PRASK
III. ročník, 2016/17
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola zimnej časti

vzorák napísal Žaba
(max. 15 b za riešenie)

1. Prehra na šachovnici

V úlohe ste mali riešiť pomerne jednoduchú hru na šachovnici. Postupy, ktoré si ukážeme v tomto vzorovom riešení sú však aplikovateľné aj na oveľa zložitejšie hry. Pustime sa teda do toho.

Podúloha a)

Na prvom obrázku máme koňa na políčku $(5, 5)$ (pri označovaní pozície budeme najskôr písať číslo riadku a potom číslo stĺpca). Našou úlohou je zistiť, ktorý hráč vyhrá ak budú obaja hrať optimálnym spôsobom. Kika ťahá ako prvá.

Prvá otázka je, čo vlastne znamená, že hráči hrajú optimálne. V podstate si to vieme predstaviť tak, že ak existuje spôsob, akým vie jeden z nich vyhrať za každých okolností a bez ohľadu na to, čo spraví jeho súper, tak sa bude podľa toho riadiť a skutočne vyhrá. Musí však vždy existovať takáto vyhrávajúca postupnosť ťahov? Ako uvidíme neskôr, musí.

Zatiaľ však nevieme spraviť nič iné, ako vyskúšať všetky možnosti. Kam teda môže Kika pohnúť koňa v prvom ťahu? Iba na pozície $(3, 6)$, $(3, 4)$, $(4, 3)$ a $(6, 3)$. Všimnime si, že tieto pozície sú si navzájom symetrické. Ak teda bude existovať vyhrávajúca postupnosť ťahov začínajúca ťahom na $(3, 6)$, bude existovať aj úplne rovnaká postupnosť ťahov pre začiatok $(6, 3)$, akurát vždy vymeníme riadok a stĺpec políčok, na ktoré sa máme posunúť. Pokračujme teda iba s políčkami $(3, 6)$ a $(3, 4)$.

Predstavme si, že Kika posunula koňa na políčko $(3, 4)$. Na ťahu je teraz Andrej a možnosti, kam sa vie pohnúť sú $(1, 5)$, $(1, 3)$, $(2, 2)$ a $(4, 2)$. No ale políčko $(2, 2)$ je jedno zo štyroch finálnych políčok. Hneď ako Andrej posunie koňa na toto políčko vyhral, pretože Kika už nebude môcť spraviť ďalší ťah. Ak chce Kika vyhrať, tak sa preto nemôže posunúť na políčko $(3, 4)$, lebo Andrej (ktorý hrá optimálne a nemýli sa) by vedel vyhrať.

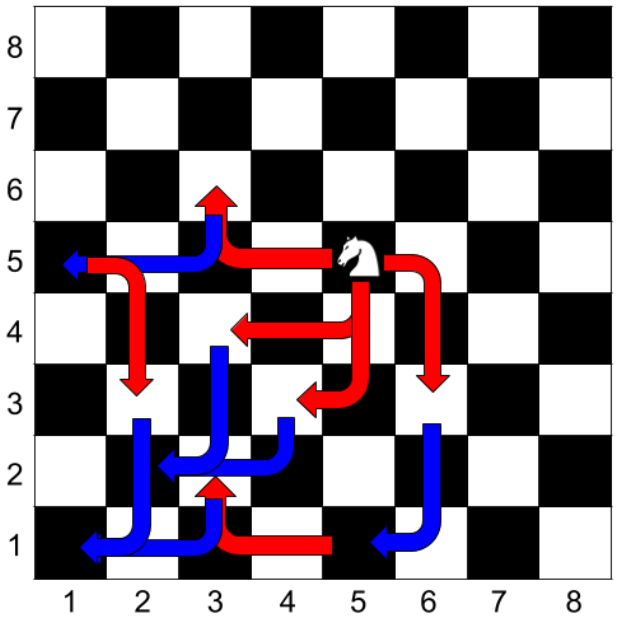
Ostáva teda možnosť, v ktorej sa Kika posunie na políčko $(3, 6)$. V tomto prípade má Andrej nasledovné možnosti, kam môže koňa posunúť: $(1, 7)$, $(1, 5)$, $(2, 4)$ a $(4, 4)$. Ani z jedného z týchto políčok nevie Andrej vyhrať okamžite, tak ako v predchádzajúcom prípade. Mali by sme teda opäť skúsiť všetky možnosti. Všimnime si však, že políčko $(2, 4)$ je príliš blízko cieľa a Kika by zneho vedela hneď vyhrať pohybom na $(1, 2)$. Tam teda Andrej ťahať nebude. Nádejne však vyzerá políčko $(1, 5)$. Z neho má totiž Kika iba jednu možnosť ako sa pohnúť a to na políčko $(2, 3)$. A ak sa Kika pohne na políčko $(2, 3)$, tak Andrej sa pohne na $(1, 1)$ a vyhrá.

Keď sa teda Kika posunie na $(3, 6)$, tak Andrej spraví krok na políčko $(1, 5)$ čím si zaručí, že Kika potiahne na $(2, 3)$ a on v ďalšom kole vyhrá ťahom na $(1, 1)$. Vidíme teda, že bez ohľadu na to, čo spraví Kika, Andrej to vie zahrať tak aby vyhral. Pre Andreja teda existuje vyhrávajú stratégia, ktorá je znázornená aj na obrázku nižšie. Červenou sú znázornené všetky ťahy, ktoré môže robiť Kika a modrou ťahy, ktoré bude robiť Andrej s cieľom vyhrať.

V tejto podúlohe bol však aj druhý obrázok, v ktorom bol kôň na začiatku na políčku $(6, 7)$. Musíme opäť skúšať všetky možnosti? Nie, ak budeme šikovný a všimneme si, že jeden z možných ťahov, ktoré vie Kika spraviť je posunúť koňa na políčko $(5, 5)$. Čo je presne predchádzajúci prípad. A o tom sme si ukázali, že hráč, ktorý začína s koňom na pozícii $(5, 5)$ zaručene prehrá. Keď však Kika spraví prvý ťah na políčko $(5, 5)$, v podstate vytvorí novú hru, kde kôň začína na políčku $(5, 5)$ a začínajúcim hráčom je Andrej. Tým pádom bude Kika vedieť robiť to isté čo Andrej pred chvíľou a tým pádom vyhrať. V druhej hre preto zvíťazí Kika.

Podúloha b)

V podúlohe b) je našou úlohou vyplniť celú tabuľku 8×8 číslami 0 a 1, pričom 0 znamená, že ak bude kôň na začiatku na tejto pozícii, Kika prehrá a 1 znamená, že z tejto pozície Kika vyhrá. Napríklad sme už zistili, že okrem štyroch rohových políčok, bude 0 aj na políčku $(5, 5)$. Na políčku $(6, 7)$ však bude 1, lebo z tejto pozície Kika vyhrá.



Toto označenie však nie je úplne najšťastnejšie, lebo na definovanie používa Kiku. Ak by teda napríklad Kika nezačínala, museli by sme všetky čísla prepísať. Namiesto toho budeme používať nasledovné dva pojmy:

- prehrávajúca pozícia (označená 0) – ak je kôň na tejto pozícii, hráč, ktorý je na ťahu prehrá
- vyhrávajúca pozícia (označená 1) – ak je kôň na tejto pozícii, hráč, ktorý je na ťahu vyhrá

Vcelku jednoduché a stále je to tá istá vec, ktorú po nás chcelo zadanie. Všimnime si teraz, čo to znamená, pre náš druhý príklad. Kika začínala na pozícii $(6, 7)$, ktorá ako už vieme je vyhrávajúca. Následne koňa presunula na pozíciu $(5, 5)$, ktorá je prehrávajúca a naozaj, Andrej, ktorý bol v tom momente na ťahu vyhrať nevedel. Táto pozícia bola pre neho ako hráča naťahu naozaj prehrávajúca.

Pozrime sa teraz na nasledovný obrázok. Sú na ňom vyznačené políčka, o ktorých zaručene vieme, že sú vyhrávajúce. Z každého z týchto políčok sa totiž vieme posunúť do jedného zo štyroch rohových miest. Tieto rohové pozície sú prehrávajúce, lebo ak sa na nich ocitne kôň, hráč, ktorý je na ťahu prehrá. A teda hráč, ktorý vie koňa premiestniť zo svojej pozície na jedno z týchto políčok je vo vyhrávajúcej pozícii.

5					
4	**1**	**1**	**1**		
3	**1**	**1**	**1**	**1**	
2	**0**	**0**	**1**	**1**	
1	**0**	**0**	**1**	**1**	
	1	2	3	4	5

Z týchto pozorovaní vieme vyvodiť dve pravidlá:

- Pozícia je vyhrávajúca, ak **aspoň jeden ťah**, ktorý z tejto pozície vedie, ide do prehrávajúcej pozície.
- Pozícia je prehrávajúca ak **všetky ťahy** z tejto pozície vedú do vyhrávajúcej pozície.

Správnosť týchto úvah je zjavná. Ak z našej pozície vedie ťah do prehrávajúcej pozície, môžeme tento ťah spraviť, náš súper sa ocitne v prehrávajúcej pozícii a teda vieme vyhrať. Naopak, ak sme v prehrávajúcej pozícii, nech spravíme čokoľvek, stav, v ktorom sa ocitne náš súper po ľubovoľnomi našom ťahu bude vyhrávajúci a preto prehráme. Tieto dve definície sa navyše doplňujú a preto pre každé políčko platí práve jedna z nich – každé políčko je teda buď prehrávajúce alebo vyhrávajúce.

