A-II-1 Sizyfos a balvany

Podúloha A

Inverziou nazveme každú dvojicu (i, j) takú, že $i < j$, ale $H[i] > H[j]$.
Zjavne platí, že postupnosť je usporiadaná (od najmenšieho prvku po najväčší) práve vtedy, ak neobsahuje žiadnu inverziu. Postupnosť dĺžky n môže obsahovať nanajvýš $n(n-1)/2$ inverzií. (Toto nastáva, ak sú všetky prvky navzájom rôzne a usporiadané opačne – vtedy každá dvojica indexov tvorí inverziu.)
Každá výmena, ktorú spraví Sizyfos, zníži počet inverzií v poli H presne o jednu. V každom kole teda klesne počet inverzií v poli H. Preto kôl, v ktorých Sizyfos niečo vymení, môže byť dokopy len nanajvýš toľko, koľko bolo na začiatku v poli H inverzií.

Niekoľko úvah navyše

Sizyfov postup vždy na konci vyrobí usporiadané pole. (Ide o tzv. bublinové triedenie, po anglicky bubble sort.) Ak totiž máme pole, ktoré ešte nie je usporiadané, musí existovať aspoň jeden index i, pre ktorý neplatí $H[i-1] \leq H[i]$. To ale znamená, že v nasledujúcom kole Sizyfos ešte spraví aspoň jednu ďalšiu výmenu (keď príde k najmenšiemu takémuto i). Proces teda môže skončiť len vtedy, ak už je celé H usporiadané.
Pre počet kôl vieme dokázať aj tesnejší horný odhad: kôl nikdy nebude viac ako n. Všimnime si, že keď Sizyfos odchádza doprava z pozície i tak vždy platí, že na pozícii i je najväčší z balvanov, ktoré dovtedy počas tohto kola videl. (Toto vieme dokázať matematickou indukciou.) To ale znamená, že v prvom kole sa určite najväčší zo všetkých balvanov dostane na úplný koniec, potom v druhom kole sa druhý najväčší balvan dostane na predposledné miesto, a tak ďalej – pre každé i platí, že po i-tom kole už bude najväčších i prvkov poľa H zaručene na správnych miestach (a už sa odtiaľ nikdy nepohnú).

Podúloha B

Prvky ťažšie ako w nám vstup rozdelia na kratšie samostatné úseky. Ľahko nahliadneme, že na každom z týchto úsekov Sizyfos postupne spraví presne tie isté akcie, ktoré by spravil, keby existoval len tento úsek balvanov.
Stačí nám teda vstupnú postupnosť H rozdeliť na ťažké balvany a úseky ľahkých balvanov. Ťažké balvany necháme na mieste a každý úsek ľahkých balvanov usporiadame. Toto vieme spraviť s časovou zložitosťou $O(n \log n)$.

Podúloha C: analýza

Pre jednoduchosť predpokladajme, že už sme H rovnako ako v podúlohe B rozdelili na samostatné úseky – inými slovami, predpokladajme, že žiaden prvok v našom H nie je priťažký.
Všimnime si, že zatiaľ čo balvany, ktoré sú priťažké na svoju pozíciu, môžu v jednom kole precestovať veľa pozícií smerom doprava, žiaden balvan sa v žiadnom kole nevie posunúť viac ako o jednu pozíciu doľava. Keď totiž spravíme výmenu, ktorá nejaký balvan X posunie doľava, tak sa následne pohneme ďalej doprava a teda sa už v tomto kole na balvan X nikdy nepozrieme.

Pozrime sa na začiatočnú postupnosť balvanov a na každý z nich si napíšme dve čísla: modrou kriedou číslo pozície, na ktorej začína; červenou kriedou číslo pozície, na ktorej bude, keď Sizyfov postup skončí.
Teraz zoberieme zelenú farbu. Pre každý balvan si spočítame, o koľko napravo je od svojej správnej pozície: od modrého čísla odpočítame červené a tento výsledok si na balvan zapíšeme zelenou farbou. Tento výsledok môže byť aj záporný ak je balvan naľavo od svojej správnej pozície.
Tvrdíme, že počet kôl, v ktorých sa niečo zmení, je presne rovný najväčšiemu zelenému číslu.

