y，可能有 x 吃 y，x 与 y 同类，x 被 y 吃。但由于关系还是明显的，1 倍大小、2 倍大小的并查集都不能满足需求，3 倍大小不就行了！类似上面，我们将并查集分为 3 个部分，每个部分代表着一种动物种类。

设我们有 n 个动物，开了 3n 大小的种类并查集，其中 1 ~ n 的部分为 A 群系，n + 1~ 2n 的部分为 B 群系，2n + 1 ~ 3n 的部分为 C 群系。

我们可以认为 A 表示中立者，B 表示生产者，C 表示消费者。此时关系明显：A 吃 B，A 被 C 吃。
当然，我们也可以认为 B 是中立者，这样 C 就成为了生产者，A 就表示消费者。（还有 1 种情况不提及了）

联想一下 2 倍大小并查集的做法，不难列举出：当 A 中的 x 与 B 中的 y 合并，有关系 x 吃 y；当 C 中的 x 和 C 中的 y 合并，有关系 x 和 y 同类等等……

但仍然注意，我们不知道某个动物属于 A，B，还是 C，我们 3 个种类都要试试！也就是说，每当有 1 句真话时，我们需要合并 3 组元素。容易忽略的是，题目中指出若 x 吃 y，y 吃 z，应有 x 被 z 吃。
这个关系还能用种类并查集维护吗？答案是可以的。若将 x 看作属于 A，则 y 属于 B，z 属于 C。最后，根据关系 A 被 C 吃可得 x 被 z 吃。

```
//0 是同类，1 是被吃，2 是吃人——rank[]
#include <iostream>
using namespace std;
const int maxn=5e5+5;
int pre[maxn],ranks[maxn],n,m,Q,ans;

void init()   //初始化
{
   for(int i=0; i<=n; i++)
   {
      pre[i]=i;
      ranks[i]=0;
```

```
    }
}

int finding(int x)   //跟 how many dogs 一样，
{            // //要定义一个值来保存父亲，之后加的是原来的父亲而不是更新后的父亲
    if(pre[x]==x) return x;
    int t=pre[x];
    pre[x]=finding(pre[x]);
    ranks[x]=(ranks[x]+ranks[t])%3;
    return pre[x];
}

void join(int d,int x,int y)   //联系起来，因为是 y 被 x 吃，所以 y 的爸爸是 x
{
    int fx=finding(x),fy=finding(y);
    pre[fy]=fx;
    ranks[fy]=(ranks[x]-ranks[y]+d-1+3)%3;
}

int main()
{
    scanf("%d%d",&n,&m);
    init();
    ans=0;
    while(m--)
    {
        int x,y;
        scanf("%d%d%d",&Q,&x,&y);
        if(x<1||y<1||x>n||y>n||(Q==2&&x==y)) ans++;//自己不能吃自己
        else if(finding(x)==finding(y))   //要是已经有联系了
        {
            if(Q==1&&ranks[x]!=ranks[y]) ans++;   //同类的话，rank 要一样
            if(Q==2&&(ranks[x]+1)%3!=ranks[y]) ans++;   //不同类的话，x 被 y 吃，所以 rank[y]应该是 rank[x]+1
        }
        else join(Q,x,y);   //没联系，自己加联系
    }
    printf("%d\n",ans);
    return 0;
}
```

3[2926] 团伙

在某城市里住着 n 个人，任何两个认识的人不是朋友就是敌人，而且满足：
1、我朋友的朋友是我的朋友；

2、我敌人的敌人是我的朋友；
所有是朋友的人组成一个团伙。告诉你关于这 n 个人的 m 条信息，即某两个人是朋友，或者某两个人是敌人，请你编写一个程序，计算出这个城市最多可能有多少个团伙？
输入
第 1 行为 n 和 m，1<n<1000,1≤m≤100 000；
以下 m 行，每行为 p x y，p 的值为 0 或 1，p 为 0 时，表示 x 和 y 是朋友，p 为 1 时，表示 x 和 y 是敌人。
输出
一个整数，表示这 n 个人最多可能有几个团伙。

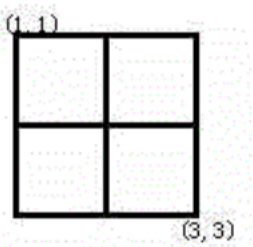
样例输入
6 4
1 1 4
0 3 5
0 4 6
1 1 2
样例输出
3

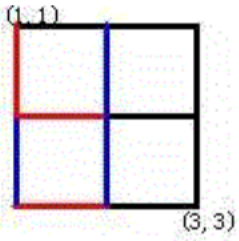
```
include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;
int n,m;
int a[1001][1001],father[1001];//a[i][j]存储第 i 个人的第 j 个敌人
int find(int);
void merge(int,int);
int find(int);
void work();
int main()
{
   work();
   return 0;
}
void work()
{
   memset(a,0,sizeof(a));
   cin>>n>>m;
   for(int i=1;i<=n;i++) father[i]=i;//初始化为自己
   int p,x,y;
   for(int i=1;i<=m;i++)
   {
      cin>>p>>x>>y;
      if(p==0) merge(x,y);//如果 x，y 是亲戚，则合并
      if(p==1)//如果 x，y 是敌人，则把 x 并入 y 的所以敌人中，y 并入 x 的敌人中
```

```
    {
      for(int j=1;j<=a[x][0];j++) merge(y,a[x][j]);
      for(int j=1;j<=a[y][0];j++) merge(x,a[y][j]);
      a[y][0]++;a[y][a[y][0]]=x;//a[x][0]记录的 x 的敌人数
      a[x][0]++;a[x][a[x][0]]=y;
    }
  }
  int tot=0;//入果自己就是树的根表明是一个集合
  for(int i=1;i<=n;i++) if(father[i]==i) tot++;
  cout<<tot<<endl;
}
void merge(int x,int y)
{
  x=find(x);
  y=find(y);
  if(x!=y)father[x]=y;
}
int find(int x)
{
  if(father[x]==x) return x;
  father[x]=find(father[x]);
  return father[x];
}
```

4[2908] 格子游戏

Alice 和 Bob 玩了一个古老的游戏：首先画一个 n × n 的点阵（下图 n = 3）
接着，他们两个轮流在相邻的点之间画上红边和蓝边：





直到围成一个封闭的圈（面积不必为 1）为止，“封圈”的那个人就是赢家。因为棋盘实在是太大了(n ≤ 200)，他们的游戏实在是太长了！他们甚至在游戏中都不知道谁赢得了游戏。于是请你写一个程序，帮助他们计算他们是否结束了游戏？

输入
输入数据第一行为两个整数 n 和 m。m 表示一共画了 m 条线。以后 m 行，每行首先有两个数字(x, y)，代表了画线的起点坐标，接着用空格隔开一个字符，假如字符是"D "，则是向下连一条边，如果是"R "就是向右连一条边。输入数据不会有重复的边且保证正确。
输出
输出一行：在第几步的时候结束。假如 m 步之后也没有结束，则输出一行“draw”。

样例输入

3 5

1 1 D

1 1 R

1 2 D

2 1 R

2 2 D

样例输出

4

这道题用结构体实现二维的并查集，在每一次比较的时候只需要把 x 和 y 都比较一下就可以；每一次划线，先判断一下，如果起点在集合里，终点也在集合里，那么符合条件，输出当时的步数就可以了；如果不符合上面的条件，把点入集合；如果到最后还没有符合条件的情况，输出 draw 就可以了。

```
#include<iostream>
using namespace std;
struct node
{
   int x;
   int y;
};
node father[205][205];
node find(node p)
{
   if(father[p.x][p.y].x!=p.x||father[p.x][p.y].y!=p.y)
      father[p.x][p.y]=find(father[p.x][p.y]);
   return father[p.x][p.y];
}
void join(node a,node b)
{
   node fa=find(a);
   node fb=find(b);
   if(fa.x!=fb.x||fa.y!=fb.y)
      father[fa.x][fa.y]=fb;
}
int n,m,x,y;
char c;
int main()
{
   cin>>n>>m;
   bool flag=false;//false  表示未形成环
   for(int i=1; i<=n; i++)
      for(int j=1; j<=n; j++)
         father[i][j].x=i,father[i][j].y=j;//初始化每个元素的父节点都是自己
```

```
  for(int i=1; i<=m; i++)
  {
    cin>>x>>y>>c;
    if(flag)continue;//已经找到环就不处理，但后面还有输入需要继续
    node a,b;//定义两个划线的点
    a.x=x,a.y=y;//初始化起点
    if(c=='D')//向下画
      b.x=x+1, b.y=y;
    else//向右画
      b.x=x,b.y=y+1;
    node fa=find(a);//找祖先
    node fb=find(b);
    if(fa.x==fb.x&&fa.y==fb.y)//祖先相同，表示原来集合存在这两个点。
    //再画上去以后就形成环
    {
      cout<<i<<endl;
      flag=true;//形成环，输出步骤
    }
    else
      join(a,b);//开始合并
  }
  if(!flag)
    cout<<"draw"<<endl;
  return 0;
}
```

课前练习

```
1. #include <iostream>
#include <string>
using namespace std;
int a[9];
int ans,i,t,len;
string s;
int main()
{
    cin>>t;
    a[0]=1;
    for(i=1; i<=8; i++)
        a[i]=a[i-1]*t;
    cin>>s;
    len=s.length();
    ans=0;
```

```
    for(i=len-1; i>=0; i--)
    {
        if(s[i]=='0')
            ans=ans+a[len-i-1];
    }
    cout<<ans<<endl;
    return 0;
}
```

输入：

5 10010

输出：_____

2. [2017]（切割绳子）有 n 条绳子，每条绳子的长度已知且均为正整数。绳子可以以任意正整数长度切割，但不可以连接。现在要从这些绳子中切割出 m 条长度相同的绳段，求绳段的最大长度是多少。

输入：第一行是一个不超过 100 的正整数 n，第二行是 n 个不超过 106 的正整数，表示每条绳子的长度，第三行是一个不超过 108 的正整数 m。

输出：绳段的最大长度，若无法切割，输出 Failed。

```
#include <iostream>
using namespace std;
int n, m, i, lbound, ubound, mid, count;
int len[100]; // 绳子长度
int main() { cin >> n; count = 0;
for (i = 0; i < n; i++) { cin >> len[i];
  _____(1)_____;
}
cin >> m;
if ( _____(2)_____ ) {
cout << "Failed" << endl;
return 0;
}
lbound = 1;
ubound = 1000000;
while (_____(3)_____) {
mid =_____(4)_____ ;
count = 0;
for (i = 0; i < n; i++)
_____(5)_____;
if (count < m) ubound = mid - 1;
else
    lbound = mid;
}
cout << lbound << endl; return 0;
}
```

习题

1.
```
#include <iostream>
using namespace std;
const int n1 = 4, n2 = 5;
int main()
{
    int max, s, i, j, k, t;
    int a[n1 + 1][n2 + 1];
    for (i = 1; i <= n1; i++)
        for (j = 1; j <= n2; j++)
            cin >> a[i][j];
    s = 0;
    for (j = 1; j <= n2; j++)
    {
        max = 0;
        for (i = 1; i <= n1; i++)
        {
            if (max < a[i][j])
                max = a[i][j];
        }
        s = s + max;
    }
    cout << s << endl;
    return 0;
}
```

输入：

1 2 3 4 5

6 9 13 5 7

11 12 8 15 14

21 20 18 16 17

输出：_____

2.[2007] （棋盘覆盖问题）在一个$2k \times 2k$ 个方格组成的棋盘中恰有一个方格与其他方格不同（图中标记为-1 的方格），称之为特殊方格。现用L 型（占3 个小格）纸片覆盖棋盘上除特殊方格的所有部分，各纸片不得重叠，于是，用到的纸片数恰好是$(4k-1)/3$。在下表给出的一个覆盖方案中，k=2，相同的3个数字构成一个纸片。

下面给出的程序是用分治法设计的，将棋盘一分为四，依次处理左上角、右上角、左下角、右下角，递归进行。请将程序补充完整。

2 2 3 3

2 -1 1 3

4 1 1 5

4 4 5 5

```
#include <iostream.h>
#include <iomanip.h>
int board[65][65],tile; // tile 为纸片编号
void chessboard(int tr,int tc,int dr,int dc,int size)// dr,dc 依次为特殊方格的行、列号
{int t,s;
if (size==1)
1                    ;
t=tile++;
s=size/2;
if( 2                  )
chessboard(tr,tc,dr,dc,s);
else
{board[tr+s-1][tc+s-1]=t;
3                  ;
}
if(dr<tr+s && dc>=tc+s)
chessboard(tr,tc+s,dr,dc,s);
else
{board[tr+s-1][tc+s]=t;
4                      ;
}
if(dr>=tr+s && dc<tc+s)
chessboard(tr+s,tc,dr,dc,s);
else
{board[tr+s][tc+s-1]=t;
   5                    ;
}
if(dr>=tr+s && dc>=tc+s)
chessboard(tr+s,tc+s,dr,dc,s);
else
{board[tr+s][tc+s]=t;
 6                      ;
}
}
void prt1(int b[][65],int n)
{int i,j;
for(i=1;i<=n;i++)
{for(j=1;j<=n;j++)
cout<<setw(3)<<b[i][j];
cout<<endl;;
}
}
void main()
```

```
{int size,dr,dc;
cout<<"input size(4/8/16/64):"<<endl;
cin>>size;
cout<<"input the position of special block(x,y):"<<endl;
cin>>dr>>dc;
board[dr][dc]=-1;
tile++;
chessboard(1,1,dr,dc,size);
prt1(board,size);
}
```

3[2903] 你现在你需要编制家谱，为了将问题简化，你将得到一些亲戚关系的信息，如 Marry 和 Tom 是亲戚，Tom 和 Ben 是亲戚，等等。从这些信息中，你可以推出 Marry 和 Ben 是亲戚。请写一个程序，对于我们的关于亲戚关系的提问，以最快的速度给出答案。

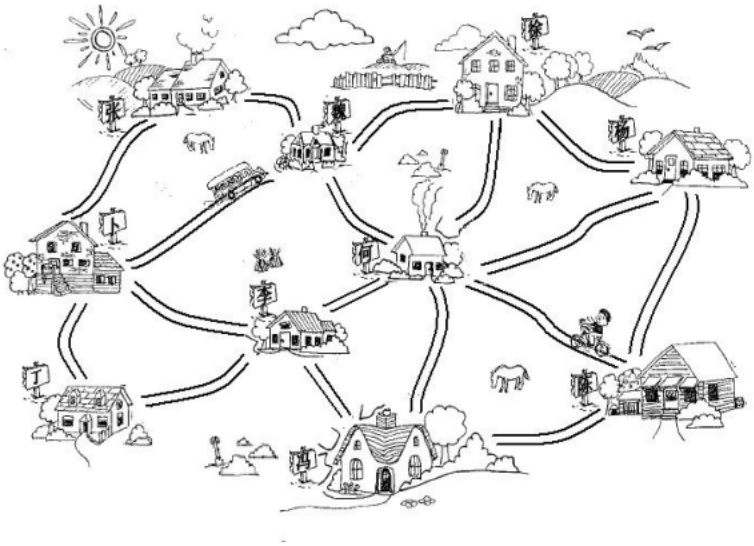
输入由两部分组成。第一部分以 N，M 开始。N 为问题涉及的人的个数(1≤N≤20000)。这些人的编号为 1,2,3,..., N。下面有 M 行(1≤M≤1000000)，每行有两个数 ai,biai,bi ，表示已知 aiai 和 bibi 是亲戚。
第二部分以 Q 开始。以下 Q 行有 Q 个询问(1≤ Q ≤1000000)，每行为 ci,dici,di ，表示询问 cici 和 didi 是否为亲戚。

```
#include<iostream>
 #include<cstdio>
   using namespace std;
   #define maxn 20001
   int father[maxn];
   int m,n,i,x,y,q;

   int find(int x)                //用递归的实现
   {
      if (__1_____________________)
      else return x;
   }
   void unionn(int r1,int r2)
   {
      ___2_____________
   }
int main()
   {
      cin >> n >> m;
      for (i = 1; i <= n; i++)
         father[i] = i;          //建立新的集合，其仅有的成员是 i
      for (i = 1; i <= m; i++)
      {
         scanf("%d%d",&x,&y);
         int r1 = ___3_____________
```

```
    int r2 = 4
    if ( 5 )
  }
  cin >> q;
  for (i = 1; i <= q; i++)
  {
    scanf("%d%d",&x,&y);
    if (find(x) == find(y)) printf("Yes\n");
      else printf("No\n");
  }
  return 0;
}
```

第 11 讲 图论基础

图 G（Graph）由两个集合 V（Vertex）和 E（Edge）组成，记为 G=(V,E)，其中 V 是顶点的有限集合，记为 V(G)，E 是连接 V 中两个不同顶点（顶点对）的边的有限集合，记为 E(G)。

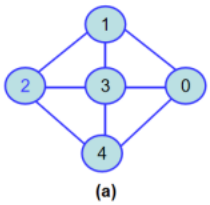


11.1 基本术语

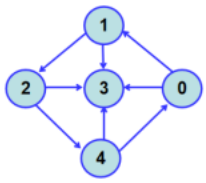
1. **端点和邻接点**

在一个无向图中，若存在一条边(i，j)，则称顶点 i 和顶点 j 为此边的两个端点，并称它们互为邻接点。

在一个有向图中，若存在一条边<i，j>，则称此边是顶点 i 的一条出边，同时也是顶点 j 的一条入边；称顶点 i 和顶点 j 分别为此边的起始端点（简称为起点）和终止端点（简称终点）；称顶点 i 和顶点 j 互为邻接点。



(a)



(b)

