72. ročník Matematické olympiády – 2022/2023

Řešení úloh ústředního kola kategorie P – 1. soutěžní den

P-III-1 Raut

Nejprve vyřešme jednodušší úlohu – co kdyby si Vašek s sebou na přednášku nemohl vzít žádný zákusek? To je známá úloha, takzvaný problém batohu.

Tu můžeme řešit pomocí dynamického programování. Pořídíme si tabulku DP o rozměrech $(n+1)\cdot(T+1)$, kde (i,j) nám říká maximální celkovou mňamóznost zákusků, které jsme schopni sníst za j vteřin s použitím zákusků 1 až i. Tuto tabulku budeme počítat po řádcích. Nultý řádek je jednoduchý – jsou to samé nuly. Pro i-tý řádek máme pro každé j dvě možnosti – buďto sníme i-tý zákusek, nebo ne. Pochopitelně vezmeme vždy lepší variantu z těchto dvou. Tedy $DP[i,j] = \min(m_i + DP[i-1, j-t_i], DP[i-1,j])$. Pomocí této rekurence jsme schopni řešit jednodušší úlohu v $\mathcal{O}(nT)$ – výsledek totiž najdeme v $DP[n,T]$.

Jak nám toto pomůže na původní úlohu? Můžeme vyzkoušet všechny možnosti, jak vybrat zákusek na přednášku – těch je n. Pro zbylé můžeme napočítat batoh předchozím algoritmem a vrátit celkové optimum. Tím umíme hledat hodnotu optima v čase $\mathcal{O}(n^2T)$, což nám stačí na vyřešení úlohy pro $n \leq 100$.

To, jak zároveň spočítat, která volba zákusků dává optimum, vyřešíme až úplně nakonec – pro všechny algoritmy je myšlenka stejná.

Co když jsou zákusky stejně dobré, jak dlouho je trvá sníst?

Nyní trávíme hodně času výběrem zákusku, který si vzít na přednášku, protože pro každý musíme napočítat batoh. Ukážeme, že když zákusky jsou tak mňamózní, jak dlouho je trvá sníst, pak si nikdy nepohoršíme, když vezmeme ten nejmňamóznější zákusek na přednášku.

Uvažujme nějaké řešení, které nebere nejmňamóznější zákusek na přednášku. Nahlédněme pro oba možné případy, že existuje řešení, které vezme nejmňamóznější zákusek a je alespoň tak dobré.

- Nejmňamóznější zákusek jsme nikdy nesnědli. Pak je vždy výhodnější zákusek, který bereme, nyní vyměnit za ten nejmňamóznější, čímž si polepšíme.
- Nejmňamóznější zákusek jsme snědli o přestávce. Pak si nepohoršíme, když jej vyměníme za ten, který si bereme na přednášku. Celková mňamóznost totiž zůstane stejná, a pokud stihneme sníst ten nejmňamóznější zákusek během přestávky, tak zvládneme i nějaký méně mňamózní.

Nyní je jednoduché úlohu za těchto podmínek dořešit – vybereme nejmňamóznější zákusek na přednášku a pro zbytek dopočítáme batoh (stejné zákusky jsou nerozlišitelné, takže je-li nejmňamóznějších zákusků více, můžeme vybrat kterýkoliv). To nám zabere jen $\mathcal{O}(nT)$.

Vzorové řešení

Pro jednoduchost nyní předpokládejme, že zákusky jsou vzestupně seřazeny dle doby, jak dlouho je jíme. Jinak je můžeme v čase $\mathcal{O}(n \log n)$ seřadit.

Pro vyřešení úlohy na plný počet bodů je třeba si uvědomit, že argumenty z předchozí části jsou ve skutečnosti mnohem obecnější.

První pozorování nám totiž říká, že vždy, když vybereme nějakou množinu zákusků, kterou sníme o přestávce, je nejvýhodnější vzít si na přednášku ten nejmňamóznější zbývající zákusek.

Druhé nám říká, že vybereme-li, které zákusky sníme, pak nic nepokazíme tím, že na přednášku si z nich vybereme ten, který nám potrvá sníst co nejdéle.

Spojením těchto dvou pozorování dostáváme, že je-li i nejvyšší index takový, že jsme snědli i-tý zákusek o přestávce, pak jedná-li se o optimum pro nějakou volbu zákusku na přednášku, pak se jedná o optimum, i pokud si na přednášku vezmeme nejmňamóznější ze zákusků $i+1$ až n.

