EGOI 2024 - Day 2

Discussione e implementazione delle possibili soluzioni ai problemi di una gara internazionale di informatica

Sofia Maragò

Agosto 2024 (rev. Dicembre 2024)

Contents

1 Nota introduttiva **2**

2 A - Circle passing **3**
2.1 Testo originale del problema . . . 3
2.2 Risoluzione . . . 4
2.2.1 Soluzione completa . . . 4

3 B - Bikeparking **6**
3.1 Testo originale del problema . . . 6
3.2 Risoluzione . . . 7
3.2.1 $x_i = x_j = y_i = y_j$ (subtask 2) . . . 7
3.2.2 $N, x, y \leq 100$ e $x_i, y_i \leq 1$(subtask 3, 4) . . . 8
3.2.3 Soluzione completa . . . 9

4 C - Lightbulbs **11**
4.1 Testo originale del problema . . . 11
4.2 Risoluzione . . . 13
4.2.1 $N \leq 10$ (subtask 1, 2) . . . 13
4.2.2 $Q \leq 800$ (subtask 3, 38 punti) . . . 14
4.2.3 Soluzione completa . . . 16

5 D - Make them meet **17**
5.1 Testo originale del problema . . . 17
5.2 Risoluzione . . . 18
5.2.1 Grafo a stella (subtask 1) . . . 18
5.2.2 Grafo a linea (subtask 3) . . . 19
5.2.3 Grafo completo (subtask 2) . . . 20
5.2.4 Grafo ad albero (subtask 4) . . . 20
5.2.5 Soluzione completa . . . 22

1 Nota introduttiva

In questa sezione illustrerò delle considerazioni generali sulla natura di questo elaborato e sul contenuto dello stesso.

Da due anni ormai le gare di informatica, una disciplina nota internazionalmente come competitive programming, sono divenute una parte molto importante della mia vita. Tra seconda e terza ho avuto modo di appassionarmi di questa materia e partecipare ad alcune gare internazionali con la squadra italiana (EGOI 2023 e 2024, WEOI 2024) e lussemburghese (IOI 2024).

Questa dispensa nasce soprattutto dall'esigenza di preparare la gara più importante a cui abbia partecipato fino ad oggi (IOI24). Lo scopo è quindi quello di riflettere su una gara a cui ho partecipato quest'anno e pertanto indagare a fondo i problemi e le loro strategie risolutive, cercando di trovare approcci che mi sono sfuggiti in gara, autonomamente, e confrontandomi in qualche caso anche con le soluzioni ufficiali.

Verranno dunque presentati quattro problemi, che formavano il secondo giorno di gara delle EGOI 2024 (svoltesi a Veldhoven, 21-27/07/2024). Per ogni problema si avrà il testo completo, (riportato in italiano e completo di subtask e limitazioni dell'input) seguito da una possibile traccia risolutiva seguita in gara o elaborata a posteriori. La soluzione sarà articolata in una parte descrittiva dell'approccio seguito, una breve dimostrazione (una semplice motivazione, non eccessivamente rigorosa) e una possibile implementazione in c++ (il codice sarà in formato da gara dunque con più enfasi su efficenza e velocità, soprattutto in scrittura, che mantenibilità effettiva, per maggiore chiarezza corredato in ogni caso da commenti esplicativi).

I testi completi e le soluzioni ufficiali possono essere trovati sul sito ufficiale di questa edizione delle EGOI, oppure sul sito del comitato organizzativo generale delle stesse, nella sezione tasks (in versione sia italiana che inglese), mentre i problemi stessi, con la possibilità di sottoporre codice per la valutazione, possono essere trovati sulla piattaforma Kattis.

2 A - Circle passing

2.1 Testo originale del problema

È il primo giorno di liceo per Anouk; la sua insegnante di ginnastica fa fare alla classe giochi per imparare i nomi, come attività di riscaldamento. Ci sono $2N$ studenti nella classe. La maggior parte di loro non si conosce, ma ci sono M coppie di migliori amici che fanno tutto insieme. Ogni studente ha al massimo un migliore amico.

L'insegnante dispone tutti gli studenti in cerchio, assegnando consecutivamente a ciascuno studente un numero compreso tra 0 e $2N-1$. Più specificamente, per ogni $0 \leq i < 2N-1$, gli studenti i e $i+1$ stanno uno accanto all'altro. Inoltre, anche gli studenti 0 e $2N-1$ stanno uno accanto all'altro.

Poiché l'insegnante vuole che tutti conoscano nuovi studenti, i migliori amici devono stare il più lontano possibile l'uno dall'altro. Ciò significa che se uno studente i (con $0 \leq i < N$) ha un migliore amico, allora il migliore amico si trova sul lato opposto nella posizione $i+N$.

Nel gioco l'insegnante seleziona due studenti x e y e passa una palla allo studente x. L'obiettivo è far arrivare la palla allo studente y, ma ogni studente può passare la palla solo a uno studente di cui conosce già il nome. Ovviamente, i migliori amici si conoscono per nome. Mentre venivano spiegate le regole, ogni studente ha imparato i nomi dei due studenti che stanno accanto a lui. A parte questo, nessuno conosce altri nomi.

Il gioco viene ripetuto Q volte, e ogni volta l'insegnante sceglie due nuovi studenti. Poiché gli studenti non sono stati troppo attenti, non hanno imparato nuovi nomi durante il gioco. Qual è il numero minimo di passaggi necessari per far arrivare la palla dallo studente x allo studente y, in ogni partita?

Input e output: La prima riga di input contiene tre numeri interi, N , M e Q, dove $2N$ è il numero di studenti nella classe di Anouk, M è il numero di coppie di migliori amici e Q è il numero di partite giocate.

La seconda riga contiene M numeri interi $k_0, ..., k_{M-1}$, dove k_i descrive la i-esima coppia di migliori amici. Per ogni i, i migliori amici si trovano rispettivamente nelle posizioni k_i e $k_i + N$.

Le seguenti Q righe contengono ciascuna due numeri interi, x_i e y_i , i due studenti selezionati nella partita i.

Devi restituire Q interi su altrettante righe, la i-esima riga deve contenere un singolo numero intero, il numero minimo di passaggi necessari nella partita i.

Limiti:

- $2 \leq N \leq 5 \cdot 10^8$
- $1 \leq M \leq 5 \cdot 10^5$ e $M \leq N$
- $1 \leq Q \leq 2 \cdot 10^4$
- $0 < k_0 < k_1 < ... < k_{M-1} < N$
- $0 \leq x_i, y_i < 2 \cdot N$ e $x_i \neq y_i$.
- (tempo massimo di esecuzione: 2 secondi)

Punteggio e subtask: Il punteggio totale è la somma dei punteggi ottenuti per ogni gruppo di test, cioè ogni subtask. Il punteggio per un gruppo di test è assegnato se la risposta è corretta e l'esecuzione ha durata minore del limite di tempo per ogni test che ne fa parte. Alcune subtask hanno ulteriori limitazioni specifiche come dalla seguente tabella:

Subtask	Punteggio massimo	Limiti
1	14	$M = 1$ e $x = k$. In altre parole, esiste una sola coppia di migliori amici e in ogni gioco lo studente che inizia con la palla ha un migliore amico
2	20	$N, M, Q \leq 1000$
3	22	$N \leq 10$ e $M, Q \leq 1000$
4	17	$x = 0$ per tutti i
5	27	nessuna limitazione aggiuntiva

2.2 Risoluzione

A differenza degli altri problemi presentati, essendo questo problema di minore complessità (ed essendo stato risolto durante la gara) verrà presentata solamente la soluzione completa, senza l'analisi delle idee parziali per ogni subtask.

