// ========== LIS ==========

```
int C[maxn], B[maxn], dp[maxn];//C 是辅助阵列
int LIS(int n){//n 表示作 LIS 的数据范围
  int mmin, mmax, ret=0;
  memset(C, 0x3f, sizeof(C));//inf=0x3f3f3f3f
  for(int i=1; i<=n; i++){//阵列从 1 开始
    mmin=1, mmax=i;
    while(mmin<mmax){//二分
      int mid=(mmin+mmax)/2;
      if(C[mid]<B[i]) mmin=mid+1;
      else mmax=mid;
    }
    dp[i]=mmin;
    C[mmin]=B[i];
    ret=max(ret, dp[i]);//更新答案
  }
  return ret;
}
```

// ========== LIS RUJIA=======

```
// A[1] ~ A[N]
for (int i=2;i<=N+1;i++) g[i]=INF;
for (int i=1;i<=N;i++) {
int k=lower_bound(g+2,g+N+2,A[i]) - g;
// non-decreasing: 用 upper_bound
dp[i] = k-1; //  以 A[i]结尾的 LIS 长度
Ans = max(Ans, dp[i]);
g[k] = A[i];
}
```

// ========== 四边形优化 ==========

dp(i,j)=min{dp(i,k-1),dp(k,j)}+w(i,j)(i≤k≤j)
(min 也可以改为 max) 如果满足：
1.区间包含的单调性: if (i≤i'<j≤j') then w(i',j)≤w(i,j')
2. 四边形不等式: if (i≤i'<j≤j') then w(i,j)+w(i',j')≤w(i',j)+w(i,j')
// 再定义 s(i,j) 表示 m(i,j) 取得最优值时对应的下标(即 i≤k≤j 时,k 处的 w 值最大,则 s(i,j)=k) 我们有 s(i,j)单调 ie.
s(i,j-1)≤s(i,j)≤s(i+1,j)
故而 dp(i,j)=min{dp(i,k-1)+dp(k,j)}+w(i,j) 且 s(i,j-1)≤k≤s(i+1,j) (min 可以改为 max)

// ========== 状态压缩 ==========

```
// O(3^n)
for (int S = 1; S < (1<<n); S++)
for (int S0 = (S-1)&S; S0; S0 = (S0-1)&S)
int S1 = S-S0; //S 被拆成两个集合 S0 & S1
```

// ========== 统计逆序对 ==========

```
int cnt=0; // 逆序对个数
int a[MaxN], c[MaxN];
void MergeSort(int l, int r) {
// if a[1]~a[N] then r = N+1
    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])
            // 如果算上(x,x)则改成 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];
    }
}int main() { a[1] = 4; a[2] = 2; a[3] = 1;
MergeSort(1,4); cout << cnt;}
```

// ========== Min - Max ==========

```
struct State{ //状态
int score; //当前状态的打分, 对大分玩家越有利分越高
inline bool isFinal();//判断是否为终局的函数
    inline void expand(int player,
    vector<State>& children);
    //拓展子节点的函数
};
int alphabeta(State& s, int player, int alpha, int beta){
//0 是大分玩家, 1 是小分玩家
    if(s.isFinal()) return s.score;
    vector<State> children;
    s.expand(player, children);//生成儿子
    for(auto child:children){
        int v=alphabeta(child, player^1, alpha, beta);//对儿子状态打分
        !player ?alpha = max(alpha, v)
        :beta=min(beta, v);
        //选取最有利的情况
        if(beta<=alpha) break;
        //alpha-beta 剪枝
    }
    return !player?alpha:beta;
}
```

// ========== STL example =========

```
#include <set>
//差别在与 set 中不允许有重复元素，multiset 中允许有重复元素。
int main() {
multiset<int> myset;
myset.clear();
printf("%d\n", myset.empty());
for (int i=10; i; i--)
    myset.insert(i*10);
// 10 20 30 40 50 60 70 80 90
multiset<int>::iterator itlow, itup, it;
itlow=myset.lower_bound (30);
itup=myset.upper_bound (60);
myset.erase(itlow,itup);// 10 20 70 80 90
// map<int,int>::iterator it
// cout >> it->first >> it->second
printf("size == %d\n", (int)myset.size());
myset.erase(10);
//20 70 80 90
it = myset.find(70);

printf("count == %d\n", (int)myset.count(80)); //返回容器中元素等于 key 的元素的个数
}
```

// ========== DSU 并查集 ==========

```
int p[maxn], Rank[maxn];
//p 记录祖先, Rank 记录秩
void init(int n){
    for(int i=1; i<=n; i++)
        p[i]=i, Rank[i]=0;
}
int Find(int x){//路径压缩找祖先
return p[x]==x?x:p[x]=Find(p[x]); }
void Union(int x, int y){
    int xr=Find(x), yr=Find(y);
    if(xr==yr) return;
    //如果祖先相同直接退出
    if(Rank[xr]>Rank[yr]) p[yr]=xr;
    //启发式合并
    else{
        p[xr]=yr;
        if(Rank[xr]==Rank[yr]) Rank[yr]++;
    }
}
```

// ========== RMQ ==========

```
// d[i][j]: 从 i 位开始 长度为 2^j 的一段元素
// 所有 max 直接改为 min 也可以直接用
void RMQ_init(const vector<int>& A) {
  for(int i = 0; i < A.size(); i++)
    d_max[i][0] = A[i];
  for (int j=1; (1<<j) <= n; j++)
  for (int i=0; i+(1<<j)-1 < n; i++)
      d_max[i][j]=max( d_max[i][j-1],
      d_max[i+(1<<(j-1))][j-1] );
}
int RMQ_Min(int L, int R) {
   int k = 0;
   while((1<<(k+1)) <= R-L+1) k++;
    return max( d_max[L][k],
    d_max[R-(1<<k)+1][k] );
}
```

// ========== 莫队算法 ==========

```
莫队 （不带区间修改）
// 左端点所在分块作为第一关键字 右端点大小作为第二关键字
struct Cmd {   int l, r, id;
friend bool operator < (const Cmd &a, const Cmd &b)     {
if (belong[a.l] == belong[b.l])
return a.r < b.r;
else return belong[a.l] < belong[b.l];   }
} cmd[maxm];
int ans[maxm], belong[maxn];
int cnt[maxk];  // cnt[i] = j 表示当前区间内有 j 个颜色为 i 的东西
inline void upd(int &now, int pos, int v) {   // 更新 now
    // 维护 now -= cnt[pos];
    //        cnt[pos] += v;
    //        now += cnt[pos];          }
inline void solve(void)   {
int L=1,R=0; //[L,R]为当前维护好的区间
int now = 0;  // now 为当前区间的答案
for (int i = 1; i <= M; i++)   {
for (; L < cmd[i].l; L++) upd(now, L, -1);
for (; R > cmd[i].r; R--) upd(now, R, -1);
for (; L > cmd[i].l; L--) upd(now, L - 1, 1);
for (; R < cmd[i].r; R++) upd(now, R + 1, 1);
    if (cmd[i].l == cmd[i].r) {
        ans[cmd[i].id] =…; continue;  }
ans[cmd[i].id] = now;
}  } // end of solve()
int main() {
int blocksize = sqrt(N);
for (int i = 1; i <= N; i++)   // [1, N]
    belong[i] = (i - 1) / blocksize + 1;
for (int i = 1; i <= M; i++) {
    read(cmd[i].l), read(cmd[i].r);
    cmd[i].id = i;   }
sort(cmd + 1, cmd + M + 1); solve();
for (int i = 1; i <= M; i++)
        printf("%d\n", ans[i]);
}
```

// ========= 树状数组 =========

```
int n,m, bit[600005]; // size == maxn
```

```
int lowbit(int u){return u&(-u);}
//最后一位 1 在的地方
void edit(int u,int v) { //a[u]的值增加 v
  for(int j=u;j<=n;j+=lowbit(j))
    bit[j]+=v;
}
int query(int p) { //区间和 a[1]+...+a[n]
  int ans=0,i;
  for(i=p;i>0;i-=lowbit(i))
    ans+=bit[i];
  return ans;
}
// a[1~n]
int main() {
  for(i=1;i<=n;i++) {
    scanf("%d",&val);
    edit(i,val);
  }
  for(i=1;i<=m;i++) {
    scanf("%d%d%d",&t,&a,&b);
    if(t==1)//单点修改
      edit(a, b);
    if(t==2)//区间查询[]
    printf("%d\n", query(b)-query(a-1));
  }
  return 0;
}
```

// ======== STL 名次树 ========

```
vector<int> tree;
int find(int x) { // x 的排名
   return lower_bound(tree.begin(),tree.end(),x)
   -tree.begin()+1;
}
int main() {
    scanf("%d", &n);
    tree.reserve(maxn);
    for (int i=1; i<=n; i++) {
        scanf("%d%d", &opt, &x);
        switch(opt) {
        case 1:
tree.insert(upper_bound(tree.begin(),
tree.end(),x),x); break;
        case 2:
        tree.erase(lower_bound(tree.begin(),tree.e
    nd(),x)); break;
        case 3:printf("%d\n",find(x));break;
         case 4: // 输出排名为 x 的数
         printf("%d\n",tree[x-1]);break;
         case 5: // 找 x 的前驱
printf("%d\n",
*--lower_bound(tree.begin(),tree.end(),x))
break;
         case 6: // 后继
         printf("%d\n",*upper_bound(tree.begin(),
tree.end(),x));break;
        }
    }
}
```