Jenže nějaký index i, ve kterém nastává optimum, musí existovat. A pro ten nejlepší volba zákusků s indexy nejvýše i na přestávku je problém batohu. Jenže počítat batoh pro všechna i je opět moc pomalé. Stačí si však uvědomit, že to vůbec není potřeba a že při počítání batohu pro $i = n$ jej napočítáme pro všechna i.

Optimální řešení je tedy $\max_i(DP[i, T] + \max_{i<j\leq n} m_j)$. Hodnoty $\max_{i<j\leq n} m_j$ si můžeme v $\mathcal{O}(n)$ předpočítat. Dostáváme tak algoritmus s časovou složitostí $\mathcal{O}(n \cdot (T + \log n))$, kde logaritmus je kvůli třídění. Paměťová složitost je $\mathcal{O}(nT)$, protože si pamatujeme celou tabulku.

Nyní zbývá rozmyslet, jak hledat nejen hodnotu optima, ale i jaké zákusky sníst, aniž by nám to pokazilo složitosti.

Zjistit, pro jaká i a j nabývá předchozí výraz maxima, není těžké – jen si vždy tato nejlepší i a j updatujeme. Zbývá tak domyslet, jak se znalostí i získat i jaké zákusky sníst o přestávce.

Když provádíme dynamické programování, můžeme si vždy uložit, v jakém případě (snědení / nesnědení zákusku) jsme snědli mňamóznější zákusky. S touto informací již můžeme začít v bodě $DP[i, T]$ a vždy se posouvat odpovídajícím způsobem zpět, tedy buď do $DP[i-1, T]$, nebo do $DP[i-1, T-t_i]$. V druhém případě je třeba aktualizovat, jaké zákusky jsme snědli – tedy přidat i-tý zákusek.

To stačí na plný počet bodů. Pokud by se nám však nelíbil logaritmus ve výsledné časové složitosti, můžeme se jej zbavit. Stačí si rozmyslet, že pro správnou funkčnost algoritmu nám stačí seřadit pouze všechny prvky, které jsou menší nebo rovny T. Ostatní prvky pak můžeme nechat na konci seznamu v libovolném pořadí – stejně žádný z nich nebudeme moci sníst o přestávce. Ty relevantní prvky tak umíme setřídit v $\mathcal{O}(T)$ pomocí přihrádkového třídění. Tedy dostaneme obě složitosti algoritmu $\mathcal{O}(nT)$.

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

int main() {
	int n, T;
	cin >> n >> T;

	vector<vector<int>> dp(n + 1, vector<int> (T + 1));
	vector<pair<int,pair<int, int>>> cakes(n);
	vector<int> ms(n);
	for (int i = 0; i < n; ++i) cin >> ms[i];
	for (int i = 0; i < n; ++i) {
		int x; cin >> x;
		cakes[i] = {x, pair<int,int>(ms[i], i + 1)};
	}

	sort(cakes.begin(), cakes.end());

	// Dynamika.
	for (int i = 0; i < n; ++i) {
		for (int j = 0; j <= T; ++j) {
			dp[i+1][j] = dp[i][j];
			int prev = j - cakes[i].first;
			if (prev >= 0) dp[i + 1][j] = max(dp[i + 1][j],
				dp[i][prev] + cakes[i].second.first);
		}
	}

	// Předpočítání suffixových maxim.
	vector<int> maxima(n, cakes[n-1].second.first);
	vector<int> maxvals(n, cakes[n-1].second.second);
	for (int i = n - 2; i >= 0; --i) {
		if (cakes[i].second.first > maxima[i+1])
			maxvals[i] = cakes[i].second.second;
		else
			maxvals[i] = maxvals[i + 1];
		maxima[i] = max(maxima[i+1], cakes[i].second.first);
	}

	// Nalezení optima.
	int best = -1;
	int p = -1;
	for (int i = 0; i < n; ++i) {
		int curr = dp[i][T] + maxima[i];
		if (curr > best) {
			best = curr;
			p = i;
		}
	}

	// Nalezení, které zákusky je třeba sníst.
	int row = p;
	int time = T;
	vector<int> eaten;
	while (row) {
		if (dp[row][time] != dp[row - 1][time]) {
			time -= cakes[row-1].first;
			eaten.push_back(cakes[row-1].second.second);
```

