75. ročník Matematické olympiády – 2025/2026

Úlohy domácího kola kategorie P

Řešení úloh odevzdávejte pomocí webového rozhraní, které je dostupné na stránkách olympiády *https://mo.mff.cuni.cz/*. Tam také najdete podrobnější instrukce k odevzdávání a další informace o kategorii P.

Úlohy P-I-1 a P-I-2 jsou praktické. Vaším úkolem v nich je vytvořit a odladit efektivní program v jednom z následujících programovacích jazyků: C, C++ 17, Python 3.11, C# 11, Java 11 nebo Pascal. Řešení těchto dvou úloh odevzdávejte přes webové rozhraní ve formě zdrojového kódu. Odevzdaná řešení budou automaticky vyhodnocena pomocí připravených vstupních dat a výsledky vyhodnocení se dozvíte krátce po odevzdání. Pokud váš program nezíská plný počet 10 bodů, můžete své řešení opravit a znovu odevzdat.

Vzhledem k různé výkonnosti stejného algoritmu implementovaného v různých programovacích jazycích nezaručujeme, že je ve všech podporovaných jazycích možné získat plný počet bodů – může se stát, že program nestihne doběhnout do časového limitu, i když implementuje algoritmus s optimální časovou složitostí. Časové limity jsou (s rezervou) nastavené dle autorských řešení implementovaných v C++.

Úloha P-I-3 je teoretická, neodevzdává se tedy program, ale písemné řešení obsahující popis algoritmu, který tuto úlohu efektivně vyřeší. Součástí řešení také musí být zdůvodnění správnosti tohoto algoritmu a odhad jeho časové a paměťové složitosti. Do řešení nemusíte psát odpovídající program, algoritmus stačí zapsat ve vhodném pseudokódu nebo dokonce jenom slovně, je-li popis dostatečně podrobný a srozumitelný. Hodnotí se nejen správnost, ale také efektivita zvoleného postupu řešení. Řešení této teoretické úlohy odevzdávejte ve formě souboru typu PDF přes výše uvedené webové rozhraní.

Úloha P-I-4 má teoretickou i praktickou část. K teoretickým podúlohám odevzdejte písemné řešení obsahující ILP, jejich popis a zdůvodnění správnosti. K praktickým podúlohám dostanete k dispozici testovací vstupy a odevzdávají se k nim pouze odpovídající optimální řešení, jejichž hodnoty stačí napsat do vašeho písemného řešení. Řešení této úlohy tedy odevzdávejte také ve formě souboru typu PDF přes webové rozhraní.

V řešení teoretických úloh nemusíte detailně popisovat jednoduché operace jako vstupy, výstupy, implementaci jednoduchých matematických vztahů, vyhledávání v poli, třídění apod. Blíže viz *https://mo.mff.cuni.cz/p/knihovna.html*.

Řešení všech úloh můžete odevzdávat do 15. listopadu 2025. Opravená řešení a seznam postupujících do krajského kola najdete na webových stránkách olympiády o několik týdnů poté.

P-I-1 Semafory

V Absurdistánu je n měst, mezi kterými lze cestovat po m jednosměrných silnicích. Každá silnice má svou vlastní délku, která udává, jak dlouho trvá cesta z výchozího města do cílového města. Není ale možné cestovat libovolně: na začátku každé jednosměrné silnice je *semafor,* který střídá zelenou a červenou barvu. Na silnici i můžeme vjet pouze tehdy, když je na příslušném semaforu zelená. Můžeme však počkat, až semafor naskočí na zelenou, a teprve pak vyrazit. Každý semafor má čtyři parametry:

- $z_i > 0$: jak dlouho na něm svítí zelená.
- $c_i \geq 0$: jak dlouho na něm svítí červená. Jestliže $c_i = 0$, pak na semaforu svítí pouze zelená.
- $s_i \in \{\texttt{Z}, \texttt{C}\}$, $f_i > 0$: parametry udávající počáteční stav semaforu. První z nich udává, zda na začátku svítí zelená ($s_i = \texttt{Z}$), nebo červená ($s_i = \texttt{C}$). Parametr f_i říká, jak dlouho ještě bude tato barva svítit, než se semafor přepne na opačnou barvu. Když na začátku svítí zelená, platí $f_i \leq z_i$; jestliže na začátku svítí červená, pak $f_i \leq c_i$. Pokud $c_i = 0$, semafor nemůže začínat jako červený, nutně tedy platí $s_i = \texttt{Z}$.

Například když $z_i = 5$, $c_i = 2$, $s_i = \texttt{Z}$ a $f_i = 1$, tak se na semaforu střídají světla následovně:

1. Nejdřív zelená svítí ještě f_i jednotek času, tedy od času 0 (včetně) do času $0 + f_i = 1$ (mimo). V čase 0,998 tedy ještě svítí zelená, ale v čase 1 už právě naskočila červená a na silnici nesmíme vjet.
2. Poté svítí červená od času 1 (včetně) do času $1 + c_i = 3$ (mimo).
3. Potom zelená od času 3 (včetně) do času $3 + z_i = 8$ (mimo).
4. Potom červená od času 8 (včetně) do času 10 (mimo).
5. Potom opět zelená od času 10 (včetně) do času 15 (mimo).
6. Potom opět červená od času 15 (včetně) do času 17 (mimo).
7. ...

Z celočíselných časů tedy na silnici smíme vjet v čase 0, potom v časech $3, 4, 5, 6, 7$, dále v časech $10, 11, 12, 13, 14$, atd.

Soutěžní úloha

Máte popis celé silniční sítě v Absurdistánu: počet měst n ($1 \leq n \leq 100\,000$), počet jednosměrných silnic m ($0 \leq m \leq 200\,000$) a pro každou silnici i její délku t_i (přesněji řečeno, jak dlouho trvá cesta po ní, $1 \leq t_i \leq 10^9$) a parametry z_i, c_i, s_i a f_i ($z_i, c_i, f_i \leq 10^9$) semaforu na jejím začátku. Města jsou očíslovaná od 1 do n. V čase 0 se nacházíte ve městě 1 a chcete se co nejrychleji dostat do města n. Zjistěte, jak nejrychleji je to možné.