2. **顶点的度、入度和出度**

在无向图中，顶点所具有的边的数目称为该顶点的度。在有向图中，以顶点 i 为终点的入边的数目，称为该顶点的入度。以顶点 i 为始点的出边的数目，称为该顶点的出度。一个顶点的入度与出度的和为该顶点的度。

若一个图中有 n 个顶点和 e 条边，每个顶点的度为 di（1≤i≤n），则有：

$$e=\frac{1}{2}\sum_{i=1}^{n} d_i$$

例 1 一个无向连通图中有 16 条边，所有顶点的度均小于 5，度为 4 的顶点有 3 个，度为 3 的顶点有 4 个，度为 2 的顶点有 2 个，则该图有_____个顶点。

A.10　　B.11　　C.12　　D.13

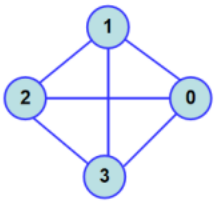
解：设该图有 n 个顶点，图中度为 i 的顶点数为 ni（0≤i≤4），显然 n0=0，n=3+4+2+n1+n0=9+n1，而度之和=4×3+3×4+2×2+n1=28+n1，而度之和=2e=32，所以有 28+n1=32，得 n1=4，n=9+n1=13。本题答案

为 D。

3. 完全图

若无向图中的每两个顶点之间都存在着一条边，有向图中的每两个顶点之间都存在着方向相反的两条边，则称此图为完全图。

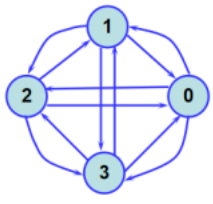
显然，完全无向图包含有条边，完全有向图包含有 n(n-1)条边。例如，图(a)所示的图是一个具有 4 个顶点的完全无向图，共有 6 条边。图(b)所示的图是一个具有 4 个顶点的完全有向图，共有 12 条边。



(a)

4. 稠密图、稀疏图

当一个图接近完全图时，则称为稠密图。相反，当一个图含有较少的边数（即当 e<<n(n-1)）时，则称为稀疏图。



(b)

5. 子图

设有两个图 G=(V，E)和 G’=(V’，E’)，若 V’是 V 的子集，即 V’ V，且 E’是 E 的子集，即 E’ E，则称 G’是 G 的子图。例如图(b)是图(a)的子图，而图(c)不是图(a)的子图。

6. 路径和路径长度

在一个图 G=(V，E)中，从顶点 i 到顶点 j 的一条路径是一个顶点序列(i,i1,i2,…,im,j)，若此图 G 是无向图，则边(i,i1)，(i1,i2)，…，(im-1,im)，(im,j)属于 E(G)；若此图是有向图，则<i,i1>，<i1,i2>，…，<im-1,im>，<im,j>属于 E(G)。

路径长度是指一条路径上经过的边的数目。若一条路径上除开始点和结束点可以相同外，其余顶点均不相同，则称此路径为简单路径。例如，有图中，(0，2，1)就是一条简单路径，其长度为 2。

7. 回路或环

若一条路径上的开始点与结束点为同一个顶点，则此路径被称为回路或环。开始点与结束点相同的简单路径被称为简单回路或简单环。

例如，右图中，(0，2，1，0)就是一条简单回路，其长度为 3。

8. 连通、连通图和连通分量

在无向图 G 中，若从顶点 i 到顶点 j 有路径，则称顶点 i 和 j 是连通的。

若图 G 中任意两个顶点都连通，则称 G 为连通图，否则称为非连通图。

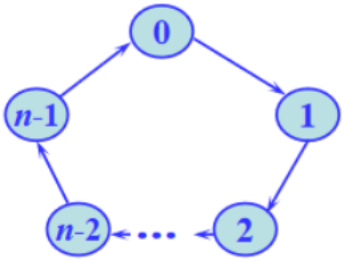
无向图 G 中的极大连通子图称为 G 的连通分量。显然，任何连通图的连通分量只有一个，即本身，而非连通图有多个连通分量。

9. 强连通图和强连通分量

在有向图 G 中，若从顶点 i 到顶点 j 有路径，则称从顶点 i 到 j 是连通的。

若图 G 中的任意两个顶点 i 和 j 都连通，即从顶点 i 到 j 和从顶点 j 到 i 都存在路径，则称图 G 是强连通图。

有向图 G 中的极大强连通子图称为 G 的强连通分量。显然，强连通图只有一个强连通分量，即本身，非强连通图有多个强连通分量。



10. 权和网

图中每一条边都可以附有一个对应的数值，这种与边相关的数

值称为权。权可以表示从一个顶点到另一个顶点的距离或花费的代价。边上带有权的图称为带权图，也称作网。

例 2　有 n 个顶点的有向强连通图最多需要多少条边？最少需要多 少条边？

解：有 n 个顶点的有向强连通图最多有 n(n-1)条边（构成一个有向完全图的情况）；最少有 n 条边（n 个顶点依次首尾相接构成一个环的情况）。

11.2 图的存储结构

1　邻接矩阵存储方法

邻接矩阵是表示顶点之间相邻关系的矩阵。设 G=(V，E)是具有 n（n＞0）个顶点的图，顶点的顺序依次为 0～n-1，则 G 的邻接矩阵 A 是 n 阶方阵，其定义如下：

（1）如果 G 是无向图，则：　A[i][j]=1：若(i,j)∈E(G)　0:其他

（2）如果 G 是有向图，则：　A[i][j]=1：若<i,j>∈E(G)　0:其他

（3）如果 G 是带权无向图，则：　A[i][j]= wij ： 若 i≠j 且(i,j)∈E(G)　0：i=j　∞：其他

（4）如果 G 是带权有向图，则：　A[i][j]= wij ： 若 i≠j 且<i,j>∈E(G)　0：i=j　∞：其他

主要特点：一个图的邻接矩阵表示是唯一的。

2 邻接表存储方法

图的邻接表存储方法是一种顺序分配与链式分配相结合的存储方法。在表示含 n 个顶点的图的邻接表中，每个顶点建立一个单链表，第 i（0≤i≤n-1）个单链表中的结点表示依附于顶点 i 的边（对有向图是以顶点 i 为尾的边）。每个单链表上附设一个表头结点，将所有表头结点构成一个表头结点数组。边结点（或表结点）和表头结点的结构如下：

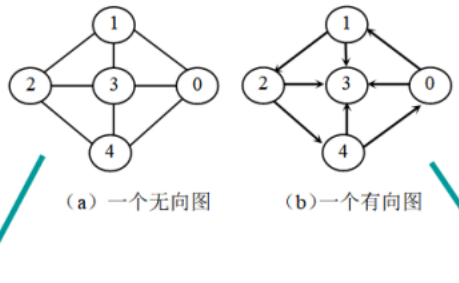
边结点（或表结点）

adjvex	nextarc	weight

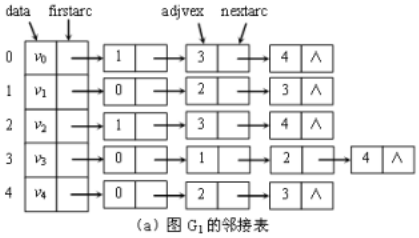
表头结点

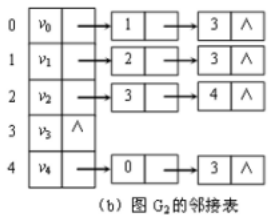
data	firstarc

其中，边结点由三个域组成，adjvex 指示与顶点 i 邻接的顶点的编号，nextarc 指示下一条边的结点，weight 存储与边相关的信息，如权值等。表头结点由两个域组成，data 存储顶点 i 的名称或其他信息，firstarc 指向顶点 i 的链表中第一个边结点。



邻接表主要特点：一个图的邻接表表示是不唯一的。这是因为在每个顶点对应的单链表中，各边结点的链接次序可以是任意的，取决于建立邻接表的算法以及边的输入次序。



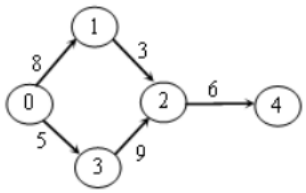


11.3 图的搜索

深度优先搜索遍历的过程是：

（1）从图中某个初始顶点 v 出发，首先访问初始顶点 v。

（2）选择一个与顶点 v 相邻且没被访问过的顶点 w 为初始顶点，再



从 w 出发进行深度优先搜索，直到图中与当前顶点 v 邻接的所有顶点都被访问过为止。

例如，对于以下有向图：从顶点 0 开始进行深度优先遍历，可以得到如下访问序列：0 1 2 4 3，0 1 3 2 4。

而 0 1 3 2 4 不是深度优先遍历序列。

11.4 案例讲解

1 [3310] 用邻接矩阵存储有向图

用邻接矩阵存储有向图，并输出各顶点的出度和入度。

输入

输入描述：

输入文件中包含多个测试数据，每个测试数据描述了一个有向图。每个测试数据的第一行为一个正整数 n (1≤n≤100)，表示该有向图的顶点数目，顶点的序号从 1 开始计起。接下来包含若干行，每行为两个正整数，用空格隔开，分别表示一条边的起点和终点。每条边出现一次且仅一次，图中不存在环和重边。0 0 代表该测试数据的结束。输入数据最后一行为 0，表示输入数据结束。

输出

输出描述：

对输入文件中的每个有向图，输出两行：第 1 行为 n 个正整数，表示每个顶点的出度；第 2 行也为 n 个正整数，表示每个顶点的入度。每两个正整数之间用一个空格隔开，每行的最后一个正整数之后没有空格。

样例输入	样例输出
7 1 2 2 3 2 5 2 6 3 5 4 3 5 2 5 3 6 5 0 0 0	1 3 1 1 2 1 0 0 2 3 0 3 1 0

写法 1：

```
#include <stdio.h>
#include <string.h>
#define MAXN 100
int Edge[MAXN][MAXN];  //邻接矩阵
int main( )
{
  int i, j;  //循环变量
  int n;  //顶点个数
  int u, v;      //边的起点和终点
```

写法 2：

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
int a[105][105];
int main(){
  int n;
  while(~scanf("%d",&n)&&n!=0){
    memset(a,0,sizeof (a));
```

```
    int od, id; //顶点的出度和入度
    while( 1 )
    {
        scanf( "%d", &n );   //读入顶点个数 n
        if( n==0 )   break;
        memset( Edge, 0, sizeof(Edge) );
        while( 1 )
        {
            scanf( "%d%d", &u, &v );//读入边的起点和终点
            if( u==0 && v==0 )   break;
            Edge[u-1][v-1] = 1; //构造邻接矩阵
        }
        for( i=0; i<n; i++ ) //求各顶点的出度
        {
            od = 0;
            for( j=0; j<n; j++ )   od += Edge[i][j];
            if(i==0)   printf( "%d", od );
            else   printf( " %d", od );
        }
        printf( "\n" );
        for( i=0; i<n; i++ ) //求各顶点的入度
        {
            id = 0;
            for( j=0; j<n; j++ )   id += Edge[j][i];
            if(i==0)   printf( "%d", id );
            else   printf( " %d", id );
        }
        printf( "\n" );
    }
    return 0;
}
```

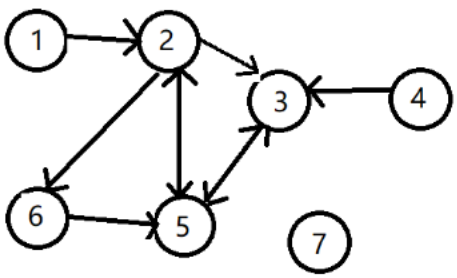
```
    int u,v;
    while(~scanf("%d %d",&u,&v)&&(u+v)!=0){
        a[u][v]=1;
    }
    for(int i=1;i<=n;i++){
        int tmp=0;
        for(int j=1;j<=n;j++){
            tmp=tmp+a[i][j];
        }
        if(i!=1) printf(" ");
        printf("%d",tmp);
    }
    printf("\n");
    for(int i=1;i<=n;i++){
        int tmp=0;
        for(int j=1;j<=n;j++){
            tmp=tmp+a[j][i];
        }
        if(i!=1) printf(" ");
        printf("%d",tmp);
    }
    printf("\n");

    }
    return 0;
}
```

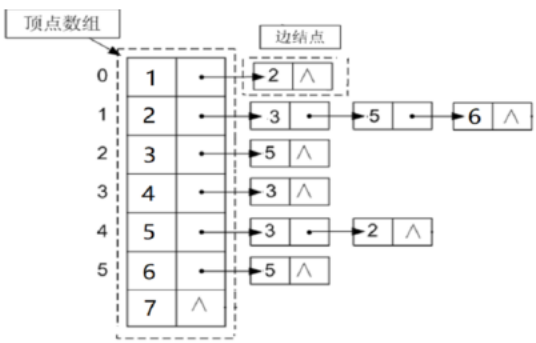
2 [3311] 用邻接表存储有向图

用邻接矩阵存储有向图，并输出各顶点的出度和入度。

样例输入

```
7
1 2
2 3
2 5
2 6
3 5
4 3
```





5 2
5 3
6 5
0 0
0
样例输出
1 3 1 1 2 1 0
0 2 3 0 3 1 0

```
#include<bits/stdc++.h>
using namespace std;
int head1[102],head2[102],next1[102],next2[102],tot;
void add(int f,int t)
{
  next1[tot]=head1[f];
  head1[f]=tot;
  next2[tot]=head2[t];
  head2[t]=tot;
  tot++;
}
int main()
{
  int n,i;
  while(cin>>n)
  {
    tot=1;
    int a=1,b=1;
    if(n==0) break;
    memset(head1,0,sizeof(head1));
    memset(head2,0,sizeof(head2));
    memset(next1,0,sizeof(next1));
    memset(next2,0,sizeof(next2));
    cin>>a>>b;
    while(a!=0||b!=0)
    {
      add(a,b);
      cin>>a>>b;
    }
    int od,id;
    for(i=1;i<=n;i++)
    {
      od=0;
      for(int j=head1[i];j;j=next1[j])
      {
```

```
            od++;
        }
        if(i==1) cout<<od;
        else cout<<" "<<od;
    }
    cout<<endl;
    for(i=1;i<=n;i++)
    {
        id=0;
        for(int j=head2[i];j;j=next2[j])
        {
            id++;
        }
        if(i==1) cout<<id;
        else cout<<" "<<id;
    }
    cout<<endl;
  }
}
```

3 [3311]链式前向星

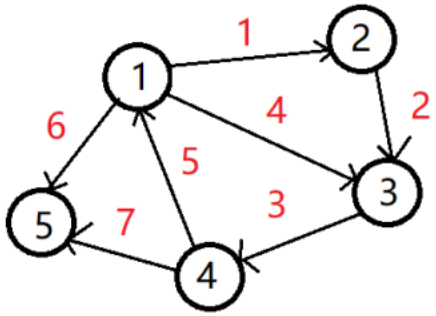
前向星是以**存储边的方式来存储图**，先将边读入并存储在连续的数组中，然后按照边的起点进行排序，这样数组中起点相等的边就能够在数组中进行连续访问了。它的优点是实现简单，容易理解，缺点是需要在所有边都读入完毕的情况下对所有边进行一次排序，带来了时间开销，实用性也较差，只适合离线算法

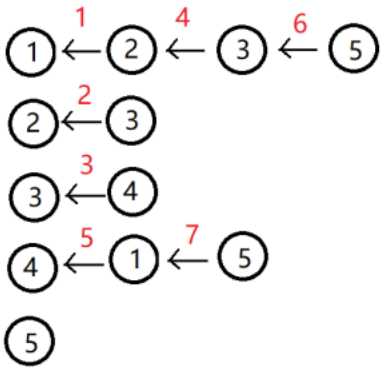
链式前向星和邻接表类似，也是链式结构和线性结构的结合，每个结点 i 都有一个链表，链表的所有数据是从 i 出发的所有边的集合（对比邻接表存的是顶点集合），边的表示为一个四元组(u, v, w, next)，其中(u, v)代表该条边的有向顶点对，w 代表边上的权值，next 指向下一条边。

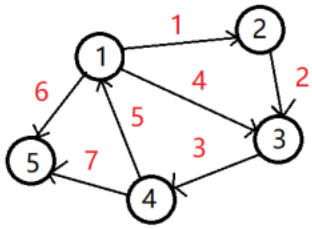
具体的，我们需要一个边的结构体数组 edge[MAXM]，MAXM 表示边的总数，所有边都存储在这个结构体数组中，并且用 head[i]来指向 i 结点的第一条边。

链式前向星其实就是静态建立的邻接表，时间效率为 O（m），空间效率也为 O（m）。遍历效率也为 O（m）。对于下面的数据，第一行 5 个顶点，7 条边。接下来是边的起点，终点和权值。也就是边 1 -> 2 权值为 1。

```
5 7
1 2 1
2 3 2
3 4 3
1 3 4
4 1 5
1 5 6
4 5 7
```







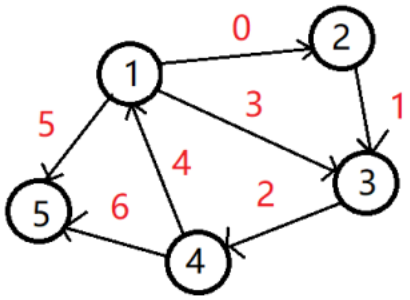
我们先对上面的 7 条边进行编号第一条边是 0 以此类推编号【0~6】，
然后我们要知道两个变量的含义：

Next，表示与这个边起点相同的上一条边的编号。

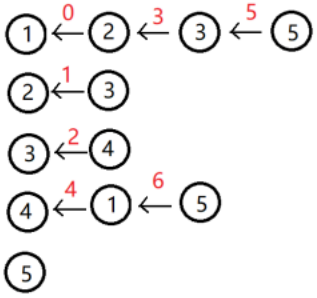
head[i]数组，表示以 i 为起点的最后一条边的编号。

head 数组一般初始化为-1，

为什么是 -1 后面会讲到。加边函数是这样的：