```
        }
        --row;
    }

    for (int index = 0; index < (int)eaten.size(); ++index) {
        if (index) cout << " ";
        cout << eaten[index];
    }
    cout << "\n" << maxvals[p] << "\n";
}
```

P-III-2 Lezení

Na vstupu máme zadaných n dvojic $(b_1, \ell_1), \ldots, (b_n, \ell_n)$ a chceme najít n nezáporných reálných čísel $t_1, \ldots, t_n$ takových, že

$$\sum_{i=1}^{n} t_i b_i \geq B, \quad \sum_{i=1}^{n} t_i \ell_i \geq L$$

a hodnota $\sum_{i=1}^{n} t_i$ je nejmenší možná. To si představíme jako úlohu o vektorech v rovině: Máme zadaných n vektorů (které všechny leží neostře v prvním, tj. „pravém horním" kvadrantu) a chceme je sečíst s nezápornými koeficienty tak, aby výsledný vektor byl po složkách alespoň tak velký jako (B, L). Všimněte si, že pokud λ je nějaká konstanta, pak $\lambda \sum_{i=1}^{n} t_i b_i = \sum_{i=1}^{n} (\lambda t_i) b_i$ a analogicky pro ℓ_i. Tato vlastnost se označuje jako *linearita*.

Předpokládejme, že máme nějaké optimální řešení $t_1, \ldots, t_n$. Postupně o takovém řešení dokážeme několik pozorování, která nám nakonec umožní vymyslet řešení úlohy. První pozorování je, že nemůže najednou platit, že

$$\sum_{i=1}^{n} t_i b_i > B \quad \text{a} \quad \sum_{i=1}^{n} t_i \ell_i > L.$$

Kdyby totiž platilo obojí najednou, můžeme zvolit $\lambda < 1$ takové, že

$$\lambda \sum_{i=1}^{n} t_i b_i \geq B \quad \text{a} \quad \lambda \sum_{i=1}^{n} t_i \ell_i \geq L.$$

Z linearity potom plyne, že $\lambda t_1, \ldots, \lambda t_n$ je také řešení úlohy, a protože $\lambda t_1 + \cdots + \lambda t_n < t_1 + \cdots + t_n$, dostáváme spor s tím, že $t_1, \ldots, t_n$ bylo optimální řešení.

Dvě stěny

Pozorování z předchozího odstavce stačí na vyřešení vstupů, kdy jsou v Praze jen dvě stěny, a tedy na zisk dvou bodů. Řekněme, že Ondra stráví t_1 hodin na první stěně a t_2 hodin na druhé stěně. Potřebujeme minimalizovat $t_1 + t_2$ za podmínek $t_1, t_2 \geq 0$, $t_1 b_1 + t_2 b_2 \geq B$ a $t_1 \ell_1 + t_2 \ell_2 \geq L$. Nejprve vyzkoušíme varianty, kdy $t_1 = 0$ nebo $t_2 = 0$. V takovou chvíli je jednoduché spočítat, kolik hodin musí Ondra strávit na dané stěně. Dále můžeme předpokládat, že $t_1 > 0$ a $t_2 > 0$.

Z předchozího odstavce víme, že alespoň v jedné z nerovností $t_1b_1 + t_2b_2 \geq B$ a $t_1\ell_1 + t_2\ell_2 \geq L$ nastane rovnost. Pokud nastane v první, můžeme vyjádřit $t_2 = (B - t_1b_1)/b_2$, dosadit to do druhé nerovnosti a dostaneme

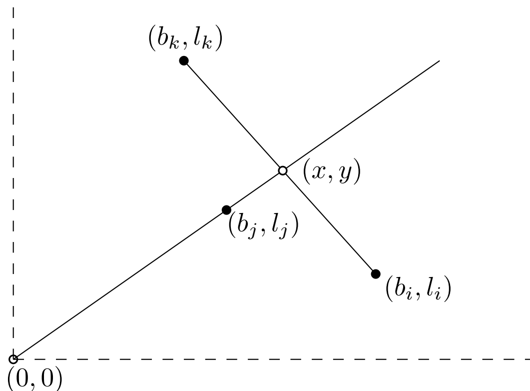
$$t_1\ell_1 + \frac{B - t_1b_1}{b_2}\ell_2 \geq L.$$

To je lineární nerovnost o jedné neznámé, takže ji můžeme jednoduše vyřešit a získat tak řešení t_1, t_2. Rovnost ale může nastat i ve druhé nerovnosti. V takovém případě uděláme analogické kroky a dostaneme (potenciálně jiné) řešení t_1', t_2'. Tím máme celkem čtyři kandidáty (dva, v nichž Ondra navštíví pouze jedno stěnu, a dva z tohoto odstavce) a z nich vybereme toho s nejmenším celkovým časem stráveným na stěnách.

Někteří z vás si možná všimli malých chyb v předchozím odstavci: Jednak nám může vyjít, že Ondra má na některé ze stěn strávit záporné množství času. My ale předpokládali, že $t_1, t_2 > 0$, takže v takovém případě můžeme dané potenciální řešení ignorovat. Kromě toho tam máme ještě jednu nepřesnost: Kdyby existovalo reálné číslo α takové, že $b_1 = \alpha b_2$ a $\ell_1 = \alpha\ell_2$, pak po dosazení dostaneme nerovnost $B\frac{\ell_2}{b_2} \geq L$, která neobsahuje žádné neznámé. Je to proto, že vektory (b_1, ℓ_1) a (b_2, ℓ_2) jsou *lineárně závislé*. Každopádně v tomto případě je zřejmé, že je-li $\alpha \geq 1$, může Ondra všechen čas strávit na první stěně, jinak všechen čas stráví na druhé stěně, takže použijeme řešení pro jednu stěnu.