Dôkaz: Je zjavné, že kôl musí byť *aspoň* toľko. Napr. ak najväčším zeleným číslom je 3, znamená to, že máme balvan 3 kroky napravo od správnej pozície. No a v každom kole sa tento balvan posunie nanajvýš o jednu pozíciu doľava.
Prečo bude kôl *presne* toľko? Tvrdíme, že úplne všetky balvany, na ktoré sme napísali *kladné* zelené číslo, sa v prvom kole pohnú *presne* o jednu pozíciu doľava. Pozrime sa totiž na ľubovoľný balvan X, ktorý sa momentálne nachádza napravo od pozície, na ktorej má skončiť. Naľavo od X je teraz viac balvanov ako tam má byť na

konci, je tam teda aspoň jeden balvan, ktorý je od X ťažší. Vieme, že keď počas tohto kola Sizyfos príde k balvanu X, bude naľavo od neho najväčší z balvanov, ktoré Sizyfos dovtedy videl – čiže tam bude balvan ťažší ako X. To ale znamená, že s ním následne balvan X zaručene vymeníme, čím ho posunieme o jednu pozíciu doľava.
Ak by sme teda po skončení kola všetky čísla napísané na kameňoch zmazali a prepočítali, budeme vidieť, že sa nám všetky kladné zelené čísla o jedna zmenšili. Ak sa teda maximum zelených čísel v každom kole o 1 zmenší, tak po práve takom počte kôl budú balvany usporiadané.
Musíme si však ešte rozmyslieť, že sa nám nemôže stať, aby nám balvan presúvaný *doprava* zhoršil situáciu. Pozrime sa na ľubovoľnú výmenu, kde ťažší balvan Y vymeníme s ľahším balvanom X. Po výmene je Y presne o 1 políčko napravo od X. Zároveň vieme, že cieľová pozícia pre Y je aspoň o 1 políčko napravo od cieľovej pozície pre X (lebo Y je ťažší ako X). Aktuálne zelené číslo pre Y je preto ešte stále menšie alebo rovné ako aktuálne zelené číslo pre X, a teda nemá ako zvýšiť hodnotu ich maxima.

Podúloha C: algoritmus

Pre každý súvislý úsek balvanov ľahších ako w vytvoríme nové pole, ktorého prvkami budú usporiadané dvojice $(H[i], i)$. Toto pole klasickým algoritmom v čase $O(n \log n)$ usporiadame. Keď dvojica $(H[i], i)$ skončí v usporiadanom poli na indexe j, znamená to, že tento balvan má na sebe na začiatku modré číslo i a červené číslo j. Takto teda vieme efektívne zistiť, ktorý balvan je najviac napravo od správnej pozície, a to nám povie počet kôl, v ktorých budeme robiť výmeny.

Program riešiaci podúlohy B aj C:

Listing programu (C++)

