

P-II-1 Sisyfos a balvany**Podúloha A**

Inverze je dvojice (i, j) taková, že $i < j$, ale $H[i] > H[j]$. Je zřejmé, že posloupnost je uspořádaná (od nejmenšího prvku k největšímu), právě když neobsahuje žádné inverze. Posloupnost délky n může obsahovat nejvýše $n(n-1)/2$ inverzí (k tomu dochází, pokud jsou všechny prvky různé a seřazené v opačném pořadí; pak každá dvojice indexů tvoří inverzi).

Každá výměna provedená Sisyfem sníží počet inverzí v poli H přesně o jedna. V každém kole se tedy počet inverzí v poli H snižuje. Počet kol, ve kterých Sisyfos provede výměnu, proto může být maximálně roven počtu inverzí v poli H na začátku.

Několik dalších úvah

Sisyfův postup vždy na konci vytvoří setříděné pole (jeho algoritmus postup se nazývá *bublínkové třídění*): Pokud máme pole, které ještě není seřazené, musí existovat alespoň jeden index i takový, že $H[i-1] > H[i]$. To však znamená, že v následujícím kole Sisyfos provede alespoň jednu další výměnu (nejpozději když dosáhne tohoto indexu i). Proces proto může skončit pouze tehdy, když je celé pole H setříděné.

Můžeme dokázat, že kol dokonce nikdy nebude více než n : V prvním kole dopraví na konec pole balvan s nejvyšší hmotností, který se nacházel nejvíc vpravo. Ten na této pozici zůstane a už nebude řazení ovlivňovat. Nyní stačí uvažovat pole bez posledního prvku a argument zopakovat. A pokud to uděláme n -krát, tak každý balvan bude na pozici odpovídající jeho váze.

Podúloha B

Prvky těžší než w nelze přemístit, rozdělují tak vstup na kratší samostatné úseky, mezi kterými se balvany nemohou přesouvat. Je snadné si představit, že na každém z těchto úseků bude Sisyfos postupně provádět přesně stejné akce, jaké by prováděl, kdyby existoval pouze tento úsek balvanů.

Stačí tedy rozdělit vstupní posloupnost H na těžké balvany a úseky lehkých balvanů. Těžké balvany necháme na místě a každý úsek lehkých balvanů zvlášť setřídíme. To můžeme provést s časovou složitostí $\mathcal{O}(n \log n)$.

Podúloha C

Obdobně jako v podúloze B jsou úseky oddělené balvany váhy větší než w nezávislé, tuto podúlohu tedy stačí vyřešit pro každý úsek zvlášť a vrátit maximum z výsledků. Zaměřme se nyní na řešení pro jeden úsek; můžeme tedy předpokládat, že žádný prvek v poli H není příliš těžký.

Pro každý balvan si také můžeme předpočítat pozici, na které skončí: Jednoduše pole setřídíme (bez toho, že bychom prohazovali prvky se stejnou hodnotou)

a podíváme se, kde se balvany na konci nachází. To zvládneme v čase $\mathcal{O}(n \log n)$, který nám zabere třídění.

Uvažujme balvan b , který se na začátku aktuálního kola nachází na pozici p a jehož koncová pozice je k . Tvrdíme, že

- (1) Pokud se balvan b nachází ostře napravo od své koncové pozice, tedy $p > k$, pak se v tomto kole posune přesně o jednu pozici doleva.
- (2) Pokud se balvan b nachází na své koncové pozici nebo nalevo od ní, tedy $p \leq k$, pak na konci kola stále bude na pozici menší nebo rovné k .

Proč to platí? Podívejme se na situaci poté, co Sisyfos dorazil v tomto kole na pozici $p - 1$ hned nalevo od balvanu b . Na tuto pozici dopřesouval balvan b' – jeden z nejtěžších balvanů, které se od balvanu b nacházely nalevo. Pokud je b' těžší než b , prohodí ho s ním a k balvanu b se již nevrátí, posune ho tedy o právě jednu pozici doleva; proto platí (1) i (2). Jinak jsou všechny balvany nalevo od b nanejvýš tak těžké jako b a všechny těžší jsou tedy napravo od něj. Z toho vidíme, že balvan b nemůže být ostře napravo od své cílové pozice, a proto platí $p \leq k$. Od teď Sisyfos balvan b bude přesouvat pouze doprava. Pokud ho takto dopřesouvá na pozici k , nalevo od něj se v tomto okamžiku nachází $k - 1$ balvanů, které jsou nanejvýš tak těžké, jako balvan b . Napravo od něj se tedy nachází právě ty balvany, jejichž koncová pozice je alespoň $k + 1$, a ty jsou alespoň tak těžké jako balvan b . V následujícím kroku tedy balvan b s balvanem na pozici $k + 1$ neprohodí a balvan b skončí na pozici k ; proto platí (2).

