ACM/ICPC 代码库

吉林大学计算机科学与技术学院 **2005** 级

2007-2008

文档变更记录

修订日期	修订内容	修订版本	修订人
2007	创建	1.0	jojer（sharang、xwbsw、Liuctic）
2008.10	修订	1.1	Fandywang

目录

Graph 图论

```
/*==================================================*\
 | DAG 的深度优先搜索标记
 | INIT: edge[][]邻接矩阵; pre[], post[], tag全置0;
 | CALL: dfstag(i, n);   pre/post:开始/结束时间
\*==================================================*/
int edge[V][V], pre[V], post[V], tag;
void dfstag(int cur, int n)
{  // vertex: 0 ~ n-1
    pre[cur] = ++tag;
    for (int i=0; i<n; ++i) if (edge[cur][i]) {
        if (0 == pre[i]) {
            printf("Tree Edge!\n");
            dfstag(i, n);
        } else {
            if (0 == post[i]) printf("Back Edge!\n");
            else if (pre[i] > pre[cur])
                printf("Down Edge!\n");
            else printf("Cross Edge!\n");
        }
    }
    post[cur] = ++tag;
}
```

```
/*==================================================*\
 | 无向图找桥
 | INIT: edge[][]邻接矩阵;vis[],pre[],anc[],bridge 置0;
 | CALL: dfs(0, -1, 1, n);
\*==================================================*/
int bridge, edge[V][V], anc[V], pre[V], vis[V];
void dfs(int cur, int father, int dep, int n)
{    // vertex: 0 ~ n-1
    if (bridge) return;
    vis[cur] = 1; pre[cur] = anc[cur] = dep;
    for (int i=0; i<n; ++i) if (edge[cur][i]) {
        if (i != father && 1 == vis[i]) {
            if (pre[i] < anc[cur])
                anc[cur] = pre[i];//back edge
        }
        if (0 == vis[i]) {          //tree edge
            dfs(i, cur, dep+1, n);
            if (bridge) return;
            if (anc[i] < anc[cur]) anc[cur] = anc[i];
            if (anc[i] > pre[cur]) { bridge = 1; return; }
        }
    }
    vis[cur] = 2;
}
```

```
/*==================================================*\
 | 无向图连通度(割)
 | INIT: edge[][]邻接矩阵;vis[],pre[],anc[],deg[]置为0;
 | CALL: dfs(0, -1, 1, n);
 | k=deg[0], deg[i]+1(i=1…n-1)为删除该节点后得到的连通图个数
 | 注意:0作为根比较特殊!
\*==================================================*/
int edge[V][V], anc[V], pre[V], vis[V], deg[V];
void dfs(int cur, int father, int dep, int n)
{// vertex: 0 ~ n-1
    int cnt = 0;
    vis[cur] = 1; pre[cur] = anc[cur] = dep;
    for (int i=0; i<n; ++i) if (edge[cur][i]) {
        if (i != father && 1 == vis[i]) {
            if (pre[i] < anc[cur])
                anc[cur] = pre[i];//back edge
        }
        if (0 == vis[i]) {          //tree edge
            dfs(i, cur, dep+1, n);
            ++cnt; // 分支个数
            if (anc[i] < anc[cur]) anc[cur] = anc[i];
            if ((cur==0 && cnt>1) ||
               (cnt!=0 && anc[i]>=pre[cur]))
                ++deg[cur]; // link degree of a vertex
        }
    }
    vis[cur] = 2;
}
```

```
/*==================================================*\
 | 最大团问题 DP + DFS
 | INIT: g[][]邻接矩阵;
 | CALL: res = clique(n);
\*==================================================*/
int g[V][V], dp[V], stk[V][V], mx;
int dfs(int n, int ns, int dep){
    if (0 == ns) {
        if (dep > mx) mx = dep;
        return 1;
    }
    int i, j, k, p, cnt;
    for (i = 0; i < ns; i++) {
        k = stk[dep][i]; cnt = 0;
        if (dep + n - k <= mx) return 0;
        if (dep + dp[k] <= mx) return 0;
        for (j = i + 1; j < ns; j++) {
            p = stk[dep][j];
            if (g[k][p]) stk[dep + 1][cnt++] = p;
        }
        dfs(n, cnt, dep + 1);
    }
    return 1;
}
int clique(int n){
    int i, j, ns;
    for (mx = 0,  i = n - 1; i >= 0; i--) {
        // vertex: 0 ~ n-1
        for (ns = 0, j = i + 1; j < n; j++)
            if (g[i][j]) stk[1][ ns++ ] = j;
        dfs(n, ns, 1); dp[i] = mx;
    }
    return mx;
}
```

```
/*==================================================*\
 | 欧拉路径 O(E)
 | INIT: adj[][]置为图的邻接表; cnt[a]为a点的邻接点个数;
 | CALL: elpath(0);   注意:不要有自向边
\*==================================================*/
int adj[V][V], idx[V][V], cnt[V], stk[V], top;
int path(int v){
    for (int w ; cnt[v] > 0; v = w) {
        stk[ top++ ] = v;
        w = adj[v][ --cnt[v] ];
        adj[w][ idx[w][v] ] = adj[w][ --cnt[w] ];
// 处理的是无向图—-边是双向的, 删除v->w后, 还要处理删除w->v
    }
    return v;
}
void elpath (int b, int n){         // begin from b
    int i, j;
    for (i = 0; i < n; ++i)         // vertex: 0 ~ n-1
        for (j = 0; j < cnt[i]; ++j)
            idx[i][ adj[i][j] ] = j;
    printf("%d", b);
    for (top = 0; path(b) == b && top != 0; ) {
        b = stk[ --top ];
        printf("-%d", b);
    }
    printf("\n");
}
```

```
/*==================================================*\
 | Dijkstra 数组实现 O (N^2)
 | Dijkstra --- 数组实现(在此基础上可直接改为STL的Queue实现)
 | lowcost[] --- beg到其他点的最近距离
 | path[] -- beg为根展开的树, 记录父亲结点
\*==================================================*/
#define INF 0x03F3F3F3F
const int N;
int path[N], vis[N];
void Dijkstra(int cost[][N], int lowcost[N], int n, int beg){
    int i, j, min;
    memset(vis, 0, sizeof(vis));
    vis[beg] = 1;
    for (i=0; i<n; i++){
        lowcost[i] = cost[beg][i]; path[i] = beg;
    }
    lowcost[beg] = 0;
```

```
    path[beg] = -1;   // 树根的标记
    int pre = beg;
    for (i=1; i<n; i++){
      min = INF;
      for (j=0; j<n; j++)
        // 下面的加法可能导致溢出,INF不能取太大
            if (vis[j]==0 &&
lowcost[pre]+cost[pre][j]<lowcost[j]){
                lowcost[j] = lowcost[pre] + cost[pre][j];
                path[j] = pre;
            }
        for (j=0; j<n; j++)
            if (vis[j] == 0 && lowcost[j] < min){
                min = lowcost[j]; pre = j;
            }
      vis[pre] = 1;
    }
}
/*==================================================*\
 | Dijkstra O(E * log E)
 | INIT: 调用init(nv, ne)读入边并初始化;
 | CALL: dijkstra(n, src); dist[i]为src到i的最短距离
\*==================================================*/
#define typec int                        // type of cost
const typec inf = 0x3f3f3f3f;            // max of cost
typec cost[E], dist[V];
int e, pnt[E], nxt[E], head[V], prev[V], vis[V];
struct qnode {
    int v; typec c;
    qnode (int vv = 0, typec cc = 0) : v(vv), c(cc) {}
    bool operator < (const qnode& r) const { return c>r.c; }
};
void dijkstra(int n, const int src){
    qnode mv;
    int i, j, k, pre;
    priority_queue<qnode> que;
    vis[src] = 1; dist[src] = 0;
    que.push(qnode(src, 0));
    for (pre = src, i=1; i<n; i++) {
        for (j = head[pre]; j != -1; j = nxt[j]) {
            k = pnt[j];
            if (vis[k] == 0 &&
                dist[pre] + cost[j] < dist[k]){
                dist[k] = dist[pre] + cost[j];
                que.push(qnode(pnt[j], dist[k]));
                prev[k] = pre;
            }
        }
        while (!que.empty() && vis[que.top().v] == 1)
            que.pop();
        if (que.empty()) break;
        mv = que.top(); que.pop();
        vis[pre = mv.v] = 1;
    }
}
inline void addedge(int u, int v, typec c){
    pnt[e] = v; cost[e] = c; nxt[e] = head[u]; head[u] = e++;
}
void init(int nv, int ne){
    int i, u, v; typec c;
    e = 0;
    memset(head, -1, sizeof(head));
    memset(vis, 0, sizeof(vis));
    memset(prev, -1, sizeof(prev));
    for (i = 0; i < nv; i++) dist[i] = inf;
    for (i = 0; i < ne; ++i) {
        scanf("%d%d%d", &u, &v, &c);// %d: type of cost
        addedge(u, v, c);     // vertex: 0 ~ n-1, 单向边
    }
}
/*==================================================*\
 | BellmanFord 单源最短路 O(VE)
 | 能在一般情况下，包括存在负权边的情况下，解决单源最短路径问题
 | INIT: edge[E][3]为边表
 | CALL: bellman(src);有负环返回0;dist[i]为src到i的最短距
 | 可以解决差分约束系统: 需要首先构造约束图，构造不等式时>=表示求最
小值, 作为最长路，<=表示求最大值, 作为最短路  (v-u <= c:a[u][v] =
c)
\*==================================================*/
#define typec int                      // type of cost
const typec inf=0x3f3f3f3f;            // max of cost
int n, m, pre[V], edge[E][3];
typec dist[V];
int relax (int u, int v, typec c){
    if (dist[v] > dist[u] + c) {
        dist[v] = dist[u] + c;
        pre[v] = u; return 1;
    }
    return 0;
}
int bellman (int src){
    int i, j;
    for (i=0; i<n; ++i) {
        dist[i] = inf; pre[i] = -1;
    }
    dist[src] = 0; bool flag;
    for (i=1; i<n; ++i){
      flag = false; // 优化
      for (j=0; j<m; ++j) {
        if( 1 == relax(edge[j][0], edge[j][1],
edge[j][2])  ) flag = true;
      }
      if( !flag ) break; }
    for (j=0; j<m; ++j) {
        if (1 == relax(edge[j][0], edge[j][1], edge[j][2]))
            return 0;  // 有负圈
    }
    return 1;
}
/*==================================================*\
 | SPFA(Shortest Path Faster Algorithm)
Bellman-Ford算法的一种队列实现，减少了不必要的冗余计算。 它可以在
O(kE)的时间复杂度内求出源点到其他所有点的最短路径，可以处理负边。
原理：只有那些在前一遍松弛中改变了距离估计值的点，才可能引起他们的邻
接点的距离估计值的改变。
判断负权回路：记录每个结点进队次数，超过|V|次表示有负权。
\*==================================================*/
// POJ 3159 Candies
const int INF = 0x3F3F3F3F;
const int V = 30001;
const int E = 150001;
int pnt[E], cost[E], nxt[E];
int e, head[V];  int dist[V];  bool vis[V];
int main(void){
    int  n, m;
    while( scanf("%d%d", &n, &m) != EOF ){
        int  i, a, b, c;
        e = 0;
        memset(head, -1, sizeof(head));
        for( i=0; i < m; ++i )
        {// b-a <= c, 有向边(a, b):c , 边的方向! ! !
            scanf("%d%d%d", &a, &b, &c);
            addedge(a, b, c);
        }
        printf("%d\n", SPFA(1, n));
    }
    return 0;
}
int relax(int u, int v, int c){
    if( dist[v] > dist[u] + c ) {
        dist[v] = dist[u] + c; return 1;
    }
    return 0;
}
inline void addedge(int u, int v, int c){
    pnt[e] = v; cost[e] = c; nxt[e] = head[u]; head[u] = e++;
}
int SPFA(int src, int n)
{ // 此处用堆栈实现，有些时候比队列要快
    int  i;
    for( i=1; i <= n; ++i ){ // 顶点1...n
        vis[i] = 0; dist[i] = INF;
    }
    dist[src] = 0;
    int  Q[E], top = 1;
    Q[0] = src; vis[src] = true;
    while( top ){
        int  u, v;
        u = Q[--top]; vis[u] = false;
        for( i=head[u]; i != -1; i=nxt[i] ){
```

```
            v = pnt[i];
            if( 1 == relax(u, v, cost[i]) && !vis[v] ) {
                Q[top++] = v; vis[v] = true;
            }
        }
    }
    return dist[n];
}
// 队列实现，而且有负权回路判断—POJ 3169 Layout
#define swap(t, a, b) (t=a, a=b, b=t)
const int INF = 0x3F3F3F3F;
const int V = 1001;
const int E = 20001;
int pnt[E], cost[E], nxt[E];
int e, head[V], dist[V];
bool vis[V];
int cnt[V]; // 入队列次数
int main(void){
    int n, ml, md;
    while( scanf("%d%d%d", &n, &ml, &md) != EOF ){
        int i, a, b, c, t;
        e = 0;
        memset(head, -1, sizeof(head));
        for( i=0; i < ml; ++i ) // 边方向！！！
        {// 大-小<=c, 有向边(小, 大):c
            scanf("%d%d%d", &a, &b, &c);
            if( a > b) swap(t, a, b);
            addedge(a, b, c);
        }
        for( i=0; i < md; ++i )
        {// 大-小>=c ==> 小-大<=-c, 有向边(大, 小):-c
            scanf("%d%d%d", &a, &b, &c);
            if( a < b ) swap(t, a, b);
             addedge(a, b, -c);
        }
        //for( i=1; i <= n; ++i ) printf("%d\n", dist[i]);
        printf("%d\n", SPFA(1, n));
    }
    return 0;
}
int relax(int u, int v, int c){
    if( dist[v] > dist[u] + c ) {
        dist[v] = dist[u] + c; return 1;
    }
    return 0;
}
inline void addedge(int u, int v, int c){
    pnt[e] = v; cost[e] = c; nxt[e] = head[u]; head[u] = e++;
}
int SPFA(int src, int n){// 此处用队列实现
    int i;
    memset(cnt, 0, sizeof(cnt)); // 入队次数
    memset(vis, false, sizeof(vis));
    for( i=1; i <= n; ++i ) dist[i] = INF;
    dist[src] = 0;
    queue<int> Q;
    Q.push(src); vis[src] = true; ++cnt[src];
    while( !Q.empty() ){
        int u, v;
        u = Q.front(); Q.pop(); vis[u] = false;
        for( i=head[u]; i != -1; i=nxt[i] ){
            v = pnt[i];
            if( 1 == relax(u, v, cost[i]) && !vis[v] ) {
                Q.push(v); vis[v] = true;
                if( (++cnt[v]) > n ) return -1; // cnt[i]
为入队列次数，用来判断是否存在负权回路
            }
        }
    }
    if( dist[n] == INF ) return -2; // src与n不可达，有些题
目可省！！！
    return dist[n]; // 返回src到n的最短距离，根据题意不同而改变
}
/*==================================================*\
 | 第 K 短路（Dijkstra）
 | dij变形，可以证明每个点经过的次数为小于等于K，所有把dij的数组dist
 | 由一维变成2维，记录经过该点1次，2次。。。k次的最小值。
 | 输出dist[n-1][k]即可
\*==================================================*/
//WHU1603
int g[1010][1010];
int n,m,x;
const int INF=1000000000;
int v[1010];
int dist[1010][20];
int main(){
   while (scanf("%d%d%d",&n,&m,&x)!=EOF){
      for (int i=1;i<=n;i++)
         for (int j=1;j<=n;j++)
            g[i][j]=INF;
      for (int i=0;i<m;i++){
         int p,q,r;
         scanf("%d%d%d",&p,&q,&r);
         if (r<g[p][q]) g[p][q]=r;
      }
      for (int i=1;i<=n;i++){
         v[i]=0;
         for (int j=0;j<=x;j++)
            dist[i][j]=INF;
      }
      dist[1][0]=0;
      dist[0][0]=INF;
      while (1){
         int k=0;
         for (int i=1;i<=n;i++)
            if (v[i]<x && dist[i][v[i]]<dist[k][0])
               k=i;
         if (k==0) break;
         if (k==n && v[n]==x-1) break;
         for (int i=1;i<=n;i++){
            if (v[i]<x &&
dist[k][v[k]]+g[k][i]<dist[i][x]){
               dist[i][x]=dist[k][v[k]]+g[k][i];
               for (int j=x;j>0;j--)
                  if (dist[i][j]<dist[i][j-1])
                     swap(dist[i][j],dist[i][j-1]);
            }
          }
         v[k]++;
      }
      if (dist[n][x-1]<INF) printf("%d\n",dist[n][x-1]);
      else printf("-1\n");
   }
   return 0;
}
/*==================================================*\
 | 第 K 短路（A*）
 | A* 估价函数 fi为到当前点走过的路经长度， hi为该点到终点的长度
 | gi=hi+fi;
\*==================================================*/
//WHU1603
int n,m,x,ct;
int g[1010][1010],gr[1010][1010];
int dist[1010],v[1010];
const int INF=1000000000;
struct node{
    int id,fi,gi;
    friend bool operator <(node a,node b){
        if (a.gi==b.gi) return a.fi>b.fi;
        return a.gi>b.gi;
    }
}s[2000010];
int init(){
    for (int i=0;i<=n;i++){
        dist[i]=INF;
        v[i]=1;
    }
    dist[n-1]=0;
    for (int i=0;i<n;i++){
        int k=n;
        for (int j=0;j<n;j++)
            if (v[j] && dist[j]<dist[k]) k=j;
        if (k==n) break;
        v[k]=0;
        for (int j=0;j<n;j++)
            if (v[j] && dist[k]+gr[k][j]<dist[j])
                dist[j]=dist[k]+gr[k][j];
    }
    return 1;
}
```

```
int solve(){
    if (dist[0]==INF) return -1;
    ct=0;
    s[ct].id=0;
    s[ct].fi=0;
    s[ct++].gi=dist[0];
    int cnt=0;
    while (ct){
        int id=s[0].id,fi=s[0].fi,gi=s[0].gi;
        if (id==n-1) cnt++;
        if (cnt==x) return fi;
        pop_heap(s,s+ct);
        ct--;
        for (int j=0;j<n;j++)
            if (g[id][j]<INF){
                s[ct].id=j;
                s[ct].fi=fi+g[id][j];
                s[ct++].gi=s[ct].fi+dist[j];
                push_heap(s,s+ct);
            }
    }
    return -1;
}
int main(){
    while (scanf("%d%d%d",&n,&m,&x)!=EOF){
        for (int i=0;i<n;i++)
            for (int j=0;j<n;j++)
                gr[i][j]=g[i][j]=INF;
        for (int i=0;i<m;i++){
            int x,y,z;
            scanf("%d%d%d",&x,&y,&z);
            x--,y--;
            g[x][y]<?=z;
            gr[y][x]<?=z;
        }
        init();
        printf("%d\n",solve());
    }
    return 0;
}
/*==================================================*\
 | Prim 求 MST
 | INIT: cost[][]耗费矩阵(inf为无穷大);
 | CALL: prim(cost, n); 返回-1代表原图不连通;
\*==================================================*/
#define typec int                    // type of cost
const typec inf = 0x3f3f3f3f;        // max of cost
int vis[V]; typec lowc[V];
typec prim(typec cost[][V], int n)   // vertex: 0 ~ n-1
{
    int i, j, p;
    typec minc, res = 0;
    memset(vis, 0, sizeof(vis));
    vis[0] = 1;
    for (i=1; i<n; i++) lowc[i] = cost[0][i];
    for (i=1; i<n; i++) {
      minc = inf; p = -1;
      for (j=0; j<n; j++)
        if (0 == vis[j] && minc > lowc[j]) {
            minc = lowc[j]; p = j;
        }
      if (inf == minc) return -1;        // 原图不连通
      res += minc; vis[p] = 1;
      for (j=0; j<n; j++)
            if (0 == vis[j] && lowc[j] > cost[p][j])
                lowc[j] = cost[p][j];
    }
    return res;
}
/*==================================================*\
 | 次小生成树 O(V^2)
\*==================================================*/
```

结论　次小生成树可由最小生成树换一条边得到.

证明：　可以证明下面一个强一些的结论：

T是某一棵最小生成树，T0是任一棵异于T的树，通过变换T0 --> T1 --> T2 --> ... --> Tn (T) 变成最小生成树.所谓的变换是，每次把T_i中的某条边换成T中的一条边，而且树T_(i+1)的权小于等于T_i的权.

具体操作是：

step 1. 在T_i中任取一条不在T中的边u_v.

step 2. 把边u_v去掉，就剩下两个连通分量A和B，在T中，必有唯一的边u'_v' 连结A和B.

step 3. 显然u'_v'的权比u_v小 (否则,u_v就应该在T中).把u'_v'替换u_v即得树T_(i+1).

特别地：取Tn为任一棵次小生成树，T_(n-1) 也就是次小生成树且跟T差一条边. 结论得证.

算法：只要充分利用以上结论，即得V^2的算法. 具体如下：

step 1. 先用prim求出最小生成树T. 在prim的同时，用一个矩阵max[u][v] 记录在T中连结任意两点u,v的唯一的路中权值最大的那条边的权值.(注意这里). 这是很容易做到的，因为prim是每次增加一个结点s，而设已经标号了的结点集合为W，则W中所有的结点到s的路中的最大权值的边就是当前加入的这条边. step 1 用时 O(V^2).

step 2. 枚举所有不在T中的边u_v，加入边u_v替换权为max[u][v]的边. 不断更新求最小值，即次小生成树. step 2 用时 O(E).故总时间为O(V^2).

```
/*==================================================*\
 | 最小生成森林问题(k 颗树)O(mlogm).
\*==================================================*/
```

