A-III-1 Oriešková čokoláda

Napriek tomu, že sa hra môže vyvíjať viacerými spôsobmi, celkový počet rôznych pozícii je len $rs + 1$. Špeciálnou pozíciou je celá zjedená čokoláda. Každá iná pozícia počas hry má tvar obdĺžnika, a ten je jednoznačne určený súradnicami jeho pravého dolného rohu.

V ľubovoľnej takejto hre (tzv. symetrickej hre s úplnou informáciou) si môžeme pozície rozdeliť na dva typy: *vyhrávajúce* a *prehrávajúce*. Pozícia je vyhrávajúca, ak pre hráča, ktorý je v nej na ťahu, existuje stratégia, ktorej keď sa bude držať, tak určite vyhrá. Pozícia je prehrávajúca, ak takáto stratégia existuje pre jeho protihráča.

Prázdna čokoláda a tie pozície, z ktorých neexistuje žiaden povolený ťah, sú prehrávajúce pozície.

Pre zvyšné pozície rozhodneme či sú vyhrávajúce alebo prehrávajúce na základe jednoduchého rekurzívneho vzťahu. Aby sme vyhodnotili nejakú pozíciu, stačí sa pozrieť na pozície, do ktorých sa z nej dá dostať jedným ťahom. Ak existuje z aktuálnej pozície ťah do nejakej prehrávajúcej pozície, tak je aktuálna pozícia zjavne vyhrávajúca (a dotyčný ťah je prvým ťahom jednej možnej vyhrávajúcej stratégie). A naopak, ak všetky ťahy vedú do vyhrávajúcich pozícií (teda pozícií, v ktorých bude náš protihráč následne mať vyhrávajúcu stratégiu), tak je aktuálna pozícia nutne prehrávajúca.

Zostáva iba zistiť, ktoré ťahy sú z danej pozície povolené. Potrebujeme teda vedieť zistiť, či je v poslednom riadku/stĺpci konkrétneho obdĺžnika párny počet orieškov. Priamočiare riešenie na to použije cyklus: prejdeme celý riadok a celý stĺpec a oriešky si spočítame. Takéto riešenie teda každú konkrétnu pozíciu spracuje v čase $O(r + s)$. Celková časová zložitosť tohto algoritmu je teda $O(rs(r + s))$, jeho pamäťová zložitosť je $O(rs)$.

Pokúsme sa vylepšiť vyššie uvedený algoritmus. Pre každé políčko (i, j) si okrem toho, či ide o vyhrávajúcu alebo prehrávajúcu pozíciu, budeme pamätať dve čísla $x_{i,j}$ a $y_{i,j}$: súčet počtov orieškov v spodnom riadku a v pravom stĺpci zodpovedajúceho obdĺžnika.

Keď sa teraz pozrieme na nové políčko (i, j), hodnotu $x_{i,j}$ vieme vypočítať v konštantnom čase: $x_{i,j} = x_{i,j-1} + p_{i,j}$. (Teda sa pozrieme na podobný súčet pre o 1 užší obdĺžnik, a prirátame políčko ktoré v spodnom riadku pribudlo. Pritom predpokladáme, že $x_{i,0} = 0$.)

Podobne vieme hodnotu $y_{i,j}$ určiť z $y_{i-1,j}$ pripočítaním $p_{i,j}$. No a parita hodnôt $x_{i,j}$ a $y_{i,j}$ nám rovno povie, ktoré ťahy sú v danej situácii platné a ktoré nie.

Takto dostávame algoritmus s časovou aj pamäťovou zložitosťou $O(rs)$.

Listing programu (Python)

```
# načítame vstup
R, S = [ int(x) for x in input().split() ]
P = [ [ int(x) for x in input().split() ] for r in range(R) ]

# predpočítame súčty v riadkoch a vo stĺpcoch
rowsum = [ [ 0 for s in range(S+1) ] for r in range(R+1) ]
colsum = [ [ 0 for s in range(S+1) ] for r in range(R+1) ]
for r in range(R):
    for s in range(S):
        rowsum[r+1][s+1] = rowsum[r+1][s] + P[r][s]
        colsum[r+1][s+1] = colsum[r][s+1] + P[r][s]

# o každej pozícii zistíme, či je víťazná alebo nie
winning = [ [ False for s in range(S+1) ] for r in range(R+1) ]
for r in range(1,R+1):
    for s in range(1,S+1):
        if colsum[r][s]%2==0 and not winning[r][s-1]: winning[r][s] = True
        if rowsum[r][s]%2==0 and not winning[r-1][s]: winning[r][s] = True

print('Anicka' if winning[R][S] else 'Baska')
```

Na záver už len podotkneme, že keď hodnoty $x_{i,j}$, $y_{i,j}$ a informácie o vyhrávajúcich/prehrávajúcich pozíciách počítame postupne po riadkoch, stačí si vždy pamätať len jeden predchádzajúci riadok. Takto teda vieme bez zhoršenia časovej zložitosti zlepšiť pamäťovú zložitosť na $O(s)$.