```
void add_edge(int u, int v, int w)//加边，u 起点，v 终点，w 边权
{
    edge[cnt].to = v; //终点
    edge[cnt].w = w; //权值
    edge[cnt].next = head[u];
        //以 u 为起点上一条边的编号，
        //也就是与这个边起点相同的上一条边的编号
    head[u] = cnt++;//更新以 u 为起点上一条边的编号
}
```



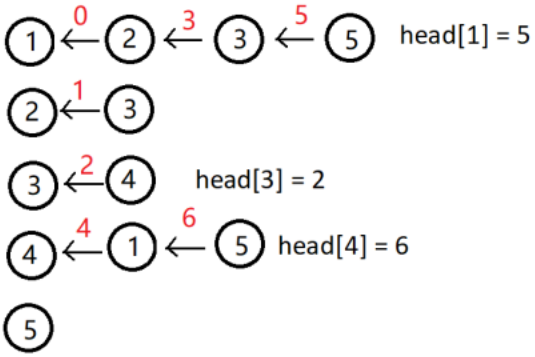
我们只要知道 next，head 数组表示的含义，

根据上面的数据就可以写出下面的过程：

对于 1 2 1 这条边：edge[0].to = 2;　edge[0].next = -1;　head[1] = 0;
对于 2 3 2 这条边：edge[1].to = 3;　edge[1].next = -1;　head[2] = 1;
对于 3 4 3 这条边：edge[2].to = 4;　edge[2].next = -1;　head[3] = 2;
对于 1 3 4 这条边：edge[3].to = 3;　edge[3].next = 0;　head[1] = 3;
对于 4 1 5 这条边：edge[4].to = 1;　edge[4].next = -1;　head[4] = 4;
对于 1 5 6 这条边：edge[5].to = 5;　edge[5].next = 3;　head[1] = 5;
对于 4 5 7 这条边：edge[6].to = 5;　edge[6].next = 4;　head[4] = 6;

head[i]数组，表示以 i 为起点的最后一条边的编号。

Next，表示与这个边起点相同的上一条边的编号



edge[0].next = -1;
edge[1].next = -1;
edge[2].next = -1;
edge[3].next = 0;
edge[4].next = -1;
edge[5].next = 3;
edge[6].next = 4;

```
#include<bits/stdc++.h>
using namespace std;
const int MAXN=100;
struct edge{
   int fr,to,nxt;
}ed[MAXN];
int head[MAXN],tot;
int indeg[MAXN];
void addedge(int a,int b)
{
   ++tot;
   ed[tot].to=b;
   ed[tot].fr=a;
   ed[tot].nxt=head[a];
   head[a]=tot;
}
int main(){
   int n,u,v;
   while(cin>>n&&n!=0){
      tot=0;
      for(int i=1;i<=n;++i)
      head[i]=-1,indeg[i]=0;
      while(cin>>u>>v){
         if(u==v&&v==0)break;
         addedge(u,v);
      }
      int sum1=0;
      for(int i=1;i<=n;i++){
         sum1=0;
         for(int j=head[i];j!=-1;j=ed[j].nxt)
         {
```

```
            int t=ed[j].to;
            indeg[t]++;
            sum1++;
        }
        cout<<sum1<<" ";
    }
    cout<<endl;
    for(int i=1;i<=n;i++){
        cout<<indeg[i]<<" ";
    }
    cout<<endl;
  }
  return 0;
}
```

4 [3365]: 图的深度优先搜索

```
#include <bits/stdc++.h>
using namespace std;
int s[101][101],n,a[101];
void dfs(int x){
  a[x]=1;
  printf("%d ",x);
  for(int i=0;i<n;i++){
    if(s[x][i]&&a[i]==0){
      dfs(i);
    }
  }
}
int main()
{
  cin>>n;
  for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
      cin>>s[i][j];
    }
  }
  for(int i=0;i<n;i++){
    if(a[i]==0){
      dfs(i);
    }
  }
  return 0;
}
```

5 [3366]图的广度优先搜索

```
#include<bits/stdc++.h>
#define INF 0x3f3f3f3f
#define maxn 100
using namespace std;
int n, sum, a[maxn][maxn], book[maxn];
int main()
{
  queue<int> que;
  scanf("%d", &n);
  for(int i = 1 ; i <= n ; ++ i){
    for(int j = 1 ; j <= n ; ++ j){
      scanf("%d", &a[i][j]);
    }
  }
  book[1] = 1;
  que.push(1);
  while(!que.empty()){
    int rec = que.front();
    cout<<rec-1<<" ";
    que.pop();
    for(int i = 1 ; i <= n ; ++ i){
      if(a[rec][i] == 1 && !book[i]){
        que.push(i);
        book[i] = 1;
      }
    }
  }
  return 0;
}
```

6[2923]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。

输入

第 1 行:一个整数 F(1≤F≤1024)，表示栅栏的数目;

第 2 到 F+1 行:每行两个整数 i,j(1≤=i,j≤500)表示这条栅栏连接 i 与 j 号顶点。

输出

输出应当有 F+1 行，每行一个整数，依次表示路径经过的顶点号。注意数据可能有多组解，但是只有上面题目要求的那一组解是认为正确的。

样例输入	样例输出
9	1
1 2	2
2 3	3
3 4	4
4 2	2
4 5	5
2 5	4
5 6	6
5 7	5
4 6	7

思路：

```
#include <bits/stdc++.h>
using namespace std;
int temp[5000], edge[505][505], k;
int maxn, minn;
void dfs(int v)
{
  for(int i=minn;i<=maxn;i++)
    if( edge[v][i])
    {
        edge[v][i]--;
        edge[i][v]--;
        dfs(i);
    }
  temp[k++]=v;
}
int main()
{
  int first_point, second_point, i, edge_num;
  while(cin>> edge_num) //输入栅栏数
  {
    memset(edge,0,sizeof(edge));
    memset(temp,0,sizeof(temp));
    minn=505;
```

```
    maxn=0;
    k=0;
    for(i =0; i < edge_num; i ++)  //每个栅栏链接的节点
    {
        cin>>first_point>>second_point;
        edge[first_point][second_point] ++;//构建图
        edge[second_point][first_point] ++;
        edge[first_point][0] ++;
        edge[second_point][0] ++;
        minn = min(minn,min(first_point,second_point));//保留最小节点标号
        maxn = max(maxn,max(first_point,second_point));//保留最大节点标号
    }
    for(i = minn; i <= maxn; i ++)
      if(edge[i][0]%2)//如果有且仅有 2 个点的度为奇数，那么它存在一条欧拉路；如果超过 2 个点的度为奇数，那么它就不存在欧拉路了。
      {
        dfs(i);
        break;
      }
    if(i==maxn+1)    dfs(1);
    for(int j=k-1;j>=0;j--)
        printf("%d\n",temp[j]);
  }
  return 0;
}
```

课前练习

1 [2005]判断质数
题目描述:给出一个正整数，判断这个数是否是质数。
输入:一个正整数n(1 ≤ n ≤ 10000)。
输出:如果n是质数，输出”YES”；否则，输出”NO”。
输入样例:10
输出样例:NO
程序：

```
#include <stdio.h>
int main()
{
int ①_____ ;
scanf("%d", &n);
if (n == 2) puts( ②_____);
else if ( ③_____ || n % 2 == 0) puts("NO");
    else
     {
```

```
        i = 3;
        while (i * i <= n)
        {
          if ( ④_____ )
          {
            puts("NO"); return 0;
          }
        i = i + 2;
        }
    puts("YES");
    }
return 0;
}
```

2 [2005] 木材加工

木材厂有一些原木，现在想把这些木头切割成一些长度相同的小段木头，需要得到的小段的数目是给定的。当然，我们希望得到的小段越长越好，你的任务是计算能够得到的小段木头的最大长度。木头长度的单位是cm。原木的长度都是正整数，我们要求切割得到的小段木头的长度也是正整数。

输入:

第一行是两个正整数N和K(1 ≤ N ≤ 10000，1 ≤ K ≤ 10000)，N是原木的数目，K是需要得到的小段的数目。接下来的N行，每行有一个1到10000之间的正整数，表示一根原木的长度。

输出:

输出能够切割得到的小段的最大长度。如果连1cm长的小段都切不出来，输出”0”。

输入样例:

```
3 7
232
124
456
```

输出样例:

```
114
```

程序：

```
#include <stdio.h>
int n, k, len[10000];
int isok(int t)
{
  int num = 0, i;
  for (i = 0; i < n; i++)
  {
    if (num >= k) break;
    num = ____①____;
  }
if ( ____②____ ) return 1;
else return 0;
}
```

```
int main() {
int i, left, right, mid;
scanf("%d%d", &n, &k);
right = 0;
for (i = 0; i < n; i++)
{
    scanf("%d", &(len[i]));
    if (right < len[i]) right = len[i];
}
right++;
③_____ ;
while (_____④_____ < right)
{
    mid = (left + right) / 2;
    if ( _____⑤_____ ) right = mid;
    else left = mid;
}
printf ("%d\n", left);
return 0;
}
```

习题

1.[2013]

```
#include <iostream>
using namespace std;
int main()
{
    const int SIZE = 100;
    int n, f, i, left, right, middle, a[SIZE];
    cin>>n>>f;
    for (i = 1; i <= n; i++)
    cin>>a[i]; left = 1;
    right = n;
    do {
        middle = (left + right) / 2;
        if (f <= a[middle])
            right = middle;
        else
            left = middle + 1;
    } while (left < right);
    cout<<left<<endl;
    return 0;
```

}输入：
12 17
2 4 6 9 11 15 17 18 19 20 21 25
输出：_____

2．[2010]

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

int main()
{
	string s;
	char m1, m2;
	int i;

	getline(cin, s);
	m1 = ' ';
	m2 = ' ';
	for (i = 0; i < s.length(); i++)
		if (s[i] > m1) {
			m2 = m1;
			m1 = s[i];
		}
		else if (s[i] > m2)
			m2 = s[i];
	cout<<int(m1)<<' '<<int(m2)<<endl;
	return 0;
}
```

输入：Expo 2010 Shanghai China
输出：_____

3 [3310] 用邻接矩阵存储有向图
用邻接矩阵存储有向图，并输出各顶点的出度和入度。
输入
输入描述：
输入每个测试数据描述了一个有向图。每个测试数据的第一行为一个正整数 n (1≤n≤100)，表示该有向图的顶点数目，顶点的序号从 1 开始计起。接下来包含若干行，每行为两个正整数，用空格隔开，分别表示一条边的起点和终点。每条边出现一次且仅一次，图中不存在环和重边。0 0 代表该测试数据的结束。输入数据最后一行为 0，表示输入数据结束。
对输入文件中的每个有向图，输出两行：第 1 行为 n 个正整数，表示每个顶点的出度；第 2 行也为 n 个正整数，表示每个顶点的入度。
#include <stdio.h> #include <string.h>

```
#define MAXN 100
int Edge[MAXN][MAXN];     //邻接矩阵
int main( )
{
    int i, j,n;     //循环变量//顶点个数
    int u, v, od, id;              //边的起点和终点//顶点的出度和入度
    scanf( "%d", &n );   //读入顶点个数 n
    memset( Edge, 0, sizeof(Edge) );
    while( 1 )
    {
        scanf( "%d%d", &u, &v );//读入边的起点和终点
        if(u==0 && v==0 )   break;
        ___1_____ //构造邻接矩阵
    }
    for( i=0; i<n; i++ ) //求各顶点的出度
    {
        od = 0;
        for( j=0; j<n; j++ )   od += ___2_____
        if(i==0)   printf( "%d", od );
        else   printf( " %d", od );
    }
    printf( "\n" );
    for( i=0; i<n; i++ ) //求各顶点的入度
    {
        id = 0;
        for( j=0; j<n; j++ )   id += ___3_____
        if(i==0)   printf( "%d", id );
        else   printf( " %d", id );
    }
    printf( "\n" );

    return 0;}
```

第 12 讲 最小生成树

所谓连通图 G 的“生成树（Spanning Tree）”，是 G 的包含其全部 n 个顶点的一个极小连通子图。它必定包含且仅包含 G 的 n-1 条边。生成树有可能不唯一。

当且仅当 G 满足下面 4 个条件之一（完全等价）：

① G 有 n-1 条边，且没有环；

② G 有 n-1 条边，且是连通的；

③ G 中的每一对顶点有且只有一条路径相连；

④ G 是连通的，但删除任何一条边就会使它不连通。

一个带权连通无向图 G（假定每条边上的权值均大于零）中可能有多棵生成树，每棵生成树中所有边上的权值之和可能不同；图的所有生成树中具有边上的权值之和最小的树称为图的最小生成树。

按照生成树的定义，n 个顶点的连通图的生成树有 n 个顶点、n-1 条边。因此构造最小生成树的准则有几条：

（1）必须只使用该图中的边来构造最小生成树；

（2）必须使用且仅使用 n-1 条边来连接图中的 n 个顶点，生成树一定是连通的；

（3）不能使用产生回路的边。

（4）最小生成树的权值之和是最小的，但一个图的最小生成树不一定是唯一的。

12.1 普里姆算法

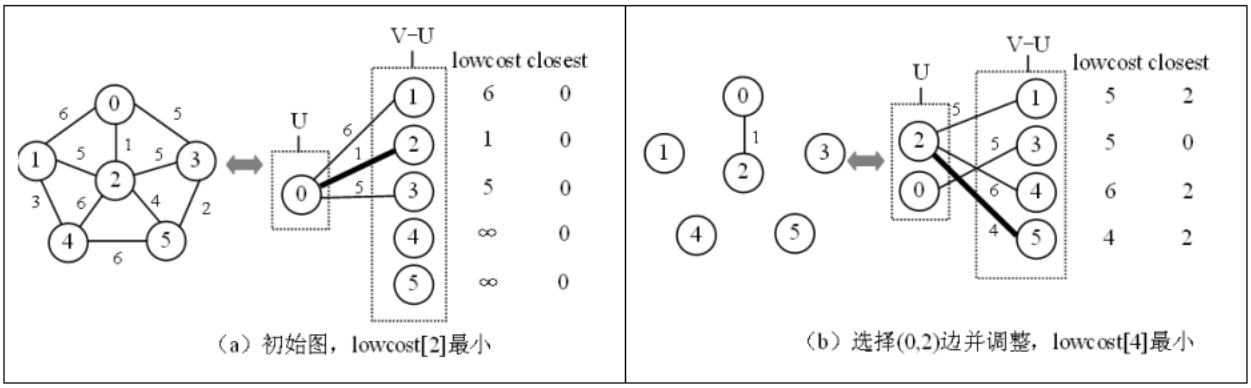
普里姆（Prim）算法是一种构造性算法。假设 G=(V,E)是一个具有 n 个顶点的带权无向连通图，T=(U,TE)是 G 的最小生成树，其中 U 是 T 的顶点集，TE 是 T 的边集，则由 G 构造从起始顶点 v 出发的最小生成树 T 的步骤如下：

（1）初始化 U={v}。以 v 到其他顶点的所有边为候选边；

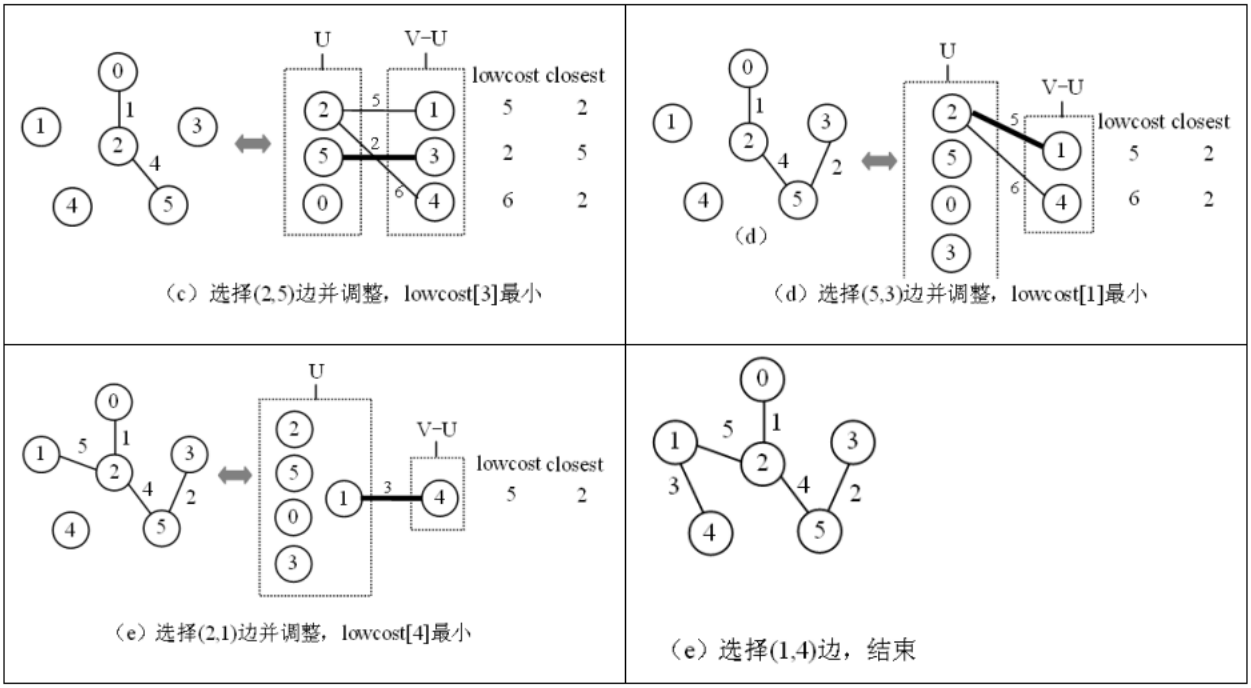
（2）重复以下步骤 n-1 次，使得其他 n-1 个顶点被加入到 U 中：

① 从候选边中挑选权值最小的边加入 TE，设该边在 V-U 中的顶点是 k，将 k 加入 U 中；

② 考察当前 V-U 中的所有顶点 j，修改候选边：若(k,j)的权值小于原来和顶点 j 关联的候选边，则用(k,j)取代后者作为候选边。



（a）初始图，lowcost[2]最小　　（b）选择(0,2)边并调整，lowcost[4]最小



规定：若 lowcost[j]=0，则表明顶点 j∈U；

若 0<lowcost[j]<∞，则顶点 j∈V-U，且顶点 j 和 U 中的顶点 closest[j]构成的边（closest[j],j）是所有与顶点 j 相邻、另一端在 U 的边中的具有最小权值的边，其最小的权值为 lowcost[j]（对于每个顶点 j∈V-U，U 中的所有顶点到顶点 j 可能有多条边，但只有一条最小边，用 closest[j]表示该顶点，lowcost[j]表示该边的权值）；若 lowcost[j]=∞，则表示顶点 j 与 closest[j]之间没有边。

