## A-III-4 Špiónske voľby

Preložme si najprv úlohu do teórie grafov. Vrcholy grafu budú predstavovať špiónov a hrany budú spájať špiónov, ktorí sa poznajú.

### Pomalé riešenia

Existujú dve pomerne priamočiare pomalé riešenia. Prvé z nich si okrem siete špiónov pamätá názor každého špióna. Keď treba zmeniť špiónov názor, tak to dokáže v konštantnom čase. Na otázku koľko bodov získali kandidáti odpovedá tak, že prejde celú sieť a pre každú dvojicu špiónov, ktorý sa poznajú zistí ako hlasovali a následne pripočíta patričné body. Toto celé beží ale v lineárnom čase.

Druhé riešenie si navyše pamätá aktuálny stav bodov. Takže na otázku o bodoch vie odpovedať v konštatnom čase. Problém ale nastáva so zmenením názoru špióna. Keď špión $x$ zmení názor, tak sa treba pozrieť na každého jeho suseda a na základe toho prepočítať body. Toto môže byť síce v niektorých prípadoch rýchle (lebo v priemere špión pozná najviac 6 iných špiónov), ale stále môžu nastať extrémne pomalé prípady (predstavte si sieť, kde špión 1 je spojený s každým a nik iný sa nepozná a následne špión 1 veľmi často mení názor). Takže v najhoršom prípade nás zmena názoru bude stáť lineárny čas od veľkosti siete.

### Stredne dobré riešenie

Vrcholy si môžeme rozdeliť na *malé* (stupňa do 6) a *veľké*. V zadaní bolo pre niektoré testovacie sady zaručené, že žiadne dva veľké vrcholy nesusedia.

Ako vieme riešiť takéto vstupy? Pre každý veľký vrchol si budeme pamätať tabuľku s informáciou, ktorého kandidáta volí koľko jeho susedov. Ak zmení názor malý vrchol, prejdeme jeho susedov, a tým, ktorí sú veľkí, upravíme tabuľku. Ak zmení názor veľký vrchol, použijeme v ňom zapamätanú tabuľku na to, aby sme dopočítali výsledok.

Takéto riešenie má konštantnú časovú zložitosť na operáciu pre sady, v ktorých žiadne dva veľké vrcholy nesusedia. (Jeho vhodným zovšeobecnením vieme dostať riešenie, ktoré bude mať vždy časovú zložitosť $O(\sqrt{n})$ na operáciu.)

### Užitočné vlastnosti siete špiónov

V tomto vzorovom riešení a aj v domácom kole sme spomínali, že priemerný počet susedov vrchola je maximálne 6. Z toho vyplýva, že v grafe vždy existuje vrchol, ktorý má nanajvýš 6 susedov. Keď tento vrchol z grafu odstránime, tak v grafe, ktorý nám ostal, zase existuje vrchol, ktorý má najviac 6 susedov.

Takto vieme postupne z grafu odstrániť všetky vrcholy. Navyše si môžeme zapamätať poradie, v akom sme vrcholy odstránili: $p_1, p_2, \ldots, p_n$. V tomto poradí nám platí, že každý vrchol $p_i$ má maximálne 6 hrán do vrcholov $p_{i+1}, \ldots, p_n$. Tieto hrany si osobitne zapamätáme a nazveme ich *dopredné hrany*. Ale pozor, stále môže mať vrchol $p_i$ veľmi veľa susedov medzi vrcholmi $p_1, \ldots, p_{i-1}$. Týchto nazveme *zadní susedia*.

Rozmyslite si, že toto predpočítavanie vieme spraviť v lineárnom čase od počtu vrcholov. Stačí si pre každý vrchol pamätať koľko má ešte neodstránených susedov a keď toto číslo klesne na 6, tak ho zaradiť do fronty.

### Vzorové riešenie

Pre každý vrchol si budeme pamätať, ako momentálne hlasuje. Navyše si budeme pre každý vrchol $p_i$ a pre každého kandidáta $j$ pamätať hodnotu $q_{i,j}$: počet zadných susedov vrcholu $p_i$, ktorí volia kandidáta $j$.

Nech teraz zmení názor vrchol $p_i$. V konštatnom čase prejdeme dopredné hrany z neho a pozrieme sa, koho volia dotyční (nanajvýš šiesti) susedia. Následne v konštantnom čase prejdeme tabuľku zapamätanú vo vrchole $p_i$, čím naraz spracujeme všetkých jeho zadných susedov.