数据结构：并查集　算法：改进Kruskal

根据Kruskal算法思想，图中的生成树在连完第n-1条边前，都是一个最小生成森林，每次贪心的选择两个不属于同一连通分量的树（如果连接一个连通分量，因为不会减少块数，那么就是不合算的）且用最“便宜”的边连起来，连接n-1次后就形成了一棵MST，n-2次就形成了一个两棵树的最小生成森林，n-3，……，n-k此后就形成了k颗树的最小生成森林，就是题目要求求解的。

```
/*==================================================*\
 | 有向图最小树形图
 | INIT: eg置为边表; res置为0; cp[i]置为i;
 | CALL: dirtree(root, nv, ne); res是结果;
\*==================================================*/
#define typec int                 // type of res
const typec inf = 0x3f3f3f3f;     // max of res
typec res, dis[V];
int to[V], cp[V], tag[V];
struct Edge { int u, v; typec c; } eg[E];
int iroot(int i){
    if (cp[i] == i) return i;
    return cp[i] = iroot(cp[i]);
}
int dirtree(int root, int nv, int ne) // root: 树根
{   // vertex: 0 ~ n-1
    int i, j, k, circle = 0;
    memset(tag, -1, sizeof(tag));
    memset(to, -1, sizeof(to));
    for (i = 0; i < nv; ++i) dis[i] = inf;
    for (j = 0; j < ne; ++j) {
        i = iroot(eg[j].u); k = iroot(eg[j].v);
        if (k != i && dis[k] > eg[j].c) {
            dis[k] = eg[j].c;
            to[k] = i;
        }
    }
    to[root] = -1; dis[root] = 0; tag[root] = root;
    for (i = 0; i < nv; ++i) if (cp[i] == i && -1 == tag[i]){
        j = i;
        for ( ; j != -1 && tag[j] == -1; j = to[j])
            tag[j] = i;
        if (j == -1) return 0;
        if (tag[j] == i) {
            circle = 1; tag[j] = -2;
            for (k = to[j]; k != j; k = to[k]) tag[k] = -2;
        }
    }
    if (circle) {
        for (j = 0; j < ne; ++j) {
            i = iroot(eg[j].u); k = iroot(eg[j].v);
            if (k != i && tag[k] == -2) eg[j].c -= dis[k];
        }
        for (i = 0; i < nv; ++i) if (tag[i] == -2) {
            res += dis[i]; tag[i] = 0;
            for (j = to[i]; j != i; j = to[j]) {
                res += dis[j]; cp[j] = i; tag[j] = 0;
            }
        }
        if (0 == dirtree(root, nv, ne)) return 0;
    } else {
        for (i = 0; i < nv; ++i) if (cp[i] == i) res += dis[i];
    }
    return 1;        // 若返回0代表原图不连通
}
/*==================================================*\
```

```
| Minimal Steiner Tree
| G(V, E), A是V的一个子集, 求至少包含A中所有点的最小子树.
| 时间复杂度: O(N^3 + N * 2^A * (2^A + N))
| INIT: d[][]距离矩阵; id[]置为集合A中点的标号;
| CALL: steiner(int n, int a);
| main()函数解决的题目: Ticket to Ride, NWERC 2006/2007
| 给4个点对(a1, b1) ... (a4, b4),
| 求min(sigma(dist[ai][bi])),其中重复的路段只能算一次.
| 这题要找出一个steiner森林, 最后要对森林中树的个数进行枚举
\*==================================================*/
#define typec int                      // type of cost
const typec inf = 0x3f3f3f3f;          // max of cost
int vis[V], id[A];              //id[]: A中点的标号
typec d[V][V], dp[1<<A][V];//dp[i][v]: 点v到点集i的最短距离
void steiner(int n, int a){
    int i, j, k, mx, mk, top = (1 << a);
    for (k = 0; k < n; k++) for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (d[i][j] > d[i][k] + d[k][j])
                d[i][j] = d[i][k] + d[k][j];
    for (i = 0; i < a; i++) {          // vertex: 0 ~ n-1
        for (j = 0; j < n; j++)
            dp[1 << i][j] = d[j][ id[i] ];
    }
    for (i = 1; i < top; i++) {
        if ( 0 == (i & (i - 1)) ) continue;
        memset(vis, 0, sizeof(vis));
        for (k = 0; k < n; k++) {           // init
            for (dp[i][k] = inf, j = 1; j < i; j++)
                if ((i | j) == i &&
                    dp[i][k] > dp[j][k] + dp[i - j][k])
                    dp[i][k] = dp[j][k] + dp[i - j][k];
        }
        for (j = 0; mx = inf, j < n; j++) { // update
            for (k = 0; k < n; k++)
                if (dp[i][k] <= mx && 0 == vis[k])
                    mx = dp[i][mk = k];
            for (k = 0, vis[mk] = 1; k < n; k++)
                if (dp[i][mk] > dp[i][k] + d[k][mk])
                    dp[i][mk] = dp[i][k] + d[k][mk];
        }
    }
}
int main(void){
    int n, a = 8;
    // TODO: read data;
    steiner(n, a);
    // enum to find the result
    for (i = 0, b = inf; z = 0, i < 256; b>z ? b=z : b, i++)
        for (j = 0; y = 0, j < 4; z += !!y * dp[y][x], j++)
            for (k = 0; k < 8; k += 2)
                if ((i >> k & 3) == j)
                    y += 3 << k, x = id[k];
    // TODO: cout << b << endl;
    return 0;
}
/*==================================================*\
| Tarjan强连通分量
| INIT: vec[]为邻接表; stop, cnt, scnt置0; pre[]置-1;
| CALL: for(i=0; i<n; ++i) if(-1==pre[i]) tarjan(i, n);
\*==================================================*/
vector<int> vec[V];
int id[V], pre[V], low[V], s[V], stop, cnt, scnt;
void tarjan(int v, int n)              // vertex: 0 ~ n-1
{
    int t, minc = low[v] = pre[v] = cnt++;
    vector<int>::iterator pv;
    s[stop++] = v;
    for (pv = vec[v].begin(); pv != vec[v].end(); ++pv) {
        if(-1 == pre[*pv]) tarjan(*pv, n);
        if(low[*pv] < minc) minc=low[*pv];
    }
    if(minc < low[v]) {
        low[v] = minc; return;
    }
   do {
        id[t = s[--stop]] = scnt; low[t] = n;
   } while(t != v);
    ++scnt;                          // 强连通分量的个数
}
```

```
/*==================================================*\
| 弦图判断
| INIT: g[][]置为邻接矩阵;
| CALL: mcs(n); peo(n);
| 第一步: 给节点编号 mcs(n)
|     设已编号的节点集合为A, 未编号的节点集合为B
|     开始时A为空, B包含所有节点.
|     for num=n-1 downto 0 do {
|         在B中找节点x, 使与x相邻的在A集合中的节点数最多,
|         将x编号为num, 并从B移入A.
|     }
| 第二步: 检查 peo(n)
|     for num=0 to n-1 do {
|         对编号为num的点x, 设所有编号>num且与x相邻的点集为C
|         在C中找出编号最小的节点y,
|         若C中存在点z!=y, 使得y与z之间无边, 则此图不是弦图.
|     }
| 检查完了, 则此图是弦图.
\*==================================================*/
int g[V][V], order[V], inv[V], tag[V];
void mcs(int n){
    int i, j, k;
    memset(tag, 0, sizeof(tag));
    memset(order, -1, sizeof(order));
    for (i = n - 1; i >= 0; i--) {  // vertex: 0 ~ n-1
        for (j = 0; order[j] >= 0; j++) ;
        for (k = j + 1; k < n; k++)
            if (order[k] < 0 && tag[k] > tag[j]) j = k;
        order[j] = i, inv[i] = j;
        for (k = 0; k < n; k++) if (g[j][k]) tag[k]++;
    }
}
int peo(int n){
    int i, j, k, w, min;
    for (i = n - 2; i >= 0; i--) {
        j = inv[i], w = -1, min = n;
        for (k = 0; k < n; k++)
            if (g[j][k] && order[k] > order[j] &&
                order[k] < min)
                min = order[k], w=k;
        if (w < 0) continue;
        for (k = 0; k < n; k++)
            if (g[j][k] && order[k] > order[w] && !g[k][w])
                return 0; // no
    }
    return 1;               // yes
}
/*==================================================*\
| 弦图的perfect elimination点排列
| INIT: g[][]置为邻接矩阵;
| CALL: cardinality(n); tag[i]为排列中第i个点的标号;
|     The graph with the property mentioned above
| is called chordal graph. A permutation s = [v1 , v2,
| ..., vn] of the vertices of such graph is called a
| perfect elimination order if each vi is a simplicial
| vertex of the subgraph of G induced by {vi ,..., vn}.
| A vertex is called simplicial if its adjacency set
| induces a complete subgraph, that is, a clique (not
| necessarily maximal). The perfect elimination order
| of a chordal graph can be computed as the following:
\*==================================================*/
procedure maximum cardinality search(G, s)
   for all vertices v of G do
      set label[v] to zero
   end for
   for all i from n downto 1 do
      choose an unnumbered vertex v with largest label
      set s(v) to i{number vertex v}
      for all unnumbered vertices w adjacent to vertex v do
         increment label[w] by one
      end for
   end for
end procedure

int tag[V], g[V][V], deg[V], vis[V];
void cardinality(int n)
{
    int i, j, k;
    memset(deg, 0, sizeof(deg));
    memset(vis, 0, sizeof(vis));
```

```
	for (i = n - 1; i >= 0; i--) {
		for (j = 0, k = -1; j < n; j++) if (0 == vis[j]) {
			if (k == -1 || deg[j] > deg[k]) k = j;
		}
		vis[k] = 1, tag[i] = k;
		for (j = 0; j<n; j++)
			if (0 == vis[j] && g[k][j]) deg[j]++;
	}
}
/*==================================================*\
 | 稳定婚姻问题 O(n^2)
\*==================================================*/
const int N = 1001;
struct People{
	bool state;
	int opp, tag;
	int list[N];	// man使用
	int priority[N]; // woman使用，有必要的话可以和list合并，以节省空间
	void Init(){ state = tag = 0; }
}man[N], woman[N];
struct R{
	int  opp; int  own;
}requst[N];
int n;
void Input(void);
void Output(void);
void stableMatching(void);
int main(void){
	//...
		Input();
		stableMatching();
		Output();
	//...
	return 0;
}
void Input(void){
	scanf("%d\n", &n);
	int  i, j, ch;
	for( i=0; i < n; ++i ) {
		man[i].Init();
		for( j=0; j < n; ++j ){ //按照man的意愿递减排序
			scanf("%d", &ch); man[i].list[j] = ch-1;
		}
	}
	for( i=0; i < n; ++i ) {
		woman[i].Init();
		for( j=0; j < n; ++j ){ //按照woman的意愿递减排序，但是，存储方法与man不同！！！！
			scanf("%d", &ch); woman[i].priority[ch-1] = j;
		}
	}
}
void stableMatching(void){
	int  k;
	for( k=0; k < n; +k ){
		int  i, id = 0;
		for( i=0; i < n; ++i )
			if( man[i].state == 0 ){
				requst[id].opp = man[i].list[ man[i].tag ];
				requst[id].own = i;
				man[i].tag += 1; ++id;
			}
		if( id == 0 ) break;
		for( i=0; i < id; ++i ){
			if( woman[requst[i].opp].state == 0 ){
				woman[requst[i].opp].opp = requst[i].own;
				woman[requst[i].opp].state = 1;
				man[requst[i].own].state = 1;
				man[requst[i].own].opp  = requst[i].opp;
			}
			else{
	if( woman[requst[i].opp].priority[ woman[requst[i].opp].opp ] > woman[requst[i].opp].priority[requst[i].own] ){ //
				man[ woman[requst[i].opp].opp ].state = 0;
				woman[ requst[i].opp ].opp = requst[i].own;
				man[requst[i].own].state = 1;
				man[requst[i].own].opp  = requst[i].opp;
				}
			}
		}
	}
}
void Output(void){
	for( int i=0; i < n; ++i ) printf("%d\n", man[i].opp+1);
}
/*==================================================*\
 | 拓扑排序
 | INIT:edge[][]置为图的邻接矩阵;count[0…i…n-1]:顶点i的入度.
\*==================================================*/
void TopoOrder(int n){
	int  i, top = -1;
	for( i=0; i < n; ++i )
		if( count[i] == 0 ){ // 下标模拟堆栈
			count[i] = top; top = i;
		}
	for( i=0; i < n; ++i )
		if( top == -1 ) { printf("存在回路\n"); return ; }
		else{
			int j = top; top = count[top];
			printf("%d", j);
			for( int k=0; k < n; ++k )
				if( edge[j][k] && (--count[k]) == 0 ){
					count[k] = top; top = k;
				}
		}
}
/*==================================================*\
 | 无向图连通分支(dfs/bfs 邻接阵)
 | DFS / BFS / 并查集
\*==================================================*/
/*==================================================*\
 | 有向图强连通分支(dfs/bfs 邻接阵)O(n^2)
\*==================================================*/
//返回分支数,id返回1..分支数的值
//传入图的大小n和邻接阵mat,不相邻点边权0
#define MAXN 100
void search(int n,int mat[][MAXN],int* dfn,int* low,int now,int& cnt,int& tag,int* id,int* st,int& sp){
	int i,j;
	dfn[st[sp++]=now]=low[now]=++cnt;
	for (i=0;i<n;i++)
		if (mat[now][i]){
			if (!dfn[i]){
	search(n,mat,dfn,low,i,cnt,tag,id,st,sp);
				if (low[i]<low[now])
					low[now]=low[i];
			}
			else if (dfn[i]<dfn[now]){
				for (j=0;j<sp&&st[j]!=i;j++);
				if (j<cnt&&dfn[i]<low[now])
					low[now]=dfn[i];
			}
		}
	if (low[now]==dfn[now])
		for (tag++;st[sp]!=now;id[st[--sp]]=tag);
}
int find_components(int n,int mat[][MAXN],int* id){
	int ret=0,i,cnt,sp,st[MAXN],dfn[MAXN],low[MAXN];
	for (i=0;i<n;dfn[i++]=0);
	for (sp=cnt=i=0;i<n;i++)
		if (!dfn[i])
			search(n,mat,dfn,low,i,cnt,ret,id,st,sp);
	return ret;
}
//有向图强连通分支,bfs邻接阵形式,O(n^2)
//返回分支数,id返回1..分支数的值
//传入图的大小n和邻接阵mat,不相邻点边权0
#define MAXN 100
int find_components(int n,int mat[][MAXN],int* id){
	int ret=0,a[MAXN],b[MAXN],c[MAXN],d[MAXN],i,j,k,t;
```

```
    for (k=0;k<n;id[k++]=0);
    for (k=0;k<n;k++)
        if (!id[k]){
            for (i=0;i<n;i++)
                a[i]=b[i]=c[i]=d[i]=0;
            a[k]=b[k]=1;
            for (t=1;t;)
                for (t=i=0;i<n;i++){
                    if (a[i]&&!c[i])
                        for (c[i]=t=1,j=0;j<n;j++)
                            if (mat[i][j]&&!a[j])
                                a[j]=1;
                    if (b[i]&&!d[i])
                        for (d[i]=t=1,j=0;j<n;j++)
                            if (mat[j][i]&&!b[j])
                                b[j]=1;
                }
            for (ret++,i=0;i<n;i++)
                if (a[i]&b[i])
                    id[i]=ret;
        }
    return ret;
}
/*==================================================*\
  | 有向图最小点基(邻接阵)O(n^2)
  | 点基B满足：对于任意一个顶点Vj，一定存在B中的一个Vi,使得Vi是Vj
  | 的前代。
\*==================================================*/
//返回点基大小和点基
//传入图的大小n和邻接阵mat,不相邻点边权0
//需要调用强连通分支
#define MAXN 100
int base_vertex(int n,int mat[][MAXN],int* sets){
    int ret=0,id[MAXN],v[MAXN],i,j;
    j=find_components(n,mat,id);
    for (i=0;i<j;v[i++]=1);
    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
            if (id[i]!=id[j]&&mat[i][j])
                v[id[j]-1]=0;
    for (i=0;i<n;i++)
        if (v[id[i]-1])
            v[id[sets[ret++]=i]-1]=0;
    return ret;
}
/*==================================================*\
 | Floyd 求最小环
\*==================================================*/
朴素算法
令e(u,v)表示u和v之间的连边，令min(u,v)表示删除u和v之间的连边之后
u和v之间的最短路，最小环则是min(u, v) + e(u, v). 时间复杂度是
O(EV^2).
改进算法
在floyd的同时，顺便算出最小环
g[i][j]=i, j之间的边长
dist:=g;
for k:=1 to n do
begin
    for i:=1 to k-1 do
      for j:=i+1 to k-1 do
        answer:=min(answer, dist[i][j]+g[i][k]+g[k][j]);
    for i:=1 to n do
      for j:=1 to n do

dist[i][j]:=min(dist[i][j],dist[i][k]+dist[k][j]);
end;
最小环改进算法的证明
一个环中的最大结点为k(编号最大), 与他相连的两个点为i, j, 这个环的最
短长度为g[i][k]+g[k][j]+i到j的路径中所有结点编号都小于k的最短路
径长度. 根据floyd的原理, 在最外层循环做了k-1次之后, dist[i][j]则
代表了i到j的路径中所有结点编号都小于k的最短路径
综上所述,该算法一定能找到图中最小环.
const int INF = 1000000000;
const int N = 110;
int  n, m;        // n:节点个数, m:边的个数
int  g[N][N];     // 无向图
int  dist[N][N];  // 最短路径
int  r[N][N];     // r[i][j]: i到j的最短路径的第一步
int  out[N], ct;  // 记录最小环
int solve(int i, int j, int k){// 记录最小环
    ct = 0;
    while ( j != i ){
        out[ct++] = j;
        j = r[i][j];
    }
    out[ct++] = i; out[ct++] = k;
    return 0;
}
int main(void){
    while( scanf("%d%d", &n, &m) != EOF ){
        int  i, j, k;
        for ( i=0; i < n; i++ )
            for ( j=0; j < n; j++ ){
                g[i][j] = INF; r[i][j] = i;
            }
        for ( i=0; i < m; i++ ){
            int x, y, l;
            scanf("%d%d%d", &x, &y, &l);
            --x; --y;
            if ( l < g[x][y] ) g[x][y] = g[y][x] = l;
        }
        memmove(dist, g, sizeof(dist));
        int Min = INF; // 最小环
        for ( k=0; k < n; k++ ){//Floyd
            for ( i=0; i < k; i++ )// 一个环中的最大结点为k(编
号最大)
                if ( g[k][i] < INF )
                    for ( j=i+1; j < k; j++ )
                        if ( dist[i][j] < INF && g[k][j]
< INF && Min > dist[i][j]+g[k][i]+g[k][j] ){
                            Min =
dist[i][j]+g[k][i]+g[k][j];
                            solve(i, j, k); // 记录最小环
                        }
            for ( i=0; i < n; i++ )
                if ( dist[i][k] < INF )
                    for ( j=0; j < n; j++ )
                        if ( dist[k][j] < INF &&
dist[i][j] > dist[i][k]+dist[k][j] ){
                            dist[i][j] =
dist[i][k]+dist[k][j];
                            r[i][j] = r[k][j];
                        }
        }
        if ( Min < INF ){
            for ( ct--; ct >= 0; ct-- ){
                printf("%d", out[ct]+1);
                if ( ct ) printf(" ");
            }
        }
        else printf("No solution.");
        printf("\n");
    }
    return 0;
}
/*==================================================*\
 | 2-sat 问题
 * N个集团，每个集团2个人，现在要想选出尽量多的人，
 * 且每个集团只能选出一个人。如果两人有矛盾，他们不能同时被选中
 * 问最多能选出多少人
\*==================================================*/
const int MAXN=3010;
int n,m;
int g[3010][3010],ct[3010],f[3010];
int x[3010],y[3010];
int prev[MAXN], low[MAXN], stk[MAXN], sc[MAXN];
int cnt[MAXN];
int cnt0, ptr, cnt1;

void dfs(int w){
    int min(0);
    prev[w] = cnt0++;
    low[w] = prev[w];
    min = low[w];
    stk[ptr++] = w;
    for(int i = 0; i < ct[w]; ++i){
        int t = g[w][i];
        if(prev[t] == -1)
            dfs(t);
        if(low[t] < min)
```

```
            min = low[t];
    }
    if(min < low[w]){
        low[w] = min;
        return;
    }
    do{
        int v = stk[--ptr];
        sc[v] = cnt1;
        low[v] = MAXN;
    }while(stk[ptr] != w);
    ++cnt1;
}
void Tarjan(int N){
    //传入N为点数，结果保存在sc数组中，同一标号的点在同一个强连通分量内，
    //强连通分量数为cnt1
    cnt0 = cnt1 = ptr = 0;
    int i;
    for(i = 0; i < N; ++i)
        prev[i] = low[i] = -1;
    for(i = 0; i < N; ++i)
        if(prev[i] == -1)
            dfs(i);
}
int solve(){
    Tarjan(n);
    for (int i=0;i<n;i++){
        if (sc[i]==sc[f[i]]) return 0;
    }
    return 1;
}
int check(int Mid){
    for (int i=0;i<n;i++)
        ct[i]=0;
    for (int i=0;i<Mid;i++){
        g[f[x[i]]][ct[f[x[i]]]++]=y[i];
        g[f[y[i]]][ct[f[y[i]]]++]=x[i];
    }
    return solve();
}
int main(){
    while (scanf("%d%d",&n,&m)!=EOF && n+m){
        for (int i=0;i<n;i++){
            int p,q;
            scanf("%d%d",&p,&q);
            f[p]=q,f[q]=p;
        }
        for (int i=0;i<m;i++) scanf("%d%d",&x[i],&y[i]);
        n*=2;
        int Min=0,Max=m+1;
        while (Min+1<Max){
            int Mid=(Min+Max)/2;
            if (check(Mid)) Min=Mid;
            else Max=Mid;
        }
        printf("%d\n",Min);
    }
    return 0;
}
```