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

int main() {
    int n, w;
    cin >> n >> w;
    vector<int> H(n);
    for (int &h : H) cin >> h;
    H.push_back(w+1); // zarazka
    vector<int> vystup;
    int zac = 0;
    int dolava = 0; // o kolko najviac dolava musel nejaky prvok ist
    while (zac < n) {
        // najdeme usek balvanov, ktorymi vieme hybat
        int kon=zac;
        while (H[kon] <= w) ++kon;
        // usporiadame ich, pricom si zapamatame kde zacinali
        vector< pair<int,int> > balvany;
        for (int i=zac; i<kon; ++i) balvany.push_back( { H[i], i } );
        sort( balvany.begin(), balvany.end() );
        // najdeme ten co siel najviac dolava
        for (int i=zac; i<kon; ++i) dolava = max( dolava, balvany[i-zac].second - i );
        // vyplnime vystup
        for (int i=zac; i<kon; ++i) vystup.push_back( balvany[i-zac].first );
        vystup.push_back( H[kon] );
        zac = kon+1;
    }
    for (int i=0; i<n; ++i) cout << vystup[i] << (i+1 == n ? "\n" : "␣");
    cout << "pocet␣kol:␣" << (dolava+1) << endl;
}
```

A-II-2 Obdĺžnik

Odporúčané čítanie pred týmto vzorovým riešením: https://www.ksp.sk/kucharka/skalarny_a_vektorovy_sucin/ a prípadne aj https://www.ksp.sk/kucharka/konvexny_obal/.

Už v zadaní sme videli, ako vyriešiť prípad, keď sú všetky body na jednej priamke. Môžeme ho ošetriť ako špeciálny prípad. Vo zvyšku riešenia budeme potom predpokladať, že tento špeciálny prípad nenastáva.

Hľadanie riešenia si ešte trochu zjednodušíme nasledovnou úvahou: Majme ľubovoľný obdĺžnik, ktorý má na obvode všetky zadané body. Pre každú jeho stranu zvlášť sa pozrime, či obsahuje niektorý zo zadaných bodov. Ak nie, môžeme náš obdĺžnik zmenšiť tak, že túto stranu budeme posúvať bližšie ku protiľahlej až kým na nejaký zadaný bod nenarazí. Na konci tohto procesu dostaneme obdĺžnik, ktorý nielenže obsahuje všetky zadané body, ale navyše platí, že na každej jeho strane leží aspoň jeden zo zadaných bodov.
Z vyššie uvedenej úvahy vieme, že ak existuje nejaké riešenie, tak nutne existuje aj riešenie s touto dodatočnou vlastnosťou. Stačí nám teda hľadať takýto obdĺžnik.

Pomalšie efektívne riešenie

Riešenie úlohy si môžeme výrazne uľahčiť tým, že nájdeme konvexný obal všetkých zadaných bodov.
Ak riešenie našej úlohy existuje, čo vieme povedať o tomto konvexnom obale?
Na každej strane nášho obdĺžnika máme jeden alebo viac bodov. Tieto nám na každej strane obdĺžnika určia bod alebo úsečku (od prvého po posledný bod na danej strane), ktorá zjavne leží na obvode konvexného obalu. A keďže už žiadne iné body nemáme, náš konvexný obal je zároveň konvexným obalom týchto štyroch bodov či úsečiek. Ešte inými slovami, keď pre každú stranu obdĺžnika zoberieme len konce jej úsečky (resp. len jeden bod, ak je jediný), dostaneme práve všetky vrcholy konvexného obalu. Z toho teda vyplýva, že konvexným obalom bude nutne nanajvýš osemuholník.
Navyše, keďže bodov máme aspoň desať, na niektorej strane obdĺžnika ich bude ležať viacero, a teda na nej bude ležať niektorá strana konvexného obalu.
Pôvodnú úlohu teda môžeme vyriešiť nasledovne: Nájdeme konvexný obal všetkých n zadaných bodov. (Toto vieme spraviť v čase $O(n \log n)$.) Ak má viac ako 8 vrcholov, hľadaný obdĺžnik neexistuje. Ak niektorý zadaný bod leží vo vnútri konvexného obalu, obdĺžnik tiež neexistuje. Vo zvyšných prípadoch stačí vyskúšať nanajvýš 8 prípadov: vyskúšame všetky možnosti pre to, ktorá strana konvexného obalu leží na strane obdĺžnika. Každý z týchto prípadov vieme vyskúšať v konštantnom čase. Detaily kontroly budú podobné vzorovému riešeniu, ktoré si popíšeme nižšie.

Vzorové riešenie

Úlohu vieme vyriešiť aj v lineárnom čase a bez explicitnej konštrukcie všeobecného konvexného obalu.
Začneme nasledovnou úvahou: Spomedzi zadaných n bodov zoberme ľubovoľných päť. Ak všetky ležia na obvode obdĺžnika, z Dirichletovho princípu musia existovať (aspoň) dva z nich, ktoré ležia na tej istej strane. Prezrieme teda všetky dvojice spomedzi vybraných piatich bodov. Pre každú z nich vyriešime jednoduchšiu úlohu: budeme hľadať obdĺžnik, ktorý má navyše tú vlastnosť, že jedna jeho strana leží na priamke určenej týmito dvoma bodmi. Ak ľubovoľná z týchto úloh bude mať riešenie, máme aj riešenie pôvodnej úlohy. A naopak, ak žiadna z týchto úloh nebude mať riešenie, budeme mať istotu, že hľadaný obdĺžnik neexistuje.
Možností, ktoré takto musíme vyskúšať, je len desať – jedna pre každú dvojicu bodov z našej vybranej pätice. Ak každú možnosť odskúšame v lineárnom čase, dostaneme aj celkovo lineárne riešenie.

Riešenie našej jednoduchšej úlohy začneme tým, že zoberieme všetkých n bodov a zistíme, ktoré z nich ležia na našej priamke. Toto vieme spraviť vektorovým súčinom: C leží na priamke AB ak má vektorový súčin $\overrightarrow{AB}$ a $\overrightarrow{AC}$ veľkosť nula.
Z bodov, ktoré ležia na našej priamke, následne vyberieme „prvý a posledný“ – teda koncové body najkrajšej úsečky na ktorej všetky ležia. Toto vieme spraviť skalárnym súčinom: hľadáme najmenšiu a najväčšiu hodnotu skalárneho súčinu $\overrightarrow{AB}$ a $\overrightarrow{AC}$.
Teraz teda máme úsečku, ktorá musí celá ležať na jednej strane hľadaného obdĺžnika.