Hotovo? Ešte nie. Síce sme už spočítali nové počty bodov kandidátov, potrebujeme ešte ale upraviť niektoré hodnoty v tabuľke $q$. Zmenil sa totiž práve hlas vrcholu $p_i$. Čo treba zmeniť? Tabuľky pre tie vrcholy, do ktorých vedú dopredné hrany z vrcholu $p_i$. No a takých je najviac 6, v každej meníme dve hodnoty, dokopy teda v tabuľke $q$ spravíme najviac 12 zmien. S využitím dopredných hrán vieme tieto zmeny ľahko spraviť v konštantom čase.

Celkovo teda vieme v konštantom čase zmeniť názor špióna a v konštantom čase povedať body jednotlivých kandidátov. Navyše potrebujeme spraviť predpočítanie v čase $O(n)$. Na uloženie grafu a tabuliek nám stačí $O(n)$ pamäte.

### Listing programu (C++)

```
#include <vector>
#include <queue>
#include <cstdio>
using namespace std;

static vector<vector<int> > g;  // cely graf
static vector<vector<int> > gp; // dopredne hrany
static vector<vector<int> > q;
static vector<int> votes;
static vector<int> state;

void spioni (int n, int poznaju_sa[][2], int voli[]) {
  g.resize(n);
  gp.resize(n);
  state.resize(5);
  votes.resize(n);
  q.resize(n, vector<int>(5));

  // pocty nevymazanych susedov
  vector<int> ps(n);
  for (int i = 0; i < m; i++) {
    g[poznaju_sa[i][0]].push_back(poznaju_sa[i][1]);
    g[poznaju_sa[i][1]].push_back(poznaju_sa[i][0]);
    ps[poznaju_sa[i][0]]++;
    ps[poznaju_sa[i][1]]++;
    if (voli[poznaju_sa[i][0]] == voli[poznaju_sa[i][1]]) {
      state[voli[poznaju_sa[i][0]]]++;
    }
  }

  queue<int> fr;
  // oznacenie pre vymazane vrcholy
  vector<bool> vymaz(n, false);
  vector<bool> sprac(n, false);
  for (int i = 0; i < n; i++) {
    if (ps[i] <= 6) {
      fr.push(i);
      vymaz[i] = true;
    }
  }

  while (!fr.empty()) {
    int x = fr.front(); fr.pop();
    sprac[x] = true;
    for (int i = 0; i < g[x].size(); i++) {
      if (sprac[g[x][i]]) continue;
      gp[x].push_back(g[x][i]);
      if (vymaz[g[x][i]]) continue;

      ps[g[x][i]]--;
      if (ps[g[x][i]] <= 6) {
        fr.push(g[x][i]);
        vymaz[g[x][i]] = true;
      }
    }
  }

  for (int i = 0; i < n; i++) {
    votes[i] = voli[i];
    for (int j = 0; j < gp[i].size(); j++) {
      q[gp[i][j]][votes[i]]++;
    }
  }
}

void zmen_nazor (int spion, int voli) {
  // najprv odcitame co treba
  state[votes[spion]] -= q[spion][votes[spion]];
  for (int i = 0; i < gp[spion].size(); i++) {
    if (votes[spion] == votes[gp[spion][i]]) {
      state[votes[spion]]--;
    }
    q[gp[spion][i]][votes[spion]]--;
  }

  votes[spion] = voli;
  state[votes[spion]] += q[spion][votes[spion]];
  for (int i = 0; i < gp[spion].size(); i++) {
    if (votes[spion] == votes[gp[spion][i]]) {
      state[votes[spion]]++;
```

```
    }
    q[gp[spion][i]][votes[spion]]++;
  }
}

void pocet_bodov (int pocet[5]) {
  for (int i = 0; i < 5; i++) {
    pocet[i] = state[i];
  }
}
```

## A-III-5 Uhlopriečky

Úloha môže na prvý pohľad pôsobiť trocha odstrašujúco. Hráme sa s nejakým $n$-uholníkom plným uhlopriečok. Tie navyše spĺňajú nejaké čudesné pravidlo, že každý z $n-2$ trojuholníkov má aspoň jednu hranu spoločnú s pôvodným $n$-uholníkom.

Preto prvá vec, o čo chceme robiť, je uvidieť zadanie z iného uhla, nájsť v ňom niečo jednoduché, čo autor tak usilovne schovával.

