Riešenia kategórie B

B-II-1 Nutela

Tí, ktorí ste nevymysleli vzorové riešenie, nemali by ste hneď házdať flintu do žita. Za 3 body stačilo napísať riešenie, ktoré skontroluje všetky možné dvojice téglikov.

No keďže 3 body nie sú veľa, radšej si ukážeme, ako bez veľkej námahy vylepšiť toto riešenie na celých 8 bodov. Nebudeme skúšať dvojice, ale skúsime si postupne vybrať každý téglik ako prvú nutelu z dvojice a následne druhý téglik nejako, čo najrýchlejšie dohľadať.

Keď máme zvolený prvý téglik, vieme jeho veľkosť, označme si ju c. Druhý téglik teda musí mať veľkosť od $a - c$ po $b - c$, a hľadáme ho v usporiadanom poli. To vieme robiť binárnym vyhľadávaním, podobne, ako keď hľadáme meno v telefónnom zozname, alebo hádame, aké číslo si myslí kamarát, či nebodaj hľadáme poklad s pomocou vysávača (viď úlohu 3).

Keďže binárne vyhľadávanie má časovú zložitosť $O(\log n)$, celý algoritmus bude bežať v čase $O(n \log n)$. Pamäťová zložitosť bude $O(n)$.

Vzorové riešenie

Vieme, že ak tam niekde správne nutely sú, budú v intervale od prvého po posledný téglik (vrátane). (Kde inde by mohli byť?) Počas algoritmu sa budeme tento interval snažiť zmenšovať, až kým buď nenájdeme správnu dvojicu, alebo neskončíme s intervalom dĺžky 1 a vypíšeme „nedá sa“.

Stačí dokola opakovať nasledujúce zmenšovanie.

Majme nejaký rad $m > 1$ utriedených téglikov s veľkosťami $x_1 < x_2 < \cdots < x_m$. Potom môžu nastať tri možnosti.

- Buď prvý a posledný téglik majú dobrý súčet ($a \leq x_1 + x_m \leq b$), vtedy vypíšeme ich hodnoty a skončíme.
- Môžu mať aj primalý súčet ($x_1 + x_m < a$). Vtedy vieme, že prvý z téglikov určite nebude patriť do správnej dvojice. Totiž ak uvažujeme dvojicu x_1 a x_i pre nejaké i, tak vieme, že $x_1 + x_i \leq x_1 + x_m < a$, čo je príliš málo na to, aby to bola správna dvojica. Preto môžeme prvý téglik zahodiť a zaujímať sa len o zvyšných $m - 1$.
- Posledná možnosť je, že prvý a posledný téglik majú priveľký súčet ($x_1 + x_m > b$). Veľmi podobnou úvahou zistíme, že môžeme z intervalu uvažovaných téglikov vynechať posledný téglik.

Pokiaľ sa na polici žiadna vyhovujúca dvojica téglikov nenachádzala, skončíme časom s jediným téglikom a v tej chvíli už môžeme dať zápornú odpoveď.

Časová zložitosť bude linárna od počtu téglikov, $O(n)$, pretože v každom kroku spravíme nejaké výpočty, ktoré prebehnú v konštantnom čase, a následne ak sme ešte neskončili, tak jeden téglik zahodíme. No a dokopy môžeme zahodiť nanajvýš $n - 1$ téglikov. Rýchlejšie sa túto úlohu už nedá, lebo musíme aj tak načítať celý vstup. (Čo keby správna dvojica bola až na konci?)

Pamäťová zložitosť bude tiež $O(n)$, pretože si uchovávame všetky tégliky naraz v pamäti. (Zlepšiť to takisto nevieme, lebo v okamihu, keď načítavame posledný téglik, potrebujeme si pamätať ostatné. Keby sme si nejaký nepamätali, môže sa stať, že jediná správna dvojica bude posledný téglik a ten, ktorý si nepamätáme.)