A-III-2 Jarné upratovanie

Ako bolo v zadaní veľakrát spomenuté, v tejto úlohe ste mali narábať šikovne so súbormi. Prvé, čo ste mohli spraviť je postupne pre každé číslo raz prejsť celý súbor a overiť, či sa v ňom vyskytuje. Za takéto riešenie ste, samozrejme, veľa bodov získať nemohli – a ani za jeho drobné vylepšenia, ktoré napríklad naraz kontrolovali vždy 1000 čísel.

Pre sofistikovanejšie riešenia už budeme musieť použiť ďalšie súbory.

Delenie intervalov

Základnou myšlienkou lepšieho riešenia je šikovné rozdeľovanie problému na menšie. Náš problém vyzerá nasledovne: Máme súbor, v súbore sú čísla z nejakého intervalu, a nejaké z nich chýbajú. Pôvodný „veľký“ interval si rozdelíme na niekoľko nových „malých“ intervalov tak, aby boli všetky malé intervaly (aspoň zhruba) rovnako veľké. Pre každý malý interval si spravíme súbor, do ktorého budeme zapisovať čísla, ktoré doň patria. A navyše si pre každý malý interval budeme v pamäti udržiavať počítadlo hovoriace koľko čísel z neho sme už videli.

Celé takéto prerozdelenie veľkého intervalu na malé vieme zjavne spraviť v čase nanajvýš priamo úmernom dĺžke veľkého intervalu.

Ako nám takéto delenie intervalu pomôže? Tak, že v ňom teraz vieme pokračovať rekurzívne ďalej. V niektorých malých intervaloch nič nechýba – počet čísel, ktoré sme do nich dali, je rovný ich dĺžke. Tie môžeme spokojne odignorovať. No a každý malý interval, v ktorom niečo chýba, je teraz nový problém, ktorý môžeme riešiť samostatne, nezávisle na iných.

Delenie na polovicu

Vo vyššie popísanom riešení sme úmyselne neuviedli jeden detail: na koľko malých intervalov chceme deliť veľký? Pozrime sa najskôr na možnosť, kedy počet malých intervalov bude 2. Inými slovami, aktuálny interval vždy rozdelíme na polovice, zistíme, koľko čísel chýba v ktorej z nich, a potom sa zavoláme na každú, v ktorej chýba aspoň jedno.

Akú najhoršiu časovú zložitosť môže mať toto riešenie?

Na začiatku máme jeden súbor so zhruba n číslami. Ten rozdelíme na dva súbory so zhruba $n/2$ číslami, čo nám zaberie $O(n)$ krokov výpočtu. Každý z nich ďalej rozdelíme na dva súbory s $n/4$ číslami, čo nám opäť dokopy zaberie $O(n)$ krokov výpočtu. A tak ďalej.

Keď už budeme mať dostatočne malé intervaly, nastane vytúžená situácia, keď niektoré z nich budú kompletné. Tie nemusíme spracúvať a môžeme ich odignorovať, čím ušetríme čas.

Lenže ak nám chýba k čísel a tie sú rozmiestnené zhruba rovnomerne, tak počas prvých $\log_2 k$ delení nášho intervalu nám budú vznikať samé menšie intervaly, v ktorých stále ešte niečo chýba.

(Napríklad ak by bolo $k = 16$, tak najhoršie, čo sa nám môže stať, je, že rozdelíme súbor na polovice, polovice na štvrtiny, štvrtiny na osminy, osminy na šestnástiny, a ešte stále v každej šestnástine chýba práve jedno z čísel.)

Dokopy teda v takomto najhoršom prípade určite spravíme aspoň rádovo $n \log k$ krokov výpočtu. Dá sa dokázať aj silnejšie tvrdenie: toto riešenie má v najhoršom prípade časovú zložitosť $\Theta(n \log k)$.