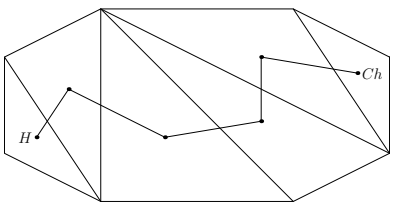
### Ohmatanie úlohy

Najprv si trocha preformulujme cieľ hry. Podľa zadania vyhrá hráč, ktorý spraví červený trojuholník. O ťah skôr to však znamená, že prehrá hráč, ktorý ofarbí druhú hranu nejakého trojuholníka. Trojuholníky, ktoré už majú nejakú hranu zafarbenú, nazvime obsadené. Strany obsadených trojuholníkov nazvime obsadené hrany.

Zjavne teda dobrá stratégia je neofarbovať obsadené hrany, kým sa to dá. Hráč, ktorý už nemá na výber neobsadené hrany, prehrá.

Teraz poďme skúmať to divné pravidlo o povolených trojuholníkoch. Čo znamená, že má trojuholník aspoň jednu hranu spoločnú s pôvodným $n$-uholníkom? Z opačného uhla pohľadu sa dá tá istá veta povedať tak, že trojuholník má najviac dve hrany spoločné s inými trojuholníkmi. Teda každý trojuholník susedí najviac s dvoma ďalšími. S trochou počítania si vieme dokonca zrátať, že (okrem prípadu, keď $n = 3$ a celý $n$-uholník je jeden nerozdelený trojuholník) máme práve $n-4$ trojuholníkov s dvoma susedmi a dva trojuholníky majú jedného suseda.

Toto pozorovanie nám stačí na to, aby sme si všimli, že trojuholníky tvoria akoby dlhého trojuholníkového „hada“. Na jednom konci tvorí trojuholník s jedným susedom hlavu, potom nasleduje telo hada, tvorené trojuholníkmi s dvoma susedmi a chvost hada je zasa trojuholník s jedným susedom. Lepšie je to asi vidieť na obrázku nižšie, kde je ukážka nejakého rozdelenia 8-uholníka, kde hlava je označená písmenom $H$, a chvost $Ch$.




Podobného hada nájdete, keď si skúsite v ľubovoľnom správne rozdelenom $n$-uholníku pospájať susedné trojuholníky. (Hadovi sa odborne hovorí duálny graf. Pre ľubovoľnú triangluláciu je jej duálnym grafom nejaký strom. Vrcholy, kde sa strom vetví, zodpovedajú práve trojuholníkom, ktoré nemajú ani jednu stranu na obvode $n$-uholníka.)

Pri pohľade na obrázok by nám mohol skrsnúť nápad, že samotné trojuholníky nie sú vôbec dôležité a dôležitý je len ich počet, resp. dĺžka hada. Aby sme si túto hypotézu overili, skúsme sa pozrieť, čo sa stane, keď spravíme nejaký ťah, teda ofarbíme nejakú hranu.

Na začiatku hry máme jeden veľký neofarbený $n$-uholník zložený z $n-2$ neobsadených trojuholníkov. Očíslujme si ich od 1 (hlava) po $n-2$ (chvost). V prvom ťahu môže Usamec ofarbiť buď nejakú stranu $n$-uholníka, alebo nejakú uhlopriečku.

Ofarbením strany $n$-uholníka sme obsadili niektorý jeden trojuholník, nech je $i$-ty v poradí. Keď teraz odstránime obsadené hrany (zvyšné dve strany $i$-teho trojuholníka), dostaneme vo všeobecnosti dve nezávislé časti hracieho plánu – ofarbovanie hrán v jednej z nich nijako neovplyňuje situáciu v druhej.

Navyše tie nové časti sú skoro mnohouholníky. Jediný rozdiel oproti pôvodnému $n$-uholníku je v tom, že z hlavového a tiež z chvostového trojuholníka môže chýbať jedna strana na obvode. To nám ale vôbec nevadí, lebo tieto trojuholníky majú na obvode 2 strany, z ktorých sa aj tak môže ofarbiť len jedna a je jedno, ktorá to bude.

Tak či tak, ofarbením nejakej strany $n$-uholníka, ktorá je zároveň stranou $i$-teho trojuholníka, sa nám pôvodný had dĺžky $n-2$ rozpadne na dva hady dĺžok $i-1$ a $n-2-i$.

