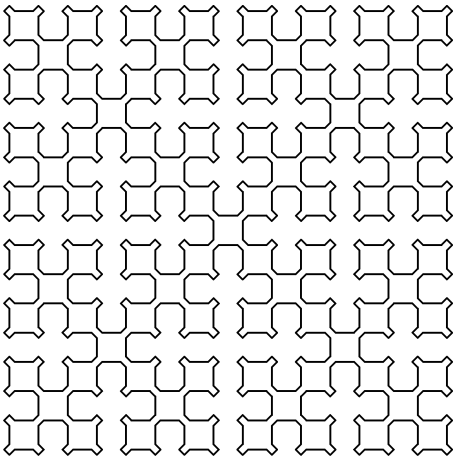
MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

X. ročník – 1997/98



Matematicko-fyzikální fakulta
University Karlovy

MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

X. ročník – 1997/98

Matematicko-fyzikální fakulta
University Karlovy

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož desátý ročník se vám dostává do rukou, patří k nejznámějším aktivitám pořádaným MFF UK pro zájemce o informatiku a programování z řad studentů středních škol. Řešíce úlohy našeho semináře, středoškoláci získávají praxi ve zdolávání nejrůznějších algoritmických problémů, jakož i hlubší náhled na mnohé discipliny informatiky. Proto některé úlohy *KSP* svou obtížností vysoko přesahují rámec běžného středoškolského vzdělání, a tudíž i požadavky při přijímacím řízení na vysoké školy, MFF UK z toho nevyjímaje. To ovšem vůbec neznamená, že nemá smysl takové problémy řešit – při troše přemýšlení není příliš obtížné nějaké (i když někdy ne to nejlepší) řešení nalézt. Nakonec – posuďte sami.

KSP probíhá tak, že student od nás jednou za čas dostane poštou zadání několika (čtyř či pěti) úloh, v klidu domácího krbu je (ne nutně všechny) vyřeší, svá řešení v přiměřeně vzhledné podobě sepíše a do určeného termínu zašle na níže uvedenou adresu. My je poté (víceméně obratem) opravíme a spolu se vzorovými řešeními a výsledkovou listinou pošleme při vhodné příležitosti zpět na adresu studenta. Tento cyklus se nazývá *série,* resp. kolo.

Za jeden školní rok obvykle proběhnou čtyři série, v letech hojnějších pak pět. Závěrečným bonbónkem je pak pravidelné *soustředění* nejlepších řešitelů semináře, konané obvykle na začátku ročníku dalšího a zahrnující bohatý program čítající jak aktivity ryze odborné (přednášky na různá zajímavá témata apod.), tak aktivity ryze neodborné (kupříkladu hry a soutěže v přírodě).

Náš korespondenční seminář není ojedinělou aktivitou svého druhu v Evropě – existují korespondenční semináře z fyziky a matematiky při MFF UK, jakož i jiné programátorské semináře (kupříkladu bratislavský). Rozhodně si však nekonkurujeme, ba právě naopak – každý seminář nabízí něco trochu jiného, řešitelé si mohou vybrat z bohaté nabídky úloh a najdou se i takoví nadšenci, kteří úspěšně řeší několik seminářů najednou.

Velice rádi vám odpovíme na libovolné dotazy jak ohledně studia informatiky na naší fakultě, tak i stran jakýchkoliv informatických či programátorských problémů. Jakýkoliv problém, jakákoliv iniciativa či nabídka ke spolupráci je vítána na adrese:

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
118 00 Praha 1

e-mail: *ksp@mff.cuni.cz*
www: *http://atrey.karlin.mff.cuni.cz/ksp/*

Zadání úloh

10-1-1 Hanojské věže — 10 bodů

Bylo-nebylo. V dalekém městě Hanoji od nepaměti stál staroslavný klášter. Mniši v tomto klášteře žijící udržovali již po několik tisíciletí pozoruhodnou tradici: V jedné místnosti na stole z drahocenného dřeva ležela stříbrná deska se třemi křišťálovými hroty, na nichž bylo navlečeno dohromady 64 zlatých disků navzájem různých velikostí. Na počátku byly prý všechny disky na prvním z hrotů, seřazeny od největšího (ten byl dole) k nejmenšímu (nahoře). Každého dne pak za zvuku zvonů přenesli mniši obřadně jeden z disků na jiný hrot, a to tak, že nikdy neležel disk větší na disku menším. Legenda praví, že až se jim podaří všechny disky přemístit na třetí hrot, nastane konec světa.

Vaším úkolem je vymyslet algoritmus a napsat program řešící zobecněný případ této úlohy: na prvním hrotu máte N různých disků a máte je přenést na hrot třetí s dodržením uvedené podmínky. Program dostane na vstupu číslo N a jeho výstupem je posloupnost tahů, každý z nich popsaný dvěma čísly hrotů: odkud a kam se disk přenáší (jelikož je možno hýbat vždy jen s nejvyšším diskem na daném hrotu, je tak postup popsán jednoznačně). Spočtěte, kolik tahů (v závislosti na N) váš algoritmus použije a dokažte, že lépe to nejde.

Příklad: Pro $N = 3$ program odpoví: `1,3 1,2 3,2 1,3 2,1 2,3 1,3.`

10-1-2 Bílá paní — 10 bodů

Na jednom hradě žila již několik staletí jedna bílá paní. Civilizace se na ní bohužel výrazně podepsala – v posledních letech ze záře světel nedalekého města oslepla a navíc byla z návštěvníků hradu tak nervózní, že se už nedokázala ani soustředit na projití zavřenými dveřmi, takže se o to od jedné nepříjemné nehody ani nepokoušela.

Hrad se skládal ze samých komnat uspořádaných do tvaru čtverce a každá komnata měla čtvery dveře vedoucí do komnat sousedních (obvodové zdi hradu si můžeme představovat jako dveře, které se doposud nikomu nepodařilo odemknout). Naše bílá paní neměla jenom smůlu, ale také kocoura, jenž jí čas od času utekl a zalezl do jedné z komnat a ona jej pak musela hledat. Ač byla slepá, dokázala chodit rovně, ale když narazila na zavřené dveře, odrazila se od nich – tedy směr jejího pohybu změnil se v směr přesně opačný. Navíc ještě dokázala poznat, jestli je její kocour u ní (v téže komnatě) či nikoliv.

Napište algoritmus a program, který zjistí, zda bílá paní může či nemůže svého kocoura najít, známe-li její pozici a směr pohybu, jakož i polohu kocoura v hradu (kocour tvrdě spí a při tom se naštěstí pohybovat neumí). Na vstupu dostanete rozměr hradu M, směr, jímž se paní pohybuje (0=sever, 1=jih,

2=východ, 3=západ) a též mapu hradu – to je pole znaků $M \times M$, v němž `0` znamená otevřenou místnost, `1` uzavřenou místnost (dveře spojující dvě otevřené místnosti jsou otevřené, všechny ostatní pak uzamčené), `A` otevřenou místnost, v níž je bílá paní (ta je v mapě právě jedna) a `B` otevřenou místnost, v níž je kocour (též unikátní). Výstupem programu bude odpověď, zda paní svého kocoura najde a pokud ano, v kolikátém kroku se jí to povede (jeden krok odpovídá posunutí do sousední místnosti či odražení od dveří).

10-1-3 Bermudský trojúhelník — 10 bodů

Na jednom z ostrovů bermudského souostroví se znenadání objevil následující hlavolam: je to trojúhelník složený z čísel (na prvním řádku je jedno číslo, na druhém dvě, ..., až na n-tém jich je n) – například tedy takto:

```
    1
   2 3
  1 0 1
 3 1 5 3
```

Úkolem je nalézt takovou cestu vedoucí od vrcholu trojúhelníku k jeho základně (v každém kroku se dostáváme na nižší řádek a to buď o pozici doleva nebo doprava), na níž leží čísla o co největším součtu. Zkuste vymyslet takový algoritmus, aby se dal v rozumném čase realizovat za pomoci tužky a papíru (to jest nepotřeboval milony operací či ohromnou paměť) a zapište jej jako program. Vstupem budiž počet řádků n a jejich obsah, výstupem pak posloupnost znaků `-` a `+`, která pro každý řádek říká, jdeme-li dále doleva či doprava. Je-li nejlepších cest více, stačí vypsat pouze jednu.

Příklad: Pro výše uvedené zadání je jednou ze správných odpovědí `++-`.

10-1-4 No to snad ne! — 10 bodů

Jistě již všichni znáte, jak funguje dvojková, trojková a jiné číselné soustavy: Každé číslo zapsané v soustavě o základu b číslicemi $d_k d_{k-1} \ldots d_1 d_0$ má hodnotu $V = \sum_{n=0}^{k} d_n b^n$. Pro zapisování čísel záporných ovšem potřebujeme před číslo ještě přidat znaménko minus. Nešlo by to bez něj?

Murphyho zákon sice říká, že končí-li novinový titulek otazníkem, odpověď zní „Ne!“, ale v tomto případě lze opravdu odpovědět kladně a dokonce ani nebudeme lhát: stačí totiž za základ soustavy b zvolit nějaké záporné číslo (například -2) a použít tytéž číslice jako pro původní kladný základ. Je až k podivu, že v takovéto šílené soustavě (váhy jednotlivých řádů vycházejí 1, -2, 4, -8 atd.) se dá zapsat libovolné celé číslo bez použití znaménka... ($0 \to$ `0`, $1 \to$ `1`, $2 \to$ `110`, $3 \to$ `111`, $4 \to$ `100`, $-1 \to$ `11`, $-2 \to$ `10` atd.).

Na tomto místě se samozřejmě okamžitě nabízí zadat jako úlohu napsání programu pro převod mezi desítkovou a touto soustavou, jakož i zpětně. Ale tak snadné to samozřejmě nebude...

Napište program, který na vstupu dostane číslo r zapsané v desítkové soustavě ($1 \leq r \leq 100$) a číslo x v minus-dvojkové soustavě (pozor, toto číslo může být *nepříjemně dlouhé* – až miliony cifer) a zjistí, zda je číslo x dělitelné číslem r.

10-2-1 Byrok(r)ati — 10 bodů

V královském paláci se ke značnému zoufalství všech poddaných rozbujela byrokracie – královští ouřadníci si navzájem vytvořili n papalášských funkcí a do nich se jmenovali (označíme si je třeba čísly 1 až n). A když někdo přišel do paláce žádaje audienci u krále, musel jít za ouřadníkem, který byl pro daný den zvolen jako lord tiskový mluvčí (nechť má číslo L) a vyžádat si od něj povolení k audienci.

Jenže jak už to v byrokratických systémech bývá, týmová práce jest nezbytná (asi proto, že umožňuje svalit vinu na ostatní), a tak lord tiskový mluvčí vydá povolení pouze tehdy, dostane-li potvrzení o odbornosti tazatele, potvrzení o loayalitě králi, dále pak glejt o bezúhonnosti, jakož i výpis z rejstříku tortury – tedy vlastně mnoho dalších lejster, která vydávají jiní ouřadníci. Ale vydají je obvykle jen tehdy, přinese-li jim ubohý poddaný glejty od jiných ouřadníků atd. Každý z nich ovšem vydává pouze jedno lejstro (takže každého z nich má smysl navštěvovat nejvýše jednou).

Ale poddaní se nedali – na cestě ke královskému paláci stojí stánek byrokata (to je člověk, jenž se specializuje na potírání byrokracie) a z jedné strany k němu chodí donašeči z paláce dodávající informace o tom, který z pánů ouřadů zrovna chce potvrzení od kterých ostatních (například jakožto čísla o_{ij} jsoucí nenulová, pokud i-tý ouřada požaduje k vydání svého lejstra jiné od j-tého ouřady), zatímco ze strany druhé přicházejí poddaní a ptají se, v jakém pořadí mají papaláše navštívit, aby povolení k audienci od lorda L získali. (Všechny osoby v tomto příběhu jsou smyšlené a jakákoliv podobnost s pohádkou o kohoutkovi a slepičce, s realitou, jakož i s úlohou o topologickém třídění grafu je čistě dílem náhody.)

Úkol pro vás: vymyslet program, který bude řešit byrokatův problém.

Příklad: Pro

$$o = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \quad \text{a} \quad L = 1$$

je jedním z možných pořadí 4, 3, 2, 1, zatímco pro tatáž o_{ij} a $L = 5$ není úkol splnitelný, protože papaláši 5 a 6 se navzájem posílají jeden k druhému.

10-2-2 Minitónní posloupnost — 12 bodů

Napište program, který pro dané číslo n vytvoří posloupnost délky n obsahující každé z čísel 1 až n, a to takovou, aby délka nejdelší monotónní (to jest rostoucí či klesající) podposloupnosti byla co nejmenší. Podposloupností posloupnosti $a_1, \ldots, a_n$ se rozumí jakákoliv posloupnost $a_{i_1}, \ldots a_{i_k}$, kde $i_1 < i_2 < \ldots < i_k$.

Kupříkladu pro $n = 5$ program vygeneruje 1, 4, 5, 2, 3 nebo 3, 2, 1, 4, 5 atd. Nejdelší monotónní podposloupnost má v obou případech délku 3.

10-2-3 Turingův stroj — 10 bodů

Turingův stroj sestává z pásky a řídící jednotky. Páska Turingova stroje má jen jeden konec, a to levý (doprava je nekonečná) a je rozdělena na políčka. Na každém políčku se nachází právě jeden znak z abecedy Σ (to je nějaká konečná množina, o které navíc víme, že obsahuje znak Λ). Nad páskou se pohybuje hlava stroje, v každém okamžiku je nad právě jedním políčkem.

Řídící jednotka stroje je v každém okamžiku v jednom stavu ze stavové množiny Q (opět nějaká konečná množina) a rozhoduje se podle přechodové funkce $f(q, z)$, která pro každou kombinaci stavu q a znaku z, který je zrovna pod hlavou, dává uspořádanou trojici (q', z', m), přičemž q' je stav, do kterého řídící jednotka přejde v dalším kroku, z' znak, kterým bude nahrazen znak z umístěný pod hlavou a konečně m je buďto L nebo R podle toho, zda se má hlava posunout doleva nebo doprava.

Výpočet Turingova stroje probíhá takto: na počátku je hlava nad nejlevějším políčkem, na počátku pásky jsou uložena vstupní data (zbytek pásky je vyplněn symboly Λ) a řídící jednotka je ve stavu q_0. V každém kroku výpočtu se Turingův stroj podívá, co říká funkce f o kombinaci aktuální stav + znak pod hlavou, načež znak nahradí, přejde do nového stavu a posune hlavu v udaném směru. Takto pracuje do té doby, než narazí hlavou do levého okraje pásky, čímž výpočet končí a páska obsahuje výstup stroje.

Příklad: Následující tabulka definuje Turingův stroj nad abecedou $\Sigma = \{\mathbf{0}, \mathbf{1}, \Lambda\}$, který ze vstupního slova složeného ze znaků **0** a **1** odstraní všechny jedničky. Počátečním stavem je stav 0, políčka tabulky označená '—' nemohou být strojem nikdy dosažena.

	0	**1**	Λ
0	$0, \mathbf{0}, R$	$0, \mathbf{1}, R$	$1, \Lambda, L$
1	$1, \mathbf{0}, L$	$2, \mathbf{0}, R$	—
2	$2, \mathbf{0}, R$	—	$3, \Lambda, L$
3	$1, \Lambda, L$	—	—

Úkol: Popište Turingův stroj realizující výpočet součinu dvou čísel. Representaci čísel na pásce stroje si zvolte, jakou uznáte za vhodné.

10-2-4 Bynáriho logaritmus — 10 bodů

Představte si Pascal, v němž typ *integer* může obsahovat až 1024-bitové číslo (alternativně Céčko, kde platí totéž pro typ *int*). Napište v takovémto jazyce co nejrychlejší program, který pro dané číslo n spočte jeho Bynáriho logaritmus, což je nejmenší číslo k takové, že $2^k \geq n$.

K dispozici máte všechny běžné pascalské (resp. céčkovské) operace: sčítání, odčítání, násobení, dělení, bitový and, or, xor i negaci atd.

10-2-5 Kostky jsou vrženy — 12 bodů

Známý metrolog baron Bynári přišel s následujícím problémem: máte na stole rovnoramenné váhy a hromadu n kostek. Vaším cílem je z těchto kostek za pomoci zmíněných vah nalézt k nejtěžších.

Navrhněte algoritmus, který bude osobě obsluhující váhy (nečiňmež, prosím, žádné neodůvodněné předpoklady o její inteligenci) diktovat, co má dělat: v každém kroku vypíše čísla kostek, které má osoba dát na levou misku a čísla těch, které má dát na misku pravou. Načež osoba kostky naskládá a odpoví, jestli byla těžší levá miska, pravá miska či obě vyrovnané. Takto algoritmus pokračuje až do okamžiku, kdy si je jist výsledkem.

Příklad: $n = 3$, $k = 2$. První vážení: na levé misce `1`, na pravé `2`, výsledek `L` (levá těžší). Druhé vážení: nalevo `1`, napravo `3`, výsledek: `P`. Hotovo, nejtěžší je `3` a druhá nejtěžší `1`.

10-3-1 Domečkologie — 10 bodů

Ano, jak jste již zaručeně uhodli podle názvu úlohy, opravdu půjde o kreslení domečků jedním tahem. Ale nikoliv jen onoho zprofanovaného domečku z osmi čar, nýbrž domečku obecného: je dáno n bodů v rovině svými souřadnicemi $(x_1, y_1), \ldots, (x_n, y_n)$ a m hran (pro každou víte čísla dvou vrcholů, které spojuje, na pořadí nezáleží).

Napište program, jenž nalezne jednotahové nakreslení zadaného domečku (tedy vlastně posloupnost vrcholů takovou, že půjdete-li přes ně v určeném pořadí, nakreslíte každou hranu právě jednou) či odpoví, že domeček nelze jedním tahem nakreslit.

10-3-2 Sirky jsou vrženy — 11 bodů

V casinu v Las Miseras hráli podivuhodnou hru pro dva hráče: na stole ležely dvě hromádky zápalek, na počátku na každé z nich leželo N zápalek a

bylo dáno celé kladné číslo k. V každém tahu mohl jeden z hráčů odebrat k zápalek buďto z první nebo z druhé hromádky, případně z obou z nich najednou. Hráči se pravidelně střídali, prohrál ten, který již nic odebrat nemohl.

Navrhněte strategii pro tuto hru a vytvořte program, který ji s vámi (tedy spíše proti vám) bude hrát.

10-3-3 Lloydova devítka — 10 bodů

Mějme čtvercovou krabičku rozdělenou na 3×3 políčka, na něž jsou rozmístěny hrací kameny s čísly od jedné do osmi (každý je tam právě jednou) a jedno políčko je volné. V každém tahu můžete posunout na volné políčko libovolný z kamenů, které s ním sousedí. Cílem je dosáhnout konfigurace

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & \bullet \end{array}$$

Napište program, který vymyslí *nejkratší* posloupnost tahů vedoucí z konfigurace zadané k cílové, případně sdělí, že to nejde.

10-3-4 Hic sunt leones — 10 bodů

Představte si, že vám někdo zadal konvexní mnohoúhelník v rovině souřadnicemi jeho vrcholů

$$(x_1, y_1), \ldots, (x_n, y_n)$$

přesně v tom pořadí, jak jsou spojeny hranami (hrany tedy vedou vždy z i-tého do $(i+1)$-ního vrcholu a pak z n-tého do prvního) a vy máte o daném bodě (x, y) rozhodnout, leží-li uvnitř nebo vně tohoto mnohoúhelníka (hranice je také uvnitř).

Zkuste napsat co nejrychlejší program, který to zjistí (předpokládejte, že pro jeden mnohoúhelník budete testovat velké množství bodů, takže klidně můžete trávit nějaký čas předpočítáváním pomocných dat, jen když pak bude vyhledávání co nejrychlejší).

10-3-5 Terucempa — 10 bodů

V jedné jeskyni žilo 12 trpaslíků (Arnošt, Bříťa, Cecil, David, Eda, Filip, Gustav, Horymír, Ivan, Janek, Kája a Luděk) a každé ráno hned po rozednění vylezli do nedaleké soutěsky umýt se v potoce. Leč soutěska byla úzká, a tak museli stát ve frontě. Jelikož to byli trpaslíci vzdělaní a ne nějací permoníci mdlého rozumu, domluvili se, že každé ráno budou čekat v pořadí jiném, aby nikdo z nich nebyl ve výhodě proti ostatním.

Když přemýšleli, jak to zařídit, Cecila napadlo, že to vezmou systematicky: každé své pořadí (odborně řečeno *permutaci,* okamžitě dodal Arnošt) si

zapíší jako jedno slovo z písmenek A až L (ejhle, to máme ale šikovná jména, napadlo Janka) a vezmou ta slova podle abecedy – první den to první (*ABCDEFGHIJKL*), druhého druhé (*ABCDEFGHIJLK*) až někdy v budoucnosti (pamatujte, že trpaslíci jsou dlouhověcí, přidal se k rozhovoru Ivan) poslední (*LKJIHGFEDCBA*) a pak zase od začátku.