// ========= 替罪羊树 =========

```
namespace Scapegoat_Tree {
#define MAXN (100000 + 10)
  const double alpha = 0.75;
  struct Node {
    Node * ch[2]; //ch[0]=left, ch[1]=right
    int key, size, cover; // size 为有效节点的数
量, cover 为节点总数量
    bool exist;   // 是否存在（是否被删除,不是
真正删除 只 invalid）
    void PushUp() {
      size = ch[0]->size + ch[1]->size + (int)exist;
      cover = ch[0]->cover + ch[1]->cover + 1;
    }
    bool isBad() { // 判断是否需要重构
      return ((ch[0]->cover > cover * alpha + 5) ||
(ch[1]->cover > cover * alpha + 5));
    }
  };
  struct STree {
  protected:
    Node mem_pool[MAXN];    //内存池，直接
分配好避免动态分配内存占用时间
    Node *tail, *root, *null;  // 用 null 表示 NULL
的指针更方便，tail 为内存分配指针，root 为根
    Node *bc[MAXN]; int bc_top; // 储存被删除
的节点的内存地址，分配时可以再利用这些地
址

    Node * NewNode(int key) {
      Node * p = bc_top ? bc[--bc_top] : tail++;
      p->ch[0] = p->ch[1] = null;
      p->size = p->cover = 1; p->exist = true;
      p->key = key;
      return p;
    }
    void Travel(Node * p, vector<Node *>&v) {
      if (p == null) return;
      Travel(p->ch[0], v);
      if (p->exist) v.push_back(p); // 构建序列
      else bc[bc_top++] = p; // 回收
      Travel(p->ch[1], v);
    }
    Node * Divide(vector<Node *>&v, int l, int r) {
      if (l >= r) return null;
      int mid = (l + r) >> 1;
      Node * p = v[mid];
      p->ch[0] = Divide(v, l, mid);
      p->ch[1] = Divide(v, mid + 1, r);
      p->PushUp(); // 自底向上维护，先维护子
树
      return p;
```

```
    }
    void Rebuild(Node * &p) {
      static vector<Node *>v; v.clear();
      Travel(p, v); p = Divide(v, 0, v.size());
    }
    Node ** Insert(Node *&p, int val) {
      if (p == null) {
        p = NewNode(val);
        return &null;
      }
      else {
        p->size++; p->cover++;
        // 返回值储存需要重构的位置，若子树也需要重构，本节点开始也需要重构，以本节点为根重构
        Node ** res = Insert(p->ch[val >= p->key], val);
        if (p->isBad()) res = &p;
        return res;
      }
    }
    void Erase(Node *p, int id) {
      p->size--;
      int offset = p->ch[0]->size + p->exist;
      if (p->exist && id == offset) {
        p->exist = false;
        return;
      }
      else {
        if (id <= offset) Erase(p->ch[0], id);
        else Erase(p->ch[1], id - offset);
      }
    }
  public:
    void Init() {
      tail = mem_pool;
      null = tail++;
      null->ch[0] = null->ch[1] = null;
      null->cover = null->size = null->key = 0;
      root = null; bc_top = 0;
    }
    STree() { Init(); }
    void Insert(int val) {
      Node ** p = Insert(root, val);
      if (*p != null) Rebuild(*p);
    }
    int Rank(int val) {
      Node * now = root;
      int ans = 1;
      while (now != null) { // 非递归求排名
        if (now->key >= val) now = now->ch[0];
        else {
          ans += now->ch[0]->size + now->exist; //判断 now 是否 valid
          now = now->ch[1];
        }
      }
      return ans; // ans >= 1
    } // 若 val 属于(1th,2th) 则 Rank(val)=2
    int Kth(int k)  {
      Node * now = root;
      while (now != null) { // 非递归求第 K 大
        if (now->ch[0]->size + 1 == k && now->exist) return now->key;
        else if (now->ch[0]->size >= k) now = now->ch[0];
        else k -= now->ch[0]->size + now->exist, now = now->ch[1];
      }
      return -1; // k 非法
    }
    void Erase(int val) {
      Erase(root, Rank(val));
       if (root->size < alpha * root->cover)
          Rebuild(root);
    }
    void Erase_kth(int k) {
      Erase(root, k);
       if (root->size < alpha * root->cover)
          Rebuild(root);
    }
  };
#undef MAXN
}
int main() {
  using namespace Scapegoat_Tree;
  STree solver; solver.Init();
  int T; cin >> T;
  while (T--) {
    int opt, x; scanf("%d%d", &opt, &x);
    switch(opt) {
      case 1: solver.Insert(x);break;
      case 2: solver.Erase(x);break;
       case 3:
       printf("%d\n", solver.Rank(x));break;
       case 4:
       printf("%d\n", solver.Kth(x));break;
       case 5:
printf("%d\n", solver.Kth(solver.Rank(x) -1));break;
      case 6: printf("%d\n", solver.Kth(solver.Rank(x+1)));break;
    }
  }
  return 0;
}
```

// ============ 线段树 ============

```
ll n, m; // index 1~n 一共 m 次操作
ll op, qL, qR, v;
// 每次 update 或 query 前 都必须 clarify
// 对于 set: v >= 0 !!!
ll _sum, _min, _max; // 每次 query 前都要 init
const ll maxnode = 1<<17;
```

```
const ll INF = 0x3f3f3f3f3f3f3f3f;

struct IntervalTree{
ll addv[maxnode*4],setv[maxnode*4];
ll sumv[maxnode*4],minv[maxnode*4];
ll maxv[maxnode*4];

  void maintain(ll o, ll L, ll R){
    ll lc = o*2, rc = o*2+1;
    sumv[o] = maxv[o] = minv[o] = 0;
    if(L < R){
      sumv[o] = sumv[lc] + sumv[rc];
      maxv[o] = max(maxv[lc], maxv[rc]);
      minv[o] = min(minv[lc], minv[rc]);
    }
    if(setv[o] >= 0){
      //when set included
      minv[o] = maxv[o] = setv[o];
      sumv[o] = setv[o] * (R-L+1);
    }
    if(addv[o]){
      minv[o] += addv[o];
      maxv[o] += addv[o];
      sumv[o] += addv[o] * (R-L+1);
    }
  }
  void pushdown(ll o){  // when set
    ll lc = o*2, rc = o*2+1;
    if(setv[o] >= 0){
      setv[lc] = setv[rc] = setv[o];
      addv[lc] = addv[rc] = 0;
      setv[o] = -1;
    }
    if(addv[o]){
      addv[lc] += addv[o];
      addv[rc] += addv[o];
      addv[o] = 0;
    }
  }
  void update(ll o, ll L, ll R){
    ll lc = o*2, rc = o*2+1;
    if(qL <= L && qR >= R){
      if(op == 2) { // set
        setv[o] = v;
        addv[o] = 0;
      }
      else { //op==1 :Add
        addv[o] += v;
      }
    }
    else{
      pushdown(o);  //when set
      ll M = L + (R-L)/2;
      if(qL <= M) update(lc, L, M);
      else maintain(lc, L, M); //when set
      if(qR > M) update(rc, M+1, R);
      else maintain(rc, M+1, R);//when set
    }
    maintain(o, L, R);
  }
  void query(ll o, ll L, ll R, ll add) {
    //只需要 set 时可以删去第四个参数
    if(setv[o] >= 0){ // when set included
      ll v = setv[o] + addv[o] + add;
      _sum += v * (min(R, qR)-max(L, qL)+1);
      _max = max(_max, v);
      _min = min(_min, v);
    }
    else if(qL <= L && qR >= R){
      //当前区间完全包含于询问中
      _sum += sumv[o] + add*(R-L+1);
      _max = max(_max, maxv[o]+add);
      _min = min(_min, minv[o]+add);
    }
    else{ // 递归统计 累加参数 add
      ll lc = o*2, rc = o*2+1;
      ll M = L + (R-L)/2;
      if(qL <= M) query(lc, L, M, add+addv[o]);
      if(qR > M) query(rc, M+1, R, add+addv[o]);
    }
  }
} tree;

int main(){
  scanf("%lld%lld",&n,&m);
  memset(&tree, 0, sizeof(tree)); // important!!

  for (ll i=1; i<=n; i++) {
    scanf("%lld", &v);
    qL = qR = i;
    op = 1;
    tree.update(1, 1, n);
  }
  if (s == "add") {
    scanf("%lld%lld%lld",&qL,&qR,&v);
    op = 1;
    tree.update(1, 1, n);
  }
  if (s == "set") {
    scanf("%lld%lld%lld",&qL,&qR,&v);
    op = 2;
    tree.update(1, 1, n);
  }
  if (s == "sum") {
    scanf("%lld%lld",&qL,&qR);
    _sum = 0; _max = -INF; _min = INF;
    tree.query(1, 1, n, 0);
    printf("%lld\n", _sum);
  }
}
```

// ========= Trie 树 =========

```
const int maxnode = 4000010;
const int sigma_size = 26;
struct Trie {
   int ch[maxnode][sigma_size];
   int sz, val[maxnode];
   void clear()
{sz=0;memset(ch[0],0,sizeof(ch[0]));}
   // 一开始只有一个根节点 0
   // 点的编号: 0 ~ sz
   int idx(char c){return c-'a';}
   void insert(char *s, int v) {
      int u = 0, n = strlen(s);
      for(int i=0;i<n;i++) {
         int id = idx(s[i]);
         if(!ch[u][id]) {
            ++sz;
            memset(ch[sz],0,sizeof(ch[sz]));
            val[sz]=0;
            ch[u][id]=sz;
         }
         u = ch[u][id];
      }
      val[u] += v;
   }
   int search(char *s) {
      int u = 0, n = strlen(s);
      for(int i=0;i<n;i++) {
         int id = idx(s[i]);
         if(ch[u][id]==0)
            return 0;
         u = ch[u][id];
      }
      return val[u];
   }
};
Trie trie;
```