Pokiaľ by sme neofarbili stranu $n$-uholníka ale nejakú jeho uhlopriečku, obsadili by sme tak dva susedné trojuholníky. Ak je to $i$-ty a $i+1$-vý, rozpadne sa pôvodný had na dvoh hadov s dĺžkami $i-1$ a $n-3-i$.

Všimnime si, že v každom momente vieme ľubovoľného hada rozseknúť na ľubovoľnom mieste. Čiže priebeh hry vôbec nezávisí od toho, ako Maru na začiatku nakreslila uhlopriečky, len od čísla $n$. Zrazu sa nám hra veľmi zjednodušila.

## Stratégia

Zopakujme si teda ako vyzerá hra. Na začiatku máme jedného hada dĺžky $n-2$. Usamec a Maru sa striedajú v ťahoch (Usamec začína). Ťah spočíva v tom, že si vyberieme nejakého hada, ktorý má nenulovú dĺžku $k$ a rozsekneme ho na dve časti. Tieto časti môžu mať ľubovolné nezáporné dĺžky, ale súčet ich dĺžok musí byť $k-1$ alebo $k-2$. (V $n$-uholníkovej hre môžu obaja hráči samozrejme robiť aj iné ťahy, ale na ľubovoľný ťah, ktorý nepredstavujú rozseknutie hada, protihráč odpovie dokončením červeného trojuholníka.)

Takáto hra sa dá riešiť nejakým všeobecným veľkým kladivom, napríklad Grundyho číslami, no my si vôbec nepotrebujeme komplikovať život. Stačí si všimnúť jednu veľmi jednoduchú vyhrávajúcu stratégiu pre Usamca.

Usamec v prvom ťahu rozsekne hada na dve rovnako dlhé časti. (Teda ak mal pôvodný had nepárnu dĺžku, Usamec obsadí stredný trojuholník, inak obsadí stredné dva.) Následne symetricky opakuje Maruine ťahy: ak Maru spraví nejaký ťah v jednej časti, Usamec spraví analogický ťah v druhej.

Takto je zaručené, že po každom Maruinom ťahu bude vedieť Usamec tiež spraviť ťah. (Ak Maru rozsekla nejakého hada, tak to znamená, že v symetrickej časti existuje rovnaký had, ktorého môže Usamec rozseknúť rovnakým spôsobom.) Preto postupom času (každým ťahom sa zmenší celková dĺžka hadov, takže hra môže trvať najviac $n$ ťahov) Maru nebude môcť spraviť žiaden ťah – všetky hady budú mať nulovú dĺžku. Vtedy bude Maru musieť ofarbiť nejakú hranu už obsadeného trojuholníka, no a v nasledujúcom ťahu Usamec vyhrá.

## Implementácia

Celkom príjemne sa to celé programuje, ak si nejakým spôsobom zabezpečíme prevod medzi pôvodnou hrou a sekaním hada. Hodí sa nám napríklad funkcia, ktorej keď zadáme, stranu $n$-uholníka alebo uhlopriečku, vráti číslo trojuholníka(ov), do ktorého patrí. Tiež sa zíde opačná funkcia, ktorá pre daný trojuholník povie nejakú jeho neobsadenú stranu.

Časová zložitosť samotného algoritmu bude $O(n)$ na celú hru. (Predpočítanie a prvý ťah zaberú čas $O(n)$ a každý ďalší ťah zaberie čas $O(1)$.) Pamäťová zložitosť bude tiež $O(n)$, keďže si potrebujeme pamätať hrací plán a aktuálny stav hry.

Naša implementácia uvedená nižšie je trochu „ťažkotonážna“. V časovej zložitosti má (kvôli použitiu usporiadaných množín a máp) navyše logaritmický faktor. Namiesto čísel pracujeme všade, kde sa dá, priamo so samotnými objektami – hranami a trojuholníkmi. Aj takto sa to dá robiť :-)

## Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <set>
#include <map>
using namespace std;