Ako teda v mriežke $r \times s$ zitíme, ktoré pozície sú prehrávajúce a ktoré vyhrávajúce? Vždy keď chceme o nejakom políčku (x, y) zistiť, či je vyhrávajúce alebo prehrávajúce, potrebujeme túto informáciu poznať o políčkach $(x+1, y-2)$, $(x-1, y-2)$, $(x-2, y+1)$ a $(x-2, y-1)$. Pričom o štyroch rohových políčkach vieme, že sú prehrávajúce. Naviac, aby sme tieto informácie mali vždy k dispozícii, budeme tieto hodnoty počítať po diagonálach.

Pre každé políčko sa teda pozrieme na všetky štyri políčka, na ktoré sa z neho vieme dostať. Ak je medzi nimi aspoň jedna 0, tak vieme, že toto políčko je vyhrávajúce (označíme 1) a ak sú medzi nimi iba 1, tak toto políčko bude prehrávajúce. Tento postup bude samozrejme fungovať pre ľubovoľne veľkú šachovnicu.

Na našej webovej stránke je nižšie uvedený obrázok pohybujúce sa GIF, ktoré vám ukáže, ako ste mohli postupne vypočítať hľadané hodnoty.

1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1

Podúloha c)

V tejto podúlohe je na šachovnici položených viacero koní. To, že je viac koní na jednom políčku nám neprekáža a v jednom ťahu môže hráč posunúť ľubovoľne veľa koní podľa predchádzajúcich pravidiel. Musí však posunúť aspoň jednu figúrku.

Ak chceme riešiť podobnú úlohu, asi je dobré si ju vyskúšať s dvoma, troma koňmi a sledovať, ako sa takáto hra správa. To však v tomto vzorovom riešení robiť nebudeme. Skúsime sa nad tým iba zamyslieť. Všimnime si, že hráči hrajú akoby viacero hier naraz, pričom v každom ťahu môžu spraviť ťah v ľubovoľnom množstve týchto hier.

Často nám tiež pomôže, ak sa zamyslíme, čo je úplne posledná pozícia, ktorá v tejto hre nastane. Je jasné, že hra skončí ak budú všetky kone v štyroch rohových políčkach šachovnice. Teda na políčkach, kde boli v predpošlej hre napísané 0.

S týmto pozorovaním už vieme vymyslieť správne riešenie. Toto riešenie definuje vyhrávajúce a prehrávajúce pozície nasledovne:

- Pozícia je prehrávajúca, ak všetky kone stoja na políčkach s číslom 0 (čísla vypočítané v predchádzajúcej podúlohe.)
- Pozícia je vyhrávajúca, ak aspoň jeden kôň stojí na políčku s číslom 1 (čísla vypočítané v predchádzajúcej podúlohe.)

Ostáva už len dokázať, že takéto riešenie je správne, teda overiť, že z každej vyhrávajúcej pozície vedie ťah do prehrávajúcej pozície a z každej prehrávajúcej vedú všetky ťahy do vyhrávajúcej pozície (prečítajte si túto vetu ešte raz a poriadne sa zamyslite čo hovorí).

Ak sme vo vyhrávajúcej pozícii, niektoré kone (aspoň jeden) stoja na políčkach s číslom 1. Toto sú hry, v ktorých vyhrávame (sme v ich vyhrávacej pozícii). Ak sa teda každým z týchto koní posunieme na políčko s 0 (podľa podúlohy b) takéto políčko existuje), súper ostane v situácii, keď budú všetky kone na políčku s 0 a teda bude v prehrávajúcej pozícii.

Naopak, ak sme v prehrávajúcej pozícii a pohneme ľubovoľným koňom, posunieme ho na políčko s číslom 1. A keďže aspoň jedného koňa pohnúť musíme, zanecháme nášho súpera vo vyhrávajúcej pozícii bez ohľadu na to, čo spravíme.

Ak chce teda Kika vyhrať, musí spraviť nasledovné. Pre každé políčko šachovnice si spočíta, či je daná pozícia 0 alebo 1, presne tak ako v podúlohe b). Následne pohne každého koňa, ktorý stojí na pozícii s číslom 1 na pozíciu s číslom 0. Tým zanechá Andreja v pozícii so samými 0 a Andrej prehrá. Ak však na začiatku všetky kone stoja na pozícii s 0, tak Kika vie, že prehrá bez ohľadu na to čo spraví.

Postupy, ktoré ste videli v tomto vzorovom riešení sa dajú aplikovať všeobecne. Definícia prehrávajúcich a vyhrávajúcich pozícii bude zakaždým rovnaká, aj keď ich počítanie môže byť rôzne. S ich pomocou by sme teda

mali byť schopní vyhrať ľubovoľnú hru dvoch hráčov, ktorú neovplyvňuje náhoda (Monopoly alebo Človeče asi víťaznú stratégiu nemá) a takisto žiadna informácia nie je skrytá (napr. kartové hry, v ktorých náš ťah závisí od toho, aké karty má na ruke náš protihráč). Napríklad pre hru šach existuje víťazná (v skutočnosti skôr neprehrávajúca, teda vedúca k remíze) stratégia. Jediný problém je, že šach má natoľko rôznych možných ťahov a stavov, že sa nám pre ňu nedarí zistiť, ktoré pozície sú vyhrávajúce a ktoré prehrávacjúce. S dostatočne silným počítačom (ktorý zatiaľ ešte neexistuje) by sme to však mali vedieť zistiť.

vzorák napísal Syseľ
(max. 15 b za riešenie)

2. Pestovanie termosenzitívneho ovocia v Absurdistane

Po tom, čo si od vás genetickí inžinieri z Absurdistanu dva krát vypýtali pomoc, zistili, že skupinka informatikov riešila podobné problémy už pred vami. Kľúčové slovo, ktoré treba gúgliť je *L-Systém*. Na wikipédii[1] nájdete okrem iného aj kopu pekných obrázkov, takže by ste sa tam určite mali aspoň pozrieť. Ak by ste sa však pokúsili nájsť niečo o Ananásovej, Citrónovej alebo Fibonacciho sade pravidiel, veľa by ste toho nenašli. Preto si o nich niečo povieme tu.

Ananásová sada

$\mathrm{A} \xrightarrow{T} \mathrm{ABBB}$	$\mathrm{A} \xrightarrow{S} \mathrm{AB}$
$\mathrm{B} \xrightarrow{T} \mathrm{B}$	$\mathrm{B} \xrightarrow{S} \mathrm{B}$

Už v zadaní sa objavila nápoveda, že každé bobulové semeno vyprodukuje jedno nové do ďalšieho roku a teda sa zachová bez ohľadu na teplotu. Ananás taktiež zachová sám seba a ešte popri tom vyprodukuje nejaké nové bobule. V teplom roku tri, v studenom iba jednu. Začínajúc s jedným ananásom teda vieme každý rok vyprodukovať o jeden alebo o tri plody viac ako v tom predošlom. Asi vás teda neprekvapí, že po dostatočne dlhej dobe vieme dosiahnuť ľubovoľný počet plodov.

Požadovaných 42 plodov dosiahneme napríklad po 41 studených rokoch. Prečo sa ponáhľať? V duchu tohto hesla môžeme pokračovať aj pri najneskoršom dosiahnutí 4247 plodov. Bude na to treba len 4246 studených rokov. Winter is coming! Ak by sme, naopak, chceli 4247 plodov čo najskôr, potrebovali by sme $4246 \div 3 = 1415 + \frac{1}{3}$ teplých rokov. Keďže výsledok nie je celé číslo, budeme ich musieť doplniť jedným studeným rokom.

Správne odpovede teda sú:

a) 42 plodov dosiahneme napríklad postupnosťou 41 studených rokov
b) 4247 plodov dosiahneme najskôr po 1416 rokoch a najneskôr po 4246 rokoch.

Citrónová sada

$\mathrm{C} \xrightarrow{T} \mathrm{CE}$	$\mathrm{C} \xrightarrow{S} \mathrm{DE}$
$\mathrm{D} \xrightarrow{T} \mathrm{CE}$	$\mathrm{D} \xrightarrow{S} \mathrm{D}$
$\mathrm{E} \xrightarrow{T} \mathrm{EE}$	$\mathrm{E} \xrightarrow{S} \mathrm{E}$

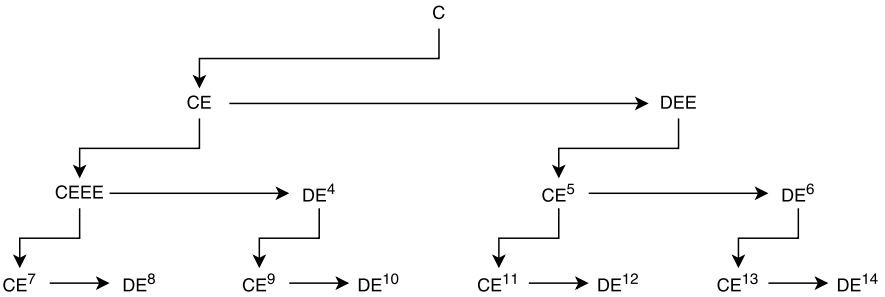
Človek by si povedal, že egreše sú len obyčajné bobule. A počas studených rokov je to aj pravda – len zachovávajú svoj rod. Avšak počas teplých rokov sa zákerne zdvojnásobia. Také niečo by sa citrónu s dyňou nemohlo stať. Z týchto dvoch plodín každý rok totiž existuje iba jedna a to práve jeden kus. V teplom roku vyrastie na ich rastlinách nový citrón (presne jeden) a v studenom zase nová dyňa (znova presne jedna). Môžeme teda povedať, že v závislosti od teploty predošlého roka, máme zasadený jeden citrón alebo jednu dyňu a niekoľko egrešov.