Vaše řešení by mělo číst vstupní data ze standardního vstupu a výsledky vypisovat na standardní výstup; více detailů naleznete na *https://mo.mff.cuni.cz/p/otazky_a_odpovedi.html*.

Formát vstupu

Na prvním řádku vstupu jsou dvě celá čísla n a m, udávající počet měst a silnic. Na i-tém z následujících m řádků je popis i-té silnice, který se skládá ze sedmi mezerou oddělených údajů (šest přirozených čísel a jeden znak `Z` nebo `C`) v následujícím pořadí:

1. a_i: ze kterého města silnice vede.
2. b_i: do kterého města silnice vede, $a_i \neq b_i$.
3. t_i: jak dlouho trvá cesta po této silnici.
4. z_i, c_i, s_i, f_i: parametry semaforu, jejichž význam je vysvětlen výše.

Mezi dvěma městy může vést i více silnic.

Formát výstupu

Na výstup vypište jediné celé číslo: Nejmenší možný čas, ve kterém lze dorazit do města n, nacházíme-li se v čase 0 ve městě 1. Upozorňujeme, že toto číslo může být velké, dbejte tedy na to, abyste k jeho reprezentaci použili celočíselný typ s dostatečným rozsahem (např. `long long int` v C++). Jestliže se z města 1 do města n nedá dostat, vypište místo toho -1.

Bodování

Ve všech vstupech platí $n \leq 100\,000$, $m \leq 200\,000$ a $t_i, z_i, c_i \leq 10^9$. Vstupy jsou rozdělené do několika testovacích sad s různými dodatečnými omezeními. Body za každou sadu dostanete, jestliže váš program správně vyřeší všechny testovací vstupy v této sadě.

sada	*body*	*omezení*
1	2	všechny silnice i splňují $z_i = 1$, $c_i = 1$ a $t_i \leq 10$
2	1	všechny silnice i splňují $z_i + c_i = 1\,000$, $n \leq 5\,000$ a $t_i = 1$
2	1	všechny silnice i splňují $z_i + c_i \leq 8$, $n \leq 5\,000$ a $t_i = 1$
3	2	na všech semaforech je stále zelená ($c_i = 0$)
4	2	všechny semafory mají stejné parametry z_i, c_i, s_i, f_i a v čase 0 na všech z nich právě naskočila zelená ($s_i = \mathtt{Z}$, $f_i = z_i$)
5	2	žádná další omezení

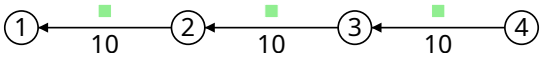
Příklady

Vstup:

```
4 3
4 3 10 1 0 Z 1
3 2 10 1 0 Z 1
2 1 10 1 0 Z 1
```

Výstup:

```
-1
```



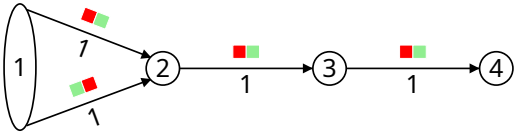
Jelikož jsou silnice jednosměrné, lze se po nich sice dostat z města 4 do města 1, naopak to ale nejde.

Vstup:

```
4 4
1 2 1 1 1 C 1
1 2 1 1 1 Z 1
2 3 1 1 1 C 1
3 4 1 1 1 C 1
```

Výstup:

```
4
```



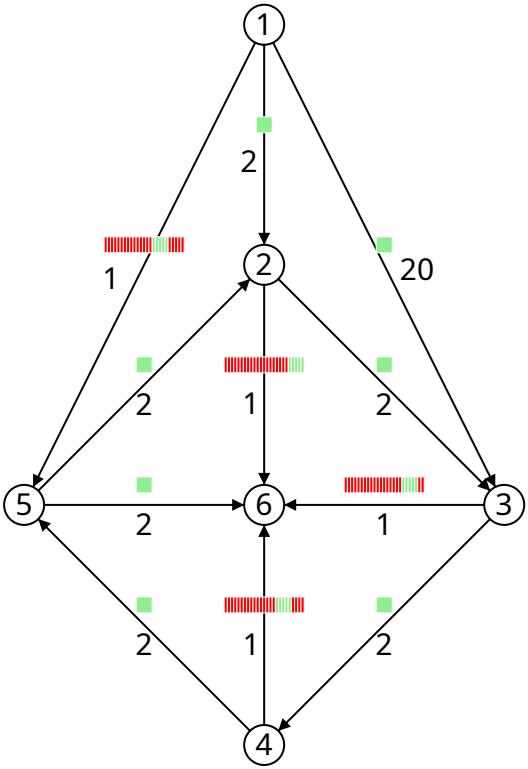
Jediná možnost je jet $1 \to 2 \to 3 \to 4$. Z města 1 lze do 2 vyrazit hned, jelikož na jedné ze dvou silnic $1 \to 2$ máme zelenou. Do města 2 dorazíme v čase 1, a na silnici $2 \to 3$ nám právě v tomto okamžiku naskočí zelená – můžeme tedy opět rovnou pokračovat. Do města 3 dorazíme v čase 2, zde se nám ale semafor právě přepnul na červenou. Musíme tedy počkat jednu jednotku času, než se semafor přepne zpět na zelenou, a až poté můžeme vyrazit. Celkem tedy budeme 3 jednotky času cestovat a 1 čekat.

Vstup:

```
6 11
1 2 2 1 0 Z 1
1 3 20 1 0 Z 1
1 5 1 5 20 C 15
2 3 2 1 0 Z 1
3 4 2 1 0 Z 1
4 5 2 1 0 Z 1
5 2 2 1 0 Z 1
2 6 1 5 20 C 20
3 6 1 5 20 C 18
4 6 1 5 20 C 16
5 6 2 1 0 Z 1
```

Výstup:

```
10
```



Jedna možná trasa s trváním 10 *je* $1 \to 2 \to 3 \to 4 \to 5 \to 6$. *Pokud bychom do města* 6 *chtěli dorazit z jiného města než z* 5, *tak bychom museli na červené čekat alespoň do času* $16 > 10$. *Jako poslední je tedy jistě nejvýhodnější použít silnici* $5 \to 6$. *A jak se nejrychleji dostaneme do města* 5*? Jediné další možnosti kromě výše uvedené trasy jsou* $1 \to 3 \to 4 \to 5$ *(ale silnice* $1 \to 3$ *je dlouhá) nebo přímo* $1 \to 5$ *(tam ale dlouho čekáme na semaforu). Ani jedna z nich tedy není lepší.*

Poslední příklad se od toho předchozího liší pouze tak, že má silnice $1 \to 5$ *parametr* $f_i = 6$ *namísto* $f_i = 15$. *Zde je optimálním řešením počkat ve městě* 1 *do času* 6, *kdy na tomto semaforu naskočí zelená, a po trase* $1 \to 5 \to 6$ *dorazit do cíle v čase* 9.