2.2.1 Soluzione completa

Idea: La considerazione fondamentale che porta alla soluzione di questo problema è che il minor numero di passaggi può essere ottenuto solamente nei seguenti modi:

- senza utilizzare un lancio tra migliori amici, percorrendo cioè il cerchio per posizioni adiacenti, in un verso o nell'altro.
- utilizzando un singolo lancio tra migliori amici, scelto tra i due più vicini (che possono anche coincidere o non esistere) al ragazzo x.

Essendo le possibili opzioni da considerare poche, si può procedere verificando ad una ad una se consentono di ottenere il minimo numero di passaggi. Per il primo caso è sufficiente una sottrazione delle posizioni, il secondo va invece ottimizzato, in quanto una ricerca lineare della prima coppia di migliori amici prima e dopo una data posizione è troppo dispendiosa per i limiti del problema. Questa ottimizzazione può essere implementata a mano con una ricerca per bisezione o utilizzando una struttura dati presente nella standard library del c++, il set.

Dimostrazione: In questo problema bisogna fondamentalmente dimostrare che utilizzare più passaggi tra migliori amici, o utilizzare un passaggio tra migliori amici più distante dalla posizione iniziale è sconveniente.

La prima di queste affermazioni si dimostra considerando che un passaggio fra migliori amici è equivalente a spostarsi di N sul cerchio, quindi esssendo che il cerchio è costituito da $2N$ posizioni, svolgere un numero pari di questi passaggi è equivalente a non svolgerne nessuno, mentre svolgerne un numero dispari è equivalente a svolgerne esattamente uno.

La seconda affermazione invece si dimostra considerando cosa succederebbe se si utilizzasse un passaggio più distante dalla posizione di partenza:

- finchè x e y si trovano sullo stesso semicirconferenza descritto dal segmento che unisce le due posizioni di migliori amici, spostare la coppia di migliori amici non cambia il numero di passaggi necessari.
- se invece x e y si trovano su semicirconferenze diverse, allontanare la coppia di migliori amici da x (e di conseguenza anche da y per simmetria) significa aumentare il numero di passaggi adiacenti necessari di 2 per ogni posizione sulla circonferenza.

Pertanto l'idea abbozzata precedentemente è corretta, la complessità di questa soluzione è $\mathcal{O}(Q \log M)$. Per ogni query infatti si calcola la distanza tra due punti un numero costante di volte, e si determina in $\mathcal{O}(\log M)$ la coppia di amici più vicina al ragazzo x una volta. Considerando i limiti posti sui dati in input questa soluzione è sufficentemente efficiente.

Implementazione: Un possibile modo di implementare l'idea discussa:

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

long long calcola_distanza(long long pi, long long pf) {
    // Questa funzione ritorna la minima distanza tra due punti sulla
    // circonferenza (in posizioni pi e pf) utilizzando solo passaggi
    // tra posizioni adiacenti
    pi += 2*N; pf += 2*N; // normalizzo possibili numeri negativi
    pi %= 2*N; pf %= 2*N; // normalizzo possibili numeri >=2N
    return min(abs(pi-pf), 2*N-abs(pi-pf));
}

int main() {
    long long M, Q; cin >> N >> M >> Q;
    set<long long> ca; //lista ordinata di posizioni che hanno un migliore amico
    for (int i = 0; i < M; i++) {
        int a; cin >> a;
        ca.insert(a); ca.insert(a+N); //inserisco la coppia di amici
    }
    // inserisco due posizioni aggiuntive che costituiscono la coppia
    // precedente alla prima posizione e quella successiva all'ultima
    // posizione.
    long long pr = *ca.begin(), lt = *ca.rbegin();
    ca.insert(2*N+pr); ca.insert(lt-2*N);
    for (int i = 0; i < Q; i++) {
        long long x, y; cin >> x >> y;
        // seguono tre calcoli di distanze puri, processando lo scambio
        // tra amici separatamente e usando la funzione definita sopra
        // il minimo numero di passaggi è memorizzato in ma e stampato
        long long ma = calcola_distanza(x, y); //senza migliori amici
        long long l = *ca.lower_bound(x); // trovo la prima coppia precedente
        ma = min(ma, 1+calcola_distanza(x, l)+calcola_distanza(l+N, y));
        long long r = *prev(ca.lower_bound(x)); //trovo la prima coppia successiva
        ma = min(ma, 1+calcola_distanza(x, r)+calcola_distanza(r+N, y));
        cout << ma << "\n";
    }
}
```

3 B - Bikeparking

3.1 Testo originale del problema

Sanne ha recentemente concepito un'idea imprenditoriale redditizia: affittare un parcheggio per biciclette premium presso la stazione ferroviaria di Eindhoven.

Per massimizzare i suoi profitti, ha diviso i parcheggi per biciclette in N diverse zone, numerate da 0 e $N-1$. La zona 0, la zona premium, si trova molto vicino ai binari del treno. Le zone con numero più alto sono costituite da parcheggi di qualità più bassa (più alto è il numero, peggiore è il parcheggio). Il numero di parcheggi nella zona t è x_t.

Agli utenti che parcheggiano le biciclette viene assegnato un parcheggio tramite un'app. ogni utente ha un livello di abbonamento che prevederebbe un parcheggio nella zona corrispondente. Tuttavia, i termini di servizio non garantiscono agli utenti un parcheggio in quella zona.

Se a un utente con livello di abbonamento s viene assegnato un parcheggio nella zona t, si verifica una delle tre situazioni seguenti:

1. Se $t < s$, l'utente sarà felice e darà un voto positivo all'app.
2. Se $t = s$, l'utente sarà soddisfatto e non fara nulla.
3. Se $t > s$, l'utente si arrabbierà e darà unn voto negativo all'app.

Oggi, l'app di Sanne ha $y_0 + y_1 + ... + y_{N-1}$ utenti, dove y_s è il numero di utenti con livello di abbonamento s e Sanne ha bisogno del tuo aiuto per assegnare gli utenti ai parcheggi. Ad ogni utente verrà assegnato esattamente un parcheggio per la sua bici. Nessun parcheggio può essere assegnato a più di un utente, ma è accettabile che alcuni parcheggi non vengano assegnati a nessun utente.

Sanne vuole massimizzare la valutazione della sua app. Sia U il numero di voti positivi e D il numero di voti negativi. Il tuo compito è massimizzare $U - D$.

Input e output: La prima riga di input contiene un numero inntero N, il numero di zone e dei livelli di abbonnamento.

La seconda riga contiene N numeri interi $x_0, ..., x_{N-1}$, il numero di parcheggi nelle diverse zone.

La terza riga contiene N numeri interi $y_0, ..., y_{N-1}$, il numero di utenti con ciascun livello di abbonamento.

Devi restituire un numero intero, il valore massimo possibile di $U - D$, assegnando in modo ottimale gli utenti ai parcheggi.

Limiti:

- $1 \leq N \leq 3 \cdot 10^5$.
- $0 \leq x_i, y_i \leq 10^9$ per $i = 0, 1, ..., N-1$.
- $y_0 + y_1 + ... + y_{N-1} \leq x_0 + x_1 + ... + x_{N-1} \leq 10^9$.
- (tempo massimo di esecuzione: 1 secondo)

Punteggio e subtask: Il punteggio totale è la somma dei punteggi ottenuti per ogni gruppo di test, cioè ogni subtask. Il punteggio per un gruppo di test è assegnato se la risposta è corretta e l'esecuzione ha durata minore del limite di tempo per ogni test che ne fa parte. Alcune subtask hanno ulteriori limitazioni specifiche come dalla seguente tabella:

Subtask	Punteggio massimo	Limiti
1	16	$N = 2$ e $x_i, y_i \leq 100$
2	9	$x_i = x_j = y_i = y_j$ per tutti i, j. In altre parole, tutti i x e y nell'input sono uguali
3	19	$x_i, y_i \leq 1$
4	17	$N, x_i, y_i \leq 100$
5	32	nessuna limitazione aggiuntiva

3.2 Risoluzione

In questo e nei restanti problemi, la ricerca della soluzione sarà affrontata sfruttando i suggerimenti dati dalle subtask, che sono fornite per guidare alla soluzione e consentire l'attribuzione di punteggi parziali in base al progresso svolto sul problema. Le subtask saranno affrontate in un ordine basato sulla loro difficoltà e progressivo avvicinamento alla soluzione finale.

3.2.1 $x_i = x_j = y_i = y_j$ (subtask 2)

Idea: La subtask 1, può essere molto facilmente risolta provando tutti i casi possibili e implementando i controlli, sarà quindi omessa.

Per la seconda subtask invece è sufficiente restituire $x_0 \cdot (N-2)$, che corrisponde ad assegnare un parcheggio migliore a tutti gli utenti meno quelli che avevano l'abbonamento premium che saranno mandati nel parcheggio peggiore. L'unico caso da gestire separatamente è ovviamente $N = 1$ dove si può ottenere solamente 0.

Dimostrazione: È evidente che gli utenti con abbonamento y_0 non potranno mai dare voti positivi, se teniamo i loro voti nulli lo stesso si può dire per chi ha abbonamento y_1, e così via. Risulta quindi necessario avere dei voti negativi per ottenere un bilancio non nullo. Conviene avere voti negativi dagli utenti con abbonamento migliore perché il voto non cambia in base alla differenza di qualità dall'abbonamento acquistato e in questo modo ogni altro utente può dare un voto positivo.

Implementazione: Un possibile modo di implementare l'idea discussa:

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

int main() {
    int N; cin >> N;
    vector<int> x(N), y(N);
    for (int i = 0; i < N; i++) cin >> x[i];
    for (int i = 0; i < N; i++) cin >> y[i];
    // calcolo e stampo semplicemente la risposta
    cout << max(0, x[0]*(N-2));
}
```