Network 网络流

```
/*==================================================*\
 | 二分图匹配（匈牙利算法 DFS 实现）
 | INIT: g[][]邻接矩阵;
 | CALL: res = MaxMatch();
 | 优点：实现简洁容易理解，适用于稠密图，DFS找增广路快。
 | 找一条增广路的复杂度为O(E)，最多找V条增广路，故时间复杂度为O(VE)
\*==================================================*/
const int MAXN = 1000;
int uN, vN;              // u, v数目，要初始化！！！
bool g[MAXN][MAXN];     // g[i][j] 表示xi与yj相连
int xM[MAXN], yM[MAXN]; // 输出量
bool chk[MAXN];          // 辅助量检查某轮y[v]是否被check
bool SearchPath(int u){
    int v;
   for(v = 0; v < vN; v++)
        if(g[u][v] && !chk[v])
        {
            chk[v] = true;
            if(yM[v] == -1 || SearchPath(yM[v]))
            {
                yM[v] = u; xM[u] = v;
                return true ;
            }
        }
    return false ;
}
int MaxMatch(){
    int u, ret = 0 ;
   memset(xM, -1, sizeof (xM));
   memset(yM, -1, sizeof (yM));
   for(u = 0; u < uN; u++)
        if(xM[u] == -1){
            memset(chk, false, sizeof (chk));
          if(SearchPath(u)) ret++;
        }
  return ret;
}
/*==================================================*\
 | 二分图匹配（匈牙利算法 BFS 实现）
 | INIT: g[][]邻接矩阵;
 | CALL: res = MaxMatch();Nx, Ny初始化！！！
 | 优点：适用于稀疏二分图，边较少，增广路较短。
 | 匈牙利算法的理论复杂度是O(VE)
\*==================================================*/
const int MAXN = 1000;
int g[MAXN][MAXN], Mx[MAXN], My[MAXN], Nx, Ny;
int chk[MAXN], Q[MAXN], prev[MAXN];
int MaxMatch(void) {
    int res = 0;
    int qs, qe;
    memset(Mx, -1, sizeof(Mx));
    memset(My, -1, sizeof(My));
    memset(chk, -1, sizeof(chk));
    for (int i = 0; i < Nx; i++){
        if (Mx[i] == -1){
            qs = qe = 0;
            Q[qe++] = i;
            prev[i] = -1;

            bool flag = 0;
            while (qs < qe && !flag){
                int u = Q[qs];
                for (int v = 0; v < Ny && !flag; v++)
                    if (g[u][v] && chk[v] != i) {
                        chk[v] = i; Q[qe++] = My[v];
                        if (My[v] >= 0) prev[My[v]] = u;
                        else {
                            flag = 1;
                            int d = u, e = v;
                            while (d != -1) {
                                int t = Mx[d];
                                Mx[d] = e;    My[e] = d;
                                d = prev[d]; e = t;
                            }
                        }
                    }
                qs++;
            }
            if (Mx[i] != -1) res++;
        }
    }
    return res;
}
/*==================================================*\
 | 二分图匹配（Hopcroft－Carp 的算法）
 | INIT: g[][]邻接矩阵;
 | CALL: res = MaxMatch();   Nx, Ny要初始化！！！
 | 时间复杂度为O（V^0.5 E）
\*==================================================*/
const int MAXN = 3001;
const int INF = 1 << 28;
int g[MAXN][MAXN], Mx[MAXN], My[MAXN], Nx, Ny;
int dx[MAXN], dy[MAXN], dis;
bool vst[MAXN];
bool searchP(void){
    queue<int> Q;
    dis = INF;
    memset(dx, -1, sizeof(dx));
    memset(dy, -1, sizeof(dy));
    for (int i = 0; i < Nx; i++)
        if (Mx[i] == -1){
            Q.push(i); dx[i] = 0;
        }
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        if (dx[u] > dis)  break;
        for (int v = 0; v < Ny; v++)
            if (g[u][v] && dy[v] == -1) {
                dy[v] = dx[u]+1;
                if (My[v] == -1)  dis = dy[v];
                else{
                    dx[My[v]] = dy[v]+1;
                    Q.push(My[v]);
                }
            }
    }
    return dis != INF;
}
bool DFS(int u){
    for (int v = 0; v < Ny; v++)
        if (!vst[v] && g[u][v] && dy[v] == dx[u]+1) {
            vst[v] = 1;
            if (My[v] != -1 && dy[v] == dis) continue;
            if (My[v] == -1 || DFS(My[v])) {
                My[v] = u; Mx[u] = v;
                return 1;
            }
        }
    return 0;
}
int MaxMatch(void){
    int res = 0;
    memset(Mx, -1, sizeof(Mx));
    memset(My, -1, sizeof(My));
    while (searchP()) {
        memset(vst, 0, sizeof(vst));
        for (int i = 0; i < Nx; i++)
            if (Mx[i] == -1 && DFS(i)) res++;
    }
    return res;
}
/*==================================================*\
 | 二分图最佳匹配（kuhn munkras 算法 O(m*m*n)）
 | 邻接距阵形式，复杂度O(m*m*n) 返回最佳匹配值，传入二分图大小m,n
 | 邻接距阵mat，表示权，match1,match2返回一个最佳匹配，未匹配顶点
 | match值为-1，一定注意m<=n，否则循环无法终止，最小权匹配可将权值
 | 取相反数
 | 初始化：for( i=0 ; i < MAXN ; ++i )
              for( j=0 ; j < MAXN ; ++j ) mat[i][j] = -inf;
 | 对于存在的边：mat[i][j] = val ; // 注意，不能有负值
\*==================================================*/
#include <string.h>
```

```
#define MAXN 310
#define inf 1000000000
#define _clr(x) memset(x,-1,sizeof(int)*MAXN)

int kuhn_munkras(int m,int n,int mat[][MAXN],int* match1,int* match2){
     int s[MAXN],t[MAXN],l1[MAXN],l2[MAXN],p,q,ret=0,i,j,k;
     for (i=0;i<m;i++){
          for (l1[i]=-inf,j=0;j<n;j++)
               l1[i]=mat[i][j]>l1[i]?mat[i][j]:l1[i];
          if( l1[i] == -inf ) return -1;// 无法匹配!
     }
     for (i=0;i<n;l2[i++]=0);
     for (_clr(match1),_clr(match2),i=0;i<m;i++){
          for (_clr(t),s[p=q=0]=i;p<=q&&match1[i]<0;p++)
               for (k=s[p],j=0;j<n&&match1[i]<0;j++)
                    if (l1[k]+l2[j]==mat[k][j]&&t[j]<0){
                         s[++q]=match2[j],t[j]=k;
                         if (s[q]<0)
                              for (p=j;p>=0;j=p)
     match2[j]=k=t[j],p=match1[k],match1[k]=j;
                    }
          if (match1[i]<0){
               for (i--,p=inf,k=0;k<=q;k++)
                    for (j=0;j<n;j++)
                         if (t[j]<0&&l1[s[k]]+l2[j]-mat[s[k]][j]<p)
                              p=l1[s[k]]+l2[j]-mat[s[k]][j];
               for (j=0;j<n;l2[j]+=t[j]<0?0:p,j++);
               for (k=0;k<=q;l1[s[k++]]-=p);
          }
     }
     for (i=0;i<m;i++)
     {// if处理无匹配的情况!!
          if( match[i] < 0 ) return -1;
          if( mat[i][match[i]] <= -inf ) return -1;
          ret+=mat[i][match1[i]];
     }
     return ret;
}
/*==================================================*\
 | 无向图最小割 O(N^3)
 | INIT: 初始化邻接矩阵g[][];
 | CALL: res = mincut(n);
 | 注: Stoer-Wagner Minimum Cut;
 | 找边的最小集合，若其被删去则图变得不连通（我们把这种形式称为最小
 | 割问题）
\*==================================================*/
#define typec int                  // type of res
const typec inf = 0x3f3f3f3f;      // max of res
const typec maxw = 1000;           // maximum edge weight
typec g[V][V], w[V];
int a[V], v[V], na[V];
typec mincut(int n){
    int i, j, pv, zj;
    typec best = maxw * n * n;
    for (i = 0; i < n; i++) v[i] = i; // vertex: 0 ~ n-1
    while (n > 1) {
        for (a[v[0]] = 1, i = 1; i < n; i++) {
            a[v[i]] = 0; na[i - 1] = i;
            w[i] = g[v[0]][v[i]];
        }
        for (pv = v[0], i = 1; i < n; i++ ) {
            for (zj = -1, j = 1; j < n; j++ )
                if (!a[v[j]] && (zj < 0 || w[j] > w[zj]))
                    zj = j;
                a[v[zj]] = 1;
                if (i == n - 1) {
                    if (best > w[zj]) best = w[zj];
                    for (i = 0; i < n; i++)
                        g[v[i]][pv] = g[pv][v[i]] +=
                               g[v[zj]][v[i]];
                    v[zj] = v[--n];
                    break;
                }
                pv = v[zj];
                for (j = 1; j < n; j++) if(!a[v[j]])
                    w[j] += g[v[zj]][v[j]];
        }
    }
    return best;
}
/*==================================================*\
 | 有上下界的最小(最大)流
 | INIT: up[][]为容量上界; low[][]为容量下界;
 | CALL: mf = limitflow(n,src,sink); flow[][]为流量分配;
 | 另附: 循环流问题
 | 描述: 无源无汇的网络N, 设N是具有基础有向图D=(V, A)的网络.
 |       l和c分别为容量下界和容量上界. 如果定义在A上的函数
 |       f满足: f(v, V) = f(V, v). V中任意顶点v,
 |       l(a)<=f(a)<=c(a), 则称f为网络N的循环流.
 | 解法: 添加一个源s和汇t, 对于每个下限容量l不为0的边(u, v),
 |       将其下限去掉, 上限改为c-l, 增加两条边(u, t), (s, v),
 |       容量均为l. 原网络存在循环流等价于新网络最大流是满流.
\*==================================================*/
int up[N][N], low[N][N], flow[N][N];
int pv[N], que[N], d[N];
void maxflow(int n, int src, int sink)
{   // BFS增广, O(E * maxflow)
    int p, q, t, i, j;
    do{
        for (i = 0; i < n; pv[i++] = 0) ;
        pv[t = src] = src + 1; d[t] = inf;
        for (p=q=0; p<=q && !pv[sink]; t=que[p++])
            for (i=0; i<n; i++) {
                if(!pv[i]&&up[t][i]&&(j=up[t][i]-flow[t][i])>0)
                    pv[que[q++]=i]=+t+1, d[i]=d[t]<j?d[t]:j;
                else if (!pv[i]&&up[i][t]&&(j=flow[i][t])>0)
                    pv[que[q++]=i]=-t-1, d[i]=d[t]<j?d[t]:j;
            }
        for (i=sink; pv[i] && i!=src; ) {
            if(pv[i]>0)flow[pv[i]-1][i]+=d[sink],i=pv[i]-1;
            else flow[i][-pv[i]-1]-=d[sink], i=-pv[i]-1;
        }
    } while (pv[sink]);
}
int limitflow(int n, int src, int sink)
{
    int i, j, sk, ks;
    if (src == sink) return inf;
     up[n][n+1] = up[n+1][n] = up[n][n] = up[n+1][n+1] = 0;
    for (i = 0; i < n; i++) {
        up[n][i] = up[i][n] = up[n+1][i] = up[i][n+1] = 0;
        for (j = 0; j < n; j++) {
            up[i][j] -= low[i][j];
            up[n][i] += low[j][i];
            up[i][n+1] += low[i][j];
        }
    }
    sk = up[src][sink]; ks = up[sink][src];
    up[src][sink] = up[sink][src] = inf;
    maxflow(n+2, n, n+1);
    for (i = 0; i < n; i++)
        if (flow[n][i] < up[n][i]) return -1;
    flow[src][sink] = flow[sink][src] = 0;
    up[src][sink] = sk; up[sink][src] = ks;
    // ! min: src <- sink;   max: src -> sink;
    maxflow(n, sink, src);
    for (i = 0; i < n; i++) for (j = 0; j < n; j++) {
        up[i][j] += low[i][j]; flow[i][j] += low[i][j];
    }
    for (j = i = 0; i < n; j += flow[src][i++]) ;
    return j;
}
/*==================================================*\
 | Dinic 最大流 O(V^2 * E)
 | INIT: ne=2; head[]置为0; addedge()加入所有弧;
 | CALL: flow(n, s, t);
\*==================================================*/
#define typec int                    // type of cost
const typec inf = 0x3f3f3f3f;        // max of cost
struct edge { int x, y, nxt; typec c; } bf[E];
int ne, head[N], cur[N], ps[N], dep[N];
void addedge(int x, int y, typec c)
{    // add an arc(x -> y, c); vertex: 0 ~ n-1;
     bf[ne].x = x; bf[ne].y = y; bf[ne].c = c;
```

```
    bf[ne].nxt = head[x]; head[x] = ne++;
    bf[ne].x = y; bf[ne].y = x; bf[ne].c = 0;
    bf[ne].nxt = head[y]; head[y] = ne++;
}
typec flow(int n, int s, int t)
{
    typec tr, res = 0;
    int i, j, k, f, r, top;
    while (1) {
        memset(dep, -1, n * sizeof(int));
        for (f = dep[ps[0] = s] = 0, r = 1; f != r; )
            for (i = ps[f++], j = head[i]; j; j = bf[j].nxt)
            {
                if (bf[j].c && -1 == dep[k = bf[j].y]){
                    dep[k] = dep[i] + 1; ps[r++] = k;
                    if (k == t) { f = r; break; }
                }
            }
        if (-1 == dep[t]) break;

        memcpy(cur, head, n * sizeof(int));
        for (i = s, top = 0; ; ) {
            if (i == t) {
                for (k = 0, tr = inf; k < top; ++k)
                    if (bf[ps[k]].c < tr)
                        tr = bf[ps[f = k]].c;
                for (k = 0; k < top; ++k)
                    bf[ps[k]].c -= tr, bf[ps[k]^1].c += tr;
                res += tr;  i = bf[ps[top = f]].x;
            }
            for (j=cur[i]; cur[i]; j = cur[i] =bf[cur[i]].nxt)
                if (bf[j].c && dep[i]+1 == dep[bf[j].y]) break;
            if (cur[i]) {
                ps[top++] = cur[i];
                i = bf[cur[i]].y;
            }
            else {
                if (0 == top) break;
                dep[i] = -1; i = bf[ps[--top]].x;
            }
        }
    }
    return res;
}
/*==================================================*\
 | HLPP 最大流 O(V^3)
 | INIT: network g; g.build(nv, ne);
 | CALL: res = g.maxflow(s, t);
 | 注意: 不要加入指向源点的边, 可能死循环.
\*==================================================*/
#define typef int                  // type of flow
const typef inf = 0x3f3f3f3f;      // max of flow
typef minf(typef a, typef b) { return a < b ? a : b; }

struct edge {
    int u, v; typef cuv, cvu, flow;
    edge (int x=0, int y=0, typef cu=0,
          typef cv=0, typef f=0)
        : u(x), v(y), cuv(cu), cvu(cv), flow(f) {}
    int other(int p) { return p == u ? v : u; }
    typef cap(int p) {
        return p == u ? cuv-flow : cvu+flow;
    }
    void addflow(int p, typef f) { flow += (p == u ? f : -f); }
};

struct vlist {
    int lv, next[N], idx[2 * N], v;
    void clear(int cv) {
        v = cv; lv = -1;
        memset(idx, -1, sizeof(idx));
    }
    void insert(int n, int h) {
        next[n] = idx[h]; idx[h] = n;
        if (lv < h) lv = h;
    }
    int remove() {
        int r = idx[lv];  idx[lv] = next[idx[lv]];
        while (lv >= 0 && idx[lv] == -1) lv--;
        return r;
    }
    bool empty() { return lv < 0; }
};

struct network {
    vector<edge> eg;
    vector<edge*> net[N];
    vlist list;
    typef e[N];
    int v, s, t, h[N], hn[2 * N], cur[N];
    void push(int);
    void relabel(int);
    void build(int, int);
    typef maxflow(int, int);
};
void network::push(int u) {
    edge* te = net[u][cur[u]];
    typef ex = minf(te->cap(u), e[u]);
    int p = te->other(u);
    if (e[p] == 0 && p != t) list.insert(p, h[p]);
    te->addflow(u, ex); e[u] -= ex; e[p] += ex;
}
void network::relabel(int u) {
    int i, p, mh = 2 * v, oh = h[u];
    for (i = net[u].size()-1; i >= 0; i--) {
        p = net[u][i]->other(u);
        if (net[u][i]->cap(u) != 0 && mh > h[p] + 1)
            mh = h[p] + 1;
    }
    hn[h[u]]--; hn[mh]++; h[u] = mh;
    cur[u] = net[u].size()-1;

    if (hn[oh] != 0 || oh >= v + 1) return;
    for (i = 0; i < v; i++)
        if (h[i] > oh && h[i] <= v && i != s) {
            hn[h[i]]--; hn[v+1]++; h[i] = v + 1;
        }
}
typef network::maxflow(int ss, int tt) {
    s = ss; t = tt;
    int i, p, u;  typef ec;

    for (i = 0; i < v; i++) net[i].clear();
    for (i = eg.size()-1; i >= 0; i--) {
        net[eg[i].u].push_back(&eg[i]);
        net[eg[i].v].push_back(&eg[i]);
    }

    memset(h, 0, sizeof(h)); memset(hn, 0, sizeof(hn));
    memset(e, 0, sizeof(e)); e[s] = inf;
    for (i = 0; i < v; i++) h[i] = v;
    queue<int> q; q.push(t); h[t] = 0;
    while (!q.empty()) {
        p = q.front(); q.pop();
        for (i = net[p].size()-1; i >= 0; i--) {
            u = net[p][i]->other(p);
            ec = net[p][i]->cap(u);
            if (ec != 0 && h[u] == v && u != s) {
                h[u] = h[p] + 1; q.push(u);
            }
        }
    }
    for (i = 0; i < v; i++) hn[h[i]]++;

    for (i = 0; i < v; i++) cur[i] = net[i].size()-1;
    list.clear(v);
    for (; cur[s] >= 0; cur[s]--) push(s);

    while (!list.empty()) {
        for (u = list.remove(); e[u] > 0; ) {
            if (cur[u] < 0) relabel(u);
            else if (net[u][cur[u]]->cap(u) > 0 &&
                h[u] == h[net[u][cur[u]]->other(u)]+1)
push(u);
            else cur[u]--;
        }
    }
    return e[t];
}
void network::build(int n, int m) {
```

```
    v = n; eg.clear();
    int a, b, i;  typef l;
    for (i = 0; i < m; i++) {
        cin >> a >> b >> l;
        eg.push_back(edge(a, b, l, 0)); // vertex: 0 ~ n-1
    }
}
/*=====================================================*\
 | 最小费用流 O(V * E * f)
 | INIT: network g; g.build(v, e);
 | CALL: g.mincost(s, t); flow=g.flow; cost=g.cost;
 | 注意: SPFA增广, 实际复杂度远远小于O(V * E);
\*=====================================================*/
#define typef int                  // type of flow
#define typec int                  // type of dis
const typef inff = 0x3f3f3f3f;      // max of flow
const typec infc = 0x3f3f3f3f;      // max of dis
struct network
{
    int nv, ne, pnt[E], nxt[E];
    int vis[N], que[N], head[N], pv[N], pe[N];
    typef flow, cap[E]; typec cost, dis[E], d[N];
    void addedge(int u, int v, typef c, typec w) {
        pnt[ne] = v; cap[ne] = c;
        dis[ne] = +w; nxt[ne] = head[u]; head[u] = (ne++);
        pnt[ne] = u; cap[ne] = 0;
        dis[ne] = -w; nxt[ne] = head[v]; head[v] = (ne++);
    }
    int mincost(int src, int sink) {
        int i, k, f, r; typef mxf;
        for (flow = 0, cost = 0; ; ) {
            memset(pv, -1, sizeof(pv));
            memset(vis, 0, sizeof(vis));
            for (i = 0; i < nv; ++i) d[i] = infc;
            d[src] = 0; pv[src] = src; vis[src] = 1;

            for (f = 0, r = 1, que[0] = src; r != f; ) {
                i = que[f++]; vis[i] = 0;
                if (N == f) f = 0;
                for (k = head[i]; k != -1; k = nxt[k])
                    if(cap[k] && dis[k]+d[i] < d[pnt[k]])
                    {
                        d[pnt[k]] = dis[k] + d[i];
                        if (0 == vis[pnt[k]]) {
                            vis[pnt[k]] = 1;
                            que[r++] = pnt[k];
                            if (N == r) r = 0;
                        }
                        pv[pnt[k]]=i; pe[pnt[k]]=k;
                    }
            }
            if (-1 == pv[sink]) break;

            for (k = sink, mxf = inff; k != src; k = pv[k])
                if (cap[pe[k]] < mxf) mxf = cap[pe[k]];
            flow += mxf; cost += d[sink] * mxf;

            for (k = sink; k != src; k = pv[k]) {
                cap[pe[k]] -= mxf; cap[pe[k] ^ 1] += mxf;
            }
        }
        return cost;
    }
    void build(int v, int e) {
        nv = v; ne = 0;
        memset(head, -1, sizeof(head));
        int x, y; typef f; typec w;
        for (int i = 0; i < e; ++i) {
            cin >> x >> y >> f >> w; // vertex: 0 ~ n-1
            addedge(x, y, f, w);// add arc (u->v, f, w)
        }
    }
} g;
/*=====================================================*\
 | 最小费用流 O(V^2 * f)
 | INIT: network g; g.build(nv, ne);
 | CALL: g.mincost(s, t); flow=g.flow; cost=g.cost;
 | 注意: 网络中弧的cost需为非负. 若存在负权, 进行如下转化:
 |  首先如果原图有负环, 则不存在最小费用流. 那么可以用Johnson
 | 重标号技术把所有边变成正权, 以后每次增广后进行维护, 算法如
 | 下:
 |  1. 用bellman-ford求s到各点的距离phi[];
 |  2. 以后每求一次最短路, 设s到各点的最短距离为dis[]:
 |     for i=1 to v do
 |       phi[v] += dis[v];
 | 下面的代码已经做了第二步, 如果原图有负权, 添加第一步即可.
\*=====================================================*/
#define typef int                         // type of flow
#define typec int                         // type of cost
const typef inff = 0x3f3f3f3f;             // max of flow
const typec infc = 0x3f3f3f3f;             // max of cost

struct edge {
    int u, v; typef cuv, cvu, flow; typec cost;
    edge (int x, int y, typef cu, typef cv, typec cc)
        :u(x), v(y), cuv(cu), cvu(cv), flow(0), cost(cc){}
    int other(int p) { return p == u ? v : u; }
    typef cap(int p) { return p == u ? cuv-flow : cvu+flow; }
    typec ecost(int p) {
        if (flow == 0) return cost;
        else if(flow > 0) return p == u ? cost : -cost;
        else return p == u ? -cost : cost;
    }
    void addFlow(int p, typef f) { flow += (p == u ? f : -f); }
};

struct network {
    vector<edge> eg;
    vector<edge*> net[N];
    edge *prev[N];
    int v, s, t, pre[N], vis[N];
    typef flow; typec cost, dis[N], phi[N];
    bool dijkstra();
    void build(int nv, int ne);
    typec mincost(int, int);
};
bool network::dijkstra()
{   // 使用O(E * logV)的Dij可降低整体复杂度至 O(E * logV * f)
    int i, j, p, u; typec md, cw;
    for (i = 0; i < v; i++) dis[i] = infc;
    dis[s] = 0; prev[s] = 0; pre[s] = -1;
    memset(vis, 0, v * sizeof(int));
    for (i = 1; i < v; i++) {
        for (md = infc, j = 0; j < v; j++)
            if (!vis[j] && md > dis[j]) {
                md = dis[j]; u = j;
            }
        if (md == infc) break;
        for (vis[u] = 1, j = net[u].size()-1; j >= 0; j--)
        {
            edge *ce = net[u][j];
            if(ce->cap(u) > 0) {
                p = ce->other(u);
                cw = ce->ecost(u) + phi[u] - phi[p];
                // !! assert(cw >= 0);
                if(dis[p] > dis[u]+cw) {
                    dis[p] = dis[u] + cw;
                    prev[p] = ce;   pre[p] = u;
                }
            }
        }
    }
    return infc != dis[t];
}
typec network::mincost(int ss, int tt) {
    s = ss; t = tt;
    int i, c;  typef ex;

    flow = cost = 0;
    memset(phi, 0, sizeof(phi));
    // !! 若原图含有负消费的边, 在此处运行Bellmanford
    // 将phi[i](0<=i<=n-1)置为mindist(s, i).

    for (i = 0; i < v; i++) net[i].clear();
    for (i = eg.size()-1; i >= 0; i--) {
        net[eg[i].u].push_back(&eg[i]);
        net[eg[i].v].push_back(&eg[i]);
    }

    while(dijkstra()) {
```

```
        for (ex = inff, c = t; c != s; c = pre[c])
            if (ex > prev[c]->cap(pre[c]))
                ex = prev[c]->cap(pre[c]);
        for (c = t; c != s; c = pre[c])
            prev[c]->addFlow(pre[c], ex);
        flow += ex; cost += ex * (dis[t] + phi[t]);
        for (i = 0; i < v; i++) phi[i] += dis[i];
    }
    return cost;
}
void network::build(int nv, int ne) {
    eg.clear(); v = nv;
    int x, y; typef f; typec c;
    for (int i = 0; i < ne; ++i) {
        cin >> x >> y >> f >> c;
        eg.push_back(edge(x, y, f, 0, c));
    }
}
/*==================================================*\
 | 最佳边割集
\*==================================================*/
#define MAXN 100
#define inf 1000000000

int max_flow(int n,int mat[][MAXN],int source,int sink){
    int v[MAXN],c[MAXN],p[MAXN],ret=0,i,j;
    for (;;){
        for (i=0;i<n;i++)
            v[i]=c[i]=0;
        for (c[source]=inf;;){
            for (j=-1,i=0;i<n;i++)
                if (!v[i]&&c[i]&&(j==-1||c[i]>c[j]))
                    j=i;
            if (j<0) return ret;
            if (j==sink) break;
            for (v[j]=1,i=0;i<n;i++)
                if (mat[j][i]>c[i]&&c[j]>c[i])

    c[i]=mat[j][i]<c[j]?mat[j][i]:c[j],p[i]=j;
        }
        for (ret+=j=c[i=sink];i!=source;i=p[i])
            mat[p[i]][i]-=j, mat[i][p[i]]+=j;
    }
}

int best_edge_cut(int n,int mat[][MAXN],int source,int
sink,int set[][2],int& mincost){
    int m0[MAXN][MAXN],m[MAXN][MAXN],i,j,k,l,ret=0,last;
    if (source==sink)
        return -1;
    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
            m0[i][j]=mat[i][j];
    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
            m[i][j]=m0[i][j];
    mincost=last=max_flow(n,m,source,sink);
    for (k=0;k<n&&last;k++)
        for (l=0;l<n&&last;l++)
            if (m0[k][l]){
                for (i=0;i<n+n;i++)
                    for (j=0;j<n+n;j++)
                        m[i][j]=m0[i][j];
                m[k][l]=0;
                if
(max_flow(n,m,source,sink)==last-mat[k][l]){
                    set[ret][0]=k;
                    set[ret++][1]=l;
                    m0[k][l]=0;
                    last-=mat[k][l];
                }
            }
    return ret;
}
/*==================================================*\
 | 最佳点割集
\*==================================================*/
#define MAXN 100
#define inf 1000000000
int max_flow(int n,int mat[][MAXN],int source,int sink){
    int v[MAXN],c[MAXN],p[MAXN],ret=0,i,j;
    for (;;){
        for (i=0;i<n;i++)
            v[i]=c[i]=0;
        for (c[source]=inf;;){
            for (j=-1,i=0;i<n;i++)
                if (!v[i]&&c[i]&&(j==-1||c[i]>c[j]))
                    j=i;
            if (j<0) return ret;
            if (j==sink) break;
            for (v[j]=1,i=0;i<n;i++)
                if (mat[j][i]>c[i]&&c[j]>c[i])

    c[i]=mat[j][i]<c[j]?mat[j][i]:c[j],p[i]=j;
        }
        for (ret+=j=c[i=sink];i!=source;i=p[i])
            mat[p[i]][i]-=j,mat[i][p[i]]+=j;
    }
}

int best_vertex_cut(int n,int mat[][MAXN],int* cost,int
source,int sink,int* set,int& mincost){
    int m0[MAXN][MAXN],m[MAXN][MAXN],i,j,k,ret=0,last;
    if (source==sink||mat[source][sink])
        return -1;
    for (i=0;i<n+n;i++)
        for (j=0;j<n+n;j++)
            m0[i][j]=0;
    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
            if (mat[i][j])
                m0[i][n+j]=inf;
    for (i=0;i<n;i++)
        m0[n+i][i]=cost[i];
    for (i=0;i<n+n;i++)
        for (j=0;j<n+n;j++)
            m[i][j]=m0[i][j];
    mincost=last=max_flow(n+n,m,source,n+sink);
    for (k=0;k<n&&last;k++)
        if (k!=source&&k!=sink){
            for (i=0;i<n+n;i++)
                for (j=0;j<n+n;j++)
                    m[i][j]=m0[i][j];
            m[n+k][k]=0;
            if
(max_flow(n+n,m,source,n+sink)==last-cost[k]){
                set[ret++]=k;
                m0[n+k][k]=0;
                last-=cost[k];
            }
        }
    return ret;
}
/*==================================================*\
 | 最小边割集
\*==================================================*/
#define MAXN 100
#define inf 1000000000
int max_flow(int n,int mat[][MAXN],int source,int sink){
    int v[MAXN],c[MAXN],p[MAXN],ret=0,i,j;
    for (;;){
        for (i=0;i<n;i++)
            v[i]=c[i]=0;
        for (c[source]=inf;;){
            for (j=-1,i=0;i<n;i++)
                if (!v[i]&&c[i]&&(j==-1||c[i]>c[j]))
                    j=i;
            if (j<0) return ret;
            if (j==sink) break;
            for (v[j]=1,i=0;i<n;i++)
                if (mat[j][i]>c[i]&&c[j]>c[i])

    c[i]=mat[j][i]<c[j]?mat[j][i]:c[j],p[i]=j;
        }
        for (ret+=j=c[i=sink];i!=source;i=p[i])
            mat[p[i]][i]-=j,mat[i][p[i]]+=j;
    }
}

int min_edge_cut(int n,int mat[][MAXN],int source,int
```

```
sink,int set[][2]){
     int m0[MAXN][MAXN],m[MAXN][MAXN],i,j,k,l,ret=0,last;
     if (source==sink)
          return -1;
     for (i=0;i<n;i++)
          for (j=0;j<n;j++)
               m0[i][j]=(mat[i][j]!=0);
     for (i=0;i<n;i++)
          for (j=0;j<n;j++)
               m[i][j]=m0[i][j];
     last=max_flow(n,m,source,sink);
     for (k=0;k<n&&last;k++)
          for (l=0;l<n&&last;l++)
               if (m0[k][l]){
                    for (i=0;i<n+n;i++)
                         for (j=0;j<n+n;j++)
                              m[i][j]=m0[i][j];
                    m[k][l]=0;
                    if (max_flow(n,m,source,sink)<last){
                         set[ret][0]=k;
                         set[ret++][1]=l;
                         m0[k][l]=0;
                         last--;
                    }
               }
     return ret;
}
/*==================================================*\
 | 最小点割集（点连通度）
\*==================================================*/
#define MAXN 100
#define inf 1000000000

int max_flow(int n,int mat[][MAXN],int source,int sink){
     int v[MAXN],c[MAXN],p[MAXN],ret=0,i,j;
     for (;;){
          for (i=0;i<n;i++)
               v[i]=c[i]=0;
          for (c[source]=inf;;){
               for (j=-1,i=0;i<n;i++)
                    if (!v[i]&&c[i]&&(j==-1||c[i]>c[j]))
                         j=i;
               if (j<0) return ret;
               if (j==sink) break;
               for (v[j]=1,i=0;i<n;i++)
                    if (mat[j][i]>c[i]&&c[j]>c[i])

     c[i]=mat[j][i]<c[j]?mat[j][i]:c[j],p[i]=j;
          }
          for (ret+=j=c[i=sink];i!=source;i=p[i])
               mat[p[i]][i]-=j,mat[i][p[i]]+=j;
     }
}

int min_vertex_cut(int n,int mat[][MAXN],int source,int
sink, int* set){
     int m0[MAXN][MAXN],m[MAXN][MAXN],i,j,k,ret=0,last;
     if (source==sink||mat[source][sink])
          return -1;
     for (i=0;i<n+n;i++)
          for (j=0;j<n+n;j++)
               m0[i][j]=0;
     for (i=0;i<n;i++)
          for (j=0;j<n;j++)
               if (mat[i][j])
                    m0[i][n+j]=inf;
     for (i=0;i<n;i++)
          m0[n+i][i]=1;
     for (i=0;i<n+n;i++)
          for (j=0;j<n+n;j++)
               m[i][j]=m0[i][j];
     last=max_flow(n+n,m,source,n+sink);
     for (k=0;k<n&&last;k++)
          if (k!=source&&k!=sink){
               for (i=0;i<n+n;i++)
                    for (j=0;j<n+n;j++)
                         m[i][j]=m0[i][j];
               m[n+k][k]=0;
               if (max_flow(n+n,m,source,n+sink)<last){
                    set[ret++]=k;
                    m0[n+k][k]=0;
                    last--;
               }
          }
     return ret;
}
/*==================================================*\
 |最小路径覆盖 O(n^3)
```

路径覆盖：就是在图中找一些路经，使之覆盖了图中的所有顶点，且任何一个顶点有且只有一条路径与之关联。
最小路径覆盖：就是找出最少的路径条数，使之成为P的一个路径覆盖．路径覆盖与二分图匹配的关系：最小路径覆盖＝｜P｜－最大匹配数；
其中最大匹配数的求法是把P中的每个顶点pi分成两个顶点pi'与pi''，如果在p中存在一条pi到pj的边，那么在二分图P′中就有一条连接pi'与pj''的有向边（求二分图匹配时必须是单向边）；这里pi' 就是p中pi的出边，pj''就是p中pj 的一条入边；
有向图：最小路径覆盖＝｜P｜－最大匹配数；
无向图：最小路径覆盖＝｜P｜－最大匹配数/2；

```
\*==================================================*/
/*==================================================*\
 | 最小点集覆盖
 | 结论：一个二分图中的最大匹配数等于这个图中的最小点覆盖数。
\*==================================================*/
```

Structure 数据结构

```
/*==================================================*\
 | 求某天是星期几
\*==================================================*/
char *name[] = { "monday", "tuesday", "wednesday",
"thursday", "friday", "saturday", "sunday" };
int main(void)
{
    int d, m, y, a;
    printf("Day: "); scanf("%d",&d);
    printf("Month: "); scanf("%d",&m);
    printf("Year: ");  scanf("%d",&y);
    // 1月2月当作前一年的13,14月
    if (m == 1 || m == 2) { m += 12; y--; }
    // 判断是否在1752年9月3日之前
   if ((y < 1752) || (y == 1752 && m < 9) ||
        (y == 1752 && m == 9 && d < 3))
        a = (d + 2*m + 3*(m+1)/5 + y + y/4 +5) % 7;
    else
        a = (d + 2*m + 3*(m+1)/5 + y + y/4 - y/100 + y/400)%7;
    printf("it's a %s\n", name[a]);
    return 0;
}
/*==================================================*\
 | 左偏树 合并复杂度 O(log N)
 | INIT: init()读入数据并进行初始化;
 | CALL: merge() 合并两棵左偏树; ins() 插入一个新节点;
 |       top() 取得最小结点; pop() 取得并删除最小结点;
 |       del() 删除某结点; add() 增/减一个结点的键值;
 |       iroot() 获取结点i的根;
\*==================================================*/
#define typec int              // type of key val
const int na = -1;
struct node { typec key; int l, r, f, dist; } tr[N];
int iroot(int i){              // find i's root
    if (i == na) return i;
   while (tr[i].f != na) i = tr[i].f;
   return i;
}
int merge(int rx, int ry){     // two root: rx, ry
   if (rx == na) return ry;
   if (ry == na) return rx;
   if (tr[rx].key > tr[ry].key) swap(rx, ry);
   int r = merge(tr[rx].r, ry);
   tr[rx].r = r; tr[r].f = rx;
   if (tr[r].dist > tr[tr[rx].l].dist)
      swap(tr[rx].l, tr[rx].r);
   if (tr[rx].r == na) tr[rx].dist = 0;
   else tr[rx].dist = tr[tr[rx].r].dist + 1;
   return rx;                // return new root
}
int ins(int i, typec key, int root){ // add a new node(i, key)
   tr[i].key = key;
   tr[i].l = tr[i].r = tr[i].f = na;
   tr[i].dist = 0;
   return root = merge(root, i);  // return new root
}
int del(int i) {              // delete node i
   if (i == na) return i;
   int x, y, l, r;
   l = tr[i].l; r = tr[i].r; y = tr[i].f;
   tr[i].l = tr[i].r = tr[i].f = na;
   tr[x = merge(l, r)].f = y;
   if (y != na && tr[y].l == i) tr[y].l = x;
   if (y != na && tr[y].r == i) tr[y].r = x;
   for ( ; y != na; x = y, y = tr[y].f) {
      if (tr[tr[y].l].dist < tr[tr[y].r].dist)
         swap(tr[y].l, tr[y].r);
      if (tr[tr[y].r].dist + 1 == tr[y].dist) break;
      tr[y].dist = tr[tr[y].r].dist + 1;
   }
   if (x != na) return iroot(x); // return new root
   else return iroot(y);
}
node top(int root){
    return tr[root];
}
node pop(int &root){
    node out = tr[root];
    int l = tr[root].l, r = tr[root].r;
    tr[root].l = tr[root].r = tr[root].f = na;
    tr[l].f = tr[r].f = na;
    root = merge(l, r);
    return out;
}
int add(int i, typec val)  // tr[i].key += val
{
     if (i == na) return i;
     if (tr[i].l == na && tr[i].r == na && tr[i].f == na) {
         tr[i].key += val;
         return i;
     }
     typec key = tr[i].key + val;
     int rt = del(i);
     return ins(i, key, rt);
}
void init(int n){
     for (int i = 1; i <= n; i++) {
         scanf("%d", &tr[i].key); //%d: type of key
         tr[i].l = tr[i].r = tr[i].f = na;
         tr[i].dist = 0;
     }
}
/*==================================================*\
 | 树状数组
 | INIT: ar[]置为0;
 | CALL: add(i, v): 将i点的值加v;  sum(i): 求[1, i]的和;
\*==================================================*/
#define typev int                       // type of res
typev ar[N];                           // index: 1 ~ N
int lowb(int t) { return t & (-t) ; }
void add(int i, typev v) {
     for ( ; i < N; ar[i] += v, i += lowb(i));
}
typev sum(int i) {
     typev s = 0;
     for ( ; i > 0; s += ar[i], i -= lowb(i));
     return s;
}
/*==================================================*\
 | 二维树状数组
 | INIT: c[][]置为0; Row,Col要赋初值
\*==================================================*/
const int N = 10000;
int  c[N][N]; int  Row, Col;
inline int Lowbit(const int &x){// x > 0
     return x&(-x);
}
int Sum(int i, int j){
     int  tempj, sum = 0;
     while( i > 0 ){
         tempj = j;
         while( tempj > 0 ){
             sum += c[i][tempj];
             tempj -= Lowbit(tempj);
         }
         i -= Lowbit(i);
     }
     return sum;
}
void Update(int i, int j, int num){
     int  tempj;
     while( i <= Row ){
         tempj = j;
         while( tempj <= Col ){
             c[i][tempj] += num;
             tempj += Lowbit(tempj);
         }
         i += Lowbit(i);
     }
}
/*==================================================*\
```

```
 | Trie 树(k 叉)
 | INIT: init();
 | 注: tree[i][tk]>0时表示单词存在, 当然也可赋予它更多含义;
\*==================================================*/
const int tk = 26, tb = 'a';  // tk叉; 起始字母为tb;
int top, tree[N][tk + 1];     // N: 最大结点个数
void init(){
    top = 1;
    memset(tree[0], 0, sizeof(tree[0]));
}
int search(char *s){ // 失败返回0
    for (int rt = 0; rt = tree[rt][*s - tb]; )
        if (*(++s) == 0) return tree[rt][tk];
    return 0;
}
void insert(char *s, int rank = 1){
    int rt, nxt;
    for (rt = 0; *s; rt = nxt, ++s) {
        nxt = tree[rt][*s - tb];
        if (0 == nxt) {
            tree[rt][*s - tb] = nxt = top;
            memset(tree[top], 0, sizeof(tree[top]));
            top++;
        }
    }
tree[rt][tk] = rank;//1表示存在0表示不存在, 也可以赋予其其他含义
}
void delete(char *s){ // 只做标记, 假定s一定存在
    int rt = 0;
    for ( ; *s; ++s) rt = tree[rt][*s - tb];
    tree[rt][tk]=0;
}
int prefix(char *s){ // 最长前缀
    int rt = 0, lv;
    for (lv = 0; *s; ++s, ++lv) {
        rt = tree[rt][*s - tb];
        if (rt == 0) break;
    }
    return lv;
}
/*==================================================*\
 | Trie 树(左儿子右兄弟)
 | INIT: init();
\*==================================================*/
int top;
struct trie { char c; int l, r, rk; } tree[N];
void init(){
    top = 1;
    memset(tree, 0, sizeof(tree[0]));
}
int search(char *s) {            // 失败返回0
    int rt;
    for (rt = 0; *s; ++s) {
        for (rt = tree[rt].l; rt; rt = tree[rt].r)
            if (tree[rt].c == *s) break;
        if (rt == 0) return 0;
    }
    return tree[rt].rk;
}
void insert(char *s, int rk = 1){ //rk: 权或者标记
    int i, rt;
    for (rt = 0; *s; ++s, rt=i) {
        for (i = tree[rt].l; i; i = tree[i].r)
            if (tree[i].c == *s) break;
        if (i == 0) {
            tree[top].r = tree[rt].l;
            tree[top].l = 0;
            tree[top].c = *s;
            tree[top].rk = 0;
            tree[rt].l = top;
            i = top++;
        }
    }
    tree[rt].rk=rk;
}
void delete(char *s){   // 假定s已经存在, 只做标记
    int rt;
    for (rt = 0; *s; ++s) {
        for (rt = tree[rt].l; rt; rt = tree[rt].r)
            if (tree[rt].c == *s) break;
    }
    tree[rt].rk = 0;
}
int prefix(char *s){  // 最长前缀
    int rt = 0, lv;
    for (lv = 0; *s; ++s, ++lv) {
        for (rt = tree[rt].l; rt; rt = tree[rt].r)
            if (tree[rt].c == *s) break;
        if (rt == 0) break;
    }
    return lv;
}
/*==================================================*\
 | 后缀数组 O(N * log N)
 | INIT: n = strlen(s) + 1;
 | CALL: makesa(); lcp();
 | 注: height[i] = lcp(sa[i], sa[i-1]);
\*==================================================*/
char s[N];                              // N > 256
int n, sa[N], height[N], rank[N], tmp[N], top[N];
void makesa(){                          // O(N * log N)
    int i, j, len, na;
    na = (n < 256 ? 256 : n);
    memset(top, 0, na * sizeof(int));
    for (i = 0; i < n ; i++) top[ rank[i] = s[i] & 0xff ]++;
    for (i = 1; i < na; i++) top[i] += top[i - 1];
    for (i = 0; i < n ; i++) sa[ --top[ rank[i] ] ] = i;
    for (len = 1; len < n; len <<= 1) {
        for (i = 0; i < n; i++) {
            j = sa[i] - len; if (j < 0) j += n;
            tmp[ top[ rank[j] ]++ ] = j;
        }
        sa[ tmp[ top[0] = 0 ] ] = j = 0;
        for (i = 1; i < n; i++) {
            if (rank[ tmp[i] ] != rank[ tmp[i-1] ] ||
                rank[ tmp[i]+len ]!=rank[ tmp[i-1]+len ])
                top[++j] = i;
            sa[ tmp[i] ] = j;
        }
        memcpy(rank, sa , n * sizeof(int));
        memcpy(sa  , tmp, n * sizeof(int));
        if (j >= n - 1) break;
    }
}
void lcp(){                             // O(4 * N)
    int i, j, k;
    for (j = rank[height[i=k=0]=0]; i < n - 1; i++, k++)
        while (k >= 0 && s[i] != s[ sa[j-1] + k ])
            height[j] = (k--), j = rank[ sa[j] + 1 ];
}
/*==================================================*\
 | 后缀数组 O(N)
 | INIT: n = strlen(s) + 1;
 | CALL: makesa()求sa[];
\*==================================================*/
char s[N];
int n, sa[4*N], rank[N], height[N];
int buf[4*N], ct[N], sx[N], sax[N];

inline bool leq(int a, int b, int x, int y)
{ return (a < x || a == x && b <= y); }
inline bool leq(int a, int b, int c, int x, int y, int z)
{ return (a < x || a == x && leq(b, c, y, z)); }
inline int geti(int t, int nx, int sa[])
{ return (sa[t]<nx ? sa[t]*3+1 : (sa[t]-nx)*3+2); }

static void radix(int a[], int b[], int s[], int n, int k)
{  // sort a[0..n-1] to b[0..n-1] with keys in 0..k from s
    int i, t, sum;
    memset(ct, 0, (k + 1) * sizeof(int));
    for (i = 0; i < n;  ++i) ct[s[a[i]]]++;
    for (i = 0, sum = 0; i <= k;  ++i) {
        t = ct[i]; ct[i] = sum; sum += t;
    }
    for (i = 0; i < n; i++) b[ct[s[a[i]]]++] = a[i];
}
void suffix(int s[], int sa[], int n, int k)
{   // !!! require s[n] = s[n+1] = s[n+2] = 0, n >= 2.
    int i, j, e, p, t;
    int name = 0, cx = -1, cy = -1, cz = -1;
```

```
    int nx = (n+2)/3, ny = (n+1)/3, nz = n/3, nxz = nx+nz;
    int *syz = s + n + 3, *sayz = sa + n + 3;

    for (i=0, j=0;  i < n + (nx - ny);  i++)
        if (i%3 != 0) syz[j++] = i;
    radix(syz , sayz, s+2, nxz, k);
    radix(sayz, syz , s+1, nxz, k);
    radix(syz , sayz, s  , nxz, k);
    for (i = 0;  i < nxz;  i++) {
        if (s[ sayz[i] ] != cx || s[ sayz[i] + 1 ] != cy ||
            s[ sayz[i] + 2 ] != cz) {
            name++;  cx = s[ sayz[i] ];
            cy = s[ sayz[i] + 1 ];  cz = s[ sayz[i] + 2 ];
        }
        if (sayz[i] % 3 == 1) syz[ sayz[i] / 3 ] = name;
        else syz[ sayz[i]/3 + nx ] = name;
    }
    if (name < nxz) {
        suffix(syz, sayz, nxz, name);
        for (i = 0; i < nxz;  i++) syz[sayz[i]] = i + 1;
    } else {
        for (i = 0; i < nxz;  i++) sayz[syz[i] - 1] = i;
    }
    for (i = j = 0; i < nxz;  i++)
        if (sayz[i] < nx) sx[j++] = 3 * sayz[i];
    radix(sx, sax, s, nx, k);
    for (p=0, t=nx-ny, e=0;  e < n;  e++) {
        i = geti(t, nx, sayz); j = sax[p];
        if ( sayz[t] < nx ?
             leq(s[i], syz[sayz[t]+nx], s[j], syz[j/3]) :
             leq(s[i], s[i+1], syz[sayz[t]-nx+1],
                 s[j], s[j+1], syz[j/3+nx]) ) {
                sa[e] = i;
                if (++t == nxz) {
                    for (e++; p < nx; p++, e++)
                        sa[e] = sax[p];
                }
        }
        else {
            sa[e] = j;
            if (++p == nx) for (++e; t < nxz; ++t, ++e)
                sa[e] = geti(t, nx, sayz);
        }
    }
}
void makesa(){
    memset(buf, 0, 4 * n * sizeof(int));
    memset(sa, 0, 4 * n * sizeof(int));
    for (int i=0; i<n; ++i) buf[i] = s[i] & 0xff;
    suffix(buf, sa, n, 255);
}
/*==================================================*\
 | RMQ 离线算法 O(N*logN)+O(1)
 | INIT: val[]置为待查询数组; initrmq(n);
\*==================================================*/
int st[20][N], ln[N], val[N];
void initrmq(int n){
    int i, j, k, sk;
    ln[0] = ln[1] = 0;
    for (i = 0; i < n; i++) st[0][i] = val[i];
    for (i = 1, k = 2; k < n; i++, k <<= 1) {
        for (j = 0, sk = (k >> 1); j < n; ++j, ++sk) {
            st[i][j] = st[i-1][j];
            if (sk < n && st[i][j] > st[i-1][sk])
                st[i][j] = st[i-1][sk];
        }
        for (j=(k>>1)+1; j <= k; ++j) ln[j] = ln[k>>1] + 1;
    }
    for (j=(k>>1)+1; j <= k; ++j) ln[j] = ln[k>>1] + 1;
}
int query(int x, int y) // min of { val[x] ... val[y] }
{
    int bl = ln[y - x + 1];
    return min(st[bl][x], st[bl][y-(1<<bl)+1]);
}
/*==================================================*\
| RMQ(Range Minimum/Maximum Query)-st 算法(O(nlogn + Q))
| ReadIn() 初始化数组a[0...n-1];
| InitRMQ()利用st算法( O(nlogn) )进行预处理;
| Query()根据输入的下标查询最值(O(Q))
| Hint: 下标范围:0...n-1,如果为1...n须稍做修改; 此处实现的的是求
| 最大值, 如果求最小值需要把max->min
| Call: ReadIn(n); InitRMQ(n); Query(Q);
\*==================================================*/
const int N = 200001;
int  a[N], d[20];
int  st[N][20];
int main(void){
    int  n, Q;
    while( scanf("%d%d", &n, &Q) != EOF )   {
        ReadIn(n); InitRMQ(n); Query(Q);
    }
    return 0;
}
void ReadIn(const int &n){
    int  i;
    for( i=0; i < n; ++i ) scanf("%d", &a[i]);
}
inline int max(const int &arg1, const int &arg2){
    return arg1 > arg2 ? arg1 : arg2;
}
void InitRMQ(const int &n){
    int  i, j;
    for( d[0]=1, i=1; i < 21; ++i ) d[i] = 2*d[i-1];
    for( i=0; i < n; ++i ) st[i][0] = a[i];
    int  k = int( log(double(n))/log(2) ) + 1;
    for( j=1; j < k; ++j )
        for( i=0; i < n; ++i ){
            if( i+d[j-1]-1 < n ){
                st[i][j] = max(st[i][j-1],
                               st[i+d[j-1]][j-1]);
            }
            else break; // st[i][j] = st[i][j-1];
        }
}
void Query(const int &Q){
    int  i;
    for( i=0; i < Q; ++i ){
        int  x, y, k; // x, y均为下标:0...n-1
        scanf("%d%d", &x, &y);
       k = int( log(double(y-x+1))/log(2.0) );
        printf("%d\n", max(st[x][k], st[y-d[k]+1][k]));
    }
}
/*==================================================*\
 | RMQ 离线算法 O(N*logN)+O(1)求解 LCA
 | INIT: val[]置为待查询数组; initrmq(n);
\*==================================================*/
const int N = 10001; // 1<<20;
int pnt[N], next[N], head[N]; // 邻接表
int e;                        // 边数
bool visited[N];              // 初始为0, 从根遍历
int  id;
int  dep[2*N+1], E[2*N+1], R[N]; // dep:dfs遍历节点深度, E:dfs
序列, R:第一次被遍历的下标
void DFS(int u, int d);
int  d[20], st[2*N+1][20];
void Answer(void){
    int  i, Q;
    scanf("%d", &Q);
    for( i=0; i < Q; ++i ){
        int  x, y;
        scanf("%d%d", &x, &y); // 查询x,y的LCA
        x = R[x]; y = R[y];
        if( x > y ){
            int  tmp = x; x = y; y = tmp;
        }
        printf("%d\n", E[ Query(x, y) ]);
    }
}
void DFS(int u, int d){
    visited[u] = 1;
    R[u] = id; E[id] = u; dep[id++] = d;
    for( int i=head[u]; i != -1; i=next[i] )
        if( visited[ pnt[i] ] == 0 ){
            DFS(pnt[i], d+1);
            E[id] = u; dep[id++] = d;
        }
}
void InitRMQ(const int &id){
```

```
    int i, j;
    for( d[0]=1, i=1; i < 20; ++i ) d[i] = 2*d[i-1];
    for( i=0; i < id; ++i ) st[i][0] = i;
    int k = int( log(double(n))/log(2.0) ) + 1;
    for( j=1; j < k; ++j )
        for( i=0; i < id; ++i ){
            if( i+d[j-1]-1 < id ){
                st[i][j] = dep[ st[i][j-1] ] >
dep[ st[i+d[j-1]][j-1] ] ? st[i+d[j-1]][j-1] : st[i][j-1];
            }
            else break; // st[i][j] = st[i][j-1];
        }
}
int Query(int x, int y){
    int k;   // x, y均为下标:0...n-1
    k = int( log(double(y-x+1))/log(2.0) );
    return dep[ st[x][k] ] > dep[ st[y-d[k]+1][k] ] ?
st[y-d[k]+1][k] : st[x][k];
}
/*==================================================*\
 | LCA 离线算法 O(E)+O(1)
 | INIT: id[]置为-1; g[]置为邻接矩阵;
 | CALL: for (i=0; i<n; ++i) if (-1==st[i]) dfs(i, n);
 |    LCA转化为RMQ的方法: 对树进行DFS遍历, 每当进入或回溯到
 | 某个结点i时, 将i的深度存入数组e[]最后一位. 同时记录结点i在
 | 数组中第一次出现的位置, 记做r[i]. 结点e[i]的深度记做d[i].
 | LCA(T,u,v), 等价于求E[RMQ(d,r[u],r[v])], (r[u]<r[v]).
\*==================================================*/
int id[N], lcs[N][N], g[N][N];
int get(int i){
    if (id[i] == i) return i;
    return id[i] = get(id[i]);
}
void unin(int i, int j){
    id[get(i)] = get(j);
}
void dfs(int rt, int n) {   // 使用邻接表可优化为 O(E)+O(1)
    int i;
    id[rt] = rt;
    for (i = 0; i < n; ++i) if (g[rt][i] && -1 == id[i]) {
        dfs(i, n); unin(i, rt);
    }
    for (i = 0; i < n; ++i) if (-1 != id[i])
        lcs[rt][i] = lcs[i][rt] = get(i);
}
/*==================================================*\
 | 带权值的并查集
 | INIT: makeset(n);
 | CALL: findset(x); unin(x, y);
\*==================================================*/
struct lset{
    int p[N], rank[N], sz;
    void link(int x, int y) {
        if (x == y) return;
        if (rank[x] > rank[y]) p[y] = x;
        else p[x] = y;
        if (rank[x] == rank[y]) rank[y]++;
    }
    void makeset(int n) {
        sz = n;
        for (int i=0;i<sz;i++) {
            p[i] = i; rank[i] = 0;
        }
    }
    int findset(int x) {
        if (x != p[x]) p[x] = findset(p[x]);
        return p[x];
    }
    void unin(int x, int y) {
        link(findset(x), findset(y));
    }
    void compress() {
      for (int i = 0; i < sz; i++) findset(i);
    }
};
/*==================================================*\
 | 快速排序
\*==================================================*/
void ksort(int l, int h, int a[]){
    if (h < l + 2) return;
    int e = h, p = l;
    while (l < h) {
        while (++l < e && a[l] <= a[p]);
        while (--h > p && a[h] >= a[p]);
        if (l < h) swap(a[l], a[h]);
    }
    swap(a[h], a[p]);
    ksort(p, h, a); ksort(l, e, a);
}
/*==================================================*\
 | 2 台机器工作调度
\*==================================================*/
```