Na vás je, abyste napsali program odpovídající na otázku, v jakém pořadí stáli trpaslíci ve frontě N-tého dne.

10-4-1 Wagón klád — 10 bodů

Ve wagónu je ohromné množství tyčí různých délek. Vy potřebujete zjistit, jaké všechny délky jdou z těchto tyčí sestavit. Máte však málo lepidla, takže nemůžete spojovat dohromady více než dvě tyče. Vrchní královský dřevolepič po vás chce, abyste napsali program, který tuto úlohu vyřeší, navíc ovšem žádá, abyste každou délku oznámili právě jednou.

Ovšem celé to má ještě jeden háček: tyčí je tolik, že jste rádi, že se vám vejde do paměti seznam jejich délek – na to, aby se vám tam vešel seznam součtů dvojic, není ani pomyšlení.

Příklad: Pro seznam $1, 2, 3$ je správným výsledkem $2, 3, 4, 5, 6$.

10-4-2 DĚLITEL — 11 bodů

Činíme další pokus odradit vás od řešení KSP za pomoci Turingových strojů: Zkuste sestrojit Turingův stroj hledající pro daná dvě celá kladná čísla x a y jejich největšího společného dělitele $[x, y]$ – tedy největší přirozené číslo takové, které je dělitelem jak x, tak y.

Hodnotí se nejen funkčnost, ale i rychlost výpočtu. Nezapomeňte spočítat i časovou a paměťovou náročnost svých strojů.

Nápověda: Uvědomte si, že platí:

- $[x, y] = [y, x]$
- $[x, y] = [x - y, y]$
- Je-li x sudé a y liché, $[x, y] = [x/2, y]$.
- Jsou-li x i y sudá, $[x, y] = 2 \cdot [x/2, y/2]$.

10-4-3 Sssssttttrrrriiinng — 10 bodů

Mějme velice dlouhý řetězec znaků $z_1 \dots z_n$, dokonce tak dlouhý, že se nám mimo něj vejde do paměti už jen krátký program a několik málo pomocných proměnných. A nyní jsme se rozhodli, že uvnitř řetězce prohodíme dva nestejně dlouhé nepřekrývající se substringy $z_i \dots z_j$ a $z_r \dots z_s$ $(1 \le i \le j < r \le s \le n)$ – což znamená přerovnat znaky do pořadí

$$z_1 \dots z_{i-1} z_r \dots z_s z_{j+1} \dots z_{r-1} z_i \dots z_j z_{s+1} \dots z_n.$$

Zkuste to s danou pamětí v co nejkratším čase provést.

Příklad: Pro string `neceločíselný`, $i = 3$, $j = 5$, $r = 7$, $s = 12$ je kýženým výsledkem `nečíselnocelý`.

10-4-4 Duraloví dravci — 11 bodů

V Absurdistánu měli N měst spojených navzájem leteckými linkami S různých dopravních společností. Ale jak již to v tomto státě bývá, byrokracie se rozmohla tak, že i policisté hlídkovali jezdíce na úředním šimlu, a proto každá dopravní společnost měla různé tarify za překlad nákladu z letadel různých jiných společností. Vám přinesli následující tabulky:

- $p(s, i, j)$ – cena účtovaná společností s za přepravu vašeho nákladu z města i do města j nebo ∞, pokud taková letecká linka neexistuje.
- $m(i, r, s)$ – cena účtovaná za přeložení nákladu z letadla společnosti r do letadla společnosti s v městě i nebo ∞, pokud tomu úřední předpisy brání.

Máte vyřešit, jak svůj náklad co nejlevněji přepravit z města 1 do města N.

10-5-1 Společná podmatice — 10 bodů

Jsou dány matice $A[n \times n]$ a $B[n \times n]$. Nalezněte jejich největší společnou podmatici (to jest taková a, b, c, d, p, q, že pro každé $0 \leq i < p$ a $0 \leq j < q$ je $A_{a+i,b+j} = B_{c+i,d+j}$ a $p \cdot q$ je největší možné).

10-5-2 Kob ýlam ám alybok? — 10 bodů

Don Coyote, pán zámku Barabizonu, měl podivnou zálibu: hledal v knihách co nejdelší úseky, které vypadají stejně i při čtení pozpátku (například slovo '*nepotopen*' nebo větu '*Jelenovi pivo nelej*' – na mezery nehleděl). Zkuste napsat program, který jeho problém snadno a rychle vyřeší.

10-5-3 Cycloose — 8 bodů

Mějme následující jednoduchý Pascalský prográmek:

```
var r:real;
begin r := 1; while r < 1e30 do r := r+1; end.
```

Nebo ekvivalent v Céčku:

```
main() { float r; for(r=0;r<1e30;r++); }
```

Otázka zní: Doběhne při rychlostech dnešních běžných počítačů do konce příštího století? Tisíciletí? A nebo do zítřejšího rána?

10-5-4 S radostí přijmi svůj (o)sud — 10 bodů

Ztroskotali jste na pustém ostrově ... tedy on alespoň na začátku pustě vypadal, ale ke svému nemalému úžasu jste zjistili, že jej obývají duchové dávno ztroskotavších byzantských obchodníků s vínem a že se neustále hádají o jednom problému: mají několik (řekněme k) nádob známých objemů $v_1, \ldots, v_k$ pint a potřebují jimi odměřit p pint vína. Chtějí to ovšem odměřit přesně (žádné odhadování, to do seriózního obchodu nepatří! ... tedy alespoň v říši nevinných duchů vinných obchodníků), takže vždy mohou jen naplnit určitou nádobu až po okraj vínem (ať již ze sudu nebo z jiné nádoby) nebo nějakou nádobu zcela vyprázdnit (ať již zpět do sudu či do jiné nádoby).

Zkuste vymyslet, jak pro dané velikosti nádob vymyslet, jak na co nejmenší počet přelití odměřit p pint vína. (Duchové vám totiž slíbili, že když jim pomůžete, žízní nezahynete...)

Příklad: Jsou-li objemy nádob $3, 5, 7$ pint a potřebujeme odměřit 1 pintu vína, přeléváme $S \to 3$ (S je sud; máme $0, 0, 7$), $3 \to 1$ $(3, 0, 4)$, $1 \to S$ $(0, 0, 4)$ a $3 \to 1$ $(3, 0, 1)$.

O časové složitosti

Při opravování řešení jsme byli přímo zavaleni dotazy typu „co to vlastně je ta časová a paměťová složitost?“ – následujících pár řádků budiž chápáno jako pokus o odpověď na otázky podobného druhu:

Pro každý problém obvykle existuje více různých algoritmů tento problém řešících, ale zřídkakdy bývají všechny „stejně dobré“. Jedna z možností, jak algoritmy porovnávat, je zkoumat, za jak dlouho se k řešení dopracují a kolik paměti k tomu potřebují. Tyto vlastnosti se většinou popisují pomocí takzvané časové a paměťové složitosti algoritmu. Obojí se vyjádřuje analogicky, pročež se zaměříme například na složitost časovou:

Předem můžeme vyloučit porovnávání podle času v sekundách potřebného ke zpracování jedněch určitých vstupních dat, jelikož jeden algoritmus může pracovat rychleji než jiný pro nějaká vstupní data a daleko pomaleji pro jiná. Rozumné by mohlo být vyjádřit čas výpočtu v závislosti na nějaké míře velikosti vstupních dat (obvykle se označuje písmenem N) – to jest např. počet prvků zpracovávané posloupnosti – tedy vyjádřit čas jako nějakou funkci $t(N)$.

Jenže ouha, hodnoty funkce t budou různé, budeme-li program spouštět na různých počítačích a navíc se tato funkce bude velice obtížně určovat. Vypomůžeme si proto *asymptotickým* vyjádřením této funkce, jež nám (poněkud neexaktně řečeno) udává, jak rychle funkce t roste v závislosti na N, přičemž nás bude zajímat zejména její chování pro velká N (pro malé velikosti vstupů jsou i pomalé algoritmy vcelku použitelné). Nalezneme tedy nějakou „jednoduchou“ funkci $f(N)$ (například nějakou mocninu N), o které víme, že roste

stejně rychle jako $t(N)$, ovšem bez ohledu na multiplikativní konstanty (např. $3N^2$ roste stejně rychle jako N^2, stejně tak $7N^3 + 5N^2 - 10$ se chová pro velká N stejně jako N^3 apod.). Takové srovnání chování funkcí značíme $t = O(f)$.

Typické algoritmy mají časové složitosti $O(1)$ (konstantní), $O(\log N)$ (logaritmická), $O(N^k)$ (polynomická) nebo $O(N^k \cdot \log N)$, popřípadě $O(2^N)$ (exponenciální, natolik pomalá, že prakticky nepoužitelná – již pro $N = 1000$ byste se výsledku nedočkali v nejbližších stoletích).

Pro ty z vás, kteří chtějí znát přesnou definici: $t = O(f)$ právě tehdy, existuje-li konstanta $c > 0$ taková, že pro každé x platí $t(x) \leq c \cdot f(x)$.

Prahnete-li po hlubším vysvětlení tohoto tématu, můžete nahlédnout do knihy Algoritmy a programovací techniky od RNDr. Pavla Töpfera.

Vzorová řešení

10-1-1 Hanojské věže — Martin Bělocký

Tato úloha nepatřila mezi problémové. Buď jste ji řešili metodou „Rozděl a panuj“ nebo analýzou možných tahů, což bylo komplikovanější, a téměř nikdo jste nenapsali důkaz správnosti algoritmu. Zajímavé nám přišlo, že téměř polovina lidí řešících úlohu rekurzí napsala chybně, že paměťová složitost je exponenciální.

My použijeme metodu „Rozděl a panuj“. Naši úlohu rozdělíme na menší části, které následně vyřešíme. K tomu, abychom mohli přenést libovolný disk D z tyče A na tyč B, potřebujeme, aby byly všechny disky menší než D na pomocném disku C, jinak by to bylo proti pravidlům. (♯) Pokud tedy budeme přemisťovat disk D, nemusíme se zabývat většími disky než D, neboť ty nemohou dle pravidel ovlivnit přemisťování.

Algoritmus:

In: $N > 0$ ... počet disků na tyči A; chceme je přenést na tyč B.

1. Přeneseme $N-1$ disků z A na C.
2. Přeneseme 1 disk z A na B (v této úrovni největší disk).
3. Přeneseme $N-1$ disků z C na B.

Kroky 1 a 3 se vyřeší stejně. Algoritmus je konečný, neboť se rozkládá na menší úlohy nejvíce do hloubky N (parametr N pro podúlohu je vždy o 1 menší). Algoritmus je korektní, neboť jednotlivé kroky algoritmu jsou v souladu s pravidly (to jsme již ukázali výše). Algoritmus je tedy správný.

Zbývá dokázat, že neexistuje algoritmus, který by úlohu vyřešil rychleji. Kdyby ano, potom bychom buď (a) museli použít jiný algoritmus, nebo (b) bychom mohli zlepšit jednu z jeho částí. Každý algoritmus však musí dodržovat (♯), tedy musí obsahovat náš algoritmus. (b): krok 2 už zlepšit nelze, neboť v něm přenášíme disk z A na B přímo jedním tahem. Kroky 1 a 3 se řeší stejným algoritmem. Tedy všechny části jsou nejrychlejší možné a tedy náš algoritmus je nejrychlejší možný.

Časová složitost: z algoritmu plyne rekurentní vztah pro počet kroků: $T(N) = T(N-1) + 1 + T(N-1)$. Triviálně platí $T(1) = 1$, $T(2) = T(1) + 1 + T(1) = 2 * T(1) + 1 = 2 * 1 + 1 = 3$, $T(3) = 2 * T(2) + 1 = 2 * (2 * T(1) + 1) + 1 = 2^2 + 2^1 + 2^0 = 7$ ... $T(N) = 2^{N-1} + 2^{N-2} + \ldots + 2^0 = 2^N - 1$. Časová složitost je tedy exponenciální, $O(2^N)$.

Paměťová složitost: Algoritmus a program potřebuje v jedné úrovni vnoření paměť pro 4 proměnné, přičemž počet úrovní vnoření (rekurze) je N. Tedy paměťová složitost je lineární, $O(N)$.

Program realizuje rozdělení na podúlohy pomocí rekurze. Je přímým přepsáním algoritmu.

```
Program Hanoj;
var N:integer;

procedure prenes(kolik:integer; odkud,pomoci,kam:char);
begin
 if kolik>0 then begin
   prenes(kolik-1,odkud,kam,pomoci);
   writeln('Prenasim disk z tyce ',odkud,' na tyc ',kam);
   prenes(kolik-1,pomoci,odkud,kam);
 end;
end;

begin
 writeln ('kolik disku je na tyci A ?');
 readln(N);
 prenes(N,'A','B','C');
end.
```

10-1-2 Bílá paní — Aleš Přívětivý

Zadání této úlohy bylo zformulováno dost nešťastně, nicméně i tak přišlo mnoho řešení. Někteří řešili onu triviální úlohu, jiní si zadání přeformulovali, a pokud jejich úloha měla nějaký smysl, uznávali jsme tato řešení také.

Ze zadání plynulo, že bílá paní se umí pohybovat pouze po přímce před sebe (po odrazu opačným směrem), a tedy přímku nemůže opustit. Někteří řešitelé se zamýšleli nad možností, že stěny hradu nejsou rovnoběžné se světovými stranami, a tím pádem bílá paní začínající svůj pohyb například na sever nemusí projít (resp. narazit) dveřmi, ale může se odrážet v jedné místnosti z rohu do rohu po věky věků. S našimi informacemi o hradě ale nemůžeme takovouto situaci simulovat, takže mlčky předpokládejme, že bílá paní chodí jen a pouze rovnoběžně se stěnami hradu.

Dosažitelnost kocoura bílou paní pak přechází na určení, zda se kocour a bílá paní nachází na stejném sloupci/řádku a zda mezi nimi nejsou zavřené dveře (tj. mezi kocourem a bílou paní jsou samé nuly). Pokud jsou tyto podmínky splněny, bílá paní nalezne kocoura v konečném počtu kroků. Toto můžeme rozhodnout v kvadratickém čase a s konstantními časovými nároky.

K určení počtu kroků potřebujeme znát polohu omezujících dveří (tj. dveří, které omezují pohyb bílé paní ve sloupci/řádku) pro případ, kde dojde nejprve k odražení bílé paní od dveří a pak teprve najde kocoura. Nalezení omezujících dveří v řádku je triviální, ve sloupci je hledání horních omezujících dveří ztíženo tím, že nevíme, ve kterém sloupci se kocour a bílá paní nacházejí, a proto musíme hledat omezující dveře ve všech sloupcích, což zvyšuje konstantní paměťové nároky na lineární. To by se dalo eliminovat dvojitým přečtením dat

ze vstupu, ale to podle mého názoru není vůbec elegantní. Samotné spočtení kroků z těchto údajů je zřejmé.

Celkově má algoritmus kvadratickou časovou složitost, tj. $O(M^2)$ a lineární paměťovou složitost, tj. $O(M)$, kde M je počet místností v sloupci/řádku.

```
program b_pani;                      {casova slozitost O(N*N), pametova O(N)}

var Tahu,M,Smer:integer;

Function TahuVZ:integer;             {resi ulohu, pohybuje-li se BP po radku}
Var BpX,KocX,Left,Right:integer;    {pozice kocoura, BP, a omezujicich dveri}
    i,j:integer;
    s:string;
begin
  TahuVZ:=-1;BpX:=0;KocX:=0;Right:=M+1;
  for i:=1 to M do begin                                  {pro vsechny radky}
      Readln(input,s);Left:=0; {omezujici dvere nastav na obvodove zdi hradu}
      for j:=1 to M do             {najdi kocku, BP a omezujici zdi na radku}
          case s[j] of
            '1':if (BpX>0)and(KocX>0) then begin Right:=j;break; end
                  else if (BpX>0)or(KocX>0) then exit else Left:=j;
            'A':BpX:=j;
            'B':KocX:=j;
          end;
      if KocX<>BpX then break;                      {kocka nebo BP nalezena}
  end;
  if (KocX=0)or(BpX=0) then exit;       {kocka a BP nejsou na stejnem radku}
  if Smer=2 then
      if KocX>BpX then TahuVZ:=KocX-BpX else TahuVZ:=2*Right-KocX-BpX-1;
  if Smer=3 then
      if BpX>KocX then TahuVZ:=BpX-KocX else TahuVZ:=KocX+BpX-1-2*Left;
end;

Function TahuSJ:integer;           {resi ulohu pohybuje-li se BP ve sloupci}
Var BpX,BpY,KocX,KocY,Bottom:integer; {pozice BP, kocky a omezujicich dveri}
    Top:array[1..100] of word;    {pozice moznych hornich omezujicich dveri}
    i,j:integer;
    s:string;
begin
  TahuSJ:=-1;BpX:=0;Bpy:=0;BpX:=0;KocX:=0;KocY:=0;Bottom:=M+1;
  for i:=1 to M do Top[i]:=0;
  for i:=1 to M do begin                                {ber hrad po radkach}
      Readln(input,s);
      for j:=1 to M do         {urceni pozic kocky, BP a omezujicich dveri}
          case s[j] of
           '1':if (BpX>0)and(KocX>0) then begin Bottom:=i;break; end
                 else if (BpX>0)or(KocX>0) then exit else Top[j]:=i;
           'A':begin BpX:=j;BpY:=i;if (KocX>0)and(KocX<>BpX) then exit;end;
           'B':begin KocX:=j;KocY:=i;if (BpX>0)and(KocX<>BpX) then exit;end;
          end;
      if Bottom<>M+1 then break;   {nasli jsem uz i spodni omezujici dvere}
  end;
  if (KocX=0)or(BpX=0) then exit;    {kocka a BP nejsou ve stejnem sloupci}
  if Smer=0 then
      if BpY>KocY then TahuSJ:=BpY-KocY else TahuSJ:=BpY+KocY-2*Top[BpX]-1;
  if Smer=1 then
      if KocY>BpY then TahuSJ:=KocY-BpY else TahuSJ:=2*Bottom-BpY-KocY-1;
```

```
end;

begin
  readln(input,M,Smer);
  if Smer>1 then Tahu:=TahuVZ else Tahu:=TahuSJ;
  if Tahu=-1 then writeln('Bila pani nenajde kocoura...') else
     writeln('Najde ho v ',Tahu,' tahu.');
end.
```

10-1-3 Bermudský trojúhelník — Martin Mareš

Přestože tato úloha byla poměrně jednoduchá, objevila se ve vašich řešeních celá řada nešvarů naprosto typických pro řešení prvních sérií seminářů. Konkrétně:

- Chybějící popis algoritmu – nestačí uvést program, je též nutné zmínit se o tom, *jak* vlastně funguje.
- Chybějící důkaz správnosti algoritmu – nestačí algoritmus vysvětlit, je též záhodno uvést, *proč* zadaný problém řeší (pokud to není zjevné z principu jeho konstrukce, což zde obvykle nebylo).
- Chybějící, případně nesprávný odhad časové a paměťové složitosti algoritmu – doporučuji přečíst si úvodní text o časové složitosti v zadání první série.
- „Bells and whistles“ – v seminářových programech opravdu není nutné zabrat většinu papíru mazáním obrazovky, čekáním na stisk klávesy, barvičkami, zvuky a pazvuky, prompty při zadávání, kontrolami korektnosti vstupních dat (nejsou-li explicitně vyžadovány v zadání) atd.

Nuže dobrá, nyní již k tomu, jak úlohu opravdu vyřešit. Označme si čísla v pyramidě $a_{i,j}$, kde i je řádek a j pozice na řádku. Při tomto očíslování jistě platí, že z prvku $a_{i,j}$ můžeme pokračovat buďto do $a_{i+1,j}$ nebo do $a_{i+1,j+1}$.

Povšimněme si nyní triviálního, leč velice důležitého faktu: maximální cesta (tak budeme říkat cestám vedoucím z daného bodu až k základně a majícím ze všech takových cest největší součet čísel na ní ležících) z prvku $a_{i,j}$ se skládá z odbočky doleva resp. doprava následované opět maximální cestou z bodu $a_{i+1,j}$ resp. $a_{i+1,j+1}$ (kdyby pokračovala cestou horší než je cesta maximální, mohli bychom tuto cestu nahradit cestou maximální, čímž bychom získali cestu z $a_{i,j}$ lepší než byla původní maximální, což by byl spor).

Nyní budeme konstruovat maximální cesty z jednotlivých bodů, a to postupně po jednotlivých hladinách počínaje základnou. Pro každý bod spočteme délku maximální cesty $c_{i,j}$. Pro body základny je to triviální: $c_{n,j} = a_{n,j}$ (cesty tam končí). Nyní známe-li již všechna $c_{i,j}$ pro dané i, snadno dopočítáme i pro hladinu bezprostředně výše: cesta z $c_{i-1,j}$ buďto začíná odbočkou doleva

a má součet $a_{i-1,j} + c_{i,j}$ nebo odbočkou doprava se součtem $a_{i-1,j} + c_{i,j+1}$. Jelikož hledáme cestu maximální, vybereme z těchto dvou možností tu s větším součtem (viz úvaha v předchozím odstavci), pokud jich je více, použijeme libovolnou z nich.