P-I-2 Rozbitý robot

Na dlouhou rovnou cestu vedoucí ze západu na východ jsme postavili robota. Robotovi jsme zadali posloupnost n instrukcí, které mu říkají, jak se má pohybovat. Každý příkaz se skládá ze směru (`V` pro východ nebo `Z` pro západ) a vzdálenosti (celé číslo v metrech).

Robot je však starý a trochu rozbitý, a proto se choval trochu jinak, než jsme chtěli. Přesněji řečeno došlo k následujícím odchylkám od zadaného programu:

- Robot neměl dostatek energie, takže neprovedl celý program, ale pouze některých p *po sobě jdoucích* příkazů.
- Jeho senzory nejsou moc spolehlivé, některé příkazy proto robot provedl přesně opačným způsobem – to znamená, že ujel správnou vzdálenost, ale v opačném směru, než jsme chtěli. Během provádění programu došlo k nejvýše k takovým chybám.

Soutěžní úloha

Napište program, který z daného seznamu instrukcí pro robota a z čísel p a k vypočítá, jak daleko od své *výchozí* polohy mohl robot nejvýše skončit.

Vaše řešení by mělo číst vstupní data ze standardního vstupu a výsledky vypisovat na standardní výstup; více detailů naleznete na *https://mo.mff.cuni.cz/p/otazky_a_odpovedi.html*.

Formát vstupu

Na první řádce vstupu jsou přirozená čísla n, p a k ($0 \leq k \leq p \leq n \leq 200\,000$) udávající počet instrukcí, počet vykonaných instrukcí, a nejvyšší možný počet chyb během jejich vykonávání. Dalších n řádek vstupu popisuje instrukce pro robota; na i-té z nich je znak $s_i \in \{\texttt{V}, \texttt{Z}\}$ udávající směr a přirozené číslo d_i ($1 \leq d_i \leq 10^9$) udávající vzdálenost pohybu robota.

Formát výstupu

Vypište jedno přirozené číslo: Největší možnou vzdálenost mezi pozicemi robota na začátku a konci vykonávání zadaných instrukcí. Upozorňujeme, že toto číslo může být velké, dbejte tedy na to, abyste k jeho reprezentaci použili celočíselný typ s dostatečným rozsahem (např. `long long int` v C++).

Bodování

Pro všechny vstupy platí $0 \leq k \leq p \leq n \leq 200\,000$ a pro každé i platí, že s_i je `V` a `Z` a že $1 \leq d_i \leq 10^9$.

Vstupy jsou rozdělené do několika testovacích sad s různými dodatečnými omezeními. Body za každou sadu dostanete, jestliže váš program správně vyřeší všechny testovací vstupy v této sadě.

sada	*bodů*	*omezení*
1	2	$k = 0$, robot tedy nedělá chyby
2	2	$n \leq 1\,000$ a $p \leq 10$
3	2	$n \leq 1\,000$
4	2	$n \leq 70\,000$
5	2	žádná další omezení

Příklady

Vstup:
```
6 4 0
Z 30
V 10
V 25
Z 5
V 10
Z 30
```

Výstup:
```
40
```

Tento robot nedělá chyby. Jestliže vykonal první čtyři instrukce, skončí tam, kde začal. Vykonal-li druhou až pátou instrukci, skončil o 40 m na východ od místa, kde začal.

Vstup:
```
6 4 2
V 30
Z 10
Z 15
Z 5
Z 10
V 30
```

Výstup:
```
60
```

Jedno optimální řešení je, že robot vykonal první čtyři instrukce, ale v první z nich se spletl a jel na západ.

Vstup:
```
6 6 6
Z 1
V 2
V 3
Z 4
V 5
Z 6
```

Výstup:
```
21
```

P-I-3 Sušenky

Emička má $n \leq 10^5$ polomáčených sušenek. Všechny je položila na stůl do dlouhé řady, některé z nich otočila čokoládovou stranou nahoru a zbytek čokoládovou stranou dolů. Emička by samozřejmě ráda snědla všechny sušenky. Aby to ale bylo zajímavější, vymyslela si následující pravidla: Může jíst pouze sušenky, které jsou otočené čokoládovou stranou nahoru. A aby měly i ostatní sušenky šanci být snědeny, pokaždé, když sní sušenku, otočí její bezprostřední sousedy vzhůru nohama. Sousedními sušenkami myslíme sušenky, které ležely vedle této sušenky na začátku. Pokud tedy dvě sušenky na začátku nejsou sousední, *nestanou* se sousedními, ani když Emička sní všechny sušenky ležící mezi nimi. Emička by ráda věděla, zda a jak může sníst všechny sušenky.

Představme si například, že Emička začala se třemi sušenkami, z nichž pouze ta prostřední byla otočená čokoládovou stranou nahoru. V tomto případě nemá na výběr: musí nejprve sníst prostřední sušenku. Když tak učiní, otočí obě sousední sušenky, ty teď tedy leží čokoládou stranou nahoru. Poté je může sníst jednu po druhé (v libovolném pořadí). Jelikož tyto dvě sušenky nejsou sousední, Emička po snědení první z nich druhou neotáčí. Tato posloupnost snědení je znázorněna na následujícím obrázku (shora dolů):

Soutěžní úloha

Vymyslete pro Emičku algoritmus, který pro zadanou posloupnost sušenek rozhodne, zda a jak je může všechny sníst. Úloha má několik podúloh, které můžete řešit zcela samostatně a získat body za každou z nich zvlášť. Při hodnocení řešení budeme klást zvláštní důraz na *zdůvodnění jejich správnosti!*