Za zmienku ešte stojí, že keby sme vôbec nerobili optimalizáciu s tým, že intervaly, v ktorých nič nechýba, netreba spracúvať, dostali by sme časovú zložitosť $\Theta(n \log n)$.

Košaté delenie

Keď nám teda nestačí binárne delenie, čo takto delenie na k intervalov? Ukazuje sa, že to sa nám stále zmestí do obmedzenia na počet súborov, ak budeme šikovní.

V ľubovoľnom okamihu budeme mať na disku niekoľko súborov, ktoré predstavujú najvzájom disjunktné intervaly v ktorých niečo chýba. Keďže dokopy chýba k čísel, týchto súborov bude v ľubovoľnej chvíli nanajvýš k.

V každej iterácii si jeden ľubovoľný súbor zoberieme a prerozdelíme ho na k menších intervalov. Na toto nám bude stačiť k pomocných súborov. Do nich prerozdelíme pôvodný interval. Následne tie z nich, ktoré sú plné, zmažeme, a tie, v ktorých niečo chýba, pridáme medzi intervaly čakajúce na ďalšie spracovanie.

(Kedy to celé končí? Keď sa dostaneme k súboru predstavujúcemu interval dĺžky 1, v ktorom ale neleží žiadne z čísel zo vstupu, nedelíme ho ďalej – ale jednoducho práve nájdené chýbajúce číslo pridáme do výstupného poľa.)

Pozrime sa, či nám to pomohlo s počtom čítaní a zapisovaní. Pri prvom delení potrebujeme načítať aj zapísať všetky čísla, teda spraviť n čítaní zo súboru a n zápisov do súboru. Tým nám vzniklo k intervalov, každý dĺžky n/k. Potom, v najhoršom prípade, potrebujeme ešte raz postupne prerozdeliť všetky tieto intervaly na menšie, čím nám vzniknú intervaly dĺžky n/k^2. Aj toto prerozdeľovanie dokopy vyžaduje n čítaní a n zápisov.

Spomedzi týchto intervalov dĺžky n/k^2 však iba nanajvýš k má nejaké chýbajúce číslo. Ďalej teda pokračujeme s intervalmi, ktorých súčet dĺžok je nanajvýš $k \cdot (n/k^2) = n/k$. Ich ďalšie prerozdelenie si teda dokopy vyžiada už len n/k čítaní a n/k zápisov. Podobnú úvahu vieme spraviť aj po ďalšej iterácii delenia, tentoraz dostaneme intervaly, ktorých súčet dĺžok je nanajvýš n/k^2. A tak ďalej.

Dokopy teda náš program spraví čítaní aj zápisov čísel

$$n + n + \frac{n}{k} + \frac{n}{k^2} + \frac{n}{k^3} + \cdots < 3n$$

a teda jeho časová zložitosť je lineárna. (Vyššie uvedený odhad platí za predpokladu, že k je aspoň 2. Pre $k = 1$ použijeme lineárny algoritmus z domáceho kola.)