Ostatné body (tie, ktoré neležia na tejto priamke) musia všetky ležať v tej istej polrovine od nej. Toto opäť vieme overiť vektorovým súčinom $\overrightarrow{AB}$ a $\overrightarrow{AC}$: všetky nenulové súčiny musia mať rovnaké znamienko. Ak to nie je pravda, riešenie neexistuje.
Ostatné body si teraz ďalej rozdelíme na dve kôpky: tie, ktoré sú najďalej od našej priamky a ostatné. Toto vieme spraviť skalárnym súčinom, násobiť tentokrát budeme $\overrightarrow{AC}$ a tzv. normálový vektor našej priamky – teda vektor kolmý na AB. Vektor kolmý na (x, y) je napr. $(-y, x)$.
Body, ktoré sú najďalej, musia všetky ležať na protiľahlej strane hľadaného obdĺžnika. Zvyšné body (ak ešte

nejaké ostali) nazvime bočné. Všetky bočné body musia ležať na druhej dvojici rovnobežných strán. Ostáva nám už len overiť, či toto vieme dosiahnuť – teda či vieme zvyšné dve strany obdĺžnika zvoliť tak, aby pokryli všetky bočné body.

Úplne všetky body si kolmo premietneme na pôvodnú priamku AB. (Toto vieme opäť spraviť skalárnym súčinom.) Body ležiace priamo na nej a kolmé priemety bodov z druhej rovnobežnej priamky nám určujú, odkiaľ pokiaľ musia minimálne siahať strany obdĺžnika ležiace na týchto priamkach. Žiaden z bočných bodov nemôže mať svoj kolmý priemet vo vnútri tohto intervalu. A navyše, aj „naľavo“ aj „napravo“ od tohto intervalu musí platiť, že ak tam nejaké bočné body máme, tak musia mať všetky ten istý kolmý priemet – dotyčná strana musí naraz prechádzať všetkými z nich.

Stačí nám teda spomedzi všetkých priemetov vybrať „prvý a posledný“ a potom overiť, či sa každý z bočných bodov premietne na jeden z týchto dvoch extrémov. Ak nie, riešenie neexistuje, ak áno, existuje a hľadaný obdĺžnik už ľahko zostrojíme: dva jeho vrcholy budú ležať práve v extrémnych priemetoch a tretí nájdeme tak, že sa z jedného z nich posunieme po normálovom vektore o vzdialenosť, v ktorej má byť protiľahlá strana.

Listing programu (C++)

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

// BEGIN geometricke funkcie

typedef complex<double> point;
typedef vector<point> point_seq;

const double EPSILON = 1e-7;
bool is_negative(double x) { return x < -EPSILON; }
bool is_zero(double x) { return abs(x) <= EPSILON; }
bool is_positive(double x) { return x > EPSILON; }

bool are_equal(const point &A, const point &B) { return is_zero(real(B)-real(A)) && is_zero(imag(B)-imag(A)); }
double dot_product  (const point &A, const point &B) { return real(A) * real(B) + imag(A) * imag(B); }
double cross_product(const point &A, const point &B) { return real(A) * imag(B) - real(B) * imag(A); }
double size(const point &A) { return sqrt(real(A) * real(A) + imag(A) * imag(A)); }
point normal(const point &smer) { return point( -imag(smer), real(smer) ) / size(smer); }

// END geometricke funkcie

int n;
point_seq vstup;

point_seq find_first_and_last(const point_seq &X) {
    point unit = (X[1] - X[0]) / size(X[1] - X[0]);
    vector<double> dot_products;
    for (auto x:X) dot_products.push_back( dot_product( unit, x-X[0] ) );
    double mn = *min_element(dot_products.begin(), dot_products.end());
    double mx = *max_element(dot_products.begin(), dot_products.end());
    return { X[0] + unit*mn, X[0] + unit*mx };
}

point_seq test_line(const point &A, const point &B) {
    // roztriedime vsetky body podla toho, kde lezia vzhladom na priamku
    point_seq nalavo, napravo;
    for (int i=0; i<n; ++i) {
        double vp = cross_product(B-A, vstup[i]-A);
        if (is_positive(vp)) nalavo.push_back(vstup[i]);
        if (is_negative(vp)) napravo.push_back(vstup[i]);
    }
    if (!nalavo.empty() && !napravo.empty()) return {};

    // osetrime pripad kedy su vsetky na priamke
    if (nalavo.empty() && napravo.empty()) {
        auto odpoved = find_first_and_last(vstup);
        odpoved.push_back( odpoved[0] + normal( odpoved[1]-odpoved[0] ) );
        return odpoved;
    }

    // najdeme tie co nie su najdalej od priamky -- musia byt na bokoch
    double maxvz = 0;
    point_seq mimo = nalavo.empty() ? napravo : nalavo;
    for (auto x : mimo) maxvz = max( maxvz, abs( dot_product(x-A, normal(B-A)) ) );
    point_seq boky;
    for (auto x : mimo) {
        if (!are_equal( maxvz, abs( dot_product(x-A, normal(B-A)) ) )) boky.push_back(x);
    }

    // premietneme vsetko na nasu priamku
    point_seq priemety;
    point unit = (B-A) / size(B-A);
    for (auto x : vstup) priemety.push_back( A + unit*dot_product(unit, x-A) );
```

```
    // skontrolujeme, ci su priemety bocnych bodov na koncoch
    auto krajne = find_first_and_last(priemety);
    for (auto x : boky) {
        auto pr = A + unit*dot_product(unit, x-A);
        if (!are_equal(pr, krajne[0]) && !are_equal(pr, krajne[1])) return {};
    }

    // vyrobime tri rohy obdlznika
    auto norm = normal(krajne[1]-krajne[0]);
    auto cand1 = krajne[0] + maxvz*norm, cand2 = krajne[0] - maxvz*norm;
    if (is_negative(cross_product( B-A, cand1-A ))) swap(cand1, cand2);
    auto treti = nalavo.empty() ? cand2 : cand1;
    return { krajne[0], krajne[1], treti };
}

int main() {
    cin >> n;
    vstup.resize(n);
    for (int i=0; i<n; ++i) { double x, y; cin >> x >> y; vstup[i] = point(x,y); }
    for (int a=0; a<5; ++a) for (int b=0; b<a; ++b) {
        auto odpoved = test_line( vstup[a], vstup[b] );
        if (odpoved.empty()) continue;
        for (auto b : odpoved) cout << b << endl;
        return 0;
    }
    cout << "NIE" << endl;
}
```

