75. ročník Matematické olympiády – 2025/2026

Úlohy krajského kola kategorie P

Krajské kolo 75. ročníku MO kategorie P se koná v úterý 20. 1. 2026 v dopoledních hodinách. Na řešení úloh máte 4 hodiny čistého času. V krajském kole MO-P se neřeší žádná praktická úloha, pro zajištění rovných podmínek řešitelů ve všech krajích je použití počítačů při soutěži zakázáno (s výjimkou nahrávání řešení do odevzdávacího systému). Zakázány jsou rovněž jakékoliv další pomůcky kromě psacích potřeb (např. knihy, výpisy programů, kalkulačky, mobilní telefony). Řešení každé úlohy vypracujte na samostatný list papíru.

Řešení každé úlohy má obsahovat *popis řešení,* to znamená slovní popis principu zvoleného algoritmu, *argumenty zdůvodňující jeho správnost* (případně důkaz správnosti algoritmu), *diskusi o efektivitě* vašeho řešení (časová a paměťová složitost). Není možné odkazovat se na vaše řešení úloh domácího kola.

Do řešení nemusíte psát odpovídající program, algoritmus stačí zapsat ve vhodném pseudokódu nebo dokonce jenom slovně, je-li popis dostatečně podrobný a srozumitelný. Nemusíte detailně popisovat jednoduché operace jako vstupy, výstupy, implementaci jednoduchých matematických vztahů, vyhledávání v poli, třídění apod. Podrobnější seznam algoritmů a datových struktur, které považujeme za obecně známé a můžete je používat bez podrobnějšího popisu, naleznete na konci zadání.

Za každou úlohu můžete získat maximálně 10 bodů. Hodnotí se nejen správnost řešení, ale také kvalita jeho popisu a efektivita zvoleného algoritmu. Algoritmy posuzujeme podle jejich časové složitosti, tzn. závislosti doby výpočtu na velikosti vstupních dat. Záleží přitom pouze na řádové rychlosti růstu této funkce. V zadání každé úlohy najdete přibližné limity na velikost vstupních dat. Efektivním vyřešením úlohy rozumíme to, že váš program spuštěný s takovými daty na současném běžném počítači dokončí výpočet během několika sekund.

Vzorová řešení úloh naleznete krátce po soutěži na webových stránkách olympiády *https://mo.mff.cuni.cz/p/*. Na stejném místě bude zveřejněn i seznam úspěšných řešitelů krajského kola a seznam řešitelů postupujících do ústředního kola. Svá opravená řešení s komentáři opravovatelů najdete v OSMO.

P-II-1 Sisyfos a balvany

Známého řeckého krále Sisyfa bohové odsoudili k tomu, aby navěky marně valil balvan do kopce. Nedávno mu ale díky změnám v pracovních předpisech byli nuceni tento trest upravit a místo toho nyní balvany přenáší a řadí dle váhy.

Bohové ho postavili na začátek řady n balvanů očíslovaných zleva doprava od 0 do $n-1$; hmotnost balvanu stojícího na pozici i budeme označovat jako $H[i]$. Úkolem Sisyfa je seřadit balvany od nejlehčího po nejtěžší, tedy tak, aby na konci platilo $H[0] \leq H[1] \leq \cdots \leq H[n-1]$. Aby se vyhnul nutnosti přenášet balvany na dlouhé vzdálenosti, rozhodl se Sisyfos postupovat následovně:

- Jde podél balvanů zleva doprava.
- Kdykoli narazí na balvan, který má bezprostředně nalevo od sebe těžší balvan, vymění je, těžší balvan posune o jednu pozici doprava a lehčí balvan o jednu pozici doleva. Formálněji řečeno, když Sisyfos dorazí na pozici $i \geq 1$ a vidí, že $H[i-1] > H[i]$, prohodí hodnoty $H[i-1]$ a $H[i]$.
- Poté se vrátí na začátek řady (přitom balvany neprohazuje).

Tomuto postupu budeme říkat *kolo*. Sisyfos ho opakuje až do chvíle, kdy už v nějakém kole neprovede žádnou další výměnu.

