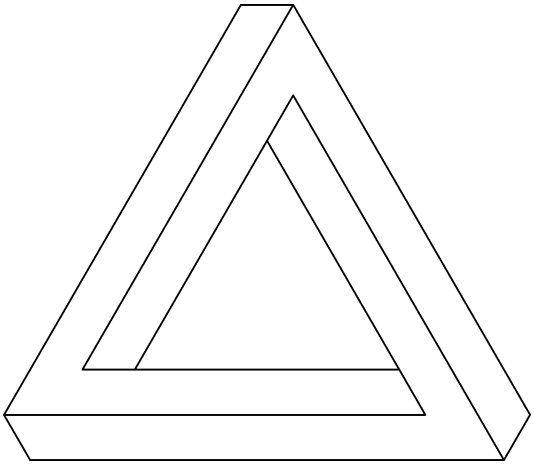
MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

IX. ročník – 1996/97



Matematicko-fyzikální fakulta
University Karlovy

MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

IX. ročník – 1996/97

Matematicko-fyzikální fakulta
University Karlovy

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož devátý 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 brněnský a 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

9-1-1 Cesta kulhavého koně — 10 bodů

Bylo-nebylo. Na jedné šachovnici žil kulhavý kůň. Je to poněkud neobvyklá figurka, ale její jméno trefně vystihuje, jakých tahů je schopna: pohybuje se na šachovnici v liché tahy jako kůň (v jedné ze souřadnic o jedno pole, v druhé o dvě), v sudé jako král (v každé souřadnici o nejvýše jedno pole) – v prvním tahu tedy táhne jako kůň, v druhém jako král, ve třetím jako kůň atd.

Vaším úkolem je navhrnout algoritmus a napsat program, který pro danou velikost šachovnice ($M \times N$) a dané souřadnice dvou políček (levé dolní políčko má souřadnice $[0, 0]$, pravé dolní $[M-1, 0]$ a levé horní $[0, N-1]$) zkonstruuje nejkratší cestu, kterou může kulhavý kůň dojít z výchozího políčka na cílové, a vypíše souřadnice všech touto cestou navštívených polí (nejkratší cesta je ta, jež jich navštíví nejméně).

9-1-2 Kupec Bengálský — 8 bodů

Starý pirát Cor Sàir odešel do výslužby a usadil se v malé přístavní vísce kdesi na zapadlém pobřeží. Jako odměnu za své služby dostal od svých druhů devět dělových koulí, z nichž jedna byla prý uvnitř plná zlata ukrytého na horší časy.

Po několika letech se Cor Sàir rozhodl, že zlatou kouli vypátrá a ostatních se nějak zbaví – zavazily mu totiž v jeho skrovné chaloupce. Záhy se dovtípil, že se to snadno pozná podle hmotnosti – železné koule by měly vážit všechny stejně, zatímco zlatá o poznání více. Jediné váhy ve vsi měl ovšem hokynář Mer Chant. Když za ním pirát přišel, seznal, že klepy se šíří rychle a kupec tuší, že by jedna z koulí mohla ukrývat bohatství, a ve své zištnosti žádá za každé zvážení na svých rovnoramenných vahách tři z koulí (splatné po konci vážení), domnívaje se, že se tak domůže všech koulí dříve, než pirát stačí zjistit, která je ta pravá.

Cor Sàir se ovšem nenechal nachytat a přes tyto náročné podmínky vymyslel postup zaručeně vedoucí ke zjištění oné zlaté koule, aniž by ji musel obchodníkovi odevzdat (a to dokonce bez použití násilí). Dokážete to také vy? (Na rovnoramenných vahách se váží tak, že nějaké koule položíme na levou misku, nějaké na pravou a dozvíme se, která miska je těžší. Obchodníkova závaží jsou nepoužitelná, poněvadž jsou příliš lehká.)

9-1-3 Strašidlóóó — 10 bodů

Město Hauntry bylo pravoúhlou sítí ulic rozděleno na $M \times N$ bloků domů, v každém z nich žil určitý počet nájemníků. Nebydlel-li v domě nikdo, záhy

se tam usadila strašidla. Starosta si nechal vypracovat plán města s přesnými počty obyvatel jednotlivých domů a položil si zajímavou otázku: kde je největší obdélník složený ze samých strašidelných domů?

Na vás je, abyste sestrojili program schopný pro dané rozměry města a matici rozložení obyvatel co nejrychleji nalézt největší (tedy maximální plochu mající) obdélník v této matici složený ze samých nul.

9-1-4 !ʇǝɹɔǝs qoT — 15 bodů

Poslední úloha každé série tohoto ročníku *KSP* bude nějak souviset s šifrováním. Romantická historie šifrování (viz literatura) se sice pěkně čte, ale úlohy z programování zkonstruované na jejím základě jsou nezajímavé (příliš jednoduché nebo naopak téměř neřešitelné). Proto začneme rovnou s moderními šiframi.

Velká skupina v dnešní době používaných šifer je založena na operacích s velmi velkými přirozenými čísly (např. stomístnými). Abychom vás na tuto skutečnost připravili, bude první úloha seriálu pouze „zahřívací“:

Vymyslete, jak v počítači nejlépe reprezentovat velmi velká přirozená čísla, a naprogramujte procedury nebo funkce pro základní operace s nimi: sčítání, odčítání, násobení a dělení se zbytkem.

Literatura:

(1) Otokar Grošek, Štefan Porubský: Šifrovanie – algoritmy, metódy, prax, Grada 1992.

(2) Dr. Vlastimil Klíma: Utajené komunikace, seriál o šifrování v časopisu Chip, 6/94 – 12/95.

9-2-1 Psí funkce — 10 bodů

Funkce $\Psi(n)$ je pro přirozená čísla n definována následujícím předpisem:

- $\Psi(1) = 2$
- $\Psi(n+1) = 2^{\Psi(n)}$

Napište program, který pro dané *velké* přirozené číslo $n > 1000$ spočte hodnotu $\Psi(n) \bmod 13$.

9-2-2 Tiskařský šotek — 10 bodů

V tiskárně jedněch (pro jistotu nejmenovaných) novin se usídlil tiskařský šotek. Čas od času se rozhodl, že v sázeném textu prostě přidá, ubere nebo vymění nějaké písmenko, a tak změní jeho smysl (obvykle na něco žertovně laděného – například „Pražský průmysl masný“ na „Pražský průmysl mastný“).

Ledva redaktoři zjistili přítomnost šotka, pořídili si počítačový systém pro kontrolu pravopisu (tzv. spelling-checker, to jest program, který upozorňuje na

jemu neznámá slova). Proto šotek musí své změny provádět tak, aby pokaždé vzniklo spelling-checkeru známé slovo a jeho čin tak zůstal (alespoň do vytištění příslušné stránky) v tajnosti.

Vás požádal, abyste mu ku pomoci v jeho počínání napsali jednoduchý program, jenž dostane k dispozici kompletní seznam slov známých spelling-checkeru a pak se jej bude šotek moci ptát, na jaká všechna známá slova může zadané slovo změnit popsanými „jednopísmenkovými“ úpravami.

9-2-3 Překla-datel — 10 bodů

Jednoho dne se stalo, že na Zemi přistáli Ufouni, malá to zelená stvoření obývající zapomenuté kouty Vesmíru. Nabídli nám, že si z paměti jejich palubního počítače můžeme odčerpat libovolné množství informací o jejich rodné planetě, pokud to ovšem dokážeme.

Jediným problémem při připojování ufounského počítače k našemu se ukázalo být to, že Ufouni zapisují čísla ve dvojkové soustavě v přesně opačném pořadí číslic než jak činíme my – to jest s nejméně významnou číslicí na začátku místo na konci (například číslo 34 zapisujeme jako 10010 a oni jako 01001).

Při konstrukci překládacího zařízení je tedy klíčovým problémem vytvořit program, který k danému 128-bitovému číslu N nalezne číslo N', jež má přesně opačný binární zápis: má-li N binární zápis $a_{127}a_{126}\ldots a_1a_0$, musí mít N' zápis $a_0a_1\ldots a_{126}a_{127}$.

Překládací stroj je ovšem programován ve velice jednoduchém programovacím jazyce, který disponuje stovkou 128-bitových proměnných a jedním jediným příkazem: přiřazením typu $a = b \circ c$, kde a je proměnná, b a c jsou proměnné nebo konstanty a $\circ$ je jedna z operací `+` (sčítání), `-` (odčítání), `*` (násobení), `/` (dělení), `&` (bitový logický součin), `|` (bitový logický součet), `<` (bitový posun b doleva o c bitů) a `>` (bitový posun b doprava o c bitů). Vstup dostanete v proměnné x, výstup budiž na konci výpočtu uložen v proměnné y.

Napište *co nejkratší* program pro překládací stroj, který nalezne příslušné „reverzní číslo“.

9-2-4 Nesčetní námětové — 10 bodů

Za devatero horami a devatero řekami byla velká země. Tamější obyvatelé se rozhodli, že by mohli svou tisíc let starou státní vlajku vyměnit za jinou, poněkud méně okoukanou. Sešlo se úžasné množství různých námětů, ty byly očíslovány (37: zelený, fialový a hnědý pruh, 666: čert s rudou hvězdou v rozích, 919: kosa a palička na maso apod.) a bylo vyhlášeno celonárodní referendum, které má z těchto námětů vybrat ten pravý s tím, že za právoplatného vítěze je možno považovat jen takový námět, jenž získal nadpoloviční většinu hlasů.

Vás najali, abyste napsali program pro vyhodnocování výsledků voleb. Dostane pole o N prvcích (pozor, N je velké číslo), jehož každý prvek odpovídá

jednomu hlasujícímu občanovi a jeho obsahem je číslo námětu daným občanem voleného. Program má co nejrychleji nalézt číslo, které se v poli vyskytuje více než $N/2$-krát nebo odpovědět, že žádné takové tam není.

9-2-5 Top secret! 10 bodů

Při klasickém šifrování je bezpečnost založena na utajení jak klíče pro šifrování, tak klíče pro dešifrování. Dlouho se zdálo nemožné, aby existovala šifra, u které by bylo možné zveřejnit klíč pro šifrování, aniž by byla výrazně narušena bezpečnost (klíč pro dešifrování samozřejmě musí zůstat utajen). Jedním z důvodů, který k tomu vedl, bylo, že pro se pro šifrování i dešifrování používal stejný klíč.

V *kryptosystémech s veřejným klíčem* jsou klíče dva: jeden šifrovací (ten je zveřejněn) a jeden dešifrovací (ten je utajen). Tyto dva klíče musí být *komplementární*, tj. dešifrováním zašifrované zprávy bychom měli dostat zprávu původní. Navíc nesmí být možné bez znalosti tajného klíče „rychle“ zašifrovanou zprávu rozluštit.

Mezi nejvíce používané kryptosystémy s veřejným klíčem patří systém *RSA,* jehož objev oznámili 4. dubna 1977 pánové Ronald L. Rivest, Adi Shamir a Leonard Adleman z Massachusetts Institute of Technology.

RSA funguje takto: veřejný klíč je nějaká dvojice čísel $\langle N, s\rangle$, tajný klíč je dvojice čísel $\langle N, t\rangle$. Šifrovat můžeme celá čísla x, která splňují podmínku $1 < x < N$. Šifrujeme pomocí vztahu $y = x^s \bmod N$, dešifrujeme pomocí vztahu $x = y^t \bmod N$. (Zamyslete se, jak šifrovat text a proč není vhodné šifrovat čísla $x = 0$ a $x = 1$.)

Klíče samozřejmě nesmí být náhodně zvolená čísla – o generování klíčů bude úloha příští.

Příklad: Veřejný klíč je $\langle 143, 23\rangle$, tajný $\langle 143, 47\rangle$. Šifrováním čísla 123 dostaneme číslo

$$123^{23} \bmod 143 = 63,$$

dešifrováním tohoto čísla dostaneme opět

$$63^{47} \bmod 143 = 123.$$

Úloha: V již neplatném vydání Veřejného šifrovacího seznamu je u jména KSP uvedena dvojice

$$\langle 402104495055726404839198130029, 37865494736593524343585634769 7\rangle$$

Vaším úkolem je zašifrovat pomocí tohoto veřejného klíče číslo

$$12345678901234567890123456789 0.$$

Nezapomeňte nám napsat, jak jste to rafinovaně provedli. Abyste si mohli ověřit, že vše funguje tak, jak má, prozradíme vám ještě, že tajný klíč je

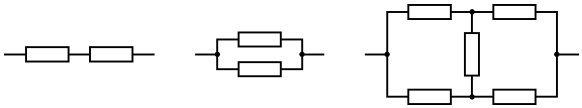
$$\langle 402104495055726404839198130029, 318469828822789734819186350713\rangle .$$

9-3-1 Odporné odpory pana Odporného — 10 bodů

Žil byl jednou jeden elektrikář, který se jmenoval pan Odporný. Náplní jeho práce bylo skládat odpory. Lidé si na něj ale často stěžovali, jelikož chtěli-li po něm poskládat z dostupných rezistorů rezistor o daném odporu, museli odporně dlouho čekat, protože panu Odpornému šlo skládání velice pomalu, a dokonce občas pan Odporný odpověděl, že úkol splnitelný není, což mu nevěřili.

Vaším úkolem je panu Odpornému pomoci a vymyslet algoritmus a napsat program, který bude daný problém řešit: to jest pro daný seznam hodnot odporů dostupných rezistorů a daný výsledný odpor buďto příslušnou kombinaci nalezne nebo odpoví, že taková neexistuje. Všechny odpory jsou zadávány v Ohmech. Pokud existuje více řešení, nalezněte to, v němž se použije nejmenší počet rezistorů.

Již dostupné (ať již přímo nebo složením jiných) rezistory je možno spojovat buďto sériově (to jest za sebe – jako na prvním obrázku) nebo paralelně (vedle sebe – na druhém obrázku). Odpor výsledné kombinace je pro sériové zapojení součtem jednotlivých dílčích odporů, pro paralelní je převrácenou hodnotou součtu jejich převrácených hodnot. Jiná zapojení (jako např. to na obrázku třetím) nejsou přípustná.



Příklad 1: Jsou k dispozici 2 rezistory o odporech 10Ω a 20Ω, požadovaný odpor je 5Ω. Výsledek: Nelze sestavit.

Příklad 2: Jsou k dispozici rezistory o odporech $(1\Omega, 1\Omega, 1\Omega, 1\Omega, 50\Omega)$, požadovaný odpor je 50,25Ω. Je možno použít například kombinaci zapsatelnou jako `(1x1x1x1)+50` (‘`x`’ značí paralelní zapojení, ‘`+`’ sériové).

9-3-2 Cosi prohnilého? — 12 bodů

V jednom nejmenovaném království zjistili královští rádcové, že existují silnice, které když se rozpadnou, silniční síť (a tím pádem i celé království) se rozdělí na dvě části. To se králi nelíbilo a uložil jim, aby kázali přednostně udržovat tyto kritické silnice. Nyní ovšem potřebují vědět, které to jsou a vás najali, abyste vyrobili program, jenž je najde.

Vstupem programu je počet měst N (města jsou očíslovaná $0, \ldots, N-1$) následovaný dvojicemi čísel, které reprezentují silnice (silnice jsou obousměrné).

Seznam je ukončen dvojicí $(0,0)$, což je okruh kolem královského paláce. Výstupem je seznam kritických silnic.

Příklad: 8 $(6,7)$ $(1,0)$ $(5,4)$ $(2,3)$ $(5,6)$ $(1,3)$ $(3,4)$ $(2,1)$ $(5,7)$ $(0,0)$. Výstup: $(0,1)$ $(3,4)$ $(4,5)$.

9-3-3 Mikroassembler — 12 bodů

Firma Sirius Cybernetics uvedla s ohromnou reklamou na trh počítač μ_1 s údajně nejjednodušší možnou instrukční sadou. Samozřejmě to (jak již tomu u reklamních sloganů bývá) není pravda. Na vás je, abyste našli přebytečnou instrukci (to jest takovou, že každý program tuto instrukci obsahující je možno přepsat na program, který dělá totéž a tuto instrukci nepoužívá) a popsali, jak ji lze nahradit. Pokud existuje takových instrukcí více, nalezněte všechny a zkuste dokázat, že ty ostatní jsou již vskutku nutné. A nyní již k tomu, jak onen počítač funguje:

μ_1 má paměť složenou z jednotlivých paměťových buněk očíslovaných přirozenými čísly, přičemž v každé buňce je uloženo opět libovolné přirozené číslo. Na počátku běhu programu jsou na smluvených místech v paměti uložena vstupní data (ostatní buňky obsahují nedefinované hodnoty), na konci výpočtu pak na jiných místech uloženy výsledky. K dispozici jsou následující instrukce:

- `INC` x – zvýšení obsahu paměťové buňky číslo x o 1.
- `DEC` x – snížení obsahu paměťové buňky číslo x o 1. Pokud tato již byla 0, zůstane 0.
- `CLR` x – uložení nuly do dané paměťové buňky.
- `JEQ` x,y,z – pokud je v paměťových buňkách x a y uloženo totéž číslo, skočí o z instrukcí dopředu (z je libovolné celé číslo), v opačném případě neprovede nic.

Program začíná běh svojí první instrukcí a končí při pokusu o skok *před* tuto instrukci. Na ukázku uveďme například program pro sečtení čísel na adresách 1 a 2, přičemž výsledek uloží na adresu 0.

```
CLR    3
CLR    0
JEQ    1,3,4
DEC    1
INC    0
JEQ    3,3,-3
JEQ    2,3,-7
DEC    2
INC    0
JEQ    3,3,-3
```

9-3-4 Komplikátor — 11 bodů

Jistě vás při čtení předchozí úlohy napadlo, že možnosti popisovaného assembleru μ_1 jsou naprosto minimální. Kupodivu to není tak docela pravda: jde v něm naprogramovat prakticky cokoliv, co jde zapsat v Pascalu. Jelikož by ale bylo přeci jen příliš pracné vytvořit překladač z Pascalu do tohoto assembleru, zkuste napsat program, který do tohoto assembleru překládá alespoň běžné celočíselné aritmetické výrazy obsahující:

- Operátory '`+`', '`-`', '`*`', '`/`' a '`%`' (zbytek po dělení) se správnými prioritami (tedy '`*`', '`/`' a '`%`' mají přednost před '`+`' a '`-`').
- Závorky '`(`' a '`)`'.
- Proměnné '*a*'–'*z*'.

Vstupem vašeho programu by měl být výraz v textovém tvaru, výstupem pak program pro μ_1 vyčíslující tento výraz, přičemž hodnoty proměnných budou zadány na adresách 1 (a) až 26 (z) a hodnota výrazu bude po skončení výpočtu uložena na adrese 0. Například ukázkový program od předchozí úlohy je jednou ze správných odpovědí pro výraz $a + b$.

9-3-5 !ʇǝɹɔǝs ɹoT — 10 bodů

Dnešní úloha se bude zabývat volbou vhodných klíčů pro šifrovací algoritmus `RSA` uvedený v minulé sérii.