Pozrime sa teraz na situáciu z iného uhľa. V teplom roku každá rastlina vyprodukuje dve semená a teda celkový počet semien bude na konci roka dvojnásobný. V studenom roku vieme vypestovať iba o jeden egreš viac a aj to len v prípade, že máme citrón (a teda predošlí rok bol teplý). V každom ďalšom studenom roku už nenastane žiadna zmena v zložení semien.

Počet semien teda môžeme buď zdvojnásobiť alebo zväčšiť o jedna. Po každom zväčšení však musíme znova zdvojnásobovať – možno nie hneď rok na to, ale ak príde opäť studený rok, nič sa nezmení. Možný vývoj situácie vieme odsledovať na nasledujúcom diagrame. Šípky nadol predstavujú teplé roky, šípky doprava predstavujú

[1] https://cs.wikipedia.org/wiki/L-syst%C3%A9m

studené roky. Aby sa nám všetky egreše zmestili na obrazovku (a vy ste ich nemuseli počítať) budeme jeden citrón a n egrešov písať ako CE^n.



Môžeme si všimnúť, že po najviac troch teplých a troch studených rokoch, vieme dosiahnuť ľubovoľný počet plodov od jedného až po 15. To nám napovedá, že by sme mohli vedieť dosiahnuť ľubovoľný počet plodov. Skúsme sa na situáciu pozrieť od konca:

1) Ak sa v nejakom roku urodil nepárny počet plodov, musel tento rok byť studený. V teplých rokoch sa totiž počet plodov zdvojnásobí a teda je nutne párny.
2) Ak sa v nejakom roku urodil párny počet plodov, mohol to byť teplý rok. Vo väčšine prípadov dokonca aj musel, skúste sa zamyslieť, prečo.

Tieto dve pozorovania nám stačia na to, aby sme vymysleli univerzálny postup na dosiahnutie ľubovoľne veľkej úrody. Ak chceme dosiahnuť nepárnu úrodu, prehlásime posledný rok za studený a od úrody odpočítame jeden plod. Ak chceme dosiahnuť párnu úrodu, prehlásime posledný rok za teplý a úrodu predelíme dvoma. Na zvyšnú úrodu aplikujeme ten istý postup a opakujeme ho až kým nám nezostane úroda veľkosti jedna, čo je počiatočný citrón. V každej situácii teda vieme, čo máme robiť. Navyše, nestane sa nám, že by sme potrebovali dva studené roky po sebe, pretože studený rok použijeme iba na vytvorenie nepárne veľkej úrody a teda rok predtým bola párne veľká, čiže rok bol teplý.

Pozrime sa na vývoj pri niekoľkých úrodách:

$$42 \xleftarrow{T} 21 \xleftarrow{S} 20 \xleftarrow{T} 10 \xleftarrow{T} 5 \xleftarrow{S} 4 \xleftarrow{T} 2 \xleftarrow{T} 1$$

$$4247 \xleftarrow{S} 4246 \xleftarrow{T} 2123 \xleftarrow{S} 2122 \xleftarrow{T} 1061 \xleftarrow{S} 1060 \xleftarrow{T} 530 \xleftarrow{T} 265$$
$$265 \xleftarrow{S} 264 \xleftarrow{T} 132 \xleftarrow{T} 66 \xleftarrow{T} 33 \xleftarrow{S} 32 \xleftarrow{T} 16 \xleftarrow{T} 8 \xleftarrow{T} 4 \xleftarrow{T} 2 \xleftarrow{T} 1$$

$$23 \xleftarrow{S} 22 \xleftarrow{T} 11 \xleftarrow{S} 10 \xleftarrow{T} 5 \xleftarrow{S} 4 \xleftarrow{T} 2 \xleftarrow{T} 1$$

Vyriešili sme teda všetky problémy:

c) Hľadaná postupnosť je TTSTTST
d) Hľadaná postupnosť je TTTTTSTTTSTTSTSTS
e) Vieme zostrojiť ľubovoľne veľkú úrodu. Postup sme popísali vyššie.

Fibonacciho sada

$\mathrm{F} \xrightarrow{T} \mathrm{G}$	$\mathrm{F} \xrightarrow{S} 0$
$\mathrm{G} \xrightarrow{T} \mathrm{FG}$	$\mathrm{G} \xrightarrow{S} \mathrm{G}$

Pozrime sa najskôr čo sa deje pri dlhých postupnostiach teplých rokov.

Po roku	0	1	2	3	4	5	6	7	8	9	10
Počet F	1	0	1	1	2	3	5	8	13	21	34
Počet G	0	1	1	2	3	5	8	13	21	34	55
Počet Spolu	1	1	2	3	5	8	13	21	34	55	89

Môžeme si všimnúť, že počty písmen F, počty písmen G ako aj počty písmen spolu tvoria fibonacciho postupnosť. (Vedeli by ste odôvodniť prečo?) Pokiaľ však príde jeden alebo viac studených rokov, všetky F nám vyhynú a počet G sa nezmení. V nasledujúcom teplom období si potom každé G začne nezávisle rozvíjať svoju fibonacciho postupnosť. Každá z týchto postupností sa rozvinie do rovankého bodu pred ďalším studeným blokom, kde sa z nich znova stanú nezávislé začiatky postupností. Takto sa vieme vystriedať ľubovoľne veľa krát a výsledok teda bude súčin niekoľkých členov fibonacciho postupnosti.

Napríklad 42 je súčin 2 a 21. Vieme teda počas troch teplých rokov zgenerovať GFG, ktoré jedným studeným rokom zredukujeme na GG a následne počkáme ešte 6 teplých rokov, kým každé z oboch G vyprodukuje 21 plodov.

Úrody veľkosti 1, 2, 3, $4 = 2 \cdot 2$, 5 a $6 = 2 \cdot 3$ teda zostrojiť vieme. Číslo 7 však nie je členom fibonacciho postupnosti a keďže je to prvočíslo, nevieme ho zapísať ako súčin iných prirodzených čísel. Úrodu veľkosti 7 teda nevieme dosiahnuť.

Tým sme zodpovedali aj posledné dve otázky:

f) Hľadaná postupnosť je TTTSTTTTTT
g) Nevieme dosiahnuť napríklad 7 plodov.

vzorák napísal Prefix
(max. 15 b za riešenie)

3. Pofidernejšie kasíno

Táto úloha bola zameraná na pravdepodobnosť – v každej podúlohe chceme tipovať tak, aby sme mali čo najväčšiu šancu na výhru, a v niektorých bolo navyše treba v popise ukázať, prečo to tak je.

Podúloha a)

V tejto podúlohe bolo dôležité všimnúť si, že nie všetky čísla padajú rovnako často. Prečo?

Predstavme si, že hádžeme štyrmi šeststennými kockami. Na to, aby padol súčet $4 = 1+1+1+1$, musí na každej kocke padnúť jednotka. Sú však štyri možnosti, ako môže padnúť súčet $5 = 1+1+1+2 = 1+1+2+1 = 1+2+1+1 = 2+1+1+1$, každá z týchto možností je rovnako pravdepodobná, ako že padne súčet 4. Pre súčet 6 máme dokonca desať možností.

Vo všeobecnosti, má každá kocka n možností, ktoré môžu padnúť a teda logicky štyri kocky majú $n \cdot n \cdot n \cdot n = n^4$ možností čo môžu padnúť. Každá z týchto n^4 možností je rovnako pravdepodobná a vieme si tiež pre každý súčet S spočítať, koľkými možnosťami vieme tento súčet dosiahnuť.

Aby sme to nemuseli počítať ručne, napíšeme si program, ktorý to spočíta za nás.

Listing programu (Python)

```
pocet_stien = int(input())
kolko_padne = [0 for i in range(pocet_stien*4+1)]
for a in range(1,pocet_stien+1):
    for b in range(1,pocet_stien+1):
        for c in range(1,pocet_stien+1):
            for d in range(1,pocet_stien+1):
                kolko_padne[a+b+c+d] += 1
for i in range(4, pocet_stien*4+1):
    print(i, kolko_padne[i])
```

Napríklad pre $n = 6$, máme dokopy $6^4 = 1296$ možností, ako môžu jendotlivé kocky padnúť. Takto vyzerá tabuľka, keď tieto možnosti zoskupíme podľa súčtu padnutých hodnôt. Na prvom riadku tabuľky je súčel S, a na druhom riadku počet možností, koľkými daný súčet mohol padnúť.

S	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
P	1	4	10	20	35	56	80	104	125	140	146	140	125	104	80	56	35	20	10	4	1

Z tohoto vidíme, že 14 vážne padá najčastejšie, a vieme aj vypočítať pravdepodobnosť, že padne. Padne 146-krát z 1296 prípadov, a teda pravdepodobnosť je $\frac{146}{1296}$, a to je približne 0.11, teda 11%.

Ostatné prípady vieme vyriešiť podobne. Keďže n je maximálne 10, vieme si rýchlo overiť, že pre každé n je najväčšia pravdepodobnosť práve v strede, teda **najčastejšie padá hodnota** $2(n+1)$.

Pokiaľ sme boli leniví písať štyri forcykly, mohli sme jednoducho spraviť program, ktorý miliónkrát hodí štyrmi kockami, čím by sme tiež zistili, že $2(n+1)$ padá najčastejšie. Nepomohlo by nám to však v teoretickej časti, lebo by sme nevedeli presne vyjadriť pravdepodobnosť tohto súčtu.

Listing programu (Python)

```
from random import randint
pocet_stien = int(input())
kolko_padne = [0 for i in range(pocet_stien*4+1)]
for i in range(1000000):
    kolko_padne[sum([randint(1, pocet_stien) for _ in range(4)])] += 1
for i in range(4, pocet_stien*4+1):
    print(i, kolko_padne[i])
```

Podúloha b)

Toto je zovšeobecnenie celkom známeho problému Monty Hall. Ukazuje sa, že sa nám vždy oplatí v druhom kole zmeniť dvere. Prečo je to tak, vieme jednoducho určiť tym, že si pre každé dvere spočítame pravdepodobnosť, že je za nimi výhra.