Soutěžní úloha

Tato úloha má několik podúloh; můžete řešit každou zvlášť, ale i napsat algoritmus, který vyřeší podúlohy b) a c) zároveň.

a) (2 body) Dokažte, že pro libovolnou posloupnost balvanů Sisyfos po konečném počtu kol skončí.

b) (3 body) Ukázalo se, že Sisyfos nezvládne přesunout balvany, jejichž hmotnost je větší než w. Proto upravil svůj postup tak, že pokud by měl prohodit dva balvany, z nichž je alespoň jeden moc těžký, místo toho nic neudělá a posune se na další pozici.

   Vstup tvoří čísla n, w a pole H obsahující hmotnosti balvanů v jejich počátečním pořadí. Napište algoritmus, který určí hmotnosti balvanů na jednotlivých pozicích poté, co Sisyfos skončí. Výstupem je tedy konečný obsah pole H.

c) (5 bodů) Napište algoritmus, který pro stejný vstup jako v podúloze b) určí, kolik kol bude Sisyfův postup trvat pro tuto konkrétní posloupnost balvanů a hodnotu w. Nezapomeňte dokázat správnost vašeho algoritmu!

Připomínáme, že nemusíte řešit detaily implementace vstupů a výstupů; můžete tedy například předpokládat, že na začátku běhu algoritmů řešících podúlohy b) a c) už máte vstupy načtené v poli H a proměnných n a w.

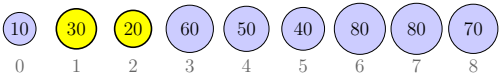
Bodování

Plný počet bodů za podúlohy b) a c) získají řešení s časovou složitostí $\mathcal{O}(n \log n)$, tedy efektivní pro $n \leq 10^5$. Řešení s časovou složitostí $\mathcal{O}(n^2)$, tedy efektivní pro $n \leq 5\,000$, získají 1 bod za každou z těchto podúloh. Jeden bod strhneme algoritmům, které fungují pouze za předpokladu, že všechny hmotnosti v poli H jsou navzájem různé.

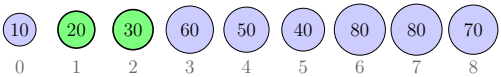
Příklady

Nechť $n = 9$ a balvany mají po řadě hmotnosti 10, 30, 20, 60, 50, 40, 80, 80, 70. Nejprve uvažujme situaci, kdy Sisyfos může přesunout všechny balvany, tj. $w \geq 80$. V prvním kole by Sisyfos postupoval následovně:

- Na pozici 1 není třeba nic dělat.
- Na pozici 2 je balvan lehčí než ten vlevo:

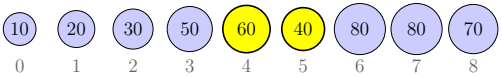


- Tyto balvany budou prohozeny:

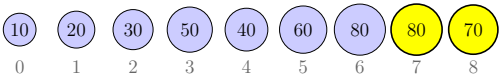


- Na pozici 3 není třeba nic dělat.
- Na pozici 4 je balvan lehčí než ten vlevo:

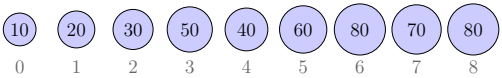
- Po jejich výměně se vyskytuje další takový případ na pozici 5:



- Na pozicích 6 a 7 není třeba nic dělat.
- Na pozici 8 provede další výměnu:

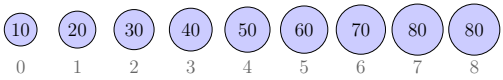


- Na konci prvního kola vypadá pole H takto:



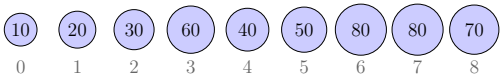
Ve druhém kole by Sisyfos provedl výměnu na pozici 4 (balvany o hmotnosti 50 a 40) a poté na pozici 7 (balvany o hmotnosti 80 a 70).