Náhodně zvolíme dvě navzájem různá dostatečně velká prvočísla $p, q, p \neq q$. Označíme si $N = p \cdot q$ a $K = (p-1) \cdot (q-1)$. (O tom, jak náhodně volit dostatečně velká prvočísla, bude úloha příští.)

Náhodně zvolíme šifrovací exponent s, $1 < s < K$ tak, aby byl nesoudělný s K.

Najdeme celé číslo $0 < t < K$, které vyhovuje rovnici $(t \cdot s) \bmod K = 1$. Je možné dokázat (to po vás nechceme), že takové číslo existuje právě jedno.

Zveřejníme dvojici $\langle N, s \rangle$ – veřejný klíč, pomocí kterého může kdokoli zašifrovat nám určenou zprávu. Do trezoru si uložíme tajný klíč – dvojici $\langle N, t \rangle$, kterou budeme dešifrovat příchozí zprávy.

Příklad: Zvolili jsme prvočísla $p = 11$, $q = 13$ a šifrovací exponent $s = 23$. Pro tyto hodnoty je $N = 143$, $t = 47$.

Úloha: Vymyslete algoritmus a sestrojte program, který pro daná dvě prvočísla vypočítá nějakou dvojici klíčů pro šifrování algoritmem RSA. (Návod: Pokuste se zobecnit Euklidův algoritmus pro hledání největšího společného dělitele.)

9-4-1 Cor Sàir strikes back — 12 bodů

Poté, co pirát ve výslužbě Cor Sàir zdárně vyřešil svá trable s hamižným hokynářem Mer Chantem (viz úloha 9-1-2), stal se v širém okolí uznávaným

odborníkem přes vážení břemen a třídění předmětů. Jednoho dne však dostal od královské výzvědné služby následující tajné poslání:

Agenti zadrželi na poště 12 značně těžkých balíků adresovaných do sousední ne tak docela spřátelené země a od zpravodajské kanceláře JPP (Jedna Paní Povídala) se dozvěděli, že jeden z těchto balíků by měl obsahovat tajné dokumenty ukradené ministerstvu deratizace. Jelikož všechny balíky vypadaly stejně, agenti záhy přišli na to, jak inkriminovanou zásilku odhalit – měla by se lišit hmotností! Netušili však, je-li lehčí nebo těžší a nechtěli mnoha váženími vzbudit přílišnou pozornost okolí (my i vy jistě tušíte, že ji vzbudí nehledě na počet vážení, ale pochopte, že *oni* to nevědí), a proto Cor Sàirovi zadali, ať vymyslí, jak co nejmenším počtem vážení na rovnoramenných vahách onen balík objevit a ještě zjistit, zda je lehčí neb težší.

Po vás pro tentokráte nechceme, abyste psali jakýkoliv program, pouze zkuste popsat postup (tedy vlastně algoritmus), jak tento problém vyrešit.

9-4-2 Green Bit Problem — 10 bodů

Jak již bylo zmíněno v předminulé sérii, na Zemi se začali vyskytovat Ufouni, malá to zelená stvoření obývající zapomenuté kouty Vesmíru a my si s nimi již umíme vyměňovat informace prostřednictvím 128-bitových binárních čísel – v úloze 9-2-3 jste vyřešili, jak taková čísla převádět do nám čitelného tvaru (to jest jak je zrcadlově převracet).

Při komunikaci s Ufouny se ovšem objevil ještě jeden zajímavý problém: jak u takového dlouhého čísla *rychle* zjisit, kolik je v něm jedničkových bitů. Pro tento účel se používá téhož překládacího stroje, jenž se programoval pro převracení bitů. Vstup dostanete v proměnné x, výstup budiž na konci výpočtu uložen v proměnné y.

9-4-3 Hellish Clocks — 13 bodů

V jednom pekle měli svérázný systém měření času: v každém z N pater pekla byly jedny hodiny ukazující na svém ciferníku čas v pekelných hodinách (těch je každý den 37). Jelikož mnohé z hodin se opožďovaly, předcházely, občas zastavovaly či dělaly jiné nepředloženosti, pořídili si čerti centrální ovládání: od každých hodin vedly dráty k řídícímu pultu v Luciferově kanceláři, na němž bylo N tlačítek, jimiž mělo jít posunout ručičku daných hodin na nejbližší další pekelnou hodinu.

Jenže jak to už v pekle bývá, i tento systém měl nějaký háček: každé tlačítko totiž neposouvalo jenom jedny hodiny a jenom jednu hodinu, ale ručičky na hned několika cifernících o různé počty hodin. Nakonec se ale podařilo zjistit, jak které tlačítko funguje.

Vaším úkolem je napsat program, kterému čerti zadají, o kolik hodin posune které tlačítko kupředu ručičky na kterém ciferníku, kolik které hodiny

zrovna ukazují a na jaký čas si je přejí nastavit (pro všechny ciferníky stejný) a program odpoví, kolikrát zmáčknout které tlačítko (na pořadí mačkání evidentně nezáleží), aby bylo dané konfigurace dosaženo, případně sdělí, že to nejde.

9-4-4 Whatsit? 10 bodů

Zkuste přijít na to, k čemu je (mimo zmatení řešitelů, samozřejmě) určena následující pascalská funkce a jak by se dala naprogramovat efektivněji:

```
function whatsit(a,b:word):word;
const s:string='><+*>>*#<$*#>>+++*><#<:=]'
 + '#-><$->+<+<#><$-<+><#>>!*+*><>+#:-)';
var k:array [0..6] of word;
begin
   k[0]:=1; k[1]:=1; k[2]:=2;
   k[3]:=a; k[4]:=b;
   while chr(k[0])<=s[0] do begin
   case s[k[0]] of
      '<': k[1]:=(k[1]+6) mod 7;
      '>': k[1]:=(k[1]+2) mod 7;
      '#': if k[k[1]]<>0 then k[0]:=k[2];
      '$': k[2]:=k[0];
      '-': dec(k[k[1]]);
      '+': inc(k[k[1]]);
      '!': k[k[1]]:=k[1];
      else inc(k[k[1]],k[k[1]]);
      end;
   inc(k[0]);
   end;
whatsit:=k[6];
end;
```

9-4-5 Something rotten? 12 bodů

Váš program z minulé série na hledání kritických silnic si královští úředníci velice oblíbili a nyní vás žádají o vyřešení podobného problému: potřebují zjistit, kolik silnic se jim může rozpadnout, aniž by se království rozdělilo na dvě části.

Tedy: vstupem programu je počet měst N (města jsou očíslována $0, \ldots, N-1$) následovaný dvojicemi čísel reprezentujících obousměrné silnice. Seznam je ukončen dvojicí $(0,0)$, což je okruh kolem královského paláce. Výstupem je počet silnic, které se mohou rozpadnout a přitom ještě nerozdělit silniční síť na dvě části.

Příklad: 5 $(0,1)$ $(0,2)$ $(0,3)$ $(1,2)$ $(2,3)$ $(1,4)$ $(2,4)$ $(3,4)$ $(0,0)$. Výstup: 2.

9-4-6 !tɘɿɔɘƨ qoT 10 bodů

Minule jsme vám slíbili, že tato úloha bude o generování náhodných velkých prvočísel. A opravdu vás nezklameme. Taková prvočísla se generují většinou

tak, že se najde velké náhodné číslo a pak se testuje na prvočíselnost. Pokud se jedná o číslo složené, test na prvočíselnost opakujeme s číslem o jedničku větším (pokud nemáme opravdu dokonalý generátor náhodných čísel, není vhodné generovat nové náhodné číslo, ježto pak není zaručena konečnost algoritmu).

Ale jak testovat u velkých čísel prvočíselnost? Klasický algoritmus – pokusné dělení čísly od 2 od $\sqrt{n}$ – je nepoužitelný, jelikož bychom se výsledku nedočkali. Nejčastěji se používá pravděpodobnostní algoritmus založený na následujícím tvrzení, které je rozšířením malé Fermatovy věty:

(i) Jestliže pro nějaké $0 < a < n$ platí

$$a^{n-1} \bmod n \neq 1,$$

je n číslo složené. A obráceně:

(ii) Jestliže pro nějaké $0 < a < n$ platí

$$a^{n-1} \bmod n = 1,$$

je n prvočíslo s pravděpodobností větší než 0,5.

V praxi se ukazuje, že tato pravděpodobnost je daleko větší než 0,5 a že pro náhodná a jsou pravděpodobnosti rozumně nezávislé.
Návod na testování prvočíselnosti čísla n:

1. Vyloučíme triviální případy: provedeme pokusné dělení n prvočísly od 2 do (třeba) 1000. Pokud je n některým z nich dělitelné, je již vše jasné.
2. Vygenerujeme náhodné číslo $0 < a < n$ a vypočteme $a^{n-1} \bmod n$. Pokud vyjde něco jiného než 1, je n jasně složené a končíme.
3. Bod 2. několikrát (třeba 20-krát) zopakujeme, abychom zvýšili pravděpodobnost správnosti výsledku.
4. Prohlásíme, že číslo n je asi prvočíslo.

Z teoretického hlediska by se mohlo zdát, že generování prvočísel pouze s nějakou praděpodobností je nepoužitelné. Ve vesmíru a životě je ale málokdy něco na 100%. Když počítače, na kterých výpočty provádíme, mají pravděpodobnost chyby 10^{-4}, můžeme jásat nad prvočísly s pravděpodobností chyby 10^{-10}

Úloha: Běžné funkce pro generování náhodných čísel generují pouze čísla *pseudonáhodná:* posloupnost „náhodných“ čísel se začne po čase opakovat. Zdůvodněte, proč je tento fakt při generování klíčů pro RSA nepříjemný, a zkuste vymyslet, jak generovat opravdu náhodná čísla. Při řešení tohoto problému se nemusíte omezovat na čistě programátorské postupy. Provedení praktických experimentů a použití znalostí z jiných oborů (např. fyziky nebo chemie) oceníme zvláštní prémií.

Poznámka: Popsaný algoritmus ovšem není *zcela* korektní – negeneruje totiž všechna prvočísla se stejnou pravděpodobností. Zkuste se zamyslet nad tím, proč a jak by se tento problém dal odstranit.

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

9-1-1 Cesta kulhavého koně aneb Chyták! — Martin Mareš

Ačkoliv byla tato úloha původně míněna jako něco jednoduššího „na zahřátí“, byli jsme nemile překvapeni, že téměř nikdo neposlal korektní řešení. Vyskytovaly se následující neúspěšné pokusy:

1. Heuristiky – to jest řešení založená na různých více či méně podivných předpokladech (typu „když se budu pohybovat tak, abych postupně zmenšoval součet absolutních hodnot svých souřadnic a souřadnic cíle, bude taková cesta zaručeně nejkratší“), které měly společné jedno: chyběl korektní důkaz toho, že nalezená cesta bude vskutku optimální.

A také že bylo velice často snadné vymyslet protipříklad. To si dokonce uvědomili i někteří autoři takovýchto řešení a začali do nich přidávat dodatečné omezující podmínky typu „když jsou dvě možnosti podle mého základního kriteria rovnocenné, vyberu ten tah, kterým se dostanu blíže ke středu“ nebo „pokud jsem již od cíle vzdálen v každé ose méně než 3 pozice, dojedu podle přiložené tabulky“ – leč opět opomněli zdůvodnit, proč jiné „špatné“ situace nastat nemohou.

Pro tato „vylepšovaná“ řešení ovšem již hledání protipříkladů nebylo tak snadným, pročež ne každému jsme takový s opraveným řešením poslali. Za všechny zde budiž uveden „universální protipříklad“ na alespoň polovinu chybných algoritmů: šachovnice 4×4, vycházíme z levého horního rohu $(0,0)$ a chceme se dostat do pravého dolního $(3,3)$. Optimální cesta je například $(0,0)$ – $(2,1)$ – $(1,2)$ – $(3,3)$. Typická heuristika skáče z $(0,0)$ na $(2,1)$, pak na $(3,2)$ nebo $(2,3)$ a odtamtud se již jediným koňským skokem zaručeně do cíle nedostane, tudíž vygeneruje cestu, která není nejkratší (pokud ovšem vůbec nějakou nalezne – mnohé z vašich programů se prostě na takovémto zadání zacyklily na věčné časy).

2. Jednoduchý algoritmus typu „vlna“ – začneme políčkem se vzdáleností 0, pak hledáme políčka o vzdálenosti 1, ... vždy z políček o vzdálenosti n hledáme políčka se vzdáleností $n+1$, prostě klasický postup hledání do šířky.

Na tomto postupu ještě nic špatného není, ale přeci jen tu je jeden háček: nestačí si totiž pro každé políčko pamatovat délku nejkratší cesty ze startu do něj a tyto tahy se pokoušet prodlužovat – tím totiž zapomínáte na to, že tahy se musí pravidelně střídat. Ačkoliv nejkratší možný tah vedoucí na pole (x,y) končí tahem koně, nejkratší tah na pole $(x+2,y+1)$ například může být skokem koně z (x,y) s tím, že na (x,y) se dostanete o 1 tah delší cestou, která končí tahem krále, abyste mohli správně navázat.

Místo tohoto poněkud zmateného popisu asi poslouží lépe ukázka, kdy to skutečně selže. Není pro ni ovšem třeba chodit příliš daleko: universální protipříklad pro heuristiky prokáže svoji universalitu i zde. Prvním tahem koně se můžete dostat na $(1,2)$ a $(2,1)$, z těchto pozic pak králem na libovolnou pozici mimo $(0,0)$ a $(3,3)$, tedy i na $(1,2)$ a $(2,1)$, což na první pohled vůbec není rozumné, neboť tam se dokážeme dostat jedním tahem (což si program rovněž uvědomí a na tyto tahy spokojeně zapomene). Nu ano, ale pole $(3,3)$ je dostupné ve třetím tahu (tedy koněm) pouze z $(1,2)$ a $(2,1)$, na která se ovšem musíme dostat králem, takže i takovýto program správné řešení nenalezne.

3. Backtracking alias prohledávání do hloubky není z principu špatně – na rozdíl od dříve zmíněných „řešení“ alespoň správnou cestu vždy najde – ale zato za rekordně dlouhou dobu, která závisí na velikosti šachovnice exponenciálně. Takže pro šachovnici 30×30 je již v praxi naprosto nepoužitelný (miliony let na výpočty obvykle k dispozici nemáme).

A jak tedy? Po tomto vyčerpávajícím (spíše čtenáře nežli téma) výčtu špatných řešení přikročme k nějakému správnému: Použijeme opět prohledávání do šířky, ovšem trošku upravené, aby fungovalo korektně. Pro každé pole si nebudeme pamatovat délku *jedné* nejkratší cesty do něj, nýbrž cest dvou – té, která končí tahem koně a té, která končí tahem krále.

Na počátku víme, že do výchozího políčka se můžeme dostat cestou délky nula, která de facto končí tahem krále – žádný tah tam sice nebyl, ale dál táhneme tak, jako bychom na něj navazovali – to jest koněm.

Nyní postupně hledáme všechna políčka o vzdálenostech 1, 2, $\ldots n$, a to tak, že máme-li nalezena všechna políčka vzdálená od počátku méně než k, snadno najdeme i ta, jež mají vzdálenost k – jsou to taková, do kterých se umíme dostat koňským skokem (je-li k liché) nebo královským krokem (je-li sudé) z políček o vzdálenosti $k-1$.

Leč nás pro každé políčko zajímají pouze délky *nejkratší* „královské“ a „koňské“ cesty, tudíž dostaneme-li se novým tahem délky k někam, kam jsme se již dokázali dostat kratším tahem *stejného typu*, můžeme nový tah s klidným svědomím ignorovat, aniž bychom udělali stejnou chybu jako v „řešení“ č. 2, jelikož všechna možná pokračování nového tahu mohou být (kratšími!) pokračováními onoho kratšího tahu, aniž bychom porušili požadavek střídání.

Bylo by ale zbytečné v každém kroku hledat všechna políčka o dané vzdálenosti v nějaké matici. Použijeme tedy fintu: zavedeme frontu F dosažitelných, ale ještě nezpracovaných políček, která bude na začátku obsahovat pouze výchozí pole, během výpočtu pak nezpracovaná políčka seřazená vzestupně podle jejich vzdáleností od startu. Každý krok pak bude vypadat takto:

1. Vyber jedno políčko (x,y) z F (tedy nejbližší ještě nezpracované). Pokud byla fronta prázdná, jsme v koncích a cesta neexistuje.

2. Pokud se toto políčko rovná cílovému, nalezli jsme slíbenou nejkratší cestu a končíme.
3. Vezmeme všechna políčka z (x, y) dosažitelná, přičemž zkoušíme napojit jak tah koně, tak tah krále (pro každý víme, jak bude dlouhý, neboť si to pamatujeme v poli).
4. Pro každé z těchto políček testujeme, zda jsme se do nich již dokázali dostat cestou daného typu, která byla kratší. Pokud nikoliv, poznamenáme si tuto (pro daný typ tahu zaručeně nejkratší) vzdálenost a políčko zařadíme na konec fronty.

Každé políčko se tedy do fronty mohlo dostat nejvýše dvakrát – jednou jako koncové pro královský tah, jednou pro tah koňský. Odhadnout, jak velkou frontu budeme potřebovat, je netriviální, ale více než $2M \cdot N$ políček se nám v ní zaručeně nesejde (M a N jsou rozměry šachovnice).

Zbývá ještě dořešit, jak vypisovat nalezenou cestu. Možnosti jsou dvě: jedna by byla začít v cílovém políčku, to vypsat, najít dostupné (resp. doskočné – podle vzdálenosti) políčko s vzdáleností menší o 1 (zaručeně existuje) a tím pokračovat, jako by ono bylo tím cílovým atd. až se dobereme k políčku výchozímu. Nebo můžeme ke každému poli poznamenat pro oba typy tahů nejen jak dlouhý byl minimální tah, ale také odkud přišel a stejným způsobem vypsat cestu „pozadu“. Vybereme si první variantu, jelikož je paměťově méně náročná.

A jak to je s časovou a paměťovou složitostí? Každé políčko (těch bylo $M \cdot N$) jsme „vzali do ruky“ nejvýše dvakrát a udělali pro něj nějaké konstantní množství operací. Při vypisování cesty jsme zaručeně nepoužili žádné políčko více jak jednou. Časová složitost je tedy $O(M \cdot N)$. Paměť potřebujeme jednak na frontu ($2M \cdot N$ položek), jednak na pole obsahující vzdálenost ($k \cdot M \cdot N$ položek, kde k je nějaké přirozené číslo). Dohromady tedy rovněž $O(M \cdot N)$.