对应的 Prim(v)算法如下 ：

Prim 算法时间复杂度为 $O(n^2)$，其中 n 为图的顶点个数。由于与 e 无关，所以普里姆算法特别适合于稠密图求最小生成树。

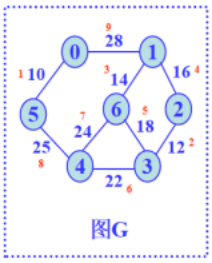
```
void prim(){
    int i,j,minn;
    int cnt=1,sum=0;
    for(i=1;i<=n;i++){
        vis[i]=0;//初始全部的点在集合A中
        cost[i]=e[1][i];
    }
    vis[1]=1;//把初始第一点添加到B集合
    while(cnt<n){
        minn=inf;
        for(i=1;i<=n;i++)
            if(vis[i]==0&&cost[i]<minn){//在集合v中找d[i]最小的权值
                minn=cost[i];
                j=i;//找到新的添加到集合u中的点
            }
        vis[j]=1;//把新的点添加到集合u中
        sum=sum+cost[j];//求和，计算最小距离
        cnt++;
        for(i=1;i<=n;i++){//更新cost[i]
            if(vis[i]==0&&e[j][i]<cost[i])
                cost[i]=e[j][i];
        }
    }
    cout<<sum<<endl;
}
```

12.2 克鲁斯卡尔算法

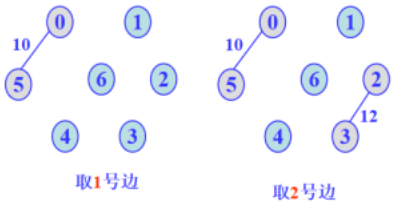
克鲁斯卡尔(Kruskal)算法是一种按权值的递增次序选择合适的边来构造最小生成树的方法。假设 G=(V,E)是一个具有 n 个顶点的带权连通无向图，T=(U,TE)是 G 的最小生成树，则构造最小生成树的步骤如下：

（1）置 U 的初值等于 V（即包含有 G 中的全部顶点），TE 的初值为空集（即图 T 中每一个顶点都构成一个分量）。

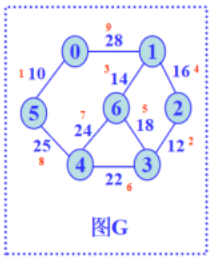
（2）将图 G 中的边按权值从小到大的顺序依次选取：若选取的边未使生成树 T 形成回路，则加入 TE；否则舍弃，直到 TE 中包含 n-1 条边为止。

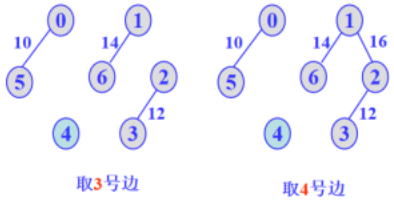


按边大小递减排序

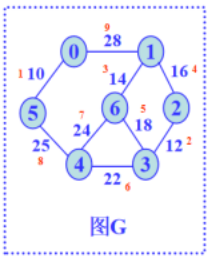


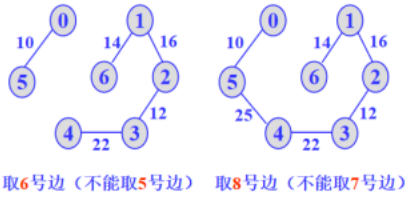
克鲁斯卡尔算法求解最小生成树的过程（1）





克鲁斯卡尔算法求解最小生成树的过程（2）





克鲁斯卡尔算法求解最小生成树的过程（3）

在实现克鲁斯卡尔算法 Kruskal()时，用一个数组 E 存放图 G 中的所有边，要求它们是按权值递增排列的，为此先从图 G 的邻接矩阵中获取所有边集 E，再采用直接插入排序法对边集 E 按权值递增排序。

通过改进可以降低该算法的时间复杂度，通常认为克鲁斯卡尔算法的时间复杂度为 O(elog2e)。由于与 n 无关，所以克鲁斯卡尔算法特别适合于稀疏图求最小生成树。

```
bool cmp (node a,node b) {
    return a.dis < b.dis;
} // 对边的权值做从小到大排序
int find_fa(int x) { //查找父节点
    if(x == fa[x]) return x;
    return fa[x] = find_fa(fa[x]);
}
int Union (int x, int y) {

    int fx = find_fa(x), fy = find_fa(y);
    if(fx == fy) return 0;
    fa[fx] = fy;
    return 1;
}
```

```
void kruskal() {
    int ans = 0;
    for(int i = 1; i <= n; i ++) fa[i] = i;
    sort(e+1, e+1+m, cmp);
    for(int i = 1; i <= m; i ++) {
        if(Union(e[i].u, e[i].v)) {
            ans += e[i].dis;
        }
    }
    printf("%d\n", ans);
}
```

12.3 案例讲解

1 [3316] 还是畅通工程

某省调查乡村交通状况，得到的统计表中列出了任意两村庄间的距离。省政府“畅通工程”的目标是使全省任何两个村庄间都可以实现公路交通（但不一定有直接的公路相连，只要能间接通过公路可达即可），并要求铺设的公路总长度为最小。请计算最小的公路总长度。

测试输入包含若干测试用例。每个测试用例的第 1 行给出村庄数目 N (< 100)；随后的 N(N-1)/2 行对应村庄间的距离，每行给出一对正整数，分别是两个村庄的编号，以及此两村庄间的距离。为简单起见，村庄从 1 到 N 编号。当 N 为 0 时，输入结束，该用例不被处理。

对每个测试用例，在 1 行里输出最小的公路总长度。

样例输入	样例输出
3 1 2 1 1 3 2 2 3 4 4 1 2 1 1 3 4 1 4 1 2 3 3 2 4 2 3 4 5 0	3 5

克鲁斯卡尔算法：

```
#include<iostream>
#include <algorithm>
using namespace std;
const int inf = 0x3f3f3f3f;
int n, m, fa[101];
struct node {
   int u, v, cost;
} e[10000 + 10];

bool cmp (node a, node b) {
    1
}

int find_fa(int x) {
   if(x == fa[x]) return x;
   return 2;
}

int Union(int x, int y) {
   int fx = 3
```

prim 算法：

```
#include<iostream>
using namespace std;
#define maxn 100
const int inf=0xffff;
int e[maxn][maxn],cost[maxn];
int vis[maxn];
int m,n,j;
// cost[i] 表示 i 节点当前加入集合中的最小花费
// 集合 u 到集合 v 的最小权值
// vis [i] 标记 i 节点是否已加入集合 u 中
// dis [i][j] 表示 i -> j 的长度
    //U：当前生成树顶点集合，
    //V：不属于当前生成树的顶点集合。
void prim(){
   int i,j,minn;
   int cnt=1,sum=0;
   for(i=1;i<=n;i++){
      1          //初始全部的点在集合 A 中
      2
   }
```

```
   Int fy = 4_____
   if (fx == fy) return 0;
   5_____
   return 1;
}

int kruskal() {
   sort(6_____);
   for (int i = 1; i <= n; i ++) fa[i] = i;
   int ans = 0;

   for (int i = 1; i <= n*(n-1)/2; i ++) {
      if (7_____) {
         ans += e[i].cost;
      }
   }
   return ans;
}

int main(){
   while(cin >> n && n) {
      for(int i = 1; i <= n*(n-1)/2; i++) {
         scanf("%d%d%d", &e[i].u, &e[i].v, &e[i].cost);
      }
      cout <<8_____ << endl;

   }

   return 0;
}
```

```
   3_____//把初始第一点添加到 B 集合
   while(cnt<n){
      minn=inf;
      for(i=1;i<=n;i++)
         if( 4_____){
            //在集合 v 中找 d[i]最小的权值
            minn=cost[i];
            j=i;//找到新的添加到集合 u 中的点
         }
      5_____//把新的点添加到集合 u 中
      sum=sum+cost[j];//求和，计算最小距离
      cnt++;
      for(i=1;i<=n;i++){//更新 cost[i]
         if( 6_____)
            cost[i]=e[j][i];
      }
   }
   cout<<sum<<endl;
}
int main(){
   while(cin>>n){
      if(n==0)break;
      m=n*(n-1)/2;
      for(int i=1;i<=n;i++)
         for(int j=1;j<=n;j++)
            if(i==j) e[i][j]=0;
            else    e[i][j]=inf;
      int u,v,w;
      for(int i=1;i<=m;i++){
         cin>>u>>v>>w;
         7_____
      }
      prim();
   }
   return 0;
}
```

2 [2910] 城市公交网建设问题

题目描述

有一张城市地图，图中的顶点为城市，无向边代表两个城市间的连通关系，边上的权为在这两个城市之间修建高速公路的造价，研究后发现，这个地图有一个特点，即任一对城市都是连通的。现在的问题是，要修建若干高速公路把所有城市联系起来，问如何设计可使得工程的总造价最少？

输入
n（城市数，1<≤n≤100）
e（边数）
以下 e 行，每行 3 个数 i,j,wiji,j,wij，表示在城市 i,j 之间修建高速公路的造价。
输出
n-1 行，每行为两个城市的序号，表明这两个城市间建一条高速公路。
注意看测试数据的输出顺序，有排序要求。

样例输入	样例输出
5 8 1 2 2 2 5 9 5 4 7 4 1 10 1 3 12 4 3 6 5 3 3 2 3 8	1 2 2 3 3 4 3 5

Prim 算法：

```
#include <iostream>
#include <algorithm>
using namespace std;
#define inf 0x3f3f3f3f
int n,m,e[110][110];//e 数组保存输入原始数据
struct Node
{
        int pre;//边的起始点
        int to;//边的终点
        int n;//权值
}dis[110],ans[110];
//dis[i]记录集合 u 的顶点到集合 v 到顶点的最短距离
int k=0;
bool cmp(Node a,Node b)
{// 先按起始点排序，再按终点排序
        if (a.pre!=b.pre) return a.pre<b.pre;
        else return a.to<b.to;
}
void prim()
{
        int book[110]={0};//记录当前点是否已经被处理
        for (int i=1;i<=n;i++)
        {//输入数据
                dis[i].n=e[1][i];//权值
```

```
            dis[i].pre=1;
            dis[i].to=i;
        }
        __1__ // 赋值为1，表示起点加入被处理点的集合
        int minn,u;
        for (int i=0;i<n-1;i++)
        {
            minn=inf;
            for (int j=1;j<=n;j++)
            {     //如果当前点为处理，且权值最小
                if (__2__)
                {
                    minn=dis[j].n;
                    u=j;
                }
            }
            if (dis[u].pre<dis[u].to)
                ans[k].pre=dis[u].pre,ans[k].to=dis[u].to;
            else ans[k].pre=dis[u].to,ans[k].to=dis[u].pre;
            k++;//
            __3__book[u]=1;//把权值最小的边的节点加入 处理集合
            for (int j=1;j<=n;j++)
                if (__4__)
                {//更新dis[j]数组，记录集合u中的点到 集合v中的的距离
                    dis[j].n=e[u][j];//u->j
                    dis[j].pre=u;
                    dis[j].to=j;
                }
        }
}
int main ()
{
        cin>>n>>m;
        int u,v,w;
        memset(e,inf,sizeof(e));//初始化为极大值
        for (int i=1;i<=n;i++)
            e[i][i]=0;
        for (int i=0;i<m;i++)
        {//输入数据
            cin>>u>>v>>w;
            e[u][v]=w;
            e[v][u]=w;
        }
        prim();
```

```
        sort(ans,ans+k,cmp);
        for (int i=0;i<k;i++)
                cout<<ans[i].pre<<" "<<ans[i].to<<endl;
        return 0;
}
```

克鲁斯卡尔算法实现：

```
#include <bits/stdc++.h>
using namespace std;
#define maxn 10010
struct edge
{
        int from,to,flag,value;
        // flag 标记改变是否被选择 value 权值
}num[maxn];
int parent[110], n,m;
bool cmp1(edge x,edge y)
{//按照权值排序
        return x.value<y.value;
}
bool cmp2(edge x,edge y)
{//按照边排序
        if(x.from==y.from)
                return x.to<y.to;
        return x.from<y.from;
}
int find(int x)
{//找到父节点
        if(x==parent[x])
                return x;
        int fx=find(parent[x]);
        parent[x]=fx;
        return fx;
}
void kruskal()
{
        for(int i=1;i<=m;i++)
        {
                int fx=find(num[i].from);
                int fy=find(num[i].to);
                if(fx!=fy)
                {
                        parent[fx]=fy;
                        num[i].flag=1;//选取的边设置为 1
```

```
                }
        }
}
int main()
{
        scanf("%d%d",&n,&m);
        for(int i=1;i<=n;i++)
                parent[i]=i;//初始自己为父节点
        for(int i=1;i<=m;i++)
        {
                int x,y;
                scanf("%d%d%d",&x,&y,&num[i].value);
                num[i].from=min(x,y);
                num[i].to=max(x,y);
                num[i].flag=0;
        }
        sort(num+1,num+1+m,cmp1);//按照权值排序
        kruskal();
        sort(num+1,num+1+m,cmp2);//按照起始点排序
        for(int i=1;i<=m;i++)
        {
                if(num[i].flag)
                        printf("%d %d\n",num[i].from,num[i].to);
        }
        return 0;
}
```

3 [2937] 局域网

某个局域网内有 n(n≤100)台计算机，由于搭建局域网时工作人员的疏忽，现在局域网内的连接形成了回路，我们知道如果局域网形成回路那么数据将不停的在回路内传输，造成网络卡的现象。因为连接计算机的网线本身不同，所以有一些连线不是很畅通，我们用 f(i,j)表示 i,j 之间连接的畅通程度(f(i,j)≤1000)，f(i,j)值越小表示 i,j 之间连接越通畅，f(i,j)为 0 表示 i,j 之间无网线连接。现在我们需要解决回路问题，我们将除去一些连线，使得网络中没有回路，并且被除去网线的Σf(i,j)最大，请求出这个最大值。

第一行两个正整数 n k

接下来的 k 行每行三个正整数 i j m 表示 i,j 两台计算机之间有网线联通，通畅程度为 m。

输出

一个正整数，Σf(i,j)的最大值。

样例输入

5 5

1 2 8

1 3 1

1 5 3

2 4 5

3 4 2
样例输出
8

方法 1：

```
#include<bits/stdc++.h>
using namespace std;
int f[105];
struct node{
   int u,v,w;
}s[105];
bool cmp1(node a,node b)
{
   return a.w<b.w;
}
int find(int a)
{
   while(f[a]!=a)a=f[a];
   return a;
}
int main()
{
   int n,k,num=0,ans=0;;
   scanf("%d%d",&n,&k);
   for(int i=1;i<=n;i++)
      f[i]=i;
   for(int i=0;i<k;i++)
   {
      scanf("%d%d%d",&s[i].u,&s[i].v,&s[i].w);
      num+=s[i].w;
   }
   sort(s,s+k,cmp1);
   for(int i=0;i<k;i++)
   {
      int x1=find(s[i].u);
      int x2=find(s[i].v);
      if(x1!=x2)
      {
         f[x2]=x1;
         ans+=s[i].w;
      }
      else continue;
   }
   printf("%d",num-ans);
   return 0;
```

方法 2：

```
#include<cstdio>
#include<iostream>
#include<cstdlib>
#include<cmath>
#include<cstring>
using namespace std;
int a,b,c,i,j,k,l,m,n,inf=9999999,sum,max1;
int e[101][101],minn[101];
bool u[101];
void prim(){
   memset(minn,0x7f,sizeof(minn));
   memset(u,1,sizeof(u));//初始化为 True，表示所有点未被访问
   minn[1]=0;
   for(i=1;i<=n;i++)
   {
      k=0;                  //寻找一个与已访问的点相连的权值
      for(j=1;j<=n;j++)//最小的未被访问的点 k
         if(u[j] && minn[j]<minn[k])
            k=j;
      u[k]=false;//将 k 加入最小生成树，标记已访问
      for(j=1;j<=n;j++)//修改与 k 相连的所有未被访问的点
      if(u[j] && e[k][j]<minn[j])
      minn[j]=e[k][j];
   }
}
int main()
{
   cin>>n>>m;
   for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
         if(i==j)
            e[i][j]=0;
      else
         e[i][j]=inf;//构造邻接矩阵
   for(i=1;i<=m;i++)
   {
      cin>>a>>b>>c;
      e[a][b]=e[b][a]=c;
      max1+=c;//max 储存所有的畅通程度
```