Označme jako m maximum z rozdílů počátečních a koncových pozic balvanů (tento rozdíl je kladný, pokud balvan začíná napravo od své koncové pozice, a záporný, pokud začíná nalevo od ní). Nachází-li se tedy balvan na začátku napravo od své cílové pozice k , nejprve se v každém kole přesunuje o jednu pozici doleva, dokud nedorazí na pozici k (to nastane nejpozději po m kolech) a poté se už nedostane zpět napravo od pozice k . Nachází-li se na začátku na pozici k nebo nalevo od ní, během celého postupu se nikdy nedostane napravo od pozice k .

Po m kolech je tedy každý balvan na své koncové pozici nebo nalevo od ní. Snadno nahlédneme, že to je možné pouze pokud jsou balvany setříděné dle velikosti a každý je přesně na své koncové pozici. Postup tedy zabere přesně m kol. Po určení koncových pozic nám určení m zabere pouze lineární čas, časová složitost řešení tedy je $\mathcal{O}(n \log n)$. Paměťová složitost je $\mathcal{O}(n)$.

Následující program řeší podúlohy B a C výše popsáním způsobem.

Program (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); // zarážka

vector<int> vystup;
int zac = 0;
int doleva = 0; // o kolik nejvíc doleva musíme posunout nějaký balvan

while (zac < n) {
    // najdeme aktuální úsek lehkých balvanů
    int kon=zac;
    while (H[kon] <= w)
        ++kon;
    // uspořádáme si je, přičemž si pamatujeme jejich počáteční pozice
    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, který se posunul nejvíc doleva
    for (int i=zac; i<kon; ++i)
        doleva = max( doleva, balvany[i-zac].second - i );
    // vyplníme výstupní pole pro podúlohu B
    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: " << (doleva+1) << endl;