3.2.2 $N, x, y \leq 100$ e $x_i, y_i \leq 1$ (subtask 3, 4)

Idea: Questo problema si risolve con una tecnica detta *greedy*, cioè in cui l'algoritmo risolutivo consiste in scelte ottime locali che risultano sempre anche delle scelte ottime globali. Come spesso accade con questo tipo di problemi, si possono trovare molti approci che funzionano in casi particolari e limitati (come la subtask 2), verrà però presentato un approccio che porta più direttamente alla soluzione completa.

Una possibile idea che risolve questa subtask costituisce la soluzione non ottimizzata in complessità del problema. In particolare si può dimostrare che assegnare tutti gli utenti possibili in modo da renderli felici, poi ripetere con quelli per cui è stato impossibile per renderli soddisfatti e infine assegnare i rimanenti è l'assegnamento migliore ottenibile. Per questa subtask è sufficente implementare quest'idea così come è stata descritta in complessità $\mathcal{O}(\sum_{i=0}^{N} x_i + y_i)$

Dimostrazione: Per dimostrare la correttezza di quest'idea si suppone di avere una soluzione ottima per un qualche sottoinsieme di utenti e parcheggi. Si prende quindi un utente a non ancora assegnato e un parcheggio $b < a$ non ancora assegnato, in modo che non ci siano utenti o parcheggi non ancora assegnati nell'intervallo (b, a). Si dimostra ora che la soluzione migliore che si può avere è assegnare il parcheggio b ad a, lasciando ogni altro assegnamento nella soluzione invariato. Si suppone per assurdo di avere un'altra coppia di utente e parcheggio, (c, d), facenti parte della soluzione parziale, in modo che sia più conveniente assegnare a a d e b a c (con ovviamente $a \neq c$ e $b \neq d$, altrimenti scambiare è equivalente a non scambiare). Qui è necessario analizzare ogni caso:

- se $b < a < d$ scambiando, qualsiasi cosa succeda con c, il voto di a, passa da positivo a negativo quindi è impossibile che scambiare risulti migliore
- se $d < b < a < c$ scambiando l'assegnamento si ottiene lo stesso valore
- se $c < d < b < a$ scambiato l'assegnamento si ottiene lo stesso valore
- se $d < c < b < a$ scambiando l'assegnamento si ottiene -2
- se $b < a = d$ scambiando, il voto di c può cambiare solo se $b = c$, non potendosi trovare tra a e b (quindi tra d e b), mentre il voto di a diminuisce, quindi si ottiene al massimo lo stesso valore
- se $d < b = c < a$ scambiando l'assegnamento si ottiene -1
- se $c = d < b < a$ scambiando l'assegnamento si ottiene -1
- se $b < a < c = d$ scambiando l'assegnamento si ottiene -1

Quindi avendo analizzato ogni possibile configurazione si può concludere che finchè non ci sono utenti e parcheggi non ancora assegnati, ad un utente con abbonamento a può essere assegnato il parcheggio b in una soluzione ottima. Detto in altre parole, possiamo assegnare ad ogni utente il primo parcheggio disponibile, se esiste, che gli permetta di dare un voto positivo. Poi naturalmente si assegnano i voti nulli, per minimizzare quelli negativi rimanenti.

Implementazione: Un possibile modo di implementare l'idea discussa:

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

int main() {
    int N; cin >> N;
    vector<int> x(N), y(N);
    for (int i = 0; i < N; i++) cin >> x[i];
    for (int i = 0; i < N; i++) cin >> y[i];
```

```
    stack<int> la; // strttura dati lifo che mantiene le posizioni in cui
    // sono rimasti posti disponibili
    int tot = 0; // la risposta finale verrà salvata in questa variabile
    for (int i = 0; i < N; i++) {
        // assegno tutti gli utenti che possono dare un voto positivo
        // aggiornando man mano lo stack di posizioni disponibili
        while (!la.empty() && y[i] > 0) {
            tot++;
            y[i]--;
            x[la.top()]--;
            if (x[la.top()] == 0) la.pop();
        }
        if (x[i] > 0) la.push(i);
    }
    for (int i = 0; i < N; i++) {
        // assegno tutti gli utenti che possono dare un voto neutro
        while (y[i] > 0 && x[i] > 0) {
            x[i]--; y[i]--;
        }
        // assegno infine gli utenti che daranno un voto negativo
        // (è assicurato nelle limitazioni che troveranno un posto
        // non è importante quale)
        tot -= y[i];
    }
    cout << tot;
}
```

3.2.3 Soluzione completa

Idea: L'idea è analoga alla subtask precedente, ma l'algoritmo va ottimizzato fino ad una complessità di $\mathcal{O}(N)$, per fare ciò è sufficiente processare $y[i]$ e $x[i]$ a gruppi di individui con le stesse proprietà invece che singolarmente.

Dimostrazione: Rimane da dimostrare solamente che la complessità processando correttamente gruppi di $x[i]$ e $y[i]$ è effettivamente $\mathcal{O}(N)$. Nel primo step ad ogni iterazione si fa in modo di esaurire almeno una qualità di posti parcheggio o un costo di abbonamento, che dunque, sommati, sono al massimo $2N$, nel secondo step similmente tutti i posti in $x[i]$ vengono esauriti mentre il terzo step era già svolto in complessità corretta, pertanto l'algoritmo è corretto e ottimizzato ai fini del problema.

Implementazione: Un possibile modo di implementare l'idea discussa:

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

int main() {
    int N; cin >> N;
    vector<int> x(N), y(N);
    for (int i = 0; i < N; i++) cin >> x[i];
    for (int i = 0; i < N; i++) cin >> y[i];
    stack<int> la; // posizioni in cui sono rimasti posti disponibili
    int tot = 0;
    for (int i = 0; i < N; i++) {
        while (!la.empty() && y[i] > 0) {
            // assegno gli utenti più efficacemente per gruppo di utenti
            // dello stesso abbonamento assegnati allo stesso parcheggio
            int as = min(x[la.top()], y[i]); // numero di utenti assegnati
            tot += as;
            y[i] -= as;
            x[la.top()]-= as;
            if (x[la.top()] == 0) la.pop();
        }
```

```
        if (x[i] > 0) la.push(i);
    }
    for (int i = 0; i < N; i++) {
        // allo stesso modo di prima assegno i neutri e i negativi, nuovamente
        // ottimizzando tutti i neutri in blocco
        y[i] -= x[i];
        tot -= max(0, y[i]);
    }
    cout << tot;
}
```

4 C - Lightbulbs

4.1 Testo originale del problema

Poco dopo aver fondato la sua azienda di lampadine a Eindhoven nel 1891, Frederik Philips fece una grande scoperta: le lampadine che illuminano un raggio infinito in direzione orizzontale o verticale. Con questa nuova scoperta, vuole rivoluzionare l'interior design delle case moderne.