Dvě stěny vždy stačí

Další pozorování je, že existuje optimální řešení, v němž jsou nejvýše dvě hodnoty z $t_1, \ldots, t_n$ nenulové. Pro spor předpokládejme, že to není pravda a že $t_1, \ldots, t_n$ je optimální řešení, které má nejmenší počet nenulových hodnot, ale existují $1 \leq i < j < k \leq n$ taková, že $t_i, t_j, t_k > 0$. Bez újmy na obecnosti můžeme předpokládat, že vektory $(b_1, \ell_1), \ldots, (b_n, \ell_n)$ jsou seřazené podle úhlu, který svírají s kladnou poloosou x. Tedy vektor (b_j, ℓ_j) je mezi vektory (b_i, ℓ_i) a (b_k, ℓ_k). Podívejme se nyní na úsečku spojující body (b_i, ℓ_i) a (b_k, ℓ_k) a na přímku spojující body $(0, 0)$ a (b_j, ℓ_j). Označme si jako (x, y) bod, kde se tahle úsečka a přímka protnou.



Jelikož (x, y) leží na úsečce spojující body (b_i, ℓ_i) a (b_k, ℓ_k), existuje reálné číslo $0 \leq \lambda \leq 1$ takové, že $x = \lambda b_i + (1 - \lambda)b_k$ a $y = \lambda \ell_i + (1 - \lambda)\ell_j$.

Rozebereme nyní dvě možnosti. Pokud $(0, 0)$ a (b_j, ℓ_j) leží na (neostře) stejné straně úsečky (b_i, ℓ_i), (b_k, ℓ_k), potom je bod (x, y) „dále“ od počátku než bod (b_j, ℓ_j), a kdyby existovala stěna s parametry (x, y), mohl by Ondra strávit na této hypotetické stěně t_j hodin místo j-té stěny a nepohoršil by si. Protože ale $x = \lambda b_i + (1-\lambda)b_k$ a $y = \lambda \ell_i + (1 - \lambda)\ell_j$, můžeme tuto hypotetickou stěnu (x, y) nasimulovat pomocí i-té a k-té stěny tak, že na i-té stěně strávíme o λt_j více hodin a na k-té stěně strávíme o $(1 - \lambda)t_j$ více hodin. Formálně zapsáno, $t_j b_j \leq \lambda t_j b_i + (1 - \lambda)t_j b_k$ a totéž pro ℓ_j. Můžeme tedy vytvořit nové řešení $t'_1, \ldots, t'_n$ takové, že $t'_j = 0$, $t'_i = t_i + \lambda t_j$, $t'_k = t_k + (1 - \lambda)t_j$ a $t'_\ell = t_\ell$ pro všechny ostatní stěny. Toto je jistě vyhovující řešení a navíc platí $\sum_{\ell=1}^{n} t'_\ell = \sum_{\ell=1}^{n} t_\ell$, takže je to optimální řešení. Zároveň má ale o jednu nenulovou hodnotu méně než $t_1, \ldots, t_n$, což je spor.

Pokud $(0, 0)$ a (b_j, ℓ_j) leží na opačných stranách úsečky (b_i, ℓ_i), (b_k, ℓ_k), je stěna (b_j, ℓ_j) ostře lepší než hypotetická stěna (x, y). V předchozím odstavci jsme si rozmysleli, že pokud Ondra na i-té stěně stráví λ hodin a na k-té stěně stráví $(1 - \lambda)$ hodin, je to totéž, jako by na stěně (x, y) strávil 1 hodinu. A kdyby tu hodinu strávil místo toho na j-té stěně, tak by si pomohl. Označme $\tau = \min(\frac{t_i}{\lambda}, \frac{t_k}{(1-\lambda)})$, potom může Ondra strávit navíc τ hodin na j-té stěně, z času na i-té stěně ubrat $\lambda\tau$ hodin a z času na k-té stěně ubrat $(1 - \lambda)\tau$ hodin. Stejně jako v předchozím odstavci se toho Ondra takto naučí alespoň tolik jako v původním řešení, zabere mu to stejně času, ale buď na i-tou, anebo na k-tou stěnu vůbec nebude muset, tedy jsme našli řešení s menším počtem nenulových hodnot, což je spor.

Toto pozorování nám dává kvadratické řešení a stačí na zisk pěti bodů: Pro každou dvojici stěn spočítáme jako v řešení za dva body optimální časy, kdyby měl Ondra k dispozici jen tyto dvě stěny. Dvojic stěn je řádově n^2, takže v čase $\mathcal{O}(n^2)$ spočítáme nejlepší možnosti pro všechny dvojice a z nich nakonec vybereme optimální řešení.