Ve třetím kole by Sisyfos neprovedl žádné výměny. Proces by tedy skončil po třech kolech seřazeným polem H:



Kdyby byl Sisyfos o něco slabší ($w = 59$), postupoval by v prvním kole následovně:

- Na pozici 1 není třeba provádět žádnou akci.
- Na pozici 2 provede výměnu (hmotnosti 30 a 20).
- Na pozici 3 není třeba nic dělat.
- Na pozici 4 nemůže provést výměnu, takže neudělá nic.
- Na pozici 5 provede výměnu (hmotnosti 50 a 40).
- Na pozicích 6 a 7 není třeba provádět žádnou akci.
- Na pozici 8 nelze provést výměnu, takže tyto dva balvany také zůstanou na svém místě.
- Na konci prvního kola vypadá pole H následovně:



Ve druhém kole by Sisyfos neprovedl žádné výměny, proces by tedy skončil po dvou kolech ve výše zobrazeném stavu.

Kdyby byl Sisyfos ve stejné výchozí situaci ještě slabší ($w = 27$), neprovedl by v prvním kole žádné změny a celý proces by ihned skončil.

P-II-2 Obdélník

Augiáš se rozhodl postavit nový chlév ve tvaru obdélníku pro svůj dobytek. Již ze dřívějška mu na jeho pozemku stojí několik sloupů, na něž mohou být stěny chléva upevněny. Jelikož je nejen nepořádný, ale i lakomý, chtěl by k tomuto účelu využít všechny tyto sloupy.

Soutěžní úloha

V rovině máte dáno $n \geq 10$ navzájem různých bodů, udávajících pozice sloupů. Navrhněte algoritmus, který určí, zda všechny tyto body leží na obvodu nějakého (libovolně otočeného) obdélníku. Pokud ano, najděte jeden takový obdélník.

Formát vstupu

Na prvním řádku vstupu je přirozené číslo n. Následuje n řádků popisujících zadané body; na i-tém z nich jsou dvě celá čísla x_i a y_i, která udávají souřadnice i-tého bodu.

Formát výstupu

Vypište buď řetězec „`NE`“, jestliže hledaný obdélník neexistuje, nebo souřadnice tří z vrcholů nějakého obdélníka, na jehož obvodu leží všechny zadané body. Tyto souřadnice nemusí být celočíselné.

Bodování

Při psaní algoritmu můžete předpokládat, že všechny operace s reálnými čísly jsou přesné. Jinými slovy, není třeba řešit zaokrouhlovací chyby, které by mohly nastat při praktické implementaci.

Abyste získali plný počet bodů, musíte najít řešení s časovou složitostí $\mathcal{O}(n)$, tedy efektivní pro $n \leq 10^7$, a dokázat jeho správnost. Řešení s časovou složitostí $\mathcal{O}(n \log n)$ (efektivní pro $n \leq 10^5$) mohou získat až 8 bodů a řešení s časovou složitostí $\mathcal{O}(n^2)$ (efektivní pro $n \leq 5\,000$) mohou získat až 6 bodů. Pomalejší správná řešení mohou získat až 4 body.

Příklady

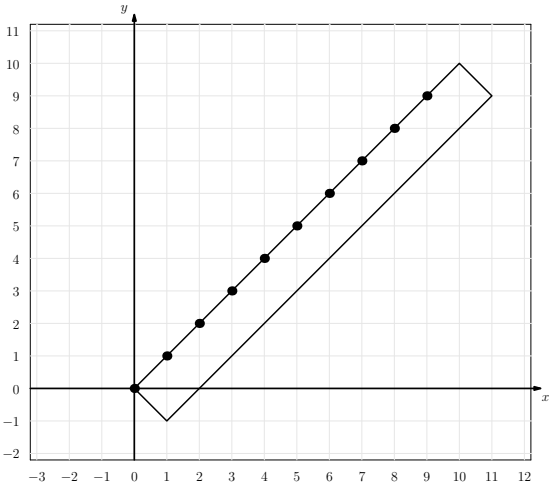
Vstup:

```
10
0 0
1 1
2 2
3 3
9 9
8 8
7 7
6 6
5 5
4 4
```

Výstup:

```
0 0
10 10
11 9
```

V tomto příkladu body leží na jedné přímce, řešením je tedy libovolný obdélník, jehož jedna strana obsahuje všechny zadané body.



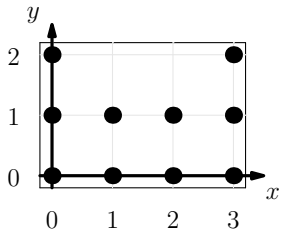
Vstup:

```
10
0 0
1 0
2 0
3 0
0 1
1 1
2 1
3 1
0 2
3 2
```

Výstup:

```
NE
```

Žádný obdélník nemá na hranici všechny zadané body.