Pro účely tohoto úkolu označíme sušenky s čokoládovou stranou nahoru jako `C` a sušenky s čokoládovou stranou dolů jako `P`.

a) (1 bod) Představte si, že sušenky jsou zpočátku uspořádány v pořadí `CCCCCCC`, tj. 7 sušenek s čokoládovou stranou nahoru. Může je Emička sníst všechny? Pokud ano, ukažte jak. Pokud ne, vysvětlete proč.

b) (1 bod) Představte si, že sušenky jsou zpočátku uspořádány v pořadí `CPPPPPC`. Může je Emička sníst všechny? Pokud ano, ukažte jak. Pokud ne, vysvětlete proč.

c) (5 bodů) Navrhněte algoritmus, který jako vstup přijme řetězec písmen `C` a `P` představující sušenky v řadě a rozhodne, zda existuje nějaké pořadí, ve kterém je Emička dokáže sníst všechny (při dodržení výše uvedených pravidel). Nezapomeňte na zdůvodnění správnosti vašeho algoritmu (tj.

že když váš algoritmus odpoví ano, pak takové pořadí skutečně existuje, a naopak pokud odpoví ne, tak pro libovolné pořadí jedení nakonec zbude alespoň jedna sušenka, kterou Emička sníst nemůže).

Za vyřešení této podúlohy pouze pro zvláštní případ, kdy jsou všechny sušenky na začátku otočeny čokoládovou stranou nahoru (řetězec se skládá pouze z písmen `C`), můžete získat až 2 body.

d) (3 body) Zatímco Emička nedávala pozor, kočka snědla část sušenek. Nyní můžeme popsat situaci na stole pomocí znakového řetězce `C`, `P` a `-` (prázdné místo po sušence). Sušenku x na stole nazýváme *pěknou,* pokud ji Emička může sníst jako první; tedy, že existuje postup, kterým Emička může sníst všechny zbývající sušenky, přičemž sušenku x sní jako první.

Navrhněte algoritmus, který spočítá, kolik je na stole pěkných sušenek. Pokud všechny sušenky na stole Emička žádným způsobem sníst nemůže, odpověď je samozřejmě 0.

Příklad

Vstup:	*Výstup:*
`7`	`3`
`CCC-CP`	

Na stole jsou tři sušenky čokoládovou stranou nahoru, pak mezera a pak další dvě sušenky, první čokoládovou stranou nahoru a ta druhá dolů. Emička je může sníst všechny. Aby toho dosáhla, musí začít s první, třetí nebo pátou sušenkou (v původním pořadí). Všimněte si, že i když je druhá sušenka čokoládovou stranou nahoru, pokud ji Emička sní jako první, nebude schopna dojíst ostatní sušenky.

Vstup:	*Výstup:*
`3`	`0`
`C-P`	

Sníst obě sušenky není možné – snědení první neobrátí tu druhou.

P-I-4 Řešič to vyřeší

K této úloze se vztahuje studijní text uvedený na následujících stranách. Doporučujeme vám nejprve si ho přečíst a až potom se vrátit k samotným soutěžním úkolům. Ze stejného studijního textu budou vycházet i úlohy v dalších kolech soutěže.

Za každou správně vyřešenou podúlohu získáte jeden bod. Řešení teoretických podúloh můžete odevzdat ve dvou formách:

- *Buď jako slovní popis toho, jak z konkrétních vstupních hodnot vytvořit ILP, jehož optimální řešení odpovídá optimálnímu řešení výše definovaného optimalizačního problému.*
- *Nebo jako komentovaný program v běžném programovacím jazyce, který čte konkrétní vstupní hodnoty a vytváří (např. vypisuje na výstup) odpovídající ILP.*

V obou případech nezapomeňte na zdůvodnění správnosti.

Testovací vstupy k praktickým podúlohám jsou k dispozici na adrese https://mo.mff.cuni.cz/p/75/p1-4-vstupy.zip. Pro tyto podúlohy do svého řešení napište pouze hodnotu výsledku pro zadaný vstup, nemusíte k nim psát žádný další popis. Body za tyto podúlohy získáte za správně vyřešené vstupy bez ohledu na techniku, kterou k tomu použijete (tedy i když ve svých výpočtech vůbec nepoužijete ILP).

Podúlohy A–E: investice

Máme k dispozici e eur. Existuje n projektů (očíslovaných od 1 do n), do kterých můžeme investovat, každý z nich buď podpoříme, nebo nepodpoříme. U každého projektu známe částku s_i potřebnou k jeho podpoře a hodnotu v_i, kterou získáme zpět jako výnos, pokud jej podpoříme.

Mezi projekty existují konflikty: některé dvojice projektů si přímo konkurují a antimonopolní úřad vydal nařízení, že můžeme podpořit maximálně jeden projekt z každé takové dvojice. Existuje k konfliktů. U každého z nich známe čísla $c_{i,1}$ a $c_{i,2}$ projektů, které nemůžeme oba zároveň podpořit.

Mezi projekty existují také závislosti: například projekt pěstování bio zelí nelze realizovat, pokud není podpořen projekt instalace efektivního zavlažovacího systému, takže pokud chceme podpořit zelí, musíme podpořit také zavlažování. Existuje z závislostí. Pro každou závislost máme zadány čísla dvou projektů: pokud chceme podpořit projekt číslo $d_{i,1}$, musíme podpořit také projekt číslo $d_{i,2}$. Jinými slovy, tato závislost nám brání podpořit $d_{i,1}$ a zároveň nepodpořit $d_{i,2}$. Závislosti mohou být zcela libovolné. Zejména pokud máme dvě závislosti, které říkají, že x závisí na y a že y závisí na x, znamená to, že můžeme podporovat buď oba tyto projekty zároveň, nebo žádný z nich.

V praktických podúlohách jsou informace o projektech reprezentovány v textovém souboru následujícím způsobem:

- První řádek: přirozená čísla e a n udávající náš kapitál a počet projektů.
- Druhý řádek: přirozená čísla $s_1, \ldots, s_n$ udávající částky potřebné na podporu jednotlivých projektů.