2台机器, n件任务, 必须先在S1上做, 再在S2上做. 任务之间先做后做任意. 求最早的完工时间. 这是一个经典问题: 2台机器的情况下有多项式算法(Johnson算法), 3台或以上的机器是NP-hard的. Johnson算法:

(1) 把作业按工序加工时间分成两个子集,
  第一个集合中在S1上做的时间比在S2上少,
  其它的作业放到第二个集合.
  先完成第一个集合里面的作业, 再完成第二个集合里的作业.

(2) 对于第一个集合, 其中的作业顺序是按在S1上的时间的不减排列;
  对于第二个集合, 其中的作业顺序是按在S2上的时间的不增排列.

```
/*==================================================*\
 | 比较高效的大数
 | < , <= , + , - , * , / , %(修改/的最后一行可得)
\*==================================================*/
const int base = 10000;     // (base^2) fit into int
const int width = 4;        // width = log base
const int N  = 1000;        // n * width: 可表示的最大位数
struct bint{
    int ln, v[N];
    bint (int r = 0) { // r应该是字符串!
        for (ln = 0; r > 0; r /= base) v[ln++] = r % base;
    }
    bint& operator = (const bint& r) {
        memcpy(this, &r, (r.ln + 1) * sizeof(int));// !
        return *this;
    }
} ;
bool operator < (const bint& a, const bint& b){
    int i;
    if (a.ln != b.ln) return a.ln < b.ln;
    for (i = a.ln - 1; i >= 0 && a.v[i] == b.v[i]; i--);
    return i < 0 ? 0 : a.v[i] < b.v[i];
}
bool operator <= (const bint& a, const bint& b){
    return !(b < a);
}
bint operator + (const bint& a, const bint& b){
    bint res; int i, cy = 0;
    for (i = 0; i < a.ln || i < b.ln || cy > 0; i++) {
        if (i < a.ln) cy += a.v[i];
        if (i < b.ln) cy += b.v[i];
        res.v[i] = cy % base; cy /= base;
    }
    res.ln = i;
    return res;
}
bint operator - (const bint& a, const bint& b){
    bint res; int i, cy = 0;
    for (res.ln = a.ln, i = 0; i < res.ln; i++) {
        res.v[i] = a.v[i] - cy;
        if (i < b.ln) res.v[i] -= b.v[i];
        if (res.v[i] < 0) cy = 1, res.v[i] += base;
        else cy = 0;
    }
    while (res.ln > 0 && res.v[res.ln - 1] == 0) res.ln--;
    return res;
}
bint operator * (const bint& a, const bint& b){
    bint res; res.ln = 0;
    if (0 == b.ln) { res.v[0] = 0; return res; }
    int i, j, cy;
    for (i = 0; i < a.ln; i++) {
        for (j=cy=0; j < b.ln || cy > 0; j++, cy/= base) {
            if (j < b.ln) cy += a.v[i] * b.v[j];
            if (i + j < res.ln) cy += res.v[i + j];
            if (i + j >= res.ln) res.v[res.ln++] = cy % base;
            else res.v[i + j] = cy % base;
        }
    }
```

```
    return res;
}
bint operator / (const bint& a, const bint& b)
{   // ! b != 0
    bint tmp, mod, res;
    int i, lf, rg, mid;
    mod.v[0] = mod.ln = 0;
    for (i = a.ln - 1; i >= 0; i--) {
        mod = mod * base + a.v[i];
        for (lf = 0, rg = base -1; lf < rg; ) {
            mid = (lf + rg + 1) / 2;
            if (b * mid <= mod) lf = mid;
            else rg = mid - 1;
        }
        res.v[i] = lf;
        mod = mod - b * lf;
    }
    res.ln = a.ln;
    while (res.ln > 0 && res.v[res.ln - 1] == 0) res.ln--;
    return res;         // return mod 就是%运算
}
int digits(bint& a)                     // 返回位数
{
    if (a.ln == 0) return 0;
    int l = ( a.ln - 1 ) * 4;
    for (int t = a.v[a.ln - 1]; t; ++l, t/=10) ;
    return l;
}
bool read(bint& b, char buf[])  // 读取失败返回0
{
    if (1 != scanf("%s", buf)) return 0;
    int w, u, ln = strlen(buf);
    memset(&b, 0, sizeof(bint));
    if ('0' == buf[0] && 0 == buf[1]) return 1;
    for (w = 1, u = 0; ln; ) {
        u += (buf[--ln] - '0') * w;
        if (w * 10 == base) {
            b.v[b.ln++] = u; u = 0; w = 1;
        }
        else w *= 10;
    }
    if (w != 1) b.v[b.ln++] = u;
    return 1;
}
void write(const bint& v){
   int i;
    printf("%d", v.ln == 0 ? 0 : v.v[v.ln - 1]);
    for (i = v.ln - 2; i >= 0; i--)
        printf("%04d", v.v[i]);  // ! 4 == width
    printf("\n");
}
/*===================================================*\
 | 普通的大数运算
\*===================================================*/
const int MAXSIZE = 200;
void Add(char *str1, char *str2, char *str3);
void Minus(char *str1, char *str2, char *str3);
void Mul(char *str1, char *str2, char *str3);
void Div(char *str1, char *str2, char *str3);
int main(void){
    char  str1[MAXSIZE], str2[MAXSIZE], str3[MAXSIZE];
     while( scanf("%s %s", str1, str2) == 2 ){
        if( strcmp(str1, "0") ){
            memset(str3, '0', sizeof(str3)); // !!!!!
            Add(str1, str2, str3);
            printf("%s\n", str3);
            memset(str3, '0', sizeof(str3));
            Minus(str1, str2, str3);
            printf("%s\n", str3);
            memset(str3, '0', sizeof(str3));
            Mul(str1, str2, str3);
            printf("%s\n", str3);
            memset(str3, '0', sizeof(str3));
            Div(str1, str2, str3);
            printf("%s\n", str3);
        }
        else {
            if( strcmp(str2, "0") )
printf("%s\n-%s\n0\n0\n", str2, str2);
            else printf("0\n0\n0\n0\n");
```

```
        }
    }
    return 0;
}
void Add(char *str1, char *str2, char *str3)
{// str3 = str1 + str2;
    int  i, j, i1, i2, tmp, carry;
    int  len1 = strlen(str1), len2 = strlen(str2);
    char  ch;

    i1 = len1-1; i2 = len2-1;
    j = carry = 0;

    for( ; i1 >= 0 && i2 >= 0; ++j, --i1, --i2 ){
        tmp = str1[i1]-'0'+str2[i2]-'0'+carry;
        carry = tmp/10;
        str3[j] = tmp%10+'0';
    }
    while( i1 >= 0 ){
        tmp = str1[i1--]-'0'+carry;
        carry = tmp/10;
        str3[j++] = tmp%10+'0';
    }
    while( i2 >= 0 ){
        tmp = str2[i2--]-'0'+carry;
        carry = tmp/10;
        str3[j++] = tmp%10+'0';
    }
    if( carry ) str3[j++] = carry+'0';
    str3[j] = '\0';

    for( i=0, --j; i < j; ++i, --j ){
        ch = str3[i]; str3[i] = str3[j]; str3[j] = ch;
    }
}

void Minus(char *str1, char *str2, char *str3)
{// str3 = str1-str2 (str1 > str2)
    int  i, j, i1, i2, tmp, carry;
    int  len1 = strlen(str1), len2 = strlen(str2);
    char  ch;

    i1 = len1-1; i2 = len2-1;
    j = carry = 0;

    while( i2 >= 0 ){
        tmp = str1[i1]-str2[i2]-carry;
        if( tmp < 0 ) {
            str3[j] = tmp+10+'0'; carry = 1;
        }
        else {
            str3[j] = tmp+'0'; carry = 0;
        }
        --i1; --i2;  ++j;
    }
    while( i1 >= 0 ){
        tmp = str1[i1]-'0'-carry;
        if( tmp < 0 ) {
            str3[j] = tmp+10+'0'; carry = 1;
        }
        else{
            str3[j] = tmp+'0'; carry = 0;
        }
        --i1; ++j;
    }
    --j;
    while( str3[j] == '0' && j > 0 ) --j;
    str3[++j] = '\0';

    for( i=0, --j; i < j; ++i, --j ){
        ch = str3[i]; str3[i] = str3[j]; str3[j] = ch;
    }
}
void Mul(char *str1, char *str2, char *str3){
    int  i, j, i1, i2, tmp, carry, jj;
    int  len1 = strlen(str1), len2 = strlen(str2);
    char  ch;

    jj = carry = 0;
```

```
    for( i1=len1-1; i1 >= 0; --i1 ){
        j = jj;
        for( i2=len2-1; i2 >= 0; --i2, ++j ){
            tmp =
(str3[j]-'0')+(str1[i1]-'0')*(str2[i2]-'0')+carry;
            if( tmp > 9 ){
                carry = tmp/10; str3[j] = tmp%10+'0';
            }
            else {
                str3[j] = tmp+'0'; carry = 0;
            }
        }
        if( carry ) {
            str3[j] = carry+'0'; carry = 0; ++j;
        }
        ++jj;
    }
    --j;
    while( str3[j] == '0' && j > 0 ) --j;
    str3[++j] = '\0';

    for( i=0, --j; i < j; ++i, --j ){
        ch = str3[i]; str3[i] = str3[j]; str3[j] = ch;
    }
}

void Div(char *str1, char *str2, char *str3){
    int  i1, i2, i, j, jj, tag, carry, cf, c[MAXSIZE];
    int  len1 = strlen(str1), len2 = strlen(str2), lend;
    char  d[MAXSIZE];

    memset(c, 0, sizeof(c));
    memcpy(d, str1, len2);
    lend = len2; j = 0;

    for( i1=len2-1; i1 < len1; ++i1 ){
        if( lend < len2 ){
            d[lend] = str1[i1+1]; c[j] = 0;
            ++j; ++lend;
        }
        else if( lend == len2 ){
            jj = 1;
            for( i=0; i < lend; ++i ){
                if( d[i] > str2[i] ) break;
                else if( d[i] < str2[i] ){
                    jj = 0;  break;
                }
            }
            if( jj == 0 ){
                d[lend] = str1[i1+1]; c[j] = 0;
                ++j; ++lend;
                continue;
            }
        }
        if( jj==1 || lend > len2 ){
            cf = jj=0;
            while( d[jj] <= '0' && jj < lend ) ++jj;
            if( lend-jj > len2 ) cf = 1;
            else if( lend-jj < len2 ) cf = 0;
            else{
                i2 = 0; cf = 1;
                for( i=jj; i < lend; ++i ){
                    if( d[i] < str2[i2] ){
                        cf = 0;  break;
                    }
                    else if( d[i] > str2[i2] ){
                        break;
                    }
                    ++i2;
                }
            }//else
            while( cf ){
                i2 = len2-1; cf = 0;
                for( i=lend-1; i >= lend-len2; --i ){
                    d[i] = d[i]-str2[i2]+'0';
                    if( d[i] < '0' ){
                        d[i] = d[i]+10;   carry = 1;
                        --d[i-1];
                    }
                    else carry = 0;
                    --i2;
                }
                ++c[j];  jj=0;
                while( d[jj] <= '0' && jj < lend ) ++jj;
                if( lend-jj > len2 ) cf = 1;
                else if( lend-jj < len2 ) cf = 0;
                else{
                    i2 = 0; cf = 1;
                    for( i=jj; i < lend; ++i ){
                        if( d[i] < str2[i2] ){
                            cf = 0;  break;
                        }
                        else if( d[i] > str2[i2] ){
                            break;
                        }
                        ++i2;
                    }

                }//else
            }//while
            jj = 0;
            while( d[jj] <= '0' && jj < lend ) ++jj;
            for( i=0;i < lend-jj; ++i ) d[i] = d[i+jj];
            d[i] = str1[i1+1];    lend = i+1;
            ++j;
        }//else
    }//for
    i = tag = 0;
    while( c[i] == 0 ) ++i;
    for( ; i < j; ++i, ++tag ) str3[tag] = c[i]+'0';
    str3[tag] = '\0';
}
/*==================================================*\
 | 最长公共递增子序列 O(n^2)
 | f记录路径，DP记录长度, 用a对b扫描，逐步最优化。 ZOJ2432
\*==================================================*/
int f[N][N], dp[N];
int gcis(int a[], int la, int b[], int lb, int ans[])
{ // a[1…la], b[1…lb]
    int i, j, k, mx;
    memset(f, 0, sizeof(f));
    memset(dp, 0, sizeof(dp));
    for (i = 1; i <= la; i++) {
        memcpy(f[i], f[i-1], sizeof(f[0]));
        for (k = 0, j = 1; j <= lb; j++) {
            if (b[j-1] < a[i-1] && dp[j] > dp[k]) k = j;
            if (b[j-1] == a[i-1] && dp[k] + 1 > dp[j]) {
                dp[j] = dp[k] + 1,
                f[i][j] = i * (lb + 1) + k;
            }
        }
    }
    for (mx = 0, i = 1; i <= lb; i++)
        if (dp[i] > dp[mx]) mx = i;
    for(i=la*lb+la+mx, j=dp[mx]; j;
        i=f[i/(lb+1)][i%(lb+1)],j--)
        ans[j-1] = b[i % (lb + 1) - 1];
    return dp[mx];
}
/*==================================================*\
 | 0-1 分数规划
\*==================================================*/
    t1 * x1 + t2 * x2 + ... + tn * xn
r = ---------------------------------
    c1 * x1 + c2 * x2 + ... + cn * xn
给定t[1..n], c[1..n], 求x[1..n]使得sigma(xi)=k且r最大(小).
为了让r最大, 先设计子问题z(r) = (t1 * x1 + .. + tn * xn) - r *
(c1 * xn + .. + cn * xn);
假设r的最优值为R. 则有:
z(r) < 0 当且仅当 r > R;
z(r) = 0 当且仅当 r = R;
z(r) > 0 当且仅当 r < R;
于是可二分求R.
/*==================================================*\
 | 最长有序子序列 (递增/递减/非递增/非递减)
\*==================================================*/
const int N = 1001;
int  a[N], f[N], d[N]; // d[i]用于记录a[0...i]的最大长度
int bsearch(const int *f, int size, const int &a) {
    int  l=0, r=size-1;
```

```
    while( l <= r ){
        int  mid = (l+r)/2;
  if( a > f[mid-1] && a <= f[mid] ) return mid;// >&&<= 换为: >= && <
        else if( a < f[mid] ) r = mid-1;
        else l = mid+1;
    }
}
int LIS(const int *a, const int &n){
     int  i, j, size = 1;
     f[0] = a[0]; d[0] = 1;
     for( i=1; i < n; ++i ){
     if( a[i] <= f[0] ) j = 0;              // <= 换为: <
     else if( a[i] > f[size-1] ) j = size++;// > 换为: >=
          else j = bsearch(f, size, a[i]);
          f[j] = a[i]; d[i] = j+1;
     }
     return size;
}
int main(void){
    int  i, n;
    while( scanf("%d",&n) != EOF ){
    for( i=0; i < n; ++i ) scanf("%d", &a[i]);
    printf("%d\n", LIS(a, n)); // 求最大递增/上升子序列(如果为最大非降子序列,只需把上面的注释部分给与替换)
    }
    return 0;
}
/*==================================================*\
 | 最长公共子序列
\*==================================================*/
int LCS(const char *s1, const char *s2)
{// s1:0...m, s2:0...n
     int m = strlen(s1), n = strlen(s2);
     int i, j;
     a[0][0] = 0;
     for( i=1; i <= m; ++i ) a[i][0] = 0;
     for( i=1; i <= n; ++i ) a[0][i] = 0;
     for( i=1; i <= m; ++i )
          for( j=1; j <= n; ++j ){
           if(s1[i-1]==s2[j-1]) a[i][j] = a[i-1][j-1]+1;
          else if(a[i-1][j]>a[i][j-1])a[i][j]= a[i-1][j];
          else a[i][j] = a[i][j-1];
       }
     return a[m][n];
}
/*==================================================*\
 | 最少找硬币问题（贪心策略-深搜实现）
\*==================================================*/
int  value[7] = {100, 50, 20, 10, 5, 2, 1};
int  count[7]; // count[i]:value[i]硬币的个数
int  res[7];
bool flag;
int main(void){
     //...
          flag = false; // 标识是否已经找到结果
          for( i=0; i < 7; ++i ) res[i] = 0;
          DFS(pay, 0); // pay为要找的钱数
          if( flag ){
               printf("Accept\n%d", res[0]);
               for( i=1; i < 7; ++i ) printf(" %d", res[i]);
               printf("\n");
          }
          else printf("Refuse\n"); // 无法正好找钱
     //...
}
void DFS(int total, int p){
     if( flag ) return ;
     if( p == 7 ) {
          if( total == 0 ) flag = true;
          return ;
     }
     int  i, max = total/value[p];
     if( max > count[p] ) max = count[p];
     for( i=max; i >= 0; --i ){
          res[p] = i;
          DFS(total-i*value[p], p+1);
          if( flag ) return ;
     }
}
/*==================================================*\
 | 棋盘分割
 | 将一个 8＊8 的棋盘进行如下分割：将原棋盘割下一块矩形棋盘并使剩下部分也是矩形，再将剩下的部分继续如此分割，这样割了(n-1)次后，连同最后剩下的矩形棋盘共有 n 块矩形棋盘。(每次切割都只能沿着棋盘格子的边进行) 原棋盘上每一格有一个分值，一块矩形棋盘的总分为其所含各格分值之和。现在需要把棋盘按上述规则分割成 n 块矩形棋盘，并使各矩形棋盘总分的均方差最小。 均方差…，其中平均值…，xi 为第 i 块矩形棋盘的总分。请编程对给出的棋盘及 n，求出 O'的最小值。
 | POJ 1191 棋盘分割
\*==================================================*/
#define min(a, b)  ( (a) < (b) ? (a) : (b) )
const int oo = 10000000;
int  map[8][8];
double  C[16][8][8][8][8];//c[k][si][ei][sj][ej]: 对 矩 阵 //map[si...sj][ei...ej]分割成 k 个矩形(切割 k-1 刀)的结果
double ans; // 平均值
int n;      // 分成 n 块矩形棋盘
void input(void);
void reset(void);
double caluate(int i1, int j1, int i2, int j2);
void dp(int m, int si, int sj, int ei, int ej);

int main(void){
    int  m, i, j, k, l;
    while( scanf("%d", &n) != EOF ){
        input(); reset();
        for( m=1; m <= n; m++ )
            for( i=0; i < 8; i++ )
                for( j=0; j < 8; j++ )
                    for( k=0; k < 8; k++ )
                        for( l=0; l < 8; l++ ){
                            if( (k-i+1)*(l-j+1) < m ) C[m][i][j][k][l] = oo;
                            else{
                                if( m == 1 ){
                                    C[m][i][j][k][l] = pow( (caluate(i,j,k,l)-ans), 2 );
                                }
                                else{
                                    dp(m, i, j, k, l);
                                }
                            }
                        }
         printf("%.3lf\n", sqrt(C[n][0][0][7][7]/n));
    }
    return 0;
}
void input(void){
    int  i, j;
    double sum = 0;
    for( i=0; i < 8; i++ )
        for( j=0; j < 8; j++ ){
            scanf("%d", &map[i][j]);
            sum += map[i][j];
        }
    ans = sum/double(n); // 平均值
}
void reset(void){
    int  i, j, k, l, m;
    for( m=0; m <= n; m++ )
        for( i=0; i < 8; i++ )
            for( j=0; j < 8; j++ )
                for( k=0; k < 8; k++ )
                    for( l=0; l < 8; l++ )
                        C[m][i][j][k][l] = 0;
}
double caluate(int i1, int j1, int i2, int j2){
    double sum=0;
    int i, j;
    for( i=i1; i <= i2; i++ )
        for( j=j1; j <= j2; j++ ) sum += map[i][j];
    return sum;
}
void dp(int m, int si, int sj, int ei, int ej){
    int i, j;
    double  mins = oo;
    for( j=sj; j < ej; j++ ) {// 竖刀
         mins = min(mins, C[1][si][sj][ei][j]+C[m-1][si][j+1][ei][ej]);
```

```
        mins                    =               min(mins,
C[m-1][si][sj][ei][j]+C[1][si][j+1][ei][ej]);
    }
    for( i=si; i < ei; i++ ) { // 横刀
        mins                    =               min(mins,
C[1][si][sj][i][ej]+C[m-1][i+1][sj][ei][ej]);
        mins                    =               min(mins,
C[m-1][si][sj][i][ej]+C[1][i+1][sj][ei][ej]);
    }
    C[m][si][sj][ei][ej] = mins;
}
/*==================================================*\
 | 汉诺塔
 | 1,2,...,n 表示 n 个盘子.数字大盘子就大.n 个盘子放在第 1 根柱子上.大
 | 盘不能放在小盘上. 在第 1 根柱子上的盘子是 a[1],a[2],...,a[n].
 |a[1]=n,a[2]=n-1,...,a[n]=1.即 a[1]是最下面的盘子. 把 n 个盘子
 ||移动到第 3 根柱子. 每次只能移动 1 个盘子，且大盘不能放在小盘上. 问|
 第 m 次移动的是哪一个盘子，从哪根柱子移到哪根柱子.例如：n=3,m=2. 回
 |答是 : 2 1 2, 即移动的是 2 号盘，从第 1 根柱子移动到第 2 根柱子 。
 | HDU 2511  汉诺塔 X
\*==================================================*/
```