Predpokladajme, že máme n dverí. Takže keď si náhodne vyberieme dvere, pravdepodobnosť, že sme vybrali tie správne je $\frac{1}{n}$. Potom nám otvorí vedúci hry k dverí, čím nijako neovplyvní, či bola výhra za prvými vybratými dverami. Šanca, že sme ich vybrali správne je stále len $\frac{1}{n}$.

Zároveň však o k dverách vieme, že za nimi nie je výhra, inak povedané pravdepodobnosť, že za nimi je výhra, je 0. Vieme, že za nejakými dverami výhra musí byť, pretože súčet pravdepodobností pre všetkých n dverí dohromady je 1.

Ostalo nám $n-k-1$ dverí (všetky mínus otvorené mínus náš prvý typ), pre ktoré súčet pravdepodobností musí byť $1-k\cdot 0-\frac{1}{n}$ (všetky mínus otvorené mínus náš prvý tip). Každé z týchto $n-k-1$ dverí sú pre nás nerozlíšiteľné, teda majú z nášho pohľadu rovnakú pravdepodobnosť na výhru. Táto pravdepodobnosť je celková pravdepodobnosť predelená počtom dverí, čiže

$$P_{n,k}=\frac{1-\frac{1}{n}}{n-k-1}$$

$$P_{n,k}=\frac{n-1}{n(n-k-1)}$$

To je viac ako $\frac{1}{n}$ (ľahko porovnáme vzorky, ak ich prenásobíme spoločným menovateľom $n(n-k-1)$). Tak sme dokázali, že vždy je lepšie **v druhom kole svoje rozhodnutie zmeniť a teda vybrať iné dvere ako sme vybrali v prvom**.

Možností ako riešenie implementovať bolo viacero. Najjednoduchšie je vybrať si vždy najprv dvere s číslom 1 a potom postupne prechádzať dvere s vyššími číslami, až kým nenájdeme také, ktoré neboli otvorené (pozri implementáciu v C++ na konci vzorového riešenia). Dalo sa však aj postupovať tak, že si vytvoríme množinu dverí bez prvých, naraz odstránime z tejto množiny čísla, ktoré sme videli na vstupe a vypíšeme náhodný prvok zostávajúcej množiny (pozri implementáciu v Pythone na konci vzorového riešenia).

Pre $n=5$, $k=2$ je pravdepodobnosť, že náš program uhádne správne dvere $\frac{5-1}{5(5-2-1)}=\frac{4}{10}=0.4$.

Podúloha c)

Túto podúlohu vyriešime veľmi podobne ako podúlohu a), ale namiesto toho, aby sme našli správnu odpoveď dopredu, bude ju hľadať sám program po tom ako dostane vstupné dáta (počet mincí, hodnoty mincí a zvyšky).

Taktiež nechceme spočítať jeden najčastejší druhý zvyšok, ale chceme ho nájsť osobitne pre každý možný prvý zvyšok. V podstate chceme spočítať dvojrozmernú tabuľku (dvojrozmerné pole) `P[][]`, rozmerov $m_1\times m_2$, pričom na políčku `P[i][j]` bude počet možností, ako hodmi n mincí dosiahneme zvyšok i po delení m_1 a j po delení m_2.

Ak nás potom bude zaujímať, aký je najčastejší druhý zvyšok pre prvý zvyšok i, stačí nájsť najväčšiu hodnotu v i tom riadku tabuľky.

Ako čo najjednoduchšie vyskúšať všetky možnosti? Jedna možnosť je použiť rekurziu, ako je to v C++ riešení na konci. Aké sú možnosti hodu 0 mincami? No ak ničím nehodíme, dostaneme súčet nula, takže máme jednu mužnosť. Aké sú možnosti pri hode n mincami, pre $n>0$? No buď padne c_1, alebo c_2 a pre obe možnosti potrebujeme vyskúšať, ako sa dá hodiť $n-1$ mincami.

Druhá možnosť, ako sa dá vidieť v Pythonovom riešení, je postupne prejsť všetky čísla od 0 po 2^n-1. Ak si tieto čísla zapíšeme v binárnej sústave, dostaneme všetky možnosti, ako môže vyzerať posledných n bitov. Potom nám už len stačí namapovať si – ak i-ty bit od konca je 0, padla na i-tej minci prvá strana, inak padla druhá strana. Vo forcykli prejdeme všetky tieto čísla, každé si prevedieme na postupnosť hodov, spočítame jej súčet a jeho zvyšky po delení m_1 a m_2, a na pozíciu v poli pripočítame 1.

Vytváranie poľa P je relatívne pomalé – či už ide o rekurziu, alebo musíme prejsť všetky čísla od 0 po 2^n-1 a pre každé prejsť n bitov tohoto čísla. Z tohoto dôvodu ho spočítame len raz, a potom dokola využívame.

Bonus: Pre $n=7$, $c_1=1$, $c_2=2$, $m_1=3$, $m_2=4$ dostaneme takúto tabuľku `P`.

21	21	0	0
0	7	35	1
7	0	1	35

Z tabuľky vieme vyčítať, že v 42 zo 128 prípadoch je prvý zvyšok 0. Vtedy máme $\frac{21}{42} = 0.5$ šancu, že uhádneme. V pokiaľ je prvý zvyšok 1 alebo 2 (oba tieto prípady nastávajú s pravdepodobnosťou $\frac{43}{128}$) máme šancu $\frac{35}{43}$ uhádnuť. Celková šanca, že uhádneme je $\frac{21}{42} \cdot \frac{42}{128} + \frac{35}{43} \cdot \frac{43}{128} + \frac{35}{43} \cdot \frac{43}{128} = \frac{21+35+35}{128} = \frac{91}{128}$, čo je približne 0.71, teda až 71%. A to je výrazne väčšia šanca, ako keby sme tipovali zvyšok náhodne.

Listing programu (Python)

```
from random import randint, choice

def tipniA(pocet_stien):
    print(2*pocet_stien+2, flush=True)

def tipniB(pocet_dveri, kolko_otvorim):
    print(1)
    prazdne_dvere = [int(x) for x in input().split()]
    ostatne = list(set(list(range(2,pocet_dveri+1))) - set(prazdne_dvere))
    print(choice(ostatne), flush=True)

pripraveneP = None
def pripravP(n, hodnota1, hodnota2, modulo1, modulo2):
    global pripraveneP
    if pripraveneP: return pripraveneP
    pripraveneP = [[0 for i in range(modulo2)] for j in range(modulo1)]
    for mask in range(2**n):
        hodnota = 0
        for i in range(n):
            if (1<<i) & mask: hodnota += hodnota1
            else: hodnota += hodnota2
        pripraveneP[hodnota % modulo1][hodnota % modulo2] += 1
    return pripraveneP

def tipniC(n, hodnota1, hodnota2, modulo1, modulo2):
    pocty = pripravP(n, hodnota1, hodnota2, modulo1, modulo2)
    moznosti = pocty[int(input())]
    print(moznosti.index(max(moznosti)), flush=True)

hra = input()
parametre_hry = [int(x) for x in input().split()]
pocet_tipov = int(input())
tipovacia_funkcia = {'A':tipniA, 'B':tipniB, 'C':tipniC}[hra]
for t in range(pocet_tipov):
    tipovacia_funkcia(*parametre_hry)
```

Listing programu (C++)

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

void hrajA() {
    int n, testov, najlepsia = 0;
    cin >> n >> testov;
    // Zistim si ako casto pada ktora moznost
    vector<int> pocetMoznosti(4*n+5, 0);
    for(int a=1; a<=n; ++a)
        for(int b=1; b<=n; ++b)
            for(int c=1; c<=n; ++c)
                for(int d=1; d<=n; ++d)
                    pocetMoznosti[a+b+c+d]++;
    // Vyberiem najlepsiu
    for(int i=0; i<pocetMoznosti.size(); ++i)
        if (pocetMoznosti[i] > pocetMoznosti[najlepsia])
            najlepsia = i;
    for(int i=0; i<testov; ++i)
        cout << najlepsia << endl;
}

void hrajB() {
    int n, k, testov, dvere;
    cin >> n >> k >> testov;
    for(int i=0; i<testov; ++i) {
        cout << 1 << endl;
        vector<bool> suOtvorene(n+1, false);
        // Zistim, ktore dvere su otvorene
        for(int j=0; j<k; ++j) {
            cin >> dvere;
            suOtvorene[dvere] = true;
        }
        // Najdem prve zavrete dvere s cislom aspon 2
        dvere = 2;
        while(suOtvorene[dvere]) dvere++;
        cout << dvere << endl;
    }
```