- Třetí řádek: přirozená čísla $v_1, \ldots, v_n$ udávající výnosy s projektů, podpoříme-li je.
- Čtvrtý řádek: přirozené číslo k udávající počet konfliktů.
- Následujících k řádků: Na i-tém z nich jsou čísla projektů $c_{i,1}$ a $c_{i,2}$ popisující i-tý konflikt.
- Následující řádek: přirozené číslo z udávající počet závislostí.
- Následujících z řádků: Na i-tém z nich jsou čísla projektů $d_{i,1}$ a $d_{i,2}$ popisující i-tou závislost.

A) Předpokládejme, že neexistují žádné konflikty ani závislosti. Sestavte ILP, jehož optimální řešení nám řekne, které projekty podpořit, pokud chceme mít na konci (když obdržíme výnosy) co nejvíce peněz.

B) Upravte ILP z předcházející podúlohy tak, aby zohledňoval konflikty mezi projekty.

C) Upravte ILP z podúlohy A) nebo B) tak, aby zohledňoval závislosti mezi projekty.

D) Soubor `D.txt` obsahuje testovací vstup ve výše popsaném formátu. V tomto testovacím vstupu platí $k = z = 0$, tedy mezi projekty neexistují žádné konflikty ani závislosti. Najděte optimální sadu projektů, které máme podpořit. Jako řešení této podúlohy napište celkovou částku peněz, kterou budeme mít na konci, a čísla podporovaných projektů.

E) Najděte a ve stejném formátu odevzdejte optimální řešení pro soubor `E.txt`. V tomto testovacím vstupu je k i z nenulové.

Příklad

Vstup:

```
100000 4
50000 50000 20000 40000
100000 95000 26000 900000
1
1 2
2
4 3
3 2
```

Výstup:

```
Optimální řešení: 151000
Podpoř projekty: 2 3
```

Projekt 4 nemůžeme podpořit, jelikož kvůli závislostem bychom museli podpořit i projekty 2 a 3 a na to nemáme dost peněz. Podpořit projekt 1 se nám nevyplatí, jelikož pak bychom kvůli konfliktu nemohli podpořit projekt 2 a následně kvůli závislosti ani projekt 3.

Podúlohy F-J: Chovatelé prasat

V našem kraji je s sýpek a c chovatelů prasat; sýpky jsou očíslovány od 1 do s a chovatelé od 1 do c.

Na podzim můžeme ve sýpkách uskladnit bukvice a žaludy. Sýpka i má kapacitu k_i kg ovoce a můžeme v ní skladovat libovolný poměr bukvic a žaludů. Oba druhy

plodů lze sbírat v libovolném množství v okolních lesích. Sklizeň nás stojí 5000 eur za brigádníky, bez ohledu na to, kolik čeho a do kterých sýpek jim dáme nasbírat.

V zimě budou chovatelé chtít od nás koupit krmivo. Chovatel j je ochoten koupit nanejvýš $g_{j,1}$ kg bukvic a nanejvýš $g_{j,2}$ kg žaludů a je ochoten zaplatit $h_{j,1}$ za každý celý kilogram bukvic a $h_{j,2}$ eur za každý celý kilogram žaludů. Všechny tyto objednávky už teď známe.

Objednané plody zašleme kurýrem. Kurýr je ochoten pro každého chovatele z každého skladu odvézt jednu zásilku o hmotnosti nejvýše w kg. Do zásilky můžeme zabalit libovolný počet kg bukvic a libovolný počet kg žaludů, pokud jejich celková hmotnost nepřekročí limit. Cena zásilky závisí na odesílateli a příjemci, ale vždy je přímo úměrná hmotnosti zásilky: za každý kilogram plodů odeslaných ze sýpky i chovateli j zaplatíme $\ell_{i,j}$ eur.

Všechny výše uvedené číselné údaje jsou kladná celá čísla. V praktických podúlohách popsanou situaci reprezentujeme v textovém souboru následujícím způsobem:

- První řádek: čísla s a c udávající počet sýpek a chovatelů.
- Druhý řádek: čísla $k_1, \ldots, k_s$ udávající kapacity sýpek.
- Následujících c řádků: Na j-tém z nich jsou čísla $g_{j,1}$, $g_{j,2}$, $h_{j,1}$ a $h_{j,2}$ popisující j-tého chovatele.
- Následující řádek: číslo w udávající nejvyšší možnou hmotnost každé zásilky
- Následujících s řádků: Na i-tém z nich jsou čísla $\ell_{i,1}, \ldots, \ell_{i,c}$ udávající ceny přepravy z i-té sýpky k chovatelům

V jednotlivých podúlohách platí:

F) Předpokládejme, že všichni chovatelé chtějí kupovat pouze bukvice (tj. všechny $g_{j,2} = 0$). Sestavte ILP, jehož optimální řešení bude odpovídat největšímu možnému zisku, kterého můžeme dosáhnout prostřednictvím optimálního sběru, distribuce a prodeje krmiva.

G) Upravte ILP z řešení podúlohy F) tak, aby optimálně řešilo celý obecný problém.

H) Najděte a odešlete optimální řešení pro testovací vstup v souboru `H.txt`. V testovacím vstupu je pouze jedna sýpka ($s = 1$). Do řešení stačí napsat nejlepší možný zisk, není třeba zapisovat, kolik čeho se má přepravit z jakého místa do jakého místa.

I) Najděte a odevzdejte optimální řešení pro vstup v souboru `I.txt`. V tomto testovacím vstupu nikdo nechce kupovat žaludy (všechna $g_{j,2} = 0$).

J) Najděte a odešlete optimální řešení pro testovací vstup v souboru `J.txt`. V tomto testovacím vstupu neplatí omezení z podúloh H) a J)

Příklad

Vstup:

```
2 3
100 200
1000 1000 10 10
120 70 100 30
20 30 50 60
80
9 15 23
11 28 12
```

Výstup:

```
Optimální zisk: 6980
sýpka 1: bukvice 60 žaludy 40
sýpka 2: bukvice 100 žaludy 30
zásilka s1->ch1: bukvice 20
zásilka s1->ch2: bukvice 40 žaludy 40
zásilka s2->ch2: bukvice 80
zásilka s2->ch3: bukvice 20 žaludy 30
```

Obsah sýpek a zásilek ve svém řešení specifikovat nemusíte.