Frederik progetta un'installazione elaborata con suo figlio Gerard. Installano N^2 lampadine in una griglia $N \times N$ in una stanza. Tuttavia vogliono illuminare l'intera stanza con il minor numero possibile di lampadine accese per risparmiare elettricità. Ogni lampadina è orizzontale, nel senso che illumina tutti i quadrati della sua riga, o verticale, nel senso che illumina tutti i quadrati della sua colonna.

Il disegno 1 mostra un esempio di una lampadina verticale (sinistra) e orizzontale (destra).

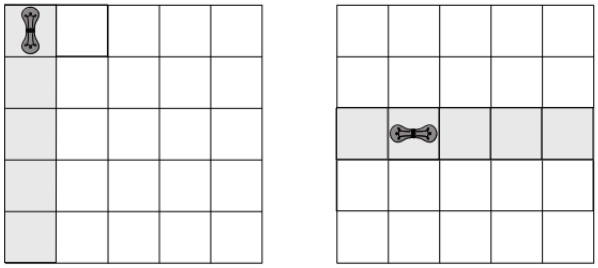

Figure 1: Schema di funzionamento delle lampadine

Sfortunatamente, non erano abbastanza concentrati durante l'installazione delle lampadine e non ricordano quali lampadine si accendono in orizzontale q quali in verticale. Conducono invece alcuni esperimenti per capire quali lampadine utilizzare per illuminare l'intera stanza. Gerard resta nella stanza con le lampadine, mentre Frederik aziona gli interruttori da un'altra stanza.

In ogni esperimento Frederik accende alcune lampade e Gerard segnala quanti quadrati sono illuminati in totale; un quadrato illuminato da due o più lampadine separate viene conteggiato una sola volta. Non importa quante lampadine vengano accese durante gli esperimenti, ma hanno fretta e idealmente vogliono condurre il minor numero di esperimenti possibile.

Il tuo compito è preparare gli esperimenti in modo da trovare una disposizione delle lampadine che illumini l'intera stanza e utilizzi il minor numero di lampadine. Possono condurre al massimo 2.000 esperimenti. Tuttavia, otterrai un punteggio più alto se utilizzeranno meno esperimenti.

Interazione: Questo è un problema interattivo. Il tuo programma deve leggere una riga con un numero intero N, l'altezza e la larghezza della griglia. Poi, il tuo programma deve interagire con il valutatore.

Per condurre un esperimento, devi prima stampare una riga con un punto interrogativo "?". Nelle seguenti N righe, stampa una griglia $N \times N$ di 0 e 1, indicando quali lampadine devono essere spente (0) o accese (1). Poi il tuo programma deve leggere un singolo numero intero l ($0 \leq l \leq N$), il numero di quadrati della griglia illuminati accendendo le lampadine specificate.

Quando sei pronto a rispondere con la configurazione finale, stampa una riga con un punto esclamativo "!", seguita da N righe con la griglia nello stesso formato di cui sopra.

Affinché la tua risposta venga considerata corretta, le lampadine devono illuminare tutta la griglia e il numero di lampadine accese deve essere il minimo possibile. Successivamente, il tuo programma deve terminare.

Il grader non è adattivo, il che significa che la griglia di lampadine viene determinata completamente prima dell'inizio dell'interazione.

Limiti:

- $3 \leq N \leq 100$
- si possono eseguire al massimo 2000 esperimenti.
- (tempo massimo di esecuzione: 4 secondi)

Punteggio e subtask: Il punteggio totale è la somma dei punteggi ottenuti per ogni gruppo di test, cioè ogni subtask. Il punteggio per un gruppo di test è assegnato se la risposta è corretta e l'esecuzione ha durata minore del limite di tempo per ogni test che ne fa parte. Alcune subtask hanno ulteriori limitazioni specifiche come dalla seguente tabella:

Subtask	Punteggio massimo	Limiti
1	11	$N = 3$
2	11	$N \leq 10$
3	78	nessuna limitazione aggiuntiva

Nella subtask 3, il punteggio P è assegnato in base al numero Q di query effettuate, secondo la seguente formula:

$$P = \begin{cases} (2000 - Q) \cdot \frac{29}{900} & \text{se } 200 \leq Q \leq 2000 \\ 58 + (200 - Q) \cdot \frac{4}{23} & \text{se } 85 \leq Q \leq 200 \\ 78 & \text{se } Q \leq 85 \end{cases}$$

Il grafico 2 mostra l'andamento di questa funzione.

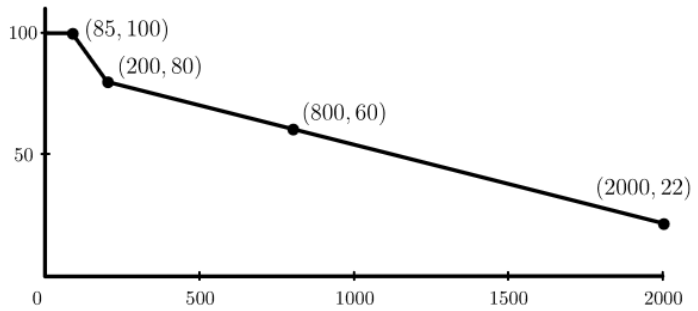

Figure 2: Punteggio sull'intero problema in funzione del massimo numero di query utilizzate

4.2 Risoluzione

4.2.1 $N \leq 10$ (subtask 1, 2)

Idea: I limiti sulla dimensione della griglia consentono di controllare il verso di ogni lampadina. Per prima cosa si trova il verso di una lampadina arbitraria (per facilità quella in posizione (0, 0)), accendendola assieme ad ogni altra lampadina della sua riga, una per volta. Se si ottiene un risultato pari a 2*N almeno una volta, la lampadina è verticale, se si ottiene un risultato di N almeno una volta, è orizzontale, se non si ottiene mai uno di questi risultati è sufficente confrontare un'altra coppia qualsiasi (dato che il verso di tutte le restanti lampadine di questa riga è diverso da quello della prima) per dedurre il suo verso.

Si procede poi confrontando ogni lampadina della griglia, una per volta con la lampadina scelta, se si ottiene N o 2N, la lampadina in questione ha lo stesso verso di quella scelta, altrimenti i raggi si intersecano in esattamente una casella quindi le lampadine hanno versi diversi. Questo approccio richiede nel caso peggiore 111 query, numero che si potrebbe ottimizzare ancora ma ai fini di queste subtask non è necessario.

Il numero minimo di lampadine da accendere per illuminare la griglia è sempre N. Infatti o in ogni riga c'è una lampadina orizzontale, oppure la riga è composta da sole lampadine verticali, per cui è sufficiente accenderla completamente per coprire la griglia.