```
}

typedef vector<int> vi;
typedef vector<vi> vii;
vii P;
int n, c1, c2, mod1, mod2, testov;

void vyskusajHody(int n, int doterajsiSucet) {
    if (n == 0) {
        P[doterajsiSucet % mod1][doterajsiSucet % mod2]++;
        return;
    }
    vyskusajHody(n-1, doterajsiSucet + c1);
    vyskusajHody(n-1, doterajsiSucet + c2);
}

void hrajC() {
    cin >> n >> c1 >> c2 >> mod1 >> mod2 >> testov;
    P = vii(mod1, vi(mod2, 0));
    // Spocitam si spravne odpovede
    vyskusajHody(n, 0);
    for(int i=0; i<testov; ++i) {
        int zvysok1, najlepsi = 0;
        cin >> zvysok1;
        // Najdem najlepsi zvysok2
        for(int j = 0; j<mod2; ++j) {
            if (P[zvysok1][j] > P[zvysok1][najlepsi])
                najlepsi = j;
        }
        cout << najlepsi << endl;
    }
}

int main() {
    char poduloha;
    cin >> poduloha;
    switch(poduloha) {
        case 'A': hrajA(); break;
        case 'B': hrajB(); break;
        case 'C': hrajC(); break;
    }
}
```

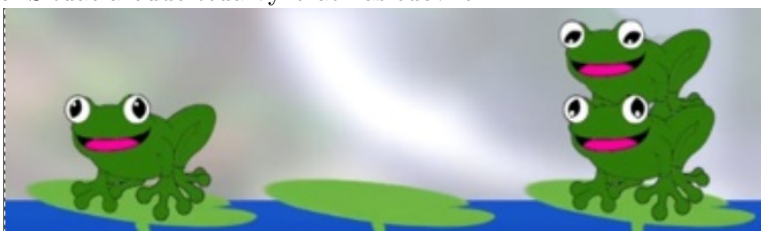
4. Párty Žiab

vzorák napísal Paulínka
(max. 15 b za riešenie)

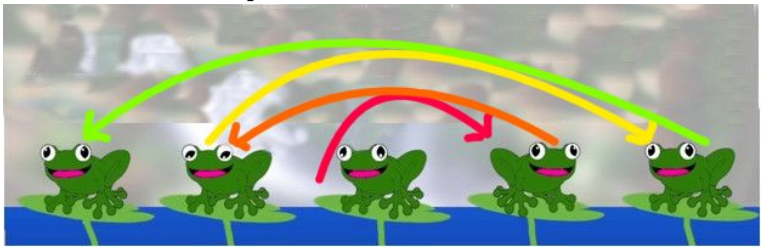
Riešenie bez žaby Michala

Pozrime sa najprv na ľahšiu úlohu, v ktorej sa žaba Michal nenachádzal. Našou úlohou je teda iba zhromaždiť všetky žaby na jednom lekne.

Na začiatok sa pokúsme vyriešiť úlohu s malým počtom žiab. Napríklad, ak máme iba jednu žabu, úloha je ľahká, pretože všetky (jedna) žaby sú spolu na jednom lekne. Pridajme teda aj druhú žabu. Problém je stále jednoduchý, stačí ak žaba z prvého lekna skočí na druhé lekno a môže začať párty. Do takejto situácie pridajme na začiatok tretie lekno. Situácia bude teda vyzerať nasledovne.



Všimnime si, že keďže na pravom lekne sedia dve žaby, vedia skočiť na vzdialenosť 2. Čím skočia presne na prvé lekno, kde ich čaká tretia žaba. Tieto tri žaby teraz vedia skákať skokom dĺžky 3 a keď na pravej strane pridáme štvrtú žabu, tá bude od nich vzdialená presne o 3 lekná. Potom vieme naľavo pridať piatu žabu, ktorá je vzdialená o 4 lekná a takto nastriedačku pokračovať ako dlho budeme chcieť.



Postup skákania je teda pomerne jednoduchý. Začneme so žabou, ktorá sedí na strednom lekne. Tá skočí na susedné lekno, následne obe žaby skočia opačným smerom na lekno vo vzdialenosti 2, kde sa stretnú s treťou

žabou, opäť zmenia smer a skočia na lekno, vzdialené 3. Po každom skoku teda zmenia smer, ktorým skáču a veľkosť ich skoku sa zväčší o 1.

Pozor si musíme dať iba v prípade, keď je žiab párne veľa. V takom prípade sú totiž dve stredné žaby. Môžeme si vybrať ľubovoľnú z nich, potom si však musíme dať pozor, aby prvý skok išiel do správneho smeru – tam kde je viac žiab.

Takéto riešenie vieme implementovať pomerne jednoducho. Budeme si pamätať, na ktorom lekne stojí naša kôpka žiab (na začiatku je to lekno $\frac{n+1}{2}$), koľko ich na tomto lekne je (na začiatku 1), a teda aký veľký skok vedia robiť a smer, ktorým majú skákať. Smer skoku v podstate určuje, či chceme pri počítaní lekna, na ktoré skočia dĺžku ich skoku pričítavať alebo odčítavať. Na to môžeme použiť buď hodnoty `true` a `false` alebo aj 1 a -1.

Riešenie so žabom Michalom

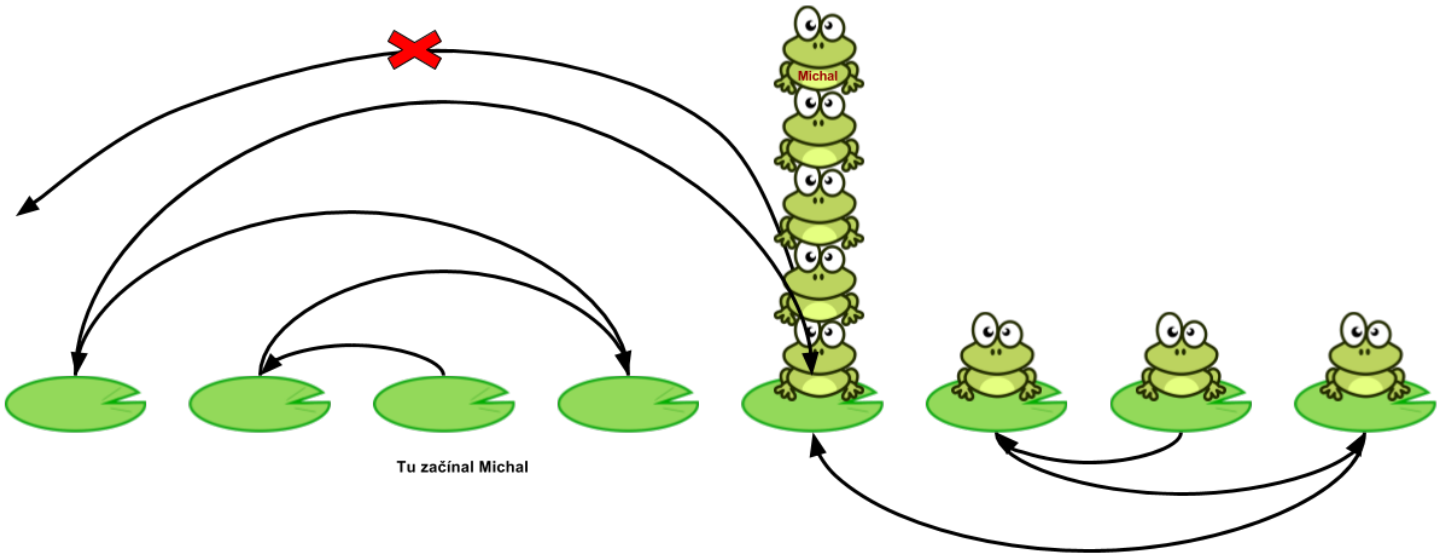
Pribudla nám žaba Michal, ktorý nechce, aby po ňom zvyšné žaby skákali. Skúsme sa zamyslieť nad tým, ako nám môže pomôcť riešenie ľahšej podúlohy. Existuje v pôvodnom riešení žaba, na ktorú nikto nikdy neskočí? Áno, žaba ktorou sme začínali, teda sa nachádzala v strede, bude neustále navrchu. Ak by teda žaba Michal bol v strede, úlohu by sme vedeli riešiť tým istým spôsobom.

Ako však vyriešiť prípady, keď Michal nebude v strede? Predstavme si, že Michal sedí na úplne prvom lekne. Vynechajme na zatiaľ toto lekno a pozrime sa na zvyšných $n-1$. Medzi nimi sa Michal nenachádza, takže ak chceme, aby sa tieto žaby stretli, môžeme použiť riešenie ľahšej podúlohy. Naviac si všimnime, že použitím tohto postupu skončia žaby na jednom z krajných leknien, pretože skáču od stredu do krajov. A naviac, vieme ovplyvniť aj to, na ktorom kraji skončia. Stačí ak správne zvolíme smer prvého skoku – pre nepárne n, ak chceme skončiť na ľavom kraji, musíme najskôr skákať doprava, v opačnom prípade musíme najskôr skočiť doľava.

Z tohto pozorovania nám ale vyplýva, že zvyšných $n-1$ žiab vieme zhromaždiť na ľavom krajnom lekne, na lekne, ktoré je druhé v poradí, lebo na prvom sedí Michal. A keď sú už na druhom lekne všetky zvyšné žaby, Michal na nich jednoducho vyskočí.

A čo sa stane ak bude Michal na druhom lekne? Potom jedna možná postupnosť skokov je najskôr dostať $n-2$ žiab, ktoré sú napravo na tretie lekno v poradí. Toto vieme spraviť bez problémov, lebo medzi nimi Michal nie je. Následne Michal skočí na prvé lekno a spolu so žabou, ktorá sa na ňom nachádza skočia na lekno tri, ktoré je vzdialené 2.

Nuž a teraz si riešenie zovšeobecnime. Zoberme žabu Michala a začnime s ním skákať nastriedačku do oboch strán tak ako v prvom riešení. V nejakom momente budeme musieť zastať, pretože Michal (so všetkými žabami pod ním) nebude môcť skočiť ďalej, lebo by skočil mimo lekien. To ale znamená, že smerom, ktorým chce skočiť už na leknách nie sú žiadne ďalšie žaby. Všetky pozbieral predchádzajúcimi skokmi. A naviac, v opačnom smere sa na každom lekne nachádza jedna žaba, pretože to sú lekná, na ktoré Michal ešte nestihol skočiť.



My však cheme všetky žaby na jednom lekne. Na tento úsek lekien, na ktorých sedia žaby (vrátane lekna s Michalom a žabami pod ním) vieme použiť predchádzajúci algoritmus, pričom budeme chcieť, aby sa žaby stretli na lekne, na ktorom stojí Michal. Tu však nastane problém, pretože potom predsa nejaké žaby skočia na Michala.

Uvedomme si však, že nám na poradí, v ktorom tieto skoky budeme robiť nezáleží. Môžeme teda najskôr nechať skákať žaby, ktoré Michal počas svojich skokov nestretne a až potom začať skákať s Michalom. Takto sa

všetci stretnú na spoločnom lekne a Michal bude navrchu.

Ostáva nám takéto riešenie naprogramovať. Jeden možný spôsob je skákať s Michalom a ukladať si skoky, ktoré robí. Pomocou toho zistíme, na ktorom lekne sa majú žaby stretnúť, odsimulujeme skoky ostatných žiab, pričom ich skoky naozaj vypisujeme a až potom vypíšeme skoky s Michalom.

Druhá možnosť je dopredu vypočítať, na ktorom lekne sa žaby stretnú. To vieme robiť aj jednoduchým skúšaním každého lekna. Pre toto lekno totiž bude musieť platiť, že Michal sa na začiatku nachádza v strede medzi ním a jedným z krajov jazera. Následne vieme simulovať obe skákania.

V oboch prípadoch bude časová zložitosť takéhoto riešenia $O(n)$ a pamäť vie byť pri správnej implementácii dokonca konštatná.

Listing programu (Python)