return 0;
}

```

P-II-2 Obdélník

Před čtením tohoto vzorového řešení se doporučujeme seznámit s pojmy a algoritmy popsánymi zde: <https://ksp.mff.cuni.cz/encyklopedie/geometrie/>

V zadání jsme již viděli, jak řešit případ, kdy všechny body leží na jedné přímce. Ve zbytku řešení proto můžeme předpokládat, že tento speciální případ nenastane.

Hledání řešení můžeme ještě trochu zjednodušit následujícím úvahou: Vezměme libovolný obdélník, který má všechny dané body na svém obvodu. Pro každou stranu zvlášť zkontrolujeme, zda obsahuje některý z daných bodů. Pokud ne, můžeme obdélník zmenšit tak, že tuto stranu posuneme blíže k protilehlé straně, dokud nenarazí na jeden ze zadaných bodů. Na konci tohoto procesu získáme obdélník, který nejenže obsahuje všechny zadané body, ale také má alespoň jeden ze zadaných bodů na každé ze svých stran. Stačí nám tedy hledat pouze řešení splňující tuto vlastnost.

Pomalejší řešení

Problém si můžeme výrazně usnadnit tím, že najdeme konvexní obal všech daných bodů. Pokud existuje řešení, co můžeme říci o tomto konvexním obalu?

Na každé ze čtyř stran našeho obdélníku máme jeden nebo více bodů. Vezměme si nejkratší úsečku obsahující všechny tyto body (pokud je jen jeden, tato úsečka je triviální a skládá se pouze z tohoto bodu). Povšimněme si, že tato úsečka je nutně součástí hranice konvexního obalu. Jelikož všechny zadané body leží na hranici obdélníka, všechny leží na (alespoň) jedné z těchto čtyř úseček. Konvexní obal zadaných bodů je tedy roven konvexnímu obalu těchto čtyř úseček. Pokud si tedy z každé takové úsečky vezmeme její krajní body, získáme všechny vrcholy konvexního obalu. Z toho vyplývá, že konvexní obal bude nutně nejvýše osmiúhelník. Jelikož máme navíc zadáno více než osm bodů, alespoň jedna z netriviálních úseček tvořících hranici konvexního obalu bude obsažena v jedné ze stran obdélníku.

Problém tedy můžeme vyřešit následovně: Najdeme konvexní obal všech n zadaných bodů (to zvládneme v čase $\mathcal{O}(n \log n)$). Pokud má více než 8 vrcholů, hledaný obdélník neexistuje. Pokud některý z daných bodů leží uvnitř konvexního obalu, obdélník také neexistuje. Ve zbývajících případech stačí vyzkoušet maximálně 8 případů volby úsečky, která leží v jedné ze stran obdélníku. Poté už je snadné zbylé strany dopočítat (detaily viz v níže popsáném vzorovém řešení).

Vzorové řešení

Problém můžeme také vyřešit v lineárním čase, aniž bychom explicitně konstruovali konvexní obal.

Začneme následující úvahou: Vezmeme libovolných pět z daných n bodů. Pokud všechny leží na obvodu obdélníku, pak podle Dirichletova principu musí existovat (alespoň) dva z nich, které leží na stejné straně. Prozkoumáme tedy všechny dvojice A a B vybraných pěti bodů. Pro každou z nich vyřešíme jednodušší problém: budeme hledat obdélník, který má navíc tu vlastnost, že jedna z jeho stran leží na přímce určené těmito dvěma body. Pokud má některý z těchto problémů řešení, máme také řešení původního problému. Naopak pokud žádný z těchto úkolů nemá řešení, hledaný obdélník jistě neexistuje.

Tímto způsobem vyzkoušíme pouze deset možností. Pokud otestujeme každou možnost v lineárním čase, získáme celkově také lineární řešení.

Zbývá tedy popsat, jak nalézt obdélník za dodatečného předpokladu, že jedna jeho strana leží na přímce AB . Nejprve určíme, které ze zadaných bodů leží na této přímce. To můžeme udělat pomocí vektorového součinu: Bod C leží na přímce AB , pokud je vektorový součin vektorů \overrightarrow{AB} a \overrightarrow{AC} nulový.

Z bodů, které leží na naší přímce, pak vybereme první a poslední. To můžeme provést pomocí skalárního součinu: hledáme nejmenší a největší hodnoty skalárního součinu \overrightarrow{AB} a \overrightarrow{AC} .

Ostatní body (ty, které neleží na této přímce) musí ležet ve stejné polorovině od ní. To můžeme opět ověřit pomocí stejného vektorového součinu: Ten musí mít stejné znaménko pro všechny body C , pro které je odlišný od nuly. Pokud toto neplatí, řešení neexistuje.

Ostatní body nyní rozdělíme body do dvou skupin: Ty, které jsou nejdále od naší přímky, a ostatní. To můžeme provést pomocí skalárního součinu, tentokrát

vynásobením \overrightarrow{AC} s *normálovým vektorem* naší přímkou, tj. vektorem kolmým na \overrightarrow{AB} (vektor kolmý na vektor (x, y) je například $(-y, x)$).

Body, které jsou nejdále, musí ležet na opačné straně obdélníku, který hledáme. Zbývající body (pokud nějaké zbývají) nazveme *bočními body*. Všechny boční body musí ležet na druhé dvojici rovnoběžných stran. Zbývá jen ověřit, zda toho můžeme dosáhnout, tedy zda můžeme vybrat zbývající dvě strany obdélníku tak, aby pokryly všechny boční body.

Všechny body promítneme kolmo na původní přímkou AB (to můžeme opět provést pomocí skalárního součinu). Nechť CD je nejkratší úsečka, obsahující body ležící přímo na přímce AB a kolmé průměty všech bodů z druhé rovnoběžné přímky. Celá tato úsečka CD musí být obsažena ve straně obdélníka na přímce AB , proto žádný z bočních bodů nemůže mít svůj kolmý průmět ostře uvnitř úsečky CD . Navíc průměty všech bočních bodů, které leží na jedné straně úsečky CD , musí být stejné – všechny musí ležet na jedné straně obdélníka kolmé na přímkou AB . Pokud toto nastává, snadno sestrojíme hledaný obdélník: Dva jeho vrcholy jsou „nejlevější“ a „nejpravější“ z kolmých průmětů na přímkou AB . Třetí můžeme najít tak, že se od jednoho z nich posuneme podél normálového vektoru o vzdálenost, ve které se nachází protilehlá strana.

Program (C++):

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

// geometrické funkce

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); }

// řešení úlohy

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) {
    // rozřídíme všechny body dle jejich pozice vzhledem k přímce AB
    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 {};

    // ošetříme případ, kdy všechny zadané body jsou na stejné přímce
    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 body, které nejsou nejdále od přímky AB, tedy boční body
    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);
    }

    // vše promítneme na přímku AB
    point_seq priemety;
    point unit = (B-A) / size(B-A);
    for (auto x : vstup)
        priemety.push_back( A + unit*dot_product(unit, x-A) );

    // zkontrolujeme, zda průměty bočních bodů jsou
    // "nejlevější" a "nejpravější" na AB
    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 {};
    }

    // vyrobíme tři rohy obdélníka
    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 tretí = nalavo.empty() ? cand2 : cand1;
    return { krajne[0], krajne[1], tretí };
}

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 << "NE" << endl;
return 0;
}