A je toto řešení skutečně optimální? Se smutkem v očích vám mohu prozradit, že nikoliv. Existuje řešení s časovou složitostí $O(M + N)$, nicméně je poměrně složité a jeho důkaz (alespoň nejlepší, který se nám podařilo nalézt) je velice dlouhý, pročež jej zde neuvedeme. Nastiňmež alespoň základní myšlenku: Netriviálním trikem je možno pro každá dvě políčka spočítat v konstantním čase jejich vzdálenost (to je právě to, co se musí složitě dokazovat) a poté můžeme použít první z popsaných metod na vypisování cesty přímo, aniž bychom před tím cokoliv počítali.

```
#include <stdio.h>

#define MAX 30                          /* Mez velikosti šachovnice */
int M, N;                               /* Skutečná velikost šachovnice */
int x0, y0;                             /* Výchozí pole */
int x1, y1;                             /* Cílové pole */
int F[2*MAX*MAX][2];                    /* Fronta */
int r, w;                               /* Čtecí a zápisový index fronty */
```

```
typedef vzdal[MAX][MAX];
vzdal K, H;                                   /* Královské a koňské vzdálenosti */

/* Tabulky možných tahů: Pro oba typy seznam rozdílů pro obě osy zvlášť, ukončeno
kombinací (0,0) */

int Kx[] = { -1, -1, -1, 0, 0, 1, 1, 1, 0 };    /* Král */
int Ky[] = { -1, 0, 1, -1, 1, -1, 0, 1, 0 };
int Hx[] = { -2, -2, -1, -1, 1, 1, 2, 2, 0 };    /* Kůň */
int Hy[] = { -1, 1, -2, 2, -2, 2, -1, 1, 0 };

void cesty (int vzd, vzdal K, int *dx, int *dy, int xx, int yy)
{                                             /* Zkouší všechny tahy z pozice (xx,yy) */
int x, y;

    if (vzd < 0)                              /* Běda, sem se nedostaneme! */
        return;
    while (*dx || *dy)
        {
            x = xx + *dx++; y = yy + *dy++;
            if (x < M && y < N && K[x][y] < 0)   /* Sem to ještě neumíme! */
                {
                    F[w][0] = x; F[w][1] = y; w++;    /* Do fronty s tím! */
                    K[x][y] = vzd+1;          /* Už umíme... */
                }
        }
}

int hledej (void)                             /* Hledání cesty */
{
int x, y;

    r = 0; w = 1; F[0][0] = x0; F[0][1] = y0;    /* Inicializujeme frontu */
    for (x=0; x<M; x++)                       /* Na počátku vše nedosažitelné... */
        for (y=0; y<N; y++)
            K[x][y] = H[x][y] = -1;
    K[x0][y0] = 0;                            /* ... mimo počátečního pole */

    while (r < w)                             /* Dokud je ve frontě něco */
    {
        x = F[r][0]; y = F[r][1]; r++;        /* Vezmi z fronty */
        if (x == x1 && y == y1)               /* V cíli? */
            return 1;
        cesty (K[x][y], H, Hx, Hy, x, y);    /* Zkoušíme prodlužovat */
        cesty (H[x][y], K, Kx, Ky, x, y);
    }
    return 0;
}

void vypis (void)                             /* Výpis nalezené cesty */
{
int x, y;                                     /* Aktuální pozice */
int dist;                                     /* Vzdálenost od startu */
int kun;                                      /* Poslední tah koněm */
int *dx, *dy;
```

```
    x = x1; y = y1;                          /* Od konce... */
    kun = (H[x][y] >= 0 && H[x][y] < K[x][y]);   /* A koněm nebo králem? */
    dist = kun ? H[x][y] : K[x][y];
    for (; ; )
        {
            dist--;                          /* A zase o 1 blíž... */
            printf ("(%d,%d)\n", x, y);
            if (x == x0 && y == y0)
                break;                       /* Hotovo */
            if (kun)
                dx=Hx, dy=Hy;
            else
                dx=Kx, dy=Ky;
            while (*dx || *dy)               /* Hledáme předchozí */
                {
                    int xx=x + *dx++, yy=y + *dy++;
                    if (xx < M && yy < N && (kun ? K : H) [xx][yy] == dist)
                        {
                            x = xx, y = yy;
                            break;
                        }
                }
            kun = !kun;
        }
}
int main (void)
{
    scanf ("%d%d%d%d%d%d", &M, &N, &x0, &y0, &x1, &y1);
    if (hledej ()) vypis ();
    else puts ("Nenalezeno!");
    return 0;
}
```

9-1-2 Kupec Bengálský — Pavel Machek

Cor Sàirovi to dalo dosti přemýšlení a jednu noc kvůli tomu dokonce nespal, ale výsledek stál za to. Když ve smluvený den přišel k Mer Chantovi, jenž se jen poťouchle usmíval, pravil: „Dobrá, ty bídný Mer Chante, přistupuju na tvé platební podmínky. Koule, které dostaneš, však budu vybírat já!“ Mer Chant ochotně souhlasil a dokonce se rozesmál nahlas. Plácli si tedy a vážení mohlo začít. O několik chvil později Cor Sàir spokojeně odcházel, drže si v podpaží kouli plnou zlata a Mer Chant jen tiše běsnil v koutku svého hokynářství. *(dle řešení Pavla Šedivého).*

A jak to Cor Sàir udělal?

Inu snadno: vzal nějakých šest koulí a položil je na váhy. Nemusel být génius na to, aby věděl, že na každou misku musí položit 3. Po ustálení vah už věděl, ve které trojici je zlatá koule:

Pokud byla jedna z misek těžší než druhá, byla zaručeně zlatá koule mezi těmi na první misce.

Hmm... a co když zůstaly váhy v rovnováze? To přeci znamená, že obě trojice jsou stejně těžké a že zlatá koule je mezi třemi koulemi, které dosud nebyly váženy.

Cor Sàir tedy věděl, ve které trojici je zlatá koule a také ve kterých šesti koulích zlatá určitě není – a třemi z nich zaplatil Mer Chantovi za první vážení.

Potom Cor Sàir vzal dvě ze tří „nadějných" koulí a položil je na misky vah. Po dalším vážení věděl (obdobně jako v předchozím vážení), která koule je zlatá a už z předchozího vážení měl 3 koule, jimiž Mer Chantovi zaplatil.

A Mer Chant nakonec zjistil, že i kdyby chtěl za každé vážení koule čtyři, i tak by jej Cor Sàir přelstil stejným způsobem...

9-1-3 Strašidlóóó — Martin Bělocký

Tuto úlohu všichni nějak vyřešili, až na pár nefunkčních programů. Nutno zdůraznit, že mnozí ztratili body za chybějící nebo nedostatečný popis algoritmu a hlavně za chybějící popis programu.

Největší problém byl ve složitosti algoritmu. Někteří napsali i tu nejjednodušší metodu, která byla zkusit každý bod matice ($těchje M \cdot N$) vzít jako levý horní roh hledaného obdélníka a vyzkoušet k němu všechny možné pravé dolní rohy (těch je přibližně $M \cdot N$) a vyzkoušet, zda obdélník s danými souřadnicemi je plný nul (přibližně $M \cdot N$ testů). Což dává dohromady přibližnou časovou složitost algoritmu $M^3 \cdot N^3$, která nikoho nemůže uspokojit.

Dle zadání se v obdélníku (budeme mu říkat matice) mohou vyskytovat nulové a nenulové hodnoty. Velikost nenulových hodnot nás nezajímá, proto ze všech nenulových hodnot uděláme nulové a ze všech nulových hodnot uděláme jedničky.

Abychom se nemuseli zabývat okraji, přidáme kolem celé matice pás z nul.

Dále si vytvoříme 2 pomocné matice (A, B) o stejné velikosti, a na místa jedniček napíšeme počet jedniček, které se nachází přímo pod danou jedničkou (v matici A) a nad danou jedničkou (matice B) (včetně).

Pův. matice (po úpravě 0/1)	A	B
	000000	000000
0111	001340	001110
1011	030230	010220
1111	022120	021330
1101	011010	032040
	000000	000000

Úvaha: Hledaná strana maximálního jedničkového obdélníku musí přiléhat buď k okraji matice nebo k nulovému prvku. Protože jsme si okolo celé matice vytvořili pás nul, můžeme prohlásit, že strana maximálního jedničkového

obdélníka přiléhá k nějakému nulovému prvku.

Nechť M^i_j je nějaký nenulový prvek v matici. Hledáme maximální jedničkový obdélník přiléhající levou stranou k prvku M^i_j. Postupujeme od prvku M^i_j směrem doprava (tj. po prvcích $M^i_{j+1}, M^i_{j+2} \ldots$), dokud nenarazíme opět na nulu. Ta zprava omezuje „nejdelší“ jedničkový obdélník přiléhající levou stranou k M^i_j. Pro kazdý prvek M^i_{j+b}, který cestou potkáme, určíme maximální jedničkový obdélník přiléhající levou stranou k M^i_j takový, že jeho pravá strana je ve sloupci $j + b$.

A jak takový obdélník vypadá? Jeho pravý i levý okraj je již omezen $(j \ldots j + b)$, zbývá tedy určit pouze jeho horní a dolní okraj. Hledaný dolní okraj je určen „hloubkami jedniček“ pod prvky M^i_{j+1}, M^i_{j+2}, … – je dán jejich minimem, tedy

$$H = \min(M^i_j, M^i_{j+1}, \ldots, M^i_{j+b}).$$

Stejně tak horní okraj je určen „výškami jedniček“ nad prvky M^i_{j+1}, M^i_{j+2}, … – je dán také jejich minimem. Tyto „hloubky“ a „výšky“ jedniček však již máme spočítány v maticích A a B. Tedy postupujeme od prvku M^i_j k prvku M^i_{j+b} a v každém kroku spočítáme velikost maximálního obdélníka přiléhajícího k M^i_j a s pravým okrajem ve sloupci $j + k$, kde $j < j + k \leq j + b$. Je-li spočítaná velikost větší, než aktuální maximální velikost, pak jsme našli dosud největší jedničkový obdélník.

Složitost a popis programu: Nejprve vytváříme matice A a B. Vytvoření znamená projít každý prvek matice a přičíst jedničku k hodnotě prvku nad (pod) ním. Tedy vytvoření jedné matice má čas. složitost $O(M \cdot N)$. Pro druhou matici totéž. Poté postupujeme postupně přes všechny prvky matice A a zároveň B a každý prvek testujeme na nulu (přiléhavá nula), pokud ano, jdeme po nenulových prvcích doprava až k další nule a pro každý nenulový prvek se provádí testování dalších nenulových prvků – napravo až do další nuly – a výpočet: test na doposud největší obsah. Hledání probíhá ve dvou vnořených for-cyklech. Výpočet uvnitř cyklu se provádí pouze pro nulové prvky a probíhá přes všechny nenulové prvky napravo. Nulových prvků je v matici nejvýše $N \cdot M$. Pro žádný prvek neprobíhá výpočet dvakrát. Tedy celková časová složitost je $O(M \cdot N)$. Pokud jste toto nepochopili, pak si zkuste rozmyslet, že by se dal vnitřní for-cyklus přepsat na while cyklus, aby se žádný prvek matice nemusel testovat dvakrát. Prostorová (paměťová) složitost programu je $O(M \cdot N)$. Matice B je v našem případě zbytečná, neboť bychom její prvky mohli zapisovat přímo do původní matice, která je v našem programu při výpočtu nevyužitá.

```
const MaxM=100; MaxN=100;

var Mesto,MatA,MatB:array[0..MaxM+1,0..MaxN+1] of integer;
    M,N:integer;
    i,j,b:integer;
```

```
    MinNad,MinPod:integer;
    Vel:integer;
    MaxVel:integer;
    Mi,Mj,Mk,Ml:integer;

begin
  {Sem patri vstup dat a prevod na jednicky a nuly, ale kazdy si jej jiste domysli sam...}

  {naplneni okraju nulami v matici Mesto }
  for i:=0 to M+1 do begin
    Mesto[0,i]:=0;
    Mesto[N+1,i]:=0;
  end;
  for i:=0 to N+1 do begin
    Mesto[i,0]:=0;
    Mesto[i,M+1]:=0;
  end;

  {vytvoreni matic MatA a MatB}
  for j:=1 to M do begin
    for i:=N+1 downto 1 do
      if Mesto[i,j]=0 then MatA[i,j]:=0
      else MatA[i,j]:=MatA[i+1,j]+1; {jednicka nad jednickami}
    for i:=1 to N do
      if Mesto[i,j]=0 then MatB[i,j]:=0
      else MatB[i,j]:=MatB[i-1,j]+1; {jednicka pod jednickami}
  end;

  {vyhledani nejvetsiho nuloveho obdelnika}
  MaxVel:=0;
  for i:=1 to N do
    for j:=0 to M-1 do
      if Mesto[i,j]=0 then begin
        { Projdi jednicky vpravo od Mesto[i,j] }
        b:=1;
        MinPod:=maxint;
        MinNad:=maxint;
        while Mesto[i,j+b]=1 do begin
          if MatA[i,j+b]<MinPod then MinPod:=MatA[i,j+b];
          if MatB[i,j+b]<MinNad then MinNad:=MatB[i,j+b];
          Vel:=b*(MinNad+MinPod-1);
          if MaxVel<Vel then begin
            MaxVel:=Vel;
            Mi:=i-MinNad+1;
            Mj:=j+1;
            Mk:=i+MinPod-1;
            Ml:=j+b;
          end;
          b:=b+1;
        end
      end;

  {tisk vysledku}
  if MaxVel=0 then writeln('Ve meste nestrasi !')
  else
    writeln('Maximalni strasidelny obdelnik je od [',Mi,',',Mj,'] do [',
            Mk,',',Ml,'] a ma velikost ',MaxVel,'.');
end.
```

9-1-4 !ǝɹǝɔɹǝʇ qoT — Jan Kotas

Jak v počítači reprezentovat velká čísla? Nejjednodušší je omezit shora délku velkých čísel na nejvýše např. sto míst a udržovat je v poli a konstantní délky n:

$$x = \sum_{i=0}^{n-1} z^i \cdot x_i$$

Je dobré zvolit soustavu, jejíž základ z je mocninou dvojky; v následujícím řešení je $z = 256$. Použití desítkové soustavy sice eliminuje problémy se vstupem a výstupem, ale rutiny pro počítání jsou pak složitější a pomalejší – je lepší optimalizovat rychlost počítání s čísly proti rychlosti vstupu a výstupu.

Počítání s velkými čísly se dělá přesně tak, jak se počítá na papíře (snad netřeba vysvětlovat znovu – viz učebnice matematiky pro nižší ročníky škol základních).

Časové složitosti operací ($n = \log N_{max}$ je délka největšího čísla N_{max}):

* sčítání, odčítání: $O(n)$
* násobení: $O(n^2)$, existuje i algoritmus se složitostí cca $O(n^{1.6})$.
* dělení $O(n^2)$, existuje též rychlejší algoritmus, ale velice složitý.

Nedostatky následujícího vzorového řešení:

- Pro číslo je alokován blok paměti konstantní velikosti, počítá se stále v plné přesnosti. Je otázka, zda dynamické alokování paměti je užitečné, protože je časově náročnější, navíc se většinou počítá s čísly zhruba stejné velikosti.
- Čísla se často kopírují jako blok paměti, přitom by stačilo nějak chytře předávat pouze ukazatele.
- Rutiny nemají jednotné volací konvence: kupříkladu `Add(A,A,A)` bude fungovat, kdežto `Mul(A,A,A)` nikoliv.
- Bylo by pěkné, kdyby se operace s velkými čísly zapisovaly stejně jako operace s normálními čísly: např. místo `Add(A,B,C)` psát `C := A + B`. V Pascalu to možné není, v `C++` je.
- Něco by bylo dobré kvůli rychlosti napsat v assembleru, to by ale nebylo zcela podle pravidel semináře!

```
unit BigInt;

interface

const MaxBytes = 100;
const MaxBits = 8 * MaxBytes;

type TBigInt = array [0..MaxBytes-1] of Byte;
```

```
procedure Init(var R: TBigInt; X: Byte); { R := X }
procedure Zero(var R: TBigInt); { R := 0 }

function IsZero(const A: TBigInt): Boolean; { A = 0 }

const Overflow: Boolean = False; { True = nastalo preteceni }

procedure Add(var R: TBigInt; const A,B: TBigInt);           { R := A + B }
function Sub(var R: TBigInt; const A,B: TBigInt): Boolean;   { R := A - B }
procedure Mul(var R: TBigInt; A: TBigInt; const B: TBigInt); { R := A * B }
procedure DivMod(var A: TBigInt; B: TBigInt; var R: TBigInt); { R := A div B; A := A mod B }

var Base: TBigInt; { zaklad pro prevod z a na retezce }

procedure StrToBig(const S: String; var A: TBigInt);
function BigToStr(A: TBigInt): String;

implementation

procedure Init(var R: TBigInt; X: Byte);
var I: Integer;
 begin
  R[0] := X;
  for I := 1 to MaxBytes-1 do R[I] := 0;
 end;

procedure Zero(var R: TBigInt);
begin Init(R,0) end;

function IsZero(const A: TBigInt): Boolean;
var I: Integer;
 begin
  IsZero := True;
  for I := 0 to MaxBytes-1 do if A[I] <> 0 then IsZero := False;
 end;

procedure SetBit(var A: TBigInt; I: Integer);
 begin A[I shr 3] := A[I shr 3] or (1 shl (I and 7)) end;

function IsBit(const A: TBigInt; I: Integer): Boolean;
 begin IsBit := A[I shr 3] and (1 shl (I and 7)) <> 0 end;

function Bits(const A: TBigInt): Integer;
var I: Integer;
 begin
  I := MaxBits;
  repeat
   I := I - 1;
   if IsBit(A,I) then Break;
  until I = 0;
  Bits := I + 1;
 end;

procedure Add(var R: TBigInt; const A,B: TBigInt);
var I, X, Carry: Integer;
 begin
  Carry := 0;
  for I := 0 to MaxBytes-1 do
```

```
   begin
    X := Integer(A[I]) + Integer(B[I]) + Carry;
    if X < 256 then begin R[I] := X; Carry := 0 end
    else begin R[I] := X - 256; Carry := 1 end;
   end;
  if Carry <> 0 then Overflow := True;
 end;

function Sub(var R: TBigInt; const A,B: TBigInt): Boolean;
var I, X, Carry: Integer;
 begin
  Carry := 0;
  for I := 0 to MaxBytes-1 do
   begin
    X := Integer(A[I]) - Integer(B[I]) - Carry;
    if X >= 0 then begin R[I] := X; Carry := 0 end
    else begin R[I] := X + 256; Carry := 1 end;
   end;
  Sub := Carry <> 0;
 end;

function ShiftRight(var A: TBigInt): Boolean; { A := A >> 1 }
var I, X, Carry: Integer;
 begin
  Carry := 0;
  for I := MaxBytes-1 downto 0 do
   begin
    X := A[I] + Carry;
    if odd(X) then Carry := 256 else Carry := 0;
    A[I] := X shr 1;
   end;
  ShiftRight := Carry <> 0;
 end;

procedure Mul(var R: TBigInt; A: TBigInt; const B: TBigInt);
var I: Integer;
 begin
  Zero(R);
  for I := 0 to Bits(B)-1 do
   begin
    if IsBit(B,I) then Add(R,R,A);
    Add(A,A,A);
   end;
 end;

procedure DivMod(var A: TBigInt; B: TBigInt; var R: TBigInt);
var I,N: Integer;
    E: TBigInt;
 begin
  N := Bits(B);
  if N = 0 then begin Overflow := True; Exit end;
  N := Bits(A)-N;
  Zero(R);
  for I := 1 to N do Add(B,B,B);
  for I := N downto 0 do
   begin
    if not Sub(E,A,B) then
      begin A := E; SetBit(R,I) end;
```

```
    ShiftRight(B);
   end;
 end;

procedure StrToBig(const S: String; var A: TBigInt);
var I: Integer;
    B: TBigInt;
 begin
  Zero(A);
  for I := 1 to Length(S) do
   begin
    Mul(A,A,Base);
    Init(B,ord(S[I])-ord('0'));
    Add(A,A,B);
   end;
 end;

function BigToStr(A: TBigInt): String;
var Result: String;
    B: TBigInt;
 begin
  if IsZero(A) then begin BigToStr := '0'; Exit end;
  Result := '';
  repeat
   DivMod(A,Base,B);
   Result := chr(A[0]+ord('0')) + Result;
   A := B;
  until IsZero(A);
  BigToStr := Result;
 end;

begin
 Init(Base, 10);
end.

{ Priklad pouziti unity }
uses BigInt;
var A,B,C: TBigInt;
begin
 StrToBig('698465618676954',A);
 StrToBig('787764568798645',B);
 Mul(A,A,B);
 StrToBig('3848624735242929240',B);
 Add(A,A,B);
 StrToBig('19765416174515758751',B);
 DivMod(A,B,C);
 Writeln(BigToStr(A));
 if BigInt.Overflow then writeln('Nastalo preteceni.');
end.
```

9-2-1 Psí funkce — Robert Šámal

Čísla $\Psi(n)$ jsou značně velká už pro malá n, proto rozhodně není možné řešit úlohu tak, že spočítáme hodnotu $\Psi(n)$ a vydělíme ji třinácti. Uděláme proto něco jiného – ukážeme, že pro $n > 3$ je $\Psi(n) \bmod 13 = 3$. Program pak bude zcela triviální.