Listing programu (C++)

```
#include <stdio.h>
#include <stdlib.h>
#define MAXK 100

/* Správa súborov, ktorá je tu hlavne na to, aby sme zaručili neprekračovanie počtu súborov */
static unsigned num_tmp_files;
static FILE *file_pool[2 * MAXK];
static unsigned file_pool_size;

static FILE * get_temporary_file (void) {
    FILE *ret;
    if (file_pool_size > 0) {
        ret = file_pool[--file_pool_size];
        rewind (ret);
    } else {
        char tmpname[10];
        sprintf (tmpname, "tmp%d", num_tmp_files++);
        ret = fopen (tmpname, "w+");
    }
    return ret;
}

static void release_tmp_file (FILE *f) { file_pool[file_pool_size++] = f; }

static void remove_temporary_files (void) {
    unsigned i;
    for (i = 0; i < num_tmp_files; i++) {
        char tmpname[10];
        sprintf (tmpname, "tmp%d", i);
        remove (tmpname);
    }
}

typedef struct { unsigned f, t, c; FILE *s; } interval;

/* Zoznam intervalov, ktoré ešte potrebujeme spracovať */

static interval ke_zpracovani[MAXK];
static unsigned int_ke_zprac;

/* Spracuj interval <f,t>, v ktorom je c čísel, nachádza sa v súbore s a deliť ho budeme na d častí */
static void zpracuj_interval (unsigned f, unsigned t, unsigned c, FILE *s, unsigned d) {
    FILE *tmpfs[MAXK];
    unsigned pocet[MAXK];
    unsigned m = t - f + 1;
    unsigned i, plen, a, ai;

    for (i = 0; i < d; i++) pocet[i] = 0;

    /* Určíme si veľkosti jednotlivých intervalov a priradíme im pracovné súbory, ak spracúvame krátky interval,
    * rozdeľujeme ho na podintervaly dĺžky 1 */
    if (m > d) {
        for (i = 0; i < d; i++)
        tmpfs[i] = get_temporary_file ();
```

```
        plen = (m + d - 1) / d;
    } else {
        plen = 1;
    }

    /* Porozdeľujeme čísla do podintervalov  */
    for (i = 0; i < c; i++) {
        fscanf (s, "%u", &a);
        ai = (a - f) / plen;
        pocet[ai]++;
        if (m > d)
        fprintf (tmpfs[ai], "%d\n", a);
    }

    /* Intervaly, v ktorých chýba aspoň jeden packet zaradíme do zoznamu intervalov na spracovanie, prípadne
     * ich vypíšeme, ak v nich chýbajú všetky packety */
    for (i = 0; i < d; i++) {
        unsigned l = f + plen * i, u = l + plen - 1;
        interval nint;
        if (u > t) u = t;
        if (pocet[i] == u - l + 1) {
            if (m > d)
            release_tmp_file (tmpfs[i]);
            continue;
        }
        if (pocet[i] == 0) {
            unsigned j;
            for (j = l; j <= u; j++)
            printf ("%d ", j);
            if (m > d)
            release_tmp_file (tmpfs[i]);
            continue;
        }
        nint.f = l;
        nint.t = u;
        nint.c = pocet[i];
        rewind (tmpfs[i]);
        nint.s = tmpfs[i];
        ke_zpracovani[int_ke_zprac++] = nint;
    }
}

int main(void) {
    FILE *fin = fopen ("pakety.txt", "r");
    unsigned n, k, d;

    fscanf (fin, "%u%u", &n, &k);
    d = k < 2 ? 2 : k;

    int_ke_zprac = 0;
    zpracuj_interval (1, n, n - k, fin, d);
    while (int_ke_zprac > 0) {
        interval a = ke_zpracovani[--int_ke_zprac];
        zpracuj_interval (a.f, a.t, a.c, a.s, d);
        release_tmp_file (a.s);
    }
    remove_temporary_files ();
    return 0;
}
```

A-III-3 Magické siete

Postupne si ukážeme riešenia všetkých troch podúloh v poradí, v akom boli zadané.

Simulácia CH pomocou ONE

Ako prvé si môžeme všimnúť, že podmienkou ONE(n, n, j) si vieme vynútiť, že n musí byť 0 a j musí byť 1. Vieme teda mať v našej sieti pomocné premenné ktoré majú zaručene hodnotu nula a jedna.

Tieto premenné môžeme teraz ďalej používať. Napríklad nám bude užitočná podmienka ONE(x, x', n). Keďže už vieme, že $n = 0$, musí mať hodnotu 1 práve jedna z premenných x a x'. Inými slovami, x' je negáciou x.

Ďalšia užitočná podmienka, ktorú vieme pomocou ONE simulovať, je podmienka NAND$_2(y, z)$ (binárny NAND, teda not-and). Táto podmienka je splnená, ak aspoň jedna z premenných y a z nadobúda hodnotu 0. Ako na to? Jednoducho: zoberieme si novú pomocnú premennú p, ktorá nie je použitá nikde inde, a potom nám stačí jediná podmienka: ONE(y, z, p). Pre pevne zvolené y a z vieme nájsť vyhovujúce p práve vtedy, ak neplatí $y = z = 1$.

Tým už máme vlastne polovicu konštrukcie obmedzenia CH hotovú. Ostáva nám ešte zabezpečiť, aby aspoň jedna z premenných x a y mala hodnotu 1.

Ale aj to spravíme ľahko. Totiž podmienka „aspoň jedna z premenných x a y má hodnotu 1“ je ekvivalentná s podmienkou „aspoň jedna z premenných x' a y' má hodnotu 0“, pričom x' a y' sú negácie x a y. No a tie vieme vyrobiť a tak túto podmienku zapísať.