一号柱有 n 个盘子,叫做源柱.移往 3 号柱,叫做目的柱.2 号柱叫做中间柱.
全部移往 3 号柱要 f(n) = (2^n) - 1 次.
最大盘 n 号盘在整个移动过程中只移动一次,n-1 号移动 2 次,i 号盘移动 2^(n-i)次.
1 号盘移动次数最多,每 2 次移动一次.
第 2k+1 次移动的是 1 号盘,且是第 k+1 次移动 1 号盘.
第 4k+2 次移动的是 2 号盘,且是第 k+1 次移动 2 号盘.
......
第(2^s)k+2^(s-1)移动的是 s 号盘,这时 s 号盘已被移动了 k+1 次.
每 2^s 次就有一次是移动 s 号盘.
第一次移动 s 号盘是在第 2^(s-1)次.
第二次移动 s 号盘是在第 2^s+2^(s-1)次.
......
第 k+1 次移动 s 号盘是在第 k*2^s+2^(s-1)次.
1--2--3--1 叫做顺时针方向,1--3--2--1 叫做逆时针方向.
最大盘 n 号盘只移动一次:1--3,它是逆时针移动.
n-1 移动 2 次:1--2--3,是顺时针移动.
如果 n 和 k 奇偶性相同,则 k 号盘按逆时针移动,否则顺时针.

```
int main(void){
    int  i, k;
    scanf("%d", &k);
    for( i=0; i < k; i++ ){
         int  n, l;
         int64  m, j;
       __int64  s, t;
       scanf("%d%I64d", &n, &m);
       s = 1; t = 2;
       for( l=1; l <= n; l++ ){
             if( m%t == s ) break;
          s = t; t *= 2;
       }
       printf("%d ", l);
       j = m/t;
       if( n%2 == l%2 ){// 逆时针
          if( (j+1)%3 == 0 ) printf("2 1\n");
          if( (j+1)%3 == 1 ) printf("1 3\n");
          if( (j+1)%3 == 2 ) printf("3 2\n");
       }
       else{// 逆时针
          if( (j+1)%3 == 0 ) printf("3 1\n");
          if( (j+1)%3 == 1 ) printf("1 2\n");
          if( (j+1)%3 == 2 ) printf("2 3\n");
       }
    }
    return 0;
}
/*==================================================*\
 | STL 中的 priority_queue
\*==================================================*/
priority_queue< string, vector<string>, greater<string> >
asc; // 按值小的优先
priority_queue< string, vector<string>, less<string> > desc;
// 按值大的优先
/*==================================================*\
 | 堆栈
\*==================================================*/
const int MAXSIZE = 10000;
int  a[MAXSIZE], heapsize;
inline void swap(int i, int j){
    int  temp = a[i]; a[i] = a[j]; a[j] = temp;
}
inline int Parent(int i){ return i >> 1; }
inline int Left(int i){ return 1 << i; }
inline int Right(int i){ return (1 << i) + 1; }
// 保持堆的性质
void  MaxHeapify(int i){
     int  l = Left(i), r = Right(i), largest;
     if( l <= heapsize && a[l] > a[i] ) largest = l;
     else largest = i;
     if( r <= heapsize && a[r] > a[largest] ) largest = r;
     if( largest != i ){
          swap(i, largest); MaxHeapify(largest);
     }
}
void BuildMaxHeap(int *arr, int n){
     heapsize = n;
     for( int i=heapsize/2; i > 0; --i ) MaxHeapify(i);
}
void HeapSort(int *arr, int n){
     BuildMaxHeap(arr, n);
     for( int i=n; i > 1; --i ){
          swap(1, i); heapsize--;
          MaxHeapify(1);
     }
}
/*==================================================*\
 | 区间最大频率
 | You are given a sequence of n integers a1 , a2 , ... , an
in non-decreasing order. In addition to that, you are given
several queries consisting of indices i and j (1 ≤ i ≤ j ≤
n). For each query, determine the most frequent value among
the integers ai , ... , aj. POJ 3368  Frequent values
求区间中数出现的最大频率
方法一:线段树.
   先离散化。因为序列是升序，所以先将所有值相同的点缩成一点。这样n规
模就缩小了。建立一个数据结构
   记录缩点的属性：在原序列中的值id,和该值有多少个num
   比如序列
   10
   -1 -1 1 1 1 1 3 10 10 10
   缩点后为:下标  1 2 3 4
                id -1 1 3 10
               num 2   4 1 3
   然后建树，树的属性有区间最大值(也就是频率)和区间总和。
   接受询问的时候。接受的是原来序列的区间[be,ed]
   我们先搜索一下两个区间分别在离散化区间后的下标。
   比如接受[2,3]时候相应下标区间就是[1,2];[3,10]的相应下标区间是
[2,4];
   处理频率的时候,我们发现两个极端,也就是左右两个端点的频率不好处理。
因为它们是不完全的频率
   也就是说有部分不在区间内。但是如果对于完全区间，也就是说左右端点下
标值完全在所求区间内。
   比如上例的[2,3]不好处理。但是如果是[1,6]，或是[1,10]就很好处理
了，只要像RMQ一样询问区间最大值就可以了。
方法二:RMQ. 我们可以转化一下问题。将左右端点分开来考虑。
   现在对于离散后的询问区间我们可以分成3个部分.左端点，中间完全区间，
右端点。
   对于中间完全区间线段树或RMQ都能轻松搞定。只要特判一左右的比较一下
就得最后解了。
\*==================================================*/
const int N = 100010;
struct NODE{
     int  b, e;  // 区间[b, e]
     int  l, r;  // 左右子节点下标
     int  number;// 区间内的最大频率值
     int   last;  // 以 data[e]结尾且与 data[e]相同的个
数:data[e-last+1]...data[e]
}node[N*2+1];
int len, data[N];
int main(void){
     int  n;
     while( scanf("%d", &n), n ){
          int  i, q, a, b;
          scanf("%d", &q);
          for( i=0; i < n; i++ ) scanf("%d", &data[i]);
          len = 0; // 下标
          build(0, n-1);
          while( q-- ){
               scanf("%d%d", &a, &b);
```

```
            printf("%d\n", query(0, a-1, b-1)); // 输出区间的最大频率值,而非 data[]
        }
    }
    return 0;
}
int build(int a, int b){ // 建立线段树
    int temp = len, mid = (a+b)/2;
    node[temp].b = a, node[temp].e = b;
    len++;
    if( a == b ){
        node[temp].number = 1;
        node[temp].last = 1; //
        return temp;
    }
    node[temp].l = build(a, mid);
    node[temp].r = build(mid+1, b);

    int left_c=node[temp].l, right_c=node[temp].r, p, lcount=0, rcount=0, rec, max=0;

    rec = data[mid]; p = mid;
    while( p >= a && data[p] == rec ) { p--, lcount++; }
    node[left_c].last = lcount; //

    rec = data[mid+1]; p = mid+1;
    while( p <= b && data[p] == rec ) { p++, rcount++; }
    node[right_c].last = rcount;//

    if( data[mid] == data[mid+1] )  max = lcount+rcount;

    if( node[left_c].number > max ) max = node[left_c].number;
    if( node[right_c].number > max ) max = node[right_c].number;
    node[temp].number = max;

    return temp;
}
int query(int index, int a, int b){
    int begin=node[index].b, end=node[index].e, mid=(begin+end)/2;

    if( a == begin && b == end ) return node[index].number;

    if( a > mid )  return query(node[index].r, a, b);
    if( b < mid+1 ) return query(node[index].l, a, b);

    int temp1, temp2, max;
    if( node[index].l > 0 )  temp1 = query(node[index].l, a, mid);
    if( node[index].r > 0 ) temp2 = query(node[index].r, mid+1, b);
    max = temp1 > temp2 ? temp1 : temp2;

    if( data[mid] != data[mid+1] ) return max;

    temp1 = node[ node[index].l ].last > (mid-a+1) ? (mid-a+1) : node[ node[index].l ].last;
    temp2 = node[ node[index].r ].last > (b-mid) ? (b-mid) : node[ node[index].r ].last;
    if( max < temp1+temp2 ) max = temp1+temp2;

    return  max;
}
/*==================================================*\
 | 取第 k 个元素
 | k=0..n-1,平均复杂度 O(n) 注意 a[]中的顺序被改变
\*==================================================*/
#define _cp(a,b) ((a)<(b))
typedef int elem_t;
elem_t kth_element(int n,elem_t* a,int k){// a[0…n-1]
    elem_t t,key;
    int l=0,r=n-1,i,j;
    while (l<r){
        for (key=a[((i=l-1)+(j=r+1))>>1];i<j;){
            for (j--;_cp(key,a[j]);j--);
            for (i++;_cp(a[i],key);i++);
            if (i<j) t=a[i],a[i]=a[j],a[j]=t;
        }
        if (k>j) l=j+1;
        else r=j;
    }
    return a[k];
}
/*==================================================*\
 | 归并排序求逆序数
 | (也可以用树状数组做)
 | a[0…n-1] cnt=0; call: MergeSort(0, n)
\*==================================================*/
void MergeSort(int l, int r){
    int mid, i, j, tmp;
    if( r > l+1 ){
        mid = (l+r)/2;
        MergeSort(l, mid);
        MergeSort(mid, r);
        tmp = l;
        for( i=l, j=mid; i < mid && j < r; ){
            if( a[i] > a[j] ){
                c[tmp++] = a[j++];
                cnt += mid-i; //
            }
            else c[tmp++] = a[i++];
        }
        if( j < r ) for( ; j < r; ++j ) c[tmp++] = a[j];
        else  for( ; i < mid; ++i )  c[tmp++] = a[i];
        for ( i=l; i < r; ++i )  a[i] = c[i];
    }
}
/*==================================================*\
 | 逆序数推排列数
 | 动态规划：f(n,m)表示逆序数为 m 的 n 元排列的个数，则
 | f(n+1,m)=f(n,m)+f(n,m-1)+...+f(n,m-n)(当b<0 时,f(a,b)=0)
 | 优化 又考虑到如果直接利用上式计算时间复杂度为 O(n^3)，我们分析上
 | 式不难发现 f(n+1,m)=f(n,m)+f(n+1,m-1)
 | if( m-n-1 >= 0 ) f(n+1, m) -= f(n, m-n-1).
 | JOJ 2443
\*==================================================*/
const int N = 1001;
const int C = 10001;
const long MOD = 1000000007;
long arr[N][C];
long long temp;

int main(void){
    int i, j;
    arr[1][0] = arr[2][0] = arr[2][1] = 1;
    for( i=3; i < N; ++i ){
        arr[i][0] = 1;
        long h = i*(i+1)/2+1;
        if( h > C ) h = C;
        for( j=1; j < h; ++j ){
            temp = arr[i-1][j] + arr[i][j-1];
            arr[i][j] = temp%MOD;
            if( j-i >= 0 ) {
                arr[i][j] -= arr[i-1][j-i];
                if( arr[i][j] < 0 )
                {//注意: 由于 arr[i][j]和 arr[i-1][j-i]都是模过的，所以可能会得到负数
                    arr[i][j] += MOD;
                }
            }
        }
    }
    while( scanf("%d %d", &i, &j) != EOF ){
        printf("%ld\n", arr[i][j]);
    }
    return 0;
}
/*==================================================*\
 | 二分查找
\*==================================================*/
// 在[l, r)范围内查找值 v，返回下标
// 假设 a 数组已经按从小到大排序
// 失败返回-1
int bs(int a[], int l, int h, int v){
    int m;
    while ( l < h ){
        m = ( l + h ) >> 1;
        if (a[m] == v) return m;
```

```
        if (a[m] <  v) l=m+1;
        else           h=m;
    }
    return -1;
}
/*==================================================*\
 | 二分查找（大于等于 v 的第一个值）
\*==================================================*/
// 传入参数必须 l <= h
// 返回值 l 总是合理的
int bs(int a[], int l, int h, int v){
    int m;
    while ( l < h ){
        m = ( l + h ) >> 1;
        if (a[m] < v) l=m+1;
        else          h=m;
    }
    return l;
 }
/*==================================================*\
 | 所有数位相加
 | dig(x)  := x                          if 0 <= x <= 9
 | dig(x)  := dig(sum of digits of x)    if x >= 10
\*==================================================*/
方法一：模拟
int dig(int x){
     if( x < 10 ) return x;
     int sum = 0;
     while( x ) { sum += x%10; x /= 10; }
     return dig(sum);
}
方法二：公式  【不太明白...】
int dig(int x){ return (x+8)%9+1; }
```

Number 数论

```
/*==================================================*\
 | 递推求欧拉函数 phi(i)
\*==================================================*/
for (i = 1; i <= maxn; i++) phi[i] = i;
for (i = 2; i <= maxn; i += 2) phi[i] /= 2;
for (i = 3; i <= maxn; i += 2) if(phi[i] == i) {
    for (j = i; j <= maxn; j += i)
        phi[j] = phi[j] / i * (i - 1);
}
/*==================================================*\
 | 单独求欧拉函数 phi(x)
\*==================================================*/
unsigned euler(unsigned x)
{// 就是公式
    unsigned i, res=x;
    for (i = 2; i < (int)sqrt(x * 1.0) + 1; i++)
        if(x%i==0) {
            res = res / i * (i - 1);
            while (x % i == 0) x /= i; // 保证i一定是素数
        }
    if (x > 1) res = res / x * (x - 1);
    return res;
}
/*==================================================*\
 | GCD 最大公约数
\*==================================================*/
int gcd(int x, int y){
    if (!x || !y) return x > y ? x : y;
    for (int t; t = x % y; x = y, y = t);
    return y;
}
/*==================================================*\
 | 快速 GCD
\*==================================================*/
int kgcd(int a, int b){
    if (a == 0) return b;
    if (b == 0) return a;
    if (!(a & 1) && !(b & 1)) return kgcd(a>>1, b>>1) << 1;
    else if (!(b & 1)) return kgcd(a, b>>1);
    else if (!(a & 1)) return kgcd(a>>1, b);
    else return kgcd(abs(a - b), min(a, b));
}
/*==================================================*\
 | 扩展 GCD
 | 求x, y使得gcd(a, b) = a * x + b * y;
\*==================================================*/
int extgcd(int a, int b, int & x, int & y){
   if (b == 0) { x=1; y=0; return a; }
   int d = extgcd(b, a % b, x, y);
   int t = x; x = y; y = t - a / b * y;
   return d;
}
/*==================================================*\
 | 模线性方程 a * x = b (% n)
\*==================================================*/
void modeq(int a, int b, int n)   // ! n > 0
{
    int e, i, d, x, y;
    d = extgcd(a, n, x, y);
    if (b % d > 0) printf("No answer!\n");
    else {
        e = (x * (b / d)) % n;
        for (i = 0; i < d; i++) // !!! here x maybe < 0
            printf("%d-th ans: %d\n", i+1, (e+i*(n/d))%n);
    }
}
/*==================================================*\
 | 模线性方程组
 | a=B[1](% W[1]); a=B[2](% W[2]); ... a=B[k](% W[k]);
 | 其中W, B已知, W[i]>0且W[i]与W[j]互质, 求a; (中国余数定理)
\*==================================================*/
int china(int b[], int w[], int k){
    int i, d, x, y, m, a = 0, n = 1;
    for (i = 0; i < k; i++) n *= w[i];   // ! 注意不能overflow
    for (i = 0; i < k; i++) {
        m = n / w[i];
        d = extgcd(w[i], m, x, y);
        a = (a + y * m * b[i]) % n;
    }
    if (a > 0) return a;
    else return (a + n);
}
/*==================================================*\
 | 筛素数 [1..n]
\*==================================================*/
bool is[N]; int prm[M];
int getprm(int n){
    int i, j, k = 0;
    int s, e = (int)(sqrt(0.0 + n) + 1);
    memset(is, 1, sizeof(is));
    prm[k++] = 2; is[0] = is[1] = 0;
    for (i = 4; i < n; i += 2) is[i] = 0;
    for (i = 3; i < e; i += 2) if (is[i]) {
        prm[k++] = i;
        for (s = i * 2, j = i * i; j < n; j += s)
            is[j] = 0;
        // 因为j是奇数，所以+奇数i后是偶数，不必处理!
    }
    for ( ; i < n; i += 2) if (is[i]) prm[k++] = i;
    return k;          // 返回素数的个数
}
/*==================================================*\
 | 高效求小范围素数 [1..n]
\*==================================================*/
int prime[500],num,boo[2500];
for( i=2; i <= 2300; i++ ) boo[i] = 0;
for( i=2; i <= 50; i++ )
   for( k=i*2; k <= 2300; k += i )
        boo[k] = 1;
int  num = 0;
for( i=2; i <= 2300; i++ )
    if( boo[i] == 0 )  prime[num++] = i;
/*==================================================*\
 | 随机素数测试(伪素数原理)
 | CALL: bool res = miller(n);
 | 快速测试n是否满足素数的'必要'条件, 出错概率很小;
 | 对于任意奇数 n>2 和正整数 s, 算法出错概率 <= 2^(-s);
\*==================================================*/
int witness(int a, int n)
{
    int x, d=1, i = ceil(log(n - 1.0) / log(2.0)) - 1;
    for ( ; i >= 0; i--) {
        x = d; d = (d * d) % n;
        if (d==1 && x!=1 && x!=n-1) return 1;
        if (((n-1) & (1<<i)) > 0) d = (d * a) % n;
    }
    return (d == 1 ? 0 : 1);
}
int miller(int n, int s = 50)
{
    if (n == 2) return 1;
    if ((n % 2) == 0) return 0;
    int j, a;
    for (j = 0; j < s; j++) {
        a = rand() * (n-2) / RAND_MAX + 1;
    // rand()只能随机产生[0, RAND_MAX)内的整数
    // 而且这个RAND_MAX只有32768直接%n的话永远也产生不了
    // [RAND-MAX, n)之间的数
        if (witness(a, n)) return 0;
    }
    return 1;
}
/*==================================================*\
 | 组合数学相关
\*==================================================*/
```

1. {1, 2, ... n}的r组合a1, a2, ... ar出现在所有r组合中的字典序位置编号, C(n, m)表示n中取m的组合数; index =
C(n, r) - C(n - a1, r) - C(n - a2, r-1) - ... - C(n - ar, 1)

2. k * C(n, k) = n * C(n-1, k-1);
   C(n, 0) + C(n, 2) + ... = C(n, 1) + C(n, 3) + ...
   1 * C(n, 1) + 2 * C(n, 2) + ... + n * C(n, n) = n * 2^(n-1)

```
3. Catalan数:  C_n = C(2*n, n) / (n+1)
               C_n = (4*n-2)/(n+1) * C_n-1
               C_1 = 1

4. 第二类Stirling数: S(p, k) = k * S(p-1, k) + S(p-1, k-1).
   S(p, 0) = 0, (p>=1);  S(p, p) = 1, (p>=0);
   且有 S(p, 1) = 1, (p>=1);
        S(p, 2) = 2^(p-1) - 1, (p>=2);
        S(p, p-1) = C(p, 2);
   含义: 将p个元素划分到k个同样的盒子, 每个盒子非空的方法数.
               1      k
   S(p, k) = --- * sigma((-1)^t * C(k, t) * (k-t)^p)
              k!     t=0

5. Bell数: B_p = S(p, 0) + S(p, 1) + ... + S(p, p)
   B_p = C(p-1,0)*B_0 + C(p-1,1)*B_1 + ... C(p-1,p-1)*B_(p-1)

6. 第一类stirling数:
   s(p, k)是将p个物体排成k个非空的循环排列的方法数.
   (或者: 把p个人排成k个非空圆圈的方法数)
   s(p, k) = (p-1) * s(p-1, k) + s(p-1, k-1);
/*==================================================*\
 | Polya 计数
 | c种颜色的珠子, 组成长为s的项链, 项链没有方向和起始位置;
\*==================================================*/
int gcd (int a, int b) { return b ? gcd(b,a%b) : a; }
int main (void){
    int c, s;
    while (scanf("%d%d", &c, &s)==2) {
        int k;
        long long p[64]; p[0] = 1; // power of c
        for (k=0 ; k<s ; k++) p[k+1] = p[k] * c;
        // reflection part
        long long count = s&1 ? s * p[s/2 + 1] :
                          (s/2) * (p[s/2] + p[s/2 + 1]);
        // rotation part
        for (k=1 ; k<=s ; k++) count += p[gcd(k, s)];
        count /= 2 * s;
        cout << count << '\n';
    }
    return 0;
}
/*==================================================*\
 | 组合数 C(n, r)
\*==================================================*/
int com(int n, int r){// return C(n, r)
    if( n-r > r ) r = n-r; // C(n, r) = C(n, n-r)
    int i, j, s = 1;
    for( i=0, j=1; i < r; ++i ){
        s *= (n-i);
        for( ; j <= r && s%j == 0; ++j ) s /= j;
    }
    return s;
}
/*==================================================*\
 | 最大 1 矩阵
\*==================================================*/
bool a[N][N];
int Run(const int &m, const int &n)// a[1...m][1...n]
{// O(m*n)
    int i, j, k, l, r, max = 0;
    int col[N];
    for( j=1; j <= n; ++j )
        if( a[1][j] == 0 ) col[j] = 0;
        else{
            for( k=2; k <= m && a[k][j] == 1; ++k );
            col[j] = k-1;
        }
    for( i=1; i <= m; ++i ){
        if( i > 1 ){
            for( j=1; j <= n; ++j )
                if( a[i][j] == 0 ) col[j] = 0;
                else{
                    if( a[i-1][j] == 0 ){
                        for( k=i+1; k <= m && a[k][j] == 1; ++k );
                        col[j] = k-1;
                    }
                }
        }
        for( j=1; j <= n; ++j )
            if( col[j] >= i ){
                for( l=j-1; l > 0 && col[l] >= col[j]; --l ); ++l;
                for( r=j+1; r <= n && col[r] >= col[j]; ++r ); --r;
                int res = (r-l+1)*(col[j]-i+1);
                if( res > max ) max = res;
            }
    }
    return max;
}
/*==================================================*\
 | 约瑟夫环问题(数学方法)
 | n个人(编号 1...n),先去掉第 m 个数,然后从 m+1 个开始报 1,
 | 报到 k 的退出,剩下的人继续从 1 开始报数.求胜利者的编号.
 | POJ 3157  And Then There Was One
\*==================================================*/
int main(void){
    int n, k, m;
    while( scanf("%d%d%d", &n, &k, &m), n || k || m ){
        int i, d, s=0;
        for( i=2; i <= n; ++i ) s = (s+k)%i;
        k = k%n; if( k == 0 ) k=n;
        d = (s+1) + (m-k);
        if( d >= 1 && d <= n ) printf("%d\n", d);
        else if( d < 1 ) printf("%d\n", n+d);
        else if( d > n ) printf("%d\n", d%n);
    }
    return 0;
}
/*==================================================*\
 | 约瑟夫环问题(数组模拟)
\*==================================================*/
int main(void){
    int n, k, m;
    while( scanf("%d%d%d", &n, &k, &m), n || k || m ){
        int i, cur;
        for( i=0; i < m-1; ++i ) a[i] = i;
        for( ; i < n; ++i ) a[i] = i+1;
        cur = m-1;
        while( --n ){
            cur = (cur+k-1)%n;
            for( i=cur+1; i < n; ++i ) a[i-1] = a[i];
        }
        printf("%d\n", a[0]+1);
    }
    return 0;
}
/*==================================================*\
 | 取石子游戏 1
 | 1堆石子有n个,两人轮流取.先取者第1次可以取任意多个, 但不能全部取
 |完.以后每次取的石子数不能超过上次取子数的2倍。取完者胜.先取者负输
 |出"Second win".先取者胜输出"First win". JOJ 1063
\*==================================================*/
const int N = 51;
double arr[N] = {2, 3};
int main(void){
    int i;
    double n;
    for( i=2; i < N; ++i ) arr[i] = arr[i-1] + arr[i-2];
    while( scanf("%lf", &n), n != 0 ){
        for( i=0; i < N; ++i )
            if( arr[i] == n ){
                printf("Second win\n"); break;
            }
        if( i == N ) printf("First win\n");
    }
    return 0;
}
/*==================================================*\
 | 集合划分问题
 | n 元集合分划为 k 类的方案数记为 S(n, k), 称为第二类 Stirling 数。
 | 如{A,B,C}可以划分{{A},{B},{C}}, {{A,B},{C}}, {{B,C},{A}},
 | {{A,C},{B}},{{A,B,C}}。即一个集合可以划分为不同集合(1…n 个)
 | 的种类数    HDU 一卡通大冒险
 | CALL: compute(N); 每当输入一个 n, 输出 B[n]
\*==================================================*/
```

```
const int N = 2001;
int data[N][N], B[N];
void NGetM(int m, int n)// m个数n个集合
{// data[i][j]:i个数分成j个集合
    int  min, i, j;
    data[0][0] = 1; //
    for( i = 1; i <= m; ++i ) data[i][0] = 0;
    for( i = 0; i <= m; ++i ) data[i][i+1] = 0;
    for( i = 1; i <= m; ++i ){
        if( i < n ) min = i;
        else min = n;
        for( j = 1; j <= min; ++j ){
        data[i][j] = (j*data[i-1][j] + data[i-1][j-1]);
        }
    }
}
void compute(int m){// b[i]:Bell数
    NGetM(m, m);
    memset(B, 0, sizeof(B));
    int i, j;
    for( i=1; i <= m; ++i )
        for( j=0; j <= i; ++j ) B[i] += data[i][j];
}
/*==================================================*\
 | 大数平方根（字符串数组表示）
\*==================================================*/
void Sqrt(char *str){
    double i, r, n;
    int  j, l, size, num, x[1000];
    size = strlen(str);
    if( size == 1 && str[0] == '0' ){
        printf("0\n");  return;
    }
    if( size%2 == 1 ){
        n = str[0]-48; l = -1;
    }
    else{
        n = (str[0]-48)*10+str[1]-48; l = 0;
    }
    r = 0; num = 0;
    //////////////////
    while(true){
        i = 0;
        while( i*(i+20*r) <= n ) ++i;
        --i;
        n -= i*(i+20*r);
        r = r*10+i;
        x[num] = (int)i;
        ++num;
        l += 2;
        if( l >= size )   break;
n = n*100+(double)(str[l]-48)*10+(double)(str[l+1]-48);
    }
    //////////////////
    for(j = 0;j < num; j++)   printf("%d", x[j]);
    printf("\n");
}
/*==================================================*\
 | 大数取模的二进制方法
\*==================================================*/
```