/*
 * mame pas trojuholnikov
 * ten si mozeme predstavit nasledovne:
```

```
 *
 * |_|_|_|_|_|_|
 *
 * kde | su v principe uhlopriecky a _ su strany, kazde |_| predstavuje jeden trojuholnik
 * v tomto poradi si hrany ocislujeme:
 *
 * |_|_|_|_|_|_|
 * 0123456789abc
 *
 * teraz vieme lahko tahat symetricky + ku kazdemu trojuholniku si pamatat kolko uz ma cervenych hran
 */

struct hrana {
    int x,y;
    hrana(int a=-1, int b=-1) : x( min(a,b) ), y( max(a,b) ) {}
};
bool operator< (const hrana &A, const hrana &B) { if (A.x != B.x) return A.x < B.x; return A.y < B.y; }
bool operator== (const hrana &A, const hrana &B) { return (A.x == B.x) && (A.y == B.y); }
bool operator!= (const hrana &A, const hrana &B) { return !(A==B); }

struct trojuholnik {
    int x,y,z;
    trojuholnik(int a=-1, int b=-1, int c=-1) { x=min(a,min(b,c)); z=max(a,max(b,c)); y=a+b+c-x-z; }
};
bool operator< (const trojuholnik &A, const trojuholnik &B)
    { if (A.x != B.x) return A.x < B.x; if (A.y != B.y) return A.y < B.y; return A.z < B.z; }
bool operator== (const trojuholnik &A, const trojuholnik &B) { return (A.x == B.x) && (A.y == B.y) && (A.z == B.z); }
bool operator!= (const trojuholnik &A, const trojuholnik &B) { return !(A==B); }

static int N;                                       // pocet vrcholov mnohouholnika

static map<hrana,int> hrana_na_id;                  // hrana_na_id[h] je poradie hrany h vo vyssie popisanom ocislovani
static vector<hrana> poradie;                       // toto je to spominane poradie hran
static vector<trojuholnik> poradie_trojuholnikov;   // a toto je jemu zodpovedajuce poradie trojuholnikov
static vector<int> cervenych_hran_v_trojuholniku;   // navyse si ku kazdemu z nich pamatame, kolko uz ma cervenych hran

static set<hrana> cervene_hrany;                    // mnozina cervenych hran; pouzijeme ju len pri uplne poslednom tahu
static hrana posledny_tah_maru;
static trojuholnik vyherny_trojuholnik;             // ak uz existuje trojuholnik, ktory vieme dokoncit, toto je on

inline int prevv(int x) { return ((x+N-1)%N); }
inline int nextv(int x) { return ((x+1)%N); }


void hraci_plan(int cN, int U[][2]) {
    N = cN;
    // spravime si mnozinu uhlopriecok a mnozinu vsetkych hran
    set<hrana> uhlopriecky;
    for (int n=0; n<N-3; ++n) uhlopriecky.insert( hrana( U[n][0]-1, U[n][1]-1 ) );
    set<hrana> hrany;
    for (int n=0; n<N-3; ++n) hrany.insert( hrana( U[n][0]-1, U[n][1]-1 ) );
    for (int n=0; n<N; ++n) hrany.insert( hrana( n, nextv(n) ) );

    // pre kazdu uhlopriecku najdeme trojuholniky na jej stranach
    set<trojuholnik> T1;
    for (int n=0; n<N-3; ++n) {
        int u = U[n][0]-1, v = U[n][1]-1;
        if (hrany.count(hrana(prevv(u),v))) T1.insert(trojuholnik(u,v,prevv(u)));
        if (hrany.count(hrana(nextv(u),v))) T1.insert(trojuholnik(u,v,nextv(u)));
        if (hrany.count(hrana(prevv(v),u))) T1.insert(trojuholnik(u,v,prevv(v)));
        if (hrany.count(hrana(nextv(v),u))) T1.insert(trojuholnik(u,v,nextv(v)));
    }

    // pre kazdy trojuholnik najdeme uhlopriecky ktore obsahuje
    // zaroven pre kazdu uhlopriecku najdeme trojuholniky ktore ju obsahuju
    map<trojuholnik,vector<hrana> > uhlopriecky_trojuholnika;
    map<hrana, vector<trojuholnik> > trojuholniky_uhlopriecky;
    for (auto t : T1) {
        if (uhlopriecky.count(hrana(t.x,t.y))) {
            uhlopriecky_trojuholnika[t].push_back(hrana(t.x,t.y));
            trojuholniky_uhlopriecky[ hrana(t.x,t.y) ].push_back(t);
        }
        if (uhlopriecky.count(hrana(t.x,t.z))) {
            uhlopriecky_trojuholnika[t].push_back(hrana(t.x,t.z));
            trojuholniky_uhlopriecky[ hrana(t.x,t.z) ].push_back(t);
        }
        if (uhlopriecky.count(hrana(t.y,t.z))) {
            uhlopriecky_trojuholnika[t].push_back(hrana(t.y,t.z));
            trojuholniky_uhlopriecky[ hrana(t.y,t.z) ].push_back(t);
        }
    }