// ========= AC 自动机 =========

```
inline void insert(char* word, int value) {
   int len = strlen(word), j = 0;
   for (int i=0; i<len; ++i) {
      int c = word[i] - 'a';
      if(!ch[j][c]) ch[j][c] = ++size;
      j = ch[j][c];
   }
   val[j]+=value;
}
inline void GetFail() {
   queue<int> q;
   fail[0] = 0;
   for (int c = 0; c < Sigma_Size; ++c) {
      int p = ch[0][c];
      if(p) {
         fail[p] = last[p] = 0;
         q.push(p);
      }
   }
   while(!q.empty()) {
      int head = q.front();
      q.pop();
      for (int c = 0; c < Sigma_Size; ++c) {
         int u = ch[head][c];
         if(!u) continue;
         q.push(u);
         int v = fail[head];
         while(v && !ch[v][c]) v = fail[v];
         fail[u] = ch[v][c];
         last[u] = val[fail[u]] ? fail[u] :
last[fail[u]];
//这样保证了沿 last 数组经过的节点(除了 u
与 root) 都会是单词节点(val>0)
//val[u]有可能大于 0
      }
   } }
inline void Founded(int x) {
         for(; x; x=last[x]) cnt[x]++;}
// last[i]=j 表 j 节点表示的单词是 i 节点单词
的后缀，且 j 节点是单词节点
// 递归打印与结点 i 后缀相同的前缀节点编
号
// 进入此函数前需保证 val[x]>0
// cnt[] 记录某个点代表的单词 在文章中出
现的次数
inline void Find(char* text) {
   int j = 0, len = strlen(text);
   memset(cnt, 0, sizeof(cnt));
   for (int i=0; i<len; ++i) {
      int c = text[i] - 'a';
      while(j && !ch[j][c]) j = fail[j];
      j = ch[j][c];
      if(val[j]) Founded(j);
      else if(last[j]) Founded(last[j]);
   }
}
// main():  insert(P, 1);  GetFail(); Find(T);
```

// ========= KMP 匹配=========

```
char P[maxn]; // Pattern 短串
char T[maxn]; // Text 长串
int f[maxn];
```

```
void getFail(char* P,int* f) {
//字符串 p 自我匹配
   int m = strlen(P);
   f[0] = f[1] = 0;
   for(int i = 1; i < m; i++) {
      int j = f[i];
      while(j && P[i]!=P[j])
         j = f[j];
      f[i+1] = P[i]==P[j] ? j+1 : 0;
   }
}
void Find(char* T, char* P, int* f) {
//p 去匹配字符串 T
   int n = strlen(T), m = strlen(P);
   getFail(P, f);  //得出部分匹配表
   int j = 0;
   //j:短串的下标  i: 长串下标
   for (int i = 0; i < n; i++) {
      while (j && P[j] != T[i])
         j = f[j];
      if (P[j] == T[i]) j++;
      if(j == m)
         printf("%d ", i-m+1);
   }
   puts("");
}
int main() {
   // c++ getline(cin, P)
   // c   gets(P)
   while (gets(P))
   {gets(T); Find(T, P, f);}
}
```

// ========= 后缀数组 =========

```
const int CHARSET_SIZE = 257;
string s, s2;
int sa[maxn], rk[maxn], ht[maxn];
int cnt[maxn], rk1[maxn], rk2[maxn];
bool cmpSA(int *y,int a,int b,int k, int n) {
   int a1=y[a];
   int b1=y[b];
   int a2=a+k >= n ? -1: y[a+k];
   int b2=b+k >= n ? -1: y[b+k];
   return a1==b1 && a2==b2;
}
void buildSA(const string& str,int n,int m){
// or "const char* s"
   for(int i = 0; i < m; i++) cnt[i] = 0;
    for(int i = 0; i < n; i++)
      ++cnt[rk1[i]=(int)str[i]];
   for(int i = 1; i < m; i++) cnt[i] += cnt[i-1];
    for(int i = n-1; i >= 0; i--)
         sa[--cnt[rk1[i]]]=i;
   for(int len=1; len<=n; len<<=1) {
      int p=0;
      for(int i = n-len; i < n; i++) rk2[p++]=i;
      for(int i=0; i<n; i++)
         if( sa[i]>=len )
           rk2[p++]=sa[i]-len;
      for(int i = 0; i < n; i++) cnt[i]=0;
      for(int i = 0; i < n; i++)
      ++cnt[rk1[rk2[i]]];
      for(int i = 1; i < n; i++)
          cnt[i] += cnt[i-1];
      for(int i = n-1; i >= 0; i--)
          sa[--cnt[rk1[rk2[i]]]]=rk2[i];
      for(int i = 0; i < n; i++)
          swap(rk1[i], rk2[i]);
      int tot_rk = 1;
      rk1[sa[0]] = 0;
      for (int i = 1; i < n; i++)
      rk1[sa[i]] =
          cmpSA(rk2,sa[i],sa[i-1],len,n)
          ? tot_rk-1 : tot_rk++;
      if (tot_rk >= n) break;
   }
}
void getHeight(const string& str, int n) {
   for (int i = 0; i<n; i++) rk[sa[i]] = i;
   ht[0] = 0;
   for (int i = 0, k = 0; i < n; i++) {
      if (rk[i] == 0) continue;
      int j = sa[rk[i] - 1];
      if (k) k--;
      while (str[i + k] == str[j + k]) k++;
      ht[rk[i]] = k;
   }
}
int main() {
   getline(cin, s);
   getline(cin, s2);
   s = s + '$' + s2;
   int N = s.size();
   buildSA(s, N, CHARSET_SIZE);
   getHeight(s, N);
}
```

// ========= FFT 傅里叶=========

```
#include <algorithm>
#include <cmath>
using namespace std;
const double PI = acos(-1.0);
struct complex {
  double r,i;
  complex(double _r = 0.0,double _i = 0.0)
  {r = _r; i = _i;}
  complex operator +(const complex &b)
  {return complex(r+b.r,i+b.i);}
  complex operator -(const complex &b)
  {return complex(r-b.r,i-b.i);}
  complex operator *(const complex &b)
  {return complex(r*b.r-i*b.i,r*b.i+i*b.r);}
};
void change(complex y[],int len) {
  int i,j,k;
  for(i = 1, j = len/2;i < len-1; i++)
  {
    if(i < j)swap(y[i],y[j]);
    k = len/2;
    while( j >= k) {j -= k;k /= 2;}
    if(j < k) j += k;
  }
}
void fft(complex y[],int len,int on)
//on==-1 IDFT
{
  change(y,len);
  for(int h = 2; h <= len; h <<= 1)
  {
    complex wn(cos(-on*2*PI/h),sin(-on*2*PI/h));
    for(int j = 0;j < len;j+=h)
    {
      complex w(1,0);
      for(int k = j;k < j+h/2;k++)
      {
        complex u = y[k];
        complex t = w*y[k+h/2];
        y[k] = u+t;
        y[k+h/2] = u-t;
        w = w*wn;
      }
    }
  }
  if(on == -1)
    for(int i = 0;i < len;i++) y[i].r /= len;
}
const int MAXN = 200011;
complex x[MAXN * 4];
LL num[MAXN * 4]; int a[MAXN];
int main() {
  memset(num, 0, sizeof(num));
  for (int i = 0;i < N; i++) {
    scanf("%d", &a[i]);
    num[a[i]]++;
  }
  sort(a, a+N);
  int len_tmp = a[N-1]+1, len = 1;
  while (len < len_tmp*2)
    len <<= 1;
  for (int i=0;i<len;i++)
    x[i] = complex(num[i],0);
  fft(x, len, 1);//DFT
  for(int i = 0;i < len;i++)
    x[i] = x[i]*x[i];
  fft(x, len, -1);//IDFT
  for (int i = 0;i < len;i++)
    num[i] = (LL)round(x[i].r);
  //可能要：求组合而不是求排列
  num[i] /= 2;
  //可能要：扣除 a[i]+a[i]的情况
  num[a[i]+a[i]]--;
  //可能要：扣除带 0 的特殊情况
  Cnt -= (LL)Cnt0 * (N-1) * 2LL;// 0+ai=ai &&
ai+0=ai
  printf("%lld\n", Cnt);     }
```

// =========== Catalan ===========

Catalan 数

Cat[n]=C[2*n][n]/(n+1)

组合性质

Cat[n+1]=sum(Cat[i]*Cat[n-i] for i from 0 to n)

Cat[0]=1, Cat[n+1]=2*(2*n+1)/(n+2)*Cat[n]

Cat[n]为长度为 2*n 的 Dyck 词总数(长度为 2*n 的 Dyck 词由 n 个'X'和 n 个'Y'组成, 对于其任意前缀, 有 count('X')>=count('Y'))

Cat[n]为给长度(n+1)的序列打上括号的不同方案数

Cat[n]为有(n+1)片叶子的不同完全二叉树数

Cat[n]为 n*n 网格线从左下角到右上角不经过左上部分的最短路径数

Cat[n]为用不相交直线将凸(n+2)边形划分为 n 个三角形的方案数

Cat[n]为有 n 个非叶子节点的不同二叉树数

// ========== Combination ==========

```
ll fast_pow(ll x, ll k, ll p);
int Combination(int m, int n, int p){
      ll nom=1, den=1;
      for(int i=m-n+1; i<=m; i++)
      {nom*=i; nom%=p; }
      for(int i=2; i<=n; i++)
      {den*=i; den%=p; }
      den=fast_pow(den, p-2, p);
      return (nom*den)%p;
}
ll C[maxn][maxn];
int Combination_table(int n, ll MOD){
      memset(C, 0, sizeof(C));
      C[0][0]=1;
```

```
	for(int i=1; i<=n; i++){
		C[i][0]=1;
		for(int j=1; j<=i; j++)
		C[i][j]=(C[i-1][j-1]+C[i-1][j])%MOD;
	}
}
```

// ====== cont_frac 连分数逼近 ======

```
ll a[maxn];
/* 连分数逼近由欧几里德算法求解
n/d=a[0]+1/(a[1]+1/(a[2]+1/(a[3]+1/(...+1/ a[len-1]))))  */
int cont_frac(ll n, ll d){
	ll r;
	int len=0;
	while(d){
		a[len++]=n/d;
		r=n%d; n=d; d=r;
	}
	return len;
}
```

// ========== Euler_Phi ==========

```
int euler_phi(int n){
	int m=int(sqrt(n+0.5)), res=n;
	for(int i=2; i<=m; i++) if(n%i==0){
		res=res/i*(i-1);
		while(n%i==0) n/=i;
	}
	if(n>1) res=res/n*(n-1);
	return res;
}
int phi[maxn];
void phi_table(int n){
	memset(phi, 0, sizeof(phi));
	phi[1]=1;
	for(int i=2; i<=n; i++) if(!phi[i])
		for(int j=i; j<=n; j+=i){
			if(!phi[j]) phi[j]=j;
			phi[j]=phi[j]/i*(i-1);
		}
}
```