Výsledná sieť simulujúca $\mathrm{CH}(x, y, z)$ teda obsahuje nasledujúce obmedzenia:

- $\mathrm{ONE}(n, n, j)$ (n je 0 a j je 1)
- $\mathrm{ONE}(y, z, p)$ (aspoň jedno z y a z je 0)
- $\mathrm{ONE}(x, x', n)$ (x' je negáciou x)
- $\mathrm{ONE}(y, y', n)$ (y' je negáciou y)
- $\mathrm{ONE}(x', y', q)$ (aspoň jedno z x' a y' je 0)

Nemožnosť simulácie ONE pomocou CH

Na dôkaz tohto tvrdenia zafunguje technika, ktorú ukazuje príklad 3 v študijnom texte. Ukážeme teda, že pre ľubovoľnú sieť zloženú len z obmedzení typu CH vieme zobrať viaceré platné ohodnotenia a skombinovať ich do iného, ktoré na jednej strane tiež musí byť platné, ale na druhej strane nebude zodpovedať obmedzeniu, ktoré sa snažíme simulovať.

Pripomeňme si pomocnú funkciu maj_3 použitú v príklade 3. Táto funkcia (nazvaná podľa anglického slova „majority“) vracia na výstup tú z hodnôt 0 a 1, ktorá je na vstupe častejšie. Teda napr. $\mathrm{maj}_3(1,1,1) = \mathrm{maj}_3(1,0,1) = 1$, ale $\mathrm{maj}_3(0,0,1) = 0$.

Všimnime si, ako funguje obmedzenie CH. Predpokladajme, že trojice premenných (a_1, b_1, c_1), (a_2, b_2, c_2) a (a_3, b_3, c_3) spĺňajú toto obmedzenie.

Vypočítajme teraz $a = \mathrm{maj}_3(a_1, a_2, a_3)$, $b = \mathrm{maj}_3(b_1, b_2, b_3)$ a $c = \mathrm{maj}_3(c_1, c_2, c_3)$. Tvrdíme, že trojica (a, b, c) tiež nutne spĺňa obmedzenie CH.

Prečo je tomu tak?

Keďže naše pôvodné trojice premenných spĺňajú obmedzenie CH, máme aspoň jednu 1 medzi a_1 a b_1, aspoň jednu medzi a_2 a b_2 a aspoň jednu medzi a_3 a b_3. Potom ale musí platiť, že aspoň jedna z premenných a a b má hodnotu 1. (Ak by totiž bolo $a = b = 0$, tak môže mať hodnotu 1 najviac jedna z premenných a_i a najviac jedna z premenných b_i.)

Analogicky vieme ukázať, že aspoň jedna z nových premenných b a c bude mať hodnotu 0.

Predstavme si teraz, že máme sieť zloženú len z obmedzení typu CH, ktorá simuluje obmedzenie $\mathrm{ONE}(x, y, z)$. Obmedzenie ONE je splnené pre tri rôzne kombinácie hodnôt x, y a z: (0,0,1), (0,1,0) a (1,0,0). Pre každú z týchto kombinácií hodnôt teda existuje aspoň jedno prípustné ohodnotenie všetkých premenných v našej sieti (vrátane pomocných).

Vyberme si teraz tri takéto ohodnotenia premenných tak, aby každé zodpovedalo inej prípustnej kombinácii hodnôt x, y a z. Pomocné tvrdenie, ktoré sme si dokázali vyššie, nám teraz hovorí, že vieme zostrojiť nové, štvrté prípustné ohodnotenie všetkých premenných – a to tak, že pre každú premennú zoberieme maj_3 hodnôt, ktoré nadobúda v prvých troch ohodnoteniach.

V takto zostrojenom ohodnotení však platí $x = y = z = 0$, a to je spor s predpokladom, že naša sieť simuluje obmedzenie ONE.

Simulácia ľubovoľného obmedzenia pomocou ONE

Pripomeňme si zo študijného textu, že slabé obmedzenie $\mathrm{S}_{h_1 \ldots h_n}$ je také obmedzenie, ktoré je splnené pre všetky ohodnotenia premenných okrem jediného – ohodnotenia $h_1, \ldots, h_n$. Teda napr. slabé obmedzenie $\mathrm{S}_{001}(x, y, z)$ je splnené vždy okrem prípadu kedy $x = 0$, $y = 0$ a $z = 1$.