求 a^b mod c
把b化成二进制串的形式：b = (at at-1 at-2 … a1 a0)
那么有：b = at*2^t + at-1*2^(t-1) + … … + a1*2^1 + a0*2^0，其中 ai=0,1.
则：a^b mod c = a^(at*2^t + at-1*2^(t-1) + … … + a1*2^1 + a0*2^0) mod c = ((a^(a0*2^0) mod c) * a^(a1*2^1) mod c)… …
注意到：a^(2^(i+1))mod c = (a^(2^i) mod c)^2 mod c,这样就可以在常数项时间内由2^i项推出2^(i+1)项。时间复杂度为O((logb)^3).

```
int mod_exp(int a,int b0,int n)//return a^b0 % n
{
   if( a > n ) a %= n;
   int  i, d = 1, b[35];
   for( i=0; i < 35; ++i ){
      b[i] = b0%2;
      b0 /= 2;
      if( b0 == 0 )  break;
   } //b[i]b[i-1]...b[0]为b0的二进制表示
   for( ;i >= 0; --i ){
      d = (d*d)%n;
      if( b[i] == 1 )  d = (d*a)%n;
   }
   return d;
}
/*==================================================*\
 | 线性方程组 a[][]x[]=b[]
\*==================================================*/
#define MAXN 100
#define fabs(x)  ((x)>0?(x):-(x))
#define eps 1e-10
//列主元gauss消去求解a[][]x[]=b[]
//返回是否有唯一解,若有解在b[]中
int gauss_cpivot(int n,double a[][MAXN],double b[]){
    int i,j,k,row;
    double maxp,t;
    for (k=0;k<n;k++){
        for (maxp=0,i=k;i<n;i++)
            if (fabs(a[i][k])>fabs(maxp))
                maxp=a[row=i][k];
        if (fabs(maxp)<eps)
            return 0;
        if (row!=k){
            for (j=k;j<n;j++)

    t=a[k][j],a[k][j]=a[row][j],a[row][j]=t;
            t=b[k],b[k]=b[row],b[row]=t;
        }
        for (j=k+1;j<n;j++){
            a[k][j]/=maxp;
            for (i=k+1;i<n;i++)
                a[i][j]-=a[i][k]*a[k][j];
        }
        b[k]/=maxp;
        for (i=k+1;i<n;i++)
            b[i]-=b[k]*a[i][k];
    }
    for (i=n-1;i>=0;i--)
        for (j=i+1;j<n;j++)
            b[i]-=a[i][j]*b[j];
    return 1;
}

//全主元gauss消去解a[][]x[]=b[]
//返回是否有唯一解,若有解在b[]中
int gauss_tpivot(int n,double a[][MAXN],double b[]){
    int i,j,k,row,col,index[MAXN];
    double maxp,t;
    for (i=0;i<n;i++)
        index[i]=i;
    for (k=0;k<n;k++){
      for (maxp=0,i=k;i<n;i++)
            for (j=k;j<n;j++)
                if (fabs(a[i][j])>fabs(maxp))
                    maxp=a[row=i][col=j];
      if (fabs(maxp)<eps)
            return 0;
        if (col!=k){
            for (i=0;i<n;i++)

    t=a[i][col],a[i][col]=a[i][k],a[i][k]=t;

    j=index[col],index[col]=index[k],index[k]=j;
        }
        if (row!=k){
            for (j=k;j<n;j++)
             t=a[k][j],a[k][j]=a[row][j],a[row][j]=t;
            t=b[k],b[k]=b[row],b[row]=t;
        }
        for (j=k+1;j<n;j++){
            a[k][j]/=maxp;
            for (i=k+1;i<n;i++)
                a[i][j]-=a[i][k]*a[k][j];
        }
        b[k]/=maxp;
        for (i=k+1;i<n;i++)
            b[i]-=b[k]*a[i][k];
    }
    for (i=n-1;i>=0;i--)
        for (j=i+1;j<n;j++)
            b[i]-=a[i][j]*b[j];
    for (k=0;k<n;k++)
```

```
        a[0][index[k]]=b[k];
    for (k=0;k<n;k++)
        b[k]=a[0][k];
    return 1;
}
/*==================================================*\
 | 追赶法解周期性方程
周期性方程定义：
| a1 b1 c1 ...              |       =  x1
|    a2 b2 c2 ...           |       =  x2
|       ...                 |       =  ...
|cn-1 ...        an-1 bn-1  |       =  xn-1
| bn cn                an   |       =  xn
输入：a[],b[],c[],x[]
输出：求解结果 X 在 x[]中
\*==================================================*/
void run(){
    c[0] /= b[0]; a[0] /= b[0]; x[0] /= b[0];
    for (int i = 1; i < N - 1; i ++) {
        double temp = b[i] - a[i] * c[i - 1];
        c[i] /= temp;
        x[i] = (x[i] - a[i] * x[i - 1]) / temp;
        a[i] = -a[i] * a[i - 1] / temp;
    }
    a[N - 2] = -a[N - 2] - c[N - 2];
    for (int i = N - 3; i >= 0; i --) {
        a[i] = -a[i] - c[i] * a[i + 1];
        x[i] -= c[i] * x[i + 1];
    }
    x[N - 1] -= (c[N - 1] * x[0] + a[N - 1] * x[N - 2]);
    x[N - 1] /= (c[N - 1] * a[0] + a[N - 1] * a[N - 2] + b[N
- 1]);
    for (int i = N - 2; i >= 0; i --)
        x[i] += a[i] * x[N - 1];
}
/*==================================================*\
 | 阶乘最后非零位,复杂度 O(nlogn)
\*==================================================*/
//返回该位, n 以字符串方式传入
#include <string.h>
#define MAXN 10000
int lastdigit(char* buf){
const int mod[20]={1,1,2,6,4,2,2,4,2,8,4,
4,8,4,6,8,8,6,8,2};
    int len=strlen(buf),a[MAXN],i,c,ret=1;
    if (len==1) return mod[buf[0]-'0'];
    for (i=0;i<len;i++)  a[i]=buf[len-1-i]-'0';
    for (;len;len-=!a[len-1]){
        ret=ret*mod[a[1]%2*10+a[0]]%5;
        for (c=0,i=len-1;i>=0;i--)
            c=c*10+a[i],a[i]=c/5,c%=5;
    }
    return ret+ret%2*5;
}
```

递归方法求解排列组合问题

类循环排列

输入样例
3 2
输出样例
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
代码

```
#include <stdio.h>
#define MAX_N  10
int n, m;          //相当于n重循环，每重循环长度为m
int rcd[MAX_N];    //记录每个位置填的数
void loop_permutation(int l){
     int i;
     if (l == n)  {              //相当于进入了n重循环的最内层
          for (i=0; i<n; i++){
               printf("%d", rcd[i]);
               if (i < n-1) printf(" ");
          }
          printf("\n"); return ;
     }
     for (i=0; i<m; i++){        //每重循环长度为m
          rcd[l] = i;            //在l位置放i
          loop_permutation(l+1); //填下一个位置
     }
}
int main(void){
     while (scanf("%d%d", &n, &m) != EOF)
     loop_permutation(0);
     return 0;
}
```

实际上，这样的方法是用递归实现多重循环，本递归程序相当于 n 重循环，每重循环的长度为 m 的情况，所以输出共有 m^n 行。

全排列

对输入的 n 个数作全排列。
输入样例
3
1 2 3
输出样例
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
代码

```
#include <stdio.h>
#define MAX_N 10
int n;            //共n个数
int rcd[MAX_N];  //记录每个位置填的数
int used[MAX_N]; //标记数是否用过
int num[MAX_N];  //存放输入的n个数
void full_permutation(int l){
     int i;
     if (l == n){
          for (i=0; i<n; i++){
               printf("%d", rcd[i]);
               if (i < n-1)   printf(" ");
          }
          printf("\n"); return ;
     }
     for (i=0; i<n; i++) //枚举所有的数(n个)，循环从开始
          if (!used[i]){  //若num[i]没有使用过，则标记为已使用
               used[i] = 1; rcd[l] = num[i];//在l位置放上该数
               full_permutation(l+1); //填下一个位置
               used[i] = 0;            //清标记
          }
}
int read_data(){
     int i;
     if (scanf("%d", &n) == EOF) return 0;
     for (i=0; i<n; i++)   scanf("%d", &num[i]);
     for (i=0; i<n; i++)   used[i] = 0;
     return 1;
}
int main(void){
     while (read_data()) full_permutation(0);
     return 0;
}
```

程序通过 used 数组，标记数是否被用过，可以产生全排列，共有 n!种。但是，通过观察会发现，若输入的 n 个数有重复，那么在输出的 n!种排列中，必然存在重复的项，若不允许输出重复的项，该怎么做呢？

不重复排列

输入 n 个数，输出由这 n 个数构成的排列，不允许出现重复的项。
输入样例
3
1 1 2
输出样例
1 1 2
1 2 1
2 1 1
代码

```
#include <stdio.h>
#define MAX_N 10
int n, m;         //共有n个数，其中互不相同的有m个
int rcd[MAX_N];  //记录每个位置填的数
int used[MAX_N]; //标记m个数可以使用的次数
int num[MAX_N];  //存放输入中互不相同的m个数
void unrepeat_permutation(int l){
     int i;
     if (l == n){ //填完了n个数，则输出
          for (i=0; i<n; i++){
               printf("%d", rcd[i]);
               if (i < n-1)  printf(" ");
          }
          printf("\n"); return ;
     }
     for (i=0; i<m; i++)    //枚举m个本质不同的数
          if (used[i] > 0){ //若数num[i]还没被用完，则可使用次
数减
               used[i]--; rcd[l] = num[i];//在l位置放上该数
               unrepeat_permutation(l+1); //填下一个位置
               used[i]++; //可使用次数恢复
          }
}
int read_data(){
     int i, j, val;
     if (scanf("%d", &n) == EOF)  return 0;
     m = 0;
     for (i=0; i<n; i++){
          scanf("%d", &val);
          for (j=0; j<m; j++)
               if (num[j] == val){
                    used[j]++; break;
               }
          if (j == m){
```

```
            num[m] = val; used[m++] = 1;
        }
    }
    return 1;
}
int main(){
    while (read_data()) unrepeat_permutation(0);
    return 0;
}
```

本程序将全排列中的 used 标记数组改为记录输入中每个本质不同的数出现的次数，并在递归函数中使用。需要注意的是，在输入过程中，应剔除重复的数值。实际上，重复排列的产生是由于同一个位置被多次填入了相同的数，并且这多次填入又是在同一次循环过程中完成的。本方法通过统计每个本质不同的数的个数，使得循环长度由 n 变为 m，这样，i 一旦自增，就再也不会指向原先填入过的数了。这种方法剪去了重复项的生成，从而加快了搜索速度，是深度优先搜索中常用的剪枝技巧。

一般组合

输入 n 个数，从中选出 m 个数可构成集合，输出所有这样的集合。
输入样例
```
4 3
1 2 3 4
```
输出样例
```
1 2 3
1 2 4
1 3 4
2 3 4
```
代码
```
#include <stdio.h>
#define MAX_N 10
int n, m;      //从 n 个数中选出 m 个构成组合
int rcd[MAX_N];  //记录每个位置填的数
int num[MAX_N];  //存放输入的 n 个数
void select_combination(int l, int p){
    int i;
    if (l == m){ //若选出了 m 个数，则打印
        for (i=0; i<m; i++){
            printf("%d", rcd[i]);
            if (i < m-1) printf(" ");
        }
        printf("\n"); return ;
    }
    for (i=p; i < n; i++){//上个位置填的是 num[p-1]，本次从 num[p]开始试探
        rcd[l] = num[i];  //在 l 位置放上该数
        select_combination(l+1, i+1); //填下一个位置
    }
}
int read_data(){
    int i;
    if (scanf("%d%d", &n, &m) == EOF)  return 0;
    for (i=0; i<n; i++)   scanf("%d", &num[i]);
    return 1;
}
int main(){
    while (read_data())  select_combination(0, 0);
    return 0;
}
```

因为在组合生成过程中引入了变量 p，保证了每次填入的数字在 num 中的下标是递增的，所以不需要使用 used 进行标记，共 C(n, m)种组合。

全组合

输入 n 个数，求这 n 个数构成的集合的所有子集。
输入样例
```
3
1 2 3
```
输出样例
```
1
1 2
1 2 3
1 3
2
2 3
3
```
代码
```
#include <stdio.h>
#define MAX_N 10
int n; //共 n 个数
int rcd[MAX_N]; //记录每个位置填的数
int num[MAX_N]; //存放输入的 n 个数
void full_combination(int l, int p){
    int i;
    for (i=0; i<l; i++){ //每次进入递归函数都输出
        printf("%d", rcd[i]);
        if (i < l-1)  printf(" ");
    }
    printf("\n");
    for (i=p; i<n; i++){ //循环同样从 p 开始，但结束条件变为 i>=n
        rcd[l] = num[i]; //在 l 位置放上该数
        full_combination(l+1, i+1); //填下一个位置
    }
}
int read_data(){
    int i;
    if (scanf("%d", &n) == EOF)   return 0;
    for (i=0; i<n; i++)   scanf("%d", &num[i]);
    return 1;
}
int main(){
    while (read_data()) full_combination(0, 0);
    return 0;
}
```

全组合，共 2^n 种，包含空集和自身。与全排列一样，若输入的 n 个数有重复，那么在输出的 2^n 种组合中，必然存在重复的项。避免重复的方法与不重复排列算法中使用的类似。

不重复组合

输入 n 个数，求这 n 个数构成的集合的所有子集，不允许输出重复的项。
输入样例
```
3
1 1 3
```
输出样例
```
1
1 1
1 1 3
1 3
3
```
代码
```
#include <stdio.h>
#define MAX_N 10
int n, m;      //输入 n 个数，其中本质不同的有 m 个
int rcd[MAX_N]; //记录每个位置填的数
int used[MAX_N];//标记 m 个数可以使用的次数
int num[MAX_N]; //存放输入中本质不同的 m 个数
void unrepeat_combination(int l, int p){
    int i;
    for (i=0; i<l; i++){ //每次都输出
        printf("%d", rcd[i]);
        if (i < l-1)  printf(" ");
    }
    printf("\n");
    for (i=p; i<m; i++)  //循环依旧从 p 开始，枚举剩下的本质不同的数
        if (used[i] > 0){//若还可以用，则可用次数减
            used[i]--; rcd[l] = num[i];   //在 l 位置放上该数
            unrepeat_combination(l+1, i); //填下一个位置
            used[i]++; //可用次数恢复
        }
}
int read_data(){
    int i, j, val;
    if (scanf("%d", &n) == EOF)   return 0;
    m = 0;
    for (i=0; i<n; i++){
        scanf("%d", &val);
        for (j=0; j<m; j++)
            if (num[j] == val){
                used[j]++; break;
            }
            if (j == m){
                num[m] = val; used[m++] = 1;
            }
    }
    return 1;
}
int main(){
```

```
    while (read_data()) unrepeat_combination(0, 0);
    return 0;
}
```
需要注意的是递归调用时，第二个参数是 i，不再是全组合中的 i+1！

| 应用

搜索问题中有很多本质上就是排列组合问题，只不过加上了某些剪枝和限制条件。解这类题目的基本算法框架常常是类循环排列、全排列、一般组合或全组合。而不重复排列与不重复组合则是两种非常有效的剪枝技巧。

模式串匹配问题总结

```
/*==================================================*\
 | 字符串 Hash
 | 注意：mod选择足够大的质数（至少大于字符串个数）
\*==================================================*/
unsigned int hasha(char *url, int mod){
    unsigned int n = 0;
    char *b = (char *) &n;
    for (int i = 0; url[i]; ++i) b[i % 4] ^= url[i];
    return n % mod;
}
unsigned int hashb(char *url, int mod){
    unsigned int h = 0, g;
    while (*url) {
        h = (h << 4) + *url++;
        g = h & 0xF0000000;
        if (g) h ^= g >> 24;
        h &= ~g;
    }
    return h % mod;
}
int hashc(char *p, int prime = 25013){
    unsigned int h=0, g;
    for ( ; *p; ++p) {
        h = (h<<4) + *p;
        if(g = h & 0xf0000000) {
            h = h ^ (g >> 24);
            h = h ^ g;
        }
    }
    return h % prime;
}
/*==================================================*\
 | KMP 匹配算法 O(M+N)
 | CALL: res=kmp(str, pat); 原串为str; 模式为pat(长为P);
\*==================================================*/
int fail[P];
int kmp(char* str, char* pat){
     int i, j, k;
     memset(fail, -1, sizeof(fail));
     for (i = 1; pat[i]; ++i) {
         for (k=fail[i-1]; k>=0 && pat[i]!=pat[k+1];
                           k=fail[k]);
         if (pat[k + 1] == pat[i]) fail[i] = k + 1;
     }
     i = j = 0;
     while( str[i] && pat[j] ){ // By Fandywang
        if( pat[j] == str[i] ) ++i, ++j;
        else if(j == 0)++i;//第一个字符匹配失败，从str下个字符开始
        else j = fail[j-1]+1;    }
       if( pat[j] ) return -1;
       else return i-j;
}
/*==================================================*\
 | Karp-Rabin 字符串匹配
 | hash(w[0..m-1]) =
 | (w[0] * 2^(m-1) + ... + w[m-1] * 2^0) % q;
 | hash(w[j+1..j+m]) =
 | rehash(y[j], y[j+m], hash(w[j..j+m-1]);
 | rehash(a, b, h) = ((h - a * 2^(m-1) ) * 2 + b) % q;
 | 可以用q = 2^32简化%运算
\*==================================================*/
#define REHASH(a, b, h) ((((h) - (a)*d) << 1) + (b))
int krmatch(char *x, int m, char *y, int n)
{   // search x in y
    int d, hx, hy, i, j;
    for (d = i = 1; i < m; ++i) d = (d<<1);
    for (hy = hx = i = 0; i < m; ++i) {
        hx = ((hx<<1) + x[i]); hy = ((hy<<1) + y[i]);
    }
    for (j = 0; j <= n - m; ++j) {
        if (hx == hy && memcmp(x, y + j, m) == 0) return j;
        hy = REHASH(y[j], y[j + m], hy);
    }
}
/*==================================================*\
 | 基于 Karp-Rabin 的字符块匹配
 | Text: n * m matrix;  Pattern: x * y matrix;
\*==================================================*/
#define uint unsigned int
const int A=1024, B=128;
const uint E=27;
char text[A][A], patt[B][B];
uint ht, hp, pw[B * B], hor[A], ver[A][A];
int n, m, x, y;
void init(){
    int i, j = B * B;
    for (i=1, pw[0]=1; i<j; ++i) pw[i] = pw[i-1] * E;
}
void hash(){
    int i, j;
    for (i=0; i<n; ++i) for (j=0, hor[i]=0; j<y; ++j) {
        hor[i]*=pw[x]; hor[i]+=text[i][j]-'a';
    }
    for (j=0; j<m; ++j) {
        for (i=0, ver[0][j]=0; i<x; ++i) {
            ver[0][j]*=E; ver[0][j]+=text[i][j]-'a';
        }
        for (i=1 ; i<=n-x; ++i)
            ver[i][j]=
            (ver[i-1][j]-(text[i-1][j]-'a')*pw[x-1])*E
            +text[i+x-1][j]-'a';
    }
    for (j=0, ht=hp=0; j<y; ++j) for (i=0; i<x; ++i) {
        ht*=E; ht+=text[i][j]-'a';
        hp*=E; hp+=patt[i][j]-'a';
    }
}
void read(){
    int i;
    scanf("%d%d", &n, &m);
    for (i=0; i<n; ++i) scanf("%s", text[i]);
    scanf("%d%d", &x, &y);
    for (i=0; i<x; ++i) scanf("%s", patt[i]);
}
int solve(){
    if (n==0||m==0||x==0||y==0) return 0;
    int i, j, cnt=0; uint t;
    for (i=0; i<=n-x; ++i) {
        for (j=0, t=ht; j<=m-y; ++j) {
            if (t==hp) ++cnt;
        t=(t-ver[i][j]*pw[y*x-x])*pw[x]+ver[i][j+y];
        }
        ht=(ht-hor[i]*pw[x-1])*E+hor[i+x];
    }
    return cnt;
}
```

```
int main(void){
     int T; init();
     for (scanf("%d", &T); T; --T) {
          read(); hash();
          printf("%d\n", solve());
     }
     return 0;
}
/*==================================================*\
 | 函数名: strstr
 |功 能: 在串中查找指定字符串的第一次出现
 |用 法: char *strstr(char *str1, char *str2);
 |据说strstr和KMP的算法效率差不多
\*==================================================*/
int main(void){
char *str1 = "Borland International", *str2 = "nation", *ptr;
      ptr = strstr(str1, str2);
      printf("The substring is: %s\n", ptr);
      return 0;
}
/*==================================================*\
 | BM 算法的改进的算法 Sunday Algorithm
BM算法优于KMP
SUNDAY 算法描述：字符串查找算法中，最著名的两个是KMP算法（Knuth-Morris-Pratt)和BM算法（Boyer-Moore)。两个算法在最坏情况下均具有线性的查找时间。但是在实用上，KMP算法并不比最简单的c库函数strstr()快多少，而BM算法则往往比KMP算法快上3－5倍。但是BM算法还不是最快的算法，这里介绍一种比BM算法更快一些的查找算法。
例如我们要在"substring searching algorithm"查找"search"，刚开始时，把子串与文本左边对齐：
substring searching algorithm
search
结果在第二个字符处发现不匹配，于是要把子串往后移动。但是该移动多少呢？这就是各种算法各显神通的地方了，最简单的做法是移动一个字符位置；KMP是利用已经匹配部分的信息来移动；BM算法是做反向比较，并根据已经匹配的部分来确定移动量。这里要介绍的方法是看紧跟在当前子串之后的那个字符（第一个字符串中的'i')。
显然，不管移动多少，这个字符是肯定要参加下一步的比较的，也就是说，如果下一步匹配到了，这个字符必须在子串内。所以，可以移动子串，使子串中的最右边的这个字符与它对齐。现在子串'search'中并不存在'i'，则说明可以直接跳过一大片，从'i'之后的那个字符开始作下一步的比较，如下：
substring searching algorithm
          search
比较的结果，第一个字符就不匹配，再看子串后面的那个字符，是'r',它在子串中出现在倒数第三位，于是把子串向后移动三位，使两个'r'对齐，如下：
substring searching algorithm
              search
这次匹配成功了！回顾整个过程，我们只移动了两次子串就找到了匹配位置，是不是很神啊?!可以证明，用这个算法，每一步的移动量都比BM算法要大，所以肯定比BM算法更快。
\*==================================================*/
void SUNDAY(char *text, char *patt){
     size_t  temp[256];
     size_t  *shift = temp;
     size_t  i, patt_size = strlen(patt), text_size =
strlen(text);
     cout << "size : " << patt_size << endl;
     for( i=0; i < 256; i++ ) *(shift+i) = patt_size+1;
     for( i=0; i < patt_size; i++ )
          *(shift+unsigned char(*(patt+i))) = patt_size-i;
     //shift['s']=6 步,shitf['e']=5 以此类推
     size_t  limit = text_size-patt_size+1;
     for( i=0; i < limit; i += shift[ text[i+patt_size] ] )
          if( text[i] == *patt ){
               char *match_text = text+i+1;
               size_t  match_size = 1;
               do{// 输出所有匹配的位置
                    if( match_size == patt_size ) cout << "the
NO. is " << i << endl;
               }while( (*match_text++) ==
patt[match_size++] );
          }
     cout << endl;
}
int main(void){
     char *text = new char[100];
     text = "substring searching algorithm search";
     char *patt = new char[10];
     patt = "search";
     SUNDAY(text, patt);
     return 0;
}
/*
size : 6
the NO. is 10
the NO. is 30
*/
/*==================================================*\
 | 最短公共祖先（两个长字符串）
 | The shortest common superstring of 2 strings S1 and S2 is
 | a string S with|the minimum number of characters which
 | contains both S1 and S2 as a sequence of consecutive
 | characters. HDU 1841
\*==================================================*/
const int N = 1000010;
char  a[2][N];  int  fail[N];
inline int max(int a, int b) { return ( a > b ) ? a : b; }
int kmp(int &i, int &j, char* str, char* pat){
     int  k;
     memset(fail, -1, sizeof(fail));
     for (i = 1; pat[i]; ++i){
          for (k=fail[i-1]; k>=0 && pat[i]!=pat[k+1];
k=fail[k]);
          if (pat[k + 1] == pat[i]) fail[i] = k + 1;
     }
    i = j = 0;
    while( str[i] && pat[j] ){
          if( pat[j] == str[i] ) ++i, ++j;
          else if(j == 0)++i;//第一个字符匹配失败，从str下个字
符开始
          else j = fail[j-1]+1;
     }
     if( pat[j] ) return -1;
     else return i-j;
}
int main(void){
     int  T;
     scanf("%d\n", &T);
     while( T-- ){
          int  i, j, l1=0, l2=0;
          gets(a[0]); gets(a[1]);
          int len1 = strlen(a[0]), len2 = strlen(a[1]), val;
          val = kmp(i, j, a[1], a[0]); // a[1]在前
          if( val != -1 ) l1 = len1;
          else{
               //printf("i:%d, j:%d\n", i, j);
               if( i == len2 && j-1 >= 0 && a[1][len2-1] ==
a[0][j-1] ) l1 = j;
          }
          val = kmp(i, j, a[0], a[1]); // a[0]在前
          if( val != -1 ) l2 = len2;
          else{
               //printf("i:%d, j:%d\n", i, j);
               if( i == len1 && j-1 >= 0 && a[0][len1-1] ==
a[1][j-1] ) l2 = j;
          }
     //   printf("l1:%d, l2:%d\n", l1, l2);
          printf("%d\n", len1+len2-max(l1, l2));
     }
     return 0;
}
/*==================================================*\
 | 最短公共祖先（多个短字符串）
 | pku 1699  pku 3192  pku 1795
\*==================================================*/
首先用一个数组save[i][j]来保存第j个串加在第i个串之后，第i个串所增加的长度，比如alba  bacau,把bacau加在alba后alba所增加的长度就为3.我们采用搜索的策略，以每一个串为第一个串进行搜索
for(i=1;i<=n;i++)dfs(i)//以第i个串为第一个串进行搜索。
剪枝：主要是在搜索过程中，当前面一些串的长度比当前已经找到的min还大的话就剪去该枝。
```