// ======== Fibonacci 数 ========

F[0]=F[1]=1;
F[n]=F[n-1]+F[n-2];
组合性质
F[n]=sum(C[n-k-1][k] for k=0 to floor((n-1)/2))
C[i][j]表示组合数
sum(F[i] for i=1 to n)=F[n+2]-1
sum(F[2*i+1] for i=0 to n-1)=F[2*n]
sum(F[2*i] for i=1 to n)=F[2*n+1]-1
sum(F[i]*F[i] for i=1 to n)=F[n]*F[n+1]
Catalan 性质:
F[n]*F[n]-F[n-r]*F[n+r]=(-1)^(n-r)*F[r]*F[r]
Vajda 性质: F[n+i]*F[n+j]-F[n]*F[n+i+j]=
(-1)^n*F[i]*F[j]
d'Ocagne 性质:
F[k*n+c]=sum
(C[k][i]*F[c-i]*F[n]^i*F[n+1]^(k-i) for i=0 to k)

母函数: s(x)=sum
(F[k]*x^k for k=0 to infinity)=x/(1-x-x^2)
数论性质:
gcd(F[m], F[n])=F[gcd(m, n)]
整数 N 为 Fibonacci 数的充要条件为 5*N^2+4 或 5*N^2-4 为完全平方数
p|F[p-(5/p)] (此处括号为 Legendre 标记)
如果从 1 开始计数, 则除了 F[4]=3 以外, 若下标 n 为合数则 F[n]也为合数
除了 1 以外 Fibonacci 数中仅有 8 和 144 为整次方数
除了 1, 8, 144, 所有 Fibonacci 数都有至少一个质因数不在所有比其下标更小的 Fibonacci 数的质因数的集合中
若构造一数列 A[i]=F[i]%n, n 为任意正整数, 则数列 A 有周期性且其周期不超过 6*n

// ============ primes ============

Legendre 符号 (p 为质数)
(a|p)=0 if a%p=0
(a|p)=1 if a%p!=0 and 存在整数 x x^2=a (mod p)
(a|p)=-1 otherwise
Euler 准则: 若 p 为奇质数且 p 不能整除 d 则
d^((p-1)/2)=(d|p) (mod p)
Legendre 符号是完全积性函数
二次互反律: 若 p, q 为奇质数, 则(q|p)=(p|q)*(-1)^((p-1)/2*(q-1)/2)

Mersenne 数
M[n]=2^n-1
Euclid-Euler 定理: 若 M[p]为素数, 则(2^p-1)*2^(p-1)为完全数
若 p 为奇质数, 则 M[p]的所有质因子模 2*p 同余 1
若 p 为奇质数, 则 M[p]的所有质因子模 8 同余 +/-1
M[m]与 M[n]互质的充要条件为 m 与 n 互质
若 p 与 2*p+1 皆为素数且 p 模 4 同余 3, 则 (2*p+1)为 M[p]的因子

Wilson 定理
大于 1 的自然数 n 为素数的充要条件为(n-1)!=-1 (mod n)

Fermat 多边形数定理
每一个正整数最多可以表示为 n 个 n 边形数之和

Euler 引理
对于任意奇素数 p, 同余方程 x^2+y^2+1=0 (mod p) 必有一组正整数解(x, y)满足 0<=x<p/2, 0<=y<p/2
Lagrange 的四平方和定理
每个正整数均可以表示为 4 个整数的平方和

// ============ log_mod ============

```
//解方程 a^x=b (mod n) n 为素数
int shank(int a, int b, int n){
	int m, v, e=1, i;
	m=int(sqrt(n+0.5));//复杂度为O((m+n/m)logm) 所以 m==n/m 时最快
	v=inv(fast_pow(a, m, n), n);//fast_pow(a, m, n)=(a^m)%n
	map<int, int> x;//x[j]=min(i|e[i]==j)
	x[1]=0;
	for(int i=1; i<m; i++){
		e=a*e%n;//e=(a^i)%n
		if(!x.count(e)) x[e]=i;
	}
	for(int i=0; i<m; i++){
		//a^(im)到 a^(im+m-1)
		if(x.count(b)) return i*m+x[b];
		b=b*v%n;//递推更新 b
	}
	return -1;//无解
}
```

// ============ matrix ============

```
struct parametre{int c, r;};
struct Matrix{
   long long matrix[maxn][maxn];
   parametre DIM;
   Matrix(){}
   Matrix(int c, int r){
      DIM={c, r};
      memset(matrix, 0, sizeof(matrix));
   }
   Matrix operator*(Matrix &A){
      Matrix C(DIM.c, A.DIM.r);
      memset(C.matrix, 0, sizeof(C.matrix));
      for(int i=0; i<DIM.c; i++)
         for(int j=0; j<A.DIM.r; j++)
            for(int k=0; k<DIM.r; k++)
C.matrix[i][j]=(C.matrix[i][j]+A.matrix[i][k]*matrix[k][j])%MOD;
      return C;
   }
   Matrix operator+(Matrix &A){
      Matrix C(A.DIM.c, A.DIM.r);
      for(int i=0; i<DIM.c; i++)
         for(int j=0; j<DIM.r; j++)
         C.matrix[i][j]=A.matrix[i][j]+matrix[i][j];
      return C;
   }
   void print(){
      for(int i=0; i<DIM.c; i++){
         for(int j=0; j<DIM.r; j++)
cout<<matrix[i][j]<<'\t';
         cout<<endl;
      }
   }
};
Matrix BigMatrixExpo(Matrix &A, long long n){
   Matrix B=A;
   Matrix C(A.DIM.c, A.DIM.r);
   for(int i=0; i<C.DIM.c; i++)
      for(int j=0; j<C.DIM.r; j++)
         C.matrix[i][j]=i==j;
   while(n){
      if(n&1) C=C*B;
      B=B*B;
      n>>=1;
   }
   return C;
}
//定义新矩阵 Matrix a(3, 5); a.matrix={{},{},{}};
//乘法 c=a*b; 注意 a 的第一个 parametre 等于 b 的第二个 parametre;
//加法 c=a+b;  //输出 c.print();
```

// ============ 莫比乌斯 ============

```
//完全积性函数 mo[i*j]=mo[i]*mo[j]
//sum(mo[d] for d|n)=(n==1)
//反演公式
//若 f(n)=sum(g(d) for d|n) 则
g(n)=sum(mo[n/d]*f(d) for d|n)=sum(mo[d]*f(n/d) for d|n)
//If f(i)=sum(g(d*i) for d from 1 to floor(n/i)) then
g(i)=sum(f(d*i)*mo[d] for d from 1 to floor(n/i))
bool vis[maxn+123];
int mo[maxn+123], primes[maxn+123], a[maxn+123], pcnt, N;
void mobius(){//预处理
	mo[1]=1;
	for(int i=2; i<=maxn; i++){
		if(!vis[i])
		{ mo[i]=-1; primes[pcnt++]=i; }
		for(int j=0; j<pcnt&&ll(i)*primes[j]<=maxn; j++){
			vis[i*primes[j]]=true;
			if(i%primes[j]) mo[i*primes[j]]=-mo[i];
			else{
				mo[i*primes[j]]=false;
				break;
			}
		}
	}
	for(int i=2; i<=maxn; i++) mo[i]+=mo[i-1];//mo 记录前缀和
}
```

```
//O(sqrt(n)+sqrt(m))
ll cnt_gcd(ll n, ll m, ll k){//for i from 1 to n for j from 1 to m cnt gcd(i, j)=k
	if(n>m) swap(n, m);
	ll res=0;
	n/=k, m/=k;
	for(int i=1, j=1; i<=n; i=j+1){
		j=min(n/(n/i), m/(m/i));
		res+=ll(mo[j]-mo[i-1])*(n/i)*(m/i);//前缀和 Mobius
	}
	return res;
}
```

// =========== 高斯消元 ============

```
typedef int Matrix[maxn][maxn];
void exgcd(int a, int b, int& d, int& x, int& y){
	!b?(d=a, x=1, y=0):(exgcd(b, a%b, d, y, x), y-=x*(a/b));
}
int inv(int a){
	int d, x, y;
	exgcd(a, MOD, d, x, y);
	return d==1?(x+MOD)%MOD:-1;
}
int gauss_jordan(Matrix A, int n, int m){//A 是增广矩阵, n 个未知数, m 个方程, MOD 是模, 如果 MOD 不是质数的话每次 inv 完先检测是否是-1
	int i=0, j=0;
	while(i<m&&j<n){
		int row=i;
		for(int k=i; k<m; k++){
			if(A[k][j]){
				row=k;
				break;
			}
		}
		if(row!=i) for(int k=0; k<=n; k++)
			swap(A[i][k], A[row][k]);
		if(!A[i][j]){
			j++; continue;
		}
		for(int k=0; k<m; k++){
			if(!A[k][j]||i==k) continue;
			int cur=A[k][j]*inv(A[i][j])%MOD;
			for(int t=j; t<=n; t++)
			A[k][t]=( (A[k][t]-cur*A[i][t])%MOD+MOD)%MOD;
		}
		i++;
	}
	for(int k=i; k<m; k++)
		if(A[k][n]) return -1;//无解
	if(i<n) return 0;//无限解
	for(int k=0; k<n; k++)
		A[k][n]=A[k][n]*inv(A[k][k])%MOD;
	//解存在 A[k][n]里面
	return 1;
}
```