```

Alternativní vzorové řešení

Myšlenky obou výše uvedených řešení můžeme zkombinovat do řešení s lineární časovou složitostí následujícím způsobem: Začneme hledáním konvexního obalu, ale místo jedné z metod s časovou složitostí $\mathcal{O}(n \log n)$ použijeme algoritmus „balení dárku“ (také nazývaný Jarvisův algoritmus). Tento algoritmus konstruuje konvexní obal bod po bodu a má časovou složitost $\mathcal{O}(hn)$, kde h je počet vrcholů výsledného konvexního obalu. Tento algoritmus je obecně pomalejší, protože v nejhorším případě je jeho časová složitost kvadratická v počtu bodů ve vstupu. V naší úloze však víme, že jakmile získáme devátý bod na konvexním obalu, můžeme algoritmus ukončit a dát zápornou odpověď. V lineárním čase tedy získáváme buď odpověď NE, nebo celý konvexní obal našich bodů.

P-II-3 Platónova Akademie

Pro každé i začneme v optimálním řešení studovat předmět $i + 1$ nejvýše dva měsíce po zahájení studia předmětu i , pro volbu počátečního měsíce studia předmětu $i + 1$ tedy vždy existují nejvýše tři možnosti. Vyzkoušením a kontrolou všech těchto možností pro každé i získáme správné, ale pomalé řešení s exponenciální časovou složitostí.

Efektivní řešení pro malý počet hodin

Připomeňme, že číslo p udává maximální počet hodin, které můžeme každý měsíc strávit studiem. Stručně nastiňme jedno možné řešení, které je efektivní, pokud je toto číslo p malé.

Řekněme, že už jsme se nějak rozhodli pro to, které předměty začít studovat v prvních pár měsících. Pro naše další rozhodování jsou důležité jen dva parametry: Počet x předmětů, které jsme ještě nezačali studovat, a počet z hodin, které ještě máme k dispozici v následujícím měsíci (zatímco ostatní potřebujeme na dokončení předmětů započatých v posledním měsíci).

Pro každé x a z si můžeme položit otázku, kolik nejméně měsíců nám může trvat dostudování zbývajících předmětů. Na každou takovou otázku můžeme odpovědět vyzkoušením všech možností, kolik předmětů začneme studovat v následujícím měsíci. Pro každou takovou možnost dostaneme situaci stejného typu o měsíc později, jen s jinými hodnotami x a z . Problém tedy můžeme řešit rekurzivní funkcí s parametry x a z . Pro zefektivnění můžeme přidat kešování (memoizaci), tj. výsledek pro danou kombinaci parametrů si uložíme a při opakovaném volání funkce se stejnými parametry pouze vrátíme uloženou hodnotu. Tím zajistíme, že pro každou kombinaci parametrů provádíme výpočet nejvýše jednou, a dostáváme řešení s časovou složitostí $\mathcal{O}(n^2p)$.