Implementazione: Un possibile modo di implementare l'idea discussa:

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

int chiedi_griglia(vector<vector<bool>> &g) {
    // funzione ausiliare che esegue l'interazione stampando la query
    // e ricevendo l'input come specificato nel testo
    cout << "?\n";
    for (auto i : g) {
        for (auto j : i) cout << j << " ";
        cout << "\n";
    }
    fflush(stdout);
    int r; cin >> r;
    return r;
}
void rispondi_griglia(vector<vector<bool>> &g) {
    // funzione ausiliare che stampa il risultato
    cout << "!\n";
    for (auto i : g) {
        for (auto j : i) cout << j << " ";
        cout << "\n";
    }
    fflush(stdout);
}

int main() {
    int N; cin >> N;
    // griglia che verrà popolata man mano che si ottengono informazioni
    // sulle lampadine attraverso l'interazione
    vector<vector<bool>> fin(N, vector<bool> (N));
    // 0 indica una lampadina orizzontale, 1 una lampadina verticale

    bool f = 0; //variabile che controlla se la prima riga è composta da
    // lampadine tutte nella stessa direzione a meno della prima, cioè se
    // il verso della prima non si può stabilire per soli confronti con altre
    for (int i = 1; i < N; i++) {
        vector<vector<bool>> g(N, vector<bool> (N, 0));
        g[0][0] = 1; g[0][i] = 1;
        int r = chiedi_griglia(g);
        if (r % N != 0) continue;
        // posso stabilire il verso della prima
```

```
            if (r == N) fin[0][0] = 0;
            else fin[0][0] = 1;
            f = 1; break;
        }
        if (!f) {
            // trovo il verso della prima confrontando due altre lampadine nella
            // riga (ora si è certi debbano essere uguali)
            vector<vector<bool>> g(N, vector<bool> (N, 0));
            g[0][1] = 1; g[0][2] = 1;
            int r = chiedi_griglia(g);
            if (r == N) fin[0][0] = 1;
            else fin[0][0] = 0;
        }
        // itero su tutte le altre lampadine per determinarne la direzione nel
        // confronto con la prima
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (i == 0 && j == 0) continue;
                vector<vector<bool>> g(N, vector<bool> (N, 0));
                g[0][0] = 1; g[i][j] = 1;
                int r = chiedi_griglia(g);
                if (r % N != 0) fin[i][j] = !fin[0][0];
                else fin[i][j] = fin[0][0];
            }
        }
        // determino quali N lampadine devono essere accese nella risposta
        // se tutta una riga è composta da verticali rispondo quella
        // altrimenti ne trovo una orizzontale e la segno nel vettore c
        vector<int> c(N);
        for (int i = 0; i < N; i++) {
            f = 0;
            for (int j = 0; j < N; j++) {
                if (fin[i][j] == 0) {
                    f = 1;
                    c[i] = j;
                }
            }
            if (!f) {
                vector<vector<bool>> g(N, vector<bool> (N, 0));
                for (int k = 0; k < N; k++) g[i][k] = 1;
                rispondi_griglia(g); return 0;
            }
        }
        vector<vector<bool>> g(N, vector<bool> (N, 0));
        for (int i = 0; i < N; i++) g[i][c[i]] = 1;
        rispondi_griglia(g);
    }
```

4.2.2 $Q \leq 800$ (subtask 3, 38 punti)

Idea: Sapendo che la soluzione consiste nel trovare per ogni riga una lampadina orizzontale da accendere, o restituire l'intera riga accesa, il problema si può risolvere in 800 query dato che $\lceil \log_2 100 \rceil = 7$ pertanto per ogni riga si può cercare per bisezione una lampadina orizzontale.

Si utilizzano quindi 100 query per capire i versi della prima colonna di lampadine (nello stesso modo introdotto per la prima riga nella precedente soluzione parziale). Successivamente per ogni riga se il verso della prima è orizzontale si accende quella, altrimenti si cerca per bisezione la prima lampadina orizzontale, infatti se in un prefisso della riga almeno una lampadina è orizzontale, il numero totale di lampadine accese sulla griglia non sarà divisibile per N (in particolare sarà diverso da $N \cdot l$, dove l è la lunghezza del prefisso). Se nessuna delle lampadine è orizzontale, la riga è composta da lampadine verticali e accendendole tutte si copre tutta la griglia.

Questo approccio può essere ulteriormente ottimizzato fino a $Q = 500$, (quindi 48 punti) sfruttando il fatto che se si prendono N caselle con righe e colonne distinte, almeno la metà sarà dello stesso verso, e quindi è sufficiente cercare in ricerca per bisezione solo le rimanenti.

Implementazione: Un possibile modo di implementare l'idea discussa:

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

int chiedi_griglia(vector<vector<bool>> &g); // definita come sopra
void rispondi_griglia(vector<vector<bool>> &g); //definita come sopra

int main() {
    int N; cin >> N;
    // 0 indica una lampadina orizzontale, 1 una lampadina verticale
    vector<int> pc(N, -1); // vettore che contiene il verso delle lampadine
    // della prima colonna, -1 indica una lampadina di verso ignoto
    // analogamente a prima determino i versi della prima colonna
    for (int i = 1; i < N;i++){
        vector<vector<bool>> g(N, vector<bool> (N, 0));
        g[0][0] = 1; g[i][0] = 1;
        int r = chiedi_griglia(g);
        if (r == N) {
            pc[0] = 1; pc[i] = 1;
        }
        else if (r == 2*N) {
            pc[0] = 0; pc[i] = 0;
        }
    }
    if (pc[0] == -1) {
        vector<vector<bool>> g(N, vector<bool> (N, 0));
        g[1][0] = 1; g[2][0] = 1;
        int r = chiedi_griglia(g);
        if (r == N) {
            pc[1] = 1; pc[2] = 1; pc[0] = 0;
        }
        else if (r == 2*N) {
            pc[1] = 0; pc[2] = 0; pc[0] = 1;
        }
    }
    for (int i = 0; i < N; i++) {
        if (pc[i] == -1) pc[i] = !pc[0];
    }

    vector<int> c(N); // questo vettore conterrà la prima lampadina in ogni
    // riga che è verticale
    for (int i = 0; i < N; i++) {
        if (pc[i] == 0) {
            // la prima lampadina è orizzontale
            c[i] = 0; continue;
        }
        // ricerca per bisezione sulla prima lampadina orizzontale accendendo
        // tutto il prefisso sulla data riga
        int l = 0, r = N;
        while (l < r-1) {
            int m = (l+r)/2;
            vector<vector<bool>> g(N, vector<bool> (N, 0));
            for (int j = 0; j <= m; j++) g[i][j] = 1;
            int t = chiedi_griglia(g);
            if ((t % N) == 0) l = m;
            else r = m;
        }
        if (r == N) {
            // se non è stata trovata una lampadina orizzontale, tutte le
            // lampadine sono verticali e dunque si ha la risposta
            vector<vector<bool>> g(N, vector<bool> (N, 0));
            for (int j = 0; j < N; j++) g[i][j] = 1;
```

```
            rispondi_griglia(g);
            return 0;
        }
        c[i] = r;
    }
    // come precedentemente si sfruttano le informazioni nel vettore c per
    // stampare la risposta
    vector<vector<bool>> g(N, vector<bool> (N, 0));
    for (int i = 0; i < N; i++) g[i][c[i]] = 1;
    rispondi_griglia(g);
}
```

4.2.3 Soluzione completa

Al momento della stesura di questa dispensa non è purtroppo nota all'autrice la soluzione completa a questo problema nè qualsiasi altra idea che permetta di superare il punteggio dei parziali descritti.

5 D - Make them meet

5.1 Testo originale del problema

Mila e Laura sono amiche online da molto tempo però non si sono mai incontrate nella vita reale. Attualmente stanno partecipando entrambe allo stesso evento in presenza, il che significa che si incontreranno sicuramente. Tuttavia, l'hotel in cui alloggiano entrambe è molto grande e confusionario. Pertanto, dopo diversi giorni, non si sono ancora incontrate.