Některé stěny jsou lepší než jiné

Třetí pozorování je jednoduché: Pokud existují dvě stěny (b_i, ℓ_i) a (b_j, ℓ_j) takové, že $b_i \geq b_j$ a $\ell_i \geq \ell_j$, potom můžeme na j-tou stěnu zapomenout, protože všechen čas, který Ondra stráví na j-té stěně, může místo toho strávit na i-té stěně a nepohorší si.

Toto vede k řešení za sedm bodů. Nejprve si všechny stěny seřadíme primárně sestupně podle b_i a sekundárně sestupně podle ℓ_i. Potom tuto seřazenou posloupnost od začátku projdeme a necháme si pouze ty stěny, které mají větší ℓ_i než dosud nejvyšší hodnota ℓ_i, čímž se zbavíme právě všech stěn, které jsou horší než nějaká jiná stěna. Všimněte si, že pro každé x nám zůstane nejvýše jedna stěna s $b_i = x$, tedy celkem nám zůstane nejvýše $\max(b_i)$ stěn. Na ně potom můžeme spustit kvadratické řešení a dostaneme celkovou složitost $\mathcal{O}(n \log n + \max(b_i)^2)$.

Konvexní obal

Když jsme dokazovali, že vždy stačí jít na nejvýše dvě stěny, použili jsme úvahu, že když máme dvě stěny, tak pomocí nich umíme simulovat libovolnou „virtuální“ stěnu s parametry ležícími na úsečce spojující parametry daných dvou stěn. Na hodnotě dva přitom není nic speciálního: Pokud si obecně vezmeme libovolných k stěn s indexy $1 \leq i_1, \ldots, i_k \leq n$ a libovolných k nezáporných reálných čísel $\lambda_1, \ldots, \lambda_k$ takových, že $\lambda_1 + \cdots + \lambda_k = 1$, potom pomocí stěn číslo $i_1, \ldots, i_k$ umíme simulovat virtuální stěnu s parametry $(\sum_{j=1}^{k} \lambda_j b_{i_j}, \sum_{j=1}^{k} \lambda_j \ell_{i_j})$. A takové virtuální stěny jsou přesně ty, které leží v konvexním obalu* parametrů daných k stěn.

Z toho plyne, že nám stačí uvažovat pouze ty stěny, které leží na konvexním obalu, respektive přesněji řečeno na jeho horní polovině. Lze dokázat, že v našem případě bude konvexní obal obsahovat nejvýše $\mathcal{O}(\max(b_i)^{2/3})$ bodů, a tedy získáváme řešení se složitostí $\mathcal{O}(n \log n + \max(b_i)^{4/3})$. Dokázat to dá ovšem nějakou práci a my si místo toho ukážeme ještě jedno pozorování, které nám umožní získat plný počet bodů. Ve zbytku vzorového řešení budeme předpokládat, že stěny leží na horní polovině konvexního obalu a že pro každé dvě stěny $i \neq j$ platí, že $b_i < b_j$ a $\ell_i > \ell_j$ nebo naopak. V čase $\mathcal{O}(n \log n)$ si to umíme na začátku programu zařídit.

Optimální stěny jsou sousední

Všimněte si, že v předchozím odstavci jsme zařídili, že stěny budou zároveň ležet na konvexním obalu a zároveň žádná nebude lepší než nějaká jiná. Tyto dva požadavky jsou nezávislé (na konvexním obalu mohou ležet dvě stěny takové, že jedna je lepší než jiná, a zároveň existují trojice stěn takové, že žádná není lepší než jiná, ale jen dvě z nich leží na horní polovině konvexního obalu). Klíčem k řešení je tato dvě pozorování zkombinovat:

Řekněme, že v optimálním řešení platí, že t_i a t_j jsou jediné dvě potenciálně nenulové hodnoty (tj. jedna z nich také může být nulová, což odpovídá tomu, že Ondra půjde pouze na jednu stěnu). Definujme si $\lambda = \frac{t_i}{t_i + t_j}$. Kdyby existovala stěna s parametry $(\lambda b_i + (1-\lambda) b_j, \lambda \ell_i + (1-\lambda) \ell_j)$, mohl by na ní Ondra strávit $t_i + t_j$ hodin, nejít na žádnou jinou stěnu, a skončil by s úplně stejnými schopnostmi lezení na obtížnost a boulderingu za úplně stejný čas jako v uvažovaném řešení. To znamená, že kdybychom si vytvořili všechny možné virtuální stěny ležící na úsečkách tvořených opravdovými stěnami, stačilo by nám zkoušet řešení, která využívají jen jednu stěnu.