```
#!/usr/bin/env python3
n, m = map(int, input().split())

m = max(m, -m) # ak m = -1 tak sa budeme tvarit, ze m = 1, lebo sa nam nechce pisat podmienku
m -= 1 # cislujeme od nuly

print("ANO")

if n == 1:
    # ak je len jedna zaba, uz je party
    exit()

prikazy = []
prikazyM = []

kde = m
smer = -1
dlzka = 1
if m < n - m - 1:
    smer = 1

while kde + smer * dlzka < n and kde + smer * dlzka >= 0:
    prikazyM.append("{} {}".format(kde + 1, kde + smer * dlzka + 1))
    kde = kde + smer * dlzka
    dlzka += 1
    smer *= -1

if dlzka == n:
    # michal uskakal vsetky
    print("\n".join(prikazyM))
    exit()

kdeparty = kde

dlzka = 1
if m > kde:
    if kde % 2 == 1:
        smer = 1
    else:
        smer = -1
    kde = kde // 2
else:
    if (n - kde) % 2 == 1:
        smer = 1
    else:
        smer = -1
    kde = (n - kde) // 2 + kde

while kde != kdeparty:
    prikazy.append("{} {}".format(kde + 1, kde + smer * dlzka + 1))
    kde = kde + smer * dlzka
    dlzka += 1
    smer *= -1

print("\n".join(prikazy))
print("\n".join(prikazyM))
exit()
```

5. Poriadnejšia výza

vzorák napísal Andrej
(max. 15 b za riešenie)

Vzorové riešenie, ako aj väčšina vašich, sú založené na postupnom dopĺňaní jednotlivých políčok. Samotné sady sa líšia iba zložitosťou algoritmu, ktorý na dopĺňanie musíme použiť.

Prvá vec, ktorú musíme vyriešiť je, ako budeme sudoku reprezentovať v pamäti. Z dostupných možností, je najlepším riešením použiť dvojrozmerné pole. Takéto pole vieme indexovať a pristupovať na jeho jednotlivé políčka dvomi číslami (a, b), kde a je číslo riadku a b číslo stĺpca daného políčka. Prvý riadok a stĺpec má poradové číslo 0 a teda hovoríme, že indexujeme od nuly. Z technických príčin budeme používať jeho inteligentnú implementáciu a to `vector`.

Sada 1, 2

Pustime sa teraz do prvých dvoch sád. Ako sa ukáže, ich riešenie je veľmi podobné. Sudoku budeme v oboch prípadoch prechádzať zľava doprava a zhora dole. Ak narazíme na prázdne políčko, musíme ho doplniť. V prvej sade je toto políčko jediné, ktoré v riadku chýba. Stačí sa nám teda pozrieť na to, ktoré čísla sú v danom riadku doplnené a zaznačiť si ich do pomocného poľa. Po prejdení celého riadku bude existovať iba jediné číslo medzi 1 a 9, ktoré sme nezaznačili ako nájdené. Je zrejmé, že to bude práve to hľadané číslo, ktoré chceme doplniť. Tento postup zopakujeme pre každé prázdne políčko a postupne vyplníme celé sudoku.

No a pri riešené druhej sady vieme robiť to isté, akurát pri zaznačovaní čísel budeme prechádzať stĺpec a nie riadok.

Predstavme si, že sa nachádzame na políčku (a, b), ktoré ešte nie je doplnené, teda sa na ňom nachádza 0. Ktoré políčka sa nachádzajú v rovnakom riadku? Sú to všetky políčka s rovnakou prvou súradnicou označujúcou riadok, teda s rovnakým a: $(a, 0)$, $(a, 1)$, $(a, 2)$ $\ldots$ $(a, 8)$. V našom programe potom prejdeme všetkými týmito políčkami a do pomocné poľa `pouz[]` si zaznačíme, ktoré čísla sú v riadku a doplnené. Z tohto poľa potom ľahko vyčítame, ktoré číslo v tomto riadku chýba.

Listing programu (C++)

```
#include<iostream>
#include<vector>//using kvoli vectoru
using namespace std;

int main()
{
    vector<vector<int> > vstup(9, vector<int>(9));//vytvorime dvojrozmerny vektor
    // v skutocnosti sa sprava velmi podobne ako pole a mohli by sme tuto deklarciu
    // prepisat aj na :   "int vstup[9][9];"

    for(int i=0;i<9;++i)for(int j=0;j<9;++j)
    {
        cin >> vstup[i][j];//nacitame vstup
    }

    for(int i=0;i<9;++i)for(int j=0;j<9;++j)if(vstup[i][j] == 0)
    {//narazili sme na prazdne policko na pozicii [i;j]

        bool pouz[10] = {false};//tu si pamatame, ktore cisla uz sme pouzili/videli
        //resp. su doplnene
        //na zaciatku je pole plne false

        for(int k=0;k<9;++k)pouz[vstup[i][k]] = true;
        //for(int k=0;k<9;++k)pouz[vstup[k][j]] = true;//pouzijeme v pripade stlpca

        for(int k=0;k<10;++k)if(pouz[k] == false)//ak je k nepouzite
        {
            vstup[i][j]=k;//doplnime ho a skoncime
            break;
        }
    }

    //a uz len doplnene sudoku cele vypiseme
    for(int i=0;i<9;++i)
    {
        for(int j=0;j<9;++j)cout << vstup[i][j] << "␣";
        cout<<"\n";
    }
    return 0;
}
```

Sada 3

Napriek tomu, že problém je skoro rovnaký ako v prvých dvoch sadách, implementácia je o niečo zložitejšia. Postup, ktorý môžeme zvoliť je, že budeme jednotlivé bloky vypĺňať postupne. Vždy vezmeme čísla v jednom bloku a zistíme, ktoré číslo v danom bloku chýba. Potom ho doplníme namiesto 0. Ako to však naimplementovať?

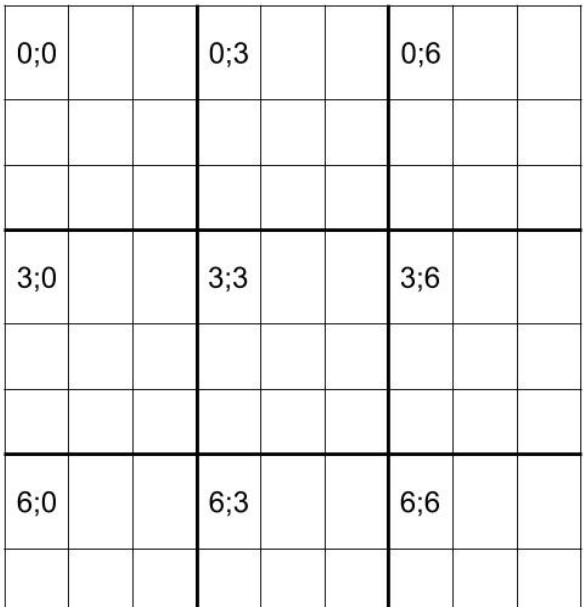
Predstavme si algoritmus, ktorý zrejme dokáže väčšina z vás bez problémov naprogramovať. Spočíva v tom, že doplní chýbajúce číslo v prvom bloku 3×3. Najskôr prejdeme menší štvorec 3×3 a zaznačíme si, tak ako v predchádzajúcich príkladoch, už doplnené čísla a tiež ktoré políčko v bloku nie je doplnené. Zo zadania vieme, že bude iba jedno. Na konci raz prejdeme zoznamom čísel a zistíme, ktoré z nich nebolo použité. Toto číslo doplníme na súradnice, na ktorých leží v danom bloku číslo 0. A keďže sa pozeráme iba na prvý blok 3×3, vieme, že prvý aj druhý index každého políčka v tomto bloku je menší ako 3. Implementácia preto môže vyzerať nasledovne:

Listing programu (C++)

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

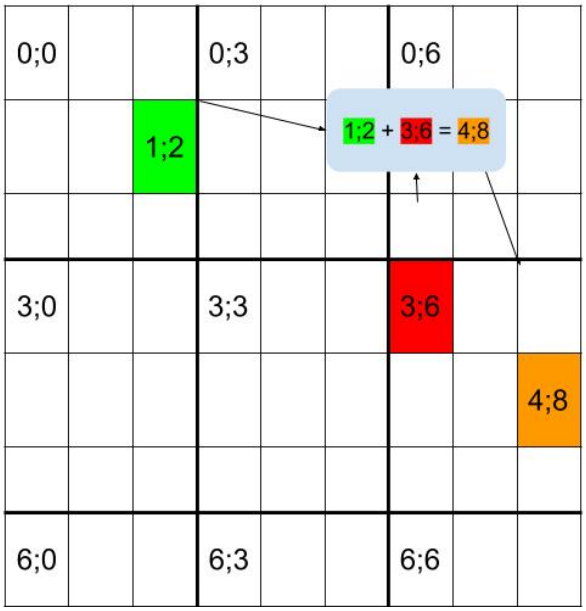
```
int main()
{
    vector<vector<int> > vstup(9, vector<int>(9));//vytvorime dvojrozmerne pole