Nejprve dokážeme, že zbytek mocniny dvou po dělení třinácti záleží jen na zbytku exponentu po dělení dvanácti. Platí totiž $2^{12} = 13 \cdot 315 + 1$, tudíž podle binomické věty platí

$$\begin{aligned}(2^{12})^m &= (13 \cdot 315 + 1)^m = (13 \cdot 315)^m + \\ &+ \binom{m}{1}(13 \cdot 315)^{m-1} + \ldots + \\ &+ \binom{m}{m-1}(13 \cdot 315)^1 + 1 = \\ &= 13 \cdot k + 1\end{aligned}$$

(k je přirozené číslo); proto je $2^{12 \cdot m + a} = (2^{12})^m \cdot 2^a = 13 \cdot k \cdot 2^a + 2^a$. Právě jsme ukázali, že $2^{12 \cdot m + a} \bmod 13 = 2^a \bmod 13$. Pro zjištění $\Psi(n) \bmod 13$ proto stačí zjistit $\Psi(n-1) \bmod 12$.

Pro $n > 3$ lze $\Psi(n-1)$ psát ve tvaru $2^{2k} = 4^k$, kde $k > 0$ je přirozené číslo ($\Psi(n-2)$ je totiž sudé). Takže (opět binomická věta):

$$\Psi(n-1) = 4^k = (3+1)^k = 3^k + \binom{k}{1}3^{k-1} + \ldots + 1.$$

Zbytek $\Psi(n-1)$ po dělení dvanácti je tedy jedno z čísel $1, 4, 7, 10$. Pokud je totiž $\Psi(n-1) = 12 \cdot a + b$, je (podle předminulé věty)

$$1 = \Psi(n-1) \bmod 3 = (3 \cdot (4 \cdot a) + b) \bmod 3 = b \bmod 3.$$

Ovšem pro $n > 3$ je $\Psi(n)$ dělitelné čtyřmi, proto je $\Psi(n-1) \bmod 12 = 4$.

Z toho, co jsme zjistili v předchozích dvou odstavcích, již přímo plyne, že pro $n > 3$ je $\Psi(n) \bmod 13 = 2^4 \bmod 13 = 3$, což jsme chtěli dokázat.

Ti, kteří za úlohu dostali (skoro) plný počet bodů, většinou postupovali podobně, jako právě skončené řešení. Častá **chyba** byla takováto: napíšu si tabulku čísel $2^n \bmod 13$, zjistím že $2^{12} \bmod 13 = 2^0 \bmod 13$, $2^{13} \bmod 13 = 2^1 \bmod 13$. Prohlásím, že je zřejmé, že se zbytky opakují s periodou dvanáct. Podobně „zjistím“, že čísla $2^n \bmod 12$ se (pro $n > 2$) opakují s periodou dva. Protože $\Psi(n-2)$ je sudé, znám zbytek $\Psi(n-1) = 2^{\Psi(n-2)}$ po dělení dvanácti a tudíž i zbytek $\Psi(n) = 2^{\Psi(n-1)}$ po dělení třinácti. V čem že je ta chyba? Pokud zkoumáme nějakou posloupnost čísel, je často vhodné napsat si tabulku prvních několik členů a hledat v ní nějaké zákonitosti. Proti tomu rozhodně nic nenamítám. Nicméně to, že nějaká zákonitost platí pro prvních několik (lhostejno zda 10, 100 nebo 10^{10}) členů, nedává žádnou záruku, že platí pro všechny členy. (Například číslo $2^{2^n} + 1$ je prvočíslo pro $n < 5$, ne však pro n větší.) Abychom dokázali něco pro všechny členy posloupnosti, je třeba buď vymyslet

nějaký důkaz (na vytvořené tabulce nezávislý), anebo zdůvodnit, proč by měla tabulka pro velká n vypadat stejně — v našem případě stačilo konstatovat, že $(2 \cdot a) \bmod m = 2 \cdot (a \bmod m) \bmod m$ a tedy každý řadek v tabulce zbytků závisí jen na předchozím řadku. Pokud je tedy k-tý řadek stejný jako l-tý, musí být i $(k+1)$-ní stejný jako $(l+1)$-ní, atd. Přidáním této úvahy se z výše uvedeného chybného řešení stane řešení správné.

9-2-2 Tiskařský šotek — Martin Mareš

Mnozí řešitelé úlohu pojali jakožto „jednorázovou“ – to jest jedno načtení dat a pak jeden dotaz a tvrdili, že jejich (naprosto triviální) řešení má časovou složitost přímo úměrnou délce slovníku a že to lépe nejde, poněvadž slovník musí alespoň jednou celý přečíst. Ostatní si naštěstí uvědomili, že takto by byla úloha „o ničem“ a vyložili si ji tím správným způsobem: jednou analýza dat a poté mnoho dotazů, přičemž záleží na časové složitosti vyhodnocení jednoho dotazu.

Předem se omlouvám všem zůčastněným za poněkud nemírné využívání symbolu $O(f)$ v celém následujícím textu. Bez této symboliky by však byly úvahy níže ještě nepřehlednější.

Asi bude nejlepší nejprve zjistit, kolik vlastně může existovat modifikací daného slova délky w (v celém dalším textu předpokládáme, že celý slovník obsahuje slova mající *v průměru* tuto délku – usnadní to odhady časové složitosti a na obecnosti to neubírá). Tedy existuje $w \cdot (a-1)$ možností, jak písmeno změnit (a je velikost abecedy), w možností, jak písmeno ubrat, a konečně $(w+1) \cdot a$ možností, jak nějaké přidat (pozn.: nikdo netvrdí, že všechny tyto možnosti dají různé výsledky – existují případy, kdy ne, ale nás zajímá případ průměrný, v němž opravdu budou různé – zkuste si to rozmyslet podrobněji). Tedy celkem $O(w \cdot a)$ variant, každá délky $O(w)$.

Porovnávat všechny varianty zadaného slova se všemi slovy ve slovníku nejspíše není rozumné – porovnání dvou slov trvá $O(w)$, pro celý slovník tedy $O(w \cdot N)$ (N je počet slov), přičemž těchto porovnání potřebujeme provést $O(w \cdot a)$. Dostaneme tedy výslednou časovou složitost $O(w^2 \cdot a \cdot N)$, což asi nebude to pravé.

O něco lepší by bylo rozdělit si slovník na několik menších podle délek slov a porovnávat pouze s těmi slovy, která mají danou délku. Tedy $O(N/w)$ porovnání $\Rightarrow$ celkem $O(w \cdot a \cdot N)$. Sice lepší, ale stále ještě nic moc.

Bystrý řešitel si ovšem na tomto místě řekne: „To nápadně připomíná hashování!“ A má pravdu... Proč jej tedy nezkusit? Nerozdělíme si slovník do $O(w)$ příhrádek podle délek slov, ale rovnou na N částí – sestrojíme nějakou funkci f, jež každému slovu přiřadí nějaké přirozené číslo od 0 do $N-1$, a podle těchto hodnot slova roztřídíme. Takto bude v každé příhrádce *průměrně* jedno slovo a jestliže budou hodnoty funkce f rozděleny dostatečně rovnoměrně,

dokonce tam bude typicky jedno nebo dvě slova.

Jestliže tedy chceme hledat nějaké slovo, spočteme si pro něj hodnotu funkce f (řekněme, že najdeme takovou, která bude spočtena v čase $O(w)$ – jednu najdete ve vzorovém programu) a budeme hledat jenom v příslušné příhrádce, ve které bude průměrně $O(1)$ slov. Každé porovnání slova s hledaným nás stojí $O(w)$, takže celkem slovo nalezneme v čase $O(w)+O(1)\cdot O(w) = O(w)$. Celkem zkoušíme $O(w \cdot a)$ variant, což nám tedy trvá $O(w^2 \cdot a)$. Lépe to jistě nejde, poněvadž v lepším čase bychom nemuseli zvládnout ani všechna řešení vypsat (každá z variant může být řešením délky $O(w)$). Všimněte si, že časová složitost hledání (počítaná samozřejmě od začátku do konce pro *průměrný* případ) vůbec nezávisí na N.

Následující vzorový program postupuje přesně dle popsaného algoritmu, přičemž obsah každé příhradky uchovává jako spojový seznam. Zbytek by již měl být zřejmý.

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

int N;                                  /* Počet slov */
struct slovo {                          /* Jedno slovo */
    struct slovo *next;
    char text[1];
};

struct slovo **p;                       /* Příhrádky (pointery na první slova v nich) */

unsigned int f (unsigned char *x)       /* Rozdělovací funkce */
{
    unsigned int z = strlen (x);
    while (*x)
        z = z*259309 + *x++;
    return z % N;
}

void cti_slovnik (void)
{
    int i, h;
    char x[256];
    struct slovo *s;

    scanf ("%d\n", &N);                 /* Počet slov */
    p = calloc (sizeof (struct slovo *), N);    /* Alokujeme paměť */
    for (i=0; i<N; i++)
      {
        gets (x);                       /* Zařazujeme další slovo */
        s = malloc (sizeof (struct slovo) + strlen (x));
        h = f (x);
        s->next = p[h];
        p[h] = s;
        strcpy (s->text, x);
```

```
    }
}
void test (char *x)                    /* Otestuje a případně vypíše slovo */
{
  struct slovo *s = p[f (x)];
  while (s)
    {
      if (!strcmp (s->text, x))
        {
          puts (x);
          return;
        }
      s = s->next;
    }
}
void dotazy (void)                     /* Odpovídá na dotaz */
{
  char x[256], y[256];
  int i, j, g;
  while (gets (x))                     /* Sem se vstupem, šotku! */
    {
      g = strlen (x);
      for (i=0; i<g; i++)              /* Zkoušíme výměny */
        for (j='a'; j<='z'; j++)
          if (x[i] != j)
            {
              strcpy (y, x);
              y[i] = j;
              test (y);
            }
      for (i=0; i<g; i++)              /* Zkoušíme výpustky */
        {
          strncpy (y, x, i);
          strcpy (y+i, x+i+1);
          test (y);
        }
      for (i=0; i<=g; i++)             /* Zkoušíme vsuvky */
        for (j='a'; j<='z'; j++)
          {
            strncpy (y, x, i);
            y[i] = j;
            strcpy (y+i+1, x+i);
            test (y);
          }
      puts ("That's all, folks!");
    }
}
void main (void)
```

```
{
    cti_slovnik ();
    dotazy ();
}
```

9-2-3 Překla-datel — Pavel Machek

Takže co s ufounskými daty? Podle mne byla úloha docela jednoduchá, ale... Sešla se spousta řešení, povětšinou pomalých (384 kroků), některá dokonce *opravdu* pomalá (512 kroků). Objevily se též nějaké nápady, jak pomocí různých triků zrychlit výpočet dvakrát, ale to stále ještě není ono.

Nejlepší (nám známé) řešení má 33 kroků. Využívá zřejmého faktu, že otočit $2n$-bitové číslo je totéž jako navzájem prohodit obě jeho n-bitové poloviny a pak každou z nich otočit zvlášť. Nejprve tedy prohodíme 2 64-bitové bloky, pak vždy 2 sousední 32-bitové (uvědomte si, že to je možno učinit najednou pro obě dvojice bloků! – viz text programu), ..., v posledním kroku pak sousední dvojice bitů. Pro prohazování sousedních bloků dané velikosti potřebujeme 5 instrukcí, v prvním prohazování se dají dvě z těchto instrukcí ušetřit.

```
a = x < 64
b = x > 64
x = a | b

a = x < 32
b = x > 32
a = a & 0xFFFFFFFF00000000FFFFFFFF00000000
b = b & 0x00000000FFFFFFFF00000000FFFFFFFF
x = a | b

a = x < 16
b = x > 16
a = a & 0xFFFF0000FFFF0000FFFF0000FFFF0000
b = b & 0x0000FFFF0000FFFF0000FFFF0000FFFF
x = a | b

a = x < 8
b = x > 8
a = a & 0xFF00FF00FF00FF00FF00FF00FF00FF00
b = b & 0x00FF00FF00FF00FF00FF00FF00FF00FF
x = a | b

a = x < 4
b = x > 4
a = a & 0xF0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0
b = b & 0x0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F
x = a | b

a = x < 2
b = x > 2
a = a & 0xCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
b = b & 0x33333333333333333333333333333333
x = a | b
```

```
a = x < 1
b = x > 1
a = a & 0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
b = b & 0x55555555555555555555555555555555
y = a | b
```

9-2-4 Nesčetní námětové — Martin Bělocký & Martin Mareš

Došlá řešení byla velice různorodá. Základní potíž spočívala v tom, že nebylo zcela jasné, zda se veškerá data mají šanci vejít do paměti a zda jsou náměty číslovány postupně či „děravě". Z toho pak vznikaly následující varianty úlohy (korektní řešení libovolné varianty bylo hodnoceno plným počtem bodů):

- Náměty jsou číslovány postupně: stačí jednoduché pole počítadel a zbytek je, doufejme, zcela jasný.
- Náměty nejsou číslovány postupně:
  - Data se nevejdou do paměti: nutno použít stromy nebo hashování a tak si zorganizovat počítadla pro jednotlivé náměty i přes jejich roztroušenost.
  - Data se vejdou do paměti: toto je pravděpodobně jediná kombinace, u které vznikne opravdu zajímavá situace. Problém se redukuje na úlohu typu „máte pole a najděte v něm hodnotu, která se vyskytuje nejvícekrát."

Naše vzorové řešení se bude zabývat třetí variantou úlohy a bude jakousi variací na klasické téma QuickSort.

Vybereme si v poli libovolnou hodnotu z (pokud možno někde „uprostřed" pokrytého rozsahu hodnot, ale to se nedá snadno zaručit) a rozdělíme všechny prvky pole na dvě části: v jedné (tu soustředíme v oblasti s indexy $1, \ldots, k$) budou prvky $x_i \leq z$, ve druhé (s indexy $k+1, \ldots, N$) pak prvky zbylé ($x_j > z$). Jesliže je libovolná část menší než $N/2$, hledané číslo v ní zaručeně nemůže být. Takto nám zbyde buďto jedna nebo dokonce žádná (v případě $2 \cdot z = N$) část ono číslo obsahující a tu můžeme zpracovat identickým způsobem, dokud nezjistíme, že obsahuje jen jediné číslo, což je číslo hledané.

Číslo z budeme určovat v jakémsi „předprůchodu" jako aritmetický průměr všech x_i a při tom ještě budeme testovat, obsahuje-li pole více než jednu hodnotu. (Samozřejmě by to šlo dělat i jinak – například prostým vybráním $x_{N/2}$.)

Časová složitost (podobně jako u QuickSortu) může v nejhorším případě být až $O(N^2)$, ale tento případ je velice atypický. V případě průměrném se dostaneme na $O(N)$. Důkaz tohoto tvrzení zde nebudeme uvádět, jelikož není příliš snadný, uvedeme ovšem „intuitivní důkaz", proč by tomu tak mělo být: předpokládejme, že zvolíme hodnotu z tak, že k bude „někde kolem poloviny velikosti úseku" – takto po prvním průchodu zbude jeden úsek velikosti o kousek

větší než $N/2$ a ve druhém průchodu buď číslo odhalíme nebo rozdělíme na dva úseky velikosti $\approx N/4$, čímž hledání skončí s negativním výsledkem.