Řešení s kubickou časovou složitostí

Začneme předběžným výpočtem prefixových součtů pro pole A a B . Díky tomu budeme umět pro libovolný souvislý úsek předmětů určit, kolik hodin nám zaberou ve kterém měsíci, pokud je začneme studovat naráz.

Označme $D[y]$ minimální počet měsíců potřebných k úplnému vystudování prvních y předmětů. Dále označme $M[x][y]$ minimální počet měsíců potřebných k úplnému vystudování prvních y předmětů tak, abychom prvních x předmětů měli dostudovaných již před koncem minulého měsíce. Z hodnot M snadno spočítáme řešení zadané úlohy. Konkrétně, $D[y]$ je jednoduše minimum ze všech $M[x][y]$ pro $x \in \{0, 1, \dots, y - 1\}$; a řešení celé úlohy je hodnota $D[n]$.

Ukažme si, jak postupně vypočítat hodnoty M a D . Zjevně máme $M[0][0] = 0$ a $D[0] = 0$. Když chceme vypočítat konkrétní hodnotu $M[x][y]$, víme, že během posledního měsíce jsme museli dostudovat předměty s čísly $x, x + 1, \dots, y - 1$. Pokud by nám tyto předměty zabrali více než p hodin, pak $M[x][y] = \infty$: Této situace nemůžeme vůbec dosáhnout.

V ostatních případech určíme $M[x][y]$ rozbořem dvou případů dle toho, co jsme dělali předposlední měsíc.

První možností je, že jsme studovali pouze stejných $y - x$ předmětů jako poslední měsíc. Optimálním řešením v této variantě je samozřejmě dostudovat prvních x předmětů co nejrychleji a poté přidat dva měsíce na následujících $y - x$ předmětů, celkem tedy budeme potřebovat $D[x] + 2$ měsíců.

Druhou možností je, že před dvěma měsíci jsme měli dostudováno pouze z předmětů, pro nějaké neznámé $z < x$, a poté během předposledního měsíce jsme dostudovali předměty s čísly $z, z + 1, \dots, x - 1$ a začali studovat předměty $x, \dots, y - 1$. Kdybychom znali konkrétní z , mohli bychom říci, jak dlouho by celý proces trval: $M[z][x] + 1$ měsíců.

Hodnotu $M[x][y]$ zjistíme výběrem nejkratší doby trvání z těchto možností. Vezmeme tedy minimum z hodnoty $D[x] + 2$ a všech hodnot $M[z][x] + 1$ takových, abychom během předposledního měsíce stíhali studovat všechny předměty s čísly $z, z + 1, \dots, y - 1$.

Hodnoty $M[x][y]$ můžeme počítat v cyklu pro všechna x , a pro každé x v cyklu pro všechna $y > x$. Poté, co takto dopočítáme hodnoty $M[x][y]$ pro všechna $y > x$,

již známe všechny hodnoty $M[x'][x+1]$ pro $x' \leq x$, a z nich můžeme určit hodnotu $D[x+1]$ jako jejich minimum.

Celkově musíme vypočítat $\mathcal{O}(n^2)$ různých hodnot v poli M . Výpočet každé z nich nám zabere nanejvýš lineární čas, protože musíme vyzkoušet $\mathcal{O}(n)$ různých hodnot z . Celková časová složitost tohoto řešení bude tedy $\mathcal{O}(n^3)$.

Vzorové řešení (s kvadratickou časovou složitostí)

Všimněte si, že všechny výpočty hodnot $M[x][y]$ pro různé y jsou velmi podobné: vždy začínáme se stejnou hodnotou $D[x]+2$ a poté zkoumáme hodnoty $M[z][x]$ pro přípustné z . Jediná věc, která se mění v závislosti na y , je to, které hodnoty jsou přípustné. Přesněji řečeno, čím menší y zvolíme, tím méně hodin potřebujeme ke studiu předmětů v posledním měsíci a tím více jich tedy můžeme použít na dokončení předmětů započatých v předposledním měsíci; to znamená, že větší z může být stále přípustné.