    for(int i=0;i<9;++i)for(int j=0;j<9;++j)
    {
        cin >> vstup[i][j];//nacitame vstup
    }

    int a, b;//tu si pamatame indexy nedoplneneho policka
    vector<bool>pouz(10, false);//tu ake cisla sme videli v nasom bloku

    for(int k=0;k<3;++k)for(int l=0;l<3;++l)//riesime len prvy maly stvorcek 3x3
    {
        if(vstup[k][l] == 0)//ak je policko prazdne, zapametame si indexy do a, b
        {
            a = k;
            b = l;
        }
        pouz[vstup[k][l]] = true;//kazdopadne si ho zaznacime
    }

    for(int f=0;f<10;++f)if(!pouz[f])//ak nie je pouzite, vieme, ze je jedine
    {
        vstup[a][b]=f;//doplnime ho
        break;
    }

    for(int i=0;i<9;++i)//uz len vsetko vypiseme
    {
        for(int j=0;j<9;++j)cout << vstup[i][j] << "␣";
        cout<<"\n";
    }

    return 0;
}
```

Ako nám to však pomôže pri dopĺňaní ostatných blokov? Pozrime sa na indexy políčok v ľavom hornom rohu každého bloku.



Rýchlo uvidíme istú zákonitosť. Ľavé horné políčko každého bloku má indexy v tvare $(0+3k, 0+3l)$, kde k a l je menšie ako 3.

Všimnime si, čo sa stane ak k políčku v prvom bloku pripočítame indexy nejakého z týchto rohových políčok.



Uvidíme, že nás toto sčítanie posunulo na rovnaké políčko, akurát v inom bloku. Presnejšie v tom bloku, ktorého ľavé horné políčko bolo pri sčítavaní použité. V našom riešení teda vieme postupne prejsť týchto 9 políčok a v každom bloku sa budeme tváriť akoby sme pracovali s blokom prvým, akurát budeme o niečo posunutí.

V prvom bloku pracujeme s políčkami $(0,0)$, $(0,1)$, $(0,2)$, $(1,0)$ ... až $(2,2)$. A napríklad v strednom bloku pracujeme s tými istými políčkami, avšak jednotlivé indexy sú posunuté o indexy ľavého horného políčka stredného bloku, teda robíme na indexoch $(0+3, 0+3)$ až $(2+3, 2+3)$.

Pomocou tejto úvahy sa vieme pohodlne presúvať po jednotlivých blokoch, čo vedie k nasledujúcej implementácii.

Listing programu (C++)

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

int main()
{
    vector<vector<int> > vstup(9, vector<int>(9));

    for(int i=0;i<9;++i)for(int j=0;j<9;++j)
    {
        cin >> vstup[i][j];//nacitame vstup
    }

    for(int i=0;i<9;i+=3)for(int j=0;j<9;j+=3)
    {
        //cout<< i <<" "<< j <<endl;//vypise suradnice laveho horneho policka
        int a, b;
        vector<bool>pouz(10, false);

        for(int k=0;k<3;++k)for(int l=0;l<3;++l)//tvarime sa, ze riesime prvy stvorcek
        {
            //v skutocnosti sa ale nachadzame na vstup[i+k][j+l] a nie na vstup[k][l]
            if(vstup[i+k][j+l] == 0)//ak je policko prazdne, zapametame si indexy do a, b
            {
                a = i+k;//v a, b uz su skutocne suradnice policka, ktore nie je
                b = j+l;//doplnene
            }
            pouz[vstup[i+k][j+l]] = true;//kazdopadne si ho zaznacime
        }

        for(int f=0;f<10;++f)if(!pouz[f])
        {
            vstup[a][b]=f;
            break;
        }
    }

    for(int i=0;i<9;++i)
    {
        for(int j=0;j<9;++j)cout << vstup[i][j] << " ";
        cout<<"\n";
    }
    return 0;
}
```

Sada 4

Riešenie štvrtej sady je zhrnutím toho, čo sme sa v tomto vzorovom riešení zatiaľ naučili. Sudoku budeme prechádzať tak ako doteraz, až pokým nebude úplne doplnená, avšak samotné dopĺňanie bude o čosi zložitejšie. Zadanie jasne hovorí, že niektoré z políčok je ľahko doplniteľné, ale nehovorí ktoré. Toto je problém, na ktorý si napíšeme pomocnú funkciu `je_jednoznacne(a,b)`. Táto funkcia na vstupe dostane naše sudoku a pozíciu jedného prázdneho políčka (a, b). Pre toto políčko vráti `true`, ak je práve toto políčko jednoznačne doplniteľné a `false` ak nie je.

Táto funkcia bude robiť to, že si vytvorí pole doplnených čísel a toto pole použije pri prejdení riadku, stĺpca aj bloku, v ktorom leží zadané políčko. Tým vlastne zistí, ktoré čísla nemôžu byť doplnené na pozíciu, ktorej jednoznačnosť overuje. Ak nám na doplnenie ostane viacero možností, nevieme políčko jednoznačne doplniť tak ako hovorí zadanie. Ak nám však ostala jediná možnosť, vieme políčko jednoznačne doplniť. Hodnotu, ktorú na toto políčko vieme vložiť vieme zistiť pomocou nášho poľa, my si však vytvoríme ešte jednu funkciu – `je_mozne(a,b,k)`.

Táto funkcia slúži ako pomocná a jej existenciu naplno doceníme pri piatej sade. Funkcia na vstupe dostane naše sudoku, súradnice políčka (a, b) a jedno číslo k. Na výstupe nám vráti `true`, ak môžeme číslo k doplniť na políčko s indexami (a, b) a v opačnom prípade vráti `false`. **Pozor**, táto funkcia nám nehovorí, či je správne doplniť číslo k na túto pozíciu. Hovorí iba to, či ho tam **môžeme** doplniť vzhľadom na aktuálny stav sudoku tak, aby sme neporušili žiadne pravidlo. Hodnotu `true` teda vráti, ak sa číslo k nenechádza ani v riadku a, ani v stĺpci b a ani v bloku, v ktorom leží (a, b).

Problém nastáva, keď chceme zistiť, v ktorom bloku sa nachádza políčko so súradnicami (a, b). Možností ako toto riešiť je viac, my použijeme asi tú najjednoduchšiu. Uvedomme si, že blok vieme jednoznačne určiť podľa rozsahu hodnôt, v ktorom sa nachádza a a b. Keďže vieme, aký je blok veľký, na jeho prejdenie nám stačí už len vedieť, kde máme začať.

Už dávnejšie sme prišli na to, že ľavé horné políčka v každom bloku majú zaujímavé súradnice, ktoré sú deliteľné trojkou. Pokúsme sa to nejak využiť. Ak sa pozrieme napríklad na stĺpec, tak sa v ňom nachádza 9 políčok, ktoré sú rovnomerne rozložené do 3 blokov. Ako zo súradnice zistíme, v ktorom bloku sa nachádza? Predsa ju vydelíme trojkou. Poďme si to overiť. Čísla 0, 1, 2 po delení 3 naozaj vrátia 0. Čo to znamená je, že sú v jednom z prvých blokov. Čísla 3, 4, 5 vrátia 1, čo o nich hovorí, že sú v jednom zo stredných blokov. Ak toto urobíme aj pre druhú súradnicu, vieme jednoducho určiť v ktorom bloku sa nachádza. Tak napríklad políčko $(5, 7)$ sa nachádza v $5/3 = 1$ bloku v rámci riadku a v $7/3 = 2$ v rámci stĺpca, pričom indexujeme od nuly. Ak sa teraz pozrieme, kde sa naozaj políčko nachádza, zistíme, že presne tam, kde sme určili.

Ako toto využiť? Pri celočíselnom delení sa zahadzuje zvyšok po delení, čo znamená, že napríklad výraz $(x/3) \cdot 3 \neq x$. Tento výraz sa nebude zakaždým rovnať pôvodnému číslu, ale prvému menšiemu alebo rovnému násobku 3. Čo sa teraz stane, keď index "bloku" vynásobíme opäť 3? Stane sa to, že sa číslo bloku zase zmení na číslo nejakého políčka a to práve ľavého horného prislúchajúceho bloku. Na získanie ľavého horného rohu bloku, v ktorom sa nachádza políčko (a, b) potrebujeme teda obe súradnice a aj b vydeliť a potom opäť prenásobiť 3.

Keď máme tieto dve funkcie napísané, vieme už túto sadu riešiť pomerne jednoducho. Sudoku prechádzame stále odznova, pokým nie je doplnená. V prípade, že narazíme na políčko, ktoré je prázdne, skontrolujeme, či ho môžeme jednoznačne doplniť. Ak áno, prejdeme cez všetky možné čísla, pričom vieme, že pre to jediné, ktoré môžeme na toto miesto doplniť, nám funkcia `je_mozne()` vráti `true`.