L'hotel è composto da N camere, numerate da 0 a $N-1$. Ogni camera ha una lampada che può essere accesa con diversi colori. Hai trovato il quadro di gestione elettrico dell'hotel, che permette di modificare i colori delle lampade. Il tuo obiettivo è guidare Mila e Laura, utilizzando le lampadine, per farle finalmente incontrare.

L'hotel può essere rappresentato come un grafo con N vertici (le camere) e M archi (i corridoi che collegano le camere). Mila e Laura inizialmente sono in due stanze diverse, ma non sai quali. Per fortuna puoi guidarle effetuando un certo numero di mosse. Ogni mossa consiste nello stampare una lista di N numeri interi, $c_1, c_2, ..., c_{N-1}$, che significa che il colore della lampadina nella stanza i diventa c_i per ogni $i = 0, 1, ..., N-1$. Mila e Laura poi guarderanno il colore della lampadina nella stanza in cui si trovano attualmente e cammineranno verso una stanza vicina la cui lampadina ha lo stesso colore. Se non esiste una tale stanza vicina, rimarranno dove sono. Se ci sono più stanze vicine la cui lampadina ha quello stesso colore, ne sceglieranno una arbitrariamente.

Se Mila e Laura si trovano nella stessa stanza o utilizzano lo stesso corridoio contemporaneamente in qualsiasi momento dei tuoi spostamenti, sei riuscita a farle incontrare. Puoi effettuare al massimo 20.000 mosse, ma otterrai un punteggio più alto se ne utilizzerai meno.

(Tieni presente che non sai da quali stanze Mila e Laura iniziano o come scelgono se hanno più stanze con lo stesso colore tra cui scegliere. La tua soluzione deve essere corretta indipendentemente dalle stanze iniziali o dalle loro scelte.)

Input e output: La prima riga contiene due numeri interi, N e M , rispettivamente il numero di camere e il numero di corridoi dell'hotel. Le seguenti righe M contengono ciascuna due numeri interi, u e v, il che significa che le stanze u e v sono collegate da un corridoio.

Stampa una riga con un numero intero K, il numero di mosse. Su ciascuna delle seguenti K righe, stampa N interi, $c_1, c_2, ..., c_{N-1}$, tali che $0 \leq c \leq N$ per tutti i. Queste K righe rappresentano le tue mosse, in ordine cronologico.

Limiti:

- $2 \leq N \leq 100$
- $N - 1 \leq M \leq \frac{N(N-1)}{2}$
- $1 \leq u_i, v_i \leq N - 1$ e $u_i \neq v_i$ per ogni 1
- È possibile raggiungere ogni stanza da ogni altra stanza. Inoltre, non ci sono corridoi che vanno da una stanza a se stessa, e non ci sono corridoi multipli tra una coppia di stanze.
- $K \leq 20.000$.
- (tempo massimo di esecuzione: 9 secondi)

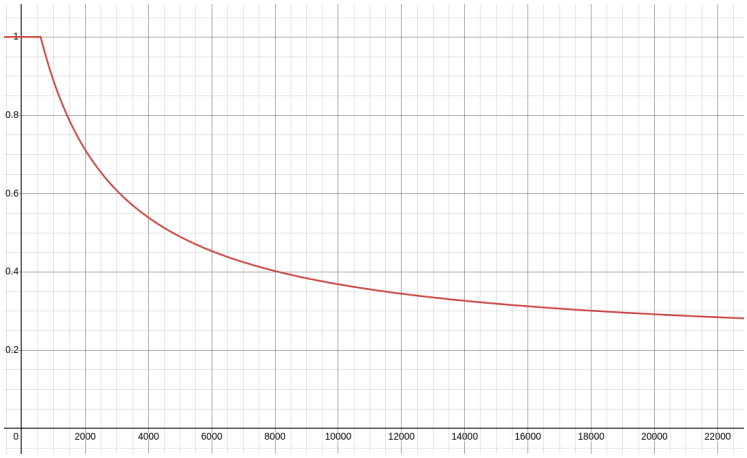

Figure 3: Punteggio in funzione del numero di mosse

Punteggio e subtask: Il punteggio P per ogni testcase (valutato da 0 a 1) è dato dalla seguente formula:

$$P = min(1, \frac{2000}{K + 1900} + \frac{1}{5})$$

L'andamento di questa funzione in K, (numero di mosse), è descritto nel grafico 3. Il punteggio totale è dato dal massimo punteggio ottenuto per ogni gruppo di test, cioè ogni subtask. Alcune subtask hanno limitazioni specifiche come dalla seguente tabella:

Subtask	Punteggio massimo	Limiti
1	10	$M = N - 1$, e i corridoi sono $(0,1),(0,2),(0,3),...,(0,N-1)$ (il grafo è una stella)
2	13	$M = \frac{N(N-1)}{2}$ (il grafo è completo)
3	11	$M = N - 1$, e i corridoi sono $(0,1),(1,2),(2,3),...,(N-2,N-1)$ (il grafo è una linea)
4	36	$M = N - 1$ (il grafo è un albero)
5	30	nessuna limitazione aggiuntiva

5.2 Risoluzione

5.2.1 Grafo a stella (subtask 1)

Idea: Una stella è un grafo formato da un nodo, che chiameremo centro, e ogni altro nodo connesso solamente al nodo centro, chiameremo questi nodi foglie. Si può dimostrare che in questo caso sono sufficienti 3 operazioni:

1. accensione di tutte le lampadine del medesimo colore
2. accensione di una coppia di lampadine, formata da centro e una foglia arbitraria (con i restanti nodi di colore diverso, è indifferente quale)

3. accensione di tutte le lampadine del medesimo colore

Dimostrazione: Distinguiamo due casi:

- se nessuna delle ragazze si trova nel nodo centro, si incontreranno in esso alla prima mossa. Infatti ogni foglia ha un solo arco percorribile (quello verso il centro).
- se una delle ragazze si trova nel nodo centro, dopo la prima mossa si sarà spostata verso una foglia, mentre l'altra si sarà mossa verso il centro. è possibile ma non assicurato che si siano incontrate, infatti dal centro ogni foglia può essere scelta arbitrariamente. La situazione è quindi equivalente a quella iniziale.

  Con la seconda operazione, o le ragazze si incontrano o quella che si trovava nel nodo centro si sarà mossa in una foglia. Quindi o il caso è risolto, o ci si è ricondotti al primo punto, con entrambe le ragazze in nodi foglie. Si riapplica quindi l'operazione che risolve il primo caso e la soluzione è completa.