Listing programu (Pascal)

```
var n,a,b:longint;
    i,zac,kon:longint;
    x:array[1..1000000] of longint;
begin
    read(n,a,b);
    for i := 1 to n do read(x[i]);
    zac := 1;
    kon := n;
    while kon-zac>0 do begin
        if x[zac] + x[kon] < a then begin
            inc(zac);
            continue;
        end;
```

```
        if x[zac] + x[kon] > b then begin
            dec(kon);
            continue;
        end;
        writeln(x[zac],'␣',x[kon]);
        exit;
    end;
    writeln('nedá␣sa');
end.
```

B-II-2 Pošta

Tento vzorák, vás dúfam naučí mnohým fintám pre pravých chlapov a pravé ženy. Najskôr si ukážeme, ako sa dá na strome hľadať cesta medzi dvoma vrcholmi a následne si ukážeme, že vo vzorovom riešení tie cesty ani nepotrebujeme. Prekvapivé? Tak to si ešte počkajte. A samozrejme, keďže sme pravé ženy a praví chlapi, nebudeme sa okúnať s metaforami ako domy a ulice, ale ako sa patrí v teórii grafov, domy nazveme vrcholy a ulice hrany v našom grafe.

Prehľadávanie do hĺbky

Prvým problémom, ktorý riešime v tejto úlohe, je hľadanie cesty medzi dvoma vrcholmi. Ak by sme toto vedeli, hneď by sme mali triviálne riešenie – pre každú dvojicu vrcholov nájdeme cestu a jej dĺžku zarátame do výsledku. Ako však nájsť cestu medzi vrcholmi x a y?

V domácom kole sme sa zoznámili s prehľadávaním do šírky, čo bol jednoduchý algoritmus na hľadanie najkratšej cesty v neohodnotenom grafe. Tu pracujeme tiež s neohodnoteným grafom a preto sa tento algoritmus dá použiť. Náš graf je však strom a so stromami je oveľa viac spätý algoritmus prehľadávania do hĺbky (skrátene DFS z anglického Depth-First Search). Ukážeme si teda, ako tento algoritmus funguje, neskôr sa nám totiž ešte zíde pri lepšom riešení.

DFS si zakladá na rekurzívnom vnáraní čoraz hlbšie a hlbšie. Presnejšie, nech začneme v nejakom vrchole x. Tento vrchol si označíme ako navštívený. Postupne sa budeme pozerať na všetkých susedov x a vždy, keď nájdeme nejakého, ktorý je zatiaľ nenavštívený, rekurzívne sa vnoríme – teda označíme ten vrchol za navštívený a začneme rovnakým spôsobom prezerať jeho susedov. Ak sme takto spracovali všetkých susedov aktuálneho vrcholu, vrátime sa v rekurzii späť. Konkrétne, algoritmus sa dá jednoducho implementovať nasledovným spôsobom.

Listing programu (C++)

```
vector< vector<int> > G; // tu mám zapamätaný graf: V[x] je zoznam vrcholov, ktoré susedia s vrcholom x
bool T[n];               // v tomto poli si budem značiť, či som vrchol už navštívil, na začiatku je plné false
int depth = -1;          // hĺbka pod začiatočným vrcholom, v ktorej práve sme

void dfs(int v) {
    ++depth;             // ideme hlbšie
    T[v]=true;           // vrchol v je odteraz navštívený
    for(int i=0; i<G[v].size(); i++) {
        int w=G[v][i];   // vrchol w je susedom vrcholu v
        if(T[w]) continue; // ak už bol navštívený, nezaujíma nás
        dfs(w);          // inak rekurzívne ofarbíme w, jeho susedov, susedov jeho susedov, atď.
    }
    --depth;             // a keď sme všetko prezreli, vynoríme sa o úroveň vyššie
}
```

Vidíme, že je to naozaj elegantné. A ako nám to pomôže v našom probléme? Vidíme, že v programe mám premennú `depth`. Tá označuje, ako ďaleko sme sa dostali od začiatočného vrcholu. Vždy keď sa vnoríme hlbšie do rekurzii, premennú zväčším, znamená to, že sme sa od x vzdialili o ďalšiu hranu, keď sa z rekurzie vynárame, premennú zmenšíme, sme o hranu bližšie. To ale znamená, že ak začneme prehľadávanie z vrcholu x, keď objavíme vrchol y, tak v premennej `depth` máme jeho vzdialenosť od vrchola x. A to je presne to, čo sme potrebovali.