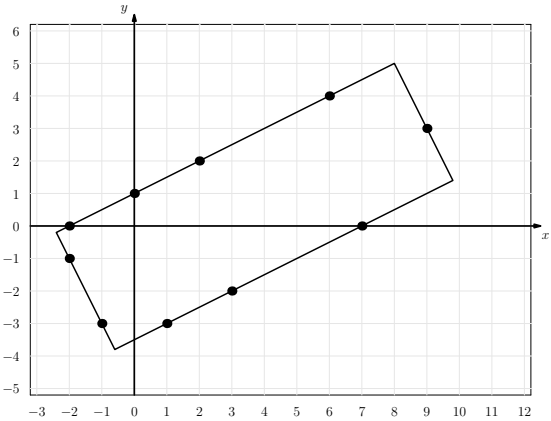
Vstup:

```
10
-2 0
0 1
2 2
6 4
9 3
7 0
3 -2
1 -3
-1 -3
-2 -1
```

Výstup:

```
-2.4 -0.2
8 5
9.8 1.4
```

Zde existuje jediné řešení.



P-II-3 Platónova Akademie

Gratulujeme, byli jste přijati ke studiu na Platónově Akademii! Nechcete ale ve školních lavicích strávit zbytečně moc času, rádi byste si proto rozvrhli předměty tak, abyste dostudovali co nejdříve.

Soutěžní úloha

Abyste vystudovali, musíte absolvovat n předmětů, očíslovaných od 0 do $n-1$. Předměty musíte studovat v předepsaném pořadí, tj. pro žádné $i < j$ nesmíte začít studovat předmět j dříve než předmět i (můžete je ale začít studovat zároveň).

Absolvování každého z předmětů vyžaduje dva po sobě jdoucí kalendářní měsíce (první měsíc chodíte na přednášky a ten druhý skládáte zkoušky). Předmět i vám zabere $A[i]$ hodin během prvního měsíce a $B[i]$ hodin během druhého měsíce. V rámci každého měsíce můžete studovat i více předmětů, nesmí vám ale dohromady zabrat více než p hodin. Speciálně tedy nesmíte začít dohromady studovat předměty, které by vám ve druhém (zkouškovém) měsíci zabraly více než p hodin, i kdyby se vám jejich první (přednáškový) měsíc do limitu p hodin vešel.

Navrhněte algoritmus, který vypočítá minimální počet měsíců potřebných k dostudování.

Formát vstupu

Na prvním řádku vstupu jsou přirozená čísla p a n. Na i-tém z n následujících řádků jsou dvě nezáporná celá čísla udávající hodnoty $A[i-1]$ a $B[i-1]$; obě tyto hodnoty jsou menší nebo rovny p.

Formát výstupu

Na výstup vypište jediné číslo, minimální počet měsíců, za něž lze dostudovat.

Bodování

Plný počet bodů mohou získat řešení s časovou složitostí $\mathcal{O}(n^2)$, tedy efektivní pro $p \leq 10^9$ a $n \leq 5\,000$. Nanejvýš 8 bodů mohou získat řešení s časovou složitostí $\mathcal{O}(n^3)$, tedy efektivní pro $p \leq 10^9$ a $n \leq 300$. Nanejvýš 6 bodů mohou získat řešení s časovou složitostí $\mathcal{O}(n^2 p)$ či obdobnou, tedy efektivní pro $p, n \leq 300$. Nanejvýš 4 body mohou získat řešení s časovou složitostí $\mathcal{O}(2^n)$ či obdobnou, tedy efektivní pro $p \leq 10^9$ a $n \leq 12$.