```
int vitez (int *x, int N)                 /* Vrací vítěze nebo −1, není-li */
{
  int l=0, r=N−1;                         /* Levý a pravý okraj zpracovávaného úseku */
  int m=N/2;                              /* Mez výhry */
  int z;                                  /* Hodnota, podle níž rozdělujeme */
  int i, j, k;                            /* Pomocné proměnné */

  while (r−l >= m)                        /* Dokud je ještě šance */
    {
      z = 0;                              /* Předprůchod */
      j = x[l];
      for (i=l; i<=r; i++)
        {
          z += x[i];
          if (x[i] != j) j = −1;
        }
      if (j >= 0)                         /* Všechny stejné ⇒ vítěz. */
        return j;
      z /= r−l+1;                         /* z je aritmetický průměr */
      i = l;                              /* A rozdělujeme... */
      j = r;
      while (i < j)
        {
          while (i <= j && x[i] <= z)
            i++;
          while (i <= j && x[j] > z)
            j−−;
          if (i < j)
            { k=x[i]; x[i]=x[j]; x[j]=k; }
        }
      if (j−l > r−i)                      /* Pokračujeme větším z úseků */
        r = j;                            /* ... levým */
      else
        l = i;                            /* ... pravým */
    }
  return −1;                              /* Smůla... */
}
```

9-2-5 Top secret! — Jan Kotas

Ze zadání vyplývá, že máme spočítat $x^s \bmod N$. Jelikož je algoritmus velmi jednoduchý, je úmyslně zapsán hodně formálně.

Budeme postupně počítat hodnoty $d_1 = \left(x^1 \bmod N\right)^2 = x^2 \bmod N$, $d_2 = \left(x^2 \bmod N\right)^2 = x^4 \bmod N$, ... Každou další hodnotu můžeme vypočítat z té

předchozí pomocí jednoho násobení (umocnění na druhou) a jednoho dělení (operace modulo):

$$\begin{aligned} d_0 &= x \\ d_{i+1} &= x^{2^{i+1}} \bmod N = (x^{2^i} \bmod N)^2 \bmod N = d_i^2 \bmod N \end{aligned}$$

Na exponent s se budeme dívat jako na číslo zapsané ve dvojové soustavě:

$$s = \sum_{i=0}^{z} 2^i s_i.$$

Pro každý jedničkový bit s_i přinásobíme k výsledku hodnotu d_i:

$$\begin{aligned} x^s \bmod N &= x^{\sum_{i=0}^{z} 2^i s_i} \bmod N = \\ &= \left(\prod_{i=0}^{z} s_i x^{2^i} \right) \bmod N = \left(\prod_{i=0}^{z} s_i d_i \right) \bmod N. \end{aligned}$$

Jelikož platí následující rovnost, bude vysledek průběžně ořezávat operací modulo, abychom nepočítali se zbytečně velkými čísly:

$$(a \cdot b) \bmod N = ((a \bmod N) \cdot (b \bmod N)) \bmod N.$$

Při výpočtu $x^s \bmod N$ podle uvedeného algoritmu spotřebujeme nejvýše $2 \log_2 s$ násobení a dělení čísel velkých nejvýše N^2. Z tabulky časových složitostí uvedené v řešení minulé série plyne, že časová složitost algoritmu je $O(\log s \log^2 N)$, paměťové nároky algoritmu jsou úměrné velikosti vstupu – $O(\log s + \log N)$.

Zašifrované číslo bylo 774778151857376825989681138 63. Čísla 0 a 1 není dobré šifrovat, protože se šifrováním nemění – tvoří tzv. pevný bod.

Vzorové řešení je v Pascalu a používá procedury a funkce z minulé série. Původně zde mělo být také řešení v `C++`. Ukázalo se ale, že je příliš dlouhé na to, aby jej bylo možno otisknout. Zmiňme se tedy jen o všeobecně použitelných myšlenkách, které v něm byly použity:

- `C++` umožňuje definovat operátory i pro uživatelem definované typy. Výpočty s velkými čísly je pak možné zapisovat tak, jak jsme zvyklí – není je třeba překládat na méně přehledné volání funkcí.
- Při výpočtech s velkými čísly je třeba velmi často jedno číslo přiřadit jinému (zkopírovat). (V Pascalu se toto zkopírování děje neviditelně jako důsledek předání proměnné hodnotou.) Je-li tato operace implementována jako zkopírování kusu paměti, což je pomalé, může běh

programu značně brzdit. Jedním z možných řešení tohoto problému je technika, jež se nazývá počítání odkazů *(reference counting)*. Objekt, nazvěme ho `BigInt`, reprezentující číslo, obsahuje pouze ukazatel na objekt `BigCore` obsahující jeho hodnotu. U tohoto objektu je pak kromě hodnoty čísla uložen také počet objektů `BigInt`, které na něj ukazují *(reference count)*. Při přiřazení se pouze zvýší počítadlo v `BigCore`, který je přiřazován, a pak sníží počítadlo u `BigCore` s původní hodnotou. Pokud toto počítadlo dosáhne nuly, je objekt zrušen.

```
procedure PowMod(var R:TBigInt; A,B,C:TBigInt); { R = A^B mod C }
var E, T: TBigInt;
       I: Integer;
begin
   Init(R,1);
   for I := 0 to Bites(B)-1 do
      begin
         if IsBit(B,I) then
            begin
               Mul(E, R, A);
               DivMod(E, C, T);
               R := E;
            end;
         Mul(E, A, A);
         DivMod(E, C, T);
         A := E;
       end;
end;
```

9-3-1 Odporné odpory pana Odporného — Martin Bělocký

Tento příklad byl značně nepraktický. Jeho řešení je totiž v samotném důsledku k ničemu. Spojováním resistorů vznikají často neceločíslené výsledky (racionální čísla) a ty lze sotva dle zadání zadat na vstupu. Nehledě na to, že odpory skutečných resistorů nejsou přesné. Tedy člověk by v praxi spíše chtěl sestavit složení resistorů, které má velikost odporu v nějakém malém intervalu kolem požadované hodnoty (velikost tohoto intervalu charakterizuje požadovanou přesnost). Naše zadání však bylo jiné. Naneštěstí složitost tohoto problému je natolik veliká, že přesný výpočet výsledku zabere tolik času, že už pro malý počet resistorů (řádově desítky) se řešení, v případě že zapojení neexistuje, nedočkáme. Pro lidi libující si v počítání složitosti je následující odstavec.

Pouhé hledání zapojení ze sériových kombinací je NP-úplný problém – úlohu je možno triviálním polynomiálním převodem transformovat na známý NP-úplný problém batohu. Z tohoto plyne, že zatím nikdo nezná řešení této podúlohy v polynomiálním čase. Naše zapojení může však navíc mít i paralelní spoje... Ptáte se tedy jistě, jak se tedy takovéto úlohy v praxi řeší? Inu vězte,

že existují složitější algoritmy, které v průměrném případě řeší např. problém batohu v čase $O(N^3)$, ale ty přesahují obtížnost našeho semináře. Můžete se o nich dozvědět na některých přednáškách na MFF UK.

Algoritmus našeho řešení je prostý — *backtracking*, tj. zkoušet všechny možnosti a nezkoumat zbytečné a nesmyslné případy, (tzv. ořezávaný backtracking. Na začátku máme k dispozici pouze pole nepoužitých odporů zadaných velikostí. Z pole vybíráme nepoužité odpory a zkoušíme je spojovat nejprve sériově a pak paralelně. Každým spojením označíme původní 2 odpory za použité a nový odpor přidáme na konec pole. Poté zavoláme rekurzivně algoritmus znovu. Algoritmus končí s dalším rekurzívním voláním, pokud nalezne v poli odpor hledané velikosti. Je-li nalezená velikost navíc s doposud nejmenším počtem resistorů, zapojení si program zapamatuje. Pokud se projde celé pole odporů a všechny odpory jsou použity, nebo už nelze dosáhnout výsledného odporu (současné zapojení se sériovým zapojením zbylých resistorů má menší odpor než je zadaný výsledný odpor).

Pokud celý algoritmus skončí a zapamatoval si nějakou kombinaci odporů, pak tato kombinace obsahuje určitě nejmenší počet resistorů – prošly se všechny možnosti a žádná nebyla menší. Výsledné zapojení se pak vypíše ve formě aritmetického výrazu, který se v průběhu algoritmu mění dle testovaného zapojení. Resistory odpovídají číslům, '`+`' je sériové zapojení, '`*`' je paralelní zapojení, ve výrazu jsou použity závorky. Tedy náš program vlastně hledá vhodné uzávorkovaní a doplnění operátorů `+`, `*` v aritmetickém výrazu. Najde-li řešení, vytiskne ho.

Správnost je zajištěna probíráním všech možností. Každé uzávorkování výrazu odpovídá jednoznačně jednomu zapojení. Např. '`(50+1)*7`' je paralelní zapojení '`7`' a seriového '`50,1`'. Pro zvýšení rychlosti začíná algoritmus s nejmenším počtem rezistorů – spojuje dvojičku. Tu pak spojí s jiným odporem. Najde-li první řešení, prohlásí jej za mimimální. Dalších řešení výrazu s větším počtem operátorů, tj. resistorů, si nevšímá. Nepočítá zbytečně kombinace odporů stejné velikosti, které už se dříve vyskytly. Mám-li 3 resistory, každý s odporem 5, je zbytečné po jednom každý z nich připojovat k jinému resistoru, dostanu totiž pokaždé stejný výsledek.

Popis programu: Program je natolik dobře komentován, že je zbytečné zde dále něco popisovat. Jen tolik, že je napsán v jazyce C a veškerou práci dělá rekurzivní procedura `zapoj`.

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

#define MaxOdp 100                  /* Velikost pole na odpory a jejich spojení */
#define MaxStr 300                  /* Maximální délka zápisu zapojení */

#define Seriove 0                   /* Odpor vznikl sériovým zapojením */
#define Paralelni 1                 /* Odpor vznikl paralelním zapojením */
```

```
#define Dan 2                          /* Odpor byl dán */

struct Odpor {                         /* Údaje o jednom typu resistoru */
       float Vel;                      /* Velikost odporu */
       char Kolik;                     /* Počet odporů dané velikosti */
       char Pouz;                      /* Počet použitých resistorů */
       int Prvni;                      /* 1. resistor, který má cenu zkoušet spojovat */
       int PocR;                       /* Počet resistorů potřebných k vytvoření tohoto
                                          odporu */
       int A, B;                       /* Čísla resistorů, z nichž odpor vznikl */
       char Typ;                       /* Typ zapojení */
};

struct Odpor Odpory[MaxOdp];           /* Odpory, kt. je možno poskládat */
int Poc;                               /* Počet odporů */
float Odp;                             /* Cílový odpor */
int MinOdp;                            /* Min. počet resistorů potřebný k postavení výsl.
                                          odporu */
char Zapojeni[MaxStr];                 /* Textové schéma výsl. zapojeni */
int ZapPos;                            /* Pozice v předchozím poli */
float MaxR;                            /* Max. odpor, který je možno sestavit */

int Nacti (struct Odpor *Odpory)
{                                      /* Načte počáteční sadu odporů */
   int i;
   int PocVel = 1;                     /* Počet počátečních velikostí odporů */
   float R = 1;                        /* Velikost odporu */

   printf ("Zadávej velikosti odporů resistorů, zadávání ukonči nulou...\nOdpor:");
   scanf ("%f", &(Odpory[0].Vel));
   if (!Odpory[0].Vel)
      return 0;                        /* Žádný odpor nebyl zadán */
   Odpory[0].Kolik = 1;
   Odpory[0].Pouz = 0;
   Odpory[0].PocR = 1;
   Odpory[0].Typ = Dan;
   Odpory[0].Prvni = 0;
   Odpory[0].A = 0;
   Odpory[0].B = 0;
   MaxR = Odpory[0].Vel;

   while (R)
      {                                /* Načítej do zadání 0 */
         printf ("Odpor :");
         scanf ("%f", &R);
         for (i = 0; i < PocVel && Odpory[i].Vel != R; i++);
                                       /* Najdi odpor stejne velikosti */
         if (i == PocVel)
            {                          /* Nebyl nalezen? */
               Odpory[PocVel].Vel = R;    /* Založí novou velikost odporu */
               Odpory[PocVel].Kolik = 1;
               Odpory[PocVel].Pouz = 0;
               Odpory[PocVel].PocR = 1;
               Odpory[PocVel].Prvni = 0;
```

```
            Odpory[PocVel].A = 0;
            Odpory[PocVel].B = 0;
            Odpory[PocVel++].Typ = Dan;
          }
        else
          Odpory[i].Kolik++;    /* Zvýší počet odporů dané velikosti */
        MaxR += R;
      }
    return PocVel - 1;
}
void UlozZapoj (int Od)
{                               /* Uložení zapojení odporu Od */
    if (Odpory[Od].Typ == Dan)
      {                         /* Byl odpor dán? */
        gcvt (Odpory[Od].Vel, 8, &Zapojeni[ZapPos]);    /* Velikost odporu */
        for (; Zapojeni[ZapPos]; ZapPos++);    /* Nalezne konec řetězce */
      }
    else
      {                         /* Uloží zapojení obou částí – pro jistotu uzávorkuje */
        Zapojeni[ZapPos++] = '(';
        UlozZapoj (Odpory[Od].A);
        if (Odpory[Od].Typ == Seriove)
          Zapojeni[ZapPos++] = '+';
        else
          Zapojeni[ZapPos++] = '*';
        UlozZapoj (Odpory[Od].B);
        Zapojeni[ZapPos++] = ')';
      }
}
void Zapoj (int Pos)
{                               /* Provede všechna zapojení odporu Pos */
    int i;                      /* Odpor, se kterým se spojím */
    int OPrvni;                 /* Původní odpor ke spojení */
    while (Odpory[Pos].Pouz == Odpory[Pos].Kolik)
      Pos++;                    /* 1. nepoužitý odpor */
    if (Odpory[Pos].Vel == Odp && Odpory[Pos].PocR < MinOdp)
      {     /* Dosáhl jsem cíloveho odporu použitím doposud nejméně resistorů */
        ZapPos = 0;
        UlozZapoj (Pos);    /* Uloží zapojení */
        Zapojeni[ZapPos] = '\0';
        MinOdp = Odpory[Pos].PocR;    /* Uschová nový minimální počet */
      }
    if (Pos + 1 < Poc)
      Zapoj (Pos + 1);          /* Ještě další odpor ⇒ nejdříve se Pos nezapojí */
    Odpory[Pos].Pouz++;         /* Tento odpor bude použit */
    Odpory[Poc++].A = Pos;      /* Nový odpor vznikne z tohoto a jiného */
    OPrvni = Odpory[Pos].Prvni;    /* Pouhé uchování pro pozdější obnovení */
    for (i = Odpory[Pos].Prvni; i < Poc - 1; i++)
      {                         /* Najdi všechny odpory, s nimiž lze Pos spojit */
        if (Odpory[i].Pouz < Odpory[i].Kolik &&
```

```
            (i >= Pos || (i < Odpory[Pos].A && i < Odpory[Pos].B)))
        {
          Odpory[Pos].Prvni = i + 1;
                /* S odporem dané velikosti jsem už prozkoušel vše až do První */
          Odpory[i].Pouz++;    /* Odpor použiji k zapojení */
          Odpory[Poc - 1].Pouz = 0;    /* Nový odpor ještě nepoužit */
          Odpory[Poc - 1].Kolik = 1;    /* Vytvoří se pouze jeden odpor */
          Odpory[Poc - 1].PocR = Odpory[Pos].PocR + Odpory[i].PocR;
                                        /* Potřebný počet resistorů */
          Odpory[Poc - 1].B = i;    /* druhy odpor zapojeni je i */
          Odpory[Poc - 1].Typ = Seriove;    /* Typ zapojení je nejprve sériové */
          Odpory[Poc - 1].Vel = Odpory[Pos].Vel + Odpory[i].Vel;
                                        /* Výsledný odpor zapojení */
          Zapoj (Pos);    /* Vzhůru na další odpory, ještě nejsme u cíle */
          Odpory[Poc - 1].Typ = Paralelni;    /* Zkus paralelní zapojení */
          Odpory[Poc - 1].Vel = Odpory[Pos].Vel *
            Odpory[i].Vel / (Odpory[Pos].Vel + Odpory[i].Vel);
          MaxR = MaxR - Odpory[Pos].Vel - Odpory[i].Vel + Odpory[Poc - 1].Vel;
                                        /* Max. odpor, kterého ještě lze dosáhnout */
          if (MaxR >= Odp)
            Zapoj (Pos);    /* Je-li šance dosáhnout výsledku, zkoušej dále */
          MaxR = MaxR + Odpory[Pos].Vel + Odpory[i].Vel - Odpory[Poc - 1].Vel;
                                        /* Obnov MaxR */
          Odpory[i].Pouz--;    /* Odpor už nebude potřeba */
        }
      }
    Odpory[Pos].Pouz--;    /* Tento odpor nebude v zapojení */
    Poc--;                 /* Vytvářený odpor už byl všude v kombinaci vyzkoušen */
    Odpory[Pos].Prvni = OPrvni;    /* Obnov původního prvního */
}
int main (void)
{
    Poc = Nacti (Odpory);
    printf ("Cílový odpor:␣");
    scanf ("%f", &Odp);
    MinOdp = MaxOdp;
    Zapoj (0);
    if (MinOdp < MaxOdp)
      printf ("Zapojení: %s\n", Zapojeni);
    else
      printf ("Zapojení neexistuje. \n");
    return 0;
}
```

9-3-2 Cosi prohnilého? Robert Šámal

Většina účastníků řešila úlohu tak, že postupně zkusili smazat každou hranu a poté otestovat, zda se království rozpadne. Tento přímočarý přístup nevede k příliš rychlému algoritmu, ale budiž. Horší je však to, jak někteří realizovali

test souvislosti; místo optimálních $O(c)$ (c je počet hran) napsali test v čase $O(c \cdot m)$ (m značí počet měst), pár expertů dokonce v exponenciálním čase!

Ti zkušenější výše uvedený triviální postup vylepšili: když zjistím, že se po vymazání silnice království nerozpadlo, tak jsem zároveň našel nějaký cyklus silnic. Žádná ze silnic na cyklu není kritická, nebudu ji proto už zkoumat. Několik světlých vyjímek použilo algoritmus s optimální rychlostí...

A nyní už k vlastnímu řešení. Je snadné si uvědomit, že kritické silnice (tzv. mosty, řečeno terminologií teorie grafů) jsou přesně ty, které nejsou součástí žádného cyklu. Jde tedy o to, jak efektivně poznat, které hrany jsou součástí nějakého cyklu. Graf budeme rekurzivně procházet, půjde o upravené procházení do hloubky (upravené proto, že v klasickém případě neprocházíme obecný graf, nýbrž strom). Při procházení vrcholy průběžně číslujeme (čísly 1, 2, ...), číslo vrcholu u značme p_u. Klíčem k řešení je návratová hodnota funkce $h(l, u)$, která realizuje prohledávání (u je aktuální vrchol, l jeho předchůdce – vrchol, ze kterého jsme se do u dostali). Jedná se o nejmenší číslo vrcholu, na něž narazíme při procházení grafu na *hlubší* úrovni než je u. Čili je to minimum z čísel už projitých sousedů, nepočítaje v to souseda l (na tyto vrcholy narazíme ihned) a z návratových hodnot rekurzivního vyvolání funkce $h(u, w)$ pro všechny w – dosud neprojité sousedy u.

Pokud jste tento popis vstřebali, tak zbývá jen nahlédnout (a dokázat!), že hrana (u, w) je součástí nějakého cyklu právě tehdy, když při zavolání $h(u, w)$ z vrcholu u dostaneme číslo $\leq p_u$. Obcházíme-li totiž nějaký takový cyklus směrem $u \to \mathtt{w} \to \mathtt{u}$, pak jednou (nejpozději v posledním kroku) narazíme na vrchol s číslem $\leq p_u$.

Naopak, nechť při procházení pod vrcholem w narazíme na vrchol v takový, že $p_v \leq p_u$. To ovšem znamená, že mezi u a v vede cesta (zde využijeme toho, že rekurzivně nezkoumáme přechůdce, tj. vrchol l. Jinak bychom mohli dostat *cestu* $u \to l \to u$) tvaru $u \to w \to$ případné další vrcholy s číslem $> p_u \to v$ (na v jsme narazili *pod* vrcholem w) a také cesta používající vrcholy s čísly $\leq p_u$ (při procházení grafu jsme někdy byli ve v, pak jsme (souvisle) přešli do u, a teprve pak jsme začali používat čisla větší než p_u). Máme tedy cyklus obsahující hranu (u, w).

Takže z vrcholu u zavoláme rekurzivně funkci postupně na všechny dosud neprojité sousedy w a je-li návratová hodnota příliš vysoká, tak hranu (u, w) vypíšeme, je totiž kritická.

Ještě krátce k programu. Graf (město) je reprezentován tak, že pro každý vrchol máme uložen seznam sousedů (v poli `sou`) a jejich pocet (v poli `st`, jakožto stupeň). Zbytek by měl být jasný.