A jak vypíšeme to, co se po nás chce (to jest maximální cestu z $a_{1,1}$)? Inu, snadno: začneme v $a_{1,1}$ a pokračujeme do $a_{2,1}$ nebo $a_{2,2}$ podle toho, zda je větší $c_{2,1}$ nebo $c_{2,2}$. Obecně z bodu $a_{i,j}$ pokračujeme do toho z bodů $a_{i+1,j}$, $a_{i+1,j+1}$, pro nějž je hodnota c větší. Takto vybereme přesně tu cestu, jejíž délku jsme vypočetli výše uvedeným postupem.

Časová složitost algoritmu je $O(N^2)$, jelikož každý prvek trojúhelníka zpracujeme maximálně dvakrát (asymptoticky rychlejší algoritmus ani nemůže existovat, ježto každý prvek musíme nutně otestovat), paměťová složitost rovněž $O(N^2)$.

Program postupuje přesně dle tohoto algoritmu, přičemž délkami maximálních cest přímo nahrazuje jednotlivé prvky trojúhelníka.

```
#include <stdio.h>

#define MAX 10
int N;                                  /* Počet řádků */
int T[MAX][MAX];                        /* Trojúhelník */

int main (void)
{
    int i, j;

    scanf ("%d", &N);                   /* Načtení vstupu */
    for (i=0; i<N; i++)
        for (j=0; j<=i; j++)
            scanf ("%d", &T[i][j]);
    for (i=N-2; i>=0; i--)              /* Výpočet délek maximálních cest */
        for (j=0; j<=i; j++)
            T[i][j] += (T[i+1][j] > T[i+1][j+1]) ? T[i+1][j] : T[i+1][j+1];
    j = 0;
    for (i=1; i<N; i++)                 /* A nyní výpis výstupu */
        if (T[i][j] > T[i][j+1])
            putchar ('-');
        else {
            putchar ('+');
            j++;
        }
    putchar ('\n');                     /* Hotovo */
    return 0;
}
```

10-1-4 No to snad ne! — Vít Novák

Minusdvojková soustava vám příliš hlavu nezamotala, což je potěšující. Jen si někteří z vás nevšimli, že číslo může mít miliony cifer – a tedy že je nejlepší je zpracovávat přímo při vstupu. Je totiž možné, že se číslo nevejde do paměti.

Nejjednodušší samozřejmě je využít přímo definice, to ale právě vynutí uložení celého minusdvojkového čísla do paměti. Výhodnější je uvědomit si jednu zákonitost, na které řešení založíme:

Když totiž nějaké číslo a má při dělení číslem k zbytek z, pak číslo $a0$ bude mít stejný zbytek po dělení číslem k, jaký bude mít číslo $z0$. Skutečně, například v desítkové soustavě $313 \bmod 3 = 1$ a při tom $3130 \bmod 3 = 1$ stejně jako $10 \bmod 3 = 1$.

Platí to skutečně vždycky? Pokud $a \bmod k = b$, pak musí pro nějaké c platit $a = k \cdot c + b$ a tedy $a \cdot s = k \cdot s \cdot c + b \cdot s$ odkud už vyplyne, že $a \cdot s \bmod k = b \cdot s \bmod k$.

Toho teď využijeme k jednoduchému algoritmu: Postupně budeme načítat jednotlivé cifry (bude to vždy jednička nebo nula), přičteme k (-2)-násobku stávajícího zbytku a „vymodulíme“ k. Když takhle zpracujeme celé číslo, dostaneme zbytek po dělení minusdvojkového čísla číslem k. Je-li to nula, bylo číslo dělitelné.

```
#include <stdio.h>

void main () {
    int k, zbytek, cifra;
    printf ("Zadej číslo v desítkové soustavě:"); scanf ("%d", &k);
    printf ("Zadej minusdvojkové číslo:");
    zbytek=0;
    getchar ();                                   /* zruší přebytečný enter */
    for (; ; ) {
       cifra=getchar ();
       if (cifra!='0'&& cifra!='1') break;
       zbytek= (zbytek* (-2)+ (cifra-'0'))%k;
    }
    if (zbytek) puts ("Není dělitelné.");
     else puts ("Je dělitelné.");
}
```

10-2-1 Byrok(r)ati — Michal Koucký

Z řešení bylo vidět, že s byrokratickým aparátem máte každý nějaké ty zkušenosti. Ne všichni je však máte stejné. Někteří z vás byli z těch byrokratů tak zmatení (nebo byli tak pilní?), že se zapomínali podívat do tašky na lejstra, zda tam již požadované lejstro nemají, a jedno a totéž lejstro sháněli při každém byrokratickém požadavku stále znovu a znovu. To u těchto lidí vedlo k tomu, že objem jejich tašky narůstal a narůstal, až přerostl únosné meze a přiblížil se exponenciální hranici. Tací se tedy zjevně nikdy audience nedočkali, neboť v průběhu shánění lejster padli vyčerpáním. Za tuto metodu shánění lejster jsme strhávali čtyři body. Skupina studovaných lidí, kteří ohřívají vodu na čaj tak, že nejprve případnou vodu z konvice vylijí, poté znovu do konvice napustí požadované množství vody a pak teprve postaví konvici na sporák, převedla tento problém na známý problém topologického třídění. Podle toho, jak se jim

zadařilo, pak úlohu řešili buď v čase kubickém či čase kvadratickém. Za takové řešení obvykle autor utržil o nějaký ten bod méně.

Ti nejzkušenější z vás, mezi které patří i Tonda Hildebrand z Jeseníku, řešili tento problém postupem, který byl výstižně popsán právě v pracech zmíněného, s byrokracií zkušeného autora, a proto si zde jeho popis dovolíme otisknout doslova. (Věříme, že díky jeho postupu budeme v byrokratickém aparátu tak zběhlí, že případné spory o autorská práva dokážeme odrazit.)

Tedy: Byrokratický systém je věc záludná a nejeden poddaný padl vyčerpáním při shánění povolení k audienci. Není se co divit, když prochodil kilometry palácových chodeb, mnohokrát se ztratil, povolení k audienci stále neměl, ale zato různých zbytečných lejster měl bezpočet. Takový hloupý sedlák udělal chybu, že se zběsile vrhnul do ještě zběsilejší rekurze, stále se vnořoval do nových ouřadů až nakonec zapomněl cestu domů.

Byrokrat je naštěstí člověk líný, nejraději by seděl u svého číslicového stroje a z donesených informací zjišťoval, jak to v tom paláci vlastně chodí. Je mu jasné, že procházet úřady na papíře je daleko rychlejší než je hledat v paláci. V zájmu pohodlí také zjistil, že na obelstění byrokratického aparátu mu stačí jedno pole `PAPALÁŠI` s tolika prvky, kolik je v zámku ouřadníků. Tady bude mít u každého ouřadníka:

- Jestli nemá jeho lejstro $\mathcal{L}$, pak pokud bude o toto lejstro požádán, tak ho musí sehnat.
- Jestli už má jeho lejstro $\mathcal{L}$, pak ouřadníka už nikdy nemusí vidět.
- Jestli shání jeho lejstro $\mathcal{L}$, aby si ho ouřadové neposílali mezi sebou.

Dále si na papírek zapisuje, v jakém pořadí lejstra získává. Protože chodil po ouřadech v určeném pořadí (a jen když musel) a na papírek si zapisoval čísla ouřadníků teprve tehdy, až lejstro opravdu držel v ruce (pomyslně), je tedy pořadí ouřadníků na papírku výsledkem a může ho (samozřejmě za honorář) předat poddanému. Pokud by se mu stalo, že při shánění určitého lejstra by po něm jiný ouřada toto lejstro chtěl (jednoduše se podívá do pole `PAPALÁŠI`), je úkol nesplnitelný a poddaný to musí zkusit zítra.

Výše uvedený popis odpovídá postupu chytrého sedláka, který dělá jen to co musí, ale poctivě. Takovýto postup spotřebuje kvadratické množství času v závislosti na počtu ouřadníků ($O(n^2)$, kde n je počet ouřadníků) a stejné množství paměťového prostoru (na zapamatování provázanosti ouřadů).
Program snad netřeba zvlášť komentovat.

```
Program Byrokrati;

const MaxN = 20;

type Lejstro = ( nemamLejstro, mamLejstro, shanimLejstro);
```

```
    { tajne znacky na potirani byrokracie }

var O        : array [1..MaxN,1..MaxN] of byte;   { vztahy mezi ouradniky }
    Papalasi : array [1..MaxN] of Lejstro;        { struktura na potirani byrokracie }
    Papirek  : string;                            { sem se zapisuje poradi navstev }
    L        : integer;                           { lord - tiskovy mluvci }
    N        : integer;                           { pocet ouradniku }
    Ch       : string;                            { pomocny retezec }
    I        : integer;                           { pomocna promenna }

procedure Byrokat(Byrokrat:integer);  { sezen lejstra pro byrokrata Byrokrat }
var i:integer;
begin
   for i:=1 to N do
    if O[Byrokrat,i]<>0 then   { chce po mne lejstro od byrokrata i ? }
     if Papalasi[i]=nemamLejstro then { pokud jeho lejstro nemam, }
     begin
        papalasi[i]:=shanimLejstro;   { tak ho jdu sehnat }
        Byrokat(i);
     end
     else
     if papalasi[i]=shanimLejstro then { pokud uz jeho lejstro shanim }
     begin
        writeln('Ukol neni splnitelny - papalas ',i,' me posila zpatky');
        halt(0);
     end;

         { Pokud jeho lejstro mam, tak ho ukazu }
   str(Byrokrat,ch);
   Papirek := Papirek + ch +',';   { je nucen mi sve lejstro vydat }
   Papalasi[Byrokrat]:=mamLejstro; { zapisu si to }
end;

procedure NactiVstup;
begin
   { Cosi nacita }
end;

begin
   NactiVstup;

   for i:=1 to N do Papalasi[i]:=nemamLejstro;
   papirek:='';

   papalasi[L]:=shanimLejstro;

   Byrokat(L);

   writeln(papirek);
end.
```

10-2-2 Minitónní posloupnost — Daniel Kráľ

Před započetím řešení úlohy je třeba si důkladně přečíst definici podposloupnosti. První řešení, které nás napadne je střídat „velká“ a „malá“ čísla a vytvořit např. pro $n = 8$ posloupnost $1, 8, 7, 2, 6, 3, 5, 4$ – tato posloupnost má však monotónní podposloupnost délky 5 a to $8, 7, 6, 5, 4$. Po chvilce zkoušení však pro $n = 8$ nalezneme posloupnost $1, 8, 7, 2, 4, 6, 3, 5$, kde délka nejdelší monotónní podposloupnosti je 4, a poté i posloupnost $7, 2, 4, 1, 8, 6, 3, 5$, kde délka nejdelší monotónní podposloupnosti je dokonce pouze 3.

Pokusme se tedy nejprve udělat pro dané n odhad délky nejdelší monotónní podposloupnosti. Uvažme libovolnou posloupnost dle zadání úlohy a každému jejímu členu přiřaďme dvojici čísel reprezentující délku nejdelší rostoucí posloupnosti a délku nejdelší klesající posloupnosti jím končící. Tedy např. pro posloupnost $7, 2, 4, 1, 8, 6, 3, 5$ z prvního odstavce tímto postupem vytvoříme $7 \to (1, 1), 2 \to (1, 2), 4 \to (2, 2), 1 \to (1, 3), 8 \to (3, 1), 6 \to (3, 2), 3 \to (2, 3), 5 \to (3, 3)$. Délka nejdelší monotónní podposloupnosti je rovna maximu z přiřazených čísel. Použité dvojice čísel se neopakují, nebot všechny členy posloupnosti jsou různé. Byla-li některému členu přiřazena např. dvojice $(5, ?)$, byla určitě jinému členu přiřazena dvojice $(4, ?)$ – „každá alespoň dvouprvková posloupnost má předposlední prvek“. Má-li tedy nejdelší monotónní podposloupnost délku k, lze vytvořit k^2 dvojic čísel 1 až k a tedy $k^2 \geq n$.

Jestliže pro dané n vypíše program posloupnost, jejíž nejdelší monotónní podposloupnost má délku $\lceil \sqrt{n} \rceil$ (pro neznalé horní celá část odmocniny z n), je tato posloupnost optimální. Nejobtížnější je to samozřejmě učinit pro $n = k^2$, neboť pro menší n můžeme hledanou posloupnost získat z posloupnosti pro k^2 vynecháním příliš velikých členů. Jedním z mnoha možných řešení je vytváření pro k^2 posloupností sestavených z k klesajících bloků: $3, 2, 1, 6, 5, 4, 9, 8, 7$ ($k = 3$). Analogicky pro větší k.

Napsat program realizující daný algoritmus je nyní již snadné. Časová složitost je (při dobré realizaci) lineární, paměťová je dokonce konstantní.

```
#include <stdio.h>

int main ()
{
    int i, j, k, n;
    scanf ("%d", &n);
    for (k=0, i=n; i>0; k++)
        i-=2*k+1;                                   /* horní celá část odmocniny */
    for (i=0; i<k*k; i++)
        if ( (j=i-2* (i%k)+k)<=n) printf ("%d\n", j);
                                                    /* generujeme přímo pro druhou mocninu */
    return 0;
}
```

10-2-3 Turingův stroj — Martin Mareš

U této úlohy se nám sešlo relativně málo řešení, zato většina z nich opravdu fungovala. Základní problém byl ovšem v určování časové a paměťové složitosti – i u Turingova stroje samozřejmě mají tyto pojmy rozumný smysl: paměťová složitost algoritmu je počet použitých políček pásky (řídící tabulka stroje se do ní nepočítá, jelikož je konstantní pro všechna vstupní data a u běžných programů se také do paměťové složitosti program nepočítá), časová složitost pak celkový počet kroků výpočtu. Uvědomte si, že paměťová složitost nikdy nemůže přesáhnout časovou (v každém kroku výpočtu můžeme použít maximálně jedno dosud nepoužité políčko) a že na rozdíl od obvyklých programovacích jazyků zde má smysl nejen asymptotické vyjádření složitosti, ale i přesná čísla (my si ovšem vystačíme s tvarem asymptotickým).

Základní otázkou ovšem je, jakou representaci čísel si zvolit: nejjednodušší by bylo užít „jedničkovou soustavu“ (též řečenou jedničkový aditivní kód – prostě číslo n je zapsáno n jedničkami), ale u ní je základní nevýhodou velká paměťová (a tím pádem i časová) náročnost algoritmu (násobíte-li čísla m a n, potřebujete paměť $O(m \times n)$ na uložení výsledku. Na druhou stranu, násobit čísla v desítkové soustavě by bylo zbytečně složité, zvolíme proto soustavu dvojkovou. Čísla budeme zapisovat tak, že nalevo (tak budeme říkat poloze nejblíže k začátku pásky) bude číslice nejnižšího řádu, napravo pak nejvyššího.

Pokud máme více čísel, můžeme je jednoduše zapsat za sebe a oddělit nějakým speciálním znakem (nabízí se kupříkladu Λ). Jelikož však jsme ve stavech stroje schopni uchovat pouze konečně mnoho informace a zpracovávaná čísla nejsou nijak omezena, musíme pak libovolnou operaci s nimi (tedy i prosté zkopírování jinam) provádět po malých kouscích a neustále skákat od jednoho čísla k druhému, čímž algoritmus oproti programovacím jazykům dovolujícím přímou adresaci paměti zpomalíme řádově tolikrát, kolik je vzdálenost začátků čísel od sebe (a to je minimálně délka kratšího z nich).

Existuje ovšem jednoduchý trik, s jehož pomocí se lze tomuto poskakování mezi čísly vyhnout: prostě obě čísla zapsat na totéž místo – buďto proloženě (tedy číslice střídavě z jednoho a z druhého čísla), nebo místo dvou číslic použít čtyři a každou novou číslicí kódovat jednu číslici prvního čísla a jednu (téhož řádu) druhého. Tak všechny algoritmy procházející obě čísla „zleva doprava“ dokážeme provést v lineárním čase. My si pro náš stroj zvolíme metodu první (prokládání), jež vede na menší abecedu za cenu zvětšení počtu stavů.
Páska stroje bude při násobení $x = a \cdot b$ vypadat takto:

$$a_0 a_1 \ldots a_n \heartsuit b_0 x_0 b_1 x_1 \ldots,$$

přičemž nevyužité číslice b a x budou mít hodnotu Λ, chápanou jako nulu. Navíc si v průběhu výpočtu budeme již zpracovaná a_i odškrtávat znakem $\sharp$. Abeceda stroje tedy jest $\Sigma = \{\mathbf{0}, \mathbf{1}, \sharp, \heartsuit, \Lambda\}$.

A nyní již k algoritmu: násobíme-li číslo b číslem a zapsaným ve dvojkové soustavě jako $a = \sum_i a_i 2^i$, je jistě $x = a \cdot b = \left(\sum_i a_i 2^i\right) \cdot b = \sum_i a_i 2^i \cdot b$. Toto nám přesně dává školský algoritmus na násobení zapsatelný takto:

1. $x \leftarrow 0$
2. Jestliže $a \bmod 2 = 1$, pak $x \leftarrow x + b$
3. $a \leftarrow \lfloor a/2 \rfloor$
4. $b \leftarrow 2 \cdot b$
5. Pokud $a \neq 0$, pokračuj krokem 2.

Přeformulováno pro Turingův stroj: Najdi nejlevější číslici a_i, která dosud není škrtnuta a škrtni ji. Pokud to byla jednička, přičti k x hodnotu b. V každém případě pak vynásob b dvěma (to jest posuň o řád doprava). Toto opakuj, dokud zbývají v a neškrtnuté číslice.

Následující tabulka definující náš stroj (S je počáteční stav; proškrtnutá okénka stroj pří korektním zadání nemůže použít) snad ani nepotřebuje dalších komentářů:

stav	**0**	**1**	♯	♡	Λ
S	$S_0/\sharp/R$	$S_1/\sharp/R$	—	$E/\heartsuit/L$	—
S_1	$S_1/\mathbf{0}/R$	$S_1/\mathbf{1}/R$	—	$A/\heartsuit/R$	—
A	$A_0/\mathbf{0}/R$	$A_1/\mathbf{1}/R$	—	—	$B/\Lambda/L$
A_0	$A/\mathbf{0}/R$	$A/\mathbf{1}/R$	—	—	$A/\mathbf{0}/R$
A_1	$A/\mathbf{1}/R$	$C/\mathbf{0}/R$	—	—	$A/\mathbf{1}/R$
C	$A_1/\mathbf{0}/R$	$C_1/\mathbf{1}/R$	—	—	$B/\mathbf{1}/L$
C_1	$C/\mathbf{0}/R$	$C/\mathbf{1}/R$	—	—	$C/\mathbf{0}/R$
B	$B/\mathbf{0}/L$	$B/\mathbf{1}/L$	—	$R_0/\heartsuit/R$	$B/\Lambda/L$
S_0	$S_0/\mathbf{0}/R$	$S_0/\mathbf{1}/R$	—	$R_0/\heartsuit/R$	—
R_0	$K_0/\mathbf{0}/R$	$K_1/\mathbf{0}/R$	—	—	$W/\mathbf{0}/L$
K_0	$R_0/\mathbf{0}/R$	$R_0/\mathbf{1}/R$	—	—	$R_0/\Lambda/R$
R_1	$K_0/\mathbf{1}/R$	$K_1/\mathbf{1}/R$	—	—	$W/\mathbf{1}/L$
K_1	$R_1/\mathbf{0}/R$	$R_1/\mathbf{1}/R$	—	—	$R_1/\Lambda/R$
W	$W/\mathbf{0}/L$	$W/\mathbf{1}/L$	$S/\sharp/R$	$W/\heartsuit/L$	$W/\Lambda/L$
E	—	—	$E/\sharp/L$	—	—

A jak je to s časovou složitostí? Při zpracování každé cifry čísla a „přejedeme“ celou využitou část pásky doprava, pak se o kus vrátíme, pak opět doprava a pak se opět vrátíme ⇒ celkem tedy $O(\log a + \log b)$, toto vykonáme $(\log a)$-krát, což dává celkem $O(\log a \cdot (\log a + \log b))$.

Zkuste si promyslet, jak by se toto řešení změnilo, když bychom místo prokládání čísel na pásce zkusili použít větší abecedu…

10-2-4 Bynáriho logaritmus — Vít Novák

Většinou jste přišli na to, že pro Bynáriho logaritmus je rozhodující první jednička daného čísla ve dvojkovém zápisu. Většinou jste také přišli na to, že mocniny dvojky je třeba zpracovat zvlášť nebo si na ně dát aspoň pozor, protože u nich je logaritmus roven exponentu v mocnině, zatímco u ostatních čísel je o jedna vyšší, než pořadí nejvyššího nenulového bitu.