Časová zložitosť prehľadávania do hĺbky vo všeobecnom súvislom grafe je $O(n+m)$, kde n je počet vrcholov a m počet hrán toho grafu. To preto, že rekurziu voláme len na neoznačený vrchol, takže funkciu `dfs()` zavoláme presne n-krát. A v rámci týchto volaní sa pozriem na každú hranu dvakát – z každého konca raz.

V strome platí, že jeho počet hrán m je presne rovný $n-1$. Preto je časová zložitosť prehľadávania do hĺbky na strome $O(n)$, teda lineárna od počtu vrcholov stromu.

Takto dostávame prvé riešenie našej pôvodnej úlohy: pre každú dvojicu (x, y) spustíme DFS z vrcholu x, zistíme vzdialenosť do y a tú pripočítame k výsledku. Takéto riešenie má časovú zložitosť $O(n^3)$.

Toto riešenie sa však dá jednoducho zlepšiť. Uvedomme si, že pri prehľadávaní do hĺbky navštívime každý vrchol práve raz. To znamená, že ak začneme prehľadávať z vrcholu x, tak postupne prejdeme všetky ostatné vrcholy. A teda vieme počas jediného prehľadávania zistiť vzdialenosť od vrcholu x ku všetkým ostatným. Takto ľahko dostaneme riešenie s časovou zložitosťou $O(n^2)$. Z každého vrcholu spustíme jedno prehľadávanie a k výsledku pripočítame vzdialenosti ku všetkým ostatným vrcholom. Za takéto riešenie by sme dostali slušných 7 bodov, my (ako praví chlapi a pravé ženy!) sa s tým však samozrejme neuspokojíme :-).

Vzorové riešenie

Myšlienka vzorového riešenia bude vychádzať z myšlienky, ktorú sme použili v domácom kole pri počítaní súčtov všetkých možných úsekov postupnosti. Mimochodom, táto úloha je všeobecnejšou verziou dotyčnej úlohy domáceho kola: Predstavte si jednu rovnú cestu a na nej $n+1$ domov tak, že vzdialenosti medzi susednými domami sú jednotlivé čísla postupnosti. Potom každý úsek postupnosti zodpovedá vzdialenosti niektorých dvoch domov. Teraz teda riešime presne tú istú úlohu, len tentokrát môže mať dedina ľubovoľnú stromovú topológiu.

Pozrime sa teda na cesty bližšie, hlavne to, z čoho sa skladajú. No predsa z jednotlivých hrán. Náš strom ich má, ako už vieme, presne $n-1$. A každá hrana nejak prispieva k celkovému výsledku. Presnejšie, konkrétna hrana je vo výsledku zarátaná toľkokrát, koľkokrát cez ňu vedie nejaká cesta. Ak by sme vedeli zistiť tento počet, vedeli by sme úlohu ľahko vyriešiť. Pre každú hranu by sme zrátali, koľko ciest cez ňu prechádza, a sčítali týchto $n-1$ čísel.

Zoberme si nejakú hranu e. Ak ju z nášho stromu vyberieme, vidíme, že náš strom sa rozpadne na dve časti – komponenty. Toto v strome platí pre ľubovoľnú hranu. Od čoho závisí, či cesta medzi vrcholmi x a y, prechádza cez hranu e? Ak vrcholy x a y ležia v tom istom komponente, cez hranu e ich spoločná cesta neprechádza, lebo existuje spojenie týchto vrcholov v rámci toho komponentu. No a naopak, ak x a y ležia v rôznych komponentoch, vtedy cesta medzi nimi musela prechádzať hranou e, lebo ona jediná spája tieto dva komponenty.