A jaká že je složitost algoritmu? Každou hranu zkoumáme dvakrát (z obou na ní ležících vrcholů), na každé město pouštíme jednou funkci `hledej(...)`. Čili časová složitost je $O(c + m)$.

```
#include <stdio.h>

#define MV 30                                   /* Maximální počet vrcholů */
#define ME 450                                  /* Maximální počet hran */

int st[MV];                                     /* Stupeň vrcholu */
int sou[MV][MV];                                /* Seznam sousedů */
int p[MV];                                      /* Pořadí při procházení */
int v;                                          /* Skutečný počet vrcholů */
int poc=0;                                      /* Počitadlo pro číslování */

int hledej (int l, int u)                       /* Zde se hledají mosty... */
{
    int i, w, pom, r;

    r = p[u] = ++poc;                           /* Očíslujeme vrchol */
    for (i=0; i<st[u]; i++)                     /* A zkoumáme sousedy... */
        if (!p[w=sou[u][i]]) {                  /* Ve w jsme ještě nebyli */
            pom = hledej (u, w);
            if (pom > p[u]) printf ("(%d, %d)\n", u, w);
                                                /* Most */
            else if (pom < r) r = pom;          /* Menší? */
        } else if (p[w] < r && w != l)
            r = p[w];                           /* Menší? */
    return r;
}

int main (void)
{
    int i, j;

    scanf ("%d", &v);
    while (scanf ("␣(%d, %d)", &i, &j), i || j) {
        sou[i][st[i]++] = j; sou[j][st[j]++] = i;
    }
    hledej (v, 0);
    return 0;
}
```

9-3-3 Mikroassembler — Martin Mareš

Tato úloha byla mírně neobvyklá v tom, že vlastně ani nešlo o vymýšlení nějakých algoritmů, ale pouze o důkazy, že něco je či není možné. Samozřejmě důkaz existence je možno (ale nikoliv nutno) provést konstrukcí hledaného objektu (což se v tomto vzorovém řešení ostatně činí), leč důkaz neexistence musí být poněkud pečlivější a lépe rozmyšlený, aby se v něm nepředpokládaly věci, které nemusí být pravda (běžnou chybou např. bylo tvrdit, že „instrukce `JEQ` je nepostradatelná, protože bez ní by nešly implementovat cykly“ bez jediného slova o tom, proč se příslušné programy nedají převést na jiné bez cyklů).

Nejprve zkusíme dokázat, bez kterých instrukcí se zaručeně obejít nelze:

- `JEQ` – jistě uznáte, že na původním μ_1 existují jak konečné, tak nekonečné programy. Ty nekonečné obsahují nekonečný cyklus, ty konečné musí dle definice končit skokem před začátek programu. A na obojí potřebujeme právě tuto instrukci, jelikož bez ní skákati nemožno. [I kdybychom odhlédli od těchto omezení, bez instrukce skoku by každý program mohl dávat jako výstupy pouze konstanty a vstupní data zvýšená či snížená o konstantu – dumejte podrobněji, proč...]
- `INC` – Mějme program, který nemá žádné vstupy a pouze jediný výstup: hodnotu 1 uloženou na adrese 1. Nechť je navíc na začátku výpočtu celá paměť vynulovaná (my sice o počátečním obsahu paměti dle definice nic nevíme, ale právě proto musí náš program fungovat i za této situace). Pak ovšem neexistuje žádná posloupnost instrukcí `DEC` a `CLR`, která by nám mohla kýženou jedničku vygenerovat, poněvadž obě z nuly udělají opět nulu (známé pravidlo „Nula od nuly pojde“) a to, že se mezi tím ještě někam skáče, na věci zaručeně nic nezmění.
- `DEC` a `CLR` současně – z analogických důvodů by nebylo možno vygenerovat nulu, byla-li by příslušná buňka nenulová (pro změnu není možnost hodnotu snížit).

Instrukci `DEC` samotnou lze ovšem nahradit velice snadno instrukcemi ostatními: Použijeme dvě pomocné paměťové buňky označené α a β (takové jistě máme k dispozici, protože každý konečný program může použít pouze konečně mnoho paměťových buněk, zatímco paměť jako taková je nekonečně velká) a `DEC` φ provedeme takto:

`CLR` α	$\alpha = 0$
`CLR` β	$\beta = 0$ (výsledek)
`JEQ` α`,`φ`,5`	$\varphi = 0 \Rightarrow$ Hotovo
`INC` α	Udržujeme $\alpha = \beta + 1$
`JEQ` α`,`φ`,3`	$\varphi = \beta + 1 \Rightarrow$ Výsledek je β
`INC` β	Zkoušíme novou hodnotu β
`JEQ` α`,`α`,-3`	Znovu test rovnosti...
`CLR` φ	A teď kopírujeme β zpět do φ
`JEQ` β`,`φ`,3`	Hotovo?
`INC` φ	Ne, zvýšíme
`JMP` φ`,`φ`,`-2	A znovu test...

Stručně: hledáme výsledek operace tak, že začneme nulou a postupně přičítáme jedničku, dokud nezjistíme, že zkoumané číslo zvýšené o jedna je rovno číslu původnímu. Jelikož všechna čísla jsou konečná, musíme v konečném čase dojít k cíli.

Instrukci `CLR` je nutno nahrazovat mírně sofistikovanějším způsobem: nemůžeme totiž snižovat nějakou proměnnou tak dlouho, dokud se nestane nulou, jelikož nemáme způsob, jak bychom to zjistili, nevíme-li o ostatních paměťových místech vůbec nic. Vezmeme si tedy dvě buňky paměti (cílovou buňku φ a pomocnou buňku α) a zařídíme, aby obsahovaly různé hodnoty (pokud jsou hodnoty náhodou stejné, prostě jednu zvýšíme o jedna). Nyní v každém kroku obě snížíme a dle definice instrukce `DEC` budou nadále různé, pokud ovšem nevyjdou obě 0 (k tomuto opět po nějaké době dojít musí, protože všechna čísla byla konečná!). Tedy takto:

`INC` α	Trik...
`JEQ` α`,`φ`,-1`	Nyní $\alpha \neq \varphi$
`JEQ` α`,`φ`,4`	Již hotovo ($\alpha = \varphi = 0$)
`DEC` α	Obě snížíme o 1
`DEC` φ	
`JMP` α`,`α`,-3`	A zkoušíme znovu...

9-3-4 Komplikátor — Robert Špalek

Nejdříve dáme dohromady něco podprográmků pro jednotlivé základní operace. Tyto prográmky nejsou nikterak těžké, přesto v nich bylo snadné nadělat spousty chyb – nejběžnější chybou bylo zničení obsahu zdrojových operandů po výpočtu (pokud byla některá proměnná použita ve výrazu vícekrát, pouze poprvé měla správnou hodnotu). Někteří si to uvědomili, a tak po každém výpočtu její hodnotu obnovili. Není ale jednodušší ji vůbec nezničit? Zde používáme 0–2 pomocné proměnné sloužící jako počítadla, takže vstupních proměnných se vůbec nemusíme dotknout.

* $Z \leftarrow X$ – Přiřazení X do Z se lehce provede postupnou inkrementací X a testováním rovnosti se Z, časová složitost je $3X + 2$, paměťová složitost je 0. První sloupec výpisu obsahuje instrukce, druhý operandy, třetí počet provedení a čtvrtý případné komentáře...

`CLR Z`	1	vynuluj
`JEQ Z,X,3`	$X + 1$	konec
`INC Z`	X	
`JEQ X,X,-2`	X	x-krát inkrementuj

* $Z \leftarrow X + Y$ – Sečtení čísel $X + Y$ vypočítáme přiřazením X do Z a přiřazením Y do T s průběžnou inkrementací Z, časová složitost je

$3X + 4Y + 4$, paměťová složitost je 1.

`CLR Z`	1	vynuluj
`JEQ Z,X,3`	$X + 1$	konec přiřazení
`INC Z`	X	
`JEQ X,X,-2`	X	X-krát inkrementuj
`CLR T`	1	vynuluj
`JEQ T,Y,4`	$Y + 1$	konec přičítání
`INC T`	Y	
`INC Z`	Y	
`JEQ X,X,-3`	Y	Y-krát inkrementuj

∗ $Z \leftarrow X - Y$ – Odečtení čísel $X - Y$ se od sečtení liší v jediném příkazu v druhém cyklu – proměnná Z se neinkrementuje, ale dekrementuje.

* $Z \leftarrow X * Y$ – Násobení $X * Y$ implementujeme jako Y-násobné přičtení čísla X, časová složitost je $3 + 5X + 4XY$, paměťová složitost je 2.

`CLR Z`	1	nuluj celek
`CLR` T_1	1	0 počet přičtení Y
`JEQ` T_1`,X,8`	$X + 1$	konec
`INC` T_1	X	
`CLR` T_2	X	0 počet přičtení 1
`JEQ` T_2`,Y,4`	$X \cdot (Y + 1)$	konec přičítání
`INC` T_2	$X \cdot Y$	
`INC Z`	$X \cdot Y$	
`JEQ Z,Z,-3`	$X \cdot Y$	Y-krát zvyš o 1
`JEQ Z,Z,-7`	X	X-krát přičti

* $Z \leftarrow X/Y$ – Při dělení X/Y budeme pořád zkoušet přičítat Y k pomocné proměnné T_1, dokud nedostaneme číslo X. Za každé úspěšné přičtění Y inkrementujeme Z. Časová složitost je $6 + 4\lfloor X/Y \rfloor + 5X$, paměťová složitost je 2. U počtů provedení instrukcí γ značí $\lfloor X/Y \rfloor$.

`CLR Z`	1	nuluj počítadlo
`JEQ Z,Y,-2`	1	dělení nulou?
`CLR` T_1	1	pomocný součet
`CLR` T_2	$\gamma + 1$	dílčí součet
`JEQ` T_2`,Y,5`	$\gamma + 1 + X$	povedlo se přičíst
`JEQ` T_1`,X,6`	$X + 1$	už jsme na konci
`INC` T_1	X	
`INC` T_2	X	
`JEQ Z,Z,-4`	X	přičítáme číslo Y
`INC Z`	γ	zvýšíme počítadlo
`JEQ Z,Z,-7`	γ	zkusíme zase přičíst

* $Z \leftarrow X\%Y$ – Zbytek po dělení $X\%Y$ určíme nejlépe tak, že budeme zvyšovat pomocnou proměnnou T až do X. Mezitím budeme zvyšovat i výsledný zbytek Z, který však při každém dosažení Y vynulujeme. Časová složitost je $5 + 2\lfloor X/Y \rfloor + 5X$, paměťová složitost je 1 (opět $\gamma = \lfloor X/Y \rfloor$).

`CLR T`	1	počítadlo
`JEQ Y,T,-2`	1	dělení nulou?
`CLR Z`	$1 + \gamma$	
`JEQ Z,Y,-1`	$1 + \gamma + X$	vynuluj zbytek
`JEQ T,X,4`	$X + 1$	už jsme na konci
`INC T`	X	
`INC Z`	X	
`JEQ Z,Z,-4`	X	zvyš obě počítadla

Konvertor aritmetických výrazů:

Tento problém se dá řešit různými přístupy. Každý z nich se ve vašich řešení aspoň jednou vyskytl. Přístupy jsou seřazeny podle elegance řešení:

1. Nejméně elegantní je nalezení podvýrazu, který můžeme vyhodnotit, vypsání assemblerovského prográmku a *nahrazení* starého podvýrazu v řetězci nově přiřazenou pomocnou proměnnou. Toto provádíme, dokud není v řetězci pouze 1 proměnná (a to výsledek). Je nasnadě, že zcela zbytečně mnohokrát modifikujeme vstupní řetězec.
2. O něco hezčí je použití rekurzivní procedury, která prochází řetězec zleva doprava. Narazí-li na levou závorku '`(`', pak se zavolá pro příslušný podvýraz a vrátí číslo proměnné, ve které je uložen podvýsledek. Díky této abstrakci můžeme předstírat, že dostaneme výraz složený pouze z proměnných a operací 2 priorit.
   V další fázi projdeme tento abstraktní řetězec zleva doprava a rozdělíme si jej na skupinky s operacemi '`*`', '`/`', '`%`', které jsou pospojovány operacemi '`+`', '`-`'. Pak můžeme pomocí zásobníkovu vyhodnotit všechny operace.
3. Nejobecnějším a asi nejelegantnějším je algoritmus *RPN* (Reverse Polish Notation), který se dá modifikovat pro libovolné přiřazení priorit jednotlivým operacím, pro vyhodnocování zleva i zprava, pro použití unárních funkcí a jiné speciality (čtěte: zvrhlosti).

Aritmetický výraz budeme číst v jednom průchodu zleva doprava, při výpočtu budeme používat dva oddělené zásobníky: jeden pro operandy a druhý pro operátory. Načteme-li operand (proměnnou), neuděláme nic než že ji uložíme na vrchol zásobníku. Tam bude čekat na vyhodnocení. Pod ní bude uložena buď ve výrazu předcházející proměnná nebo nějaký mezivýsledek vlevo od této proměnné.

Načteme-li naopak operátor, umístíme jej na vrchol zásobníku operátorů. Pak však porovnáme jeho prioritu s prioritou předchozího operátoru. Je-li vyšší (např. '`*`' před '`+`'), pak jej musíme vyhodnotit před novým operátorem (v zásobníku operandů pak už nebude vstupní proměnná, ale výsledek podvýrazu s vyšší prioritou – díky tomu se nám to samo rozdělí na prioritní podvýrazy). Takto se pokusíme vyhodnocovat předchozí operátory tak dlouho, dokud bude jejich priorita vyšší než priorita nová. Po těchto operacích se nám mnoho *čekajících* operátorů vyhodnotí, mezivýsledky se v paměti zkombinují a zůstane nám tam mezivýsledek nový.

Vyhodnocení aritmetické operace se provede vybráním dvou operandů z vrcholu zásobníku, výpočtem operace a uložením mezivýsledku.

Nyní je potřeba dořešit technické detaily: závorky a konec řetězce. Konec řetězce můžeme považovat za operaci s nejnižší prioritou vůbec (takže odstra-

ní ze zásobníku úplně všechno), která však není binární, ale unární (převezme výsledek a předá jej). Při výskytu levé závorky nesmíme nic předběžně vyhodnotit. Musíme počkat až na odpovídající pravou závorku, vyhodnotit vše mezi nimi a teprve pak předchozí operace. Nejjednodušší je považovat levou závorku '`(`' za operaci s nejvyšší prioritou vůbec (takže se před ní nic nevyhodnotí) a pravou závorku '`)`' za operaci s velmi nízkou prioritou (takže se vyhodnotí vše až k levé závorce).

Zde musíme ale udělat jednu výjimku: aby se vyhodnocování zastavilo na levé závorce a nepokračovalo dál na předchozí operace, musíme u levé závorky na zásobníku nastavit velmi nízkou prioritu. Tyto dva protichůdné požadavky na levou závorku splníme tak, že operátor má dvě priority: první v okamžiku porovnávání s předchozími operátory na zásobníku a druhou pro uložení vlastní priority na zásobník. U všech ostatních operátorů budou tyto priority stejné, ledaže bychom chtěli, aby se některý vyhodnocoval zprava doleva (vhodné např. pro umocňování). Pak bude jedna z nich o jedničku vyšší (která?).

Další tohoto vyhodnocování se dá snadno zahrnout kontrola výrazu (pomocí jedné stavové proměnné udávající, jaký objekt je zrovna očekáván). V programu použijeme vstupní funkci, která nám místo jednotlivých znaků bude přímo vracet *tokeny*, tj. nejmenší rozpoznatelné lexikální objekty (proměnné, operátory, závorky).

V zásobníku operandů budou uložena čísla proměnných. Při načtení vstupní proměnné se tam jenom uloží číslo 1–26, při spočítání mezivýsledku se tam uloží index na zaručeně novou proměnnou. Je možno implementovat hezký *memory management*, my si vystačíme se stále se zvyšujícím číslem první volné proměnné (pak budou v paměti díry), protože máme k dispozici libovolně velkou paměť. Obecně by paměťová složitost závisela na hloubce vnoření daného výrazu, které je však také lineární.

Časová složitost je $O(n)$, protože výraz projdeme jednou zleva doprava, konečnost je zřejmá, neboť v programu nepoužíváme žádné podezřelé cykly. Paměťová složitost je $O(n)$, neboť hloubka vnoření výrazu může být lineární s délkou, jako je tomu např. u výrazu $a + (b + (c + (d + \ldots)))$.

```
#include <stdio.h>

#define MAX_OPER 8
#define MAX_STACK 30

char expr[200];                          /* Výraz */

typedef struct {                         /* Definice operátoru */
    char Name;
    char CompPr, StackPr;
} TOperand;

typedef struct {                         /* Zásobník operátorů */
    char Num, Pr;
```

```
} T1Stack;

TOperand DefOp[MAX_OPER] = {
        { 0,  0,  0},                          /* Konec řetězce */
        {'(',  5,  1},
        {')',  2,  2},
        {'*',  4,  4},
        {'/',  4,  4},
        {'%',  4,  4},
        {'+',  3,  3},
        {'-',  3,  3} };

char *PgmOp[8] = {                             /* Základní operace */
        /* Vše se kopíruje na výstup, jen proměnné 'xyz' se nahradí patričnými čísly
           proměnných, a '#' adresou skoku před program. */
        "",                                                          /* 0 */
        "jeq 0,0,#\n",                                               /* Konec */
        "clr z\njeq z,x,3\ninc z\njeq x,x,-2\n",                     /* ← */
        "clr z\nclr 27\njeq 27,x,8\ninc 27\nclr 28\njeq 28,y,4\n"
            "inc 28\ninc z\njeq z,z,-3\njeq z,z,-7\n",               /* * */
        "clr z\njeq z,y,#\nclr 27\nclr 28\njeq 28,y,5\njeq 27,x,6\ninc 27\n"
            "inc 28\njeq z,z,-4\ninc z\njeq z,z,-7\n",               /* / */
        "clr 27\njeq y,27,#\nclr z\njeq z,y,-1\njeq 27,x,4\ninc 27\ninc z\n"
            "jeq z,z,-4\n",                                          /* % */
        "clr z\njeq z,x,3\ninc z\njeq x,x,-2\n"                      /* + */
            "clr 27\njeq 27,y,4\ninc 27\ninc z\njeq x,x,-3\n",
        "clr z\njeq z,x,3\ninc z\njeq x,x,-2\n"                      /* - */
            "clr 27\njeq 27,y,4\ninc 27\ndec z\njeq x,x,-3\n"};

T1Stack Oper[MAX_STACK];
int Var[MAX_STACK];
int OperCnt, VarCnt;

/* Proměnné: 0=výsledek, 1–26=vstupy, 27–28=pomocné, 29–...=mezivýsledky */

int lex (int *idx,  int *what)
{                                      /* Typ symbolu, index do řetězce, číslo symbolu */
    char a;
    for (; ; ) {
        a=expr[ (*idx)++];             /* Načti další znak */
        if (a>='a'&& a<='z') {         /* Operand */
            (*what)=a-'a'+1;
            return 1;
        } else {
            int i;
            for (i=0; i<MAX_OPER; i++)
                if (a==DefOp[i].Name) {
                    (*what)=i;
                    return 0;
                }
        }
        if (a!='␣'&& a!='\t') {
            printf ("Unknown symbol %c\n", a);
            exit (1);                  /* Přeskoč mezery */
```

```
        }
}
void output (src1, src2, dest, oper, instr)
int src1, src2, dest, oper;
int *instr;
{                       /* Vypíše kód odpovídající aritmetické operaci tohoto druhu */
        int idx=0;
        char a;
        while (a=PgmOp[oper][idx++])   /* Všechny znaky */
                switch (a) {
                        case '#':               /* Chybový skok před program */
                                printf ("%d", -*instr-1);
                                break;
                        case 'x':               /* Substituce proměnných */
                                printf ("%d", src1);
                                break;
                        case 'y':
                                printf ("%d", src2);
                                break;
                        case 'z':
                                printf ("%d", dest);
                                break;
                        case '\n':              /* Konec řádku zvyšuje */
                                (*instr)++;     /* počítadlo instrukcí */
                        default:
                                printf ("%c", a);
                }
}
void compile (void)
{
        int idx=0;                      /* Čtecí index */
        int type, what, free=29, instr=0;
        OperCnt=VarCnt=0;
        do {
                type=lex (&idx, &what);    /* Načti token */
                if (type)                  /* Proměnná */
                        Var[VarCnt++]=what;
                else {                     /* Operace */
                        while (OperCnt>0 &&
                        DefOp[what].CompPr<=Oper[OperCnt-1].Pr) {
                                output (Var[VarCnt-2], Var[VarCnt-1],
                                        free, Oper[OperCnt-1].Num, &instr);
                                Var[--VarCnt-1]=free++;
                                OperCnt--;
                        }                  /* Vyhodnoť vyšší priority */
                        if (what==2)       /* Odstranit '(' */
                                OperCnt--;
                        else if (what) {   /* Na zásobník kromě EOLN */
                                Oper[OperCnt].Num=what;
                                Oper[OperCnt++].Pr=DefOp[what].StackPr;
```

```
                }
            }
        } while (type || what);          /* Konec na konec řetězce */
        output (Var[0], 0, 0, 2, &instr);   /* Přiraď zásobník do 0 */
        output (0, 0, 0, 1, &instr);        /* Skoč před program */
}
int main (void)
{
        printf ("Enter an expression:␣");
        gets (expr);
        compile ();
        return 0;
}
```

