```c
#include <stdio.h>
#include <stdlib.h>

typedef struct _edges {
    size_t num_edges;
    size_t size; // Necessaria solo se dovete aggiungere archi dinamicamente
    size_t *edges;
} edges;

// Inizializza un grafo con n vertici
edges *graph_init(size_t n) {
    edges *es = malloc(n * sizeof(edges));

    for (size_t i = 0; i < n; i++)
        es[i].num_edges = 0;

    return es;
}

// Rialloca l'array di vertici, raddoppiandone lo spazio disponibile
void edges_double(edges *es) {
    if (es->size == 0)
        es->size = 1;

    es->edges = realloc(es->edges, sizeof(edges) * es->size * 2);
    es->size *= 2;
}

// Aggiunge un arco dal vertice from al vertice to
void graph_add_edge(edges* g, size_t from, size_t to) {
    // Parte non necessaria per l'esercizio
    // In questa implementazione compatta se volete aggiungere dinamicamente
    // archi senza conoscere a priori il numero, dovete utilizzare una politica
    // dimezza-raddoppia per ridimensionare l'array.
    // In questo caso verifichiamo che ci sia uno spazio libero, altrimenti
raddoppiamo
    if (g[from].num_edges + 1 > g[from].size) {
        edges_double(g + from);
    }
    // ------------------------------------

    g[from].edges[g[from].num_edges] = to;
    g[from].num_edges++;
}

// Colorazione ricorsiva con visita di tipo DFS, pos è il vertice che abbiamo
// appena raggiunto, c è il colore con cui stiamo cercando di colorarlo
int graph_color(edges *g, size_t n, int *colors, size_t pos, int c) {
    // Se il vertice è già stato colorato con un colore diverso allora errore
    if (colors[pos] != -1 && colors[pos] != c)
        return 0;

    colors[pos] = c;
    for (size_t i = 0; i < g[pos].num_edges; i++) {
```

```
        size_t to = g[pos].edges[i];
        // Se il vertice che abbiamo raggiunto non era colorato, proviamo a
        // colorarlo con il colore opposto
        if (colors[to] == -1 && !graph_color(g, n, colors, to, !c))
            return 0;
        // Se invece era già colorato, verifichiamo che sia colorato correttamente
        if (colors[to] != -1 && colors[to] != !c)
            return 0;
    }

    return 1;
}

// Verifica che il grafo sia bipartito, restituisce un booleano
int graph_is_bipartite(edges* g, size_t n) {
    // Non è necessaria la malloc perché in questo caso non ritorniamo
    // la colorazione
    int colors[n];

    // -1 indica che il vertice non è stato ancora visitato
    for (size_t i = 0; i < n; i++)
        colors[i] = -1;

    // La visita è di tipo DFS quindi dobbiamo ripetere la procedura su ogni
    // vertice per assicurarci di visitare ogni vertice, anche quelli isolati
    for (size_t i = 0; i < n; i++) {
        // Se il vertice è già stato colorato ignoriamo, altrimenti verifichiamo
che
        // la colorazione ricorsiva abbia successo
        if (colors[i] == -1 && !graph_color(g, n, colors, i, 0))
            return 0;
    }

    return 1;
}

int main(int argc, const char* argv[]) {
    size_t n;
    scanf("%lu", &n);

    // Inizializziamo il grafo con n vertici
    edges* g = graph_init(n);

    for (size_t from = 0; from < n; from++) {
        size_t ne;
        scanf("%lu", &ne);
        // Poiché useremo il grafo in sola lettura possiamo preallocare lo
        // spazio necessario per contenere tutti i vertici connessi
        g[from].edges = malloc(ne * sizeof(size_t));
        g[from].size = ne;

        for (size_t j = 0; j < ne; j++) {
            size_t to;
            scanf("%lu", &to);
            graph_add_edge(g, from, to);
```

```
        }
    }

    printf("%d\n", graph_is_bipartite(g, n));

    return 0;
}
```