Výdaje:

- 5000 *za sklizeň.*
- $20 \times 9 + (40 + 40) \times 15 = 1380$ *za dopravu z první sýpky.*
- $80 \times 28 + (20 + 30) \times 12 = 2840$ *za dopravu z druhé sýpky.*

$\Rightarrow$ *Celkem* 9220 *eur.*

Příjmy:

- *Chovatel 1:* $20 \times 10 = 200$.
- *Chovatel 2:* $(40 + 80) \times 100 + 40 \times 30 = 13\,200$.
- *Chovatel 3:* $20 \times 50 + 30 \times 60 = 2800$.

$\Rightarrow$ *Celkem* 16 200 *eur.*

Studijní text: Celočíselné lineární programování

V letošním ročníku olympiády se budeme zabývat optimalizačními problémy, tedy problémy, které mají mnoho různých řešení, a naším úkolem je najít to nejlepší. Například nás může zajímat:

- Jak nejlevněji navštívit všechna města na Slovensku?
- Kolik krabic potřebuji k zabalení všech svých knih při stěhování?
- Jaká je velikost největší podmnožiny řešitelů letošní MO-P, ve které se všichni navzájem znají?

Mnoho optimalizačních problémů má jednu společnou nepříjemnou vlastnost: neznáme pro ně žádné efektivní algoritmické řešení. Empiricky si dovolíme tvrdit, že do této smutné kategorie spadá drtivá většina optimalizačních problémů, s nimiž se setkáváme všude v praxi – ať už v počítačích (např. plánování procesů, směrování paketů v sítích) nebo v reálném životě (např. logistika všeho druhu, optimalizace nákladů nebo různé problémy v bioinformatice). Mimochodem, všechny tři výše uvedené problémy sem také patří.

Situace je ještě horší. Nejenže neznáme žádný algoritmus, který by tyto problémy dokázal efektivně vyřešit (tj. s polynomiální časovou složitostí vzhledem k velikosti vstupu), ale máme dokonce velmi dobré důvody se domnívat, že žádný takový algoritmus neexistuje. To souvisí s jednou z nejdůležitějších otevřených otázek současné informatiky: otázkou, zda P se rovná NP. Zjednodušeně řečeno, jde o otázku, zda každou úlohu, u které můžeme efektivně zkontrolovat správnost řešení, lze také efektivně vyřešit. Intuitivně většina vědců věří, že je to nepravděpodobné – porovnejte například, jak obtížné může být ruční vyřešení i jednoduché sudoku a jak snadné je zkontrolovat, zda bylo sudoku vyřešeno správně. Tento příklad nám také ukazuje, že znalost toho, jak zkontrolovat správnost řešení, nám obecně nic neříká o tom, jak efektivně hledat řešení.

V praxi je to ale vlastně celkem jedno. Mezi situacemi, kdy pro náš obtížný úkol neexistuje žádný efektivní algoritmus a kdy existuje, ale žádný neznáme, není z praktického hlediska velký rozdíl. Pokud potřebujeme optimálně vyřešit zadání, jsme v obou případech závislí na hrubé síle, tj. na vyzkoušení všech možností.

Ne všechna řešení založená na hrubé síle jsou však stejně dobrá. Často můžeme taková řešení zefektivnit tím, že neprohledáváme všechny možnosti, ale chytře přeskočíme co nejvíce částí vyhledávání, o kterých víme, že nevedou k nejlepšímu řešení. Pro mnoho optimalizačních problémů jsme vyvinuli specifické algoritmy, které nejsou efektivní (jejich časová je stále exponenciální vzhledem k velikosti vstupu), ale díky vhodnému „ořezání“ vyhledávání mohou vyřešit mnohem větší vstupy v rozumném čase než přímé řešení, které zkouší možnosti úplně všechny.

Někteří chytří lidé však o tom přemýšleli a uvědomili si: v mnoha z těchto jednotlivých algoritmů provádíme velmi podobně vypadající optimalizace. Mohli bychom to nějak zobecnit? V letošním ročníku olympiádě se budeme zabývat jednou z kladných odpovědí na tuto otázku.

*Celočíselné lineární programování** je způsob matematického popisu určitých optimalizačních problémů. Jeho výhodou je, že někdo již za nás odvedl veškerou opravdu náročnou práci – v současné době existuje několik velmi dobře optimalizovaných *řešičů,* které dokážou najít optimální řešení úlohy zadané takovým matematickým popisem. Navíc díky mnoha optimalizacím tyto řešiče často fungují efektivněji, než kdybychom sami psali a vylepšovali specializovaný algoritmus pro náš konkrétní úkol. To nám dává nový způsob řešení obtížných problémů: místo implementace vlastního řešení můžeme přemýšlet o tom, zda a jak můžeme tento problém zapsat jako ILP. Pokud se nám to podaří, můžeme k řešení našeho problému použít řešič ILP. A přesně to budete dělat při řešení soutěžních úloh v letošním ročníku olympiádě.

Formální definice ILP

V dalším textu bude slovo *konstanta* označovat jakékoli konkrétní (případně i záporné) *celé* číslo a slovo *proměnná* bude označovat neznámou, která může nabývat jakékoli *nezáporné celé* hodnoty.

Celočíselný lineární program (ve své základní, tzv. kanonické formě) se skládá z následujících částí:

- *Omezení:* Sada lineárních nerovností, každá ve tvaru

  $$a_{i,1} \cdot x_1 + \cdots + a_{i,n} \cdot x_n \leq b_i,$$

  kde všechny $a_{i,j}$ a b_i jsou konstanty. Řešení musí *všechna* tato omezení splňovat.

- *Cíl:* Lineární výraz ve tvaru

  $$c_1 \cdot x_1 + \cdots + c_n \cdot x_n,$$

  kde c_i jsou konstanty a x_i jsou proměnné. Hodnotu tohoto výrazu chceme maximalizovat.

Jakékoli přiřazení hodnot proměnným, pro které jsou splněna všechna omezení, se nazývá *platným* řešením. Platná řešení, pro která má cílový výraz největší možnou hodnotu, se nazývají *optimální*.