9-3-5 !ɟǝɹɔǝs qoT — Jan Kotas

Označme $M = (p-1) \cdot (q-1)$. Ze zadání vyplývá, že máme řešit rovnici

$$(t \cdot s) \bmod M = 1,$$

která se dá převést na klasickou diofantovskou rovnici

$$ax + by = 1.$$

Uvažujme Euklidův algoritmus ve variantě, která pro daná a, b nejen vypočítá jejich největšího společného dělitele $D(a,b)$, ale také poskytne konstruktivní důkaz, že $D(a,b)$ je lineární kombinací čísel a a b. (V našem případě je $D(a,b) = D(s,M) = 1$.)

```
procedure Euclid(a,b:integer; var d,x,y:integer);
var d1,x1,y1:integer;
begin
  d := a; x := 1; y := 0; d1 := b;
  x1 := 0; y1 := 1;
  repeat
    if d > d1 then begin
      x := x - x1 * (d div d1);
      y := y - y1 * (d div d1);
      d := d mod d1;
    end else begin
      x1 := x1 - x * (d1 div d);
      y1 := y1 - y * (d1 div d);
      d1 := d1 mod d;
      end;
  until (d = 0) or (d1 = 0);
  if d = 0 then begin
    d := d1;
    x := x1;
    y := y1;
    end;
end;
```

Dokážeme, že během výpočtu dvojice čísel a, b a d, d_1 mají tytéž společné dělitele, a navíc $d = ax + by$ a $d_1 = ax_1 + by_1$. Před vstupem do cyklu toto tvrzení zřejmě platí. Ukážeme, že průchodem cyklem se jeho platnost nezmění. Uvažujme případ $d > d_1$, pro opačnou situaci je úvaha analogická.

Označíme $u = D(d, d_1)$, pak $d = u \cdot \alpha$, $d_1 = u \cdot \beta$, $D(\alpha, \beta) = 1$ a $D(d \bmod d_1, d_1) = D((u \cdot \alpha) \bmod (u \cdot \beta), u \cdot \beta) = D((u \cdot \alpha) - (u \cdot \beta) \cdot ((u \cdot \alpha) \bmod (u \cdot \beta)), (u \cdot \beta)) = u \cdot D(\alpha - \beta \cdot ((u \cdot \alpha) \bmod (u \cdot \beta)), \beta) = u$.

Druhá podmínka se také zachovává: $a \cdot (x - x_1 \cdot (d \operatorname{div} d_1)) + b \cdot (y - y_1 \cdot (d \operatorname{div} d_1)) = ax + by - (ax_1 + by_1) \cdot (d \operatorname{div} d_1) = d - d_1 \cdot (d \operatorname{div} d_1) = d \bmod d_1$.

Po ukončení cyklu tedy bude $D(a, b) = D(d, d_1) = D(d, 0) = d$ resp. $D(a, b) = D(0, d_1) = d_1$.

Časová složitost algoritmu je stejná jako u algoritmu Euklidova – to jest $O(\log a + \log b)$ průchodů cyklem. (Je možné poměrně snadno dokázat, např. použitím Fibonacciho čísel, že každými dvěma průchody cyklem se d zmenší alespoň dvakrát).

Uvedeným algoritmem vypočteme *celočíselné* řešení rovnice $ax + by = D(a, b)$. Pro nalezení $0 < t < M$ musíme ještě výsledek znormalizovat podle modulu. Při normalizaci můžeme využít snadno dokazatelného faktu, že $|x| < b$, resp. $|y| < a$.

Se započítáním časových složitostí operací s čísly je časová složitost algoritmu $O(\log^3 N)$, kde $N = p \cdot q$.

Zbývá dodat postup pro šifrovací exponent s. Vygenerujeme náhodné číslo s, $1 < s < M$. Dokud není $D(M, s) = 1$, inkrementujeme. Tento postup je zaručeně konečný, protože $D(M, M - 1) = 1$. Jelikož jsou prvočísla p, q velka, je M sudé, a tedy se můžeme omezit na $s > 2$ a hledání provádět s krokem 2.

Odvození dobrého odhadu časové složitosti hledání s by vyžadovalo hlubší poznatky z teorie čísel. (Intuitivně je vidět, že s najdeme většinou velmi rychle.)

Procedura pro výpočet klíčů pro algoritmus RSA počíta s čisly typu Integer. Převedení na počítání s velkými čísly je pouze technickou záležitostí. Testování soudělnosti M a s probíhá společně s generováním t.

```
procedure RSAKey(P,Q:Integer; var S,T:Integer);
var M,D,X,Y: Integer;
begin
  M := (P-1)*(Q-1);
  S := 3 + 2*Random(M div 2 - 1);
  repeat
    Euclid(S,M,D,X,Y);
    if D=1 then break;
    inc(S,2);
  until false;
  if X<0 then T:=M+X;
end;
```

9-4-1 Cor Sàir strikes back — Pavel Machek & Martin Mareš

Začneme úvahami o nutném počtu vážení: Určitě potřebujeme aspoň 3 vážení (protože ve 2 váženích může nastat pouze 9 různých výsledků a koulí je 12). 3 vážení nám ovšem stačí (dokážeme příkladem).

Nejprve si označíme balíky písmenky. (Můžeme použít 12 státních agentů, každý si bude pamatovat jeden balík a písmeno k němu přiřazené, nebo je nahradíme propisovací tužkou a napíšeme číslo přímo na balík – ale pozor na Agentskou odborovou organizaci.) [Michal Šída]

Tučnými písmeny budeme označovat balíky, o kterých víme, že mohou být normální nebo těžší (zbyla-li jediná taková, je jistě těžší), *kurzívou* pak normální či lehčí, KAPITÁLKAMI zaručeně normální balíky a latinkou pak balíky dosud zahalené tajemstvím.

Při prvním vážení použijeme abcd – efgh [ijkl] (levá miska – pravá miska [zbytek]). V případě, že je levá miska těžší, víme, že situace vypadá takto: **abcd***efgh*IJKL, pro pravou těžší analogicky *abcd***efgh**IJKL, pro obě stejné ABCDEFGHijkl.

Nyní pro první (analogicky i pro druhý) případ: vážíme **a***ef* – **b***gh* [**cd**IJKL]. Levá miska těžší ⇒ dostaneme **a**BCDEF*gh*IJKL, pravá těžší ⇒ A**b**CD*ef*GHIJKL, obě stejné ⇒ AB**cd**EFGHIJKL.

A pro případ třetí: vážíme ij – Ak [BCDEFGHl]. Levá těžší ⇒ ABCDEFGH**ij***k*L, pravá ⇒ ABCDEFGH*ij***k**L, obě stejné ⇒ ABCDEFGHIJKl.

Všechny zbývající situace jsou vždy typu **ab***c*, *ab***c** či **ab**, případně a, které je všechny možno triviálně rozhodnout jediným vážením (v posledním případě doplníme na druhou misku zaručeně normální kouli). Tudíž třetím vážením se vždy dozvíme, který balík je jiný a dokonce i *jak* je jiný.

Pokud jste dodrželi postup a nevyskytly se určité nevhodné okolnosti (např. váhy proměňující všechny zásilky v dýně nebo vysoká hustota tajných agentů květinářské společnosti Brambořík Ibišek Slunečnice), tak jste provedli zakázku za použití 3 vážení, takže si toho všimlo minimální množství lidí (jen cca 99,99%) a vy jste se stal nejslavnějším a nejbohatším važičem všech dob a žil jste šťastně až do smrti. (Což bylo ostatně asi 2 dny, protože tajná služba nerada nechává informace o své samozřejmě legální činnosti.) [opět dle Michala Šídy]

9-4-2 Green Bit Problem — Martin Mareš

Ve světle vzorového řešení předchozí ufounské úlohy (9-2-3 – Překla-datel) jest trik na „rozlousknutí“ tohoto problému téměř zřejmý: opět budeme něco řešit pro postupně se zvětšující skupiny velikosti 2^i.

Konkrétně se budeme snažit pro všechny části velikosti 2^i vždy spočíst, kolik je ve které části jedničkových bitů, známe-li to již pro obě její poloviny

a budeme podle zneužívat toho, že počet jedniček v k-bitovém čísle se vždy do k-bitového čísla vejde (jelikož $k < 2^k$ pro každé $k > 1$, kteroužto úvahu mnozí řešitelé opomněli učiniti).

Pro dvojbitové skupiny to lze provést triviálně a dokonce pro všechny najednou: stačí si označit a původní číslo, v němž jsou na lichých místech samé nuly a na sudých místech původní číslice, b číslo, v němž jsou na lichých místech opět nuly a na sudých místech číslice z oněch lichých míst a pak tato dvě čísla sečíst. Jelikož $2 < 2^2$, k přenosu mezi jednotlivými 2-bitovými skupinami nedojde a dostaneme přesně to, co jsme chtěli.

Nyní již máme 128-bitové číslo z rozdělené na 2^{7-k} skupin velikosti 2^k, každá z nich již obsahuje počet bitů na počátku v této skupině se nacházejících. Opět zavedeme a obsahující jen skupiny na sudých místech a b původní skupiny z lichých míst přesunuté na sudá místa. A $a + b$ je rozděleno na 2^{6-k} skupin velikosti 2^{k+1}. Takto pokračujeme tak dlouho, až získáme jednu skupinu velikosti $2^7 = 128$, což je hledaný výsledek.

Program bude vypadat takto (i^j značí zopakovat j-kráte posloupnost číslic i):

$$\begin{array}{llll}
a \leftarrow x > 1, & a \leftarrow a \,\&\, (01)^{64}, & b \leftarrow x \,\&\, (01)^{64}, & y \leftarrow a + b \\
a \leftarrow y > 2, & a \leftarrow a \,\&\, (0^2 1^2)^{32}, & b \leftarrow y \,\&\, (0^2 1^2)^{32}, & y \leftarrow a + b \\
a \leftarrow y > 4, & a \leftarrow a \,\&\, (0^4 1^4)^{16}, & b \leftarrow y \,\&\, (0^4 1^4)^{16}, & y \leftarrow a + b \\
a \leftarrow y > 8, & a \leftarrow a \,\&\, (0^8 1^8)^{8}, & b \leftarrow y \,\&\, (0^8 1^8)^{8}, & y \leftarrow a + b \\
a \leftarrow y > 16, & a \leftarrow a \,\&\, (0^{16} 1^{16})^{4}, & b \leftarrow y \,\&\, (0^{16} 1^{16})^{4}, & y \leftarrow a + b \\
a \leftarrow y > 32, & a \leftarrow a \,\&\, (0^{32} 1^{32})^{2}, & b \leftarrow y \,\&\, (0^{32} 1^{32})^{2}, & y \leftarrow a + b \\
a \leftarrow y > 64, & b \leftarrow y \,\&\, 0^{64} 1^{64}, & y \leftarrow a + b. &
\end{array}$$

Získáváme tak řešení ve 27 krocích...

9-4-3 Hellish Clocks — Robert Špalek

Označme si p_i původní nastavení i-tých hodin, r_i jejich požadované nové nastavení, a_{ij} posunutí i-tých hodin při zmáčknutí j-tého tlačítka a x_j počet zmáčknutí j-tého tlačítka.

Posunutí stejných hodin po zmáčknutí různých tlačítek se sčítají a protože je sčítání komutativní, nezáleží na pořadí stisků jednotlivých tlačítek, ale pouze na jejich počtu. Bude platit následující soustava rovnic (je to pouhé rozepsání pohybu hodin):

$$\begin{array}{l}
(a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n + p_1 - r_1) \bmod 37 = 0 \\
(a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n + p_2 - r_2) \bmod 37 = 0 \\
\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
(a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n + p_n - r_n) \bmod 37 = 0,
\end{array}$$

takže pokud operace '+', '−' a '∗' budeme provádět modulo 37, můžeme sou-

stavu napsat v maticovém tvaru:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \equiv \begin{pmatrix} r_1 - p_1 \\ r_2 - p_2 \\ \vdots \\ r_n - p_n \end{pmatrix}$$

Mnozí řešitelé věděli, že podobnou soustavu lze v reálných (resp. racionálních) číslech řešit Gaußovou eliminační metodou. Málokdo si však uvědomil, že tato metoda nepožaduje práci přímo s reálnými (resp. racionálními) čísly. Pro zdárný průběh tohoto algoritmu je nutno, aby daný číselný obor byl tělesem, tj. aby splňoval některá základní pravidla (komutativita, asociativita, distributivita, jednotkový a nulový prvek) a aby ke každému *nenulovému* číslu existovalo číslo inversní.

Lze snadno ukázat, že množina Z_p zbytkových tříd po dělení prvočíslem p je tělesem. Základní vlastnosti, jako je např. komutativita, lze dokázat snadno, takže jediným problémem je existence inversního prvku. Jak již bylo v některé z předchozích sérií KSP ukázáno, plyne z *Malé Fermatovy věty*, že jsou-li čísla a a p nesoudělná, pak $a^{p-1} \equiv 1 \pmod{p}$, tedy $a \cdot a^{p-2} \equiv 1 \pmod{37}$. Poněvadž a může být pouze z množiny $\{0, 1, 2, \dots, p-1\}$, je nenulové a vždy s p nesoudělné. Pro každé nenulové a tedy můžeme nadefinovat $a^{-1} = a^{p-2} \bmod p$.

Závěr je ten, že můžeme Gaußovu eliminační metodu použít stejným způsobem, jako kdybychom pracovali např. s racionálními čísly. Pouze musíme zaměnit původní aritmetické operace za nové.

Gaußova eliminační metoda se snaží upravit matici na trojúhelníkový tvar, tj. tvar, ve kterém jsou všechny koeficienty pod hlavní diagonálou nulové. Toho dosáhne řádkovými úpravami matice (násobení rovnic nenulovým číslem, výměna rovnic, sečtení rovnic). Snadno ověříme, že tyto úpravy nemění prostor všech řešení této soustavy.

Popišme po krocích tento důležitý algoritmus:

1. Provedeme druhý krok postupně pro všechny sloupce $i = 1, 2, \dots, n$.
2. Nyní jsou sloupce $1, 2, \dots, i-1$ v požadovaném tvaru, chceme upravit také i-tý sloupec.
   a) Je-li $a_{ii} \neq 0$, pak můžeme tuto rovnici podělit a_{ii}. Dále pro všechna $j = i+1, i+2, \dots, n$ odečteme a_{ji} násobek této rovnice od j-té rovnice. Tím vynulujeme všechna čísla v sloupci pod a_{ii}.
   b) Pokud je $a_{ii} = 0$, zkusíme nalézt takové $j \in \{i+1, i+2, \dots, n\}$, že $a_{ji} \neq 0$. Pokud takové j existuje, můžeme prohodit i-tou a j-tou rovnici, bude splněna podmínka nenulovosti a můžeme postupovat podle bodu (a).

c) Pokud žádné takové j nenalezneme, je celý sloupec včetně a_{ii} nulový. Pak žádné úpravy dělat nebudeme a skočíme rovnou na další sloupec. Při zpětném dosazování mohou nastanou dvě možnosti:

- vyjde nám rovnice typu $0x = 0$, pak je řešením jakékoliv x, např. $x = 0$ (ať nemusejí čerti zbytečně mačkat tlačítka),
- nebo nám vyjde $0x = 1$, pak nemá soustava rovnic řešení.

3. Soustava je v diagonálním tvaru, tedy v poslední rovnici se vyskytne maximálně jedna neznámá, v předposlední dvě... Rovnice budeme odzadu řešit. Pokud bude existovat více řešení, zvolíme např. nulu, pokud nebude existovat žádné, ohlásíme, že řešení neexistuje.

Program je přímou implementací uvedeného algoritmu. Jsou k němu připojeny komentáře, takže jeho pochopení by nemělo činit potíže. Nezbývá než ještě poznamenat, že tabulka inversních čísel je předpočítaná, aby se urychlil výpočet.