Geometry 计算几何

```
/*==================================================*\
 | Graham 求凸包 O(N * logN)
 | CALL: nr = graham(pnt, int n, res); res[]为凸包点集;
\*==================================================*/
struct point { double x, y; };
bool mult(point sp, point ep, point op){
    return (sp.x - op.x) * (ep.y - op.y)
        >= (ep.x - op.x) * (sp.y - op.y);
}
bool operator < (const point &l, const point &r){
    return l.y < r.y || (l.y == r.y && l.x < r.x);
}
int graham(point pnt[], int n, point res[]){
    int i, len, k = 0, top = 1;
    sort(pnt, pnt + n);
    if (n == 0) return 0; res[0] = pnt[0];
    if (n == 1) return 1; res[1] = pnt[1];
    if (n == 2) return 2; res[2] = pnt[2];
    for (i = 2; i < n; i++) {
        while (top && mult(pnt[i], res[top], res[top-1]))
            top--;
        res[++top] = pnt[i];
    }
    len = top; res[++top] = pnt[n - 2];
    for (i = n - 3; i >= 0; i--) {
        while (top!=len && mult(pnt[i], res[top],
res[top-1])) top--;
        res[++top] = pnt[i];
    }
    return top;       // 返回凸包中点的个数
}
/*==================================================*\
 | 判断线段相交
\*==================================================*/
const double eps=1e-10;
struct point { double x, y; };
double min(double a, double b) { return a < b ? a : b; }
double max(double a, double b) { return a > b ? a : b; }
bool inter(point a, point b, point c, point d){
    if ( min(a.x, b.x) > max(c.x, d.x) ||
         min(a.y, b.y) > max(c.y, d.y) ||
         min(c.x, d.x) > max(a.x, b.x) ||
         min(c.y, d.y) > max(a.y, b.y) ) return 0;
    double h, i, j, k;
    h = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    i = (b.x - a.x) * (d.y - a.y) - (b.y - a.y) * (d.x - a.x);
    j = (d.x - c.x) * (a.y - c.y) - (d.y - c.y) * (a.x - c.x);
    k = (d.x - c.x) * (b.y - c.y) - (d.y - c.y) * (b.x - c.x);
    return h * i <= eps && j * k <= eps;
}
/*==================================================*\
 | 求多边形重心
 | INIT: pnt[]已按顺时针(或逆时针)排好序;
 | CALL: res = bcenter(pnt, n);
\*==================================================*/
struct point { double x, y; };
point bcenter(point pnt[], int n){
    point p, s;
    double tp, area = 0, tpx = 0, tpy = 0;
    p.x = pnt[0].x; p.y = pnt[0].y;
    for (int i = 1; i <= n; ++i) {   // point: 0 ~ n-1
        s.x = pnt[(i == n) ? 0 : i].x;
        s.y = pnt[(i == n) ? 0 : i].y;
        tp = (p.x * s.y - s.x * p.y); area += tp / 2;
        tpx += (p.x + s.x) * tp; tpy += (p.y + s.y) * tp;
        p.x = s.x; p.y = s.y;
    }
    s.x = tpx / (6 * area); s.y = tpy / (6 * area);
    return s;
}
/*==================================================*\
 | 三角形几个重要的点
 | INIT: pnt[]已按顺时针(或逆时针)排好序;
 | CALL: res = bcenter(pnt, n);
\*==================================================*/
设三角形的三条边为a, b, c，且不妨假设a <= b <= c,
三角形的面积可以根据海伦公式算得，如下:
s = sqrt(p * (p - a) * (p - b) * (p - c));
p = (a + b + c) / 2;
下面是计算该点到三角形三个顶点A,B,C的距离之和

1. 费马点(该点到三角形三个顶点的距离之和最小)
   有个有趣的结论：若三角形的三个内角均小于120度，那么该点连接
三个顶点形成的三个角均为120度；若三角形存在一个内角大于120度，
则该顶点就是费马点)计算公式如下:
若有一个内角大于120度（这里假设为角C)，则距离为a + b
若三个内角均小于120度，则距离为
sqrt((a * a + b * b + c * c + 4 * sqrt(3.0) * s) / 2),其中

2. 内心----角平分线的交点
   令x = (a + b - c) / 2, y = (a - b + c) / 2,
     z = (-a + b + c) / 2, h = s / p. 计算公式为
sqrt(x*x + h*h) + sqrt(y*y + h*h) + sqrt(z * z + h * h)

3. 重心----中线的交点, 计算公式如下:
   2.0 / 3 * (sqrt((2 * (a * a + b * b) - c * c) / 4)
         + sqrt((2 * (a * a + c * c) - b * b) / 4)
         + sqrt((2 * (b * b + c * c) - a * a) / 4))

4. 垂心----垂线的交点, 计算公式如下:
   3 * (c / 2 / sqrt(1 - cosC * cosC))
/*==================================================*\
 | 平面最近点对 O(N * logN)
\*==================================================*/
const int N = 100005;
const double MAX = 10e100, eps = 0.00001;
struct Point { double x, y; int index; };
Point a[N], b[N], c[N];
double closest(Point *, Point *, Point *, int, int);
double dis(Point, Point);
int cmp_x(const void *, const void*);
int cmp_y(const void *, const void*);
int merge(Point *, Point *, int, int, int);
inline double min(double, double);
int main(){
    int n, i;
    double d;
    scanf("%d", &n);
    while (n) {
        for (i = 0; i < n; i++)
            scanf("%lf%lf", &(a[i].x), &(a[i].y));
        qsort(a, n, sizeof(a[0]), cmp_x);
```

```
        for (i = 0; i < n; i++)
            a[i].index = i;
        memcpy(b, a, n *sizeof(a[0]));
        qsort(b, n, sizeof(b[0]), cmp_y);
        d = closest(a, b, c, 0, n - 1);
        printf("%.2lf\n", d);
        scanf("%d", &n);
    }
    return 0;
}
double closest(Point a[],Point b[],Point c[],int p,int q){
    if (q - p == 1) return dis(a[p], a[q]);
    if (q - p == 2) {
        double x1 = dis(a[p], a[q]);
        double x2 = dis(a[p + 1], a[q]);
        double x3 = dis(a[p], a[p + 1]);
        if (x1 < x2 && x1 < x3) return x1;
        else if (x2 < x3) return x2;
        else return x3;
    }
    int i, j, k, m = (p + q) / 2;
    double d1, d2;
    for (i = p, j = p, k = m + 1; i <= q; i++)
        if (b[i].index <= m) c[j++] = b[i];
    //数组c左半部保存划分后左部的点, 且对y是有序的.
    else c[k++] = b[i];
    d1 = closest(a, c, b, p, m);
    d2 = closest(a, c, b, m + 1, q);
    double dm = min(d1, d2);
     //数组c左右部分分别是对y坐标有序的, 将其合并到b.
    merge(b, c, p, m, q);
    for (i = p, k = p; i <= q; i++)
        if (fabs(b[i].x - b[m].x) < dm) c[k++] = b[i];
    //找出离划分基准左右不超过dm的部分, 且仍然对y坐标有序.
    for (i = p; i < k; i++)
    for (j = i + 1; j < k && c[j].y - c[i].y < dm; j++){
        double temp = dis(c[i], c[j]);
        if (temp < dm) dm = temp;
    }
    return dm;
}
double dis(Point p, Point q){
    double x1 = p.x - q.x, y1 = p.y - q.y;
    return sqrt(x1 *x1 + y1 * y1);
}
int merge(Point p[], Point q[], int s, int m, int t){
    int i, j, k;
    for (i=s, j=m+1, k = s; i <= m && j <= t;) {
        if (q[i].y > q[j].y) p[k++] = q[j], j++;
        else p[k++] = q[i], i++;
    }
    while (i <= m) p[k++] = q[i++];
    while (j <= t) p[k++] = q[j++];
    memcpy(q + s, p + s, (t - s + 1) *sizeof(p[0]));
    return 0;
}
int cmp_x(const void *p, const void *q){
    double temp = ((Point*)p)->x - ((Point*)q)->x;
    if (temp > 0) return 1;
    else if (fabs(temp) < eps) return 0;
    else return - 1;
}
int cmp_y(const void *p, const void *q){
    double temp = ((Point*)p)->y - ((Point*)q)->y;
    if (temp > 0) return 1;
    else if (fabs(temp) < eps) return 0;
    else return - 1;
}
inline double min(double p, double q)
{
    return (p > q) ? (q): (p);
}
/*=======================================================*\
| Liuctic 的计算几何库
| p-Lpoint    ln,l - Lline ls - Llineseglr - Lrad
| 求平面上两点之间的距离                  p2pdis
| 返回(P1-P0)*(P2-P0)的叉积。               xmulti
| 确定两条线段是否相交                    lsinterls
| 判断点p是否在线段l上                    ponls
| 判断两个点是否相等                          Euqal_Point
| 线段非端点相交                                lsinterls_A
| 判断点q是否在多边形Polygon内                  pinplg
| 多边形的面积                          area_of_polygon
| 解二次方程                            Ax^2+Bx+C=0 equa
| 点到直线距离                          p2lndis
| 直线与圆的交点，已知直线与圆相交      lncrossc
| 点是否在射线的正向                      samedir
| 射线与圆的第一个交点                  lrcrossc
| 求点p1关于直线ln的对称点p2            mirror
| 两直线夹角（弧度）                      angle_LL
\*=======================================================*/
#define infinity 1e20
#define EP 1e-10
const int MAXV = 300 ;
const double PI = 2.0*asin(1.0);    //高精度求PI
struct Lpoint {double x,y;};        //点
struct Llineseg{ Lpoint a,b;};      //线段
struct Ldir{double dx,dy;};         //方向向量
struct Lline{Lpoint p; Ldir dir;}; //直线
struct Lrad{Lpoint Sp; Ldir dir;}; //射线
struct Lround{Lpoint co; double r;};//圆
| 求平面上两点之间的距离
double p2pdis(Lpoint p1,Lpoint p2) {
    return (sqrt((p1.x-p2.x) * (p1.x-p2.x) +
                 (p1.y-p2.y) * (p1.y-p2.y)));
}
/*******************************************************
|  (P1-P0)*(P2-P0)的叉积
 若结果为正，则<P0,P1>在<P0,P2>的顺时针方向；
 若为0则<P0,P1><P0,P2>共线；
 若为负则<P0,P1>在<P0,P2>的在逆时针方向；
 可以根据这个函数确定两条线段在交点处的转向，
 比如确定p0p1和p1p2在p1处是左转还是右转，只要求
 (p2-p0)*(p1-p0)，若<0则左转，>0则右转，=0则共线
*******************************************************/
double xmulti(Lpoint p1,Lpoint p2,Lpoint p0) {
    return((p1.x-p0.x) * (p2.y-p0.y) -
           (p2.x-p0.x) * (p1.y-p0.y));
}
| 确定两条线段是否相交
double mx(double t1,double t2)
{
    if(t1>t2) return t1;
    return t2;
}
double mn(double t1,double t2)
{
    if(t1<t2) return t1;
    return t2;
}
int lsinterls(Llineseg u,Llineseg v)
{
    return( (mx(u.a.x,u.b.x)>=mn(v.a.x,v.b.x))&&
            (mx(v.a.x,v.b.x)>=mn(u.a.x,u.b.x))&&
            (mx(u.a.y,u.b.y)>=mn(v.a.y,v.b.y))&&
            (mx(v.a.y,v.b.y)>=mn(u.a.y,u.b.y))&&

(xmulti(v.a,u.b,u.a)*xmulti(u.b,v.b,u.a)>=0)&&

(xmulti(u.a,v.b,v.a)*xmulti(v.b,u.b,v.a)>=0));
}
| 判断点 p 是否在线段 l 上
int ponls(Llineseg l,Lpoint p) {
    return( (xmulti(l.b,p,l.a)==0) &&
            ( ((p.x-l.a.x)*(p.x-l.b.x)<0 ) ||
              ((p.y-l.a.y)*(p.y-l.b.y)<0 )) );
}
| 判断两个点是否相等
int Euqal_Point(Lpoint p1,Lpoint p2) {
    return((fabs(p1.x-p2.x)<EP)&&(fabs(p1.y-p2.y)<EP));
}
| 线段相交判断函数
  当且仅当u,v相交并且交点不是u,v的端点时函数为true;
int lsinterls_A(Llineseg u,Llineseg v) {
    return((lsinterls(u,v)) && (!Euqal_Point(u.a,v.a))&&
           (!Euqal_Point(u.a,v.b)) &&
(!Euqal_Point(u.b,v.a))&&
           (!Euqal_Point(u.b,v.b)));
}
/*===============================================
```

```
|  判断点 q 是否在多边形内
   其中多边形是任意的凸或凹多边形,
   Polygon中存放多边形的逆时针顶点序列
==================================================*/
int pinplg(int vcount,Lpoint Polygon[],Lpoint q)
{
     int c=0,i,n;
     Llineseg l1,l2;
     l1.a=q; l1.b=q; l1.b.x=infinity; n=vcount;
     for (i=0;i<vcount;i++) {
          l2.a=Polygon[i];
          l2.b=Polygon[(i+1)%n];
          if ((lsinterls_A(l1,l2))||
              (
              (ponls(l1,Polygon[(i+1)%n]))&&
              (
              (!ponls(l1,Polygon[(i+2)%n]))&&
              (xmulti(Polygon[i],Polygon[(i+1)%n],l1.a) *

xmulti(Polygon[(i+1)%n],Polygon[(i+2)%n],l1.a)>0)
              ||
              (ponls(l1,Polygon[(i+2)%n]))&&
              (xmulti(Polygon[i],Polygon[(i+2)%n],l1.a) *

xmulti(Polygon[(i+2)%n],Polygon[(i+3)%n],l1.a)>0)
              ) ) ) c++;
     }
     return(c%2!=0);
}
/*==================================================*\
| 计算多边形的面积
| 要求按照逆时针方向输入多边形顶点
| 可以是凸多边形或凹多边形
\*==================================================*/
double areaofp(int vcount,double x[],double y[],Lpoint
plg[])
{
  int i;
  double s;
  if (vcount<3) return 0;
  s=plg[0].y*(plg[vcount-1].x-plg[1].x);
  for (i=1;i<vcount;i++)
     s+=plg[i].y*(plg[(i-1)].x-plg[(i+1)%vcount].x);
  return s/2;
}
/*********************\
| 解二次方程 Ax^2+Bx+C=0
返回-1表示无解 返回1 表示有解
\*********************/
int equa(double A,double B,double C,double& x1,double& x2)
{
    double f=B*B-4*A*C;
    if(f<0) return -1;
    x1=(-B+sqrt(f))/(2*A);
    x2=(-B-sqrt(f))/(2*A);
    return 1;
}
| 计算直线的一般式 Ax+By+C=0
void format(Lline ln,double& A,double& B,double& C)
{
    A=ln.dir.dy;
    B=-ln.dir.dx;
    C=ln.p.y*ln.dir.dx-ln.p.x*ln.dir.dy;
}
| 点到直线距离
double p2ldis(Lpoint a,Lline ln)
{
    double A,B,C;
    format(ln,A,B,C);
    return(fabs(A*a.x+B*a.y+C)/sqrt(A*A+B*B));
}
| 直线与圆的交点，已知直线与圆相交
int lncrossc(Lline ln,Lround Y,Lpoint& p1,Lpoint& p2)
{
    double A,B,C,t1,t2;
     int zz=-1;
    format(ln,A,B,C);
     if(fabs(B)<1e-8)
     {
          p1.x=p2.x=-1.0*C/A;
          zz=equa(1.0,-2.0*Y.co.y,Y.co.y*Y.co.y
            +(p1.x-Y.co.x)*(p1.x-Y.co.x)-Y.r*Y.r,t1,t2);
          p1.y=t1;p2.y=t2;
     }
     else if(fabs(A)<1e-8)
     {
          p1.y=p2.y=-1.0*C/B;
          zz=equa(1.0,-2.0*Y.co.x,Y.co.x*Y.co.x
         +(p1.y-Y.co.y)*(p1.y-Y.co.y)-Y.r*Y.r,t1,t2);
          p1.x=t1;p2.x=t2;
     }
     else
     {
          zz=equa(A*A+B*B,2.0*A*C+2.0*A*B*Y.co.y

-2.0*B*B*Y.co.x,B*B*Y.co.x*Y.co.x+C*C+2*B*C*Y.co.y
         +B*B*Y.co.y*Y.co.y-B*B*Y.r*Y.r,t1,t2);
          p1.x=t1,p1.y=-1*(A/B*t1+C/B);
          p2.x=t2,p2.y=-1*(A/B*t2+C/B);
     }
     return 0;
}
| 点是否在射线的正向
bool samedir(Lrad ln,Lpoint P)
{
    double ddx,ddy;
    ddx=P.x-ln.Sp.x;ddy=P.y-ln.Sp.y;
    if((ddx*ln.dir.dx>0||fabs(ddx*ln.dir.dx)<1e-7)
    &&(ddy*ln.dir.dy>0||(fabs(ddy*ln.dir.dy)<1e-7)))
    return true;
     else return false;
}
| 射线与圆的第一个交点
已经确定射线所在直线与圆相交返回-1表示不存正向交点 , 否则返回1
int lrcrossc(Lrad ln, Lround Y, Lpoint& P)
{
    Lline ln2;
    Lpoint p1,p2;
    int res=-1;
    double dis=1e20;
    ln2.p=ln.Sp,ln2.dir=ln.dir;
    lncrossc(ln2,Y,p1,p2);
    if(samedir(ln,p1))
    {
        res=1;
        if(p2pdis(p1,ln.Sp)<dis)
        {
            dis=p2pdis(p1,ln.Sp);
            P=p1;
        }
    }
    if(samedir(ln,p2))
    {
        res=1;
        if(p2pdis(p2,ln.Sp)<dis)
        {
            dis=p2pdis(p2,ln.Sp);
            P=p2;
        }
    }
    return res;
}
| 求点 p1 关于直线 ln 的对称点 p2
Lpoint mirror(Lpoint P,Lline ln)
{
    Lpoint Q;
    double A,B,C;
    format(ln,A,B,C);
    Q.x=((B*B-A*A)*P.x-2*A*B*P.y-2*A*C)/(A*A+B*B);
    Q.y=((A*A-B*B)*P.y-2*A*B*P.x-2*B*C)/(A*A+B*B);
    return Q;
}
| 两直线夹角（弧度）
double angle_LL(Lline line1, Lline line2)
{
   double A1, B1, C1;
   format(line1, A1, B1, C1);
   double A2, B2, C2;
   format(line2, A2, B2, C2);
   if( A1*A2+B1*B2 == 0 ) return PI/2.0; // 垂直
```

```
   else{
      double t = fabs((A1*B2-A2*B1)/(A1*A2+B1*B2));
      return atan(t);
    }
}
```

STL

全排列函数 next_permutation

STL 中专门用于排列的函数（可以处理存在重复数据集的排列问题）

头文件：#include <algorithm>
        using namespace std;

调用：next_permutation(start, end);

注意：函数要求输入的是一个升序排列的序列的头指针和尾指针.

用法：

```
// 数组
int a[N];
sort(a, a+N);
next_permutation(a, a+N);
// 向量
vector<int> ivec;
sort(ivec.begin(), ivec.end());
next_permutation(ivec.begin(), ivec.end());
```

例子：

```
vector<int> myVec;
// 初始化代码
sort(myVec.begin(),myVec.end());
do{
     for (i = 0 ;i < size;i ++ ) cout << myVec[i] << " \t " ;
     cout << endl;
}while (next_permutation(myVec.begin(), myVec.end()));
```

ACM/ICPC 竞赛之 STL 简介

一、关于 STL

STL(Standard Template Library，标准模板库）是 C++语言标准中的重要组成部分。STL 以模板类和模板函数的形式为程序员提供了各种数据结构和算法的精巧实现，程序员如果能够充分地利用 STL，可以在代码空间、执行时间和编码效率上获得极大的好处。

STL 大致可以分为三大类：算法(algorithm)、容器(container)、迭代器(iterator)。

STL 容器是一些模板类，提供了多种组织数据的常用方法，例如 vector(向量，类似于数组)、list(列表，类似于链表)、deque(双向队列)、set(集合)、map(映象)、stack(栈)、queue(队列)、priority_queue(优先队列)等，通过模板的参数我们可以指定容器中的元素类型。

STL 算法是一些模板函数，提供了相当多的有用算法和操作，从简单如 for_each（遍历）到复杂如 stable_sort（稳定排序）。

STL 迭代器是对 C 中的指针的一般化，用来将算法和容器联系起来。几乎所有的 STL 算法都是通过迭代器来存取元素序列进行工作的，而 STL 中的每一个容器也都定义了其本身所专有的迭代器，用以存取容器中的元素。有趣的是，普通的指针也可以像迭代器一样工作。

熟悉了 STL 后，你会发现，很多功能只需要用短短的几行就可以实现了。通过 STL，我们可以构造出优雅而且高效的代码，甚至比你自己手工实现的代码效果还要好。

STL 的另外一个特点是，它是以源码方式免费提供的，程序员不仅可以自由地使用这些代码，也可以学习其源码，甚至按照自己的需要去修改它。

下面是用 STL 写的题 Ugly Numbers 的代码：

```
#include <iostream>
#include <queue>
using namespace std;
typedef pair<unsigned long, int> node_type;
int main(){
 unsigned long result[1500];
 priority_queue< node_type, vector<node_type>,
greater<node_type> > Q;
 Q.push( make_pair(1, 2) );
 for (int i=0; i<1500; i++){
  node_type node = Q.top(); Q.pop();
  switch(node.second){
  case 2: Q.push( make_pair(node.first*2, 2) );
  case 3: Q.push( make_pair(node.first*3, 3) );
  case 5: Q.push( make_pair(node.first*5, 5) );
  }
  result[i] = node.first;
 }
 int n;
 cin >> n;
 while (n>0){
  cout << result[n-1] << endl;
  cin >> n;
 }
 return 0;
}
```

在 ACM 竞赛中，熟练掌握和运用 STL 对快速编写实现代码会有极大的帮助。

二、使用 STL

在 C++标准中，STL 被组织为以下的一组头文件（注意，是没有.h 后缀的！）：

algorithm / deque / functional / iterator / list / map
memory / numeric / queue / set / stack / utility / vector

当我们需要使用 STL 的某个功能时，需要嵌入相应的头文件。但要注意的是，在 C++标准中，STL 是被定义在 std 命名空间中的。如下例所示：

```
#include <stack>
int main(){
     std::stack<int> s;
     s.push(0);
     ...
     return 0;
}
```

如果希望在程序中直接引用 STL，也可以在嵌入头文件后，用 using namespace 语句将 std 命名空间导入。如下例所示：

```
#include <stack>
using namespace std;
int main(){
     stack<int> s;
     s.push(0);
     ...
     return 1;
}
```

STL 是 C++语言机制运用的一个典范，通过学习 STL 可以更深刻地理解 C++语言的思想和方法。在本系列的文章中不打算对 STL 做深入的剖析，而只是想介绍一些 STL 的基本应用。

有兴趣的同学，建议可以在有了一些 STL 的使用经验后，认真阅读一下《C++ STL》这本书（电力出版社有该书的中文版）。

ACM/ICPC 竞赛之 STL--pair

STL 的<utility>头文件中描述了一个看上去非常简单的模板类 pair，用来表示一个二元组或元素对，并提供了按照字典序对元素对进行大小比较的比较运算符模板函数。

例如，想要定义一个对象表示一个平面坐标点，则可以：

pair<double, double> p1;
cin >> p1.first >> p1.second;
pair 模板类需要两个参数：首元素的数据类型和尾元素的数据类型。pair 模板类对象有两个成员：first 和 second，分别表示首元素和尾元素。
在<utility>中已经定义了 pair 上的六个比较运算符:<、>、<=、>=、==、!=，其规则是先比较 first，first 相等时再比较 second，这符合大多数应用的逻辑。当然，也可以通过重载这几个运算符来重新指定自己的比较逻辑。
除了直接定义一个 pair 对象外，如果需要即时生成一个 pair 对象，也可以调用在<utility>中定义的一个模板函数：make_pair。make_pair 需要两个参数，分别为元素对的首元素和尾元素。
在题 1067--Ugly Numbers 中，就可以用 pair 来表示推演树上的结点，用 first 表示结点的值，用 second 表示结点是由父结点乘以哪一个因子得到的。

```
#include <iostream>
#include <queue>
using namespace std;
typedef pair<unsigned long, int> node_type;
int main(){
 unsigned long result[1500];
 priority_queue< node_type, vector<node_type>,
greater<node_type> > Q;
 Q.push( make_pair(1, 2) );
 for (int i=0; i<1500; i++){
  node_type node = Q.top(); Q.pop();
  switch(node.second){
  case 2: Q.push( make_pair(node.first*2, 2) );
  case 3: Q.push( make_pair(node.first*3, 3) );
  case 5: Q.push( make_pair(node.first*5, 5) );
  }
  result[i] = node.first;
 }
 int n;
 cin >> n;
 while (n>0){
  cout << result[n-1] << endl;
  cin >> n;
 }
 return 0;
}
```

<utility>看上去是很简单的一个头文件，但是<utility>的设计中却浓缩反映了 STL 设计的基本思想。有意深入了解和研究 STL 的同学，仔细阅读和体会这个简单的头文件，不失为一种入门的途径。

ACM/ICPC 竞赛之 STL--vector

在 STL 的<vector>头文件中定义了 vector（向量容器模板类），vector 容器以连续数组的方式存储元素序列，可以将 vector 看作是以顺序结构实现的线性表。当我们在程序中需要使用动态数组时，vector 将会是理想的选择，vector 可以在使用过程中动态地增长存储空间。
vector 模板类需要两个模板参数，第一个参数是存储元素的数据类型，第二个参数是存储分配器的类型，其中第二个参数是可选的，如果不给出第二个参数，将使用默认的分配器。
下面给出几个常用的定义 vector 向量对象的方法示例：

vector<int> s;
定义一个空的 vector 对象，存储的是 int 类型的元素。
vector<int> s(n);
定义一个含有 n 个 int 元素的 vector 对象。
vector<int> s(first, last);
定义一个 vector 对象，并从由迭代器 first 和 last 定义的序列[first, last)中复制初值。

vector 的基本操作有：
s[i]
直接以下标方式访问容器中的元素。
s.front()
返回首元素。
s.back()
返回尾元素。
s.push_back(x)
向表尾插入元素 x。
s.size()
返回表长。
s.empty()
当表空时，返回真，否则返回假。
s.pop_back()
删除表尾元素。
s.begin()
返回指向首元素的随机存取迭代器。
s.end()
返回指向尾元素的下一个位置的随机存取迭代器。
s.insert(it, x)
向迭代器 it 指向的元素前插入新元素 val。
s.insert(it, n, x)
向迭代器 it 指向的元素前插入 n 个 x。
s.insert(it, first, last)
将由迭代器 first 和 last 所指定的序列[first, last)插入到迭代器 it 指向的元素前面。
s.erase(it)
删除由迭代器 it 所指向的元素。
s.erase(first, last)
删除由迭代器 first 和 last 所指定的序列[first, last)。
s.reserve(n)
预分配缓冲空间，使存储空间至少可容纳 n 个元素。
s.resize(n)
改变序列的长度，超出的元素将会被删除，如果序列需要扩展（原空间小于 n），元素默认值将填满扩展出的空间。
s.resize(n, val)
改变序列的长度，超出的元素将会被删除，如果序列需要扩展（原空间小于 n），将用 val 填满扩展出的空间。
s.clear()
删除容器中的所有的元素。
s.swap(v)
将 s 与另一个 vector 对象 v 进行交换。
s.assign(first, last)
将序列替换成由迭代器 first 和 last 所指定的序列[first, last)。[first, last)不能是原序列中的一部分。
要注意的是，resize 操作和 clear 操作都是对表的有效元素进行的操作，但并不一定会改变缓冲空间的大小。
另外，vector 还有其他一些操作，如反转、取反等，不再一下列举。
vector 上还定义了序列之间的比较操作运算符(>, <, >=, <=, ==, !=)，可以按照字典序比较两个序列。
还是来看一些示例代码。输入个数不定的一组整数，再将这组整数按倒序输出，如下所示：

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
    vector<int> L;
    int x;
    while (cin>>x) L.push_back(x);
    for (int i=L.size()-1; i>=0; i--) cout << L[i] << " ";
    cout << endl;
    return 0;
}
```

ACM/ICPC 竞赛之 STL--iterator 简介

iterator(迭代器)是用于访问容器中元素的指示器，从这个意义上说，iterator(迭代器)相当于数据结构中所说的"遍历指针"，也可以把 iterator(迭代器)看作是一种泛化的指针。
STL 中关于 iterator(迭代器)的实现是相当复杂的，这里我们暂时不去详细讨论关于 iterator(迭代器)的实现和使用，而只对 iterator(迭代器)做一点简单的介绍。
简单地说，STL 中有以下几类 iterator(迭代器)：
输入 iterator(迭代器)，在容器的连续区间内向前移动，可以读取容器内任意值；
输出 iterator(迭代器)，把值写进它所指向的容器中；
前向 iterator(迭代器)，读取队列中的值，并可以向前移动到下一位置(++p,p++)；
双向 iterator(迭代器)，读取队列中的值，并可以向前向后遍历容器；
随机访问 iterator(迭代器)，可以直接以下标方式对容器进行访问，vector 的 iterator(迭代器)就是这种 iterator(迭代器)；
流 iterator(迭代器)，可以直接输出、输入流中的值；
每种 STL 容器都有自己的 iterator(迭代器)子类，下面先来看一段简单的示例代码：

```
#include <iostream>
#include <vector>
using namespace std;
main()
{
    vector<int> s;
    for (int i=0; i<10; i++) s.push_back(i);
    for (vector<int>::iterator it=s.begin(); it!=s.end();
it++)
        cout << *it << " ";
    cout << endl;
    return 1;
}
```

vector 的 begin()和 end()方法都会返回一个 vector::iterator 对象，分别指向 vector 的首元素位置和尾元素的下一个位置（我们可以称之为结束标志位置）。
对一个 iterator(迭代器)对象的使用与一个指针变量的使用极为相似，或者可以这样说，指针就是一个非常标准的 iterator(迭代器)。
再来看一段稍微特别一点的代码：

```
#include <iostream>
#include <vector>
```

```
using namespace std;
main()
{
    vector<int> s;
    s.push_back(1);
    s.push_back(2);
    s.push_back(3);
    copy(s.begin(), s.end(), ostream_iterator<int>(cout, " "));
    cout <<endl;
    return 1;
}
```

这段代码中的 copy 就是 STL 中定义的一个模板函数，copy(s.begin(), s.end(), ostream_iterator<int>(cout, " "));的意思是将由 s.begin()至 s.end()(不含 s.end())所指定的序列复制到标准输出流 cout 中，用" "作为每个元素的间隔。也就是说，这句话的作用其实就是将表中的所有内容依次输出。

iterator(迭代器)是 STL 容器和算法之间的"胶合剂"，几乎所有的 STL 算法都是通过容器的 iterator(迭代器)来访问容器内容的。只有通过有效地运用 iterator(迭代器)，才能够有效地运用 STL 强大的算法功能。

ACM/ICPC 竞赛之 STL--string

字符串是程序中经常要表达和处理的数据，我们通常是采用字符数组或字符指针来表示字符串。STL 为我们提供了另一种使用起来更为便捷的字符串的表达方式：string。string 类的定义在头文件<string>中。

string 类其实可以看作是一个字符的 vector，vector 上的各种操作都可以适用于 string，另外，string 类对象还支持字符串的拼合、转换等操作。

下面先来看一个简单的例子：

```
#include <iostream>
#include <string>
using namespace std;
int main(){
    string s = "Hello! ", name;
    cin >> name;
    s += name;
    s += '!';
    cout << s << endl;
    return 0;
}
```

再以题 1064--Parencoding 为例，看一段用 string 作为容器，实现由 P 代码还原括号字符串的示例代码片段：

```
int m;
cin >> m; // P 编码的长度
string str; // 用来存放还原出来的括号字符串
int leftpa = 0; // 记录已出现的左括号的总数
for (int j=0; j<m; j++){
    int p;
    cin >> p;
    for (int k=0; k<p-leftpa; k++) str += '(';
    str += ')';
    leftpa = p;
}
```

ACM/ICPC 竞赛之 STL--stack/queue

stack(栈)和 queue(队列)也是在程序设计中经常会用到的数据容器，STL 为我们提供了方便的 stack(栈)的 queue(队列)的实现。

准确地说，STL 中的 stack 和 queue 不同于 vector、list 等容器，而是对这些容器的重新包装。这里我们不去深入讨论 STL 的 stack 和 queue 的实现细节，而是来了解一些他们的基本使用。

1、stack

stack 模板类的定义在<stack>头文件中。

stack 模板类需要两个模板参数，一个是元素类型，一个容器类型，但只有元素类型是必要的，在不指定容器类型时，默认的容器类型为 deque。

定义 stack 对象的示例代码如下：

```
stack<int> s1;
stack<string> s2;
```

stack 的基本操作有：

入栈，如例：s.push(x);

出栈，如例：s.pop();注意，出栈操作只是删除栈顶元素，并不返回该元素。

访问栈顶，如例：s.top()

判断栈空，如例：s.empty()，当栈空时，返回 true。

访问栈中的元素个数，如例：s.size()

下面是用 string 和 stack 写的解题 1064--Parencoding 的程序。

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;
int main(){
    int n;
    cin >> n;
    for (int i=0; i<n; i++){
        int m;
        cin >> m;
        string str;
        int leftpa = 0;
        for (int j=0; j<m; j++)  // 读入 P 编码，构造括号字符串
        {
            int p;
            cin >> p;
            for (int k=0; k<p-leftpa; k++) str += '(';
            str += ')';
            leftpa = p;
        }
        stack<int> s;
        for (string::iterator it=str.begin(); it!=str.end(); it++) {  // 构造 M 编码
            if (*it=='(') s.push(1);
            else{
                int p = s.top(); s.pop();
                cout << p << " ";
                if (!s.empty()) s.top() += p;
            }
        }
        cout << endl;
    }
    return 0;
}
```

2、queue

queue 模板类的定义在<queue>头文件中。

与 stack 模板类很相似，queue 模板类也需要两个模板参数，一个是元素类型，一个容器类型，元素类型是必要的，容器类型是可选的，默认为 deque 类型。

定义 queue 对象的示例代码如下：

```
queue<int> q1;
queue<double> q2;
```

queue 的基本操作有：

入队，如例：q.push(x); 将 x 接到队列的末端。

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。

访问队首元素，如例：q.front()，即最早被压入队列的元素。

访问队尾元素，如例：q.back()，即最后被压入队列的元素。

判断队列空，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()