Samozřejmě existují také ILP, které nemají optimální řešení. Může to mít dvě příčiny: buď jsou *nesplnitelné* (např. máme omezení $x_1 \leq 7$ a $-x_1 \leq -8$, čili $x_1 \geq 8$) nebo jsou *neomezené* (např. nemáme žádná omezení a chceme maximalizovat hodnotu $x_1 + 2x_2$).

* Pro tuto techniku jako celek i pro jednotlivé celočíselné lineární programy budeme v následujícím textu používat zkratku ILP, tedy Integer Linear Programming.

Flexibilnější a praktičtější definice ILP

Abych se nám s formalismem ILP pracovalo příjemněji, dovolíme trochu obecnější tvar programů:

- Povolíme také programy, jejichž cílem je minimalizovat hodnotu konkrétního výrazu, přičemž tento výraz může obsahovat také konstantní sčítanec.
- Povolíme také omezení, ve kterých je znaménko $\leq$ nahrazeno znaménkem $\geq$ nebo $=$.
- V podmínkách můžeme provádět všechny standardní aritmetické úpravy, např. vynechat sčítance ve tvaru $0 \cdot x_i$, libovolně přesouvat sčítance mezi levou a pravou stranou a používat závorky podle potřeby.

Rozmyslete si, že všechny tyto změny slouží pouze k lepší čitelnosti našich programů: například minimalizace $x + 3y + 1000$ je to samé jako jako maximalizace $-x - 3y$, podmínka $2x - 6y \geq y - 13$ je pouze jiný způsob zápisu podmínky $-2x + 7y \leq 13$ a podmínka $2x = 5y$ je stejná jako dvě podmínky $2x \leq 5y$ a $2x \geq 5y$.

Příklad: Kuřecí nugety

Stánek prodává tři různé balení kuřecích nugetů: 6 kusů za 2 eura, 9 kusů za 2,90 nebo 20 kusů za 6,10. Kolik nejvíc nugetů můžeme koupit za 32 eur?

Nesprávné hladové řešení: Když spočítáme, kolik zaplatíme za jeden nuget v každém balení, nejlepší možností je to největší. Za 32 eur můžeme koupit 5 největších balení, což nám dá 100 nugetů. To však není optimální řešení – všimněte si, že s tímto řešením nám kromě 100 nugetů zbude 1,50 eur, za které si nemůžeme koupit nic jiného. Existuje jiný způsob, jak lépe využít peníze, které máme, a získat více nugetů!

Tento úkol nelze obecně řešit hladově. Náš příklad s nugety je zvláštním případem dobře známého typu optimalizačního problému, který je obecně známý pod názvem *problém batohu.* Pro malé vstupy můžeme najít optimální řešení pomocí dynamického programování, ale obecně je řešení tohoto problému obtížné.

Lineární program: Označme x_1 jako počet malých balení, x_2 jako počet středních balení a x_3 jako počet velkých balení, které zakoupíme. Naším cílem je maximalizovat celkový počet nuget, které zakoupíme, tedy hodnotu $6x_1 + 9x_2 + 20x_3$. Musíme dodržet omezení, že celková kupní cena nesmí překročit náš rozpočet – to znamená, že (v centech, aby všechna čísla byla celá) musí platit následující: $200x_1 + 290x_2 + 610x_3 \leq 3200$.

Praktické řešení: Náš lineární program je zapsán v syntaxi, kterou rozumí řešič `lp_solve`, takto:

```
max: 6x_1 + 9x_2 + 20x_3;
200x_1 + 290x_2 + 610x_3 <= 3200;
int x_1, x_2, x_3;
```

Když požádáme `lp_solve` o řešení tohoto programu, dostaneme následující výstup:

```
Value of objective function: 102.00000000

Actual values of the variables:
x_1                           1
x_2                           4
x_3                           3
```

Zjistili jsme, že můžeme získat až 102 nugetů, když koupíme 1 malé, 4 střední a 3 velká balení. Celková cena nákupu je 31,90, takže na konci budeme mít 102 nugetů a 10 centů nazbyt.

Výběr řešiče

Pro tento studijní text jsme vybrali jeden konkrétní řešič: `lp_solve`. V řešeních příkladů používáme syntaxi, kterou tento řešič rozumí.

Na adrese *https://oi.sk/apps/ilp/* najdete několik různých návodů, který řešič zvolit a jak jej použít k řešení ILP problémů v závislosti na vašem preferovaném operačním systému a programovacím jazyce. Pro domácí kolo si také na internetu můžete najít libovolný jiný řešič a použít ten, pokud se vám náš výběr nelíbí.

Příklad: Sudoku

Někdy místo optimalizace (hledání *nejlepšího* řešení z mnoha) nás může zajímat pouze nalezení jakéhokoli platného řešení nebo rozhodnutí, zda vůbec nějaké platné řešení existuje. Samozřejmě můžeme i k řešení takových problémů použít také řešič ILP: stačí mu nedat žádný cíl (nebo mu například dát cíl maximalizovat hodnotu výrazu „0“).

Podívejme se na známý logický problém: Sudoku. V tomto problému je cílem vyplnit tabulku 9×9 čísly od 1 do 9 tak, aby každý řádek, sloupec a „velký“ čtverec 3×3 obsahoval každé číslo od 1 do 9 právě jednou.

V tomto příkladu ukážeme, jak můžeme formulovat pravidla sudoku jako ILP. Zdálo by se, že bychom potřebovali 81 proměnných: pro každý čtverec tabulky jednu proměnnou reprezentující hodnotu, která by v něm měla být. A ano, to je jeden způsob, jak formulovat sudoku jako ILP, ale to necháme na později. V tomto příkladu použijeme jiný přístup: použijeme $9 \times 9 \times 9$ booleovských (tj. logických nebo binárních) proměnných. Proměnná $x_{i,j,k}$ bude 1, pokud má být hodnota k na souřadnicích (i, j), nebo 0, pokud tam hodnota k být nemá.

Podívejme se nyní, jak by mohly vypadat všechny pravidla sudoku, pokud by byly zapsány jako lineární rovnice a nerovnice.

- V každé buňce je přesně jedno číslo. Pro každé i a j tedy platí podmínka
$$x_{i,j,1} + x_{i,j,2} + \cdots + x_{i,j,9} = 1.$$
- Každé číslo se v každém řádku objevuje přesně jednou. Pro každé i a k tedy platí podmínka
$$x_{i,1,k} + x_{i,2,k} + \cdots + x_{i,9,k} = 1.$$

- Pro každý sloupec a každý čtverec platí analogické podmínky jako pro řádky.