Příklady

Vstup:	*Výstup:*

`1000 4`
`300 100`
`300 100`
`100 800`
`700 200`
	`3`

První měsíc bychom sice mohli začít studovat předměty 0, 1 *a* 2 *dohromady, pak bychom ale na studium předmětu* 3 *měli čas až třetí a čtvrtý měsíc. Lepší bude začít v prvním měsíci pouze předměty* 0 *a* 1 *(to nám zabere* $300 + 300 = 600$ *hodin) a předměty* 2 *a* 3 *začít studovat ve druhém měsíci. Ve druhém měsíci budeme potřebovat* $100 + 100 = 200$ *hodin na dokončení předmětů* 0 *a* 1 *a* $100 + 700 = 800$ *hodin na zahájení studia předmětů* 2 *a* 3*, což se nám dohromady vejde do limitu* 1000 *hodin. Ve třetím měsíci nám dostudování předmětů* 2 *a* 3 *zabere* $800 + 200 = 1000$ *hodin.*

Vstup:	*Výstup:*

`1000 3`
`100 800`
`100 800`
`100 800`
	`4`

Žádné dva předměty nemůžeme začít studovat naráz, protože ve druhém měsíci bychom na jejich dokončení potřebovali $800 + 800 = 1600 > 1000$ *hodin. Nejlepší je tedy každý měsíc začít studovat jeden předmět.*

Vstup:	*Výstup:*

`1000 5`
`500 499`
`500 500`
`1 1`
`500 500`
`499 500`
	`4`

P-II-4 Řešič to opět vyřeší

K této úloze se vztahuje studijní text uvedený na následujících stranách, který je stejný jako v domácím kole. Úloha se skládá ze dvou nezávislých podúloh, řešit můžete kteroukoliv z nich nebo obě dvě v libovolném pořadí.

Podúloha A: Slabé závislosti (2 body)

Připomeňme zadání úlohy z domácího kola: Máme k dispozici e eur. Existuje n projektů (očíslovaných od 1 do n), do kterých můžeme investovat; každý z nich buď podpoříme, nebo ne. U každého projektu známe částku s_i potřebnou k jeho podpoře a hodnotu v_i, kterou získáme zpět jako výnos, pokud jej podpoříme. Výnos obdržíme až na konci realizace všech podpořených projektů, součet hodnot s_i těchto projektů tedy nemůže přesáhnout e.

Pro krajské kolo navíc mezi projekty mohou být *slabé závislosti* v následujícím tvaru: Projekt x_i můžeme podpořit pouze tehdy, pokud také podpoříme alespoň jeden z projektů $y_{i,1}, y_{i,2}, \ldots, y_{i,z_i}$. Například projekt pěstování bio zelí nelze uskutečnit, pokud nepodpoříme ani projekt instalace automatického zavlažování, ani projekt 3D tisku dvojručných kbelíků.

Popište, jak pro danou sadu projektů a seznam slabých závislostí mezi nimi sestavit ILP, jehož optimální řešení odpovídá sadě projektů, jejichž podpora nám přinese maximální celkový zisk. Popište celý ILP, včetně částí, které zůstávají stejné jako v domácím kole.

Příklad

Mějme $e = 110\,000$ eur a $n = 5$ projektů. Částky s_i na jejich podporu jsou po řadě 50 000, 50 000, 20 000, 40 000, 49 999 a výnosy v_i jsou po řadě 50 100, 95, 26 000, 900 000, 1.

Mezi projekty jsou dvě slabé závislosti:

- projekt 4 závisí na projektu 3
  (tedy $x_1 = 4$, $z_1 = 1$ a $y_{1,1} = 3$)
- projekt 3 závisí na projektech 2 a/nebo 5
  (tedy $x_2 = 3$, $z_2 = 2$, $y_{2,1} = 2$ a $y_{2,2} = 5$).

Optimální řešení je podpořit projekty 2, 3 a 4, na konci tak budeme mít 816 095 eur.

Pokud bychom ve stejné situaci měli jen 100 000 eur, optimální by bylo podpořit jen projekt 1.

Podúloha B: Lyžařské sjezdovky (8 bodů)

Kolem horské vesnice se nachází k kopců očíslovaných od 1 do k, na kterých můžeme budovat lyžařské sjezdovky. Na každý kopec se nám ale sjezdovky vejdou *nejvýše tři*. Každá sjezdovka postavená na kopci i musí mít celočíselnou délku, a to alespoň ℓ_i a nanejvýš u_i metrů. Postavení každého metru sjezdovky na kopci i nás stojí c_i eur. Navíc bychom chtěli, aby celková délka všech sjezdovek v celém lyžařském areálu byla přesně d metrů.