Zbývá tedy vyřešit, jak najít nejvyšší bit. A tady nám pomůže půlení intervalu. Nejvyšší jednička totiž může být buď v prvních 512 bitech, nebo ve druhých. To zjistíme prostým porovnáním našeho čísla s 2^{512}, tedy vlastně s `1 << 512` (pro pascalisty: `1 shl 512`). Řekněme, že je ve vyšších 512 bitech. Pak může být buď ve 256 nejvyšších, nebo zas v té druhé půlce, zjistíme porovnáním s `1 << (256+512)`...

Potřebujeme tak $\log_2 1024 = 10$ kroků (mimochodem, všimli jste si, že 1024 je také číslo úlohy? – pozn. M.M.)

```
#include <stdio.h>

#define bsize_of_int sizeof (int) * 8              /* Bitová šířka intu */

int main (void)
{
  int n, b_log, krok;
  krok = b_log = bsize_of_int/2;
  scanf ("%d", &n);
  for (; ; )
    {
      krok = krok >> 1;
      if (n < (1<<b_log))
        b_log-=krok;
      else if (n >= (1<< (b_log+1)))
        b_log+=krok;
      else break;
    }
  printf ("%d\n", b_log);
  return 0;
}
```

10-2-5 Kostky jsou vrženy — Martin Bělocký

Triviálně lze úlohu řešit setříděním kostek a pak vybráním prvních K největších. To je ovšem zbytečně pomalé (v čase alespoň $O(N \cdot \log N)$), neboť je zbytečné třídit příliš lehké kostky (ty nás nezajímají) ani příliš těžké (to po nás nikdo nechce).

Úlohu jste řešili dvěma způsoby: za předpokladu, že Baron Bynári je negramotný, jste se pouze soustředili na algoritmus nalezení K největších prvků z pole a prvky jste v poli pouze řadili a nepamatovali jste si jednotlivé výsledky vážení.

Někteří z vás se snažili minimalizovat počet vážení tím, že si pamatovali v jiné datové struktuře výsledky jednotlivých vážení, aby se nevážily koule, které již byly jednou vzájemně zváženy, nebo navíc někteří na základě předchozích vážení byli schopni říct, jak některé konkrétní vážení dopadne (např. vím-li, že $1 > 2$ a $2 > 3$, dá se odvodit, že $1 > 3$). Tímto řešením jste ale předpokládali gramotnost Bynáriho (musel by si výsledky zapisovat). Toto je však už řešení obecnějšího problému, než bylo myšleno zadáním. K tomuto řešení byste potřebovali další datovou strukturu – tabulku, do níž byste si zapisovali výsledky a podle konkrétních vážení doplňovali další relace mezi kostkami vyplývající z vážení. Doplňování relací by navíc zvětšilo celkovou složitost původního algoritmu, který by bez tohoto pracoval v průměru v lineárním čase vůči počtu kostek. Dále se tedy zabývejme pouze algoritmem nalezení K největších kostek řazením.

Náš algoritmus je modifikací známého třídícího algoritmu Quicksort. Kostky dáme do řady a budeme se na ně odkazovat pomocí indexu v řadě.

Vybereme si nějakou kostku v řadě, označme ji X. Rozdělíme řadu na tři části, do levé dáme kostky těžší než X, do pravé lehčí než než X a uprostřed rovné X. Nechť je počet kostek v levé části L, pravé P, uprostřed U. Máme-li najít K největších kostek a $K < L$, pak opakujeme algoritmus od začátku, ale vstupem bude pouze těchto L kostek. Pokud $K > L + U$, pak jsme našli $L + U$ hledaných kostek a opakujeme algoritmus od začátku s tím, že už hledáme pouze v pravé části s P kostkami a hledáme $K - L - U$ kostek. V ostatních případech jsme našli K největších kostek: je to L kostek vlevo a z prostředku vezmeme prvních $K - L$ kostek (neboť uprostřed jsou všechny stejně těžké).

Jinak řečeno: náš algoritmus najde K-tý největší prvek a všechny prvky nalevo od něj jsou díky vhodnému řazení větší nebo stejné.

Důkaz správnosti: Algoritmus skončí, když bude K v prostřední části, což bude nejpozději, když bude na vstupu jedna kostka, a protože v každém kroku algoritmu rozdělíme řadu kostek na alespoň dvě menší části (jinak bychom ihned skončili – vše by bylo uprostřed). Nalézáme postupně nejtěžší kostky v levé části a příliš lehké zahazujeme a v každém kroku algoritmu platí $K >$ poč. už nalezených $+ L + U$, takže o žádnou kostku ve správném výsledku nepřijdeme.

Složitost: Záleží na výběru kostky X. Kdybychom dokázali pokaždé vybrat prostřední kostku co do váhy, rozdělili bychom řadu kostek (když by se v řadě nevyskytovaly stejně těžké kostky) nejprve na polovinu, pak na čtvrtinu, osminu,..., tedy počet vážení by se dal odhadnout na $N + N/2 + N/4 + ... + 1 = 2 \cdot N$, tj. počet vážení je lineárně závislý na počtu kostek $O(N)$.

Když bychom vybírali pokaždé nejhorší možnost, tj. za X bychom vybrali např. nejlehčí kostku, počet vážení by byl přibližně $N + (N-1) + (N-2) + \ldots + K$, což je v součtu kvadratický počet vážení vůči počtu kostek – $O(N^2)$. O průměru

se však dá říci (leč přesné spočítání průměrného chování je bohužel nad rámec našeho semináře), že se bude provádět $O(N)$ vážení. Důkaz je velice podobný důkazu složitosti algoritmu Quicksort.

```
program K_nejvetsich;
{Nalezeni K nejvetsich prvku metodou zalozenou
 na modifikaci tridiciho algoritmus Quicksort}
uses crt;

const MaxN = 1000;     {maximalni pocet zpracovavanych cisel}
type  Pole = array[1..MaxN] of integer;      {ulozeni cisel}

var P: Pole;     {ulozeni tridenych udaju}
    N: 1..MaxN;  {pocet prvku v poli P}
    pom,I,K: integer;
    s:text;

function porovnej(a,b:integer):char;
var z:char;
begin
if a=b then porovnej:='='  {aby tutez kostku Bynari nezkousel davat na 2 misky najednou:-) }
else begin
  writeln('poloz na vahy kostku cislo ',a,' a kostku cislo ',b);
  writeln('jak se naklonily vahy ? ( <, >, = )');
  z:=readkey; porovnej:=z;
end end;

procedure NaleztKty(var P:Pole; Zac,Kon,K:integer);
{v poli celych cisel P v useku od indexu Zac do indexu Kon
vyhleda K-te nejmensi cislo }
var X: integer;     {hodnota pro rozdeleni na useky}
    Q: integer;     {pomocne pro vymenu prvku v poli}
    I,J: integer;   {posouvane pracovni indexy v poli}
begin
  while Zac < Kon do
    begin
    X:=P[K];        {jedna mozna volba, lze i jinak}
    I:=Zac;
    J:=Kon;
    repeat
      while porovnej(P[I],X)='>' do I:=I+1;
      while porovnej(P[J],X)='<' do J:=J-1;
      if I < J then       {vymenit prvky s indexy I a J}
        begin
        Q:=P[I]; P[I]:=P[J]; P[J]:=Q;
        I:=I+1; J:=J-1;   {posun indexu na dalsi prvky}
        end
      else if I = J then  {oba indexy ukazuji na hodnotu X}
        begin
        I:=I+1; J:=J-1    {posun indexu na dalsi prvky
                           - nutne kvuli ukonceni cyklu}
        end
    until I > J;
    {usek <Zac,Kon> je rozdelen na useky <Zac,J> a <I,Kon>}
    if K < I then Kon:=J; {dal budeme hledat v levem useku}
    if K > J then Zac:=I; {dal budeme hledat v pravem useku}
  end;   {mame k-ty nejvetsi prvek a pred nim jsou jen prvky >= }
```

```
  writeln('Nalezeno ', K, ' nejvetsich prvku: ');
  for i:=1 to k do write(P[i],',');
  writeln;
end;

begin  {hlavni program}
clrscr;
write('Pocet kostek: '); readln(N);
for I:=1 to N do P[I]:=I;   {nezname vahy cisel, diktujeme jen indexy !!!}
write('Kolik nejvetsich nalezt?: '); readln(K);
NaleztKty(P,1,N,K);
end.
```

10-3-1 Domečkologie — Martin Mareš

Jelikož se jedná o problém z teorie grafů, nebude od věci, když nejdříve zadefinujeme pár užitečných pojmů a dokážeme dvě více méně triviální tvrzení. Je pravda, že by bylo možno náš algoritmus odvodit i bez tohoto „teoretického abrakadabra“, ale pokusme se pro jednou ukázat, jak se věci dělají pořádně. Vše budeme zavádět pro *multigrafy*, což jsou grafy, v nichž může mezi dvěma vrcholy vést libovolný počet hran.

[Záměrně ukazujeme několik postupně se zlepšujících algoritmů – počínaje algoritmem nefunkčním přes algoritmus složitý až k finálnímu vzorovému programu – abychom ukázali nejen, jak řešení úlohy vypadá, ale také jak se k němu dojde, což ve většině učebnic, ať již programování či matematiky, chybí.]

Definice:

- *stupeň vrcholu* je počet hran, které z vrcholu vychází. Uvědomte si, že součet všech stupňů vrcholů je roven dvojnásobku počtu hran (každá hrana má dva konce), což je sudé číslo, takže vrcholů lichého stupně musí být sudý počet.
- *sled* je posloupnost vrcholů a hran $v_0 h_1 v_1 \ldots h_n v_n$, kde hrana h_i spojuje vrcholy v_{i-1} a v_i.
- *tah* je sled, v němž se neopakují hrany.
- *uzavřený tah* je tah, v němž $v_0 = v_n$.
- *otevřený tah* je tah, který není uzavřený.
- *cesta* je sled, v němž se neopakují vrcholy (tudíž ani hrany).
- *kružnice* je sled, v němž je $v_0 = v_n$ a jinak se žádné vrcholy (tedy ani hrany) neopakují.
- Graf je *souvislý* právě tehdy, když mezi každými dvěma vrcholy existuje cesta tyto vrcholy spojující (což je totéž, jako když mezi nimi existuje sled, uvědomte si, proč).
- *Komponenta souvislosti* je podgraf, který je souvislý a již k němu nelze doplnit žádnou hranu původního grafu, aniž by se souvislost porušila.

- Tah se nazývá *eulerovský,* pokud obsahuje všechny hrany daného grafu.

Věta (První Eulerova). V grafu existuje uzavřený eulerovský tah $\iff$ graf je souvislý a všechny vrcholy mají sudé stupně.

Důkaz. $\Rightarrow$: Kdyby graf nebyl souvislý nebo v něm existoval vrchol lichého stupně, je zjevné, že v něm nemůže být eulerovský tah. Zbývá tedy dokázat implikaci opačným směrem...

$\Leftarrow$: Vezměme nejdelší možný tah v našem grafu. Pokud není uzavřený, jistě má nějaký koncový vrchol, tohoto vrcholu se ovšem tento tah dotýká lichý-počet-krát (jednou na začátku a pak vždy vejde a vyjde a skončí jinde), ale tento vrchol má sudý stupeň, takže existuje nepoužitá hrana z tohoto vrcholu vedoucí a o tu můžeme tah prodloužit $\Rightarrow$ nebyl nejdelší a jsme ve při. Kdyby nebyl eulerovský, existuje hrana e, která v tahu není obsažena a přitom jeden z jejích vrcholů v v tahu obsažen je (graf je souvislý – pokud existuje jediná hrana e' mimo tah, vezmeme cestu z jejího libovolného krajního vrcholu do nějakého vrcholu v' tahu a za e prohlásíme první hranu na této cestě [ve směru od v'], jež v tahu není). A ve vrcholu v můžeme náš uzavřený tah rozpojit a k libovolnému z jeho konců hranu e přidat, čímž jsme jej opět prodloužili. Spor.

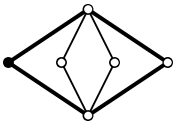
Věta (Druhá Eulerova). V grafu existuje otevřený eulerovský tah $\iff$ graf je souvislý a existují v něm právě dva vrcholy lichého stupně.

Důkaz. Směr $\Rightarrow$ je opět zřejmý. Zpět: Pokud existují dva vrcholy v_1, v_2 lichého stupně, můžeme mezi ně doplnit hranu $\hat{e}$ (jelikož pracujeme s multigrafy, nevadí, že už tam jedna taková může být), čímž vznikne graf vyhovující podmínkám předchozí věty, tudíž v něm existuje uzavřený eulerovský tah a odstraníme-li z něj přidanou hranu $\hat{e}$, dostaneme otevřený eulerovský tah v původním grafu.

Nyní je zřejmé, že zadání úlohy po nás vlastně chce nalézt v daném (multi)grafu jakýkoliv eulerovský tah a že souřadnice vrcholů jsou naprosto zbytečné. Z předchozích vět je jasné, jak vypadají nutné a postačující podmínky jeho existence. Zkusíme úlohu vyřešit a použijeme k tomu...

Hladový algoritmus. Vybereme si, kde začneme (pokud existují vrcholy lichého stupně, tak v některém takovém, jinak na tom nezáleží, jelikož tah bude uzavřený) a poté budeme tah rozšiřovat o další hrany, dokud to půjde. Na konci nutně dojdeme do druhého vrcholu lichého stupně nebo tam, kde jsme začali (kdybychom došli někam jinam, mohli bychom ještě pokračovat někam dál, protože bychom končili ve vrcholu sudého stupně, v němž jsme nezačali, a tak by musela existovat ještě další nepoužitá hrana pryč).

Vyvrácení. Ačli tento algoritmus najde uzavřený tah, respektive tah se správným začátkem a koncem, nemusí tento tah být eulerovský. Příkladem budiž například následující graf (plným kolečkem je vyznačen počáteční = koncový vrchol, zvýrazněny jsou hrany nalezeného tahu):



Oprava. Stalo se tedy, že v grafu zbyly ještě nějaké hrany v nalezeném tahu T neobsažené. Ale jak vypadá graf složený z těchto hran? Uvědomte si, že jistě jsou všechny stupně vrcholů sudé (odečetli jsme od všech stupňů vrcholů sudá čísla mimo vrcholů lichého stupně, od nichž jsme odečetli čísla lichá), takže v každé komponentě c tohoto grafu existuje uzavřený eulerovský tah T_c, který má s tahem T alespoň jeden společný vrchol v_c, v němž můžeme oba tahy rozpojit a vsunout T_c do T. Jenže tahy T_c můžeme nalézt rekurzivní aplikací téhož algoritmu…

Tento algoritmus úlohu řeší, ba dokonce při vhodné implementaci v čase $O(M)$ (počet hran budeme značit M, počet vrcholů N), nicméně je značně komplikovaný (rekurzivní dekompozice grafu, udržování a slepování seznamů apod.). Pokusíme se jej ještě trochu zjednodušit… Zkonstruujeme hladovým algoritmem nějaký základní tah T_0 a nalezneme první vrchol v, z nějž vede nepoužitá hrana e (z důkazu Eulerovy věty víme, že buďto taková hrana neexistuje, nebo je graf eulerovský a nebo nesouvislý – to, který případ nastal, vyřešíme snadno počítadlem zbývajících hran), načež zkonstruujeme uzavřený tah T_e ze zbylých hran, který v e začíná i končí, připojíme jej do T_0 a takto pokračujeme, dokud existují nepoužité hrany s tahem sousedící. Povšimněte si, že při přidávání takovýchto „podtahů“ nemůže nikdy vzniknout nepoužitá hrana dotýkající se vrcholu, který by na našem tahu byl *před* vrcholem v, takže lze vše vyřešit frontou na dosud nezpracované vrcholy v čase $O(M)$ a paměti $O(M)$ takto:

Algoritmus F (Frontální útok).

1. Spočítáme vrcholy lichého stupně, pokud jich je > 2, úloha nemá řešení, jinak zvolíme počáteční vrchol v_0 jako jeden z lichých, případně libovolný, pokud jsou všechny sudé.
2. Mějme frontu Q, z počátku obsahující počáteční vrchol v_0.
3. Vezměme vrchol v z fronty Q a ten vypišme (víme, že je zaručeně ve finálním tahu T – viz předchozí odstavec).
4. Pokud existuje dosud nepoužitá hrana e vedoucí z v, vydáme se touto hranou a pokračujeme dalšími nepoužitými hranami, dokud to jde. Všechny navštívené vrcholy přidáváme na konec fronty.
5. Opakujeme 3–4, dokud se fronta nevyprázdní.
6. Zbyla-li v grafu jakákoliv hrana, graf byl nesouvislý. Jinak jsme vypsali eulerovský tah (správnost zřejmá z tvrzení dokázaných výše).

Do třetice. Celý algoritmus ještě můžeme zjednodušit tím, že využijeme *prohledávání grafu do hloubky* jednoduchým rekurzivním algoritmem. Místo fronty tak použijeme zásobník (tím se pouze tah otočí, což není na závadu), který nám rekurze zajistí sama bez nutnosti implementovat si jej ručně. Časová ani paměťová složitost se nezmění. Vzorový program vychází z této varianty a jeho činnost by měla být zřejmá z komentářů v programu, jedině jest snad záhodno upozornit na trik s funkcí `XOR`, jímž se páruje hrana „tam“ s hranou „zpátky“.

```
#include <stdio.h>

#define MAXN 100                    /* Maximální počet vrcholů a hran */
#define MAXM 100*100

int N, M;                           /* Počet vrcholů a hran */
int d[MAXN];                        /* Stupně vrcholů */
int f[MAXN];                        /* První hrana z každého vrcholu */

struct edge {                       /* Hrana */
    int y;                          /* Kam vede (y = -1 pro použité hrany) */
    int n;                          /* Další z téhož vrcholu */
} e[2*MAXM+2];                      /* e xor 1 je opačný směr */

void go (int j)                     /* Prohledáváni grafu do hloubky */
{
    int m, y;

    while (m = f[j])                /* Bermež všechny hrany m vedoucí z j */
      {
        f[j] = e[m].n;              /* Hranu odebereme */
        if ( (y = e[m].y) >= 0)     /* Pokud ještě nebyla použita... */
          {
            e[m^1].y = -1;          /* ... označíme za použitou 'protisměrnou' hranu */
            go (y);                 /* Zavoláme se rekursivě */
            M--;                    /* A snížíme počet zpracovaných hran */
          }
      }
    printf ("%d\n", j);             /* Vracíme se zpět ⇒ vypisujeme vrcholy */
}

int main (void)
{
    int i, j, k, m;

    scanf ("%d", &N);               /* Načteme vstup */
    for (i=1; i<=N; i++)
      scanf ("%d%d", &j, &k);       /* Souřadnice ignorujeme, jsou nanic */
    scanf ("%d", &M);
    m = 2;
    for (i=1; i<=M; i++)
      {
        scanf ("%d%d", &j, &k);
        e[m].n = f[j];              /* Zařadíme jeden směr... */
        f[j] = m;
```

```
            e[m++].y = k;
            d[j]++;
            e[m].n = f[k];          /* ... a druhý. */
            f[k] = m;
            e[m++].y = j;
            d[k]++;
        }
    j = 1;                          /* Hledáme startovní vrchol */
    k = 0;
    for (i=1; i<=N; i++)
        if (d[i] % 2)
            j = i,  k++;
    if (k <= 2)
        go (j);
    if (M)                          /* Pokud něco zbylo, řešení neexistuje */
        puts ("Nemá řešení!");
    return 0;
}
```

10-3-2 Sirky jsou vrženy — Daniel Kráľ

Bohužel zadání nebylo příliš šťastně zformulováno i stalo se, že přicházela různá řešení (to je obvyklé) různých úloh, všechna ovšem označená jako `KSP-10-3-2` (to už je méně obvyklé). Někteří z Vás považovali ono magické k ze zadání úlohy za maximální počet sirek, které lze v jednom tahu odebrat, jiní za jediný možný počet sirek, který lze odebrat a další za blíže neurčenou proměnnou. Protože většina z Vás považovala k za konstantu určující, kolik sirek lze odebrat z jedné, druhé nebo z obou hromádek v jednom kole, rozebereme zde řešení takto zadané úlohy. Maximální počet bodů udělovaných za správné a úplné řešení se pohyboval dle obtížnosti vyřešené varianty od 10 do 13 bodů.

Nechť na hromádkách je n a m sirek. Protože lze odebírat pouze k sirek, záleží výsledek pouze na velikosti celých částí podílů n/k a m/k. Stačí tedy vytvořit algoritmus za předpokladu, že $k = 1$; řešení původní úlohy získáme aplikací tohoto algoritmu na případ, kdy počty sirek na hromádkách jsou celé části podílů n/k a m/k, a odebíraný počet sirek potom vynásobit k.