3、priority_queue

在<queue>头文件中，还定义了另一个非常有用的模板类 priority_queue(优先队列）。优先队列与队列的差别在于优先队列不是按照入队的顺序出队，而是按照队列中元素的优先权顺序出队(默认为大者优先，也可以通过指定算子来指定自己的优先顺序）。

priority_queue 模板类有三个模板参数，第一个是元素类型，第二个容器类型，第三个是比较算子。其中后两个都可以省略，默认容器为 vector，默认算子为 less，即小的往前排，大的往后排（出队时序列尾的元素出队）。

定义 priority_queue 对象的示例代码如下：

```
priority_queue<int> q1;
priority_queue< pair<int, int> > q2;  // 注意在两个尖括号之间一定要留空格。
priority_queue<int, vector<int>, greater<int> > q3; // 定义小的先出队
```

priority_queue 的基本操作与 queue 相同。

初学者在使用 priority_queue 时，最困难的可能就是如何定义比较算子了。

如果是基本数据类型，或已定义了比较运算符的类，可以直接用 STL 的 less 算子和 greater 算子——默认为使用 less 算子，即小的往前排，大的先出队。

如果要定义自己的比较算子，方法有多种，这里介绍其中的一种：重载比较运算符。优先队列试图将两个元素 x 和 y 代入比较运算符(对 less 算子，调用 x<y，对 greater 算子，调用 x>y)，若结果为真，则 x 排在 y 前面，y 将先于 x 出队，反之，则将 y 排在 x 前面，x 将先出队。

看下面这个简单的示例：

```
#include <iostream>
#include <queue>
using namespace std;
class T{
public:
    int x, y, z;
    T(int a, int b, int c):x(a), y(b), z(c){}
};
bool operator < (const T &t1, const T &t2){
    return t1.z < t2.z;  // 按照 z 的顺序来决定 t1 和 t2 的顺序
}
int main(){
    priority_queue<T> q;
    q.push(T(4,4,3));
    q.push(T(2,2,5));
    q.push(T(1,5,4));
```

```
    q.push(T(3,3,6));
    while (!q.empty()){
        T t = q.top(); q.pop();
        cout << t.x << " " << t.y << " " << t.z << endl;
    }
    return 0; }
```

输出结果为(注意是按照 z 的顺序从大到小出队的):

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

再看一个按照 z 的顺序从小到大出队的例子:

```
#include <iostream>
#include <queue>
using namespace std;
class T{
    public:
    int x, y, z;
    T(int a, int b, int c):x(a), y(b), z(c)
    {
    }
};
bool operator > (const T &t1, const T &t2){
    return t1.z > t2.z;
}
int main(){
    priority_queue<T, vector<T>, greater<T> > q;
    q.push(T(4,4,3));
    q.push(T(2,2,5));
    q.push(T(1,5,4));
    q.push(T(3,3,6));

    while (!q.empty()){
        T t = q.top(); q.pop();
        cout << t.x << " " << t.y << " " << t.z << endl;
    }
    return 0;
}
```

输出结果为:

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

如果我们把第一个例子中的比较运算符重载为:

```
bool operator < (const T &t1, const T &t2){
    return t1.z > t2.z;  // 按照 z 的顺序来决定 t1 和 t2 的顺序
}
```

则第一个例子的程序会得到和第二个例子的程序相同的输出结果。

再回顾一下用优先队列实现的题 1067--Ugly Numbers 的代码:

```
#include <iostream>
#include <queue>
using namespace std;
typedef pair<unsigned long int, int> node_type;
int main( int argc, char *argv[] ){
    unsigned long int result[1500];
    priority_queue< node_type, vector<node_type>,
greater<node_type> > Q;
    Q.push( make_pair(1, 3) );
    for (int i=0; i<1500; i++){
        node_type node = Q.top();
        Q.pop();
        switch(node.second){
        case 3: Q.push( make_pair(node.first*2, 3) );
        case 2: Q.push( make_pair(node.first*3, 2) );
        case 1: Q.push( make_pair(node.first*5, 1) );
        }
        result[i] = node.first;
    }
    int n;
    cin >> n;
    while (n>0){
        cout << result[n-1] << endl;
        cin >> n;
    }
    return 1;
}
```

ACM/ICPC 竞赛之 STL--map

在 STL 的头文件<map>中定义了模板类 map 和 multimap，用有序二叉树来存贮类型为 pair<const Key, T>的元素对序列。序列中的元素以 const Key 部分作为标识，map 中所有元素的 Key 值都必须是唯一的，multimap 则允许有重复的 Key 值。

可以将 map 看作是由 Key 标识元素的元素集合，这类容器也被称为"关联容器"，可以通过一个 Key 值来快速确定一个元素，因此非常适合于需要按照 Key 值查找元素的容器。

map 模板类需要四个模板参数，第一个是键值类型，第二个是元素类型，第三个是比较算子，第四个是分配器类型。其中键值类型和元素类型是必要的。

map 的基本操作有:

1、定义 map 对象，例如:

```
map<string, int> m;
```

2、向 map 中插入元素对，有多种方法，例如:

```
m[key] = value;
```

[key]操作是 map 很有特色的操作，如果在 map 中存在键值为 key 的元素对，则返回该元素对的值域部分，否则将会创建一个键值为 key 的元素对，值域为默认值。所以可以用该操作向 map 中插入元素对或修改已经存在的元素对的值域部分。

```
m.insert( make_pair(key, value) );
```

也可以直接调用 insert 方法插入元素对，insert 操作会返回一个 pair，当 map 中没有与 key 相匹配的键值时，其 first 是指向插入元素对的迭代器，其 second 为 true；若 map 中已经存在与 key 相等的键值时，其 first 是指向该元素对的迭代器，second 为 false。

3、查找元素对，例如:

```
int i = m[key];
```

要注意的是，当与该键值相匹配的元素对不存在时，会创建键值为 key 的元素对。

```
map<string, int>::iterator it = m.find(key);
```

如果 map 中存在与 key 相匹配的键值时，find 操作将返回指向该元素对的迭代器，否则，返回的迭代器等于 map 的 end()（参见 vector 中提到的 begin 和 end 操作）。

4、删除元素对，例如:

```
m.erase(key);
```

删除与指定 key 键值相匹配的元素对，并返回被删除的元素的个数。

```
m.erase(it);
```

删除由迭代器 it 所指定的元素对，并返回指向下一个元素对的迭代器。

看一段简单的示例代码:

```
#include<map>
#include<iostream>
using namespace std;
typedef map<int, string, less<int> > M_TYPE;
typedef M_TYPE::iterator M_IT;
typedef M_TYPE::const_iterator M_CIT;
int main(){
 M_TYPE MyTestMap;

 MyTestMap[3] = "No.3";
 MyTestMap[5] = "No.5";
 MyTestMap[1] = "No.1";
 MyTestMap[2] = "No.2";
 MyTestMap[4] = "No.4";

 M_IT it_stop = MyTestMap.find(2);

 cout << "MyTestMap[2] = " << it_stop->second << endl;
 it_stop->second = "No.2 After modification";
 cout << "MyTestMap[2] = " << it_stop->second << endl;

 cout << "Map contents : " << endl;
 for(M_CIT it = MyTestMap.begin(); it != MyTestMap.end();
it++){
  cout << it->second << endl;
 }
 return 0;
}
```

程序执行的输出结果为:

```
MyTestMap[2] = No.2
MyTestMap[2] = No.2 After modification
Map contents :
No.1
No.2 After modification
No.3
No.4
No.5
```

再看一段简单的示例代码:

```
#include <iostream>
#include <map>
using namespace std;
int main(){
    map<string, int> m;
    m["one"] = 1;
    m["two"] = 2;
    // 几种不同的 insert 调用方法
```

```
    m.insert(make_pair("three", 3));
    m.insert(map<string, int>::value_type("four", 4));
    m.insert(pair<string, int>("five", 5));

    string key;
    while (cin>>key){
        map<string, int>::iterator it = m.find(key);
        if (it==m.end()){
            cout << "No such key!" << endl;
        }
        else{
            cout << key << " is " << it->second << endl;
            cout << "Erased " << m.erase(key) << endl;
        }
    }
    return 0;
}
```

ACM/ICPC 竞赛之 STL--algorithm

<algorithm>无疑是 STL 中最大的一个头文件，它是由一大堆模板函数组成的。

下面列举出<algorithm>中的模板函数：

adjacent_find / binary_search / copy / copy_backward / count / count_if / equal / equal_range / fill / fill_n / find / find_end / find_first_of / find_if / for_each / generate / generate_n / includes / inplace_merge / iter_swap / lexicographical_compare / lower_bound / make_heap / max / max_element / merge / min / min_element / mismatch / next_permutation / nth_element / partial_sort / partial_sort_copy / partition / pop_heap / prev_permutation / push_heap / random_shuffle / remove / remove_copy / remove_copy_if / remove_if / replace / replace_copy / replace_copy_if / replace_if / reverse / reverse_copy / rotate / rotate_copy / search / search_n / set_difference / set_intersection / set_symmetric_difference / set_union / sort / sort_heap / stable_partition / stable_sort / swap / swap_ranges / transform / unique / unique_copy / upper_bound

如果详细叙述每一个模板函数的使用，足够写一本书的了。还是来看几个简单的示例程序吧。

示例程序之一，for_each 遍历容器：

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int Visit(int v) // 遍历算子函数
{
    cout << v << " ";
    return 1;
}
class MultInt // 定义遍历算子类
{
private:
    int factor;
public:
    MultInt(int f) : factor(f){}
    void operator()(int &elem) const{
        elem *= factor;
    }
};

int main(){
    vector<int> L;
    for (int i=0; i<10; i++) L.push_back(i);
    for_each(L.begin(), L.end(), Visit);
    cout << endl;
    for_each(L.begin(), L.end(), MultInt(2));
    for_each(L.begin(), L.end(), Visit);
    cout << endl;
    return 0;
}
```

程序的输出结果为：

```
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 10 12 14 16 18
```

示例程序之二，min_element/max_element，找出容器中的最小/最大值：

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

int main(){
    vector<int> L;
    for (int i=0; i<10; i++) L.push_back(i);
    vector<int>::iterator min_it = min_element(L.begin(),
L.end());
    vector<int>::iterator max_it = max_element(L.begin(),
L.end());
    cout << "Min is " << *min_it << endl;
    cout << "Max is " << *max_it << endl;
    return 1;
}
```

程序的输出结果为：

```
Min is 0
Max is 9
```

示例程序之三，sort 对容器进行排序：

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void Print(vector<int> &L){
    for (vector<int>::iterator it=L.begin(); it!=L.end();
it++)
        cout << *it << " ";
    cout << endl;
}
int main(){
    vector<int> L;
    for (int i=0; i<5; i++) L.push_back(i);
    for (int i=9; i>=5; i--) L.push_back(i);
    Print(L);
    sort(L.begin(), L.end());
    Print(L);
    sort(L.begin(), L.end(), greater<int>()); // 按降序排序
    Print(L);
    return 0;
}
```

程序的输出结果为：

```
0 1 2 3 4 9 8 7 6 5
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

示例程序之四，copy 在容器间复制元素：

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main()
{
    // 先初始化两个向量 v1 和 v2
    vector <int> v1, v2;
    for (int i=0; i<=5; i++) v1.push_back(10*i);
    for (int i=0; i<=10; i++) v2.push_back(3*i);

    cout << "v1 = ( " ;
    for (vector <int>::iterator it=v1.begin(); it!=v1.end();
it++)
        cout << *it << " ";
    cout << ")" << endl;

    cout << "v2 = ( " ;
    for (vector <int>::iterator it=v2.begin(); it!=v2.end();
it++)
        cout << *it << " ";
    cout << ")" << endl;

    // 将 v1 的前三个元素复制到 v2 的中间
    copy(v1.begin(), v1.begin()+3, v2.begin()+4);

    cout << "v2 with v1 insert = ( " ;
    for (vector <int>::iterator it=v2.begin(); it!=v2.end();
it++)
        cout << *it << " ";
    cout << ")" << endl;

    // 在 v2 内部进行复制，注意参数 2 表示结束位置，结束位置不参与复
制
    copy(v2.begin()+4, v2.begin()+7, v2.begin()+2);

    cout << "v2 with shifted insert = ( " ;
    for (vector <int>::iterator it=v2.begin(); it!=v2.end();
it++)
        cout << *it << " ";
    cout << ")" << endl;
```

```
    return 1;
}
```
程序的输出结果为：
```
v1 = ( 0 10 20 30 40 50 )
v2 = ( 0 3 6 9 12 15 18 21 24 27 30 )
v2 with v1 insert = ( 0 3 6 9 0 10 20 21 24 27 30 )
v2 with shifted insert = ( 0 3 0 10 20 10 20 21 24 27 30 )
```
STL in ACM

容器(container)：

迭代器(iterator)：指针

内部实现：数组 // 就是没有固定大小的数组，vector 直接翻译是向量

vector // T 就是数据类型，Alloc 是关于内存的一个什么东西，一般是使用默认参数。

支持操作：

begin()，//取首个元素，返回一个 iterator

end()，//取末尾（最后一个元素的下一个存储空间的地址）

size()，//就是数组大小的意思

clear()，//清空

empty()，//判断 vector 是否为空

[ ] //很神奇的东东，可以和数组一样操作

//举例： vector a;   //定义了一个 vector

//然后我们就可以用 a[i]来直接访问 a 中的第 i + 1 个元素！和数组的下标一模一样！

push_back()，pop_back() //从末尾插入或弹出

insert() O(N) //插入元素，O(n)的复杂度

erase() O(N) //删除某个元素，O(n)的复杂度

可以用于数组大小不定且空间紧张的情况

Iterator 用法举例：
```
int main(){
 int n,i;
 vector vi; //类似于我们定义一个数组，同 int vi[1000]; 但 vector的大小是自动调整的
 vector ::iterator itr;  //两个冒号
 while (scanf("%d",&n) != EOF) vi.push_back(n);
 for (i = 0 ; i < vi.size() ; i++) printf("%d\n",vi[i]);
 for (itr = vi.begin() ; itr != vi.end() ; itr++)
  printf("%d\n",*itr);
 return 0;
}
```
类似：双端队列，两头都支持进出

支持 push_front()和 pop_front()

是的精简版:) //栈，只支持从末尾进出

支持 push()，pop()，top()

是的精简版 //单端队列，就是我们平时所说的队列，一头进，另一头出

支持 push()，pop()，front()，back()

内部实现：双向链表 //作用和 vector 差不多，但内部是用链表实现

list

支持操作：

begin()，end()，size()，clear()，empty()

push_back()，pop_back() //从末尾插入或删除元素

push_front()，pop_front()

insert() O(1) //链表实现，所以插入和删除的复杂度的 O(1)

erase() O(1)

sort() O(nlogn)，不同于中的 sort

//不支持[ ]操作！

内部实现：红黑树 //Red-Black Tree，一种平衡的二叉排序树

set //又是一个 Compare 函数，类似于 qsort 函数里的那个 Compare 函数，作为红黑树在内部实现的比较方式

insert() O(logn)

erase() O(logn)

find() O(logn) 找不到返回 a.end()

lower_bound() O(logn) 查找第一个不小于 k 的元素

upper_bound() O(logn) 查找第一个大于 k 的元素

equal_range() O(logn) 返回 pair

允许重复元素的

的用法及 Compare 函数示例：
```
struct SS {int x,y;};
struct ltstr {
 bool operator() (SS a, SS b)
 {return a.x < b.x;}  //注意，按 C 语言习惯，double 型要写成这样：return a.x < b.x ? 1 : 0;
};
int main() {
 set st;
 …
}
```
内部实现：pair 组成的红黑树 //map 中文意思：映射！！

map //就是很多 pair 组成一个红黑树

insert() O(logn)

erase() O(logn)

find() O(logn) 找不到返回 a.end()

lower_bound() O(logn) 查找第一个不小于 k 的元素

upper_bound() O(logn) 查找第一个大于 k 的元素

equal_range() O(logn) 返回 pair

[key]运算符 O(logn) *** //这个..太猛了，怎么说呢，数组有一个下标，如 a[i]，这里 i 是 int 型的。数组可以认为是从 int 印射到另一个类型的印射，而 map 是一个任意的印射，所以 i 可以是任何类型的！

允许重复元素，没有[]运算符

内部实现：堆 //优先队列，听 RoBa 讲得，似乎知道原理了，但不明白干什么用的

priority_queue

支持操作：

push() O(n)

pop() O(n)

top() O(1)

See also: push_heap()，pop_heap() … in

用法举例：
```
priority_queue maxheap; //int 最大堆
struct ltstr {   //又是这么个 Compare 函数，重载运算符？？？不明白为什么要这么写...反正这个 Compare 函数对我来说是相当之神奇。RoBa 说了，照着这么写就是了。
 bool operator()(int a,int b)
 {return a > b;}
};
priority_queue <INT,VECTOR,ltstr> minheap; //int 最小堆
```

1.sort()

void sort(RandomAccessIterator first, RandomAccessIterator last);

void sort(RandomAccessIterator first, RandomAccessIterator last, StrictWeakOrdering comp);

区间[first,last)

Quicksort，复杂度 O(nlogn)

(n=last-first，平均情况和最坏情况)

用法举例：

1.从小到大排序(int, double, char, string, etc)
```
const int N = 5;
int main()
{
 int a[N] = {4,3,2,6,1};
 string str[N] = {"TJU","ACM","ICPC","abc","kkkkk"};
 sort(a,a+N);
 sort(str,str+N);
 return 0;
}
```
2.从大到小排序（需要自己写 comp 函数)
```
const int N = 5;
int cmp(int a,int b) {return a > b;}
int main()
{
 int a[N] = {4,3,2,6,1};
 sort(a,a+N,cmp);
 return 0;
}
```
3. 对结构体排序
```
struct SS {int first,second;};
int cmp(SS a,SS b) {
 if (a.first != b.first) return a.first < b.first;
 return a.second < b.second;
}
```

v.s. qsort() in C (平均情况 O(nlogn)，最坏情况 O(n^2))   //qsort 中的 cmp 函数写起来就麻烦多了！
```
int cmp(const void *a,const void *b) {
 if (((SS*)a)->first != ((SS*)b)->first)
  return ((SS*)a)->first – ((SS*)b)->first;
 return ((SS*)a)->second – ((SS*)b)->second;
}
qsort(array,n,sizeof(array[0]),cmp);
```
sort()系列：
```
stable_sort(first,last,cmp); //稳定排序
partial_sort(first,middle,last,cmp);//部分排序
```
将前(middle-first)个元素放在[first,middle)中，其余元素位置不定

e.g.
```
int A[12] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
partial_sort(A, A + 5, A + 12);
```

```
// 结果是 "1 2 3 4 5 11 12 10 9 8 7 6".
Detail: Heapsort ,
O((last-first)*log(middle-first))
sort()系列:
partial_sort_copy(first, last, result_first, result_last,
cmp);
//输入到另一个容器，不破坏原有序列
bool is_sorted(first, last, cmp);
//判断是否已经有序
nth_element(first, nth, last, cmp);
//使[first,nth)的元素不大于[nth,last), O(N)
e.g. input: 7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5
nth_element(A,A+6,A+12);
Output: 5 2 6 1 4 3 7 8 9 10 11 12

2. binary_search()
bool binary_search(ForwardIterator first, ForwardIterator
last, const LessThanComparable& value);
bool binary_search(ForwardIterator first, ForwardIterator
last, const T& value, StrictWeakOrdering comp);
在[first,last)中查找 value，如果找到返回 Ture,否则返回 False
二分检索，复杂度 O(log(last-first))
v.s. bsearch() in C
Binary_search()系列
itr upper_bound(first, last, value, cmp);
//itr 指向大于 value 的第一个值(或容器末尾)
itr lower_bound(first, last, value, cmp);
//itr 指向不小于 valude 的第一个值(或容器末尾)
pair equal_range(first, last, value, cmp);
//找出等于 value 的值的范围 O(2*log(last - first))
int A[N] = {1,2,3,3,3,5,8}
*upper_bound(A,A+N,3) == 5
*lower_bound(A,A+N,3) == 3

make_heap(first,last,cmp) O(n)
push_heap(first,last,cmp)  O(logn)
pop_heap(first,last,cmp)  O(logn)
is_heap(first,last,cmp)  O(n)
e.g:
vector vi;
while (scanf("%d",&n) != EOF) {
 vi.push_back(n);
 push_heap(vi.begin(),vi.end());
}

Others interesting:
next_permutation(first, last, cmp)
prev_permutation(first, last, cmp)
//both O(N)
min(a,b);
max(a,b);
min_element(first, last, cmp);
max_element(first, last, cmp);

Others interesting:
fill(first, last, value)
reverse(first, last)
rotate(first,middle,last);
itr unique(first, last);
//返回指针指向合并后的末尾处
random_shuffle(first, last, rand)
```

头文件

```
#include <vector>
#include <list>
#include <map>
#include <set>
#include <deque>
#include <stack>
#include <bitset>
#include <algorithm>
#include <functional>
#include <numeric>
#include <utility>
#include <sstream>
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <ctime>

using namespace std;
```

线段树

求矩形并的面积（线段树+离散化+扫描线）

```
/* pku1151-Atlantis
Each test case starts with a line containing a single integer
n (1 <= n <= 100)
of available maps. The n following lines describe one map each.
Each of these lines
contains four numbers x1;y1;x2;y2 (0 <= x1 < x2 <= 100000;0
<= y1 < y2 <= 100000),
not necessarily integers. The values (x1; y1) and (x2;y2) are
the coordinates of the
top-left resp. bottom-right corner of the mapped area.
*/
/*
本题与 poj 1177 picture 极相似,现在回想起来甚至比 1177 还要简单一
些.与 1177 不同的是
本题中的坐标是浮点类型的, 故不能将坐标直接离散.我们必须为它们建立一
个对应关系,用一个
整数去对应一个浮点数这样的对应关系在本题的数组 y[] 中
*/
#include<iostream>
#include<algorithm>
#include<cmath>
#include<iomanip>
using namespace std;

struct node
{
    int st, ed, c;   //c : 区间被覆盖的层数, m: 区间的测度
    double m;
}ST[802];

struct line
{
    double x,y1,y2;   //纵方向直线, x:直线横坐标, y1 y2:直线上
的下面与上面的两个纵坐标
    bool s;          //s = 1 : 直线为矩形的左边, s = 0:直线为
矩形的右边
}Line[205];
double y[205],ty[205]; //y[] 整数与浮点数的对应数组; ty[]:用来
求 y[]的辅助数组

void build(int root, int st, int ed)
{
    ST[root].st = st;
    ST[root].ed = ed;
    ST[root].c = 0;
    ST[root].m = 0;
    if(ed - st > 1){
        int mid = (st+ed)/2;
        build(root*2, st, mid);
        build(root*2+1, mid, ed);
    }
}
inline void updata(int root){
    if(ST[root].c > 0)
        //将线段树上区间的端点分别映射到 y[]数组所对应的浮点数上,
由此计算出测度
        ST[root].m = y[ST[root].ed-1] - y[ST[root].st-1];
    else if(ST[root].ed - ST[root].st == 1)
        ST[root].m = 0;
    else ST[root].m = ST[root*2].m + ST[root*2+1].m;
}
void insert(int root, int st, int ed){
```

```
    if(st <= ST[root].st && ST[root].ed <= ed){
        ST[root].c++;
        updata(root);
        return ;
    }
    if(ST[root].ed - ST[root].st == 1)return ;//不出错的话这句话就是冗余的
    int mid = (ST[root].ed + ST[root].st)/2;
    if(st < mid) insert(root*2, st, ed);
    if(ed > mid) insert(root*2+1, st, ed);
    updata(root);
}
void Delete(int root, int st, int ed){
    if(st <= ST[root].st && ST[root].ed <= ed){
        ST[root].c--; updata(root);
        return ;
    }
    if(ST[root].ed - ST[root].st == 1)return ; //不出错的话这句话就是冗余的
    int mid = (ST[root].st + ST[root].ed)/2;
    if(st < mid) Delete(root*2, st, ed);
    if(ed > mid) Delete(root*2+1, st, ed);
    updata(root);
}
int Correspond(int n, double t){
    //二分查找出浮点数 t 在数组 y[]中的位置(此即所谓的映射关系)
    int low,high,mid;
    low = 0; high = n-1;
    while(low < high){
        mid = (low+high)/2;
        if(t > y[mid])
            low = mid + 1;
        else high = mid;
    }
    return high+1;
}
bool cmp(line l1, line l2){
    return l1.x < l2.x;
}

int main()
{
    int n,i,num,l,r,c=0;
    double area,x1,x2,y1,y2;
    while(cin>>n, n){
        for(i = 0; i < n; i++){
            cin>>x1>>y1>>x2>>y2;
            Line[2*i].x = x1; Line[2*i].y1 = y1;
Line[2*i].y2 = y2; Line[2*i].s = 1;
            Line[2*i+1].x = x2; Line[2*i+1].y1 = y1;
Line[2*i+1].y2 = y2; Line[2*i+1].s = 0;
            ty[2*i] = y1; ty[2*i+1] = y2;
        }
        n <<= 1;
        sort(Line, Line+n, cmp);
        sort(ty, ty+n);
        y[0] = ty[0];
       //处理数组 ty[]使之不含重覆元素,得到新的数组存放到数组 y[]中
        for(i=num=1; i < n; i++)
            if(ty[i] != ty[i-1])
                y[num++] = ty[i];
            build(1, 1, num); //树的叶子节点与数组 y[]中的元素个数相同,以便建立一一对应的关系
            area = 0;
            for(i = 0; i < n-1; i++){
                //由对应关系计算出线段两端在树中的位置
                l = Correspond(num, Line[i].y1);
                r = Correspond(num, Line[i].y2);
                if(Line[i].s) //插入矩形的左边
                    insert(1, l, r);
                else    //删除矩形的右边
                    Delete(1, l, r);
                area += ST[1].m * (Line[i+1].x - Line[i].x);
            }
            cout<<"Test case #"<<++c<<endl<<"Total explored area: ";

    cout<<fixed<<setprecision(2)<<area<<endl<<endl;
    }
    return 0;
}
```

求矩形并的周长（线段树+离散化+扫描线）

```
/* pku1177-picture
The first line contains the number of rectangles pasted on the wall. In each of the subsequent lines, one can find the integer coordinates of the lower left vertex and the upper right vertex of each rectangle. The values of those coordinates are given as ordered pairs consisting of an x-coordinate followed by a y-coordinate. 0 <= number of rectangles < 5000  All coordinates are in the range [-10000,10000] and any existing rectangle has a positive area.
*/
#include<iostream>
#include<algorithm>
using namespace std;
struct node{
      int st,ed,m,lbd,rbd;
      int sequence_line,count;
}ST[40005];
void build(int st, int ed, int v){          //建树，区间为 [st, ed]
      ST[v].st = st; ST[v].ed = ed;
      ST[v].m = ST[v].lbd = ST[v].rbd = 0;
      ST[v].sequence_line = ST[v].count = 0;
      if(ed - st > 1){
            int mid = (st+ed)/2;
            build(st, mid, 2*v+1);
            build(mid, ed, 2*v+2);
      }
}

inline void UpData(int v){                              //更新结点区间的测度
      if(ST[v].count > 0){
            ST[v].m = ST[v].ed - ST[v].st;
            ST[v].lbd = ST[v].rbd = 1;
            ST[v].sequence_line = 1;
            return;
      }
      if(ST[v].ed - ST[v].st == 1){
            ST[v].m = 0;
            ST[v].lbd = ST[v].rbd = 0;
            ST[v].sequence_line = 0;
      }
      else {
            int left = 2*v+1, right = 2*v+2;
            ST[v].m = ST[left].m + ST[right].m;
            ST[v].sequence_line = ST[left].sequence_line + ST[right].sequence_line - (ST[left].rbd & ST[right].lbd);
            ST[v].lbd = ST[left].lbd;
            ST[v].rbd = ST[right].rbd;
      }
}

void insert(int st, int ed, int v){
      if(st <= ST[v].st && ed >= ST[v].ed){
            ST[v].count++;
            UpData(v);
            return ;
      }
      int mid = (ST[v].st + ST[v].ed)/2;
      if(st < mid)insert(st, ed, 2*v+1);
      if(ed > mid)insert(st, ed, 2*v+2);
      UpData(v);
}

void Delete(int st, int ed, int v){
      if(st <= ST[v].st && ed >= ST[v].ed){
            ST[v].count--;
            UpData(v);
            return;
      }
      int mid = (ST[v].st + ST[v].ed)/2;
      if(st < mid)Delete(st, ed, 2*v+1);
      if(ed > mid)Delete(st, ed, 2*v+2);
      UpData(v);
}
```

```
struct line{
	int x,y1,y2;//y1 < y2
	bool d;		//d=true 表示该线段为矩形左边, d=false
表示该线段为矩形的右边
}a[10003];
bool cmp(line t1, line t2){		//为线段排序的函数,方便从
左向右的扫描
	return t1.x < t2.x;
}
void cal_C(int n);
int main()
{
	int n,x1,x2,y1,y2,i,j,suby, upy;
	while(scanf("%d",&n) != EOF){
		j = 0;
		suby = 10000; upy = -10000;
		for(i = 0; i < n; i++){
			scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
			a[j].x = x1;	a[j].y1 = y1; a[j].y2 = y2;
a[j].d = 1;
			j++;
			a[j].x = x2;	a[j].y1 = y1; a[j].y2 = y2;
a[j].d = 0;
			j++;
			if(suby > y1)suby = y1;
			if(upy < y2)upy = y2;
		}
		sort(a, a+j, cmp);
		build(suby,upy,0);
		cal_C(j);
	}
	return 0;
}

void cal_C(int n){
	int i,j,k,t2,sum=0;
	t2 = 0;
	a[n] = a[n-1];
	for(i = 0; i < n; i++){
		if(a[i].d == 1) insert(a[i].y1, a[i].y2, 0);
		else  Delete(a[i].y1, a[i].y2, 0);
		sum += ST[0].sequence_line * (a[i+1].x-a[i].x)
* 2;
		sum += abs(ST[0].m - t2);
		t2 = ST[0].m;
	}
	printf("%d\n",sum);
}
```