Alternatívne vzorové riešenie

Myšlienky oboch vyššie uvedených riešení vieme skombinovať do lineárneho riešenia nasledovne: Začneme hľadať všeobecný konvexný obal, ale namiesto niektorej z metód s časovou zložitosťou $O(n \log n)$ použijeme algoritmus „balenia darčeka“ (anglicky gift wrapping, resp. Jarvisov algoritmus). Tento algoritmus konvexný obal zostrojuje postupne a má časovú zložitosť $O(hn)$, kde h je počet bodov výsledného konvexného obalu. Tento algoritmus je vo všeobecnosti pomalší, keďže v najhoršom prípade je jeho časová zložitosť až kvadratická od počtu bodov na vstupe. V našej úlohe však vieme, že akonáhle dostaneme deviaty bod na konvexnom obale, môžeme beh algoritmu ukončiť a dať negatívnu odpoveď. Takto v lineárnom čase buď dostaneme odpoveď NIE alebo hotový konvexný obal našich bodov.

A-II-3 Továreň

Pre každé i platí, že na objednávke $i+1$ začneme v optimálnom riešení robiť 0 mesiacov, 1 mesiac alebo 2 mesiace po začiatku objednávky i. To sú teda vždy tri možnosti. Vyskúšaním a skontrolovaním všetkých možností pre každé i dostávame korektné ale pomalé riešenie s exponenciálnou časovou zložitosťou.