}	}//读入数据并存入矩阵
	prim();
	
	for(i=1;i<=n;i++)
	sum+=minn[i];//累加权值
	cout<<max1-sum;
	return 0;
	}

课前练习

1【海岛建设】牛背群岛是由一大批大小接近的小岛屿组成的，涨潮时群岛中的每个小岛都会有一部分被淹入海水下面（有些特别矮小的小岛甚至整个淹入水下），退潮时这些小岛上被淹的部分又能露出了海面。为了打造特色旅游景观，政府计划在每次涨潮时被淹入水下的那些部位安装彩色灯光装置，这样在涨潮时这些部位就会在水下发出光彩夺目的景观。

现在味味已经从测绘公司得到了所有这些小岛的海拔高度，而且将这些数据进行了量化预处理。作为程序员，味味需要统计所有这些小岛中，量化高度在 2 和 5 之间（包括 2 和 5）的小岛数量总共有多少个。

下面是味味编写了一部分的程序，程序先读入牛背群岛所组成的方阵的规模 n（牛背 群岛的各个小岛刚好组成了一个 n*n 的方阵），然后以一个方阵的格式依次读入 n*n 个小 岛各自的量化高度（详细可见输入和输出样例），程序最后应能输出量化高度不小于 2 并且不大于 5 的所有小岛的总数。

输入样例:

```
5
1 2 3 5 2
7 4 2 3 6
2 1 3 6 5
1 1 5 3 4
11 9 1 4 5
```

输出样例:

```
15
```

请你帮助味味完成下面的程序。

```
#include <iostream>
using namespace std;
int main()
{
    int n, i, j, s;
    int a[101][101];
    ___①___;
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            ___②___;
    s = 0;
```

```
    for (i = 1; i <= n; i++)
        for (j = 1;    ③    ; j++)
        {
            if (a[i][j] >= 2 &&    ④    )
                ⑤    ;
        }
    cout << s;
    return 0;
}
```

2 【扫雷游戏】Windows 中的扫雷游戏是大家都熟悉的小游戏，今天，味味也设计了一个简易的扫雷游戏。味味设计的扫雷游戏功能如下：

1．程序一开始会读入扫雷的区域大小 n，表示扫雷区域有 n*n 个小方格组成，接下来会读入 n 行信息，每行有 n 个整数（每个整数可能是 0,也可能是 1），每两个整数之间用一个空格分隔。其中 0 表示所在位置的小方格内没有地雷，1 表示所在位置的小方格内有地雷（游戏开始时，扫雷中必定包含至少一个地雷）。接下来每行输入两个用空格分开的正整数 i 和 j，每一行的一对 i 和 j 表示用户用鼠标单击扫雷区域中第 i 行第 j 列位置上的小方格（就像我们 windows 中扫雷游戏一样），i 和 j 表示的位置必定在扫雷区域内。程序每输入一对 i 和 j，就马上进行相应的处理（就像我们在 windows 中鼠标单击某个小方块就会出现结果一样）。

2.程序根据读入的一对 i 和 j 的值来对扫雷区域作相应处理，具体的处理规则如下：（1）如果 i 和 j 表示的小方格内没有地雷、而且也没有被处理过（就是第 i 行第 j 列的数值是 0），那么将以该小方格为中心的一个正方形区域内所有没有地雷的小方格都赋值为-1（表示该区域的地砖被掀开），当然该正方形指的是在扫雷区域内的有效区域。如果在当前正方形区域内有一个位置号是 i1 和 j1(注意，i1!=i 并且 j1!=j)的小方格内恰好有地雷，则此地雷就被顺利扫除，将该位置标记为-2。如果该正方形区域内某些小方格已经被处理过，则对这些小方格不再做任何处理。举个例子来说明一下，假如输入信息如下左边所示，那么输出结果就如下右边所示

```
5
0 0 0 0 0                    0 0 0 0 0
0 0 1 0 0                    0 -1 -2 -1 0
0 0 0 1 0                    0 -1 -1 -2 0
0 0 0 0 0                    0 -1 -1 -1 0
0 1 0 0 0                    0 1 0 0 0
33
00
```

（2）如果 i 和 j 表示的小方格已经被处理过（就是第 i 行第 j 列的数值是-1 或者是-2），那么不作任何处理，继续去读取下一行的 i 和 j 的值。

（3）如果 i 和 j 表示的小方格刚好有地雷，并且该小主格没有被处理过（就是第 i 行和第 j 列的数值是 1），那么表示用户触雷，输出信息“GAME OVER”，程序结束。

3．如果在读入和处理 i、j 的过程中一直没有触雷，那么就一直按照位置信息处理下去，直到满足下列条件之一，就输出相应信息并结束程序：

（1）读入的 i 和 j 的值都是 0（表示用户不再在某个小方格内单击右键了），则输出 整个扫雷区域的状态（就是输出处理后来 n 行 n 列的方阵，每行中两个整数之间用一个空格分隔，末尾没有多余空格），然后程序结束。

（2）如果某次处理完后，游戏区域内所有的地雷都被扫除了，那么不必再读入一下 行的信息，直接输出

信息“YOU ARE WINNER!!”，程序结束。味味不小心把已经编好的程序误删了一些代码，请根据上面的功能要求，帮助味味把下面的程序补充完整。说明：total 变量保存一开始扫雷区中地雷的总数，ok 变量保存当前已经被扫除的地雷总数。

输入样例 1： 6 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 1 3 4 5 5 4 6 5 2	输出样例 1： GAME OVER!
输入样例 2： 5 0 0 1 1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 0 1 2 2 2 4 4 4 5 5 3 1 0 0	输出样例 2： -1 -1 -2 -2 -1 -2 -1 -1 -1 -1 -1 -1 -2 -1 -2 0 0 -1 -1 -1 0 1 -2 -1 -2

请你帮助味味完成下面的程序。

```
#include <iostream>
using namespace std;
int i, j, total, ok, x, y, n;
int a[51][51];

void pro_1()
{
  int x, y, k;
  for (x = ①; x <= i + 1; x++)
  {
    for (y = j - 1; y <= j + 1; y++)
    {
```

```
int main()
{
    cin >> n;
    total = 0;
    ok = 0;
    for (x = 1; x <= n; x++)
    {
        for (y = 1; y <= n; y++)
        {
            cin >> a[x][y];
            if (a[x][y] == 1)
                total = total + 1;
```

```
        if (x >= 1 && y >= 1 && x <= n && y <= n)
        {
            if (    ②    )
                a[x][y] = -1;
            if (a[x][y] == 1)
            {
                a[x][y] = -2;
                    ③    ;
            }

        }
      }
    }
}
```

```
            }
        }
        ④    ;
        while (i != 0 && j != 0)
        {
            if (    ⑤    )
                pro_1();
            if (total == ok)
            {
                cout << "YOU ARE WINNER!!" << endl;
                return 0;
            }
            if (a[i][j] == 1)
            {
                cout <<    ⑥    << endl;
                return 0;
            }
            cin >> i >> j;
        }
        for (x = 1; x <= n; x++)
        {
            for (y = 1; y <= n; y++)
                cout << a[x][y];
            cout << endl;
        }
        return 0;
}
```

习题

1．[2004]三角形内切圆的面积

题目描述:给出三角形三边的边长，求此三角形内切圆（如下图所示，三角形的内切圆是和三角形三边都相切的圆）的面积。输入:三个正实数 a、b、c（满足 a+b>c，b+c>a，c+a>b），表示三角形三边的边长。

输出: 三角形内切圆的面积，结果四舍五入到小数点后面 2 位。

输入样例: 3 4 5

输出样例: 3.14

程序：

```
#include <stdio.h>
#include <math.h>
int main(){
    float a, b, c, r, s, t;
    scanf("%f %f %f", &a, &b, &c);
    s = (①                    ) / 2;
```

```
    t = ②_____ (s * (s - a) * (s - b) * (s - c));
    r = t / s;
    printf(" ③_____ \n", 3.1415927 * r * ④_____ );
    return 0;
}
```

2．[2010]

```
#include <iostream>
using namespace std;
int rSum(int j)
{
    int sum = 0;
    while (j != 0) {
        sum = sum * 10 + (j % 10);
        j = j / 10;
    }
    return sum;
}
int main()
{
    int n, m, i;

    cin>>n>>m;
    for (i = n; i < m; i++)
        if (i == rSum(i))
            cout<<i<<' ';
    return 0;
}
```

输入：90 120

输出：_____

3[3316] ：某省调查乡村交通状况，得到的统计表中列出了任意两村庄间的距离。省政府“畅通工程”的目标是使全省任何两个村庄间都可以实现公路交通（但不一定有直接的公路相连，只要能间接通过公路可达即可），并要求铺设的公路总长度为最小。请计算最小的公路总长度。

输入包含若干测试用例。每个测试用例的第 1 行给出村庄数目 N (< 100)；随后的 N(N-1)/2 行对应村庄间的距离，每行给出一对正整数，分别是两个村庄的编号，以及此两村庄间的距离。为简单起见，村庄从 1 到 N 编号。

当 N 为 0 时，输入结束，该用例不被处理。

输出

对每个测试用例，在 1 行里输出最小的公路总长度。

```
#include<iostream>
#include <algorithm>
using namespace std;
const int inf = 0x3f3f3f3f;
```

```
int n, m, fa[101];
struct node {
    int u, v, cost;
} e[10000 + 10];
bool cmp (node a, node b) {
    __1_____________________
}
int find_fa(int x) {
    if(x == fa[x]) return x;
    return fa[x] = __2_____
}
int Union(int x, int y) {
    int fx = find_fa(x), fy = find_fa(y);
    if (fx == fy) return 0;
    ___3_____________________
    return 1;
}
int kruskal() {
    ___4_____________________
    for (int i = 1; i <= n; i ++) fa[i] = i;
    int ans = 0;
    for (int i = 1; i <= n*(n-1)/2; i ++) {
        if ( _____5_____) {
            ans += e[i].cost;
        }
    }
    return ans;
}
int main(){
    while(cin >> n && n) {
        for(int i = 1; i <= n*(n-1)/2; i++) {
            scanf("%d%d%d", &e[i].u, &e[i].v, &e[i].cost);
        } cout << kruskal() << endl;
 }
```

第 13 讲 最短路径

在一个不带权的图中，若从一顶点到另一顶点存在着一条路径，则称该路径长度为该路径上所经过的边的数目，它等于该路径上的顶点数减 1。由于从一顶点到另一顶点可能存在着多条路径，每条路径上所经过的边数可能不同，即路径长度不同，把路径长度最短（即经过的边数最少）的那条路径叫做最短路径，其路径长度称为最短路径长度或最短距离。

对于带权的图，考虑路径上各边上的权值，则通常把一条路径上所经边的权值之和定义为该路径的路径长度或称带权路径长度。从源点到终点可能不止一条路径，把带权路径长度最短的那条路径称为最短路径，其路径长度（权值之和）称为最短路径长度或者最短距离。

问题解法：

权值为非负的单源最短路径问题(固定源点)－Dijkstra 算法(迪克斯特拉算法)；

权值为任意值的单源最短路径问题(固定源点) －Bellman-Ford 算法(贝尔曼－福特算法)；

所有顶点之间的最短路径问题－Floyd-Warshall 算法(弗洛伊德算法)；

13.1 Dijkstra 算法

在图(a)中，考虑顶点 0 到其他顶点的最短距离是多少？

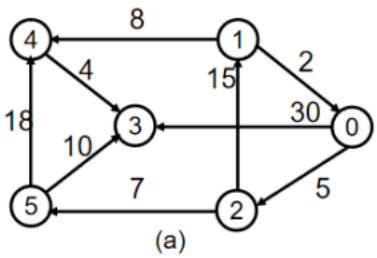
顶点 0 到顶点 1 的最短路径距离是：20

顶点 0 到顶点 2 的最短路径距离是： 5

顶点 0 到顶点 3 的最短路径距离是： 22

顶点 0 到顶点 4 的最短路径距离是： 28

顶点 0 到顶点 5 的最短路径距离是： 12



(a)

思考：这些最短距离是怎么求出来的？

为求得这些最短路径，Dijkstra 提出**按路径长度的递增次序，逐步产生最短路径的算法。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从顶点 v 到其它各顶点的最短路径全部求出为止。**

在图(a)中，考虑顶点 0 到其他顶点的最短距离是多少？

长度最短的最短路径是顶点 0 到顶点 2 的最短路径(就是顶点 0 到其他顶点的直接路径最短的路径)

长度次短的最短路径是顶点 0 到顶点 5 的最短路径(v0,v2,v5)，其中(v0,v2)就是前面求出的长度最短的最短路径。

算法思想

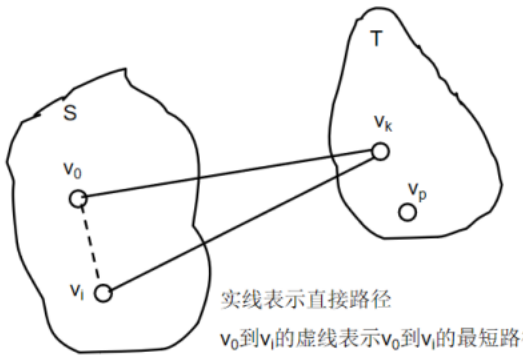
1. 设置两个顶点的集合 T 和 S：

S 中存放已找到最短路径的顶点，**初始状态时，集合 S 中只有一个顶点，即源点 v0**；

T 中存放当前还未找到最短路径的顶点；

2. **以后每求得一条最短路径(v0,…,vk)，就将 vk 加入到顶点集合 S 中，直到所有的顶点都加入到集合 S 中**，算法就结束了。

可以证明 v0 到 T 中顶点 vk 的最短路径，要么是从 v0 到 vk 的直接路径；要么是从 v0 经 S 中某个顶点 vi 再到 vk 的路径。



实线表示直接路径

v_0到v_i的虚线表示v_0到v_i的最短路径

1 初始时，S 中包含源点 v0。

2 然后不断从 T 中选取到顶点 v0 路径长度最短的顶点 u，找到后：

3 把顶点 u 加入到 S；

4 修改顶点 v0 到 T 中剩余顶点的最短路径长度值。T 中各顶点新的最短路径长度值为：min{原来的最短路径长度值，顶点 u 的最短路径长度+u 到该顶点的路径长度值}；

重复 2，直到 T 的顶点全部加入 S 为止。

在 dijkstra 算法里设置 3 个数组：

dist[n]： dist[i]表示当前找到的从源点 v0 到终点 vi 的最短路径的长度，初始状态下，dist[i]为 E[v0][i]，即邻接矩阵的第 v0 行。

S[n]： S[i]为 0 表示顶点 vi 还未加入到集合 S 中，S[i]为 1 表示 vi 已经加入到集合 S 中。初始状态下，S[v0]为 1，其余为 0，表示最初集合 S 中只有顶点 v0。

path[n]：p[i]表示在最短路径上顶点 vi 的前一个顶点号

在 dijkstra 算法里重复做以下工作：

在 dist[ ]里查找 S[i]!=1，并且 dist[i]最小的顶点 u，

将 S[u]改为 1，表示顶点 u 已经加入进来了。

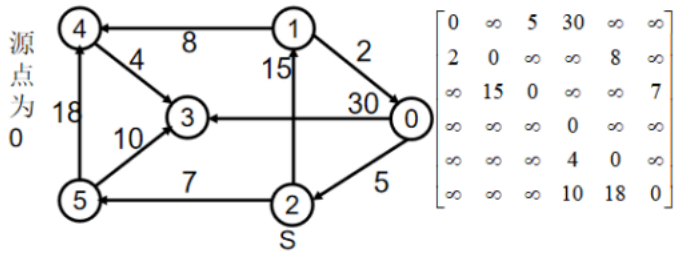
修改其它顶点的 dist[ ]及 path[ ]。当 S[k]!=1，且顶点 vk 到顶点 vu 有边(E[u][k]<MAX)，且 dist[u]+E[u][k]<dist[k]，则修改 dist[k]为 dist[u]+E[u][k]，修改 path[k]为 u。

递推公式：

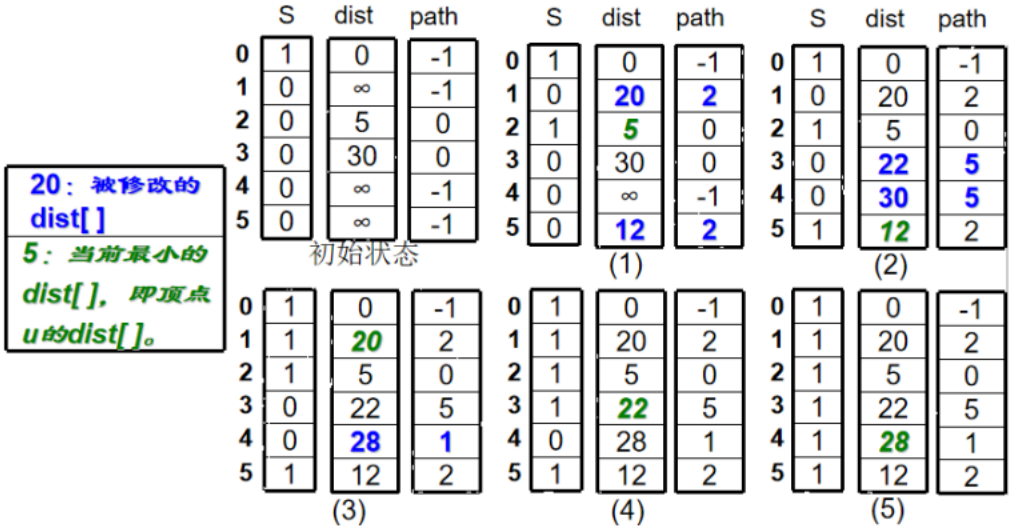
初始：dist [k] = Edge[v0][k]，v0 是源点

初始：dist [k] = min{ dist [k], dist [u] + Edge[u][k] }, vk∈T

其中 vu 是当前 dist[ ]最小的顶点。



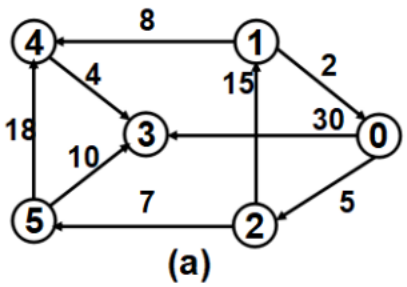
1. 在T选取dist[ ]最小的顶点u
2. 将u加入到S
3. 修改T中顶点的dist[ ]及path[ ]