// ======= pell equation 佩尔方程======

```
//用于求解标准型 Pell 方程的第(k+1)组非平凡解 (x^2-n*y^2=1)
//输入 n, k 和 MOD
//递推关系为 x[i+1]=x[0]*x[i]+n*y[0]*y[i];
//y[i+1]=y[0]*x[i]+x[0]*y[i];
//上述递推关系可由 sqrt(n)的连分数表示推出
typedef pair<ll, ll> pii;
pii res;//(xk, yk)
ll MOD;//模<必须是全局变量>
void Find(ll n, ll& x, ll& y){
	//暴力寻找特解(x0, y0)
	y=1;
	while(true){
		x=sqrt(y*y*n+1);
		if(x*x-n*y*y==1) break;
		y++;
	}
}
struct parameter{int c, r;};
struct Matrix{
  ll matrix[maxn][maxn];
  parameter DIM;
  Matrix(){}
  Matrix(int c, int r);
  Matrix operator*(Matrix &A);//带模乘法
  Matrix operator+(Matrix &A);
  void print();
};
Matrix BigMatrixExpo(Matrix &A, ll n);//带模快速幂
bool Pell(ll n, ll k){//k 为第 k 组解, 从 0 开始数
	ll t=sqrt(n)+0.5, x, y;
	if(t*t==n) return false;//仅有平凡解 (1, 0) 和 (-1, 0)
	Matrix A(2, 2);
	Find(n, x, y);
	A.matrix[0][0]=A.matrix[1][1]=x;
	A.matrix[0][1]=n*y;
	A.matrix[1][0]=y;
	A=BigMatrixExpo(A, k-1);
	res=make_pair((A.matrix[0][0]*x+A.matrix[0][1]*y)%MOD,
(A.matrix[1][0]*x+A.matrix[1][1]*y)%MOD);
	return true;
}
```

```
// ============ CRT===========
typedef long long ll;
//n 个方程为 x=a[i] (mod m[i])
ll china(int n, int* a, int* m){
	ll M=1, d, y, x=0;//M 是等价以后的模
	for(int i=0; i<n; i++) M*=m[i];
	for(int i=0; i<n; i++){
		ll w=M/m[i];
		exgcd(m[i], w, d, d, y);
		x=(x+y*w*a[i])%M;
	}
	return (x+M)%M;
}
// ============ Fraction ===========
ll gcd(ll a, ll b){ return !b?a:gcd(b, a%b); }
struct fraction{
ll num, den;
fraction(){ num=0; den=1; }
fraction(ll a, ll b)
{num=a; den=b; simplify();}
inline void reset(){ num=0; den=1;}
void simplify(){
		ll d=gcd(num, den);
		num/=d;
		den/=d;
		if(den<0){num=-num;den=-den;}
}
inline ll convert(){return num/den;}
fraction& operator = (int rhs){
		(*this).num=rhs;
		(*this).den=1;
		return *this;
}
fraction operator + (const fraction &rhs) const{
		fraction res;
		res.den=lcm(den, rhs.den);
res.num=res.den/den*num+res.den/rhs.den*rhs.n
um;
		res.simplify();
		return res;
}
fraction operator += (const fraction &rhs)
{return *this=*this+rhs;}
fraction operator + (const int &rhs) const{
		fraction r(rhs, 1);
		return *this+r;
}
fraction operator += (const int &rhs)
{return *this=*this+rhs;}
fraction operator - (const fraction &rhs) const{
		fraction res;
		res=*this+fraction(-1, 1)*rhs;
		res.simplify();
		return res;
}
fraction operator -= (const fraction &rhs)
{return *this=*this-rhs;}
fraction operator - (const int &rhs) const
{fraction r(rhs, 1);return *this-r;}
fraction operator -= (const int &rhs)
{return *this=*this-rhs;}
fraction operator * (const fraction &rhs) const{
		fraction res;
		res.num=num*rhs.num;
		res.den=den*rhs.den;
		res.simplify();
		return res;
}
fraction operator *= (const fraction &rhs)
{return *this=(*this)*rhs;}
fraction operator * (const int &rhs) const
{fraction r(rhs, 1); return (*this)*r;}
fraction operator *= (const int &rhs)
{return *this=(*this)*rhs;}
fraction operator / (const fraction &rhs) const{
		fraction res;
		res.num=num*rhs.den;
		res.den=den*rhs.num;
		res.simplify();
		return res;
}
fraction operator /= (const fraction &rhs)
{return *this=(*this)/rhs;}
fraction operator / (const int &rhs) const
{ fraction r(rhs, 1); return (*this)/r; }
fraction operator /= (const int &rhs)
	{return *this=(*this)/rhs;}
bool operator == (const fraction &rhs) const
	{return num*rhs.den==den*rhs.num;}
bool operator == (const int &rhs) const
	{return num==den*rhs;}
bool operator != (const fraction &rhs) const
	{return !(*this==rhs);}
bool operator != (const int &rhs) const
	{return !(*this==rhs);}
bool operator < (const fraction &rhs) const
	{return num*rhs.den<den*rhs.num;}
bool operator < (const int &rhs) const
	{return num<den*rhs;}
bool operator > (const fraction &rhs) const
	{return num*rhs.den>den*rhs.num;}
bool operator > (const int& rhs) const
	{return num>den*rhs;}
bool operator <= (const fraction &rhs) const
	{return *this==rhs||*this<rhs;}
bool operator <= (const int& rhs) const
	{return *this==rhs||*this<rhs;}
bool operator >= (const fraction &rhs) const
	{return *this>rhs||*this==rhs;}
bool operator >= (const int &rhs) const
	{return *this>rhs||*this==rhs;}
};
```

// ========= 辛普森积分 =========

```
double simpson(double a, double b) {
	double c = (a + b) / 2.0;
	return (F(a)+4*F(c)+F(b)) * (b-a) / 6.0;
} // 这里 F 为自定义函数
//given A as the simpson Value for the whole interval [a,b]
double asr(double a, double b, double eps, double A) {
	double c = (a + b) / 2.0;
	double L = simpson(a, c)；
	double R = simpson(c, b);
	if  (fabs(L+R-A) <= 15*eps)
		return L + R + (L+R-A)/15.0;
return asr(a, c, eps/2, L) + asr(c, b, eps/2, R);
}
double asr(double a, double b, double eps)
{ return asr(a, b, eps, simpson(a, b)); } //接口
// int main(): 调用 asr(left, right, 1e-5)
// 得到 F(x) 在[left, right]上的积分 eps 也可改为 1e-6
```

// ============ 凸包 ============

```
const double eps=1e-10;
const double PI=acos(-1);
struct Point{  double x, y;
  Point(double x=0, double y=0):x(x), y(y){}
} p[maxn], ch[maxn];
typedef Point Vector;
Vector operator + (Vector A, Vector B)
	{ return Vector(A.x+B.x, A.y+B.y); }
Vector operator - (Vector A, Vector B)
	{ return Vector(A.x-B.x, A.y-B.y); }
Vector operator * (Vector A, double p)
	{ return Vector(A.x*p, A.y*p); }
Vector operator / (Vector A, double p)
	{ return Vector(A.x/p, A.y/p); }
int dcmp(double x) {
  if(fabs(x) < eps) return 0;
  else return x < 0 ? -1 : 1; }
bool operator == (const Point& a, const Point& b)
{ return dcmp(a.x-b.x) == 0 && dcmp(a.y-b.y) == 0; }
bool operator < (const Point& a, const Point& b)
{ return a.x < b.x || (a.x == b.x && a.y < b.y); }

double cross(Vector A, Vector B)
	{ return A.x*B.y - A.y*B.x; }
double torad(double deg)
	{ return deg / 180 * PI; }
double PolygonArea(Point* p, int n){
  double area=0;
  for(int i=1; i<n-1; i++)
    area+=cross(p[i]-p[0], p[i+1]-p[0]);
  return area/2;
}
int convexhull(Point* p, int n, Point* ch) {
  sort(p, p+n);
  int m = 0;
  for(int i = 0; i < n; i++) {
    while(m > 1 && cross(ch[m-1]-ch[m-2], p[i]-ch[m-2]) <= 0) m--;
    ch[m++] = p[i];
  }
  int k = m;
  for(int i = n-2; i >= 0; i--) {
    while(m > k && cross(ch[m-1]-ch[m-2], p[i]-ch[m-2]) <= 0) m--;
    ch[m++] = p[i];
  }
  if(n > 1) m--; return m;
}
Vector Rotate(Vector A, double rad){
	// 这里 rad 是逆时针旋转的角度
	return Vector(A.x*cos(rad)-A.y*sin(rad),
	A.x*sin(rad)+A.y*cos(rad));
}
// int main()
Point o(tmpx, tmpy);
p[point_cnt++]=o; ... ...
int m=convexhull(p, point_cnt, ch);
double convex_area=PolygonArea(ch, m);
// Rotate vector(10,10) clockwise by 90 degree
// new_o = o + Rotate(Vector(10.0,10.0),-torad(90.0));
```

// ========== 点在多边形内 ==========

```
bool PNPoly(int u, int deg) {
	if (!  (vertxmin <= x[u] <= vertxmax)  ||  ! ( vertymin <= y[u] <= vertymax ) ) return 0;
	bool is_in = 0;  int i,j;
	for(i=0;i<deg;i++) {
	if(!i) j= deg-1;
	else  j= i-1;
	if ( ((poy[i] > y[u]) != (poy[j] > y[u])) && (x[u] < (pox[j] - pox[i]) * (y[u] - poy[i]) / (poy[j] - poy[i]) + pox[i]) )
			is_in = ! is_in;
	}
	return is_in;
}
```

// ======= 拓扑排序 =======

```
for(int i=0; i<cnt; i++)//入度为 0 的点入栈
    if(!indeg[i]) s.push(i);
while(!s.empty()){
   int u=s.top(); s.pop();
   T.push_back(u); //T 保存拓扑序
   for(int v:G[u]){ //G 为邻接表
      indeg[v]--;
      if(!indeg[v]) s.push(v);
   }
}
```