Popište, jak sestavit ILP, který bude mít řešení, právě když je to možné. Optimální řešení tohoto ILP navíc musí odpovídat nejlevnějšímu možnému způsobu výstavby požadované sady sjezdovek.

Jednodušší varianty

Můžete se také rozhodnout řešit jednu z následujících jednodušších variant této podúlohy:

- Za vyřešení varianty, v níž je na každém kopci možné postavit *nejvýše jednu* sjezdovku, můžete získat až 6 bodů.
- Za vyřešení varianty, v níž všechny kopce splňují $\ell_i = u_i$, můžete získat až 3 body.

Pokud odevzdáte řešení jedné z těchto jednodušších variant, uveďte to prosím výrazně hned na jeho začátku.

Příklad

Máme $k = 2$ kopce:

- Na prvním se dají stavět sjezdovky délky $\ell_1 = 1000$ až $u_1 = 1100$ metrů, a to metr za $c_1 = 100$ eur,
- na druhém sjezdovky délky $\ell_2 = 300$ až $u_2 = 400$ metrů, a to metr za $c_2 = 70$ eur.

Chceme areál s 2410 metry sjezdovek.

Jedním optimálním řešením je postavit na prvním kopci dvě sjezdovky délek 1003 a 1007 metrů a na druhém kopci jednu sjezdovku délky 400 metrů. Dohromady nás to bude stát 229 000 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 olympiády 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 olympiády.

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

Aby 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, které 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.

Knihovna základních algoritmů

Pokud ve svém řešení teoretické úlohy chcete použít nějaký základní algoritmus, jako třeba binární vyhledávání v uspořádaném poli, nemusíte ho detailně popisovat. Někdy ale nemusí být jasné, které algoritmy jsou „dostatečně základní“.

Uvádíme proto seznam algoritmů a datových struktur, které v MO-P považujeme za natolik známé, že se na ně v řešení stačí odkázat a není potřeba uvádět detaily. Pokud si algoritmus potřebujete nějak přizpůsobit, stačí popsat pouze změny od základní verze.

Ostatní algoritmy je potřeba popsat celé.

Matematika

Euklidův algoritmus

- Nalezení největšího společného dělitele čísel x, y
- Čas $\mathcal{O}(\log x + \log y)$

Eratosthénovo síto

- Nalezení prvočísel od 2 do n
- Čas $\mathcal{O}(n \log \log n)$

Faktorizace

- Rozklad čísla n na součin prvočísel zkoušením dělitelů až po $\sqrt{n}$
- Čas $\mathcal{O}(\sqrt{n})$

Posloupnosti

Označme n délku posloupnosti.

Merge sort

- Třídění sléváním
- Čas $\mathcal{O}(n \log n)$

Bucket sort

- Přihrádkové třídění
- Čas $\mathcal{O}(n + r)$, kde n je počet prvků a r je jejich rozsah

Binární vyhledávání

- Nalezení prvku x v setříděné posloupnosti, případně největšího prvku $\leq x$
- Čas $\mathcal{O}(\log n)$

Grafy

Označme n počet vrcholů grafu a m počet jeho hran.

Prohledávání do hloubky

- Čas $\mathcal{O}(n + m)$

Prohledávání do šířky

- Nalezení nejkratší cesty ze startu do všech vrcholů v neohodnoceném grafu
- Čas $\mathcal{O}(n + m)$

Dijkstrův algoritmus

- Nalezení nejkratší cesty ze startu do všech vrcholů v ohodnoceném grafu s nezápornými délkami hran
- Čas $\mathcal{O}(m + n \log n)$

Bellman-Fordův algoritmus

- Nalezení nejkratší cesty ze startu do všech vrcholů v ohodnoceném grafu
- Čas $\mathcal{O}(nm)$

Floydův-Warshallův algoritmus

- Nalezení nejkratší cesty z každého do každého vrcholu v ohodnoceném grafu
- Čas $\mathcal{O}(n^3)$

Jarníkův algoritmus

- Nalezení minimální kostry pomocí řezů a haldy
- Čas $\mathcal{O}(m + n \log n)$

Kruskalův algoritmus

- Nalezení minimální kostry pomocí Disjoint set union
- Čas $\mathcal{O}(m \log n)$