Ak si označíme počet vrcholov v prvom komponente p_a a v druhom komponente p_b, tak dostávame, že hranou e prechádza práve $p_a \cdot p_b$ ciest spájajúcich nejaké dvojice vrcholov. Bolo by teda fajn, keby sme pre každú hranu vedeli rýchlo zistiť hodnoty p_a a p_b. Dokonca nám stačí len jedna z nich, lebo vieme, že vždy platí $p_a + p_b = n$.

Vyberme si teraz nejaký vrchol x a zavesme náš strom za tento vrchol. Znamená to, že všetky ostatné visia pod ním. Odborne sa tento vrchol nazýva koreň. To znamená, že z každého vrchola trčí jedna hrana hore, smerom k x. Samozrejme okrem samotného x, keďže ten je najvyššie a máme len $n-1$ hrán. Každá hrana tento strom rozdelí na dva komponenty: jeden je tvorený spodným vrcholom hrany a všetkým, čo visí pod ním, druhý je celý zvyšok stromu. Keby sme pre pre každý vrchol zistili, koľko vrcholov visí pod ním (teda aký veľký je jeho podstrom), určovalo by nám to hodnotu p_a pre hranu vedúcu z neho dohora.

Pomocou DFS to nie je problém zistiť. Počas prehľadávania si budeme pre každý vrchol počítať, koľko vrcholov je pod ním. Toto zistím tak, že sčítam veľkosti podstromov všetkých jeho susedov, ktorý ležia nižšie ako on (čo sú vrcholy, do ktorých som sa z tohto vrchola rekurzívne vnáral) a pričítam 1 za ten samotný vrchol. Potom si k výsledku prirátam $p(n-p)$ kde p je veľkosť podstromu pre daný vrchol. Na konci teda dostanem žiadaný výsledok. Toto všetko zrátam počas jedného prehľadávanie, preto zložitosť tohto algoritmu je $O(n)$. Pamätať si potrebujem samotný graf, čo nám zaberie $O(n)$ pamäte.

Listing programu (C++)

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

long long vys=0;
int n;
vector<vector<int> > G;
```

```
vector<bool> T;          // T[x] hovorí, či už bol vrchol x navštívený
vector<int> P;           // P[x] je veľkosť podstromu s koreňom vo vrchole x

void dfs(int v) {
    T[v]=true;
    for(int i=0; i<G[v].size(); i++) {
        int w=G[v][i];
        if(T[w]) continue;
        dfs(w);          // spracujeme všetky vrcholy v podstrome s koreňom w
        P[v]+=P[w];      // pridáme (práve vypočítanú) veľkosť podstromu s koreňom w k našej veľkosti
    }
    P[v]++;              // za samotný vrchol v
    vys+=(long long)P[v]*(n-P[v]);
}