如何从 dist[ ]和 path[ ]中得到从源点到给定终点的最短路径及其长度？举例说明：设源点为 0，终点为 4，则 path[4]=1　　path[1]=2　　path[2]=0
反过来排列，得到从源点 0 到终点 4 的最短路径为 0,2,1,4，最短路径长度为 dist[4]=28。

1 [3333]: 迪杰斯特拉

如图，求最短路径。



$$\begin{bmatrix} 0 & \infty & 5 & 30 & \infty & \infty \\ 2 & 0 & \infty & \infty & 8 & \infty \\ \infty & 15 & 0 & \infty & \infty & 7 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 4 & 0 & \infty \\ \infty & \infty & \infty & 10 & 18 & 0 \end{bmatrix}$$

输入
顶点数 n　边数 m
m 条边的顶点和权值
某两个顶点
输出
顶点 0 到每个顶点的最短路径
样例输入
6 9
0 2 5
0 3 30
1 0 2
1 4 8
2 1 15
2 5 7
4 3 4
5 3 10
5 4 18
0 4
样例输出
28

```
#include<iostream>
using namespace std;
#define max_v_num 10
#define max 1000000
int Edge[max_v_num][max_v_num];
int vexnum;
```

```
void Dijistra(int v, int end){
   int dis[max_v_num],S[max_v_num];
   int i,j,k;
   for(i=0;i<vexnum;i++){
      dis[i]=Edge[v][i];//初始化 dist[i]
      S[i]=0;//让所有的顶点归到一个初始集合 T
   }
   S[v]=1,dis[v]=0;//把源点 v 加入到集合 S 中，初始化路径数组

   for(i=0;i<vexnum-1;i++){//寻路除源点外的其他顶点
      int min=max,u=v;
      for(j=0;j<vexnum;j++){//求当前源点到集合 T 中所有点的最短距离
         if(S[j]==0&&dis[j]<min){u=j;min=dis[j];}
      }
      S[u]=1;//把 u 加入到集合 T 中
      for(k=0;k<vexnum;k++){//u 点加入到集合 T 后,多出了新的路径，
         //即通过 u 点可以达到新的顶点，更新 dis[]数组
         if(S[k]==0&&Edge[u][k]<max&&dis[u]+Edge[u][k]<dis[k]){
            dis[k]=dis[u]+Edge[u][k];
         }
      }
   }
   cout<<dis[end];

}

int main(){
   int i,j;
   int n,m;//顶点数和边数
   int u,v,w;//边的起始点和改边的权值
   int x,y;//源点和终点
   cin>>n>>m;
   vexnum=n;
   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
         if(i==j)Edge[i][j]=0;
         else Edge[i][j]=max;
   for(i=0;i<m;i++){
      cin>>u>>v>>w;
      Edge[u][v]=w;
   }
   cin>>x>>y;
   Dijistra(x,y);
   return 0;
```

}

2 [2892] 最短路径问题

平面上有 n 个点（n≤100），每个点的坐标均在-10000~10000 之间。其中的一些点之间有连线。若有连线，则表示可从一个点到达另一个点，即两点间有通路，通路的距离为两点间的直线距离。现在的任务是找出从一点到另一点之间的最短路径。

输入

共 n+m+3 行，其中:

第一行为整数 n。

第 2 行到第 n+1 行（共 n 行），每行两个整数 x 和 y，描述了一个点的坐标。

第 n+2 行为一个整数 m，表示图中连线的个数。

此后的 m 行，每行描述一条连线，由两个整数 i 和 j 组成，表示第 i 个点和第 j 个点之间有连线。

最后一行：两个整数 s 和 t，分别表示源点和目标点。

输出

一行，一个实数（保留两位小数），表示从 s 到 t 的最短路径长度。

样例输入

```
5
0 0
2 0
2 2
0 2
3 1
5
1 2
1 3
1 4
2 5
3 5
1 5
```

样例输出

```
3.41
```

```
#include<bits/stdc++.h>
using namespace std;
const int INF = 0x3f3f3f;
double mp[105][105];
double dis[105];
int vis[105];
int a[105][3];
int n,m,s,d;
void init(){
   for( int i=0; i<=n; i++ ){
      dis[i]=INF;
```

```
        for( int j=0; j<=n; j++ ){
            if(j!=i)mp[i][j]=INF;
        }
    }
}
void dijkstra(int s){
    memset(vis,0,sizeof(vis));
    dis[s]=0;
    int k,min;
    for( int i1=1; i1<=n-1; i1++ ){
        min=INF;
        for( int i=1; i<=n; i++ ){
            if(!vis[i]&&dis[i]<min){
                min=dis[i];k=i;
            }
        }
        vis[k]=1;
        for( int i=1; i<=n; i++ ){
            if(!vis[i]&&dis[k]+mp[k][i]<dis[i]){
                dis[i]=dis[k]+mp[k][i];
            }
        }
    }
}
int main(){
    cin>>n;
    init();
    for( int i=1; i<=n; i++ ) cin>>a[i][0]>>a[i][1];
    cin>>m;
    for( int i=0; i<m; i++ ){
        int v1,v2;cin>>v1>>v2;
        mp[v1][v2]=mp[v2][v1]=sqrt(pow(a[v1][0]-a[v2][0],2)+pow(a[v1][1]-a[v2][1],2));
    }
    cin>>s>>d;
    dijkstra(s);
    printf("%.2f\n",dis[d]);
    return 0;
}
```

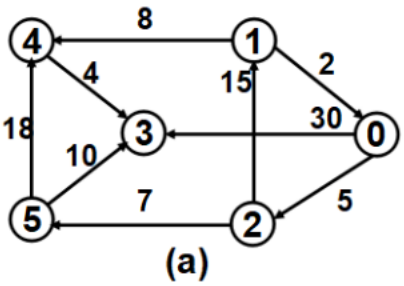
13.2 优化的 Dijkstra 算法

用邻接矩阵处理的 Dijkstra 算法的时间复杂度是 O(n*n)，可以通过邻接表进行优化到 O(n*logn)。通过优先队列实现邻接表存储。

1 [3333]: 迪杰斯特拉

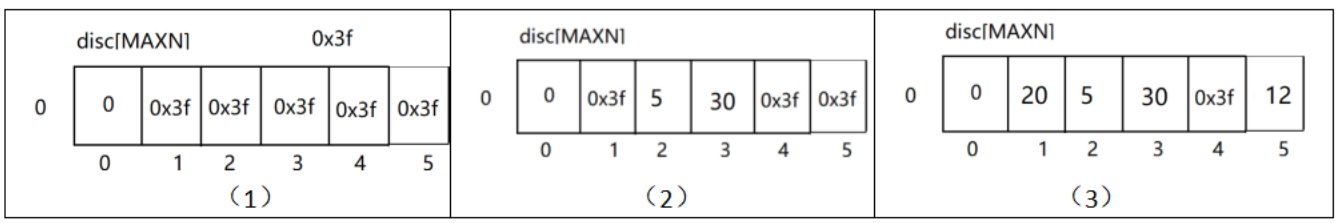
如图，求最短路径。

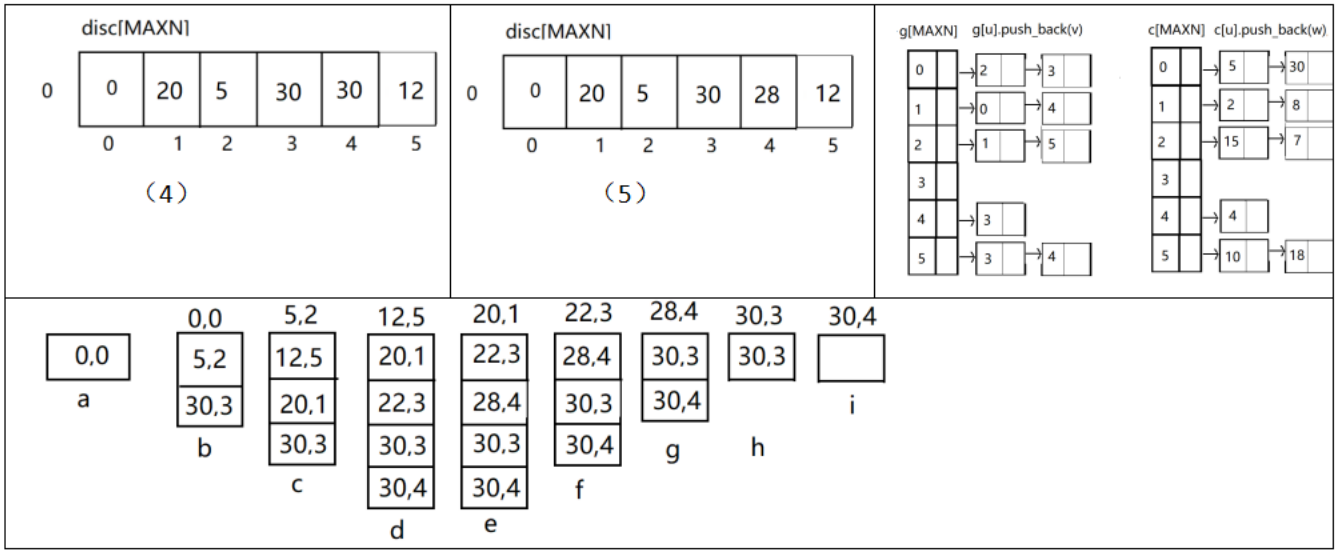
输入
顶点数 n 边数 m
m 条边的顶点和权值
某两个顶点
输出
顶点 0 到每个顶点的最短路径
样例输入



6 9
0 2 5
0 3 30
1 0 2
1 4 8
2 1 15
2 5 7
4 3 4
5 3 10
5 4 18
0 4
样例输出
28

1. 定义优先队列，初试队列为空，源点 v0 入队，初始 dis 数组。
2. 从 V0 出发，走 V2 、V3。将 V2,V3 入队，根据优先队列原则，dis 最小的处于队头，同时更新 disc[]数组。
3. 弹出 v2, 从 V2 出发，走 V1,V5,更新优先队列如 c 所示，同时更新 disc[]数组
4. 弹出 v5，从 V5 出发，走 V3, V4，更新优先队列如 d 所示,同时更新 disc[]数组
5. 弹出 v1, 从 v1 出发，走 V0,V4, 更新优先队列如 e 所示,同时更新 disc[]数组
6. 弹出 V3,从 V3 出发，无路可走，更新优先队列如 f 所示。
7. 弹出 v4,从 v4 出发，disc[]没有更新,如图 g。
8. 到结束。






```
#include<bits/stdc++.h>
using namespace std;
const int inf=0xffffff;
const int MAXN=100;
int edge[MAXN][MAXN];//边信息
int disc[MAXN];//保存源点到各点的最短距离
vector<int>g[MAXN], c[MAXN];
struct node
{
   int dis;//dis 表示源点到当前点 num 的距离
   int num; //编号
};
priority_queue<node>que;//优先队列
bool operator<(const node &a, const node &b)
{
   return a.dis > b.dis;//按照 dis 从小到大排
}

void Dijistra(int v, int end){
   memset(disc, 0x3f, sizeof(disc));
   disc[v] = 0;//初始化，源点到其他点初始距离为极大值
   node tmp;
   tmp.num = v; tmp.dis = 0;//初试源点
   que.push(tmp);//源点入队
   while(!que.empty())
   {

      node now = que.top();//得到 dis 最小点
      que.pop();
```

```
    if(now.dis != disc[now.num])
        continue;//disc 数组中存在的最短距离，如果取出来的边不是最小的就
    for(int i = g[now.num].size() - 1; i >= 0; i --)
    {//遍历邻接表
        int to = g[now.num][i];
        if(disc[to] > disc[now.num] + c[now.num][i])
        {
            disc[to] = disc[now.num] + c[now.num][i];
            que.push(node{disc[to], to});
        }
    }
  }
  cout << disc[end] << endl;
}
int main(){
  int n,m;
  int u,v,w;
  int st,ed;
  cin>>n>>m;
  //队尾添加元素：  vec.push_back();
  //队尾删除元素：  vec.pop_back();
  for(int i=0;i<m;i++){
    cin>>u>>v>>w;
    g[u].push_back(v);//u 是邻接表出来的端点，uv 是一条边
    c[u].push_back(w);//存权值
  }
  cin>>ed;
  Dijistra(0,ed);
  return 0;
}
```

2 [2901]香甜的黄油

农夫 John 发现做出全威斯康辛州最甜的黄油的方法：糖。把糖放在一片牧场上，他知道 N（1≤N≤500）只奶牛会过来舔它，这样就能做出能卖好价钱的超甜黄油。当然，他将付出额外的费用在奶牛上。

农夫 John 很狡猾。像以前的巴甫洛夫，他知道他可以训练这些奶牛，让它们在听到铃声时去一个特定的牧场。他打算将糖放在那里然后下午发出铃声，以至他可以在晚上挤奶。

农夫 John 知道每只奶牛都在各自喜欢的牧场（一个牧场不一定只有一头牛）。给出各头牛在的牧场和牧场间的路线，找出使所有牛到达的路程和最短的牧场（他将把糖放在那）。

输入

第一行：三个数：奶牛数 N，牧场数 P（2≤P≤800），牧场间道路数 C(1≤C≤1450)。

第二行到第 N+1 行: 1 到 N 头奶牛所在的牧场号。
第 N+2 行到第 N+C+1 行：每行有三个数：相连的牧场 A、B，两牧场间距（1≤D≤255），当然，连接是双向的。
输出
一行 输出奶牛必须行走的最小的距离和。
样例输入
3 4 5
2
3
4
1 2 1
1 3 5
2 3 7
2 4 3
3 4 5
样例输出
8

方法一 邻接矩阵写法，超时

```
#include<bits/stdc++.h>时间复杂度 0(N*N*P)
using namespace std;
const int inf=0x3f3f3f3f;
const int N=805;
int dis[N],a[N][N],vis[N],n,m,mm,b[N];

void dijkstra(int x)
{//奶牛数 mm，牧场数 n（2≤n≤800），牧场间道路数 m(1≤m≤1450)。
    memset(vis,0,sizeof(vis));
    for(int i=1;i<=n;i++)
        dis[i]=a[x][i];
    dis[x]=0;
    vis[x]=1;
    int k,m;
    for(int i=1;i<=n;i++)
    {
        m=inf;
        for(int j=1;j<=n;j++)
        {
            if(!vis[j]&&m>dis[j])
            {
                m=dis[j];
                k=j;
            }
        }
```

```
            if(m==inf)break;
            vis[k]=1;
            for(int j=1;j<=n;j++)
                if(!vis[j]&&dis[j]>dis[k]+a[k][j])
                    dis[j]=dis[k]+a[k][j];
        }
}
int main()
{
        int x,y,z;
        cin>>mm>>n>>m;//第一行：三个数：奶牛数 mm，牧场数 n（2≤n≤800），牧场间道路数 m(1≤m≤1450)。
        for(int i=1;i<=mm;i++)
            cin>>b[i];//每头牛所在的牧场
        for(int i=1;i<=n;i++)
            for(int j=1;j<=n;j++)
                a[i][j]=inf;
        memset(vis,0,sizeof(vis));
        for(int i=1;i<=m;i++)//边
        {
            cin>>x>>y>>z;
            a[x][y]=a[y][x]=z;
        }
        int ans=inf;
        for(int i=1;i<=mm;i++)
        {
            int sum=0;
            dijkstra(b[i]);//枚举每个牧场作为聚集点
            for(int j=1;j<=mm;j++)
            {
                sum+=dis[b[j]];
            }
            if(sum<ans) ans=sum;
        }
        cout<<ans;
        return 0;
}
```

方法二 优先队列+链式前向星

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 2e3 + 10;
//24 行的变量都是正常的链式前向星数组
int First[1000], Next[maxn * 2], w[maxn * 2], v[maxn * 2], num_edge;
//dis 表示起点到该点的距离，num 表示该点的编号
```

```
struct node
{
    int dis, num;
};
//建边
void ins(int x, int y, int c)
{
    v[++ num_edge] = y;
    Next[num_edge] = First[x];
    w[num_edge] = c;
    First[x] = num_edge;
}
priority_queue<node>que;
bool operator<(const node &a, const node &b)
{
    return a.dis > b.dis;
}
int n, p, c;
//vis 表示这个区域里有几头奶牛
int vis[maxn];
//表示每个点到起点的距离
int disc[maxn];
int dijkstra(int x)
{
    //求极小值初始化为极大值
    memset(disc, 0x3f, sizeof(disc));
    //起点到起点的距离为 0
    disc[x] = 0;
    //起点入点
    que.push(node{0, x});
    while(!que.empty())
    {
        node now = que.top();
        que.pop();
        //该点被取过了就 continue
        if(now.dis != disc[now.num])
            continue;
        for(int i = First[now.num]; i != -1; i = Next[i])
        {
            int to = v[i];
            //如果能通过 now.num 这个点使得起点到 to 这个点更近，就更新距离
            if(disc[to] > disc[now.num] + w[i])
            {
                disc[to] = disc[now.num] + w[i];
```