Stav, kdy na jedné hromádce je a a na druhé b sirek budu pro stručnost označovat jako $[a, b]$. Stav $[x, y]$ nazvu kritický, jestliže neexistuje (dovolený) tah do stavu $[u, v]$ takového, že $[u, v]$ je kritický, nebo platí-li, že $x = y = 0$. Stav $[x, y]$ nazvu vyhrávající, jestliže není kritický. Takto zavedené označení stavů je korektní, neboť mohu stavy $[x, y]$ roztřiďovat na kritické a vyhrávající v neklesajícím pořadí dle součtu $x + y$ — postup je stejný jako v důkazu následujícího tvrzení: Pro stav $[x, y]$ existuje tah, který mi zaručí výhru, právě tehdy, pokud stav $[x, y]$ je vyhrávající. Pro stav $[x, y]$ takový tah neexistuje

právě tehdy, pokud je tento stav kritický. Navíc stav $[x, y]$ je kritický, pokud x i y jsou obě sudá čísla.

Důkaz provedeme matematickou indukcí podle velikosti součtu $x + y$. Pro $x + y = 0$ tvrzení platí – jsou-li obě hromádky prázdné, tak jsem prohrál a nula je sudé číslo.

Nechť tvrzení platí pro všechny stavy $[u, v]$, kde $u + v < x + y$. Je-li stav $[x, y]$ kritický, potom všechny stavy $[u, v]$, do kterých lze přejít, jsou vyhrávající, a tedy z nich existuje tah, který zaručí soupeři výhru. Proto po mém libovolném tahu, může udělat soupeř tah, který mu zaručí výhru a tedy žádný tah z tohoto stavu nemůže zaručit výhru mně. Ze stavu $[x, y]$ lze přejít do stavů $[x - 1, y]$, $[x, y - 1]$ a $[x - 1, y - 1]$. Potom ale (dle indukčního předpokladu) jsou obě čísla x a y sudá, neboť jinak by jeden z těchto tří stavů byl kritický, což by byl spor s tím, že stav $[x, y]$ je kritický.

Je-li stav $[x, y]$ vyhrávající, potom existuje tah do stavu $[u, v]$, který je kritický. Tento tah je tím tahem, který mi zaručí výhru, neboť ať soupeř ze stavu $[u, v]$ přejde do libovolného stavu $[s, t]$, ten bude určitě vyhrávající, neboť $[u, v]$ je kritický, existuje z tohoto stavu tah, který mi zaručuje výhru. Důkaz, že alespoň jedno z čísel x a y je liché se provede stejně jako v předchozím případě.

Vyhrávající strategie tedy vypadá tak, že se snažíme, aby soupeř vždy táhl ze stavu $[x, y]$, kde x i y jsou sudá čísla. To se nám povede právě tehdy, pokud alespoň jedno z těchto čísel je liché, což lze udělat s konstantní časovou i paměťovou složitostí.

10-3-3 Lloydova devítka — Robert Špalek

Počet možných konfigurací herního plánu je vlastně počet všech permutací 8 kostiček plus jedné díry, tedy $9! = 362880$. Nejkratší posloupnost tahů vedoucí k požadovanému stavu lze při tomto počtu efektivně spočítat prohledáváním do šířky. Budeme předpokládat, že pracujeme na 32-bitovém kompilátoru a nemusíme se starat o velikost pole.

Herní plán si představíme jako graf. Vrcholy budou možné permutace kostiček a hrany budou spojovat ty permutace, které se liší pouze jedním tahem, tedy posunutím díry o 1 políčko libovolným směrem. V tomto grafu hledáme nejkratší cestu z jednoho vrcholu do druhého. Prohledávání grafu do šířky funguje následujícím způsobem:

1. *Označíme všechny vrcholy grafu jako neprobádané, pouze počáteční vrchol označíme jako nalezený nicméně ještě nezpracovaný.*
2. *Vezmeme všechny nalezené nezpracované vrcholy a pro každý z nich prohledáme všechny jeho sousedy. Pokud některý z nich byl dosud neznámý, označíme ho k zpracování v dalším tahu a zapíšeme do něj směr, odkud jsme k němu došli.*
3. *Krok 2 opakujeme tak dlouho, dokud nalézáme nové vrcholy.*

Po ukončení práce algoritmu je v každém dosažitelném vrcholu uložen směr, kterým se máme vydat, abychom došli nejkratší možnou cestou k cíli. Vrcholy, které nebyly tímto algoritmem zpracovány, jsou nedosažitelné. U Lloydovy devítky se nám množina všech permutací rozdělí na dvě poloviny: na sudé a liché permutace. To znamená, že celá jedna polovina všech konfigurací je nedosažitelná a úloha pro ní nemá řešení.

V programu používám funkce `perm2int` a `int2perm` pro vzájemně jednoznačný převod permutací na přirozená čísla, který je diskutován v úloze 10-3-5. Permutace jsou označeny čísly $0, 1, \ldots, 9! - 1$, permutace číslo 0 je základní pozice. V celém programu je použita tato konvence zápisu permutací: p_i znamená pozici ve čtverci, na níž je uloženo číslo i. Pozice ve čtvreci jsou očíslovány čísly $0, 1, \ldots, 8$ po řádcích.

Při prohledávání do šířky funkcí `search` ukládám do pole informaci o příchozím směru tímto způsobem:

- 0 označuje nezpracovanou permutaci,
- 1–4 označují plně zpracovanou permutaci, do níž jsme přišli z příslušného směru (1 → shora, 2 → zdola, 3 → zleva, 4 → zprava),
- 5–8 označují permutaci určenou pro zpracování v tomto tahu,
- 9–12 označují právě nalezené permutace, které se budou zpracovávat až v příštím tahu.

Toto označení je zvoleno, protože vyhledávání do šířky probíhá po hladinách s rostoucí vzdáleností od počátku. Abychom nemuseli konstruovat frontu, do které by se ukládaly vrcholy čekající na zpracování, označujeme si nezpracované permutace přímo v poli. Tedy kromě příchozího směru označujeme také stadium zpracování, aby se nám nepletly hladiny. Po každém průchodu označíme zpracované permutace jako stabilní a nově nalezené určíme pro zpracování v dalším kole.

Při hledání cesty i při zpětném průchodu se využívá funkce `move`, která zjistí, zda je vůbec daný tah proveditelný, a pokud ano, pak jej provede. Právě tato funkce je jediná v programu, která určuje pravidla hry. Pokud bychom ji nahradili čímkoliv jiným, dostali bychom program pro hledání strategie v libovolné jiné hře tohoto typu.

Zpětný průchod funkcí `animate` již pouze rekonstruuje cestu. Při hledání úmyslně začínáme ve startovním vrcholu, neboť při rekonstrukci postupu, jak danou konfiguraci poskládat, stačí v 1 průchodu jít podle šipek a vypisovat tahy, ale hlavně proto, že s jedním předpočítaným polem umíme najít nejkratší cesty ze všech dosažitelných konfigurací. Při výpisu mezistavů musíme vypočítat inverzní permutaci, neboť na rozdíl od konvence použité ve zbytku programu potřebujeme zjistit, jaké číslo je na pozici i.

Paměťová složitost programu je $O(n!)$, kde $n = 9$ je obecně počet políček ve čtverci. Časová složitost je $O(kn!)$, kde k je délka nejdelší z nejkratších cest

(neboť tolikrát musíme projet celé pole). Obecnou závislost k na n nebude pravděpodobně snadné zjistit.

Možná vylepšení algoritmu:

1. Kdybychom chtěli algoritmus vylepšit např. tím, že by se zastavil v průběhu výpočtu ve chvíli, kdy nalezne cílovou konfiguraci, nepomohlo by nám to, protože ačkoliv je pro $n = 9$ její vzdálenost maximálně 31 (viz. výpisy programu v průběhu výpočtu), musíme projít při prohledávání do šířky také plno jiných pozic.
2. Paměťová náročnost lze výrazně zmenšit, pokud nepoužijeme v poli permutací pro každou z nich 1 bajt, ale pouze 3 bity: 2 bity pro směr, odkud jsme přišli, a 1 bit jako flag zpracování. To ale algoritmus značně zkomplikuje, s polem znaků se pracuje přeci jenom lépe než s bitovým polem.
3. Časovou náročnost lze zlepšit z $O(kn!)$ na $O(n!)$, pokud permutace určené ke zpracování nebudeme označovat flagem v poli, ale vytvoříme si frontu. Pak ušetříme 1 bit v poli, ale musíme vyhradit 3 bajty ve frontě, maximální délka fronty je pro $n = 9$ experimentálně zjištěná < 25000. Nejvýrazněji ušetříme čas, neboť každá permutace se ve frontě objeví maximálně jednou. Bohužel pro obecné n není zřejmé, jak velká fronta bude potřeba.

```
#include <stdio.h>
#include <stdlib.h>

#define LENGTH 3                    /* velikost hracího plánu */
#define SIZE   (LENGTH*LENGTH)

int count;                          /* počet permutací kostiček */
char *find;                         /* pole pro prohledávání do šířky – viz text */

int perm2int (int *perm) {          /* převod permutace na číslo */
  int order[SIZE];
  int i, j, idx=0;
  for (i=0; i<SIZE; i++)            /* i. volné číslo je na pozici i */
    order[i]=i;
  for (i=0; i<SIZE; i++) {          /* přidej další člen */
    idx=idx*(SIZE-i)+order[perm[i]];
    order[perm[i]]=-1;
    for (j=perm[i]+1; j<SIZE; j++)  /* sniž o 1 pořadí následujících cifer */
      order[j]--;
  }
  return idx;
}

void int2perm (int idx, int *perm) {  /* převod čísla na permutaci */
  int order[SIZE];
  int i, j, divide=count;
  for (i=0; i<SIZE; i++)            /* i. volné číslo je na pozici i */
```

```
        order[i]=i;
    for (i=0; i<SIZE; i++) {
        divide /= SIZE-i;
        perm[i]=order[j=idx/divide];
        idx %= divide;
        for (j++; j<SIZE; j++)          /* na daném místě v pořadí budou vyšší cifry */
            order[j-1]=order[j];
    }
}
void init () {                          /* alokace pole */
    int i;
    count=1;
    for (i=0; i<SIZE; i++)              /* celkový počet permutací */
        count *= i+1;
    if (! (find= (char*)calloc (count, 1)))   /* alokovat paměť na permutace, vymazat ji */
        exit (1);                       /* předpokládáme 32-bitový kompilátor */
    find[0]=1+4;                        /* začínáme hledat z uspořádaného stavu */
}
void done () {                          /* uvolnění paměti */
    free (find);
}
int move (int *perm, int vect) {        /* provede na permutaci daný tah 1–4 */
    int where, i;
    switch (vect) {
    case 1:                             /* dolů */
        if (perm[SIZE-1]>=6) return 1;    /* kontrola: není díra ve spodní řadě? */
        else where=perm[SIZE-1]+3;
        break;
    case 2:                             /* nahoru */
        if (perm[SIZE-1]<3) return 1;     /* kontrola: není díra v horní řadě? */
        else where=perm[SIZE-1]-3;
        break;
    case 3:                             /* doprava */
        if (perm[SIZE-1]%3==2) return 1;    /* kontrola: není díra v pravém sloupci? */
        else where=perm[SIZE-1]+1;
        break;
    case 4:                             /* doleva */
        if (perm[SIZE-1]%3==0) return 1;
        else where=perm[SIZE-1]-1;    /* kontrola: není díra v levém sloupci? */
        break;
    default:                            /* chybný tah */
        return 2;
    }
    /* nyni prohodíme díru s danou kostičkou, číslo díry je 8, permutace obsahuje pro
       každou kostičku pozici, na níž je uložena */
    i=0;                                /* najdeme, s čím to teda prohazujeme */
    while (i<SIZE && perm[i]!=where) i++;
    perm[i]=perm[8];                    /* kostka je tam, kde byla díra */
    perm[8]=where;                      /* díra je na novém místě */
    return 0;
```

```
}
void search () {
    int added, sum, level;                  /* přidaných v posledním kole, celkem a hladina */
    int p[SIZE], r[SIZE];                   /* pracovní permutace */
    int n;                                  /* číslo nové permutace */
    int i, j;

    level=0;
    added=sum=1;
    while (added) {                         /* dokud jsme posledně nalezli aspoň 1 tah */
      printf ("%2d.level: added=%6d, all=%06d\n", level++, added, sum);
      added=0;                              /* další kolo: */
      for (i=0; i<count; i++)               /* vyzkoušíme všechny permutace */
      if (find[i]>4 && find[i]<=8) {        /* ááá, tu jsme nalezli teprve minule */
        int2perm (i, p);
        for (j=1; j<=4; j++) {              /* zkusme postupně všechny tahy */
          memcpy (r, p, SIZE*sizeof (int));
          if (!move (r, j)) {               /* provedeme jej, testujeme, zda je tah korektní */
            n=perm2int (r);
            if (!find[n]) {                 /* vede na ještě nezpracovanou pozici? */
              find[n]=j+8;                  /* označíme na zpracování */
              added++;
            }
          }
        }                                   /* tahy */
      }                                     /* permutace */
      for (i=0; i<count; i++)               /* označit zpracované permutace */
        if (find[i]>4)                      /* zpracována v tomto tahu nebo je nová? */
          find[i] -= 4;
      sum+=added;
    }
}
void animate (int nr) {      /* 'animuje' ve zpětném průchodu nalezenou cestu */
    const char ch[SIZE+1] = {"12345678."};    /* výpis nalezé cesty */
    int p[SIZE], p1[SIZE];                  /* permutace a její inverze */
    int i, j, level;
    if (!find[nr]) {                        /* nevede-li tam cesta */
      printf ("there's no way\n");
      return;
    }
    int2perm (nr, p);                       /* hledáme zpětně z rozeskládané situace */
    level=0;
    do {
      getchar ();
      printf ("%2d. %6d\n\n", level++, nr);
      for (i=0; i<SIZE; i++)                /* inverzní permutace říká pro každou pozici, */
        p1[p[i]]=i;                         /* co na ní je */
      for (i=0; i<LENGTH; i++) {
        for (j=0; j<LENGTH; j++)
          printf ("%c␣", ch[p1[i*LENGTH+j]]);
        printf ("\n");
```

```
      }
      if (!nr)                          /* už jsme to poskládali? */
        break;
      printf ("\nmoving %d\n", find[nr]);
      /* odkud jsme sem došli? potřebujeme provést opačný pohyb, což nejlépe uděláme
         negací 0. bitu, ale po odečtení 1 */
      i= ( (find[nr]-1)^1)+1;
      move (p, i);
      nr=perm2int (p);
    }while (1);
}
int main () {
    int i;
    init ();
    search ();
    do {
      printf ("perm?␣"); scanf ("%d", &i);
      animate (i);
    } while (i);
    done ();
    return 0;
}
```

10-3-4 Hic sunt leones — Vít Novák

Nejednoho z vás asi překvapí, že optimální algoritmus pro tuto úlohu pracuje s konstatní časovou složitostí. Jak toho lze dosáhnout? Představte si, že mnohoúhelník svislými řezy nařežeme na tenké plátky, dost tenké na to, aby v každém z nich ležely nejvýše dva body mnohoúhelníka. Pokud budeme mít plátky všechny stejné tloušťky t, pak snadno určíme, zda v některém z nich leží daný bod (bude mít index $(x - x_{min})/t$ v poli plátků). A každý plátek je vlastně nejvýše šestiúhelník, takže na rozhodnutí, zda bod leží uvnitř, stačí zjistit polohu vůči jeho šesti stranám. Realizace v `C++` následuje.

Co se týče analytické geometrie zde použité: rovnice přímky má tvar: $a \cdot x + b \cdot y + c = 0$. Pokud do této rovnice dosadíme souřadnice dvou bodů, které přímku určují, dostaneme $a = y_1 - y_2$, $b = x_2 - x_1$ a $c = -(a \cdot x_1 + b \cdot y_1)$, pokud pak do této rovnice dosadíme souřadnice nějakého bodu X, dostaneme 0, pokud X na přímce leží, jinak $q > 0$ resp. $q < 0$ pro jednu resp. druhou polorovinu přímkou určenou.

Trochu černá můra je paměťová složitost, ta může být hodně vysoká, ale tady nebyla rozhodující. Je nepřímo úměrná nejmenšímu rozdílu x-ových souřadnic vrcholů. A to může být hodně málo. Třeba už pro mnohoúhelník $(0, 0)$, $(0.001, 1)$, $(1, 1)$, $(1, 0)$ bude tloušťka plátku rovna 0.001, tedy vznikne 1000 plátků.

Celý program by šlo dále optimalizovat jak na přehlednost, tak na rychlost, ale asi bych si za něj 10 bodů dal, i když bych si při tom nejspíše dost nadával.

```
#include <iostream.h>
#include <stdlib.h>

class CPoint {                          /* Třida reprezentující body */
public:
        double x,y;                     /* Souřadnice bodu */
        void input();                   /* Načtení dat ze vstupu */
};

class CLine {                           /* Pomocná třída reprezentující přímku či úsečky,
                                           která nesmí být svislá. */
        CPoint *A,*B;                   /* Zadána dvěma body */
        double a,b,c;                   /* Předpočítaná data */
public:
        CLine(CPoint &A, CPoint &B);    /* Základní konstruktor */
        int upperThanLine(CPoint &pt);  /* Leží bod nad nebo pod přímkou? */
        int isBetween(CPoint &pt);      /* Leží bod uvnitř úsečky (pokud je na přímce)? */
};

class CPolygon {                        /* Mnohoúhelník */
public:
        int n;                          /* Počet vrcholů */
        CPoint *pt;                     /* Jednotlivé vrcholy (alokováno dynamicky) */
        CPolygon();                     /* Konstruktor: načtení */
        ~CPolygon();                    /* Destruktor */
};

class CSortedPoly {                     /* Indexy bodů mnohoúhelníka setříděné podle x */
        CPolygon *p;
public:
        int *index;                     /* Přístup řeší konstruktor a přetížený op. [] */
        CSortedPoly(CPolygon &poly);
        ~CSortedPoly();
        CPoint &operator [] (int i) {return p->pt[index[i]];};
};

class CCutPoints {                      /* Třída reprezentující "plátky" */
public:
        CPolygon *p;
        int cnt;                        /* Počet plátků */
        int or;                         /* Orientace mnohoúhelníka */
        double dif;                     /* Tloušťka plátku */
        int *upper, *lower;             /* Pro každý plátek horní a dolní
                                         * hranice mnohoúhelníka */
        int maxlow, maxup;
        CCutPoints(CPolygon& poly);
        ~CCutPoints();
};

class CProblem {                        /* Třída řešící naši úlohu */
        CPolygon *p;
        CCutPoints *c;                  /* Konstruktor -- předpočítá data */
public:
        CProblem();
        ~CProblem();
```

```
        int solve(CPoint &X);           /* Určí, zda X je uvnitř P */
};

int main(void) {
        CProblem p;
        CPoint X;
        for(;;) {
                cout << "Zadej bod:";
                X.input();
                if (p.solve(X)) cout << "Lezi uvnitr\n"; else
                        cout << "Lezi vne\n";
                if ((X.x==0) && (X.y==0)) break;
        }
        return 0;
}

void CPoint::input() {
        cin >> x >> y;
}

CLine::CLine(CPoint &A, CPoint &B) {
        if (A.x<B.x) {
                this->A = &A; this->B = &B;
        } else {
                this->B = &A; this->A = &B;
        }
        a = A.y - B.y;
        b = B.x - A.x;
        c = -( a*A.x + b*A.y);
}

int CLine::upperThanLine(CPoint &pt) {
        double x = a*pt.x + b*pt.y +c;
        if (x>0) return 1;
        if (x<0) return -1;
        return 0;
}

int CLine::isBetween(CPoint &pt) {
        return ((A->x<=pt.x) && (B->x>=pt.x));
}

CPolygon::CPolygon() {
        cin >> n;
        pt = new CPoint[n];
        for (int i=0;i<n;i++) pt[i].input();
}

CPolygon::~CPolygon() {
        delete[] pt;
}

CPolygon *__poly;
int cmpInd(const void *a, const void *b) {
        if ((__poly->pt[*(int *)a]).x < (__poly->pt[*(int *)b]).x) return -1;
        if ((__poly->pt[*(int *)a]).x > (__poly->pt[*(int *)b]).x) return 1;
        return 0;
}
```

```
CSortedPoly::CSortedPoly(CPolygon &poly) {
        p = &poly;
        __poly = p;
        index = new int[p->n];
        for (int i=0;i < p->n;i++) index[i]=i;
        qsort(index, p->n, sizeof(int), cmpInd);
}

CSortedPoly::~CSortedPoly() {
        delete[] index;
}

CCutPoints::CCutPoints(CPolygon &poly) {
        p = &poly;
        CSortedPoly s(poly);
        double minDif=s[p->n -1].x - s[0].x; /* Nejmenší rozdíl x */
        int i;
        for(i=0;i < p->n -1;i++) {
                dif = s[i+1].x-s[i].x;
                if (dif>0)              /* dif == 0 není zajímavý */
                        if (dif<minDif) minDif = dif;
        }
        dif = minDif;
        or = s[0].y < s[1].y ? 1 : -1;
        cnt = (int)((s[p->n -1].x - s[0].x)/minDif)+1; /* Počet dílků */
        lower = new int[cnt];
        upper = new int[cnt];
        int ptr;
        if (s[0].x == s[1].x) {         /* Vytvoření dílků */
                if (s[0].y<s[1].y) {
                        lower[0]=s.index[0];
                        upper[0]=s.index[1];
                } else {
                        lower[0]=s.index[1];
                        upper[0]=s.index[0];
                }
                ptr = 2;
        } else {
                lower[0]=upper[0]=s.index[0];
                ptr = 1;
        }
        if (s[p->n -1].x == s[p->n -2].x) {
                if (s[p->n -1].y<s[p->n -2].y) {
                        maxlow=s.index[p->n -1];
                        maxup=s.index[p->n -2];
                } else {
                        maxlow=s.index[p->n -2];
                        maxup=s.index[p->n -1];
                }
        } else {
                maxlow=maxup=s.index[p->n -1];
        }
        for(i=1;i<cnt;i++) {
                upper[i]=upper[i-1];    /* Většinou zůstanou původní body */
                lower[i]=lower[i-1];
                  // padne nějaký bod do daného výseku?
                for(;;) {
```

```
                        if (s[ptr].x < (s[0].x+dif*i)) { /* Zpracujeme všechny takové */
                          /* následující bod patří do tohoto výseku */
                                if (CLine(p->pt[upper[i-1]],p->pt[maxup])
                                        .upperThanLine(s[ptr])>0) {
                                        /* Tento bod je nahoře */
                                        upper[i]=s.index[ptr];
                                } else
                                        lower[i]=s.index[ptr]; /* Bod je dole */
                                ptr++;
                                continue; /* Zkusíme další */
                        }
                        break;          /* Neležel tam tento, nebude tam žádný další */
                }
        }
}

CCutPoints::~CCutPoints() {
        delete[] lower;
        delete[] upper;
}

CProblem::CProblem() {
        p = new CPolygon();
        c = new CCutPoints(*p);
}

CProblem::~CProblem() {
        delete c;
        delete p;
}

int CProblem::solve(CPoint &X) {
        if ( p->pt[c->lower[0]].x==X.x  /* Neleží náhodou na levé či pravé hranici? */
                && p->pt[c->lower[0]].y<=X.y
                && p->pt[c->upper[0]].y>=X.y) return 1;
        if ( p->pt[c->maxlow].x==X.x
                && p->pt[c->maxlow].y<=X.y
                && p->pt[c->maxup].y>=X.y) return 1;
        int cut = (int)((X.x- (p->pt[c->lower[0]]).x)/c->dif);
        if (cut<0 || cut>=c->cnt) return 0; /* Mimo hranice x */
                                        /* Leží pod/nad přímkami? */
        if ((CLine(p->pt[c->upper[cut]],p->pt[(c
                  ->upper[cut]+p->n+c->or)%(p->n)]).upperThanLine(X)<=0) &&
                        (CLine(p->pt[c->lower[cut]],p->pt[(c
                  ->lower[cut]+p->n-c->or)%(p->n)]).upperThanLine(X)>=0) &&
                        (cut+1 == c->cnt ||
                        ((CLine(p->pt[c->upper[cut+1]],p->pt[(c
                  ->upper[cut+1]+p->n+c->or)%(p->n)]).upperThanLine(X)<=0) &&
                         (CLine(p->pt[c->lower[cut+1]],p->pt[(c
                  ->lower[cut+1]+p->n-c->or)%(p->n)]).upperThanLine(X)>=0))))
                        return 1;
        return 0;
}
```