// ======= SCC 强连通分量 =======

```
int dfs_clock = 0, pre[MaxN],low[MaxN];
int scc_cnt, sccno[MaxN], size[MaxN];
stack<int> S; vector<int> G[MaxN];
void Add(int a,int b) {G[a].push_back(b);}
void dfs(int u) {
pre[u] = low[u] = ++dfs_clock;
S.push(u);
for (int i = 0; i < G[u].size(); i++) {
     int v = G[u][i];
     if (!pre[v]) {
          dfs(v);
          low[u] = min(low[u], low[v]);
     }
     else if (!sccno[v])
     low[u] = min(low[u],pre[v]);
}
if (low[u] == pre[u]) {
     scc_cnt++;
     int original_size = S.size(), tmp = 0;
     do {
          tmp = S.top();   S.pop();
          sccno[tmp] = scc_cnt;
     } while (tmp != u);
     size[scc_cnt] = original_size - S.size();
} } // end of dfs
int main()
{  for (int i=0; i<n; i++)  if (!pre[i])  dfs(i);  }
```

// ========= BCC 桥 割点 =========

```
struct Edge{ int u,v; };
vector<int>G[MAXN],bcc[MAXN];
vector<Edge>bridge;
int low[MAXN],pre[MAXN],dfs_clock=0,
iscut[MAXN],bccno[MAXN],bcc_cnt=0;
//bccno 是每个点在哪块 bcc 是第 i 块有哪些点
//点编号 0~n-1
stack<Edge>s;//保存在当前 bcc 中的边, 割顶的 bccno
无意义, 因为存在于多个 bcc 中
void dfs(int u, int fa)
{
   low[u]=pre[u]=++dfs_clock;
   int len=G[u].size(),i,child=0;
   for(i=0;i<len;i++)
   {
      int v=G[u][i];
      Edge e = {u,v};
      if(!pre[v]) //not accessed yet
      {
         s.push(e);//store cut edge
         dfs(v,u);
         child++;
         low[u]=min(low[u],low[v]);
         if(low[v] >= pre[u])  //if cut, bcc find
         {
            if(low[v] > pre[u])
               bridge.push_back(e);
            iscut[u]=1;
            bcc_cnt++;//bcc start from 1
            bcc[bcc_cnt].clear();
            while(1)
            {
               Edge x = s.top();
               s.pop();
               if(bccno[x.u]!=bcc_cnt)
               {
                  bcc[bcc_cnt].push_back(x.u);
                  bccno[x.u]=bcc_cnt;
               }
               if(bccno[x.v]!=bcc_cnt)
               {
                  bcc[bcc_cnt].push_back(x.v);
                  bccno[x.v]=bcc_cnt;
               }
               if(x.u==u && x.v==v) break;
            }
         }
      }
      else if( pre[u] > pre[v] && v!=fa )
      //early than father
      {
         s.push(e);
         low[u]=min(low[u],pre[v]);
      }
   }
   if(child<=1 && fa<0)
      iscut[u]=0;
}
int main() {
   scanf("%d%d",&n,&m);
   for(i=1;i<=m;i++) {
      scanf("%d%d",&a,&b);
      G[a].push_back(b);
      G[b].push_back(a);
   }
   for(i=0;i<n;i++)
```

```
    if(!pre[i])
      dfs(i,-1);
  for (int i=0; i<n; i++)
    if (iscut[i])
      printf("node %d is cut\n", i);
  for (auto e: bridge)
    printf("bridge %d -- %d\n",e.u,e.v);
}
```

// ========= 2-SAT =========

```
// node indexed from 0~n-1
#include <cstring>
int n, cnt=0, sol[MAXN*2];
vector<int> G[MAXN*2];
bool mark[MAXN*2];
void Clear() {
  for (int i = 0; i < MAXN*2; i++)
    G[i].clear();
  memset(mark, 0, sizeof(mark));
  cnt = 0;
}
//x=xval or y = yval
void add_constrain(int x,int xval,int y,int yval) {
  //x is xval OR y is yval
  x=2*x+xval;
  y=2*y+yval;
  G[x^1].push_back(y); //!x->y
  G[y^1].push_back(x); //!y->x
}
bool dfs(int u) {
  if(mark[u^1]) return false;
  if(mark[u]) return true;
  mark[u] = true;
  sol[cnt++]=u; // stack
  for(int i = 0;i<G[u].size();i++)
    if(!dfs(G[u][i]))
      return false;
  return true;
}
bool twosat() {
  for(int i=0;i<2*n;i+=2)
    if(!mark[i] && !mark[i+1]) { //未涂色
      cnt=0;
      if(!dfs(i)) { //出现 contradiction
        while(cnt) mark[sol[--cnt]]=0;
//i 的结果全部不要
        if(!dfs(i+1)) return false;
        //always contradict
      }
    }
  return true;
}
```

// ========= 2-SAT SCC =========

```
// node indexed from 0~n-1
#include <cstring>
int n, dfs_clock=0, scc_cnt=0;
int pre[MAXN*2],low[MAXN*2];
int sccno[MAXN*2];
stack<int> S;
vector<int> G[MAXN*2];
void Clear() {
  for (int i = 0; i < n*2; i++)
    G[i].clear();
  dfs_clock = scc_cnt = 0;
  memset(pre, 0, sizeof(pre));
  memset(low, 0, sizeof(low));
  memset(sccno, 0, sizeof(sccno));
}
void add_constrain(int x,int xval,int y,int yval)
{见 2-sat 普通版本的 add_constrain;}
void dfs(int u) { 间 SCC 部分的 dfs; }
bool twosat() {
  for (int i=0; i<n*2; i++)
    if (!pre[i])
      dfs(i);
  for (int i=0; i<n; i++)
    if (sccno[i*2] == sccno[i*2+1])
      return false;
  return true;
}
```

// ========== SPFA 最短路==========

```
struct Edge {int u, v, w;};
vector<Edge> edges; // need clearance
vector<int> G[maxn]; // need clearance
int dis[maxn]; bool vis[maxn]; queue<int> que;
void AddEdge(int a, int b, int c) {
   edges.push_back((Edge){a,b,c});
   G[a].push_back(edges.size()-1);
}
void Spfa(int Source) {
   memset(dis, 63, sizeof(dis));
   memset(vis, 0, sizeof(vis));
   que.push(Source); dis[Source] = 0;
   while (!que.empty()) {
      int u = que.front(); que.pop();
      for (int i=0; i<G[u].size(); i++) {
         Edge& e = edges[G[u][i]];
         int v = e.v, w = e.w;
         if (dis[v] > dis[u] + w) {
            dis[v] = dis[u] + w;
            if (!vis[v]) { vis[v] = 1;que.push(v); }
         }
      }
      vis[u] = 0;
   }
}
void Clear() {
   edges.clear();
```

```
    for (int i=0; i<maxn; i++)
        G[i].clear();
}
```

// ============ LCA ============

```
struct Edge{int u,v,w;};
vector<Edge> edges;
vector<int> G[maxn];
int dep[maxn]; //在 dfs 树上的深度
int f[maxn][maxlog], g[maxn][maxlog];
//点 index: 1~N (不能为 0)
//init: f, g = 0
//f[i,j]记录 i 结点向上走 2^j 步后所到达的祖先
//g[i,j]记 i 结点向上走 2^j 步路途中边权最小值
void dfs(int u) {
    for (int i=1;i<maxlog;i++) {
        f[u][i] = f[f[u][i-1]][i-1];
        g[u][i] = min(g[u][i-1], g[f[u][i-1]][i-1]);
    }
    for (int i=0; i<G[u].size(); i++) {
        Edge& e = edges[G[u][i]];
        int v = e.v, w = e.w;
        if (!dep[v]) {
            f[v][0]=u;
            g[v][0]=w;
            dep[v] = dep[u] + 1;
            dfs(v);
        }
    }
}
int LCA(int a,int b) {
    if (dep[a]>dep[b]) swap(a,b); //保证 b 更深
    int Ans=INF;
    for (int i=maxlog-1;i>=0;i--)
    if (dep[f[b][i]]>=dep[a]){
        Ans=min(Ans,g[b][i]);
        b=f[b][i];
    } //将 b 移动至与 a 同一深度
    if (a==b) return a; //LCA=a=b
    for (int i=maxlog-1;i>=0;i--)
        if (f[a][i]!=f[b][i]){
            Ans=min(Ans,min(g[a][i],g[b][i]));
            a=f[a][i];b=f[b][i];
        }//向上找到 LCA
    Ans=min(Ans,min(g[a][0],g[b][0]));
    return f[a][0]; //LCA=f[a][0]-f[b][0]
}
// int main()
for (int i=1; i<=n; i++) // 不连通的森林:
if (!dep[i]) { dep[i] = 1; dfs(i); }
// 牢记 dep[root] == 1 != 0
```

// ========= dijkstra =========

```
struct State {
    int u, d; State (int u = 0, int d = 0): u(u), d(d){}
    bool operator < (const State& another) const
{ return d > another.d; }
};
void dijkstra (int s) {
    memset(vis, 0, sizeof(int) * n);
    memset(dis, inf, sizeof(int) * n);
    dis[s] = 0;
    priority_queue<State> que;
    que.push(State(s, dis[s]));
    while (!que.empty()) {
        int u = que.top().u; que.pop();
        if (vis[u]) continue; vis[u] = 1;
        for (int i = 0; i < G[u].size(); i++) {
            Edge& edge = edges[G[u][i]];
            int v = edge.v, w = edge.w;
            if (dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;
                que.push(State(v, dis[v]));
            }
        }
    }
}
```

// ========= 二分图匹配 =========

```
// O(V*E)
// 点 index 不得为 0
vector<int> G[maxn];
int link[maxn];
bool vis[maxn];
int ans = 0;
int dfs(int u)  {
for (auto v : G[u])
if (!vis[v]) {
   vis[v]=1;
   if (!link[v] || dfs(link[v]))
   { link[v] = u; return 1; }
}  return 0; }
int main() {
   // 左半张图为 1~m 右半张图为 m+1~n
   // 由左半张图向右半张图连单向边
    for (int i=1;i<=M;i++)
    {memset(vis,0,sizeof(vis)); if(dfs(i)) ans++;}
    printf("%d\n", ans);
    for (int i=M+1;i<=N;i++)
    if (link[i]) printf("%d %d\n",link[i],i);
}
```