Listing programu (C++)

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

bool je_mozne(const vector<vector<int> >&sudoku, int a, int b, const int &k)
{
    for(int i=0;i<9;++i)if(sudoku[a][i] == k)return false;//prejdeme riadok
    for(int i=0;i<9;++i)if(sudoku[i][b] == k)return false;//prejdeme stlpec

    //prejdeme blok
    int m=(a/3)*3, n=(b/3)*3;
    for(int i=m;i<m+3;++i)for(int j=n;j<n+3;++j)if(sudoku[i][j] == k)return false;

    return true;
}

 bool je_jednoznacne(const vector<vector<int> >&sudoku, int a, int b)
{
    int pom=0;
    vector<bool>nachadza(10, false);

    for(int i=0;i<9;++i)nachadza[sudoku[a][i]]=true;
    for(int i=0;i<9;++i)nachadza[sudoku[i][b]]=true;

    int m=(a/3)*3, n=(b/3)*3;
    for(int i=m;i<m+3;++i)for(int j=n;j<n+3;++j)nachadza[sudoku[i][j]]=true;
```

```
    for(int i=1;i<10;++i)if(nachadza[i] == false)pom++;

    if(pom == 1)return true;
    return false;
}

int main()
{
    vector<vector<int> >sudoku(9, vector<int>(9));
    int nedoplnene = 0;

    for(int i=0;i<9;++i)for(int j=0;j<9;++j)
    {
        cin >> sudoku[i][j];
        if(sudoku[i][j] == 0)nedoplnene++;
    }

    for(int f=0;f<nedoplnene;++f)
    {
        for(int i=0;i<9;++i)for(int j=0;j<9;++j)if(sudoku[i][j] == 0)
        {
            if(je_jednoznacne(sudoku, i, j))
            {
                for(int k=1;k<10;++k)if( je_mozne(sudoku, i, j, k) == true)
                {
                    sudoku[i][j]=k;
                    break;
                }
            }
        }
    }

    for(int i=0;i<9;++i)
    {
        for(int j=0;j<9;++j)cout << sudoku[i][j] << "␣";
        cout<<"\n";
    }
    return 0;
}
```

Pokiaľ vás v tejto implementácii mätie, prečo sme funkcii nedali parametre takto `vector<vector<int> > sudoku`, ale takto `const vector<vector<int> > &sudoku`, čítajte ďalej. Pokiaľ by sme použili prvý spôsob, C++ by predpokladalo, že pôvodné pole nechceme meniť a preto by vytvorilo jeho kópiu, všetky zmeny čo sme vykonali vo funkcii by sa teda po konci funkcie zahodili, pretože toto skopírované pole by prestalo existovať. Ak však použijeme operátor `&`, hovoríme tým počítaču, že chceme aby pole poslalo ako referenciu, teda poslalo do funkcie pôvodné pole a my sme mohli zmeny vykonávať priamo v ňom. Toto je ešte dôležitejšie pri veľkých poliach, pri ktorých kopírovanie zaberá dlhý čas, čo by mohlo spôsobiť problémy.

Nakoniec už len vysvetliť kľučové slovo `const`. Ak ho použijeme pred nejakou premennou alebo poľom, hovoríme počítaču, že túto premennú/pole/ vector **nechceme meniť**. Chceme z nej iba načítavať alebo sa na ňu pozerať. Používať túto konvenciu vrelo odporúčam, pretože počítač nás pri nechcenej zmene takejto premennej upozorní a nespustí, resp. neskompiluje náš program.

Sada 5

Zatiaľ sme v podstate neskúsili poriadny bruteforce. Ten by nás však veľmi nepomohol, keďže možných sudoku je vzhľadom na naše pomery takmer nekonečne veľa. Avšak práve táto myšlienka nás teraz privedie k vzorovému riešeniu.

Predstavme si takýto algoritmus. Máme funkciu `vyries()`, ktorá na vstupe dostane sudoku a vráti nám `true` ak ju vedela vyplniť. Ako by takáto funkcia asi vyzerala? Použijeme rekurziu[2].

Prvý prípad je, že funkcia sa pozrela na zadané sudoku a nenašla v nej žiadne voľné políčka. Vtedy je sudoku zjavne vyriešené a funkcia vráti `true` a skončí. Druhý prípad je, že aspoň jedno políčko je nevyplnené. Čo táto funkcia urobí je, že prejde všetky možnosti, čo doplniť na toto voľné miesto a pri prvom čísle, ktoré nebude porušovať pravidlá sudoku, ho skúsi doplniť.

Takýmto splsobom dostane sudoku, ktoré má o jedno voľné políčko menej a môže opäť skúsiť zavolať funkciu `vyries()`. Predstavme si, že je táto funkcia ďalej úspešná a vráti `true`. Doplnilnili sme teda jedno číslo a rekurzívna funkcia ďalej zistila, že takúto sudoku vieme riešiť a toto číslo tam naozaj patrilo. V takom prípade dostaneme vyriešené sudoku.

Samozrejme, môže nastať aj druhý prípad, keď po doplnení čísla vráti rekurzívna funkcia `vyries()` hodnotu `false`. To znamená, že číslo, ktoré sme doplnili, priviedlo sudoku do nevyriešiteľného stavu. Toto číslo tam teda nepatrilo. Musíme preto na danú pozíciu skúsiť doplniť iné číslo a rekurzívnym volaním opäť overiť, či takto upravenú sudoku vieme vyriešiť.

No a v momente, keď sme pre voľné políčko vyskúšali všetky možnosti a zakaždým sme sa dostali do nevyriešiteľného stavu vieme, že táto sudoku nemôže byť správna, lebo sa nedá žiadnym spôsobom vyriešiť.

[2] https://www.ksp.sk/kucharka/rekurzia/

Vrátime preto hodnotu `false`.

Tento postup, ktorý sme použili sa v skutočnosti vyskytuje v programovaní dosť často. Nazýva sa *backtracking* a používa sa v situáciách, keď prechádzame postupne všetky možnosti, pričom dúfame, že zlé možnosti si všimneme dostatočne skoro a nebudeme sa ich kontrolovaním vôbec zaťažovať.

Backtracking prebieha v niekoľkých fázach. Máme nejakú rekurzívnu funkciu, ktorá najprv dostane reprezentáciu problému, v našom prípade pole s uloženým sudoku. Pokúsi sa v riešení problému postúpiť ďalej tým, že niečo v poli doplní, čím zmenší pôvodný problém a zavolá sama seba na toto nové pole. Ak sa jej nepodarí nájsť riešenie, vráti pole do stavu v akom ho dostala a vráti `false`. V opačnom prípade, ak sa niektorej z vetiev podarilo nájsť riešenie, teda táto vetva vrátila `true`, aj naša funkcia vráti `true` a ďalšie možnosti už neskúša.

Listing programu (C++)

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

bool je_mozne(const vector<vector<int> >&sudoku, int a, int b, const int &k)
{
    for(int i=0;i<9;++i)if(sudoku[a][i] == k)return false;
    for(int i=0;i<9;++i)if(sudoku[i][b] == k)return false;

    int m=(a/3)*3, n=(b/3)*3;
    for(int i=m;i<m+3;++i)for(int j=n;j<n+3;++j)if(sudoku[i][j] == k)return false;

    return true;
}

bool vyries(vector<vector<int> >&sudoku)//rekurzivna funkcia riesiaca sudoku
{
    for(int i=0;i<9;++i)for(int j=0;j<9;++j)if(sudoku[i][j] == 0)
    {
        for(int k=1;k<=9;++k)if( je_mozne(sudoku, i, j, k) )//mozme na toto policko dat k?
        {
            sudoku[i][j] = k;//skusime ho doplnit
            if( vyries(sudoku) )return true;//ak tam patrilo vratime sa
            sudoku[i][j] = 0;//inak tam zas dame nulu
        }

        if(sudoku[i][j] == 0)return false;//ak sme ho nevedeli vyplnit nijak, vratime sa
    }
}

int main()
{
    vector<vector<int> >vstup(9, vector<int>(9));

    for(int i=0;i<9;++i)for(int j=0;j<9;++j)cin >> vstup[i][j];

    vyries(vstup);

    for(int i=0;i<9;++i)
    {
        for(int j=0;j<9;++j)cout << vstup[i][j] << "␣";
        cout<<"\n";
    }
    return 0;
}
```

Časová zložitosť tohto prístupu nie je nijak oslnivá. Uvedomme si, že sa tu dosť spoliehame na to, že nedôjde k najhoršiemu možnému scenáru, teda vyskúšaniu všetkých možností. To sa totiž naozaj môže stať. Nevieme prečo, ale ukazuje sa, že k takejto situácii prichádza málokedy, skoro až nikdy. Všeobecnejšia verzia tohto problému, kde riešime sudoku rozmerov $n \times n$, sa ukazuje byť tažká, presnejšie NP-úplná.

Čo to znamená, že je problém NP-úplný? Detailami a teóriou vás nudiť nebudeme, stačí nám vedieť, že NP-úplné problémy sú tie, na ktoré ešte neboli vyvinuté efektívne algoritmy. V preklade to znamená, že najlepší algoritmus, ktorý sme zatiaľ vymysleli stále negarantuje, že nevyskúša všetky možnosti. V priemernom prípade si môže tento algortimus počínať výborne, avšak v najhoršom stále len vyskúša všetky možnosti, čo je práve prípad aj našeho algoritmu.

Napriek tomu, ak sa vám podarí vytvoriť algoritmus, ktorý bude takýto problém riešiť a zároveň garantovať, že nevyskúša v najhoršom prípade všetky možnosti, neváhajte nám ho poslať! Získate prvé miesto v PRASKU a aj odmenu 1 000 000 dolárov, ktorá na vyriešenie tohto problému bola vypísaná. Platí totiž, že ak sa nájde algoritmus riešiaci len jeden z NP-úplných problémov, bude sa s miernymi zmenami dať použiť na vyriešenie všetkých týchto problémov.

V prípade, že ste niečomu vo vzoráku nepochopili alebo vás inšpiroval k vyriešeniu NP-úplného problému, neváhajte ma kontaktovať na **ajo@ksp.sk**. S výhrou aj vašimi nejasnosťami sa rád podelím.