Zjavne každé obmedzenie X vieme simulovať pomocou vhodnej sady slabých obmedzení: jednoducho postupne „vymenujeme“ všetky kombinácie premenných, pre ktoré naše obmedzenie X nemá byť splnené.

Na dôkaz nášho tvrdenia nám teda stačí dokázať, že pomocou ONE vieme simulovať ľubovoľné slabé obmedzenie.

Keďže si pomocou obmedzenia typu ONE vieme vyrobiť negáciu ľubovoľnej premennej, stačí nám dokonca vedieť simulovať len slabé obmedzenia typu $S_{11\ldots1}$. Každé iné slabé obmedzenie vieme totiž ľahko previesť na takéto. Ukážeme si to na príklade.

Predpokladajme napríklad, že chceme simulovať obmedzenie $S_{0110}(a, b, c, d)$, teda obmedzenie, ktoré je splnené vždy okrem situácie kedy $(a, b, c, d) = (0, 1, 1, 0)$. Pomocou obmedzení $\mathrm{ONE}(a, a', p_1)$ a $\mathrm{ONE}(d, d', p_2)$ si vieme vynútiť, že a' a d' sú negácie premenných a a d. No a teraz nám stačí vedieť odsimulovať obmedzenie $S_{1111}(a', b, c, d')$.

Analogickú konštrukciu vieme zjavne spraviť pre ľubovoľný počet premenných a ľubovoľné iné slabé obmedzenie.

Ostáva nám teda posledný krok: pre ľubovoľné $k \geq 1$ potrebujeme ukázať, ako pomocou ONE simulovať slabé obmedzenie ktoré hovorí, že daných k premenných nesmie mať ako hodnoty samé 1.

Ručne vyriešime malé prípady:

- $S_1(a)$ vieme simulovať obmedzením $\mathrm{ONE}(a, a, p)$, kde p je nová, doteraz nikde nepoužitá premenná
- $S_{11}(a, b)$ vieme analogicky simulovať obmedzením $\mathrm{ONE}(a, b, p)$.
- $S_{111}(a, b, c)$ vieme simulovať nasledovnou sieťou: $S_{11}(a, d)$, $S_{11}(b, e)$, $S_{11}(c, f)$, $\mathrm{ONE}(d, e, f)$.

  Ako toto funguje? Posledné obmedzenie si vynúti, že práve jedna z pomocných premenných d, e a f je rovná 1. Nech je to napríklad d. Potom z podmienky $S_{11}(a, d)$ dostávame, že a musí byť 0, zatiaľ čo b a c môžu byť ľubovoľné. Jediné ohodnotenie (a, b, c) pre ktoré neexistujú žiadne prípustné (d, e, f) je teda ohodnotenie $(1, 1, 1)$, čo je presne to, čo sme chceli.

No a pre $k \geq 4$ teraz uvedieme všeobecnú konštrukciu indukciou. Predpokladajme, že sme už zostrojili sieť simulujúcu slabé obmedzenie $S_{11\ldots1}(x_1, \ldots, x_{k-1})$ s $k-1$ vstupmi.

Uvažujme teraz sieť tvorenú nasledovnými obmedzeniami: $S_{11\ldots1}(x_1, \ldots, x_{k-2}, z)$, $S_{111}(x_{k-1}, x_k, z')$, a navyše $\mathrm{ONE}(n, n, j)$ a $\mathrm{ONE}(z, z', n)$. Táto sieť simuluje slabé obmedzenie, ktoré hovorí, že spomedzi x_1 až x_k musí mať niektoré hodnotu 0.

Ako funguje? Posledné dve obmedzenia už poznáme: hovoria, že $n = 0$, $j = 1$ a z' je negáciou z. Prvé obmedzenie nám hovorí, že premenné $x_1, \ldots, x_{n-2}, z$ nesmú byť všetky naraz rovné 1. Inými slovami, ak sú $x_1, \ldots, x_{n-2}$ všetky 1, musí z byť 0. Analogicky z druhého obmedzenia dostávame, že ak x_{k-1} aj x_k sú 1, musí byť z' rovné 0. Lenže z a z' nemôžu byť naraz rovné 0, a teda nemôžu byť naraz úplne všetky x_i rovné 1. Naopak, ľahko nahliadneme, že akonáhle je čo len jedno z x_i rovné 0, vieme nájsť vyhovujúce ohodnotenie z a z'.

TRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Jozef Brandys, Michal Forišek, Askar Gafurov, Jaroslav Petrucha, Michaela Šandalová
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2015