10-3-5 Terucempa — Jirka Hanika

Jen málokdo si nepovšiml, že trpaslíci jsou dlouhověcí, a procházel po řadě všechny možné permutace; takové řešení závisí na pořadí dne lineárně, na počtu trpaslíků tedy zhruba exponenciálně a ani pro osm či dvanáct trpaslíků (zadání si v přesném počtu poněkud odporovalo, za což se poněkud omlouváme) není rychlost výpočtu rozumná.

Pokusme se seřadit trpaslíky přímo z čísla dne. Očividně, na prvním místě se každý trpaslík vystřídá 11! krát (11! znamená $11 \cdot 10 \cdot 9 \cdot \ldots \cdot 2 \cdot 1$), na druhém každý kromě toho, co už stojí na prvním místě, 10! krát a tak dále. Kolikátý volný trpaslík má stát na n-tém místě od konce, tedy určíme tak, že pořadí dne vydělíme $n!$ a necháme si jen zbytek po dělení. Tím jsme odhlédli od změn na předcházejících pozicích a stačí nám tento zbytek vydělit $(n-1)!$, čímž se dozvíme, kolikátý trpaslík z těch, co dosud nebyli umístěni, bude umístěn na tomto místě.

Je nutno si udržovat seznam zbývajících trpaslíků a každého umístěného trpaslíka z něj vyškrtnout (posunout část seznamu), takže výsledná časová složitost je $O(N^2)$, kde N je počet trpaslíků. Na pořadí dne algoritmus nezávisí. Prostorová složitost je $O(N)$.

Zdeněk Dvořák a Jan Kára seznam volných trpaslíků reprezentují binárním vyhledávacím stromem, čímž docilují průměrné rychlosti $O(N \cdot \log N)$, jež je zároveň nejlepší možná. Pro počty trpaslíků, pro které lze na dnešních nedokonalých 64-bitových počítačích zpracovávat jejich faktoriál jako přirozené číslo, však obyčejné řešení v čase $O(N^2)$ neznamená žádné zdržení a uznávali jsme jej jako optimální.

Náš vzorový program napsaný v Céčku tvoří rekurzivní procedura, která se nejprve dvanáckrát od posledního trpaslíka v řadě k prvnímu zanoří a spočte si při tom příslušné faktoriály. Pak se postupně vynořuje zpátky k poslední pozici a s pomocí těchto faktoriálů uvedeným algoritmem určuje jednotlivé trpaslíky.

```
#include <stdio.h>
#define N 12
char dwarves[N+1];                      /* Seznam dosud nezařazených trpasliků */
unsigned long int day = 1;              /* Zadá trpaslík Arnoštek */
void use (int i)                        /* Vytiskni a vyřaď i-tého trpaslíka */
{
        putchar (dwarves[i]);
        while (dwarves[i])
                dwarves[i]=dwarves[i+1], i++;
}
void previous (int fact, int n)         /* Urči n-tého trpaslíka od konce */
{
```

```
        int next_fact;
        if (n <= N) {
                next_fact = previous (fact * n,  n+1);
                use ( (day % next_fact) / fact);
        }
        return fact;                    /* Nenápadně počítáme faktoriál */
}
int main (void)
{
        int i;
        for (i=0; i<N; i++) dwarves[i] = 'A'+i;
        printf ("Arnostku, kolikateho dnes mame?␣");
        scanf ("%lud",  &day); day--; putchar ('\n');
        previous (1,  1); putchar ('\n');
        return 0;
}
```

10-4-1 Wagón klád — Aleš Přívětivý

Máme zadanou posloupnost délek klád $a_1, a_2, \ldots, a_n$ a máme vypsat všechny součty $a_i + a_j$ takové, že $i \neq j$ a bez opakování (n je délka posloupnosti). Bereme-li všechny kombinace součtů, zjistíme, že jich je maximálně $\frac{1}{2}n \cdot (n-1)$, přičemž některé se mohou opakovat. Nejjednodušší řešení by bylo vytvořit všechny tyto součty, pak je setřídit a vypsat je bez opakujících se součtů. Tento algoritmus by měl časovou složitost $O(n^2 \cdot \log n)$ a paměťovou $O(n^2)$. Jelikož ale maximální povolená paměťová složitost je $O(n)$, musíme vymyslet něco jiného.

Abychom se vyhnuli třídění, budeme součty generovat ve vzestupném pořadí. Klády si setřídíme vzestupně, tedy platí $a_1 \leq a_2 \leq \ldots \leq a_n$. To dokážeme při nejhorším v čase $O(n^2)$. Nyní si všimněme, že platí:

$$
\begin{aligned}
&a_1 + a_2 \leq a_1 + a_3 \leq a_1 + a_4 \leq \ldots \leq a_1 + a_n \\
&a_2 + a_3 \leq a_2 + a_4 \leq a_2 + a_5 \leq \ldots \leq a_2 + a_n \\
&\ldots \\
&a_i + a_{i+1} \leq a_i + a_{i+2} \leq a_i + a_{i+3} \leq \ldots \leq a_i + a_n
\end{aligned}
$$

Takových posloupností máme n (pro každou kládu jednu) a pro každou kládu tvoříme pouze dvojice s kládami s vyšším indexem. A těchto n setříděných posloupností nyní chceme sloučit do jedné velké, též setříděné posloupnosti – to provedeme následujícím postupem: U každé takové i-té posloupnosti

$$a_i + a_{i+1} \leq a_i + a_{i+2} \leq a_i + a_{i+3} \leq \ldots \leq a_i + a_n$$

si budeme pamatovat index j_i, tj. první index, pro který součet $a_i + a_{j_i}$ nebyl ještě zpracován. Na začátku je pro všechny i-té posloupnosti hodnota indexu

$j_i = i + 1$, tj. kláda tvoří dvojici s následující kládou. V následujících krocích vybereme vždy minimální součet $a_i + a_{j_i}$ (pro všechna i, kde $j_i \leq n$) a pokud není roven předchozímu zpracovanému součtu, tak jej vypíšeme. Zároveň se posuneme na další člen i-té posloupnosti inkrementací j_i. Takto procházíme součty ve vzestupném pořadí, dokud máme nějaké součty k dispozici (dokud existuje i takové, že $j_i \leq n$).

Jestliže v každém kroku vybíráme minimum z n součtů s lineární časovou složitostí, dostaneme algoritmus s časovou složitostí $O(n^3)$ a paměťovou $O(n)$. Toto lze ale ještě zlepšit, zkonstruujeme-li si pro vyhledávání minima haldu. Pak jsme schopni určit minimum z n prvků v konstantním čase a náhradu za právě odebraný prvek zatřídíme do haldy v čase $O(\log n)$, takže celková časová složitost se zlepší na $O(n^2 \cdot \log n)$. Protože haldu z vás většina zná, nebudu jí ani operace na ní zde popisovat. Pro ty, kteří se s touto datovou strukturou nesetkali, doporučuji jednu z nejznámějších knih o algoritmech – Algorithms and Data Structures od Niklause Wirtha (slovenský předklad Algoritmy a štruktúry údajov).

K výpisu programu stojí snad jen za zmínku to, že pokud nějaká posloupnost skončí a nemůže dát další součet do haldy, vloží se tam fiktivní součet (větší než dosažitelný) a pokud je celá halda plná těchto součtů, tedy i ve vrcholu je fiktivní prvek, víme, že všechny posloupnosti poskytly již všechny své prvky a skončíme.

```
#include <stdio.h>
#include <stdlib.h>

#define MaxKlad 1024
#define MaxDelka 0x7fffffff

struct TPrvek {
      int dvojice, prvni, druha;
};

int Klady[MaxKlad], n;
struct TPrvek Halda[MaxKlad];

int compare (const void *a, const void *b)      /* Porovnávací fce pro qsort */
{
    return *(int *)a-*(int *)b;
}

void Zatrid (int co, int velikost)              /* Najde prvku ve vrcholu co */
{                                               /* správnou pozici v haldě */
    struct TPrvek x=Halda[co];                  /* o velikosti velikost */
    int i=2*co, j=co;

    while (i<velikost)
    {
        if ( (i+1<velikost)&& (Halda[i].dvojice>Halda[i+1].dvojice)) i++;
        if (x.dvojice<=Halda[i].dvojice) break;
        Halda[j]=Halda[i]; j=i; i=2*j;
```

```
    }
    Halda[j]=x;
}
int main ()
{
    int i, posledni, carka;
    scanf ("%d", &n);                          /* Načte počet klád do n */
    for (i=0; i<n; i++) scanf ("%d", &Klady[i]);   /* a délky klád */
    qsort (Klady, n, sizeof (int), compare);   /* a setřídí je dle délky */
    for (i=0; i<n-1; i++)                      /* Vytvoří haldu */
      {
        Halda[i].dvojice=Klady[i]+Klady[i+1];
        Halda[i].prvni=Klady[i];
        Halda[i].druha=i+1;
      }
    Halda[n-1].dvojice=MaxDelka;               /* Poslední kláda už dvojici netvoří */
    posledni=-1; carka=0;
    while (Halda[0].dvojice<MaxDelka)          /* Dokud nenarazíme na fiktivní kládu */
      {
        if (posledni<Halda[0].dvojice)         /* Výpis pouze pro nové hodnoty */
          if (!carka) {printf ("%d", Halda[0].dvojice); carka=1; }
              else printf (",%d", Halda[0].dvojice);
        posledni=Halda[0].dvojice;
        if (++Halda[0].druha==n)               /* Náhrada novou dvojicí klád */
          Halda[0].dvojice=MaxDelka;           /* Pokud není, fiktivní hodnotou */
        else
          Halda[0].dvojice=Halda[0].prvni+Klady[Halda[0].druha];
        Zatrid (0, n);                         /* Zatřídění vrcholu do haldy */
      }
    putchar ('\n');
    return 0;
}
```

10-4-2 DĚLITEL — Martin Mareš

Většina z vás nevyužila nápovědu v zadání a aplikovala obyčejný Euklidův algoritmus s odčítáním, dosáhnuvši tím časové složitosti typu $O((m+n)\cdot\log(m+n))$. Cílem tohoto spisku je ukázat, že dělitele lze počítat daleko rychleji – konkrétně v čase $O(\log^2(m+n))$. Tedy nejdříve stručně, jak na to:

[1] Dokud jsou m i n sudá čísla, dělím obě dvěma. V proměnné k si zapamatuji, kolikrát jsem tak učinil.

[2] Pokud $m = n$, končím a za výsledek prohlásím $n \cdot 2^k$.

[3] Pokud je m či n sudé, dělím jej dvěma a jdu na [2].

[4] Pokud $m < n$: $n \leftarrow n - m$, jinak $m \leftarrow m - n$.

[5] Jdu na [2].

Ponejprv dokažme, že pokud se tento algoritmus zastaví, je jeho výsledkem největší společný dělitel $[m, n]$ čísel m a n. Vyplývá to přímo z dodané nápovědy: V kroku [1] využíváme pravidla $[2 \cdot x, 2 \cdot y] = 2 \cdot [x, y]$ (co jsme vydělili, na konci opět vynásobíme), krok [2] je podepřen triviálním faktem $[x, x] = x$, krok [3] pravidlem „x liché a y sudé $\Rightarrow [y, x] = [x, y] = [x, y/2]$“ a konečně krok [4] je klasické Euklidovo pravidlo $[x, y] = [x - y, y] = [x, y - x]$.

Nyní dokážeme konečnost algoritmu, a to tím, že ověříme, že neučiníme více průchodů cyklem než je dvojnásobek počtu číslic ve dvojkovém zápisu čísel m a n dohromady:

- Pokud je v kroku [2] kterékoliv z čísel m, n sudé, v další iteraci cyklu se toto číslo o bit zkrátí (dělení dvěma v [3]).
- Pokud jsou v kroku [2] obě čísla lichá, jedním průchodem cyklu se jedno z nich stane lichým [4] a dalším se zkrátí [3].

Tudíž na každé zkrácení m či n o bit jsme potřebovali maximálně dva průchody cyklem, z čehož přímo dostáváme jak konečnost algoritmu, tak časovou složitost $O(\log(m+n) \cdot f)$, kde f říká, jak rychle jsme schopni počítat základní operace jako sčítání, odčítání, porovnávání a dělení dvěma.

Pro ty nedočkavé: hotový Turingův stroj bude vypadat takto:

stav	**0** (00)	**1** (01)	**2** (10)	**3** (11)	♯	Λ
S	$S/\sharp/R$	$D_x/\mathbf{1}/R$	$D_y/\mathbf{2}/R$	$C_=/\mathbf{3}/R$	—	$Z/\Lambda/L$
$C_=$	$C_=/\mathbf{0}/R$	$C_</\mathbf{1}/R$	$C_>/\mathbf{2}/R$	$C_=/\mathbf{3}/R$	—	$Z/\Lambda/L$
$C_<$	$C_</\mathbf{0}/R$	$C_</\mathbf{1}/R$	$C_>/\mathbf{2}/R$	$C_</\mathbf{3}/R$	—	$S_x/\Lambda/L$
$C_>$	$C_>/\mathbf{0}/R$	$C_</\mathbf{1}/R$	$C_>/\mathbf{2}/R$	$C_>/\mathbf{3}/R$	—	$S_y/\Lambda/L$
D_x	$D_x/\mathbf{0}/R$	$D_x/\mathbf{1}/R$	$D_x/\mathbf{2}/R$	$D_x/\mathbf{3}/R$	—	$D_{0*}/\Lambda/L$
D_{0*}	$D_{0*}/\mathbf{0}/L$	$D_{0*}/\mathbf{1}/L$	$D_{1*}/\mathbf{0}/L$	$D_{1*}/\mathbf{2}/L$	$S/\sharp/R$	$S/\Lambda/R$
D_{1*}	$D_{0*}/\mathbf{2}/L$	$D_{0*}/\mathbf{3}/L$	$D_{1*}/\mathbf{2}/L$	$D_{1*}/\mathbf{3}/L$	—	—
D_y	$D_y/\mathbf{0}/R$	$D_y/\mathbf{1}/R$	$D_y/\mathbf{2}/R$	$D_y/\mathbf{3}/R$	—	$D_{*0}/\Lambda/L$
D_{*0}	$D_{*0}/\mathbf{0}/L$	$D_{*1}/\mathbf{0}/L$	$D_{*0}/\mathbf{2}/L$	$D_{*1}/\mathbf{2}/L$	$S/\sharp/R$	$S/\Lambda/R$
D_{*1}	$D_{*0}/\mathbf{1}/L$	$D_{*1}/\mathbf{1}/L$	$D_{*0}/\mathbf{3}/L$	$D_{*1}/\mathbf{3}/L$	—	—
S_x	$S_x/\mathbf{0}/L$	$S_x/\mathbf{1}/L$	$S_x/\mathbf{2}/L$	$S_x/\mathbf{3}/L$	$U_x/\sharp/R$	$U_x/\Lambda/R$
U_x	$U_x/\mathbf{0}/R$	$U_x/\mathbf{1}/R$	$V_x/\mathbf{3}/R$	$U_x/\mathbf{2}/R$	—	$W/\Lambda/L$
V_x	$V_x/\mathbf{1}/R$	$U_x/\mathbf{0}/R$	$V_x/\mathbf{2}/R$	$V_x/\mathbf{3}/R$	—	—
S_y	$S_y/\mathbf{0}/L$	$S_y/\mathbf{1}/L$	$S_y/\mathbf{2}/L$	$S_y/\mathbf{3}/L$	$U_y/\sharp/R$	$U_y/\Lambda/R$
U_y	$U_y/\mathbf{0}/R$	$V_y/\mathbf{3}/R$	$U_y/\mathbf{2}/R$	$U_y/\mathbf{1}/R$	—	$W/\Lambda/L$
V_y	$V_y/\mathbf{2}/R$	$V_y/\mathbf{1}/R$	$U_y/\mathbf{0}/R$	$V_y/\mathbf{3}/R$	—	—
V	$V/\mathbf{0}/L$	$V/\mathbf{1}/L$	$V/\mathbf{2}/L$	$V/\mathbf{3}/L$	$S/\sharp/R$	$S/\Lambda/R$
Z	$Z/\mathbf{0}/L$	$Z/\mathbf{1}/L$	$Z/\mathbf{2}/L$	$Z/\mathbf{3}/L$	$Z/\mathbf{0}/L$	$Z/\Lambda/L$

Popis stavů:

S	obě sudá → ♯, jedno sudé → dělíme dvěma
$C_=$	porovnání obou čísel
D_x	dělíme první čislo dvěma
D_y	dělíme druhé čislo dvěma
S_x	odečítáme první od druhého
S_y	odečítáme druhé od prvního
V	návrat zpět
Z	konec a zbavení se ♯

A jak toto provést na Turingově stroji? Inu, inspirujeme se v mnohém úlohou 10-2-3:

- Zápis čísel zvolíme opět binární, prokládaný a pakovaný (to jest střídavě číslice z obou čísel počínaje nejnižším řádem a dvojici číslic zakódujeme na pásce jedním symbolem). Použijeme tedy abecedu $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \Lambda, \sharp\}$, kde **0** až **3** jsou číslice, Λ je prázdný symbol (na začátku i na konci vstupu) a ♯ je pomocný symbol.
- Krok [1] provedeme tak, že na počátku výpočtu budeme všechny „dvojnuly“ od začátku pásky přepisovat na ♯ a nakonec pojedeme hlavou zpět, přepisujíce ♯ zpět na **0** a zastavíme se o okraj pásky.
- Zjištění sudosti či lichosti čísla je triviální.
- Porovnání dvou čísel na rovnost, případně $x < y$ lze provést v lineárním čase prostým průchodem od nejnižšího řádu k nejvyššímu.
- Odečtení dvou čísel taktéž.
- Vydělení jednoho z čísel dvěma odpovídá jeho posunutí po pásce o jedno políčko, rovněž pak v lineárním čase. (Je pouze nutno si uvědomit, že „proložené“ číslice druhého čísla je žádoucí zachovat.)