int main() {
    // načítame vstup
    scanf("%d ",&n);
    G.resize(n); T.resize(n,false);
    P.resize(n,0);
    for(int i=0; i<n-1; i++) {
        int x,y;
        scanf("%d %d ",&x,&y);
        x--; y--;
        G[x].push_back(y);
        G[y].push_back(x);
    }
    // spustíme prehľadávanie do hĺbky
    dfs(0);
    printf("%lld\n",vys);
    return 0;
}
```

B-II-3 Poklad

Než sa pozrieme na náš príklad s vysávačom, zamyslime sa nad jednoduchšou hrou. Niekto si myslí číslo od 1 do n a my máme uhádnuť, ktoré to je. Keď skúsime nejaké číslo, dozvieme sa, či sme uhádli, alebo je správna hodnota väčšia, alebo menšia ako náš tip.

Ak na začiatku vieme, že správna hodnota je medzi 1 a 100, dobrá stratégia je tipnúť si 50. Buď sa dozvieme "viac" (takže zostane len 50 kandidátov na správnu hodnotu), alebo "menej" (iba 49 kandidátov), alebo, ak máme šťastie, to bude presne 50.

My sa samozrejme nechceme spoliehať len na šťastie, chceme stratégiu, čo bude dobrá aj v tom najhoršom prípade. Keby sme vyberali čísla, čo sú ďaleko od stredu, ľahko by nám mohlo zostať veľa kandidátov. Zatiaľ čo ak si vždy vyberieme číslo v strede zostávajúcich kandidátov, po každom ťahu ich zostane najviac polovica. Takže tejto stratégii vždy postačí $\lfloor \log_2 n \rfloor$ ťahov. Toto sa volá binárne vyhľadávanie (lebo hľadáme správnu hodnotu tým, že to vždy delíme binárne, čiže na dve polovice).

Podúloha A: s predlžovačkou

Späť k vysávaču. To, čo by sme potrebovali, je vedieť zisťovať, či je poklad vpravo alebo vľavo od nejakého políčka s (toho, čo je v strede úseku, kde sa ešte poklad môže nachádzať). Ale vieme dostať odpoveď len na otázku, či je nové vysávané políčko k pokladu bližšie alebo ďalej ako posledné vysávané políčko (označme ho p). Čo s tým? Spravíme to tak, že povysávame políčko $2s - p$, čiže to, ktoré je oproti p, keď sa pozeráme z políčka s. Podľa hrkania vysávača sa dozvieme, či je poklad bližšie ku p alebo ku $2s - p$. A keďže sú tieto dve políčka umiestnené symetricky okolo s, hrkanie nám takto povie na ktorej strane od s ten poklad je.

Máme $n \leq 10^9$, takže nám postačí povysávať $\lfloor \log_2 10^9 \rfloor + 1 = 30$ políčok. To je pod limitom 35. (To $+1$ je prvé povysávané políčko. Nezáleží, ktoré si vyberieme – nič sa z neho nedozvieme, a v ďalšom ťahu bez ohľadu na p vieme ťahať tak, ako potrebujeme.)

Listing programu (C++)

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  string odpoved;
```

```
  int n;
  cin >> n;

  int p = 1;
  cout << "vysavaj " << p << endl;
  cin >> odpoved;

  int odkial = 1, pokial = n;   // kandidáti na poklad
  while (odkial != pokial) {
    int s = (odkial + pokial) / 2;
    int kde = 2*s - p;
    cout << "vysavaj " << kde << endl;
    cin >> odpoved;

    if (odpoved == "rovnako") {
      odkial = s; pokial = s;
      break;
    }

    bool poklad_je_mensi;
    if (kde < p) poklad_je_mensi = (odpoved == "blizsie");
    else         poklad_je_mensi = (odpoved == "dalej");

    if (poklad_je_mensi) pokial = s - 1;
    else                 odkial = s + 1;

    p = kde;
  }
  cout << "kop " << odkial << endl;
}
```

Za zmienku ešte stojí, že ak $n \leq 10^9$, tak v práve uvedenej implementácii skutočne stačia 32-bitové celé čísla. Rozmyslite si, aké najmenšie a aké najväčšie číslo môže mať vysávané políčko.

Podúloha B: bez predlžovačky

Minulé riešenie má tú slabinu, že políčko $2s - p$ nemusí byť medzi 1 a n. Keď nemáme predĺžovačku, môže sa stať, že nebudeme vedieť jedným ťahom zistiť, na ktorú stranu od políčka s poklad leží. Ako zabezpečiť, že sa zmestíme do limitu na počet povysávaných políčok, keď ani nevieme, koľko presne ten limit vlastne je?

Optimálny program síce nepoznáme, ale niečo o ňom vieme zistiť. Napríklad to, že v najhoršom prípade povysáva aspoň $\lfloor \log_2 n \rfloor$ políčok – každý program sa totiž dá prinútiť, aby ich povysával aspoň toľko.