Implementazione: Un possibile modo di implementare l'idea discussa:

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

int main() {
        int N, M; cin >> N >> M;
        for (int i = 0; i < M; i++) {
                int a, b; cin >> a >> b;
                //non è necessario memorizzare gli archi
                //essendo assicurato nel testo siano della forma
                // (0, 1), (0, 2), (0, 3), ..., (0, N - 1)
        }

        cout << "3\n"; // serviranno esattamente 3 mosse, quelle descritte sopra
        for (int i = 0; i < N; i++) {
                cout << "0 ";
        }
        cout << "\n";
        cout << "1 1 ";
        for (int i = 2; i < N; i++) {
                cout << "0 ";
        }
        cout << "\n";
        for (int i = 0; i < N; i++) {
                cout << "0 ";
        }
}
```

5.2.2 Grafo a linea (subtask 3)

Idea: Una linea è un grafo formato da archi che formano un'unica catena senza diramazioni, ogni nodo è dunque connesso a due nodi distinti ad eccezione di due nodi, che chiameremo estremi, che hanno una sola connessione. Si può dimostrare che in questo caso sono sufficienti N operazioni (non è stato verificato questo sia il minimo numero ottenibile). Le operazioni consistono in accendere lampadine dello stesso colore per coppie di nodi successivi (con colori diversi tra le coppie), alternando successivamente tra le due configurazioni possibili.

Dimostrazione: Consideriamo una sola tra le ragazze, le operazioni effettuate hanno come conseguenza il movimento di un vertice alla volta verso un estremo, e poi allo stesso modo verso l'altro. Le ragazze si possono quindi muovere inizialmente in direzioni diverse ma si può facilmente verificare che in N o meno mosse, comunque siano posizionate all'inizio, si incontrano a causa dei cambi di direzione agli estremi.

Implementazione: Un possibile modo di implementare l'idea discussa:

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

int main() {
        int N, M; cin >> N >> M;
        for (int i = 0; i < M; i++) {
                int a, b; cin >> a >> b;
                //non è necessario memorizzare gli archi
                //essendo assicurato nel testo siano della forma
                // (0, 1), (1, 2), (2, 3), ..., (N - 2, N - 1)
        }
        cout << 2*((N+1)/2) << "\n"; // numero di mosse corrispondente al doppio
        // della metà di N arrotondata per eccesso
        for (int i = 0; i < (N+1)/2; i++) {
                // le mosse come descritte sopra
                for (int j = 0; j < N; j+=2) {
                        cout << j << " ";
                        if (j+1 < N) cout << j << " ";
                }
                cout << "\n";
                cout << 0 << " ";
                for (int j = 1; j < N; j+=2) {
                        cout << j << " ";
                        if (j+1 < N) cout << j << " ";
                }
                cout << "\n";
        }
}
```

5.2.3 Grafo completo (subtask 2)

Un grafo completo è un grafo in cui ogni coppia di nodi è connessa da un arco. Analizzando attentamente questa caratteristica si può notare che la soluzione spiegata per il grafo a linea è corretta anche in questo caso. Infatti sfrutta archi che per definizione devono essere presenti nel grafo completo, e inoltre se si usano colori differenti per ogni coppia, gli archi aggiuntivi sono sempre inutilizzati in quanto ai loro estremi si avranno sempre lampadine accese con colori differenti. Pertanto l'implementazione presentata nella precedente sezione risolve entrambe queste subtask.

5.2.4 Grafo ad albero (subtask 4)

Idea: Un albero è un grafo connesso (cioè in cui ogni nodo può essere raggiunto da ogni altro seguendo gli archi del grafo), e senza cicli. Come spesso accade la soluzione dell'albero è una generalizzazione dei suoi due casi limite: la linea e la stella. in particolare l'algoritmo che costruiremo è l'estensione di quello utilizzato per la linea, basato sulla ripetizione per $2 * N$ passi di due istruzioni base.

Si sceglie come radice dell'albero una foglia, cioè un nodo connesso ad un unico altro nodo nel grafo. Questo significa che si possono definire le distanze di ogni altro nodo da questo nodo radice. I due passi iterati risultano:

1. *(operazione pari)* accensione di ogni nodo situato ad una distanza pari dalla radice di un colore diverso, accensione di ogni altro nodo con lo stesso colore del padre(cioè primo nodo nel percorso da un nodo alla radice).
2. *(operazione dispari)* accensione di ogni nodo situato ad una distanza dispari dalla radice di un colore diverso, accensione di ogni altro nodo, eccetto la radice con lo stesso colore del padre. La radice è dello stesso colore dell'unico nodo a cui è connessa.

Dimostrazione: La correttezza di questo approccio può essere dimostrata seguendo il percorso delle ragazze separatamente e dividendo l'analisi in diversi casi:

- se inizialmente una ragazza si trova in un nodo di distanza pari alla radice (ma non nella radice), durante la prima *operazione pari* si allontanerà dalla radice, e così per ogni passo fino al raggiungimento di una foglia, seguendo un percorso arbitrario. Certamente poi dalla foglia risalirà lungo l'albero verso la radice, fino a venire intrappolata nell'arco tra radice e il nodo a distanza uno dalla radice.
- se inizialmente una ragazza si trova in un nodo a distanza dispari dalla radice, durante la prima *operazione pari* si avvicinerà alla radice, e così per ogni operazione successiva fino a finire intrappolata anch'essa nell'arco tra radice e nodo a distanza uno dalla radice.
- se inizialmente una ragazza si trova nella radice, dopo la prima *operazione pari*, si troverà nel nodo appena sottostante alla radice, e poi potrà scegliere se tornare alla radice o spostarsi in un altro nodo. Questa situazione si ripresenta per le successive $2 * N$ volte. Se la ragazza resta nel ciclo radice-nodo a distanza uno dalla radice, per $2 * N$ mosse incontrerà di certo ogni altro possibile percorso. Se invece ad un certo punto si sposta in un altro nodo impiegherà altre $2 * N$ mosse per attraversare l'albero fino ad una foglia e tornare indietro

Implementazione: Un possibile modo di implementare l'idea discussa è il seguente:

```cpp
#include <bits/stdc++.h>
using namespace std;
vector<vector<int>> adj(100); // lista di nodi adiacenti a ogni nodo
vector<int> dist(100, 0); // distanza di ogni nodo dalla radice
vector<int> pad(100, -1); // padre di ogni nodo

void dfs(int att) {
        // funzione di visita per profondità dell'albero che popola il vettore
        // di distanze e del padre di ogni nodo
        for (int pros : adj[att]){
                if (pros == pad[att]) continue;
                dist[pros] = dist[att]+1;
                pad[pros] = att;
                dfs(pros);
        }
}

int main() {
        int N, M; cin >> N >> M;
        for (int i = 0; i < M; i++) {
                // leggo gli archi e popolo le liste di adiacenza
                int a, b; cin >> a >> b;
                adj[a].push_back(b); adj[b].push_back(a);
        }
        int rad, aus;
        for (int i = 0; i < N; i++) {
                // trovo una foglia e radico in essa l'albero
                if (adj[i].size() == 1) {
                        dfs(i);
                        rad = i;
                        aus = adj[i][0];
                        break;
                }
        }
        // stampo le mosse descritte sopra
        cout << 4*N << "\n";
        for (int k = 0; k < 2*N; k++) {
                for (int i = 0; i < N; i++) {
                        if ((dist[i]%2) == 0) cout << i << " ";
                        else cout << pad[i] << " ";
```

```
                }
                cout << "\n";
                for (int i = 0; i < N; i++) {
                        if ((dist[i]%2) == 1) cout << i << " ";
                        else if (i == rad) cout << aus << " ";
                        else cout << pad[i] << " ";
                }
                cout << "\n";
        }
}
```

5.2.5 Soluzione completa

Idea: L'ultima subtask del problema cioè la sua soluzione completa è un'ulteriore generalizzazione della soluzione per un albero. L'approcio qui implementato è quello descritto nella soluzione ufficiale di questo problema che può essere trovata nella stessa sezione dei testi (purtroppo solamente in versione inglese), dove è illustrata assieme ad una spiegazione sintetica della dimostrazione. Essa consiste fondamentalmente in varie considerazioni che adattano la soluzione dell'albero ad un grafo qualsiasi, distinguendo il caso in cui invece questo non è possibile a causa della quantità di archi, che però consente di utilizzare la soluzione del grafo completo.