// ========= 网络流相关笔记 =========

========== 若带权值 ==========

最小点权覆盖 = 最大流

最大点权独立集合 = 总权值 - 最小点权覆盖

========== 不带权值 ==========

二分图最小顶点覆盖 = 二分图最大匹配；

二分图最大独立集 = 节点总数数（n）- 最大匹配数

求 DAG 图的最小路径覆盖:

(在图中找尽量少的路径，使得每个节点恰好在一条路径上)

1. 把原图中所有节点 i 拆成 i 与 i’
2. 如果原图存在有向边 i->j, 则在二分图中引入边 i->j’
3. DAG 最小路径覆盖 = 节点数（n）- 最大匹配数；

============================

网络流解法的构图：

超级源点与左边集合的每一点相连，若是求最小点覆盖，权值为 1，若是求最小点权覆盖集，权值为该点的点权

超级汇点与右边集合的每一点相连，权值同上

左右集合之间练得边容量均为 INF

// ============= DINIC() 最大流 ===============

```
// O(n^2*m)
// all con[]==1 O(min(n^(2/3),m(1/2))*m)
// 二分图匹配 O(n^(0.5) * m)
#include <cstring>
#define INF 0x3f3f3f3f
struct Edge {int from, to, cap;};
vector<Edge> edges;
vector<int> G[MaxNode];
int S, T;
int dis[MaxNode],cur[MaxNode];
bool vis[MaxNode];
void Clear() {
  edges.clear();
  for (int i=0; i<MaxNode; i++) G[i].clear();
}
void Add(int from, int to, int cap) {
  edges.push_back((Edge){from,to,cap});
  edges.push_back((Edge){to,from,0});
  int m = edges.size();
  G[from].push_back(m-2);
  G[to].push_back(m-1);
}
int bfs() {
  memset(vis,0,sizeof(vis));
  queue<int> Q;
  Q.push(S); dis[S]=0; vis[S]=1;
  while(!Q.empty()) {
    int u = Q.front(); Q.pop();
    for (int i=0; i<G[u].size(); i++) {
      Edge& e = edges[G[u][i]];
      if(!vis[e.to] && e.cap) {
        vis[e.to]=1;
        dis[e.to]=dis[u]+1;
        Q.push(e.to);
      }
    }
  }
  return vis[T];
}
int dfs(int u,int lim) {
  if (u==T || !lim) return lim;
  int flow=0, f;
  for (int& i=cur[u]; i<G[u].size(); i++) {
    Edge& e = edges[G[u][i]];
    if (dis[e.to]>dis[u] && e.cap)
      if ((f=dfs(e.to,min(lim-flow,e.cap))) > 0){
      flow+=f;
      e.cap-=f;
      edges[G[u][i]^1].cap+=f;
      if (flow==lim) break;
    }
  }
  return flow;
}
int DINIC() {
  int flow=0;
  while(bfs()) {
    memset(cur, 0, sizeof(cur));
    flow+=dfs(S,INF);
  }
  return flow;
}
```

// ========== 费用流 ==========

```
// O(SPFA_const * M * MAXFLOW)
// random Graph: n=250 m=5000 con[i]<=10000: 0.85s
struct Edge {int from, to, cap, cost;};
vector<Edge> edges; vector<int> G[maxn];
int S, T, ansflow;   long long anscost;
int dis[maxn], path[maxn];  bool vis[maxn];
void Add(int from, int to, int cap, int cost) {
   edges.push_back((Edge){from,to,cap,cost});
  edges.push_back((Edge){to,from,0,-cost});
  int m0 = edges.size();
  G[from].push_back(m0-2); G[to].push_back(m0-1);
}
bool spfa() {
   memset(dis, 63, sizeof(dis));
   memset(vis, 0,sizeof(vis));
   queue<int> q; q.push(S); dis[S]=0; vis[S]=1;
   while(!q.empty()){
      int u = q.front(); q.pop();
      for(int i=0; i<G[u].size(); i++) {
         Edge& e = edges[G[u][i]]; int v = e.to;
         if(e.cap && dis[v] > dis[u]+e.cost) {
            dis[v] = dis[u]+e.cost;
            path[v] = G[u][i];
            if(!vis[v]) { q.push(v); vis[v] = 1; }
         }
      }
      vis[u] = 0;
   }
   return dis[T]<INF; //0x3f3f3f3f
}
void CostFlow() {
   ansflow = anscost = 0;
```

```
    memset(path, 0, sizeof(path));
    while (spfa()) {
        int f=INF;
        for(int x=T;x!=S;x=edges[path[x]].from)
            f = min(f,edges[path[x]].cap);
        for(int x=T;x!=S;x=edges[path[x]].from) {
            edges[path[x]].cap -= f;
            edges[path[x]^1].cap += f;
        }
        anscost += (ll)dis[T]*(ll)f;
        ansflow += f;
    }
}
// cin>>S>>T; CostFlow();
// printf("%d %lld\n",ansflow,anscost);
```

// ============= ISAP =============

```
int source, sink, p[max_nodes], num[max_nodes],
cur[max_nodes], d[max_nodes];
bool visited[max_nodes];
struct Edge{
    int from, to, cap, flow;
    Edge(){}
    Edge(int a, int b, int c, int d):from(a), to(b), cap(c),
flow(d){}
};
int num_nodes, num_edges;
vector<Edge> edges;
vector<int> G[max_nodes]; // 每个节点出发的边编号
正向边
// 预处理, 反向 BFS 构造 d 数组
void bfs(){
    memset(visited, 0, sizeof(visited));
    queue<int> Q; Q.push(sink);
    visited[sink]=true; d[sink]=0;//距离为 0
    while(!Q.empty()){
        int u=Q.front(); Q.pop();
        for(auto ix=G[u].begin(); ix!=G[u].end(); ++ix){
            Edge& e=edges[(*ix)];//因为从汇点开始 用
反向边
            if(!visited[e.to])
                visited[e.to]=true, d[e.to]=d[u]+1,
Q.push(e.to);
        }
    }
}
int augment(){// 找到一条增广路 增广
    int u=sink, rsd=0x7ffffff;
    // 从汇点到源点通过 p 追踪增广路径, rsd 为一路
上最小的残量
    while(u!=source){
        Edge& e=edges[p[u]];
        rsd=min(rsd, e.cap-e.flow);
        u=edges[p[u]].from;
    }
    u=sink;
    while(u!=source){// 从汇点到源点更新流量
        edges[p[u]].flow+=rsd;   //正向边+流量 反向边-
流量
        edges[p[u]^1].flow-=rsd;
        u=edges[p[u]].from;
    }
    return rsd;
}
int max_flow(){
    int flow=0;
    bfs();
    memset(num, 0, sizeof(num));
    for(int i=0; i<num_nodes; i++)
        num[d[i]]++;  //和 t 距离为 i 的节点
    int u=source;               //当前节点
    memset(cur, 0, sizeof(cur));
    while(d[source]<num_nodes){//s 到 t 的距离不能
超过点数
        if(u==sink) flow+=augment(), u=source;
        bool advanced=false;
        for(int i=cur[u]; i<G[u].size(); i++){
        //当前弧优化 从第 cur 个点开始做 因为前 cur-
    1 个点都已经用干净了
            Edge& e=edges[G[u][i]]; //正边
            if(e.cap>e.flow&&d[u]==d[e.to]+1){//边上有
残量 并且是一条 允许弧
                advanced=true; p[e.to]=G[u][i];  //下一个
点的 上一条弧
                cur[u]=i;        //更新 u 点的当前弧到 u
的第 i 个点
                u=e.to; break;
            }
        }
        if(!advanced){
            // 当前(过时的)剩余网络下 u 不能允许弧连
接到 t 了
            // retreat 更新分层图
            // remark: u 的邻接边不一定都是允许弧
            // 所以更新到 u 邻接边的距离+1 是新的剩
余网络中的允许弧
            int m=num_nodes-1; //默认 u 的距离是最大
值(从剩余网络中排除)
            for(auto ix=G[u].begin(); ix!=G[u].end(); ++ix)
                if(edges[*ix].cap>edges[*ix].flow)
m=min(m, d[edges[*ix].to]);
            if(--num[d[u]]==0) break;//gap 优化, 如果和 t
距离 d[u]的所有点都没了 s 和 t 一定断开了 直接退
出
            d[u]=m+1, num[d[u]]++;
            cur[u]=0;
```

```
            if(u!=source) u=edges[p[u]].from;  //retreat to 到 u 的前一个点
        }
    }
    return flow;
}
```

// ========= 次小生成树 =========

```
// 增量最小 MST: (m 次加边求 MST)回路性质 加边后删除生成树以外的所有边
// 最小瓶颈 MST/路: (最大边最小) 原图 MST 满足瓶颈性质
// 次小生成树: 边 uv 和点 uv 之间的最小瓶颈(最大边权)边交换
int n, m, fa[MAXN], maxcost[MAXN][MAXN], pre[MAXN];
bool vis[MAXN];
struct Edge{
    int u, v, w, inMST;
    Edge(int u=0, int v=0, int dist=0):u(u), v(v), w(dist){inMST=0;}
}edge[MAXM];
vector<Edge> vec[MAXN];//MST
bool cmp(const Edge& a, const Edge& b){return a.w<b.w;}
int root(int x){return fa[x]==x?x:fa[x]=root(fa[x]);}
void kruskal(){
    sort(edge, edge+m, cmp);
    for(int i=1; i<=n; i++) fa[i]=i;
    int cnt=0;
    for(int i=0; i<m; i++){
        int x=root(edge[i].u), y=root(edge[i].v);
        if(x!=y){
            fa[y]=x;
            vec[edge[i].u].push_back(Edge(edge[i].u, edge[i].v, edge[i].w));
            vec[edge[i].v].push_back(Edge(edge[i].v, edge[i].u, edge[i].w));
            edge[i].inMST=1;
            if(++cnt==n-1) break;
        }
    }
}
void dfs(int u){
    vis[u]=1;
    for(int i=0; i<vec[u].size(); i++){
        int v=vec[u][i].v;
        if(!vis[v]){ //access a new node
            //u 是 v 的父亲 在有根树中
            for(int j=1; j<=n; j++) if(vis[j])//relax from all node visited

    maxcost[j][v]=maxcost[v][j]=max(maxcost[j][u], vec[u][i].w);//j->v = max(j->u,u->v)
            dfs(v);
        }
    }
}
int nextMST(){
    int i, ans=0x3f3f3f3f;
    for(i=0; i<m; i++){
        Edge e=edge[i];
        if(!e.inMST) ans=min(ans, e.w-maxcost[e.u][e.v]);
//ans 是边权增大了多少
    }
    return ans;
}
```