```
                que.push(node{disc[to], to});
            }
        }
    }
    int ans = 0;
    //由于是求所有奶牛到该牧场的距离和最小，所以要用乘法
    for(int i = 1; i <= p; i ++)
    {
        ans += disc[i] * vis[i];
    }
    return ans;
}
int main()
{
    memset(First, -1, sizeof(First));
    scanf("%d %d %d", &n, &p, &c);

    for (int i = 1; i <=n; ++ i)
    {
        int x;
        scanf("%d", &x);
        //x 区域的奶牛数量+1
        vis[x] ++;
    }
    while(c --)
    {
        int x, y, c;
        scanf("%d %d %d", &x, &y, &c);
        //由于只说是有边，所以存双向边
        ins(x, y, c); ins(y, x, c);
    }
    //枚举把黄油放在每一个牧场，求解，最后取一个极小值
    int ans = 1e9;
    for(int i = 1; i <= p; i ++)
    {
        ans = min (ans, dijkstra(i));
    }
    printf("%d\n", ans);
    return 0;
}
```

方法三 优先队列+vector

```
#include<bits/stdc++.h>
using namespace std;
```

```
const int maxn = 2e3 + 10;
vector<int>g[maxn], w[maxn];
//dis 表示起点到该点的距离，num 表示该点的编号
struct node
{
    int dis, num;
};
//建边

priority_queue<node>que;
bool operator<(const node &a, const node &b)
{
    return a.dis > b.dis;
}
int n, p, c;
//vis 表示这个区域里有几头奶牛
int vis[maxn];
//表示每个点到起点的距离
int disc[maxn];
int dijkstra(int x)
{
    //求极小值初始化为极大值
    memset(disc, 0x3f, sizeof(disc));
    //起点到起点的距离为 0
    disc[x] = 0;
    //起点入点
    que.push(node{0, x});
    while(!que.empty())
    {
        node now = que.top();
        que.pop();
        //该点被取过了就 continue
        if(now.dis != disc[now.num])
            continue;
        for(int i = g[now.num].size() - 1; i >= 0; i --)
        {//遍历邻接表
            int to = g[now.num][i];
            if(disc[to] > disc[now.num] + w[now.num][i])
            {
                disc[to] = disc[now.num] + w[now.num][i];
                que.push(node{disc[to], to});
            }
        }
    }
```

```
    int ans = 0;
    //由于是求所有奶牛到该牧场的距离和最小，所以要用乘法
    for(int i = 1; i <= p; i ++)
    {
        ans += disc[i] * vis[i];
    }
    return ans;
}
int main()
{
    scanf("%d %d %d", &n, &p, &c);

    for (int i = 1; i <=n; ++ i)
    {
        int x;
        scanf("%d", &x);
        //x 区域的奶牛数量+1
        vis[x] ++;
    }
    while(c --)
    {
        int x, y, c;
        scanf("%d %d %d", &x, &y, &c);
        //由于只说是有边，所以存双向边
        g[x].push_back(y);//u 是邻接表出来的端点，uv 是一条边
        w[x].push_back(c);//存权值
        g[y].push_back(x);
        w[y].push_back(c);
    }
    //枚举把黄油放在每一个牧场，求解，最后取一个极小值
    int ans = 1e9;
    for(int i = 1; i <= p; i ++)
    {
        ans = min (ans, dijkstra(i));
    }
    printf("%d\n", ans);
    return 0;
}
```

课前练习

1．[2006]（全排列）下面程序的功能是利用递归方法生成从1 到n(n<10)的n 个数的全部可能的排列（不一定按升序输出）。例如，输入3，则应该输出（每行输出5 个排列）：

123 132 213 231 321 312

程序：

```
#include <iostream.h>
#include <iomanip.h>
int n ,a[10]; // a[1],a[2],…,a[n]构成n 个数的一个排列
long count=0; // 变量count 记录不同排列的个数，这里用于控制换行
void perm(int k)
{int j,p,t;
if( ①_____)
{count++;
for(p=1;p<=n;p++)
cout <<setw(1)<<a[p];
cout <<" ";
if( ②_____) cout <<endl;
return;
}
for(j=k;j<=n;j++)
{t=a[k];a[k]=a[j];a[j]=t;
③_____ ;
t=a[k]; ④_____ ;
}
}
void main()
{int i;
cout <<"Entry n:"<<endl;
cin >>n;
for(i=1;i<=n;i++) a[i]=i;
⑤_____ ;
}
```

2. [2006]由键盘输入一个奇数P (P<100,000,000)，其个位数字不是5，求一个整数S，使P×S =1111…1 (在给定的条件下，解S 必存在)。要求在屏幕上依次输出以下结果:
（1）S 的全部数字。除最后一行外，每行输出50 位数字。（2） 乘积的数字位数。
例1：输入p=13，由于13*8547=111111，则应输出（1）8547，（2）6
例2 输入p=147，则输出结果应为
（1）7558578987150415721844293272864701436l3
（2）42，即等式的右端有42 个1。
程序：

```
#include <iostream.h>
#include <iomanip.h>
void main()
{long p,a,b,c,t,n;
while (1)
{ cout <<"输入p, 最后一位为1 或3 或7 或9:"<<endl;
   cin >>p;
```

```
if ((p%2!=0)&&(p%5!=0)) // 如果输入的数符合要求，结束循环
1                    ;
}
a=0; n=0;
while (a<p)
{a=a*10+1;
n++; // 变量a 存放部分右端项，n 为右端项的位数
}
t=0;
do
{b=a/p;
cout <<setw(1)<<b;
t++;
if (   2                    )
cout <<endl;
c= 3            ; a=  4             ; n++;
} while (c>0);
cout<<endl<<"n="<<  5                 <<endl;
}
```

习题

1．[2008]

```
#include <iostream>
using namespace std;
void func(int ary[], int n )
{
    int i=0, j, x;
    j=n-1;
    while(i<j)
    {
        while (i<j&&ary[i]>0) i++;
        while (i<j&&ary[j]<0) j--;
        if (i<j){
            x=ary[i];
            ary[i++]=ary[j];
            ary[j--]=x;
        }
    }
}

int main()
{
```

```
    int a[20], i, m;
    m=10;
    for(i=0; i<m; i++)
    {
        cin>>a[i];
    }
    func(a, m);
    for (i=0; i<m; i++)
        cout<<a[i]<<" ";
    cout<< endl;
    return 0;
}
```

输入：5 4 -6 -11 6 -59 22 -6 1 10

输出：_____

2．[2007]

```
#include <iostream.h>
#include <iomanip.h>
#include "math.h"
void main()
{int a1[51]={0};
int i,j,t,t2,n=50;
for (i=2;i<=sqrt(n);i++)
if(a1[i]==0)
{t2=n/i;
for(j=2;j<=t2;j++) a1[i*j]=1;
}
t=0;
for (i=2;i<=n;i++)
if(a1[i]==0)
{cout<<setw(4)<<i; t++;
if(t%10==0) cout<<endl;
}
cout<<endl;
}
```

输出：_____

3. 优先队列求最短路

```
#include<bits/stdc++.h>
using namespace std;
const int inf=0xffffff;
const int MAXN=100;
int edge[MAXN][MAXN];//边信息
int disc[MAXN];//保存源点到各点的最短距离
```

```
 1
struct node
{
   int dis;//dis 表示源点到当前点 num 的距离
   int num; //编号
};
2                                          优先队列
bool      3
{
   return a.dis > b.dis;//按照 dis 从小到大排
}

void Dijistra(int v, int end){
   memset(disc, 0x3f, sizeof(disc));
   disc[v] = 0;//初始化，源点到其他点初始距离为极大值
   node tmp;
   tmp.num = v;
   tmp.dis = 0;//初试源点
    4                                          //源点入队
   while(!que.empty())
   {

      5                              //得到 dis 最小点
     que.pop();
      6
        continue;//disc 数组中存在的最短距离，如果取出来的边不是最小的就
      7
     {//遍历邻接表
        int to = g[now.num][i];
        if(8                                          )
        {
           disc[to] =   9
           que.push(node{disc[to], to});
        }
     }
   }
   cout << disc[end] << endl;
}
int main(){
   int n,m;
   int u,v,w;
   int st,ed;
   cin>>n>>m;
   //队尾添加元素：  vec.push_back();
```

```
    //队尾删除元素：  vec.pop_back();
    for(int i=0;i<m;i++){
        cin>>u>>v>>w;
        10                              //u 是邻接表出来的端点，uv 是一条边
        11                              //存权值
    }
    cin>>ed;
    Dijistra(0,ed);
    return 0;
}
```

第 14 讲　拓扑排序

14.1 原理

设 G=(V，E)是一个具有 n 个顶点的有向图，V 中顶点序列 v1，v2，…，vn 称为一个拓扑序列，当且仅当该顶点序列满足下列条件：

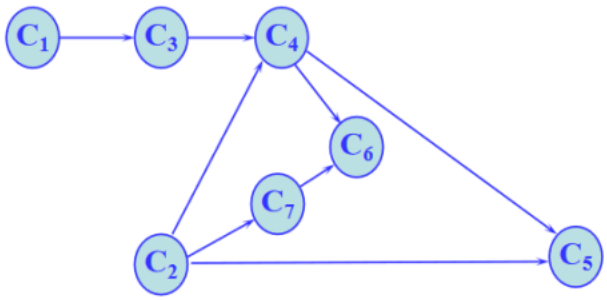
若<i，j>是图中的边（即从顶点 i 到 j 有一条路径），则在拓扑序列中顶点 i 必须排在顶点 j 之前。

在一个有向图中找一个拓扑序列的过程称为拓扑排序。

例如，计算机专业的学生必须完成一系列规定的基础课和专业课才能毕业，假设这些课程的名称与相应代号有如下关系：

课程代号	课程名称	先修课程
C_1	高等数学	无
C_2	程序设计	无
C_3	离散数学	C_1
C_4	数据结构	C_2，C_3
C_5	编译原理	C_2，C_4
C_6	操作系统	C_4，C_7
C_7	计算机组成原理	C_2

课程之间的先后关系可用有向图表示：



对这个有向图进行拓扑排序可得到一个拓扑序列：C1-C3-C2-C4-C7-C6-C5。也可得到另一个拓扑序列：C2-C7-C1-C3-C4-C5-C6，还可以得到其他的拓扑序列。学生按照任何一个拓扑序列都可以顺序地进行课程学习。

拓扑排序步骤：

（1）从有向图中选择一个没有前趋（即入度为 0）的顶点并且输出它。

（2）从网中删去该顶点，并且删去从该顶点发出的全部有向边。

（3）重复上述两步，直到剩余的网中不再存在没有前趋的顶点为止。

14.2 案例讲解

1 [3413] 拓扑排序

由某个集合上的一个偏序得到该集合上的一个全序，这个操作被称为拓扑排序。偏序和全序的定义分别如下：

若集合 X 上的关系 R 是自反的、反对称的和传递的，则称 R 是集合 X 上的偏序关系。

设 R 是集合 X 上的偏序，如果对每个 x,y∈X 必有 xRy 或 yRx，则称 R 是集合 X 上的全序关系。

由偏序定义得到拓扑有序的操作便是拓扑排序。

拓扑排序的流程如下：

1. 在有向图中选一个没有前驱的顶点并且输出之；
2. 从图中删除该顶点和所有以它为尾的弧。

重复上述两步，直至全部顶点均已输出，或者当前图中不存在无前驱的顶点为止。后一种情况则说明有向

图中存在环。

```
#include<bits/stdc++.h>
using namespace std;
vector<int> g[55];
int n,degree[55],ge[55][55];
queue<int> node;
bool TopologicalSort(){
        int num=0;
        stack<int> p;
        for(int i=0;i<n;i++)
        if(degree[i]==0){
                p.push(i);
                degree[i]=-1;
        }
        while(!p.empty()){
                int u=p.top();
                node.push(u);
                p.pop();
                for(int i=0;i<g[u].size();i++){
                        int v=g[u][i];
                        degree[v]--;
                        if(degree[v]==0){
                                p.push(v);
                                degree[v]=-1;
                        }
                }
                g[u].clear();
                num++;
        }
        if(num==n) return true;
        else return false;
}
int main(){
        cin>>n;
        for(int i=0;i<n;i++) for(int j=0;j<n;j++) cin>>ge[i][j];
        for(int i=0;i<n;i++){
                int count=0;
                for(int j=0;j<n;j++){
                        if(ge[i][j]==1) g[i].push_back(j);
                        if(ge[j][i]==1) count++;
                }
                degree[i]=count;
        }
```

```
        bool flag=TopologicalSort();
        if(flag){
                int len=node.size();
                for(int i=0;i<len;i++){
                        cout<<node.front()<<" ";
                        node.pop();
                }
                cout<<endl;
        }
        else cout<<"ERROR"<<endl;
        return 0;
}
```

2 [2893]家谱树

有个人的家族很大，辈分关系很混乱，请你帮整理一下这种关系。
给出每个人的孩子的信息。
输出一个序列，使得每个人的后辈都比那个人后列出。
输入
第 1 行一个整数 N（1≤N≤100），表示家族的人数；
接下来 N 行，第 I 行描述第 I 个人的儿子；
每行最后是 0 表示描述完毕。（碰到 0，表示当前节点的孩子结束，如此就不用表示多少个孩子的 n 了）
输出
输出一个序列，使得每个人的后辈都比那个人后列出；
如果有多解输出任意一解。
样例输入
5
0
4 5 1 0
1 0
5 3 0
3 0
样例输出
2 4 5 3 1

```
#include<bits/stdc++.h>
using namespace std;
const int maxn =135;
 1                          ;//邻接表保存边信息
int in[maxn],n;//存顶点入度信息
void topsort(){
        2
```

```
        for(int i=1;i<=n;i++){
                3_____________________)
                q.push(i);/*将入度为 0 的入队*/
        }
        while(!q.empty()){
                4_____________________;/*取出入度为 0 的*/
                q.pop();
                printf("%d ",t);/*出队输出*/
                for(5_____________________){
                        int x=g[t][i];
                        6_____________________/*对 t 出发能到的点都减一*/
                        if(!in[x]) /*更新，入度为 0 进队*/
                        7_____________________
                }
        }
}
int main(){
        int i,j,x;
        cin>>n;
        for(i=1;i<=n;i++){
                while(cin>>x&&x){
                        in[x]++;//入度+1
                        8_____________________//建立邻接表

                }
        }
        topsort();
        return 0;
}
```

课前练习

1. [2016]（读入整数）请完善下面的程序，使得程序能够读入两个 int 范围内的整数， 并将这两个整数分别输出，每行一个。
输入的整数之间和前后只会出现空格或者回车。输入数据保证合法。
例如：
输入：
123 -789
输出：
123
-789

```
#include <iostream>
```

```
using namespace std;
int readint() {
    int num = 0;    // 存储读取到的整数
    int negative = 0;    // 负数标识
    char c;    // 存储当前读取到的字符
    c = cin.get();
    while ((c < '0' || c > '9') && c != '-')
        c =    (1)    ;
    if (c == '-') negative = 1;
    else
        (2)      ;
    c = cin.get();
    while (    (3)    ) {
        (4)    ;
        c = cin.get();

    }
    if (negative == 1)
        (5)        ;
    return num;
}
int main() { int a, b;
    a = readint();
    b = readint();
    cout << a << endl << b << endl; return 0;
}
```

2. [2016]（（郊游活动）有 n 名同学参加学校组织的郊游活动，已知学校给这 n 名同学 的郊游总经费为 A 元，与此同时第 i 位同学自己携带了 Mi 元。为了方便郊 游，活动地点提供 B(≥n)辆自行车供人租用，租用第 j 辆自行车的价格为 Cj 元，每位同学可以使用自己携带的钱或者学校的郊游经费，为了方便账务管理，每位同学只能为自己租用自行车，且不会借钱给他人，他们想知道最多 有多少位同学能够租用到自行车。

本题采用二分法。对于区间[l, r]，我们取中间点 mid 并判断租用到自行 车的人数能否达到 mid。判断的过程是利用贪心算法实现的。

```
#include <iostream>
using namespace std;
#define MAXN 1000000
int n, B, A, M[MAXN], C[MAXN], l, r, ans, mid;
bool check(int nn) {
    int count = 0, i, j;
    i=(1) ;
    j=1;
    while (i <= n) {
    if ((2)    )
```

```
            count += C[j] - M[i];
            i++;
            j++;
        }
        return   (3)   ;
}void sort(int a[], int l, int r) {
    int i = l, j = r, x = a[(l + r) / 2], y;
    while (i <= j) {
        while (a[i] < x) i++;
        while (a[j] > x) j--;
          if (i <= j) {y = a[i]; a[i] = a[j]; a[j] = y; i++; j--;}
  }
  if (i < r) sort(a, i, r);
  if (l < j) sort(a, l, j);
}

int main() { int i;
cin >> n >> B >> A;
for (i = 1; i <= n; i++) cin >> M[i];
for (i = 1; i <= B; i++) cin >> C[i];
sort(M, 1, n); sort(C, 1, B); l = 0;
r = n;
while (l <= r) {
mid = (l + r) / 2;
if (   (4)   ) {
ans = mid;
l = mid + 1;
} else
r =   (5)   ;
}
cout << ans << endl; return 0;