Pokud nyní chceme vyřešit konkrétní sudoku pomocí `lp_solve`, postupujeme následovně:

- Vygenerujeme (např. pomocí jednoduchého programu napsaného v běžném programovacím jazyce) všechny výše uvedené podmínky představující obecná pravidla sudoku.
- Přidáme informaci, že všechny $x_{i,j,k}$ jsou booleovské proměnné. Toho dosáhneme přidáním podmínky $x_{i,j,k} \leq 1$ ke každé z nich. Proměnné, které mohou nabývat pouze hodnot 0 a 1, jsou však v modelování problémů tak běžné, že pravděpodobně každý řešitel bude mít speciální syntaxi pro přímé deklarování takových proměnných. Například v `lp_solve` stačí deklarovat takové proměnné jako `bin` místo `int`.
- Přidáme podmínky popisující konkrétní úkol, který se snažíme vyřešit. Například pokud máme číslo 7 již specifikované v prvním řádku a třetím sloupci úkolu, přidáme podmínku $x_{1,3,7} = 1$.

Příklad: Sudoku podruhé

Jak by vypadalo modelování sudoku, kdybychom chtěli použít proměnnou $v_{i,j}$ pro každou buňku, jejíž hodnota by přímo odpovídala hodnotě nalezené v příslušné buňce? Je zřejmé, že potřebujeme podmínky $v_{i,j} \geq 1$ a $v_{i,j} \leq 9$. Kromě těchto podmínek by stačilo přidat podmínky, které stanoví, že některé páry buněk nesmí mít stejnou hodnotu. Budeme potřebovat poměrně dost takových podmínek: jednu pro každý pár buněk ve stejném řádku, ve stejném sloupci a ve stejném čtverci 3×3. Například pro dvě pole (i, x) a (i, y) v řádku i potřebujeme podmínku $v_{i,x} \neq v_{i,y}$. Zde však narážíme na problém: tato podmínka nemá žádnou z povolených forem a nemůžeme ji přímo vyjádřit pomocí povolených podmínek.

Požadovanou podmínku můžeme zapsat jako logické OR dvou podmínek: musí platit buď $v_{i,x} < x_{i,y}$, nebo $v_{i,x} > v_{i,y}$. Jelikož všechna $v_{i,j}$ jsou celá čísla, můžeme tyto podmínky upravit do povolené formy: musí platit buď $v_{i,x} \leq v_{i,y} - 1$, nebo $v_{i,x} \geq v_{i,y} + 1$.

To však stále není v pořádku: v ILP musí být všechny podmínky splněny současně. To odpovídá logickému AND, nikoli logickému OR. Co s tím můžeme dělat?

Můžeme použít malý trik. Zavedeme novou binární proměnnou r (správně bychom ji měli nazvat například $r_{i,x,i,y}$, protože budeme potřebovat jednu novou proměnnou pro každou dvojici proměnných, které by neměly být stejné, pro lepší čitelnost ale indexy vynechme). Hodnota r nám řekne, zda by měla být menší první nebo druhá z hodnot v. Zvažme nyní následující dvě podmínky:

$$v_{i,x} - v_{i,y} \geq 1 - 10r$$
$$v_{i,y} - v_{i,x} \geq 1 - 10(1 - r) = 10r - 9$$

Pokud $r = 0$, dostaneme podmínky $v_{i,x} - v_{i,y} \geq 1$ a $v_{i,y} - v_{i,x} \geq -9$. První z nich říká, že $v_{i,x} > v_{i,y}$ a druhá je triviálně splněna pro libovolné $v_{i,x}, v_{i,y} \in \{1, 2, \ldots, 9\}$

(konstantu 10 jsme zvolili tak, aby toto platilo, využíváme tedy skutečnosti, že známe rozsah hodnot, kterých mohou $v_{i,x}$ a $v_{i,y}$ nabývat).

Naopak, pokud zvolíme $r = 1$, dostaneme jednu triviálně splněnou podmínku a jednu, která říká, že $v_{i,x} < v_{i,y}$. Přidáním této nové proměnné r a dvou výše uvedených podmínek jsme dosáhli toho, co jsme chtěli: pro libovolné $v_{i,x} \neq v_{i,y}$ můžeme splnit obě tyto podmínky, zatímco pro $v_{i,x} = v_{i,y}$ není možné splnit obě najednou.

Příklad: Co ve formalismu ILP nevyjádříme

Máme letadlo, se kterým chceme letět 1000 km z jednoho letiště na druhé. V rámci povolených rozsahů odpovídajících modelu letadla můžeme zvolit letovou hladinu h (výšku v km, ve které budeme létat, v rozmezí 10 až 13 km) a letovou rychlost v (v km/h, v rozmezí 600 až 900 km/h). Chtěli bychom minimalizovat náklady na let, tj. spotřebu paliva.

Toto lze zapsat pomocí vhodných vzorců jako optimalizační problém. Ve velmi zjednodušené podobě by to mohlo vypadat nějak takto: Doba letu bude $1000/v$. Pokud letíme ve výšce h, optimální rychlost vzhledem k odporu vzduchu je $v_{opt}(h) = 540 + 30h$. Výkon motoru je nejlepší ve výšce $h = 11{,}5$ km. Odchylka od těchto parametrů zvyšuje spotřebu paliva, ale může nás dostat k cíli rychleji. Spotřeba paliva (v kg/h) lze proto vyjádřit rovnicí $2000 + 200(h - 11{,}5)^2 + 0{,}05(v - v_{opt}(h))^2$. Celková spotřeba paliva je součinem této hodnoty a doby letu.

Ačkoli se jedná o přesné matematické vyjádření optimalizačního problému, je zde jeden háček: omezení pro h a v jsou lineární, ale funkce, jejíž hodnotu se snažíme optimalizovat, není lineární funkcí proměnných h a v. Řešič ILP nám proto s takto konkrétně zformulovaným problémem nepomůže.

Pro názornost dodejme, že ani mnohem jednodušší výraz $h \cdot v$ není lineární, protože je to součin dvou proměnných.