Již z tohoto popisu plyne, že $f = O(\log(m+n))$, tudíž celková časová složitost stroje je $O(\log^2(m+n))$.

10-4-3 Sssssttttttrrrriiinng — Aleš Přivětivý

Řešení, jak v řetězci délky n prohodit dva podřetězce (i, j) a (r, s) za podmínky $1 \leq i \leq j < r \leq s \leq n$ v lineárním čase, je mnoho, proto zde uvedu jen to, které pokládám za nejelegantnější. Definujme rotaci řetězce kolem osy jako zobrazení, které z prvního písmene řetězce udělá poslední, z druhého předposlední ... a z posledního písmene první. Každého určitě hned napadne algoritmus na rotaci řetězce kolem osy tak, aby pracoval korektně, v lineárním čase vzhledem k délce řetězce a potřeboval navíc pouze jednu proměnnou, potřebnou k výměně znaku. Nyní se podívejme, co se stane, když v zadaném řetězci

rotujeme podřetězec (i, s) – tj. podřetězec obsahující oba podřetězce (i, j) a (r, s) a část mezi nimi. A ejhle, podřetězce se vyměnily a prostředek zůstal mezi nimi, tedy jsou na požadovaném místě, až na jedinou chybičku, a to, že oba podřetězce i prostředek mezi nimi jsou převráceny. To ale vyřešíme třemi rotacemi příslušných podřetězců ... a máme výsledek. Protože se algoritmus skládá ze čtyřnásobného použití rotace (o které víme, že má lineární časové a konstantní paměťové nároky), je celková časová složitost algoritmu lineární, tj. $O(s - i)$ a paměťová bez zpracovávaného řetězce konstantní, $O(1)$. Algoritmus je tak triviální, že správnost a konečnost je ihned viditelná.

```
program ssstrrrinnnng;

procedure rotate(var base:string; left,right:integer);
var tmp:char;
begin
   while (left<right) do
     begin
       tmp:=base[left];
       base[left]:=base[right];
       base[right]:=tmp;
       inc(left); dec(right);
     end;
end;

var buffer:string;
    i,j,r,s:integer;

begin
   write('Zadej retezec: ');
   readln(buffer);
   write('Zadej i,j,r,s: ');
   readln(i,j,r,s);
   rotate(buffer,i,j); rotate(buffer,r,s); rotate(buffer,j+1,r-1);
   rotate(buffer,i,s);
   writeln('Vysledny retezec: ',buffer);
end.
```

10-4-4 Duraloví dravci — Pavel Machek

Hledání cesty v grafu. Hmm, to zavání Dijkstrovým algoritmem... Ale zase tak jednoduché to nebude. Protože přesuny mezi letišti ve městě také něco stojí, budeme muset graf rozšířit. Konkrétně každé město rozdělíme na s vrcholů, každý odpovídá jednomu letišti. (Pro teď si představujme, že každá společnost má vlastní letiště.) Mezi letišti nataháme hrany, jejichž ceny budou odpovídat cenám za přeložení nákladu.

Tohle sice vypadá hezky, ale má to jednu nepříjemnou vlastnost: nefunguje to. V našem státě totiž bují byrokracie, a tak nám nikdo nezaručuje, že přeložení nákladu od společnosti 1 ke společnosti 2 nebude dražší než přeložení nákladu $1 \to 3$, $3 \to 2$. Zato si můžeme být jisti, že přeložení nákladu přes třetí společnost bude nezákonné.

Takže si každé letiště rozdělíme na příletovou a odletovou část. Nikdy nebude existovat hrana mezi odletovou a příletovou částí, zato však vždy bude existovat hrana příletová → odletová část stejné společnosti (překládat nemusíme) a bude mít cenu 0. To by mělo učinit byrokracii za dost.

K programu: Údaje budeme ukládat do trojrozměrných polí: V 1. rozměru bude město, ve 2. společnost (takže 1. a 2. rozměr dohromady určují letiště) a 3. rozměr bude část (0=příletová, 1=odletová), takže všechny 3 rozměry udávají přesně jednu část letiště.

Všechno ostatní už je standardní Dijkstrův algoritmus. Vzhledem k tomu, že je to jeden z nejběžnějších algoritmů na hledání nejkratších cest v grafech (i v KSP se již vyskytl mnohokráte, viz např. úloha 8-5-4), nebudeme jej zde podrobně odvozovat a uvedeme pouze základní ideu: U každého vrcholu v grafu si pamatujeme vzdálenost d_v plus informaci o tom, zda je to vzdálenost finální či přechodná. Na počátku nastavíme d počátečního vrcholu na finální nulu a ostatních na přechodné $+\infty$. V každém kroku vybereme vrchol w s nejmenší přechodnou vzdáleností, tu prohlásíme za finální a aktualizujeme přechodné vzdálenosti podle hran vedoucích z w (to jest pokud se někam umíme dostat přes w kratší cestou než jsme dokázali dříve, poznamenáme si tuto cestu namísto původní).

Bohužel, výpočetní složitost Dijkstrova algoritmu je výrazně ovlivněna datovou strukturou, jíž používáme na ukládání přechodných vzdáleností a hledání minim. Optimální výsledky lze dosáhnout s použitím Fibonacciho haldy (vyhledání i aktualizace v čase $O(\log k)$, viz úloha 8-5-4), leč její implementace je velice složitá, takže se spokojíme s obyčejným lineárním vyhledáváním v poli ($O(k)$), což nám dá celkovou časovou složitost $O(n^2s^2)$ (n je počet měst, s počet společností) a paměťovou $O(n^2s + ns^2)$.

```
#include <stdlib.h>
#include <stdio.h>

#define SPOL 100
#define MEST 100

int cp[MEST][MEST][SPOL];                /* Cena za cestu z 1 do 2 společností 3 */
int cm[MEST][SPOL][SPOL];                /* Cena za překlad ve městě 1 od
                                            společnosti 2 ke společnosti 3 */

int cena[MEST][SPOL][2];                 /* Jaká je cena za přesun do této části
                                            letiště? */
int konecne[MEST][SPOL][2];              /* Je výše uvedená cena konečná? */
int odkud_m[MEST][SPOL][2];              /* Jaký je předchozí uzel na cestě? */
int odkud_s[MEST][SPOL][2];
int odkud_b[MEST][SPOL][2];

int ns, nm;

#define INF 60000                        /* Nekonečno */

/* Najdi prvek s nejmenší dočasnou hodnotou (zde by měla být Fibonacciho halda). */
```

```
int
findmin ( int *m, int *s, int *b )
{
    int i, j, k;
    int mcena = INF;
    for (i=0; i<nm; i++)
        for (j=0; j<ns; j++)
            for (k=0; k<2; k++)
                if (!konecne[i][j][k] && (cena[i][j][k] < mcena)) {
                    mcena = cena[i][j][k];
                    *m=i; *s=j; *b=k;
                }
    return mcena;
}
/* Zkus použít nové spojení z [om,os,ob] do [m,s,b] za cenu c */
void
zlepsi ( int m, int s, int b, int c, int om, int os, int ob )
{
    if (cena[m][s][b] > c) {
        cena[m][s][b] = c;
        odkud_m[m][s][b] = om;
        odkud_s[m][s][b] = os;
        odkud_b[m][s][b] = ob;
    }
}
/* Získej od uživatele nějaká vstupní data. */
void
input ( void )
{
    int i, j, k;
    printf ( "Zadej pocet spolecnosti, pocet mest:\n");
    scanf ( "%d%d", &ns, &nm );
    for (i=0; i<nm; i++)
        for (j=0; j<nm; j++)
            for (k=0; k<ns; k++) {
                if (i==j) { cp[i][j][k] = 0; continue; }
                printf ( "Cena za cestu z %d do %d spolecnosti %d:\n", i, j, k );
                scanf ( "%d", &cp[i][j][k] );
        }
    for (i=0; i<nm; i++)
        for (j=0; j<ns; j++)
            for (k=0; k<ns; k++) {
                if (j==k) { cm[i][j][k]=0; continue; }
                printf ( "Cena za preklad ve meste %d od spolecnosti %d k %d:\n", i, j, k );
                scanf ( "%d", &cm[i][j][k] );
            }
}
/* Nějaký prvek získal konečnou hodnotu, zkus použít cesty z něj. */
void
updateokoli ( int m, int s, int b )
```

```
{
    int i;
    int c = cena[m][s][b];
    if (!b)
        for (i=0; i<ns; i++)
            zlepsi (m, i, 1, c+cm[m][s][i], m, s, b );
    if (b)
        for (i=0; i<nm; i++)
            zlepsi (i, s, 0, c+cp[m][i][s], m, s, b );
}
/* Nainicializuj vstupní data */
void
init (void)
{
    int i, j, k;
    for (i=0; i<nm; i++)
        for (j=0; j<ns; j++) {
            cena[i][j][0] = cena[i][j][1] = INF;
            konecne[i][j][0] = konecne[i][j][1] = 0;
        }
    for (i=0; i<ns; i++) {
        cena[0][i][0]=0;
        konecne[0][i][0]=1;
        updateokoli (0, i, 0);
    }
}
/* Vypiš cestu */
int
vypis ( int m, int s, int b )
{
    printf ( "Mesto %d, spolecnost %d, cast %d, (az sem to stalo %d penez)\n", m, s, b,
        cena[m][s][b] );
    if (!m) return;
    vypis ( odkud_m[m][s][b], odkud_s[m][s][b], odkud_b[m][s][b] );
}
int
main ( void )
{
    input ();
    init ();
    while (1) {
        int m, s, b;
        if (findmin (&m, &s, &b) == INF) {
            printf ( "Do ciloveho mesta se nelze dostat.\n");
            return 0;
        }
        konecne[m][s][b] = 1;
        if (m == nm-1) {
            printf ( "Do ciloveho mesta se lze dostat za %d penez.\nCesta (pozpatku) je:\n",
                cena[m][s][b] );
```

```
            vypis (m, s, b);
            return 0;
          }
          updateokoli (m, s, b);
        }
        return 0;
}
```

10-5-1 Společná podmatice — Daniel Kráľ

Zkusme nejprve vyřešit následující modifikaci úlohy tohoto příkladu: Jsou zadány dvě matice stejných rozměrů ($N \times M$) a máme nalézt největší stejnou podmatici těchto matic takovou, že je v obou dvou maticích na stejném místě. Tuto úlohu lze snadno převést na hledání největší jedničkové podmatice dané matice tak, že si vytvoříme matici, která bude obsahovat nuly tam, kde se naše matice liší, a jedničky tam, kde se shodují (sledujte obrázek). Nyní vypočteme pro každý nenulový prvek matice, kolik je nad ním a pod ním jedniček a tato čísla uložíme do dvou pomocných matic. Tento výpočet lze provést v čase lineárně úměrném velikosti (obsahu) matic a to průchodem matice vhodným směrem (postupná inkrementace počtu jedniček v právě procházeném souvislém úseku v daném sloupci). Takto vzniklé matice „obalíme“ pásem nul, což nám zjednoduší další výpočet.

Největší jedničková podmatice je největší, což zejména znamená, že existuje nějaká nula těsně za jejím levým okrajem, neboť by jinak bylo tuto podmatici možné zvětšit, to nám zaručuje mimo jiné i její obalení. Náš postup bude takový, že budeme procházet matici po řádcích a od každého přechodu z nulového na nenulový prvek se budeme snažit nalézt největší podmatici, jejíž levá hranice je těsně u tohoto nulového prvku. Mezi takovýmito podmaticemi bude i hledaná (největší) podmatice, jak jsme si již řekli. To je však jednoduché naprogramovat, neboť velikost největší takovéto jednotkové podmatice šířky k, pokud za uvažovanou nulou následuje alespon k jedniček, je rovna $k\times$ součtu minima počtu jedniček pod a nad těmito jedničkami. Toto minimum lze snadno vypočítat v konstantím čase, pokud si ho budeme pamatovat pro $k-1$. Celková časová složitost tohoto algoritmu je tedy $O(NM)$ a paměťová $O(NM)$.

Nyní již snadno vyřešíme naši úlohu. Stačí postupně uvážit všechny vzájemné posuny obou matic a pro každou z nich vyřešit naši modifikovanou úlohu pro výřezy těchto matic tvořené jejich společnou plochou. Jsou-li rozměry matic $N_1 \times M_1$ a $N_2 \times M_2$, potom vzájemných posunů, tj. různých poloh jejich levých horních rohů vůči sobě, je $O((N_1 + N_2) * (M_1 + M_2))$. Pro každý takovýto posun se provede algoritmus popsaný v úvodu řešení a celková časová i paměťová složitost algoritmu bude

$$O(\max(N_1, M_1)^2 \cdot \max(N_2, M_2)^2).$$

Na závěr ještě několik poznámek k programu. Zadané matice jsou uloženy v polích `pole1` a `pole2`, jejich rozměry pak v proměnných `pole1?` a `pole2?`. Údaje o výřezech matic jsou ukládány do pole `rozdil`. Velikost výřezu je uložena v proměnných `rozdil?`, posun levého horního rohu výřezu vůči levému hornímu rohu první a druhé matice jsou uloženy v proměnných `posun1?` a `posun2?`, posun levých horních rohů obou matic pak v proměnných `posun?`. Za povšimnutí stojí sloučení obou pomocných polí z popisu algoritmu pro hledání společné podmatice v jedno tak, že nulové hodnoty jsou neshodné prvky matic, záporné udávají posun dolů (záporně vzatý) na nejbližší nenulovou hodnotu nad nulovou hodnotou v tomto sloupci, kladné udávají počet nenulových hodnot v souvislém bloku ve sloupci matice – viz obrázek.

$$\begin{pmatrix}1&2&2\\2&1&2\\1&2&2\\2&1&2\end{pmatrix}+\begin{pmatrix}2&1&1\\2&1&2\\1&2&1\\1&1&1\end{pmatrix}\rightarrow\begin{pmatrix}0&0&0\\1&1&1\\1&1&0\\0&1&0\end{pmatrix}\rightarrow$$

$$\rightarrow\begin{pmatrix}0&0&0\\1&1&1\\2&2&0\\0&3&0\end{pmatrix}\rightarrow\begin{pmatrix}0&0&0\\-1&-2&1\\2&-1&0\\0&3&0\end{pmatrix}$$

```
program podmatice;
const MAX=100;
var pole1x,pole1y,pole2x,pole2y:word;           { rozměry polí }
    pole1:array[1..MAX,1..MAX] of integer;      { vstupní pole }
    pole2:array[1..MAX,1..MAX] of integer;
    rozdil:array[0..MAX+1,0..MAX+1] of integer; { pomocné pole }
    rozdilx,rozdily:word;                       { vel.pomoc.pole }
    max_obsah:word;                             { max. obsah }
    max_x1,max_y1:word;                         { poloha podmatice }
    max_x2,max_y2:word;
    max_delkax,max_delkay:word;                 { rozměry podmatice }
    a,b,c,d,e:integer;                          { pomocné promenne }
    posunx,posuny,                              { vzájemný posun matic }
    posun1x,posun1y,                            { posun první matice }
    posun2x,posun2y:integer;                    { posun druhé matice }
    vzhuru,dolu:integer;
begin
 readln(pole1x,pole1y);
 for a:=1 to pole1x do
  for b:=1 to pole1y do
   read(pole1[a,b]);
 readln(pole2x,pole2y);
 for a:=1 to pole2x do
  for b:=1 to pole2y do
   read(pole2[a,b]);
 max_obsah:=0;
 for posunx:=-pole2x+1 to pole1x do
  for posuny:=-pole2y+1 to pole1y do
```

```
begin
 rozdilx:=pole2x+posunx;
 if rozdilx>pole1x then rozdilx:=pole1x+1 else rozdilx:=rozdilx+1;
 rozdily:=pole2y+posuny;
 if rozdily>pole1y then rozdily:=pole1y+1 else rozdily:=rozdily+1;
 if posunx<=0 then posun1x:=0 else posun1x:=posunx;
 if posuny<=0 then posun1y:=0 else posun1y:=posuny;
 if posunx>=0 then posun2x:=0 else posun2x:=-posunx;
 if posuny>=0 then posun2y:=0 else posun2y:=-posuny;
 for a:=0 to rozdilx do                      { příprava pomocného }
  for b:=0 to rozdily do                     { pole pro daný posun }
   if (a=0) or (b=0) or (a=rozdilx) or (b=rozdily) or
      (pole1[a+posun1x,b+posun1y]<>pole2[a+posun2x,b+posun2y]) then
    rozdil[a,b]:=0
   else
    rozdil[a,b]:=rozdil[a,b-1]+1;
 for a:=0 to rozdilx do
  for b:=rozdily downto 0 do
   if rozdil[a,b]=0 then
    c:=0
   else
    if c=0 then
     c:=-1
    else
     begin
      rozdil[a,b]:=c;
      dec(c)
     end;
 for b:=0 to rozdily do
  for a:=0 to rozdilx do
   if rozdil[a,b]=0 then
    begin
     c:=0;
     vzhuru:=0;
     dolu:=0;
    end
   else
    begin
     if rozdil[a,b]>0 then
      begin
       d:=rozdil[a,b]-1;   {nad}
       e:=0;               {pod}
      end
     else
      begin
       e:=-rozdil[a,b];    {pod}
       d:=rozdil[a,b]-e-1; {nad}
      end;
     if c=0 then
      begin
       c:=1;
       vzhuru:=d;
       dolu:=e;
      end
     else
      begin
       inc(c);
       if vzhuru>d then vzhuru:=d;
```

```
          if dolu>e then dolu:=e;
         end;
        if c*(vzhuru+dolu+1)>max_obsah then
         begin
          max_obsah:=c*(vzhuru+dolu+1);
          max_x1:=a+posun1x-c+1;
          max_y1:=b+posun1y-vzhuru;
          max_x2:=a+posun2x-c+1;
          max_y2:=b+posun2y-vzhuru;
          max_delkax:=c;
          max_delkay:=vzhuru+dolu+1;
         end
       end
   end;
 writeln(max_obsah);
 if max_obsah>0 then
  begin
   writeln(max_x1,' ',max_y1);
   writeln(max_x2,' ',max_y2);
   writeln(max_delkax,' ',max_delkay);
  end
end.
```

10-5-2 Kob ýlam ám alybok? — Jirka Hanika

Zaveďme rovnou potřebné termíny – řetězec, který lze číst z obou stran stejně, to jest který je zrcadlově souměrný podle středu, nazveme *palindromem*. Palindrom je *maximální*, pokud už ho nelze prodloužit o jeden znak na každé straně. Maximální palindromy mají různé středy a ten nejdelší z nich hledáme.

Otestovat všechny řetězce zadaného textu, zda nejsou palindromy, je možno v čase $O(n^3)$ (cyklus přes začátky, cyklus přes konce a cyklus testovací). Většina řešitelů ovšem snadno nahlédla redundanci danou zejména tím, že se takto – beznadějně – testují i řetězce, jejichž prostředky byly nebo budou samy o sobě zavrženy.

Lepší řešení tedy je procházet všechny potenciální středy palindromů, tedy i pozice mezi dvěma sousedními písmeny, a snažit se identifikovat co nejdelší palindrom se středem v tomto bodě. Toto řešení bude v případě velkého množství překrývajících se palindromů (jako například ve větě „Jůůůůůůůůůůůů, šnek!“) pracovat v čase $O(n^2)$; navíc pokud se nebudou palindromy překrývat o víc než pevně stanovený počet písmen, bude možno mluvit o lineárním čase.

I toto řešení však má v sobě skrytou redundanci. Pokud zkoumáme určitý střed a víme, že je tento střed někde uvnitř dříve nalezeného velmi dlouhého palindromu, máme vlastně chování okolo tohoto středu už prozkoumáno. Stačí se podívat, jak dopadl jeho zrcadlový kolega na druhé straně dlouhého palindromu. (Procházíme-li jednotlivými středy postupně, byl tento kolega zkoumán ještě před oním dlouhým palindromem.) Může se samozřejmě stát, že kolegové maximálního palindromu sahají až za hranice dlouhého palindromu: `ageloKolegajekleVelkejageloKole???` (zkoumáme střed `K` na pravé straně,

v bodu `V` je střed velkého palindromu, který sahá ještě tři znaky za nás). V každém případě ale víme, že velikost našeho palindromu je větší nebo rovna velikosti jeho zrcadlového kolegy, takže stačí použít tu a pokusit se ještě o prodloužení.

To znamená, že pracujeme nejen s momentálním středem, ale také s tabulkou všech dosud ověřených maximálních palindromů a s nejpravějším bodem, který máme podchycený některým dosavadním palindromem. Nikdy neporovnáváme dva prvky před tímto nejpravějším bodem, protože vše, co potřebujeme, už máme z výše uvedeného důvodu v tabulce: pokud úspěšně sáhneme na prvek za tímto hraničním bodem, tato hranice se tím posune; pokud při prodlužování palindromu neúspěšně sáhneme na některý prvek, ukončí se tím zpracovávaný palindrom a posune se střed. V každém kroku se tedy součet středu a tohoto hraničního bodu zvýší o jedna a tento součet lze shora omezit $O(n)$, proto provedeme nanejvýš $O(n)$ kroků a tento algoritmus je lineární v každém případě. Prostorové nároky jsou rovněž lineární.