}
```

习题

1 合并礼物

【问题描述】

圣诞节快到了，圣诞老人又要开始忙起来了，和往年一样，圣诞老人要在礼物乐园里挑选礼物送给小朋友们。

在礼物乐园，圣诞老人挑选好礼物后，把礼物按照不同的种类分成了不同的堆，现在，圣诞老人决定把所有的礼物合成一堆。

每一次合并，圣诞老人可以把两堆礼物合并到一起，消耗的体力等于两堆礼物的重量之和。可以看出，所

有的礼物经过 n-1 次合并之后，就只剩下一堆了。圣诞老人在合并礼物时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些礼物搬到他的鹿车，所以圣诞老人在合并礼物时要尽可能地节省体力。假定每个礼物重量为 1，并且已知礼物的种类和每种礼物的数目，你的任务是设计出合并的次序方案，使圣诞老人耗费的体力最小，并输出这个最小的体力耗费值。

例如有 3 种礼物，数目依次为 1，2，9。可以先将 1、2 堆合并，新堆数目为 3，耗费体力为 3。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为 12，耗费体力为 12。所以圣诞老人总共耗费体力=3+12=15。可以证明 15 为最小的体力耗费值。

【输入】

输入包括两行，第一行是一个整数 n(1<=n<=100)，表示礼物的种类数。第二行包括 n 个整数，用空格分隔，第 i 个整数 a_i(1<=a_i<=100)是第 i 种礼物的数目。

【输出】

输出包括一行，这一行只包含一个整数，也就是最小的体力耗费值。

【样例输入】

3

1 2 9

【样例输出】

15

请完善下列程序：

```
#include <iostream>
using namespace std;
long n, i, j;
long a[102];
void myQsort(int l, int r)
{
    long i, j, x, temp;
    i = l;
    j = r;
    x = a[(i + j) >> 1];
    while (    ①    )
    {
        while (a[i] < x)
            i++;
        while (x < a[j])
            j--;
        if (i <= j)
        {
            temp = a[i];
            ②    ;
            a[j] = temp;
            i = i + 1;
            ③    ;
```

```
        }
    }
    if (l < j)
        myQsort(l, j);
    if (i < r)
        ____④____;
}

int main()
{
    int total;
    cin >> n;
    for (i = 1; i <= n; i++)
        cin >> a[i];
    myQsort(1, n);
    ____⑤____;
    for (i = 1; i <= n - 1; i++)
    {
        total = total + a[i] + a[i + 1];
        a[i + 1] = a[i] + a[i + 1];
        ____⑥____;
    }
    cout << total;
    return 0;
}
```

2 【傻瓜电梯】

所谓傻瓜电梯指的是在响应用户请求时缺乏相应的“智商”，在上升或下降的过程中不能把中途的乘客捎带入电梯，而只会严格按照用户发出请求的先后顺序依次完成任务。

比如，原来电梯在 1 楼，首先 6 楼有一位乘客发出请求，要求由 6 楼乘坐到 10 楼去，此时电梯马上会上去，但在电梯上升到 3 楼时，另外一位乘客发出请求由 5 楼乘坐到 8 楼去，傻瓜电梯却不会在上升途中把 5 楼的乘客捎带上去，而只会先把 6 楼的乘客送到 10 楼，然后再下来把 5 楼的乘客送到 8 楼。

傻瓜电梯由 i 楼上升到 i+1 楼（或下降 1 楼）的时间都是 3 秒，每到达一个楼层，不管进出乘客有多少，所耽搁的时间都是 6 秒。现在告诉傻瓜电梯 n 个用户请求，计算傻瓜电梯把所有乘客送到目标楼层时总共所需要的时间。

输入的第一行包含两个整数 x（1<=x<=50）和 n(1<=n<=100)，分别表示傻瓜电梯开始所在的楼层和一共所有的请求数目。下面有 n 行，每行包含 3 个整数，依次表示该请求发出的时刻、乘客目前所在的楼层和将要去的目标楼层。其中请求发出的时间以秒为时刻单位，最大可能的值是 2000。如果某两个请求的发出时间相同，则按照输入时的先后顺序依次处理。

输出只包含一行一个整数，表示傻瓜电梯把所有乘客送到目标楼层后总共所需要的时间（从得到第一条请求时开始计算时间），单位是秒。

输入样例：

3 4

10 10 2

1 8 19
2 1 12
8 6 10
输出样例：
162

下列是用来解决该问题的程序，请将程序补充完整。

```
#include <iostream>
#include <cmath>
using namespace std;
const int maxn = 100 + 5;
int n, start;
int ti[maxn + 1], st[maxn + 1], sf[maxn + 1];

void inputdata()
{
    int i;
    cin >> start >> n;
    for (i = 1; i <= n; i++)
        cin >> ti[i] >> st[i] >> sf[i];
    return;
}

void solveprob()
{
    int i, j, k, tmp, ans;
    for (i = 1; i <= n - 1; i++)
    {
        k = i;
        for (j = i + 1; j <= n; j++)
            if (ti[k] > ti[j])
                k = ①;
        if (k != ②)
        {
            tmp = ti[k];
            ti[k] = ti[i];
            ti[i] = tmp;
            tmp = st[k];
            st[k] = st[i];
            st[i] = tmp;
            tmp = sf[k];
            sf[k] = ③;
            sf[i] = tmp;
```

```
        }
    }
    ans = 0;
    for (i = 1; i <= n; i++)
    {
        if (i > 1 && ti[i] > ans)
            ans = ti[i];
        ans = ans + abs(st[i] - start) * 3;
        ans = ans +__④__;
        if (start != st[i])
            ans =__⑤__;
        ans = ans + 6;
        start = sf[i];
    }
    cout << ans << endl;
}

int main()
{
    __⑥__;
    solveprob();
    return 0;
}
```

第 15 讲 STL

STL 是所有 C++ 编译器和所有操作系统平台都支持 的一种库，说它是一种库是因为对所有的编译器来说，STL 提供给 C++程序设计者的接口都是一样的。

STL 提供了大量的可复用的模板类和算法函数。例如，程 序员再也不用自己设计排序，搜索算法了，这些都已经是 STL 的一部分。使用 STL 的应用程序保证了得到的实现在处理速度和内存 利用方面都是高效的，因为 STL 设计者们已经为我们考虑好了。

使用 STL 编写的代码更容易修改和阅读。因为代码更短 了，很多基础工作代码已经被组件化了；虽然，STL 的优点甚多，但是 STL 的语法实在令初学者头 疼，许多人望而却步。

标准模板库 STL 关注的重点是数据结构和算法其关键组成部分是容器(containers)、算法 (algorithms)、迭代器(iterators)、函数对象(Function Object)、适配器(Adaptor)。

容器(containers)：容器是数据在内存中组织的方法，例如，数组、堆栈、队列、链表或二叉树。

常用的 STL 容器包括 vector，栈，队列，链表，set，pair, map，sort 函数，next_permutation 函数

15.1 vector

算法竞赛中，为避免出错，一般用静态数组。能开多大就开多大：

int a[1000000], dp[1000000];

Vector，也称为动态数组，运行时根据需要改变数组大小。如果空间紧张，用 STL 的 vector 建立动态数组，不易出错。

在 STL 中，标准库的全部成员在预先定义的命名空 间 std 中。如果要用类模板 vector ，有两种方法：

方法一：

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

方法二：

```
#include <vector>
using std::vector;
```

功能	例子	说明
定义 int 型数组	vector<int> a;	默认初始化，a 为空
	vector<int> b(a);	用 a 定义 b
	vector<int> a(100);	a 有 100 个值为 0 的元素
	vector<int> a(100, 6);	100 个值为 6 的元素
定义 string 型数组	vector<string> a(10,"null");	10 个值为 null 的元素
	vector<string> vec(10,"hello");	10 个值为 hello 的元素
	vector<string>b(a.begin(), a.end());	b 是 a 的复制
定义结构型数组	struct point { int x, y;}; vector<point> a;	a 用来存坐标

```
#include <cstring>
```

```
#include <vector>
#include <iostream>
using namespace std;
int ar[10] = { 12, 45, 234, 64, 12, 35, 63, 23, 12, 55 };
char* str = "Hello World";
int main()
{
        vector <int> vec1(ar, ar + 10); // first=ar,last=ar+10, 不包括 ar+10
        vector < char > vec2(str, str + strlen(str)); // first=str,last= str+strlen(str)
        cout << "vec1:" << endl;
        // 打印 vec1 和 vec2 ， const_iterator 是迭代器
        for (int i=0; i<vec1.size(); i++)cout << vec1[i]<<" ";// 输出
        //for (vector<int>::const_iterator p = vec1.begin();p != vec1.end(); ++p)
        //cout << *p<<" ";
        cout << '\n' << "vec2:" << endl;
        for (vector< char >::const_iterator p1 = vec2.begin();p1 != vec2.end(); ++p1)
        cout << *p1;
        cout << '\n';
        return 0;
}
```

定义多维数组，例如定义一个二维数组：

vector<int> a[MAXN];

它的第一维大小是固定的 MAXN，第二维是动态的。

用这个方式，可以实现图的邻接表存储。

1 [7147] 圆桌问题

圆桌上围坐着 2n 个人。其中 n 个人是好人，另外 n 个人是坏人。从第一个人开始数数，数到第 m 个人，立即赶走该人；然后从被赶走的人之后开始数数，再将数到的第 m 个人赶走……依此方法不断赶走围坐在圆桌上的人。

预先应如何安排这些好人与坏人的座位，能使得在赶走 n 个人之后，圆桌上围坐的剩余的 n 个人全是好人？

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    1                          //模拟圆桌
    int n, m;
    while(cin >> n >> m){
        table.clear();
        for(int i=0; i<2*n; i++)
            2                  //初始化
        int pos = 0;                        //记录当前位置
        for(int i=0; i<n; i++){             //赶走 n 个
```

```
            pos =    3                          //圆桌是个环，取余处理
            table.erase(table.begin() + pos); //赶走坏人，table 人数减 1
        }
        int j = 0;
        for(int i=0; i<2*n; i++){                    //打印预先安排座位
            if(!(i%50) && i)    cout<<endl;          //50 个字母一行
            if(j<table.size() && i==table[j]){ //table 留下的都是好人
                j++;
                cout<<"G";
            }
            else
                cout<<"B";
        }
        cout<<endl<<endl;                                  //留一个空行
    }
    return 0;
}
```

15.2 优先队列和 priority_queue

优先队列：优先级最高的先出队。
队列和排序的完美结合，不仅可以存储数据，还可以将这些数据按照设定的规则进行排序。
每次的 push 和 pop 操作，优先队列都会动态调整，把优先级最高的元素放在前面。

15.3 链表和 list

STL 的 list ：双向链表。它的内存空间不必连续，通过指针来进行数据的访问，高效率地在任意地方删除和插入，插入和删除操作是常数时间。
list 和 vector 的优缺点正好相反，它们的应用场景不同：
（1）vector：插入和删除操作少，随机访问元素频繁；
（2）list：插入和删除频繁，随机访问较少。

```
#include <string>    #include <list>
using namespace std;
void PrintIt(list<int> n) {
for (list<int>::iterator iter =    n.begin(); iter != n.end(); ++iter)
cout << *iter << " ";//用迭 代器进行输出循环
}
int main(void) {
list<int> listn1, listn2;
//给 listn1,listn2 初始化  listn1.push_back(123);    listn1.push_back(0);    listn1.push_back(34);
listn1.push_back(1123);
//now listn1:123,0,34,1123
```

```
listn2.push_back(100);    listn2.push_back(12);
//now listn2:100,12
```

2 士兵队列训练问题

一队士兵报数：从头开始 1 至 2 报数，凡报到 2 的出列，剩下的向小序号方向靠拢，再从头开始进行 1 至 3 报数，凡报到 3 的出列，剩下的向小序号方向靠拢，…，以后从头开始轮流进行 1 至 2 报数、1 至 3 报数直到剩下的人数不超过 3 人为止。

输入：士兵人数。

输出：剩下的士兵最初的编号。

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int t,n;
    cin>>t;
    while(t--) {
        cin>>n;
        int k=2;
        list<int> mylist;                    //定义
        list<int>::iterator it;
        for(int i=1;i<=n;i++)
            mylist.push_back(i);          //赋值
        while(mylist.size() > 3) {
            int num = 1;
            for(it = mylist.begin(); it != mylist.end(); ){
                if(num++ % k == 0)
                    it = mylist.erase(it);
                else
                    it++;
            }
            k==2 ? k=3:k=2;                    //1 至 2 报数，1 至 3 报数
        }
        for(it = mylist.begin(); it != mylist.end(); it++){
            if (it != mylist.begin())
                cout << " ";
            cout<<*it;
        }
        cout<<endl;
    }
    return 0;
}
```

15.4 set 集合

STL 的 set 用二叉搜索树实现，集合中的每个元素只出现一次，且是排好序的。访问元素的时间复杂度是 O(logn)的。

set 和 map 在竞赛题中应用很广泛。特别是需要用二叉搜索树处理数据的题目，如果用 set 或 map 实现，能极大地简化代码。

例子	说明
set<Type> A;	定义
A. insert(item);	把 item 放进 set
A.erase(item);	删除元素 item
A.clear();	清空 set
A.empty ();	判断是否为空
A.size();	返回元素个数
A.find(k);	返回一个迭代器，指向键值 k
A.lower_bound(k);	返回一个迭代器，指向键值不小于 k 的第一个元素
A.upper_bound();	返回一个迭代器，指向键值大于 k 的第一个元素

3[7149] 产生冠军

有一群人，打乒乓球比赛，两两捉对撕杀，每两个人之间最多打一场比赛。
球赛的规则如下：
如果 A 打败了 B，B 又打败了 C，而 A 与 C 之间没有进行过比赛，那么就认定，A 一定能打败 C。
如果 A 打败了 B，B 又打败了 C，而且，C 又打败了 A，那么 A、B、C 三者都不可能成为冠军。
根据这个规则，无需循环较量，或许就能确定冠军。你的任务就是面对一群比赛选手，在经过了若干场撕杀之后，确定是否已经实际上产生了冠军。

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    set<string> A, B;                    //定义集合
    string s1, s2;
    int n;
    while(cin >> n && n){
        for(int i=0; i<n; i++) {
            cin >> s1 >> s2;
            A.insert(s1);   A.insert(s2);    //所有人放进集合 A
            B.insert(s2);                    //失败者放进集合 B
        }
        if(A.size() - B.size() == 1)
            cout << "Yes" << endl;
```

```
        else
            cout << "No" << endl;
        A.clear(); B.clear();
    }
    return 0;
}
```

15.5 map 操作详解

map：关联容器，实现从键（key）到值（value）的映射。
map 效率高的原因：用平衡二叉搜索树来存储和访问。
（1）定义：map<string, int> student，存储学生的 name 和 id。
（2）赋值：例如 student[“Tom”] = 15。这里把”Tom”当成普通数组的下标来使用。
（3）查找：找学号时，直接用 student[“Tom”]表示他的 id，不用再去搜索所有的姓名。

在 map 中插入元素
改变 map 中的条目非常简单，因为 map 类已经对[]操作符进 行了重载

```
map<char,int>mymap;
        mymap['a'] = 1;
        mymap['b'] = 2;
```

或者

```
mymap.insert ( pair<char,int>('a',1) );
```

1. 查找并获取 map 中的元素
获得一个值的最简单方法

```
int ans = mymap[‘a’];
```

但是,只有当 map 中有这个键的实例时才会成功，否则会 自动插入一个实例，值为初始化值。
因此我们可以使用 find() 方法来发现一个键是否存在。传 入的参数是要查找的 key

```
if(mymap.find(‘a’)==mymap.end()){
        //没找到
}else{
        //找到 insert
}
#include <iostream>
#include <map>
using namespace std;
int main()
{
        map<char,int> mymap;
        map<char,int>::iterator it;
        mymap.insert ( pair<char,int>('a',1) );
        mymap['b'] = 2;
        mymap['c'] = 3;
        mymap['d'] = 4;
        mymap['e'] = 5;
```

```
    it=mymap.find('c');   mymap.erase (it);   mymap.erase('d');
    for ( it=mymap.begin() ; it != mymap.end(); it++ )
    cout << (*it).first << " => " << (*it).second << endl;
}
```

3[7150] 购物

女孩 dandelion 经常去购物，她特别喜欢一家叫“memory”的商店。由于春节快到了，所有商店的价格每天都在上涨。她想知道这家商店每天的价格排名。

Input：

第一行是数字 n（n <= 10000），代表商店的数量。

后面 n 行，每行有一个字符串（长度小于 31，只包含小写字母和大写字母）表示商店的名称。

然后一行是数字 m（1 <= m <= 50），表示天数。

后面有 m 部分，每部分有 n 行，每行是数字 s 和一个字符串 p，表示商店 p 在这一天涨价 s。

Output：包含 m 行，第 i 行显示第 i 天后店铺“memory”的排名。排名的定义为：如果有 t 个商店的价格高于“memory”，那么它的排名是 t + 1。

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n, m, p;
    map<string, int> shop;
    while(cin>>n) {
        string s;
        for(int i=1; i<=n; i++) cin>>s; //输入商店名字。实际上用不着处理
        cin >> m;
        while(m--) {
            for(int i=1; i<=n; i++) {
                cin >> p >> s;
                shop[s] += p;                    //用 map 可以直接操作商店，加上价格
            }
            int rank = 1;
            map<string,int>::iterator it;                //迭代器
            for(it=shop.begin(); it != shop.end(); it++)
                if(it->second > shop["memory"])     //比较价格
                    rank++;
            cout<<rank<<endl;
        }
        shop.clear();
    }
    return 0;
}
```