// ========= 曼哈顿最小生成树 =========

```
struct BIT {
   int min_val,pos;
   void init()
   {min_val=INF;pos=-1;}
} bit[maxn];
void update(int x,int val,int pos){
   for(int i=x;i>=1;i-=lowbit(i))
      if(val<bit[i].min_val)
         bit[i].min_val=val,bit[i].pos=pos;
}
int ask(int x,int m){
   int min_val=INF, pos=-1;
   for(int i=x;i<=m;i+=lowbit(i))
      if(bit[i].min_val<min_val)
         min_val=bit[i].min_val,pos=bit[i].pos;
   return pos;
}
int Manhattan_minimum_spanning_tree(int n,Point *p){
   int a[maxn],b[maxn]; // tmp
   for(int dir=0;dir<4;dir++){
      //4 种坐标变换
      if(dir==1||dir==3){
         for(int i=0;i<n;i++)
            swap(p[i].x,p[i].y);
      }
      else if(dir==2){
         for(int i=0;i<n;i++)
            p[i].x=-p[i].x;
      }
      // 我们将坐标按 X 排序(Y 为第二关键字), 将 Y-X 离散化
      // 用 BIT 来维护, 查询对于某一个(X0,Y0)
      // 查询比(Y0-X0)大的中 X1+Y1 最小的点
      sort(p, p + n);
      for(int i=0;i<n;i++){
         a[i]=b[i]=p[i].y-p[i].x;
      }
      sort(b, b + n);
```

```
    int m = unique(b,b+n)-b;
    for(int i=1;i<=m;i++)
      bit[i].init();
    //对于四种坐标变换 每次新建一个树状数组
    for(int i=n-1;i>=0;i--){ // x 从大到小 保证 X1>=X0
      int pos=lower_bound(b,b+m,a[i])-b+1;
      //BIT 中从 1 开始
      int ans=ask(pos,m);
      if(ans!=-1) // dist 函数计算的是曼哈顿距离
        addedge(p[i].id,p[ans].id,dist(i,ans));
      update(pos,p[i].x+p[i].y,i);
    }
  }
}
```

// ========= 最小树形图 （朱刘） =========

```
//如果没有 root, 加虚拟根， 和每个点权值是所有边权值和+1， 最后答案减去(和+1)
struct edge{
    int u, v, w;
    edge(int u=0, int v=0, int w=0):u(u), v(v), w(w){}
};
int n, m, ans;
vector<edge> g;
int id[maxn], inw[maxn], v[maxn], pre[maxn];
//inw:最小入边, v:一个点属于哪个环, id: 重新建图的点编号
bool zhuLiuAlg(int root){
    ans=0;
    while(true){
        for(int i=0; i<n; i++) inw[i]=INF, id[i]=-1, v[i]=-1, pre[i]=-1;
        for(int i=0; i<g.size(); i++)
            if(g[i].w<inw[g[i].v]&&g[i].v!=g[i].u)
                inw[g[i].v]=g[i].w, pre[g[i].v]=g[i].u;
        pre[root]=root, inw[root]=0;
        //判断是否可能, 因为后边修改了权值, 可以直接加到答案里
        for(int i=0; i<n; i++){
            if(inw[i]==INF) return false;
            ans+=inw[i];
        }
        //找圈 && 缩点， 缩成的点编号， 判断是否还有环
        int idx=0;
        for(int i=0; i<n; i++)
            if(v[i]==-1){
                int t=i;
                while(v[t]==-1) v[t]=i, t=pre[t];
                if(v[t]!=i||t==root) continue;
                id[t]=idx++;
                for(int j=pre[t]; j!=t; j=pre[j]) id[j]=idx-1;
            }
        if(idx==0) return true;  // 没有环了
        for(int i=0; i<n; i++) if(id[i]==-1) id[i]=idx++;
        // 重新建图
        for(int i=0; i<g.size(); i++)
            g[i].w-=inw[g[i].v], g[i].u=id[g[i].u], g[i].v=id[g[i].v];//减权值避免删边
        n=idx; root=id[root];
    }
}
```

// =========== 稳定婚姻 ===========

```
int n, m[maxn][maxn], wife[maxn], cur[maxn], w[maxn][maxn], hus[maxn];
queue<int> Q;
void solve(){
  while(!Q.empty()){
    int man=Q.front(); Q.pop();
    int woman=m[man][cur[man]++];
    if(!hus[woman]) hus[woman]=man, wife[man]=woman;//直接配对
    else if(w[woman][man]<w[woman][hus[woman]]){//如果当前男生的更好, 抛弃现在的舞伴， 重新配对
      wife[hus[woman]]=0; //被抛弃的男生重新回到单身状态
      Q.push(hus[woman]);
      hus[woman]=man, wife[man]=woman;//新的一对
    }else Q.push(man); //男生没人要
  }
  for(int i=1; i<=n; i++) printf("%d\n", wife[i]);
}
int main(){
  int T; scanf("%d", &T);
  while(T--){
    while(!Q.empty()) Q.pop();
    scanf("%d", &n);
    for(int i=1; i<=n; i++){
      for(int j=1; j<=n; j++) scanf("%d", &m[i][j]); //编号为 i 的男生第 j 喜欢的女生
      cur[i]=1, wife[i]=0, Q.push(i);//男生 i 下一个要邀请对象, 男生 i 的舞伴编号
    }
    int x;
    for(int i=1; i<=n; i++){
      for(int j=1; j<=n; j++) scanf("%d", &x), w[i][x]=j; //女生 i 心目中， 男生 x 的排名
      hus[i]=0; //编号为 i 的女生的舞伴编号
    }
    solve(); if(T) puts("");
  }
  return 0;
}
```

// =========== KM 最大权完全匹配 ===========

```
//最小点覆盖 = 最大匹配
//最大独立集 = 最小边覆盖 = 点数 - 最大匹配
//最大团 = 补图的最大独立集
//最小路径覆盖: 原图拆点 = 点数 - 拆点图最大匹配
//求最小权完备匹配:所有的边权值取其相反数，求最大权完备匹配，匹配的值再取相反数
//KM 算法的运行要求是必须存在一个完备匹配，如果求一个最大权匹配(不一定完备):把不存在的边权值赋为 0。
//求边权之积最大: 每条边权取自然对数，然后求最大和权匹配，求得的结果 a 再算出 e^a 就是最大积匹配
int G[MAXN][MAXN], ex_girl[MAXN], ex_boy[MAXN], match[MAXN], slack[MAXN], N, n, m;
bool vis_girl[MAXN], vis_boy[MAXN];
bool dfs(int girl){
    vis_girl[girl]=true;
    for(int boy=0; boy<N; ++boy){
        if(vis_boy[boy]) continue; // 每一轮匹配 每个男生只尝试一次
        int gap=ex_girl[girl]+ex_boy[boy]-G[girl][boy];
        if(gap==0){  // 如果符合要求
            vis_boy[boy]=true;
            if(match[boy]==-1||dfs(match[boy])){match[boy]=girl;return true;}// 找到一个没有匹配的男生 或者该男生的妹子可以找到其他人
        }else slack[boy]=min(slack[boy], gap);  // slack 可以理解为该男生要得到女生的倾心 还需多少期望值 取最小值 备胎的样子
    }
    return false;
}
int KM(){
    memset(match, -1, sizeof match); memset(ex_boy, 0, sizeof ex_boy);
    for(int i=0; i<N; ++i){// 每个女生的初始期望值是与她相连的男生最大的好感度
        ex_girl[i]=G[i][0];
        for(int j=1; j<N; ++j)
            ex_girl[i]=max(ex_girl[i], G[i][j]);
    }
    for(int i=0; i<N; ++i){// 尝试为每一个女生解决归宿问题
        fill(slack, slack+N, INF);// 因为要取最小值 初始化为无穷大
        while(1){
        // 为每个女生解决归宿问题的方法是 ：如果找不到就降低期望值，直到找到为止
        // 记录每轮匹配中男生女生是否被尝试匹配过
            memset(vis_girl, false, sizeof vis_girl);
            memset(vis_boy, false, sizeof vis_boy);
            if(dfs(i)) break;  // 找到归宿 退出
            // 如果不能找到 就降低期望值
            // 最小可降低的期望值
            int d=INF;
            for(int j=0; j<N; ++j)
                if(!vis_boy[j]) d=min(d, slack[j]);
            for(int j=0; j<N; ++j){
                if(vis_girl[j]) ex_girl[j]-=d;
// 所有访问过的女生降低期望值
                if(vis_boy[j]) ex_boy[j]+=d;
// 所有访问过的男生增加期望值
                else slack[j]-=d;
// 没有访问过的 boy 因为 girl 们的期望值降低，距离得到女生倾心又进了一步！
            }
        }
    }
    int res=0;// 匹配完成 求出所有配对的好感度的和
    for(int i=0; i<N; ++i) res+=G[match[i]][i];
    return res;
}
```

// ========= sublime 配置 =========

```
{
    "cmd": ["g++","-g","-O2","-std=gnu++14","-static","${file}", "-o", "${file_path}/${file_base_name}"],
    "file_regex": "^(..[^:]*):([0-9]+):?([0-9]+)?:? (.*)$",
    "working_dir": "${file_path}",
    "selector": "source.c, source.c++",
    "variants":
    [
        {
            "name": "Run",
            "cmd": ["x-terminal-emulator", "-e", "bash", "-c", "g++ -g -O2 -std=gnu++14 -static '${file}' -o '${file_path}/${file_base_name}' && '${file_path}/${file_base_name}';echo;echo; read -p 'Press any key to continue...'"]
        }
    ]
}
```