    // najdeme trojuholnik ktory ma len jedneho suseda
    trojuholnik curt(-1,-1,-1);
    for (auto t : T1) if (uhlopriecky_trojuholnika[t].size()==1u) { curt=t; break; }
    if (N==3) curt = trojuholnik(0,1,2);

    // od tohto trojuholnika postupne prejdeme vsetky a vyplnime hrany do poradia
```

```
    hrana posledna_uhlopriecka(-1,-1);
    while (true) {
        { hrana h(curt.x,curt.y); if (!uhlopriecky.count(h)) poradie.push_back(h); }
        { hrana h(curt.x,curt.z); if (!uhlopriecky.count(h)) poradie.push_back(h); }
        { hrana h(curt.y,curt.z); if (!uhlopriecky.count(h)) poradie.push_back(h); }
        poradie_trojuholnikov.push_back(curt);

        hrana dalsia_uhlopriecka(-1,-1);
        { hrana h(curt.x,curt.y); if (uhlopriecky.count(h) && h!=posledna_uhlopriecka) dalsia_uhlopriecka=h; }
        { hrana h(curt.x,curt.z); if (uhlopriecky.count(h) && h!=posledna_uhlopriecka) dalsia_uhlopriecka=h; }
        { hrana h(curt.y,curt.z); if (uhlopriecky.count(h) && h!=posledna_uhlopriecka) dalsia_uhlopriecka=h; }

        if (dalsia_uhlopriecka == hrana(-1,-1)) {
            // sme na konci
            break;
        } else {
            // ideme na dalsi trojuholnik
            poradie.push_back(dalsia_uhlopriecka);
            posledna_uhlopriecka = dalsia_uhlopriecka;
            for (auto t : trojuholniky_uhlopriecky[dalsia_uhlopriecka]) if (t != curt) { curt=t; break; }
        }
    }

    for (int n=0; n<2*N-3; ++n) hrana_na_id[poradie[n]]=n;
    posledny_tah_maru = hrana(-1,-1);
    vyherny_trojuholnik = trojuholnik(-1,-1,-1);
    cervenych_hran_v_trojuholniku.resize(N-2,0);
}

void zapln_trojuholnik(int x) {
    ++cervenych_hran_v_trojuholniku[x];
    if (cervenych_hran_v_trojuholniku[x] == 2) vyherny_trojuholnik = poradie_trojuholnikov[x];
}

void zacerven_hranu(const hrana &h) {
    cervene_hrany.insert(h);
    int id = hrana_na_id[h];
    if (id % 2) zapln_trojuholnik(id/2); else {
        if (id/2>0) zapln_trojuholnik(id/2 - 1);
        if (id/2<N-2) zapln_trojuholnik(id/2);
    }
}

void tah_usamca (int hr[2]) {
    if (posledny_tah_maru == hrana(-1,-1)) {
        // prvy tah hry ide do stredu
        zacerven_hranu( poradie[N-2] );
        hr[0] = poradie[N-2].x+1;
        hr[1] = poradie[N-2].y+1;
        return;
    }
    if (vyherny_trojuholnik != trojuholnik(-1,-1,-1)) {
        // uz vieme vyhrat
        trojuholnik t = vyherny_trojuholnik;
        { hrana h(t.x,t.y); if (!cervene_hrany.count(h)) { hr[0]=h.x+1; hr[1]=h.y+1; return; } }
        { hrana h(t.x,t.z); if (!cervene_hrany.count(h)) { hr[0]=h.x+1; hr[1]=h.y+1; return; } }
        { hrana h(t.y,t.z); if (!cervene_hrany.count(h)) { hr[0]=h.x+1; hr[1]=h.y+1; return; } }
    }
    // tahame symetricky s poslednym tahom Maru
    hrana h = poradie[ 2*N - 4 - hrana_na_id[ posledny_tah_maru ] ];
    zacerven_hranu(h);
    hr[0]=h.x+1; hr[1]=h.y+1;
}

void tah_maru (int hr[2]) {
    posledny_tah_maru = hrana( hr[0]-1, hr[1]-1 );
    zacerven_hranu( posledny_tah_maru );
}
```

**DVADSIATY ÔSMY ROČNÍK OLYMPIÁDY V INFORMATIKE**

Príprava úloh: Michal Anderle, Vladimír Boža, Michal Forišek, Ján Hozza, Marek Špano
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2013