Virtuálních stěn je samozřejmě nekonečně mnoho, takže takové řešení by zrovna moc bodů nezískalo. Ale představme si, že jsme si všechny takové virtuální stěny vytvořili a pak jsme zahodili všechny stěny, které jsou horší než nějaká jiná stěna. Potom nám zbude pouze horní hranice konvexního obalu, která je tvořená úsečkami spojující stěny, jež na konvexním obalu leží vedle sebe. To znamená, že nám stačí uvažovat pouze ty virtuální stěny, které leží na úsečce mezi dvěma sousedními body konvexního obalu. Jinak řečeno, jediné dvojice stěn, které nám stačí uvažovat, jsou

---

* Pro definici konvexního obalu a algoritmus na jeho nalezení v čase $\mathcal{O}(n \log n)$ si můžete přečíst třeba kuchařku KSP Geometrie na *https://ksp.mff.cuni.cz/kucharky/geometrie/*. Ve zbytku řešení budeme takové znalosti předpokládat.

ty, které leží na konvexním obalu a sousedí spolu. Takových dvojic je jistě $\mathcal{O}(n)$ a dostáváme tím řešení, které běží v čase $\mathcal{O}(n \log n)$ a stačí na zisk plného počtu bodů.

Poznámky

Všimněte si, že pokud jako b označíme největší hodnotu b_i na vstupu, jako ℓ označíme největší hodnotu ℓ_i na vstupu a přidáme si stěny $(b, 0)$ a $(0, \ell)$, tak horní polovina konvexního obalu bude tvořit klesající posloupnost, a tedy speciálně nebude obsahovat žádné dvě stěny takové, že by jedna byla lepší než druhá, a nemusíme tedy takové stěny řešit zvlášť. Kdyby optimální řešení jednu z těchto dvou stěn použilo, tak místo ní pošleme Ondru na stěnu, která nabývá daného maxima b resp. l.

Vraťme se k pozorování z druhého odstavce, že v jedné z nerovností

$$\sum_{i=1}^{n} t_i b_i \geq B \quad \text{a} \quad \sum_{i=1}^{n} t_i \ell_i \geq L$$

v každém optimálním řešení nastane rovnost. Pokud v optimálním řešení půjde Ondra pouze na jednu stěnu, tak nic lepšího dokázat nemůžeme. Pokud ale půjde na dvě, pak ve skutečnosti nastane rovnost v obou nerovnostech:

Předpokládejme pro spor, že máme nějaké optimální řešení, kde Ondra půjde na stěny $i < j$ (a na žádné jiné) a platí $t_i b_i + t_j b_j = B$ a $t_i \ell_i + t_j \ell_j = L + \varepsilon$ pro nějaké $\varepsilon > 0$ (kdyby rovnost nastala ve druhé nerovnici, tak je úvaha symetrická). Bez újmy na obecnosti můžeme předpokládat, že $b_i > b_j$ a $\ell_i < \ell_j$. Potom Ondra může na j-té stěně strávit o malinko méně času tak, aby přišel o nejvíce ε schopností lezení na laně a místo toho může na ještě o trochu kratší čas zajít na i-tou stěnu tak, aby tam získal tolik schopností boulderingu, o kolik přišel tím, že byl na j-té stěně o chvilinku méně. Formálně označme $\tau = \frac{\varepsilon}{\ell_j}$ a $\tau' = \frac{b_j}{b_i}\tau$, přičemž zřejmě $\tau > \tau' > 0$, a definujme $t'_i = t_i + \tau'$ a $t'_j = t_j - \tau$. Hodnoty jsme zvolili tak, aby $t'_i b_i + t'_j b_j = B$ a $t'_i \ell_i + t'_j \ell_j \geq L$, tedy je to vyhovující řešení, ale máme $t'_i + t'_j < t_i + t_j$, což je spor s tím, že řešení t_i, t_j bylo optimální.

To má geometrickou interpretaci: Optimální virtuální stěna leží přesně v průsečíku úsečky mezi i-tou a j-tou stěnou s přímkou, která spojuje body $(0, 0)$ a (B, L). Tím se dá úloha převést na hledání průsečíku přímky s horní hranicí konvexního obalu.