Zbývá dořešit drobné uživatelské požadavky na ignorování mezer, velikosti písmen, případně ošetření písmene ‘ch’. Ty vzorový program implementuje ve funkci `get_text_and_make_tokens()`; musíme si však do polí `original` a `offset` ukládat původní text a přesměrování našich vnitřních znaků na znaky původní, abychom mohli nalezený palindrom věrně vytisknout. Kromě toho mezi každými dvěma vstupními znaky generujeme jeden konstantní vnitřní znak. To je proto, abychom mohli zároveň zpracovávat palindromy liché (se středem ve skutečném znaku) i sudé (se středem v některém ‘výplňovém’ znaku), což lze docílit i bez tohoto plýtvání místem, leč za cenu dalšího znepřehlednění práce s indexy nebo zdvojení části programu.

Vzorový program pro jednoduchost předpokládá, že znaky malé i velké abecedy jdou po sobě od ‘A’ do ‘Z’ a že je známa horní mez pro délku textu.

```
#include <stdio.h>

#define min (a, b)  ( (a<b) ?a:b)
char input[1024];                       /* vnitřní representace bez mezer, velkých
                                           písmen... */
int len[1024];                          /* délky nejdelších palindromů s pevnými
                                           středy */

char original[1024];                    /* uživatelská representace – indexuje se od
                                           jedničky */
int offset[1024];                       /* tabulka převádějící indexy vnitřní na
                                           indexy vnější representace */

int orig_counter = 0;
int token_counter = 0;

int longest = 1;                        /* index středu (zatím) nejdelšího
                                           palindromu */

void summary (void)
{
```

```
        int i;
        printf ("Nejdelsi nalezeny palindrom jest:␣");
        for (i = offset [ longest - len[longest] + 2 ]; i < offset [ longest + len[longest]
            - 2 ]; i++)
                putchar (original[i]);
        printf ("\n");
        exit (0);
}
void get_text_and_make_tokens (void)
{
        char c;
        input[0] = '␣';                    /* zarážka – nebude se rovnat ničemu
                                              dalšímu */
        original[0] = '␣';                 /* hlavně ne 'c', hrozilo by falešné 'ch' */
        do {
                c = getchar ();            /* další znak ze vstupu */
                original[++orig_counter] = c;   /* ukládej původní tvar */
                if (c >= 'A'&& c <= 'Z')
                        c += 'a'- 'A';     /* velká písmena na malá */
                if (c == 'h'&& original[orig_counter-1] == 'c') {
                        input[++token_counter] = 'C';   /* 'ch' je token */
                        continue;          /* a znova */
                }
                if (c == '␣') continue;    /* mezera? znova */
                input[++token_counter] = c;
                input[++token_counter] = '-';   /* uloží tvar a ... */
                offset[token_counter] = orig_counter;   /* kde hledat originál */
        } while (c != EOF);                /* opakuj, dokud je co číst */
        input[++token_counter] = '-';      /* ← co by bez tohoto řádku drhlo? */
}
void main ()
{
        int rightmostcentre = 1;           /* kam nejdál doprava se někdo dostal */
        int rightmostedge = 1;
        int i = 0;                         /* vzdálenost od středu */
        int centre = 1;                    /* právě zpracovávaný střed */

        get_text_and_make_tokens ();
        while (1) {
                while (input[centre - i] == input[centre + i]) ++i;
                len[centre] = i;           /* palindrom délky 2i + 1 */
                if (rightmostedge < centre + i) {
                        rightmostedge = centre + i - 1;
                        rightmostcentre = centre;
                }
                if (i > len[longest]) longest = centre;
                if (input[rightmostedge] == 0) summary ();   /* dosáhli jsme na
                    konec */
                centre++;
                if (rightmostedge < centre) i = 1;
```

```
                    else i = min (len[2 * rightmostcentre - centre], rightmostedge -
                        centre);
                    /* právě jsme využili zrcadlení, ale nanejvýš na hranici */
            }
}
```

10-5-3 Cycloose wyetshny — Martin Mareš

Cyklus ze zadání rozhodně nedoběhne zítra, nedoběhne za týden, za rok, ba ani za miliardu let ... nedoběhne totiž nikdy, čímž ostatně mnohé z vás doběhl. Pointa pro nedočkavé: typ *real* má omezenou přesnost, takže během výpočtu dojdeme k r velkému natolik, že $r + 1$ se již zaokrouhlí zpět na r.

A proč tomu tak je? Inu, bude nejlepší, když se podíváme na typ *real* (resp. *float*) zblízka. V obou případech se jedná o racionální (nikoliv tedy o obecně reálná, jak se Pascalský název typu snaží namluvit) čísla ukládaná ve formátu s plovoucí řádovou tečkou (*floating-point*), což znamená, že jsou vyjádřena ve tvaru

$$x = \pm m \cdot b^e,$$

kde:

- x je kýžené číslo, jež representujeme,
- $\pm$ je jeho znaménko,
- b je základ,
- m je *mantisa* v intervalu $\langle 1/b; 1)$, uložená jako číslo v soustavě o základu b mající N číslic s desetinnou (přesněji řečeno řádovou, protože nejsme v desítkové soustavě) tečkou (či čárkou, jak chcete) na svém levém okraji (tedy pro $b = 2$ například zápis `101` znamená $2^{-1} + 2^{-3} = 0.625$),
- e je exponent, uložený jako M-bitové celé číslo se znaménkem (vlastně říká, o kolik se má řádová tečka mantisy posunout doleva [kladné e] či doprava [záporné], abychom dostali x, proto se tomuto zápisu říká zápis s plovoucí tečkou).

Pro běžné počítače je $b = 2$, $7 \leq M \leq 16$ a $23 \leq N \leq 64$, dále tedy budeme předpokládat výhradně binární representaci.

Příklad: Číslo 5 by se ve floating pointu zapsalo jako $5 = +0.625 \cdot 2^3$. Povšimněte si, že nulu analogicky zapsat nelze – volí se pro ni obvykle nějaký nestandardní zápis (reservuje se jedna hodnota exponentu pro nulu, nekonečna a jiné „podezřelé“ hodnoty nebo se v tomto případě připustí $m = 0$). Všechna ostatní čísla (pokud nejsou mimo representovaný rozsah) je možno zapsat právě jedním způsobem (vezmete jejich binární zápis, najdete první jedničku zleva, zvolíte e tak, aby se dostala právě za řádovou tečku [vše před ní již budou nuly] a výsledek prohlásíte za mantisu, ovšem musíte ji ponejprv zaokrouhlit, aby se vešla do N bitů).

A nyní již k slíbenému „protipříkladu“. Představte si číslo tvaru $r = 2^{N+1} = 0.5 \cdot 2^{N+2}$. Pro takovéto r nám vyjde $r' = r+1 = (0.5+2^{-N-2}) \cdot 2^{N+2}$, takže mantisa m' bude rovna $0.5+2^{-N-2}$, což je ovšem po zaokrouhlení na M bitů pro její uložení dostupných určitě rovno 0.5, tedy původní mantise, takže $r+1 = r' = r$. (Kdybychom volili $r = 2^N$, museli bychom uvažovat o tom, zda se m' zaokrouhlí nahoru či dolů, při $r = 2^{N+1}$ musí nutně dolů.) A teď si jen stačí uvědomit, že pro největší běžné M vyjde $r = 2^{65} < 10^{20}$, což je hluboko pod horní mezí našeho cyklu. [Aby se tento rozbor stal opravdu korektním důkazem věčnosti cyklu, bylo by ještě nutno dokázat, že naše „zastavovací“ číslo nemůže program přeskočit – toho byste ale již po tomto úvodu měli být schopni vy sami.]

10-5-4 S radostí přijmi svůj (o)sud — Michal Píše

Algoritmus provádí prohledávání stavového prostoru do šířky. Jednotlivé stavy odpovídají uspořádaným n-ticím $\langle a_1, \ldots, a_n \rangle$ popisujícím naplnění jednotlivých lahví, přechody mezi stavy pak přelitím. Tím, že vše procházíme do šířky, máme zaručeno, že nalezneme řešení s nejmenším počtem přelití.

Použijeme klasickou implementaci s frontou na dosud nezpracované stavy. Na počátku fronta obsahuje stav $\langle 0, \ldots, 0 \rangle$ (všechny lahve prázdné) a v každém kroku vezmeme jeden dosud nezpracovaný stav a vyzkoušíme všechna možná přelití, přičemž vzniklé stavy zařazujeme na konec fronty (vynecháváme samozřejmě ty, které jsme již jednou viděli, abychom předešli zacyklení).

Program se řídí přesně tímto algoritmem, jediné, co stojí za zmínku, je, že naše stavové n-tice kódujeme celými čísly, abychom jimi mohli snadno indexovat pole. Toto kódování a dekódování stavů je zabezpečeno procedurami `Koduj` a `Dekoduj`.

Pokud si označíme s velikost stavového prostoru (to jest součin kapacit lahví zvětšených o 1) a n počet lahví, časová složitost algoritmu bude $O(s \cdot n)$ (každý stav projdeme max. jednou a zkoumáme max. $O(n)$ jeho sousedů), paměťová pak $O(s)$ (potřebujeme pamatovat frontu plus pro každý stav, zda jsme v něm již byli či nikoliv).

```
Program S_radosti_prijmi_svuj_osud;

Const Lahvi=3;

Type THladiny= Array [1..Lahvi] of Byte;

Const Objem: THladiny=(2,5,11);
      KonecnyObjem=4;
      DelkaFronty=10000;

Var Hladina,ZalozniHladina: THladiny;
    Neni: Array [0..10000] of Boolean;
    Predchozi: Array[0..10000] of Word;
```

```
    Fronta: Array[0..DelkaFronty-1] of Byte;
    Zacatek,Konec: Word;
    Hotovo: Boolean;
    i,j: Word;

Procedure Prelej(Odkud,Kam: Byte);
Begin
If Odkud=0 then {Dolevam ze sudu}
           Hladina[Kam]:=Objem[Kam] {Lahev se uplne naplni}
           else
           If Kam=0 then {Vylevam do sudu}
                    Hladina[Odkud]:=0 {Lahev se uplne vyprazdni}
                    else
                    If Hladina[Odkud]+Hladina[Kam]<=Objem[Kam] then Begin
                            {Vsechno se preleje do lahve}
                            Hladina[Kam]:=Hladina[Kam]+Hladina[Odkud];
                            Hladina[Odkud]:=0;
                            End
                    else Begin {V prvni lahvi jeste neco zbyde}
                            Hladina[Odkud]:=Hladina[Odkud]+Hladina[Kam]-Objem[Kam];
                            Hladina[Kam]:=Objem[Kam];
                            End;
End;

Procedure Koduj(Hladina: THladiny; Var Kod: Word);
Var i: Byte;
Begin
Kod:=0;
For i:=1 to Lahvi do Begin {Ted objemy jednoznacne zakoduji}
                     Kod:=Kod*(Objem[i]+1);
                     Kod:=Kod+Hladina[i];
                     if Hladina[i]=KonecnyObjem then Hotovo:=True; {Nasel jsem reseni}
                     End;
End;

Procedure ZaradDoFronty;
Var Kod,Kod2: Word;
Begin
Koduj(Hladina,Kod);
If Neni[Kod] then Begin {Pokud jsem tu uz nebyl}
                  Neni[Kod]:=False;
                  Fronta[Konec]:=Kod;
                  Konec:=(Konec+1) mod DelkaFronty;
                  Koduj(ZalozniHladina,Kod2);
                  Predchozi[Kod]:=Kod2;
                  End;
End;

Procedure Dekoduj(Kod: Word; Var Hladina:THladiny);
Var i: Byte;
Begin
For i:=Lahvi downto 1 do Begin
                         Hladina[i]:=Kod mod (Objem[i]+1);
                         Kod:=Kod div (Objem[i]+1);
                         End;
End;

Procedure VyndejZFronty;
```

```
Var Kod: Word;
Begin
Kod:=Fronta[Zacatek];
Zacatek:=(Zacatek+1) mod DelkaFronty;
Dekoduj(Kod,Hladina);
End;

Begin
Hotovo:=False;
For i:=0 to 10000 do Neni[i]:=True;
Predchozi[0]:=0;
Zacatek:=0;
Konec:=0;
For i:=1 to Lahvi do Hladina[Lahvi]:=0;
ZaradDoFronty;
Repeat VyndejZFronty;
       ZalozniHladina:=Hladina;
       i:=0;
       Repeat j:=0;
              Repeat If i<>j then Begin {Zkousim prelit z kazde lahve do kazde}
                                  Prelej(i,j);
                                  ZaradDoFronty;
                                  if not Hotovo then Hladina:=ZalozniHladina;
                                  End;
                     inc(j);
              until Hotovo or (j>Lahvi);
              inc(i);
       until Hotovo or (i>Lahvi);
until Hotovo or (Zacatek=Konec);
Repeat For i:=1 to Lahvi do Write(Hladina[i],' ');
       Writeln;
       Koduj(Hladina,j);
       Dekoduj(Predchozi[j],Hladina);
until Predchozi[j]=0;
End.
```

Na závěr

Nic není věčné – ani tento ročník KSP. Děkujeme všem, kteří přispěli svou troškou do mlýna, ať již tím, že úlohy vymýšleli a opravovali, nebo tím, že nás obšťastňovali krásnými a elegantními řešeními, která nás nezřídka příjemně překvapovala.

Přejeme vám krásný rok a brzké shledání u příští ročenky KSP.

Organizátoři KSP

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
1.	Jan Kára	G U Libeň. zámku, Praha	4	22	250
2.	Zdeněk Dvořák	G Nové Město na Moravě	3	22	244
3.	Pavel Nejedlý	G Vídeňská, Brno	3	20	196
4.	Pavel Šanda	G Klatovy	3	22	195
5.	Vladimír Šišma	G M. Koperníka, Bílovec	4	18	189
6.	Ondrej Gálik	G VBN, Prievidza	3	21	172
7.	Petr Rafaj	G Bán. Bystrica	4	21	169
8.	Antonín Hildebrand	G Jeseník	4	21	166
9.	Ján Lunter	G Bán. Bystrica	3	16	163
10.	Antonín Slavík	G Národní, Karlovy Vary	4	18	155
11.	Jan Vítek	G Slaný	4	16	151
12.	Alexandr Kára	G Hellichova, Praha	3	19	149
13.	Vladimír Wiedermann	G VBN, Prievidza	3	19	139
14.	Pavel Surynek	G Vlašim	4	20	131
15.	Ján Kmeť	G VBN, Prievidza	3	17	129
16.	Ondřej Nekola	G Žižkova, Kolín	3	18	128
17.	Jan Pernička	G Jana Palacha, Mělník	4	16	106
18.	Karel Karlík	G Sladkovského n., Praha	4	22	102
19.-20.	David Holec	G Tř. kpt. Jaroše, Brno	3	14	88
	Jiří Vyskočil	G Lanškroun	4	12	88
21.	Martin Wokoun	G Tř. kpt. Jaroše, Brno	3	12	87
22.-23.	Jakub Černý	G Jana Palacha, Mělník	4	11	85
	Jakub Ouhrabka	G Jeronýmova, Liberec	4	9	85
24.-25.	Milan Kryl	G Litovel	3	20	81
	Michal Šída	G Tanvald	4	11	81
26.-27.	Michal Svoboda	G Nad alejí, Praha	4	12	77
	Radek Sýkora	G M. Koperníka, Bílovec	4	12	77
28.-29.	Tomáš Hrubý	G Klatovy	3	13	71
	David Štěrba	OA Mladá Boleslav	3	14	71
30.	Martin Děcký	G M. Koperníka, Bílovec	1	14	67
31.-32.	Jakub Bystroň	G Karviná	2	11	65
	Petr Zika	G Voděradská, Praha	3	8	65
33.	Lukáš Brožovský	G Žatec	3	9	64
34.	Tomáš Vyskočil	G Lanškroun	2	10	58
35.	Radek Pelánek	G Tř. kpt. Jaroše, Brno	4	9	57
36.-37.	Stanislav Kašný	G Sušice	4	7	52
	Lucie Petráčková	SPŠST Panská, Praha	4	8	52

38.	Jan Mysliveček	G Tř. kpt. Jaroše, Brno	3	9	51
39.	Tomáš Horáček	SPŠSE Ústí nad Labem	4	4	49
40.	Pavel Cejnar	G Jeronýmova, Liberec	4	4	47
41.	Milan Roubal	G Mikulášské n., Plzeň	3	12	45
42.	Jiří Krůta	G Rakovník	3	8	41
43.	Martin Klíma	G Havlíčkův Brod	4	4	36
44.-46.	Jan Holeček	G Tř. kpt. Jaroše, Brno	3	4	35
	Tomáš Klouček	G Česká Lípa	4	9	35
	Petr Šimeček	G Tř. kpt. Jaroše, Brno	4	4	35
47.	Miloslav Brada	G Tábor	3	4	33
48.	Oldřich Vořechovský	G Rožnov p. Radhoštěm	3	8	32
49.	Miroslav Pištěk	G Sedlčany	2	5	30
50.	Vlastimil Janda	G Humpolec	4	4	28
51.	František Němec	G Zborovská, Praha	1	8	27
52.	Martin Hejna	SPŠE Dobruška	1	4	26
53.-54.	Jan Jakubův	G Vlašim	2	4	24
	Michal Kopřiva	G Městec Králové	4	9	24
55.-57.	Radomír Budínek	G Hodonín	4	4	23
	Ladislav Kavan	G Ústavní, Praha	4	3	23
	Petr Němec	G Arabská, Praha	3	3	23
58.	Aleš Kosina	G Tř. kpt. Jaroše, Brno	3	7	21
59.-61.	Ondřej Čertík	G Zborovská, Praha	-1	2	19
	Jan Drchal	G Nad Štolou, Praha	3	5	19
	Karel Volný	G Třebíč	4	2	19
62.	Jaroslav Urban	G Litovel	2	7	18
63.	David Březina	G Vídeňská, Brno	3	2	17
64.-65.	Miroslav Černý	G Kutná Hora	3	2	16
	Marcel Kolaja	G Lanškroun	4	4	16
66.-68.	Lukáš Matějka	G Lanškroun	2	5	15
	Jan Šimáček	G Roudnice nad Labem	2	2	15
	Petr Šváb	G Litoměřická, Praha	4	3	15
69.-72.	Jan Hora	G Špitálská, Praha	3	2	14
	Ondřej Kolenatý	G Teplice	3	4	14
	Šárka Štěpánová	G Rychnov n. Kněžnou	4	3	14
	David Zimandl	G Sedlčany	3	7	14
73.	Petr Chovanec	G T. Novákové, Brno	3	2	13
74.-76.	Karel Berkovec	G Mikulášské n., Plzeň	4	2	11
	Viktor Bulánek	G Ústí nad Labem	3	4	11
	Petr Žejdl	G Hlučín	2	2	11
77.	Lenka Kučerová	G Jičín	4	1	9
78.-79.	Daniel Fiala	G Sušice	2	1	7

	Ladislav Prošek	G Louny	2	3	7
80.	Tomáš Smlsal	G Městec Králové	4	1	3
81.-97.	Petr Altman	G Buďejovická, Praha	4	0	0
	Radek Cibulka	Masarykovo G, Plzeň	?	0	0
	David Karban	G Orlová	3	0	0
	Ondřej Krajíček	G Břeclav	3	0	0
	Daniel Kupka	G Štěpánská, Praha	3	0	0
	Vít Létal	G Břeclav	4	0	0
	Ivo Malý	????	3	0	0
	Vilém Maršík	G Na Pražačce, Praha	3	0	0
	Juraj Matuš ml.	G Sabinov	1	0	0
	Jan Molič	G Hlinsko	4	6	0
	Tomáš Páleníček	G Český Těšín	4	0	0
	David Pavlík	G Zlín	3	0	0
	Tomáš Richtr	G Zlín	4	0	0
	Jiří Rohan	????	?	0	0
	Lukáš Veselý	G Jablonec nad Nisou	3	0	0
	Přemysl Volf	G Hradec Králové	?	0	0
	Martin Žďárský	????	?	0	0

Obsah

Martin Mareš a kolektiv

Korespondenční seminář v programování X. ročník

Autoři a opravující úloh:
Martin Bělocký, Jirka Hanika, Michal Koucký,
Daniel Kráľ, Pavel Machek, Martin Mareš, Vít Novák,
Michal Píše, Aleš Přívětivý, Robert Šámal, Robert Špalek

Vydala Matematicko-fyzikální fakulta University Karlovy
Oddělení vnějších vztahů a propagace
Ke Karlovu 3, 121 16 Praha 2
Praha 1998

Vytisklo Reprografické středisko MFF UK
Malostranské nám. 25, 118 00 Praha 1

76 stran, 1 obrázek
Sazba písmem Computer Modern v programu TEX
Vydání první
Náklad 300 výtisků