Topologické uspořádání

- Uspořádání všech vrcholů orientovaného acyklického grafu tak, aby hrany vedly po směru uspořádání
- Čas $\mathcal{O}(n + m)$

Geometrie

Základy

- Vzdálenost dvou bodů (čas $\mathcal{O}(1)$)
- Vzdálenost bodu a přímky (čas $\mathcal{O}(1)$)
- Průnik přímek, úhly jimi svírané (čas $\mathcal{O}(1)$)
- Obsah mnohoúhelníku (čas $\mathcal{O}(n)$, kde n je počet vrcholů)

Konvexní obal

- Nalezení konvexního obalu n bodů
- Čas $\mathcal{O}(n \log n)$, případně $\mathcal{O}(n)$ s již seřazenými body

Datové struktury

U datových struktur uvádíme operace, které podporují, s jejich složitostmi.

Fronta

- *Enqueue:* Přidání prvku na konec v $\mathcal{O}(1)$
- *Dequeue:* Odebrání prvku ze začátku v $\mathcal{O}(1)$

Zásobník

- *Push:* Přidání prvku na konec v $\mathcal{O}(1)$
- *Pop:* Odebrání prvku z konce v $\mathcal{O}(1)$

Nafukovací pole

- *Append:* Přidání prvku v *amortizovaně* $\mathcal{O}(1)$

Prefixové součty

- *Build:* Vybudování v $\mathcal{O}(n)$
- *Query:* Dotaz na součet intervalu v $\mathcal{O}(1)$

2D prefixové součty

- *Build:* Vybudování v $\mathcal{O}(nm)$, kde n a m jsou strany mřížky
- *Query:* Dotaz na součet obdélníka v $\mathcal{O}(1)$

Vyhledávací strom

- Reprezentace množiny prvků, které umíme porovnávat.
- *Find:* Nalezení pozice daného prvku, případné zahlášení jeho neexistence v $\mathcal{O}(\log n)$
- *Insert:* Přidání prvku v $\mathcal{O}(\log n)$
- *Delete:* Smazání prvku v $\mathcal{O}(\log n)$

Písmenkový strom (Trie)

- Reprezentace množiny řetězců nad konstantně velkou abecedou.
- *Find:* Zjištění, zdali je slovo ve stromě v $\mathcal{O}(\ell)$, kde ℓ je délka slova
- *Insert:* Přidání slova v $\mathcal{O}(\ell)$
- *Delete:* Smazání slova v $\mathcal{O}(\ell)$

Binární halda

- *Min:* Vrácení minima v $\mathcal{O}(1)$
- *ExtractMin:* Odstranění minima v $\mathcal{O}(\log n)$
- *Insert:* Přidání prvku v $\mathcal{O}(\log n)$
- *Delete:* Smazání prvku v $\mathcal{O}(\log n)$, známe-li jeho pozici v haldě

Disjoint set union

- *Find:* Nalezení reprezentanta množiny obsahující daný vrchol v *amortizovaně* $\mathcal{O}(\alpha(n))$, kde α je inverzní Ackermannova funkce, která sice roste do nekonečna, ale mnohem pomaleji než logaritmus.
- *Union:* Sjednocení dvou množin obsahujících vrcholy u a v v *amortizovaně* $\mathcal{O}(\alpha(n))$

Intervalový strom (s línou aktualizací)

- Pro zvolenou asociativní operaci (minimum, maximum, součet, ...) a posloupnost prvků $x_1, \ldots, x_n$
- *Build:* Vybudování v $\mathcal{O}(n)$
- *Query:* Určení hodnoty operace provedené na všechny prvky s indexy v daném intervalu v $\mathcal{O}(\log n)$
- *UpdatePoint:* Aktualizace prvku v $\mathcal{O}(\log n)$
- Je-li operace minimum, maximum či součet, můžete dále bez popisu používat i operaci *UpdateRange:* Ke všem hodnotám s indexy v daném intervalu přičte zadané číslo, pracuje v čase $\mathcal{O}(\log n)$.
- Potřebuje-li vaše řešení operaci *UpdateRange* v jiných situacích (jiná asociativní operace, jiný druh změny hodnot na intervalu), musíte vysvětlit, jak ji implementujete.