```
#include <bits/stdc++.h>
#define INF 99999999999.0

using namespace std;
typedef long long ll;
typedef pair<ll, ll> pll;

int n;
double B, L;
vector<pll> steny;

double jedna_stena (pll x) {
	return max(B / x.first, L / x.second);
}
```

```
// Předpokládáme, že x i y leží na horní části konvexního obalu,
// a tedy nemusíme řešit speciální případy a v optimálním řešení
// nastane rovnost v obou nerovnostech
double dve_steny (pll x, pll y) {
        double t2 = (L*x.first - B*x.second) / (x.first*y.second - x.second*y.first);
        double t1 = (B - t2*b2) / b1;
        if (t2 >= 0 && t1 >= 0) return t1 + t2;
        else return INF;
}

int main() {
        cin >> n >> B >> L;
        ll b, l;
        ll maxb = -1, maxl = -1
        for (int i = 0; i < n; ++i) {
                cin >> b >> l;
                steny.push_back({ b, l });
                maxb = max(maxb, b);
                maxl = max(maxl, l);
        }

        steny.push_back({ maxb, 0 });
        steny.push_back({ 0, maxl });

        // Funkce upper_convex_hull nalezne horní polovinu konvexního obalu
        // a vrátí body setříděné podle jejich polohy
        // na konvexním obalu. Speciálně první a poslední bod
        // budou naše falšené stěny { maxb, 0 } a { 0, maxl }
        vector<pll> obal = upper_convex_hull(steny);
        int m = obal.size();

        // První či poslední stěna na konvexním obalu nabývají hodnot
        // maxb resp. maxl a může se stát, že se vyplatí jít jen na jednu
        // z nich. V ostatních případech stačí ptát se na dvojice sousedních stěn.
        double result = min(jedna_stena(obal[1]), jedna_stena(obal[m - 2]));

        for (int i = 1; i < m-2; ++i)
                result = min(result, dve_steny(obal[i], obal[i+1]));

        return result;

}
```

P-III-3 Hvězdní věštci

Podúloha a)

Zjevně stačí nechat věštce uhodnout systém dálnic a ověřit, že splňuje podmínky (tvoří strom a z každého systému vychází nejvýše tři dálnice). Při ověření, zda systém dálnic tvoří strom, využijeme řešení úlohy b) krajského kola.

Řešení tedy bude následující: Pokud existuje plán dálnic splňující podmínky ze zadání, pro každý systém v něm existuje jednoznačná cesta po dálnicích do systému číslo 0 (budeme jí říkat *cesta do kořene*). V každém hvězdném systému (kromě toho s číslem 0) vyvěštíme délku d jeho cesty do kořene a jeho souseda s na této cestě. Všem svým sousedům pošleme d a zda jsou naším sousedem s. Poté ověříme následující podmínky:

(i) Soused s má cestu do kořene délky $d - 1$. V systému číslo 0 místo toho ověříme, že $d = 0$.
(ii) Nejvýše dva (či nejvýše tři pro systém číslo 0) hvězdné systémy tvrdí, že jsme jejich sousedé na cestě do kořene.

Jestliže program odpoví ANO, dálnice postavíme právě mezi systémy u a v takovými, že u tvrdí, že v je jeho soused na cestě do kořene (nebo naopak). Za každý systém kromě toho s číslem 0 je tak postavena jedna dálnice, celkový počet dálnic je tedy $N - 1$. Tento počet je nejmenší možný, jinak by jistě nebylo možné se po dálnicích dostat mezi nějakými dvěma systémy.

Podmínka (i) zaručuje, že se po dálnici z každého systému dostaneme do jiného systému s hodnotou d o jedna menší, a postupně tedy až do systému s hodnotou d rovnou 0, což může být jedině systém číslo 0. Odevšad se proto dá dostat do systému číslo 0, a tedy i kamkoliv jinam. Dle podmínky (ii) pak z každého systému vedou nejvýše tři dálnice.

Jak d tak s jsou čísla menší než N, potřebujeme tedy vyvěštit $K = 2\lceil \log_2 N \rceil$ bitů. V pseudokódu:

```
struct {
    unsigned(ceil(log(N))) d;
    unsigned(ceil(log(N))) s;
} R;

struct {
    unsigned(ceil(log(N))) d;
    bool na_ceste;
} send[D], receive[D];

if (A == 0 && R.d > 0)
    return NE;

if (A != 0 && (R.d == 0 || R.s >= D))
    return NE;

for (i = 0, ..., D - 1) {
    send[i].d = R.d;
    send[i].na_ceste = (A != 0 && i == R.s);
}

if (A != 0 && receive[R.s].d != R.d - 1)
    return NE;

pocet_na_ceste = 0;
for (i = 0, ..., D - 1)
    if (receive[i].na_ceste)
        pocet_na_ceste++;

mez = (A == 0 ? 3 : 2);
if (pocet_na_ceste > mez)
    return NE;

return ANO;
```

Podúloha b)

Jako *stupeň* systému si označme počet sousedních systémů, tedy jeho hodnotu D. Naším cílem bude sečíst stupně všech systémů; tím je každá dvojice systémů započítaná dvakrát, stačí tedy výsledek porovnat s $2M$. Dále využijeme ideu podobnou té z řešení úlohy a) z krajského kola – každý systém uhodne nějaký částečný součet a ověří, že je konzistentní s částečnými součty jeho sousedů. V úloze z krajského kola jsme to dělali v případě, že Hvězdné impérium tvoří cestu, a přímočará adaptace toho řešení by tedy byla pokusit se uhodnout cestu procházející všemi systémy. Taková cesta ale samozřejmě nemusí existovat. Místo toho si tedy necháme vyvěštit strom T obsahující všechny systémy zakořeněný například v systému číslo 0. Budeme říkat, že systém u je *potomkem* systému v, jestliže cesta v T z u do kořene prochází přes v, a jeho *dítětem*, jestliže navíc u sousedí s v v T. V každém systému si pak stačí nechat vyvěštit součet stupňů jeho potomků a ověřit, že je konzistentní, tedy roven součtu hodnot vyvěštěných v jeho dětech plus D. V kořeni pak ověříme, zda součet je nejvýše $2M$.

Strom T bychom mohli vyvěštit podobně jako v části a), tedy tím, že pro každý systém vyvěštíme jeho souseda s na cestě do kořene a délku d této cesty, na což je potřeba $2\lceil \log_2 N \rceil$ bitů. Toto řešení můžeme ještě trochu vylepšit tím, že necháme vyvěštit takový strom T, pro který je součet délek cest ze všech systémů do kořene největší možný. Strom T pak má následující vlastnost: Každý systém v různý od kořene má v Hvězdném impériu právě jednoho souseda, jehož cesta do kořene má délku $d-1$ (tímto sousedem je právě s). Kdyby měl totiž ještě nějakého jiného takového souseda w, pak uvažme strom, kde by si w jako svého souseda na cestě do kořene místo svého stávajícího vybral v. Tím by se o 2 prodloužila cesta do kořene pro w a všechny jeho potomky, což je ve sporu s volbou T.

Pro reprezentaci stromu T s touto vlastností tedy pro každý systém stačí vyvěštit délku cesty d do kořene; jeho soused na cestě do kořene je pak jeho jediný soused s cestou do kořene délky $d-1$ a jeho děti jsou jeho sousedi s cestou do kořene délky $d+1$. V tomto řešení tedy potřebujeme $\lceil \log_2 N \rceil$ bitů na reprezentaci d a $\lceil \log_2 N^2 \rceil \leq 2\lceil \log_2 N \rceil$ bitů na reprezentaci částečného součtu, dostáváme tedy $K = 3\lceil \log_2 N \rceil$.

V pseudokódu to vypadá následovně:

```
struct {
    unsigned(ceil(log(N))) d;
    unsigned(2 * ceil(log(N))) soucet;
} R;

struct {
    unsigned(ceil(log(N))) d;
    unsigned(2 * ceil(log(N))) soucet;
} send[D], receive[D];

if (A == 0 && (R.d > 0 || R.soucet > 2 * M))
    return NE;

if (A != 0 && R.d == 0)
    return NE;
```

```
for (i = 0, ..., D - 1) {
    send[i].d = R.d;
    send[i].soucet = R.soucet;
}

pocet_minus1 = 0;
suma_deti = 0;
for (i = 0, ..., D - 1) {
    if (receive[i].d + 1 == R.d)
        pocet_minus1++;
    if (receive[i].d == R.d + 1)
        suma_deti += receive[i].soucet;
}

if (R.soucet != suma_deti + D)
    return NE;

if (A != 0 && pocet_minus1 != 1)
    return NE;

return ANO;
```