Efektívne riešenie pre malý počet zamestnancov

Stručne si načrtneme jedno možné riešenie, ktoré je efektívne, ak je počet zamestancov p malý.
Vtedy vieme úlohu riešiť napr. nasledovnou úvahou: Na začiatku vyskúšame všetky možnosti pre to, koľko objednávok spravíme prvý mesiac. Zakaždým sa dostaneme do nejakej situácie, v ktorej sme sa presne rozhodli, čo sa deje počas prvého mesiaca a potrebujeme sa ďalej rozhodovať, čo robiť v druhom a ďalších. Každú takúto situáciu vieme popísať dvoma parametrami: počtom x objednávok, ktoré sme ešte nezačali robiť, a počtom z zamestnancov, ktorých máme ešte dostupných počas nasledujúceho mesiaca (keď ešte ostatní dorábajú skôr začaté objednávky).
Pre každé x a z si môžeme položiť otázku, na koľko najmenej mesiacov vieme dokončiť všetky zostávajúce objednávky. Každú takúto otázku vieme zodpovedať tak, že vyskúšame všetky možnosti, koľko nových objednávok začneme v nasledujúcom mesiaci. Pre každú takúto možnosť dostaneme o mesiac neskôr situáciu rovnakého typu, len s inými hodnotami x a z. Odpovedanie na takéto otázky vieme teda implementovať ako rekurzívnu funkciu. Keď pridáme memoizáciu (t.j. pre každé parametre funkciu vypočítame len raz a následne si zapamätáme výsledok, ktorý nám vyšiel), dostaneme časovú zložitosť $O(n^2 p)$.

Nižšie si najskôr ukážeme riešenie za 8 bodov a potom si popíšeme, ako ho zlepšiť na 10-bodové.

Riešenie s kubickou časovou zložitosťou

Začneme tým, že si predpočítame prefixové súčty pre polia A a B. Vďaka tomu budeme vedieť pre ľubovoľný súvislý úsek objednávok v konštatnom čase povedať, koľko by sme v ktorý mesiac potrebovali zamestnancov, keby sme všetky tieto objednávky robili naraz.
Označme $D[y]$ minimálny počet mesiacov, za ktoré vieme splniť prvých y objednávok. Ďalej označme $M[x][y]$ minimálny počet mesiacov, za ktoré vieme splniť prvých y objednávok, pričom prvých x už bolo hotových pred koncom posledného mesiaca. Ak by sme vedeli určiť hodnoty M, mali by sme riešenie zadanej úlohy. Totiž $D[y]$ je jednoducho minimom spomedzi všetkých $M[x][y]$ a následne $D[n]$ je hodnota, ktorú máme vypočítať ako odpoveď.
Ukážeme si, ako hodnoty M a D počítať postupne.
Na začiatku máme $M[0][0] = 0$ a $D[0] = 0$: nula objednávok vieme splniť okamžite.
Keď chceme vypočítať konkrétnu hodnotu $M[x][y]$, vieme, že počas posledného mesiaca sme museli robiť práve objednávky s číslami $x..(y-1)$. Ak nemáme dosť zamestnancov na to, aby sme tieto objednávky robili naraz, bude $M[x][y] = \infty$: takúto situáciu vôbec nevieme dosiahnuť.
V ostatných prípadoch $M[x][y]$ určíme rozborom dvoch prípadov: pozrieme sa na to, čo sme robili predposledný mesiac.
Prvou možnosťou je, že sme pracovali len na tých istých $y - x$ objednávkach ako posledný mesiac. Optimálne riešenie takéhoto typu vyzerá teda tak, že za najmenší počet mesiacov spravíme prvých x objednávok a k tomu pripočítame dva mesiace za nasledujúcich $y-x$ objednávok. Dokopy teda budeme potrebovať $D[x]+2$ mesiacov.
Druhou možnosťou je, že pred dvoma mesiacmi bolo hotových len z objednávok, pre nejaké neznáme $z < x$, a potom počas predposledného mesiaca sme naraz dokončovali objednávky $z..(x-1)$ a začínali objednávky $x..(y-1)$. Keď poznáme konkrétne z, vieme povedať, že nám to celé bude trvať $M[z][x] + 1$ mesiacov.
Hodnotu $M[x][y]$ zistíme tak, že si spomedzi týchto možností vyberieme tú najkratšie trvajúcu. Zoberieme teda minimum spomedzi hodnoty $D[x] + 2$ a všetkých hodnôt $M[z][x] + 1$ takých, že vieme počas predposledného mesiaca naraz robiť na všetkých potrebných objednávkach.
Hodnoty $M[x][y]$ budeme počítať v cykle cez všetky x a vnútri toho cyklu v cykle cez všetky $y > x$. Následne vždy, keď dopočítame všetky hodnoty $M[?][y]$, určíme z nich hodnotu $D[y]$.
Dokopy potrebujeme vypočítať $O(n^2)$ rôznych hodnôt v poli M. Výpočet každej z nich nám bude trvať nanajvýš lineárne dlho, lebo potrebujeme vyskúšať $O(n)$ rôznych hodnôt z. Celková časová zložitosť tohto riešenia bude teda $O(n^3)$.