Implementazione: Un possibile modo di implementare l'idea risolutiva del problema è il seguente:

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

void sol_riga(int N, vector<int> &ord) {
        // funzione che stampa la soluzione di una riga come descritta precedentemente
        cout << 2*((N+1)/2) << "\n";
        for (int i = 0; i < (N+1)/2; i++) {
                vector<int> col(N);
                for (int j = 0; j < N; j+=2) {
                        col[ord[j]] = ord[j];
                        if (j+1 < N) col[ord[j+1]] = ord[j];
                }
                for (int j = 0; j < N; j++) cout << col[j] << " ";
                cout << "\n";
                col[ord[0]] = ord[0];
                for (int j = 1; j < N; j+=2) {
                        col[ord[j]] = ord[j];
                        if (j+1 < N) col[ord[j+1]] = ord[j];
                }
                for (int j = 0; j < N; j++) cout << col[j] << " ";
                cout << "\n";
        }
}

// struttura che facilita il compito di calcolare e stampare la soluzione
// sullo spanning tree individuato
struct sol_albero {
        int N;
        vector<vector<int>> adj;
        vector<int> dist; // distanza di ogni nodo dalla radice
        vector<int> pad; // padre di ogni nodo

        void dfs(int att) {
                // definita esattamente come nella soluzione del solo albero
                for (int pros : adj[att]){
                        if (pros == pad[att]) continue;
                        dist[pros] = dist[att]+1;
                        pad[pros] = att;
                        dfs(pros);
                }
```

```
	}

	void risolvi(int n, vector<vector<int>> aa, int rad, int pb, int aus) {
		N = n;
		adj.resize(N); dist.resize(N); pad.resize(N);
		fill(pad.begin(), pad.end(), -1); fill(dist.begin(), dist.end(), 0);
		for (int i = 0; i < N; i++) {
			// popolo la lista di adiacenza interna con lo spanning
			// tree individuato
			adj[i] = aa[i];
		}
		dfs(rad);
		// stampo le mosse come descritte sopra
		cout << 6*N << "\n";
		for (int k = 0; k < 2*N; k++) {
			for (int i = 0; i < N; i++) {
				if ((dist[i]%2) == 0|| i == pb) cout << i << " ";
				else cout << pad[i] << " ";
			}
			cout << "\n";
			for (int i = 0; i < N; i++) {
				if ((dist[i]%2) == 1) cout << i << " ";
				else if (i == rad) cout << aus << " ";
				else cout << pad[i] << " ";
			}
			cout << "\n";
			for (int i = 0; i < N; i++) {
				if (i == rad || i == pb) cout << rad << " ";
				else cout << i << " ";
			}
			cout << "\n";
		}
	}

};

int main() {
	//freopen("output.txt", "w", stdout);
	int N, M; cin >> N >> M;
	cout << N << " " << M << "\n";
	vector<vector<int>> adj(N); // lista di adiacenza del grafo
	vector<vector<bool>> mata(N, vector<bool> (N, 0)); // matrice di adiacenza
	// che per ogni coppia di nodi segna se sono connessi da un arco
	for (int i = 0; i < M; i++) {
		int a, b; cin >> a >> b;
		cout << a << " " << b << "\n";
		// leggo gli archi e li segno
		adj[a].push_back(b);
		adj[b].push_back(a);
		mata[a][b] = 1; mata[b][a] = 1;
	}

	vector<int> pdr(N, -1); // segna per ogni nodo il nodo da cui è stato
	// visitato nella dfs
	vector<bool> vis(N, 0); // segna se un nodo è stato visitato o meno per
	// rendere la dfs lineare sul numero di nodi
	vector<int> prof(N, 0); // segna la profondità di un nodo nella visita dfs
	vector<vector<int>> sptr(N); // vettore che contiene lo spanning tree
	// individuato dalla dfs

	// eseguo una dfs non ricorsiva con l'aiuto di una struttura dati della
	// standard library di tipo lifo
	stack<int> dfs; dfs.push(0);
	while (!dfs.empty()) {
		int x = dfs.top(); dfs.pop();
		if (vis[x]) continue;
		vis[x] = 1;
		if(pdr[x] != -1) {
```

```
                prof[x] = prof[pdr[x]]+1;
                sptr[pdr[x]].push_back(x);
                sptr[x].push_back(pdr[x]);
        }
        for (int y : adj[x]) {
                if (vis[y]) continue;
                pdr[y] = x;
                dfs.push(y);
        }
}

// controllo se il grafo ha bisogno di essere risolto come una linea
fill(vis.begin(), vis.end(), 0);
bool linea = 1; int beg;
for (int i = 0; i < N; i++) {
        if (sptr[i].size() > 2) linea  = 0;
        if (sptr[i].size() == 1) beg = i;
}
if (linea) {
        // apllico la soluzione della linea trovando l'ordine dei nodi e
        // passandolo alla funzione ausiliare
        vector<int> ord;
        ord.push_back(beg); vis[beg] = 1;
        do {
                if (vis[sptr[beg][0]]) beg = sptr[beg][1];
                else beg = sptr[beg][0];
                ord.push_back(beg); vis[beg] = 1;
        } while (sptr[beg].size() == 2);
        sol_riga(N, ord);
        return 0;
}

// adatto lo spanning tree in modo che rispetti tutti i requisiti per
// applicare le mosse

// trovo il punto di diramazione più profondo
pair<int, int> gb = {0, 0};
for (int i = 0; i < N; i++) {
        if (sptr[i].size() <= 2) continue;
        gb = max(gb, {prof[i], i});
}

// impongo ad ogni figlio del punto di diramazione di essere disconnesso
// dal padre del punto di diramazione, scegliendo il ramo dove
// posizionare questo padre
vector<int> rs;
bool pa = 1; int fp;
for (int x : sptr[gb.second]) {
        if (x == pdr[gb.second]) continue;
        if (mata[x][pdr[gb.second]] || !pa) {
                rs.push_back(x);
                continue;
        }
        fp = x; pa = 0;
}
if (pa) {
        fp = rs[rs.size()-1];
        rs.pop_back();
}

//scendo sul ramo del padre scelto fp per mantenerlo una linea, cioè
// mantenere il punto di diramazione il primo punto dove quel ramo non è
// una linea
vis[gb.second] = 1;
vis[fp] = 1;
int att = fp;
while (sptr[att].size() == 2) {
        if (vis[sptr[att][0]]) att = sptr[att][1];
```

```
                else att = sptr[att][0];
                vis[att] = 1;
        }

        // ricalcolo da capo lo spanning tree mantenendo la nuova radice e tutto
        // il suo ramo fino al punto di diramazione con una seconda dfs non
        // ricorsiva, ovviamente salvo il nuovo spanning tree
        for (int i = 0; i < N; i++) {
                if (!vis[i] || i == gb.second) sptr[i].clear();
        }
        sptr[gb.second].push_back(fp);
        if(pdr[gb.second] > -1) rs.push_back(pdr[gb.second]);
        fill(pdr.begin(), pdr.end(), -1);
        for (int i : rs) {
                if (vis[i]) continue;
                pdr[i] = gb.second;
                sptr[i].push_back(gb.second);
                sptr[gb.second].push_back(i);
                dfs.push(i);
                while (!dfs.empty()) {
                        int x = dfs.top(); dfs.pop();
                        if (vis[x]) continue;
                        vis[x] = 1;
                        if(pdr[x] != -1) {
                                sptr[pdr[x]].push_back(x);
                                sptr[x].push_back(pdr[x]);
                        }
                        for (int y : adj[x]) {
                                if (vis[y]) continue;
                                pdr[y] = x;
                                dfs.push(y);
                        }
                }
        }

        // passo lo spanning tree finale alla struttura ausiliare che stampa le
        // mosse necessarie per la soluzione, con la corretta radice, padre del
        // punto di diramazione e punto di diramazione stesso
        sol_albero d;
        if (sptr[fp].size() == 1) d.risolvi(N, sptr, fp, -1, gb.second);
        else if (sptr[fp][0] == gb.second) d.risolvi(N, sptr, fp, sptr[fp][1], gb.second);
        else d.risolvi(N, sptr, fp, sptr[fp][0], gb.second);
}
```