Keď nejaký program testujeme, väčšinou si vopred vymyslíme, kde je poklad, a potom programu pravdivo hovoríme, ako je ďaleko. Ale tentoraz budeme zákerní a nezvolíme miesto pokladu dopredu. Budeme si pamätať, ktoré políčka sú ešte kandidáti (tie, že keby tam bol poklad, neprotirečilu by to ničomu, čo sme programu zatiaľ povedali), a vždy, keď ten program niečo povysáva, zvolíme takú odpoveď, aby tých kandidátov zostalo čo najviac. Časom síce zostane už len jeden kandidát, takže program ten poklad nájde, ale bude mu to dlho trvať. A nikto sa nemôže sťažovať, že sme podvádzali, lebo keby bol poklad naozaj na tom mieste, komunikácia Georga s programom by vyzerala úplne rovnako, a tiež by to trvalo tak dlho.

Keď si vyberáme, či programu odpovieme "bližšie", "ďalej" alebo "rovnako", možnosť "rovnako" nechceme (tam je najviac jeden kandidát, takže programu by bolo hneď jasné, kde je poklad), a z tých zvyšných dvoch vyberieme tú, kde nám zostane nadpolovičná väčšina kandidátov. Každým povysávaným políčkom zmenší program počet kandidátov najviac na polovicu, takže kým sa dostane na jediného kandidáta, musí povysávať aspoň $\lfloor \log_2 n \rfloor$ políčok.

Zistili sme, že aj optimálny program niekedy potrebuje $\lfloor \log_2 n \rfloor$ povysávaní, takže náš program ich má k dobru $2\lfloor \log_2 n \rfloor$. A akonáhle vieme, že ich môžeme spraviť až toľko, je to jednoduché: ak chceme vedieť, na ktorej strane od nejakého s poklad leží, môžme vyskúšať oboch susedov s. Tým pádom na jeden "ťah" potrebujeme dve povysávania, ale keďže ťahov nám vždy stačí $\lfloor \log_2 n \rfloor$, počet povysávaní sa zmestí do $2\lfloor \log_2 n \rfloor$.

Listing programu (C++)

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  string odpoved;
  int n;
  cin >> n;
```

```
  int odkial = 1, pokial = n;   // kandidáti na poklad
  while (odkial != pokial) {
    if (pokial - odkial == 1) {
      cout << "vysavaj " << odkial << endl;
      cin >> odpoved;
      cout << "vysavaj " << pokial << endl;
      cin >> odpoved;
      if (odpoved == "blizsie") {
        odkial = pokial;
      }
      else {
        pokial = odkial;
      }
      break;
    }

    int s = (odkial + pokial) / 2;
    cout << "vysavaj " << (s - 1) << endl;
    cin >> odpoved;
    cout << "vysavaj " << (s + 1) << endl;
    cin >> odpoved;
    if (odpoved == "blizsie") {
      odkial = s + 1;
    }
    else if (odpoved == "dalej") {
      pokial = s - 1;
    }
    else {   // "rovnako"
      odkial = s; pokial = s;
    }
  }
  cout << "kop " << odkial << endl;
}
```

B-II-4 Roboti

Podúloha A: behaj tam a späť

Na vyriešenie tejto úlohy stačilo pomocou povolených inštrukcií prepísať program „choď doprava kým nenájdeš kamienok, choď doľava kým nenájdeš kamienok, opakuj“. Celé to môže vyzerať napríklad nasledovne:

```
1: vpravo
2: ak je kamienok, pokracuj 4
3: pokracuj 1
4: vlavo
5: ak je kamienok, pokracuj 1
6: pokracuj 4
```

Podúloha B: nájdi stred

Existuje viacero možných riešení, my si ukážeme také, ktoré navyše po sebe aj uprace – teda vyzbiera kamienky. Budeme robiť presne to isté ako v riešení podúlohy A, len navyše vždy, keď nájdeme kamienok, ho posunieme o políčko bližšie smerom ku druhému kamienku. Keď sa oba kamienky stretnú na tom istom políčku, našli sme stred. (To spoznáme tak, že ideme položiť kamienok na políčko, kde už jeden leží. V takom prípade kamienok nepoložíme, ale naopak, zdvihneme aj ten druhý a skončíme.)