S využitím tohoto pozorování nyní vylepšíme časovou složitost řešení popsaného v předchozí části. Podobně jako v tomto řešení budeme postupně počítat všechny hodnoty $M[x][y]$ pro pevné x . Tentokrát to však uděláme v opačném směru: začneme s $y = n$ a postupně budeme snižovat y až k $y = x + 1$. Když postupně procházíme všechny hodnoty y v sestupném pořadí, v každém kroku zůstává množina přípustných hodnot z stejná nebo se zvětšuje. Místo toho, abychom procházeli znovu všechna z pro každé y , stačí vzít předchozí hodnotu a pokud přibudou nějaké nové přípustné hodnoty z , projít je a zjistit, zda nám dávají nové, lepší řešení.

Pro každé konkrétní x takto vypočítáme všechny hodnoty $M[x][y]$ v celkovém čase $\mathcal{O}(n)$, jelikož každé přípustné y a z zpracujeme právě jednou. Celkově tedy dostaneme časovou složitost $\mathcal{O}(n^2)$.

Program (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) {
        // výpočet M[x][y]
        // zkontrolujeme, zda lze zároveň studovat x, ..., y-1
        int treba_minuly = PA[y] - PA[x], treba_tento = PB[y] - PB[x];
        if (treba_minuly > p || treba_tento > p) continue;
        // jestliže ano, přibyly nějaké možné menší hodnoty z?
        while (z > 0 && treba_minuly + PB[x] - PB[z-1] <= p) {
            --z;
            best = min( best, M[z][x]+1 );
        }
        M[x][y] = best;
    }
    // teď můžeme dopočítat 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;

return 0;
}

```

P-II-4 Řešič to opět vyřeší

Podúloha A

Stejně jako v domácím kole stačí mít pro každý projekt jednu binární proměnnou. Hodnota této proměnné je 0, pokud projekt nepodporujeme, nebo 1, pokud jej podporujeme. Slabou závislost projektu x na projektech y_1, \dots, y_k modelujeme jednoduše jako nerovnost $x \leq y_1 + \dots + y_k$. Zbytek řešení zůstává stejný jako v domácím kole.

Podúloha B

Pro každý kopec i budeme mít tři binární proměnné $z_{i,1}, z_{i,2}$ a $z_{i,3}$, jejichž hodnoty udávají, zda na tomto kopci postavit první, druhou a třetí sjezdovku. V praxi je také vhodné k těmto proměnným přidat nerovnosti $z_{i,1} \geq z_{i,2} \geq z_{i,3}$. Tyto nerovnosti můžeme interpretovat tak, že druhou sjezdovku můžeme otevřít pouze v případě, že jsme otevřeli také první, a tak dále. Řešič pak nemusí zbytečně zkoumat logiky ekvivalentní větve, jako je možnost otevřít pouze třetí sjezdovku namísto pouze první sjezdovky.

Dále budeme mít pro každou sjezdovku celočíselnou proměnnou $d_{i,j}$ označující její přesnou délku: nula, pokud ji nepostavíme, nebo číslo mezi ℓ_i a u_i , pokud ji postavíme. Celková délka sjezdovek pak bude jednoduše součtem všech $d_{i,j}$. Tento součet by měl být přesně roven dané hodnotě d .

Celkové náklady na výstavbu sjezdovek budou součtem součinů $c_i \cdot d_{i,j}$ pro všechna i a j . Chceme tyto náklady minimalizovat, což bude cílem našeho ILP.

Stále nám chybí nejdůležitější část našeho programu: podmínky, které zajistí, že hodnoty $d_{i,j}$ budou mít výše uvedený rozsah. Klíčovou součástí řešení této úlohy bylo vymyslet, jak toho můžeme dosáhnout pouze pomocí lineárních nerovností.

Podívejme se na hodnoty $\ell_i \cdot z_{i,j}$ a $u_i \cdot z_{i,j}$. V obou případech se jedná o výrazy ve tvaru konstanta krát proměnná, se kterými umíme pracovat. Pokud přiřadíme proměnné $z_{i,j}$ hodnotu nula (tj. pokud se rozhodneme tuto konkrétní sjezdovku nepostavit), budou mít oba výše uvedené výrazy hodnotu 0. Naopak, pokud $z_{i,j} = 1$, budou mít tyto dva výrazy hodnoty ℓ_i a u_i . Proto stačí do našeho ILP přidat následující nerovnosti:

$$\ell_i \cdot z_{i,j} \leq d_{i,j} \leq u_i \cdot z_{i,j}.$$

Ty v obou případech zaručí správný rozsah pro délku příslušné sjezdovky.