Riešenie s kvadratickou časovou zložitosťou

Všimnime si, že všetky výpočty hodnôt $M[x][y]$ pre rôzne y sú veľmi podobné: vždy začneme s tou istou hodnotou $D[x] + 2$ a potom prezeráme hodnoty $M[z][x]$ pre prípustné z. Jediné, čo sa v závislosti od y mení, je, ktoré hodnoty sú prípustné. Presnejšie, čím menšie y si zvolíme, tým menej zamestnancov potrebujeme na poslednú sadu objednávok, a tým viac ich môžeme použiť na predposlednú – teda tým väčšie z ešte môže byť prípustné.
Kubické riešenie teraz zlepšíme na kvadratické nasledovne: Podobne ako v predchádzajúcom riešení budeme pre pevne zvolené x postupne počítať všetky hodnoty $M[x][y]$. Tentokrát to však spravíme v opačnom smere: začneme s $y = n$ a postupne budeme y znižovať až po $y = x + 1$. Keď postupne prechádzame cez všetky y v klesajúcom poradí, tak v každom kroku množina prípustných z buď ostane rovnaká, alebo sa zväčší. Namiesto toho, aby sme pre každé y znova prechádzali cez všetky z, nám stačí zobrať predchádzajúcu hodnotu a ak nám pribudli nejaké nové prípustné hodnoty z, prejsť tie a prezrieť, či nám nedajú nové lepšie riešenie.
Pre každé konkrétne x takto vypočítame všetky hodnoty $M[x][y]$ v celkovom čase $O(n)$: každé prípustné y aj každé prípustné z spracujeme práve raz. Dokopy teda dostávame časovú zložitosť $O(n^2)$.

Listing programu (C++)

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

const int NEKONECNO = 987654321;
```

```
vector<int> prefix_sums(const vector<int> &X) {
    vector<int> PX(1, 0);
    for (int x : X) {
        int next=PX.back() + x;
        PX.push_back(next);
    }
    return PX;
}

int main() {
    int p, n;
    cin >> p >> n;
    vector<int> A(n), B(n);
    for (int i=0; i<n; ++i) cin >> A[i] >> B[i];
    vector<int> PA = prefix_sums(A), PB = prefix_sums(B);

    vector<int> D(n+1, NEKONECNO);
    D[0] = 0;
    vector<vector<int> > M(n+1, vector<int>(n+1, NEKONECNO));

    for (int x=0; x<n; ++x) {
        int best = D[x] + 2, z = x;
        for (int y=n; y>x; --y) {
            // vypocet M[x][y]
            // skontrolujeme, ci vieme naraz robit joby x..(y-1)
            int treba_minuly = PA[y] - PA[x], treba_tento = PB[y] - PB[x];
            if (treba_minuly > p || treba_tento > p) continue;
            // ak ano, zistime, ci vieme predchadzajuci mesiac mensie z ako doteraz
            while (z > 0 && treba_minuly + PB[x] - PB[z-1] <= p) {
                --z;
                best = min( best, M[z][x]+1 );
            }
            M[x][y] = best;
        }
        // uz vieme D[x+1]
        for (int i=0; i<=x; ++i) D[x+1] = min( D[x+1], M[i][x+1] );
    }
    cout << D[n] << endl;
}
```

A-II-4 Nechaj to na solver II

Podúloha A

Rovnako ako v domácom kole stačí mať jednu binárnu premennú pre každý projekt. Hodnota tejto premennej je 0 ak projekt nepodporíme, resp. 1 ak áno. Slabú závislosť projektu x na projektoch y_1, y_2, y_3 namodelujeme jednoducho ako nerovnosť $x \leq y_1 + y_2 + y_3$. Zvyšok riešenia ostáva rovnaký ako v domácom kole.

Listing programu (Python)

```
import pulp

def hodnota(vyraz): return int(round(pulp.value(vyraz)))

e, n = [ int(_) for _ in input().split() ]
S = [ int(_) for _ in input().split() ]
V = [ int(_) for _ in input().split() ]
z = int(input())
D = [ [ int(_)-1 for _ in input().split() ] for __ in range(z) ] # -1 lebo interne cislujeme od 0

problem = pulp.LpProblem('Investicie', pulp.LpMaximize)
project = [ pulp.LpVariable(f'project{i}', cat='Binary') for i in range(n) ]

problem += e + sum( project[i] * (V[i] - S[i]) for i in range(n) )
problem += sum( project[i] * S[i] for i in range(n) ) <= e
for soft_dependency in D:
    problem += project[soft_dependency[0]] <= sum( project[d] for d in soft_dependency[1:] )