```
 1: vpravo
 2: ak je kamienok, pokracuj 4
 3: pokracuj 1
 4: zdvihni
 5: vlavo
 6: ak je kamienok, pokracuj 16
 7: poloz
 8: vlavo
 9: ak je kamienok, pokracuj 11
10: pokracuj 8
11: zdvihni
12: vpravo
13: ak je kamienok, pokracuj 16
14: poloz
15: pokracuj 1

16: zdvihni
```

Program môžeme ešte o jednu inštrukciu skrátiť, ak si uvedomíme, že kamienky sa určite stretnú, keď budeme posúvať ľavý kamienok doprava. Pri posúvaní pravého kamienka doľava môžeme teda vynechať kontrolu.

Podúloha C: stretnutie

Zjavne je nutné položiť nejaké kamienky. Totiž obaja roboti majú ten istý program. Ak nepoložíme nikdy žiadne kamienky, budú obaja vždy vykonávať naraz tú istú inštrukciu, a teda budú neustále robiť presne to isté – inými slovami, vzdialenosť medzi nimi sa nikdy nezmení.

Musíme teda niekedy položiť nejaký kamienok. Ak potom časom nastane situácia, v ktorej jeden robot kamienok vidí a druhý nie, môžu sa ich programy „rozsynchronizovať“ a máme šancu, aby sa niekedy neskôr aj stretli.

V našom riešení každý robot položí len jeden kamienok, a to hneď na začiatku. (Existujú však aj iné riešenia. Napríklad to naše, ako neskôr uvidíte, by rovnako dobre fungovalo, aj keby každý robot po každom kroku položil kamienok.)

Po položení kamienka sa obaja roboti vyberú doľava. Časom potom určite nastane situácia, že jeden z nich (ten, ktorý začínal viac vpravo) objaví kamienok položený druhým z nich. V tomto okamihu tento robot vie, že druhý je naľavo od neho. Čo ale s touto informáciou môže spraviť?

Hlavný trik riešenia bude v tom, že na začiatku sa obaja roboti vyberú doľava *pomaly*, len raz za niekoľko inštrukcií spravia krok. No a akonáhle jeden z robotov zistí, že on je ten viac vpravo, *zrýchli* a druhého robota dobehne.

Na spomalenie robota môžeme použiť inštrukciu `nic`, ale nie je to nutné – stačí vynechať kontrolu, či je na našom políčku kamienok. Vtedy si ale treba poriadne rozmyslieť poradie inštrukcií, aby nám nemohla vzniknúť *race condition*: ľavý robot skontroluje, že je sám, pravý príde na to isté políčko, uvidí robota, zastane, a následne sa ľavý robot, ktorý pri kontrole ešte nikoho iného nevidel, pohne doľava.

Vo vzorovom programe preto radšej použijeme navyše dve inštrukcie `nic` tak, aby prvý cyklus trval 6 sekúnd a druhý 3. Ak sa teda už pravý robot hýbe rýchlejšie, pôjde presne dvakrát tak rýchlo. Všimnite si, že tým pádom vždy, keď kontroluje ľavý robot prítomnosť pravého, kontroluje aj pravý prítomnosť ľavého.

Výsledný program (rovnaký pre oboch robotov):

```
 1: poloz

 2: vlavo
 3: nic
 4: ak je kamienok, pokracuj 8
 5: nic
 6: ak je robot, pokracuj 99
 7: pokracuj 2

 8: vlavo
 9: ak je robot, pokracuj 99
10: pokracuj 8

99: koniec
```

DVADSIATY ÔSMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Tomáš Belan, Michal Forišek, Ján Hozza
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2012