status = problem.solve()
assert pulp.LpStatus[status] == 'Optimal'
print( hodnota( problem.objective ) )
for x in project: print( x.name, '=', hodnota(x) )
```

Podúloha B

Pre každý kopec i budeme mať tri binárne premenné $z_{i,1}$, $z_{i,2}$ a $z_{i,3}$. Ich hodnoty nám budú hovoriť, či na tomto kopci postaviť prvú, druhú a tretiu zjazdovku.
V praxi sa navyše oplatí k týmto premenným rovno pridať aj nerovnosti $z_{i,1} \geq z_{i,2} \geq z_{i,3}$. Tieto nerovnosti vieme čítať tak, že druhú zjazdovku môžeme otvoriť len ak sme otvorili aj prvú, a tak ďalej. (Solver potom nemusí zbytočne prezerať logicky ekvivalentné vetvy, ako napr. možnosť otvoriť len tretiu zjazdovku namiesto len prvej zjazdovky.)
Ďalej budeme mať pre každú zjazdovku aj celočíselnú premennú $d_{i,j}$ udávajúcu jej presnú dĺžku: nula ak ju nepostavíme, resp. číslo od ℓ_i po u_i ak áno.
Celková dĺžka zjazdoviek bude jednoducho súčtom všetkých $d_{i,j}$. Tento súčet sa má presne rovnať danému d.
Celková cena postavenia zjazdoviek bude súčtom súčinov $c_i \cdot d_{i,j}$ cez všetky i a j. Túto cenu chceme minimalizovať, to teda bude cieľom nášho ILP.
Chýba nám ešte najdôležitejšia časť nášho programu: podmienky, ktoré zabezpečia, že hodnoty $d_{i,j}$ budú mať vyššie uvedený rozsah. Kľúčovou časťou riešenia úlohy bolo prísť na to, ako vieme toto dosiahnuť len pomocou lineárnych nerovností.
Všimnime si hodnoty $\ell_i \cdot z_{i,j}$ a $u_i \cdot z_{i,j}$. V oboch prípadoch ide o výraz tvaru konštanta krát premenná, čo je dovolený tvar. Ak do premennej $z_{i,j}$ priradíme nulu (teda ak sa rozhodneme túto konkrétnu zjazdovku nepostaviť), budú mať oba vyššie uvedené výrazy hodnotu 0. Naopak, ak bude $z_{i,j} = 1$, budú mať tieto dva výrazy hodnoty ℓ_i a u_i. Do nášho ILP preto pridáme nasledovné nerovnosti: $\ell_i \cdot z_{i,j} \leq d_{i,j}$ a $d_{i,j} \leq u_i \cdot z_{i,j}$. Tie v oboch prípadoch určia správny rozsah pre dĺžku príslušnej zjazdovky.

Listing programu (Python)

```
import pulp

def hodnota(vyraz): return int(round(pulp.value(vyraz)))

k = int(input())
L, U, C = [], [], []
for i in range(k):
    l, u, c = [ int(_) for _ in input().split() ]
    L.append(l)
    U.append(u)
    C.append(c)
d = int(input())

problem = pulp.LpProblem('Zjazdovky', pulp.LpMinimize)
postav = [ [ pulp.LpVariable(f'postav_{i+1}_{j+1}', cat='Binary') for j in range(3) ] for i in range(k) ]
dlzka = [ [ pulp.LpVariable(f'dlzka_{i+1}_{j+1}', cat='Integer') for j in range(3) ] for i in range(k) ]

# ciel ktory minimalizujeme: celkova cena stavby
problem += sum( C[i] * dlzka[i][j] for i in range(k) for j in range(3) )

# optimalizacia: obmedzenia pre stavanie
for i in range(k):
    problem += postav[i][1] <= postav[i][0]
    problem += postav[i][2] <= postav[i][1]

# podmienky pre spravny rozsah dlzok
for i in range(k):
    for j in range(3):
        problem += dlzka[i][j] >= L[i] * postav[i][j]
        problem += dlzka[i][j] <= U[i] * postav[i][j]

# podmienka pre celkovu dlzku
problem += d == sum( dlzka[i][j] for i in range(k) for j in range(3) )

status = problem.solve()
assert pulp.LpStatus[status] == 'Optimal', 'Uloha nema riesenie.'
print( hodnota( problem.objective ) )
for row in dlzka:
    for x in row:
        if hodnota(x):
            print( x.name, '=', hodnota(x) )
```

ŠTYRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek, Sebastian Hajdu
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2026