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

#define PRIME 37

int inverse[PRIME];                     /* Předpočítaná inversní čísla */
int n;
int **a;                                /* 0...n-1 matice, n pravá strana, n+1 řešeni */

void read_data (void) {
    int i, j;
    printf ("How many clocks?␣");
    scanf ("%d", &n);
    a=malloc (n*sizeof (int*));         /* Alokuj pole ukazatelů */
    for (i=0; i<n; i++){                /* Nastav pravé strany rovnic */
        a[i]=malloc ( (n+2)*sizeof (int));
        printf ("Clock %d: from to?␣", i+1);
        scanf ("%d%d", &j, a[i]+n);
        a[i][n]= (a[i][n]+PRIME-j)%PRIME;
    }
    for (i=0; i<n; i++)
    for (j=0; j<n; j++) {               /* Načti všechny koeficienty */
        printf ("How does %d. key move %d. clock?␣", j+1, i+1);
        scanf ("%d", a[i]+j);
    }
    for (i=0; i<PRIME; i++) {           /* Spočti inversní čísla */
        inverse[i]=1;
        for (j=0; j<PRIME-2; j++)
            inverse[i]=inverse[i]*i%PRIME;
    }
}
```

```
void swap (int *v1, int *v2, int n) {   /* Vymění složky dvou vektorů */
    int i, temp;
    for (i=0; i<n; i++) {
        temp=v1[i];
        v1[i]=v2[i];
        v2[i]=temp;
    }
}
void add (int *v1, int *v2, int mul, int n) {   /* Přičte násobek druhého vektoru */
    int i;
    for (i=0; i<n; i++)
        v1[i]= (v1[i]+v2[i]*mul)%PRIME;
}
void triangle (void) {                  /* Úprava na trojúhelnikovy tvar */
    int i, j;
    for (i=0; i<n; i++) {               /* Všechny sloupce */
        j=i;
        while (j<n && !a[j][i])         /* Najdi nenulový prvek */
            j++;
        if (j<n) {                      /* Existuje? */
            if (j!=i)                   /* Jiný řádek ⇒ prohodit */
                swap (a[i], a[j], n+1);
            add (a[i], a[i], inverse[a[i][i]]-1, n+1); /* Vynásobit řádek a[i][i]^-1 */
            for (j=i+1; j<n; j++)
                add (a[j], a[i], PRIME-a[j][i], n+1);
                    /* Přičíst vhodný násobek řádku */
        }
    }
    for (i=n-1; i>=0; i--) {            /* Zpětný průchod, všechny řádky */
        int sum=a[i][n];
        for (j=i+1; j<n; j++)           /* Přičti ostatní proměnné */
            sum= (sum+ (37-a[i][j])*a[j][n+1])%PRIME;
        if (a[i][i])                    /* Regulární koeficient */
            a[i][n+1]=sum;              /* Bude tam vždy 1 */
        else
            if (sum) {                  /* Nenulový výsledek */
                puts ("No solution");
                return;
            } else                      /* Cokoliv */
                a[i][n+1]=0;
    }
    for (i=0; i<n; i++)                 /* Výpis výsledku */
        printf ("Press %d. key %d times\n", i+1, a[i][n+1]);
}
int main (void) {
    read_data ();
    triangle ();
    return 0;
}
```

9-4-4 Whatsit? — Martin Mareš

Cílem této úlohy bylo, abyste zjistili nejenom *co* onen záhadný program dělá, ale také *proč* to vlastně dělá. Řešení bez důkazu tedy nebyla odměňována příliš štědře. Rovněž pak důkaz typu „zkoušel jsem to pro 10000 hodnot a fungovalo to“ nemá žádnou váhu (to, že program funguje pro N různých vstupů, ještě nevypovídá příliš o tom, co dělá pro ty ostatní, leda že by již žádné ostatní neexistovaly, ale v tomto příkladu bylo možností $\approx 10^9$, takže probrání všech v rozumném čase nehrozilo).

A co že vlastně program dělal? Inu, byl to interpreter jednoduchého programovacího jazyka. Text interpretovaného programu byl uložen v řetězci s, jazyk pracoval se sedmiprvkovým polem, jehož prvky si označíme k_0 až k_6. Prvek k_0 ukazoval do pole s na právě prováděný příkaz (pokud „vyjel“ pryč, program se zastavil), k_1 fungoval jako „data pointer“ (označíme si jej δ) – tedy obsahoval číslo prvku k_i, se kterým se prováděly některé operace a k_2 sloužil k uchování cílové adresy skoku. Jazyk se skládal z následujících příkazů:

- `<` Posunutí k_1 o jedno pole doleva s případným podtečením ($0 \to 6$).
- `>` Posunutí k_1 o dvě pole doprava s případným přetečením ($5 \to 0$, $6 \to 1$).
- `#` Není-li k_δ nulové, skok dle k_2 (ve skutečnosti o jednu pozici dále než kam k_2 ukazuje, protože ještě přeskakujeme na následující instrukci!).
- `$` Uložení k_0 do k_2 – tedy uchování aktuální pozice v programu
- `-` Snížení hodnoty k_δ o 1.
- `+` Zvýšení hodnoty k_δ o 1.
- `!` $k_\delta \leftarrow \delta$.
- `*` (cokoliv jiného): $\delta \leftarrow 2 \cdot \delta$.

Vstupy programu byly uloženy v k_3 a k_4, program začínal na pozici $k_0 = 1$, $\delta = k_1$ bylo inicializováno na 1 a k_2 na 2. Výsledek byl předán v k_6. Zbylý „registr“ k_5 sloužil jako pomocná proměnná.

Interpretovaný program vypadal takto (po přepsání neznámých znaků na

hvězdičky a očíslování pozic):

```
 1  >   11  *   21  #   31  -   41  -   51  +
 2  <   12  #   22  <   32  >   42  <   52  *
 3  +   13  >   23  *   33  +   43  +   53  >
 4  *   14  >   24  *   34  <   44  >   54  <
 5  >   15  +   25  *   35  +   45  <   55  >
 6  >   16  +   26  #   36  <   46  #   56  +
 7  *   17  +   27  -   37  #   47  >   57  #
 8  #   18  *   28  >   38  >   48  >   58  *
 9  <   19  >   29  <   39  <   49  !   59  -
10  $   20  <   30  $   40  $   50  *   60  *
```

A po krátkém prostudování jeho jednotlivých instrukcí dospějeme k tomu, že jeho jednotlivé části fungují takto:

1–4	$\delta = 2$, $k_2 = 6$
5–6	$\delta = 6$
7–8	$k_6 \leftarrow 2 \cdot k_6$, opakuj dokud $k_6 \neq 0$ (vše je typu `word`, pročež všechny operace fungují modulo 65536, a tak k nule dojít musíme)
9–10	$\delta = 5$, $k_2 = 10$
11–12	Prakticky stejným způsobem nulujeme k_5.
13–18	$\delta = 2$, $k_2 = 26$
19–21	$\delta = 3$ a pokud $k_3 \neq 0$, skáčeme na pozici 27.
22–26	$\delta = 2$, $k_2 = 208$, zastavujeme se (skok mimo).
27	$k_3 \leftarrow k_3 - 1$
28–30	$\delta = 4$, $k_2 = 30$
31	$k_4 \leftarrow k_4 - 1$
32–33	$\delta = 6$, $k_6 \leftarrow k_6 + 1$
34–35	$\delta = 5$, $k_5 \leftarrow k_5 + 1$
36–37	$\delta = 4$, pokud $k_4 \neq 0$, skáčeme na 31. Tímto jednoduchým cyklem přesouváme původní hodnotu k_4 do k_5 a současně ji přičítáme ke k_6, načež zbyde $k_4 = 0$.
38–40	$\delta = 5$, $k_2 = 40$
41	$k_5 \leftarrow k_5 - 1$
42–43	$\delta = 4$, $k_4 \leftarrow k_4 + 1$
44–46	$\delta = 5$ a pokud $k_5 \neq 0$, skáčeme na pozici 41. I hle, prohodili jsme k_4 a k_5!
47–49	$k_2 = \delta = 2$
50–52	$k_2 = 10$

53–57 $\delta = 5$, $k_5 = 1$, skáčeme (nepodmíněně) na pozici 11 (proměnné δ i k_2 jsou nastaveny stejně jako když jsme se tam dostali minule – vše tedy bude probíhat stejně).
58–60 Sem se nikdy nedostaneme...

Povšimněte si nyní, že se program skládá ze dvou částí: inicializace (příkazy 1–10, vlastně jen $k_6 \leftarrow 0$) a velký cyklus (11–57) prováděný k_3-krát a v každém průchodu přičítající k_4 ke k_6. Na konci vracíme k_6 jako výstup programu... Vypadá to jako násobení dvou přirozených čísel a nenechte se mýlit — *je to násobení dvou přirozených čísel!* Jak prosté, milý Watsone, záhada jest rozřešena (zejména uvědomíme-li si, že pokud $k_4 = 0$, provádíme vnitřní cyklus 65536-krát, takže tak jako tak nula po dlouhém počítání díky přetečení vyjde) a program můžeme triviálně zjednodušit na:

```
function thatsit(a,b:word):word;
begin
   thatsit := a*b;
end;
```

9-4-5 Something rotten? — Michal Koucký

Jak už to tak v našem semináři čas od času bohužel bývá (slibujeme, že se polepšíme!!), zadání této úlohy nebylo jednoznačné. Respektive ono možná jednoznačné bylo, ale odporovalo uvedenému příkladu. A po právu. Formulace zadání odpovídalo úloze, že si silničáři v království vyberou, které cesty nechají rozpadnout a tyto cesty se posléze rozpadnou. My máme silničářům sdělit, kolik cest mohou vybrat.

Vyřešit takovou úložku je veskrze triviální, neboť k tomu, aby města zůstala navzájem dosažitelná, postačuje kostra grafu silnic, která má vždy $N-1$ silnic. Pokud tedy má zůstat království souvislé, stačí nám udržovat $N-1$ vybraných cest a zbytek cest se může rozpadnout. Každý vidí, že spočítat počet cest a odečíst od toho $N-1$ si nezaslouží v zadání nabízených 12 bodů.

Tedy dvě okolnosti nasvědčovaly tomu, že zadání chce říci něco jiného, než říká. Zadání totiž chtělo říci, že silničáři nemají ve své kompetenci určit, které cesty se rozpadnou. Tamější silničáři totiž fungují tak, že implicitně nedělají nic. Když jim však ze všech koutů království přicházejí zprávy, že se dohromady rozpadá již x silnic, potřebují vědět, jestli je to stále ještě O.K., nebo jestli už náhodou nehrozí to, že se království rozpadne na dvě a více částí a měli by začít něco dělat. Jelikož poddaní neznají (případně když už znají, tak cesou alespoň zkomolí) názvy měst, hlásí silničářům pouze počty rozpadlých silnic.

Vaším úkolem bylo na základě mapy silnic zjistit, kdy musejí silničáři začít pracovat, tedy maximální počet cest, které se mohou rozpadnout, aniž by to ohrozilo celistvost království. Této úložce se v terminologii teorie grafů říká

hledání minimálního řezu v grafu, resp. v našem případě postačuje nalezení jeho velikosti. Co je to minimální řez? Řez v grafu je množina hran, které když se vypustí, tak se graf rozpadne. Minimální řez je pak samozřejmě řez, který je nejmenší ze všech možných řezů grafu. Silničáři tedy musí pracovat tehdy, pokud se počet rozpadlých silnic značně přiblížil velikosti minimálního řezu v jejich systému silnic.

Úloha hledání řezů v grafu souvisí s jinou zajímavou grafovou úložkou, s takzvanými toky v sítích. Síť není opět nic jiného než graf, avšak jelikož úloha má svůj předobraz v inženýrských sítích – potrubních a elektrických rozvodech – zkoumanému grafu se přezdívá síť. Tok v síti je pak prostě nějaké konkrétní rozložení proudu, ať již elektrického, nebo vody, na jednotlivé spojnice (hrany) sítě. Předpokládá se, že v jednom místě v síti je takzvaný zdroj, ze kterého to teče, a dále je tam jeden spotřebič, do kterého vše teče a jinde v síti již žádný proud nevzniká ani nezaniká, tj. to, co do ostatních uzlů vstoupí, to z nich i vystoupí. [*MM:* Údajně to není případ skutečných vodovodních ani elektrovodných sítí, protože tam se vše ztrácí všude...]

Evidentně spotřebič odebírá právě to, co zdroj vypouští. Když na rozložení proudu dáme omezení, že po každé spojnici může protékat buď jednotkový objem libovolným směrem, nebo nic, pak maximální objem proudu vypouštěný zdrojem nám říká, kolik je maximální počet hranově nezávislých cest mezi zdrojem a spotřebičem. To odpovídá velikosti minimálního řezu oddělujícího spotřebič od zdroje. Toto tvrzení zde ponecháme bez důkazu a odkážeme laskavého čtenáře na knihu L. Kučery Kombinatorické algoritmy. Když takto pro všechny možné spotřebiče a zdroje prozkoumáme minimální řezy, které je oddělují, nalezneme hledaný minimální řez.

Všimněte si, že můžeme zafixovat jeden konkrétní zdroj a prozkoumat jen všechny možné spotřebiče. To vyplývá z toho, že pokud se nějakým řezem graf rozpadne, speciálně tedy minimálním, pak námi vybraný zdroj zůstane v jedné z částí grafu a libovolný spotřebič z druhé části grafu nemůže odebírat více než je velikost použitého minimálního řezu. Zároveň vůči jednomu ze spotřebičů bude pro přepravu použita maximální zátěž minimálního řezu, tj. všechny jeho hrany. (Rozmyslete si správnost těchto tvrzení.)

Jak tedy zjistit velikost maximálního toku mezi daným zdrojem a spotřebičem? Tuto úložku vyřešíme tak, že budeme tok postupně zlepšovat. Na začátku řekněme, že nikde nic neteče. Poté se pokusíme nalézt takovou cestu od zdroje ke spotřebiči, že po jednotlivých hranách této cesty buď nic neteče, nebo po nich něco teče, avšak v opačném směru, tj. ve směru od spotřebiče ke zdroji. Pomocí této cesty můžeme zlepšit tok tak, že po jejích hranách, po kterých nic neteklo, necháme téci jednotkový objem směrem ke spotřebiči a po hranách, po kterých teklo něco směrem od spotřebiče, nebude téci nic. Snadno nahlédneme,

že takto zlepšený tok bude korektní, tedy v dotčených uzlech bude i nadále platit to, že co do nich vstupuje, to z nich i vystupuje, a opačně. Pozorný čtenář již jistě vytušil, že takovéto zlepšení provedeme maximálně $N - 1$-krát.

Zbývá doplnit, jak nalézt zlepšující cestu. Zlepšující cestu budeme hledat průchodem grafu do šířky a to tak, že budeme používat pouze zlepšující cesty, tj. budeme chodit pouze po hranách, kde nic neteče, nebo kde jdeme proti proudu. Takto budeme nalézat vrcholy, do kterých vede zlepšující cesta. Pokud již víme, že do daného vrcholu zlepšující cesta vede, tak ho již znovu procházet nebudeme, neboť nám postačuje nalezení nějaké zlepšující cesty. Hledání skončí tehdy, když nalezneme zlepšující cestu až ke spotřebiči, nebo tehdy, když už z žádného dosaženého vrcholu nelze jít po nevyužité hraně, či proti proudu. Jelikož při tomto průchodu musíme v nejhorším případě probrat téměř všechny hrany, je časová složitost této fáze $O(M)$, kde M je počet hran. (Předpokládáme, že $M > N$).

Celková časová složitost pak dosáhne $O(N^2M)$, neboť pro N spotřebičů budeme vyhledávat maximální tok, což znamená maximálně N-krát nalézt zlepšující cestu v čase $O(M)$. Jelikož počet hran $M < N^2$, lze též použít hrubší odhah $O(N^4)$. Paměťová složitost je $O(M)$, neboť si musíme pamatovat aktuální tok a popis grafu.

Zbývá snad jen dodat, že pro hledání minimálního řezu existuje i algoritmus pracující v čase $O(N^3)$ a taktéž pravděpodobnostní algoritmus, jenž řešení nalezne v průměrném čase $O(N^2 log^2 N)$. Pravděpodobnostní algoritmus je algoritmus, který si v průběhu výpočtu může házet kostkou. Některá tvrzení jsme zde ponechali bez důkazu (např. správnost algoritmu hledání maximálního toku), neboť rozsahem přesahují možnosti tohoto řešení. Zvídavé čtenáře tedy odkazujeme již zmíněnou knihu od Dr. Kučery, kde ke všem zde uvedeným tvrzením a algoritmům nalezne důkaz jejich správnosti. Vzorový program je pro jednoduchost napsán s časovou složitostí $O(N^4)$, neboť z hlediska počtu hran nemá optimálně napsané vyhledávání minimálního toku.

```
program KSP945;

const MAXN = 100;                       { Maximalni pocet vrcholu }

{ Pozn: cislovani mest je od 1..N. Konverze se provadi pri nacitani }

var N : integer;                        { pocet hran }
    Hrana : array [1..MAXN, 1..MAXN] of boolean; { udava pritomnost hran }
    MinRez : integer;                   { udava velikost zatim nejmensiho rezu }
    i,p : integer;                      { pomocna promenna }

    tok : array [1..MAXN, 1..MAXN] of integer; { pomocne pole pro NajdiMinRez }
                                        { udava aktualni tok }

function NajdiMinRez(z,s:integer) : integer;
{ Nalezne velikost maximalniho toku v grafu se zdrojem 'z' a spotrebicem 's' }
```

```
var i,j : integer;   { pomocne promenne }
    fronta : array [1..MAXN] of integer;  { fronta vrcholu pro prohledavani do sirky }
    ffirst, flast : integer;   { predek a zadek fronty }
    cesta : array [1..MAXN] of integer; { udava, odkud jsme navstivili dany vrchol }
    velikosttoku : integer;   { udava velikost max. toku, tj. pocet iteraci algoritmu }
begin
   for i:=1 to N do
     for j:=1 to N do tok[i,j]:=0;

   velikosttoku := 0;

   while true do
    begin
       { hleda zlepsujici cestu pruchodem do sirky }
       ffirst := 1;
       flast  := 1;
       fronta[1] := z;   { zdroj dame do fronty }

       for i:=1 to N do cesta[i]:=0;
       cesta[z] := MAXN + 1;

       while ffirst <= flast do
        begin
           j := fronta[ffirst]; inc(ffirst);   { vezmi prvni prvek z fronty }
           for i:=1 to N do
             if Hrana[j,i] and (tok[j,i] <= 0) and (cesta[i] = 0) then
                   { jdu po nevyuzite, nebo proti proudu do nenavstiveneho }
              begin
                 cesta[i] := j;                    { do 'i' prijdu z 'j' }
                 inc(flast); fronta[flast] := i;  { zarad do fronty }
              end;
           if cesta[s]<>0 then ffirst:=flast+1;
               { kdyz jsme dosahli spotrebice, nasilne ukonci }
        end;

       if cesta[s]=0 then   { nenasli jsme zlepsujici cestu ke spotrebice }
        begin
           NajdiMinRez := velikosttoku;
           exit;            { tok jiz nelze dale zlepsovat }
        end;

       inc(velikosttoku);
       { aktualizuj tok podle zlepsujici cesty }
       i:=s;
       while cesta[i] <> MAXN + 1 do
        begin
           inc(tok[cesta[i],i]); dec(tok[i,cesta[i]]);
           i:=cesta[i];
        end;
    end;
end;

procedure vstup; { nacte graf }
var i,j : integer;
begin
   write('Zadej pocet vrcholu :'); read(n);
   for i:=1 to n do
    for j:=1 to n do hrana[i,j]:=false;
```

```
   writeln('Zadavej jednotlive hrany. 0 0 ukonci');
   repeat
     read(i,j);
     if (i<>0) or (j<>0) then begin hrana[i+1,j+1]:=true; hrana[j+1,i+1]:=true; end;
   until (i=0) and (j=0);
end;

begin
   vstup;
   MinRez := n;
   for i:=2 to n do    { Pro zdroj = '1' a vsechna ostatni mesta jako spotrebice }
    begin
       p:=NajdiMinRez(1,i);
       if p<MinRez then MinRez := p;
    end;
   writeln('Silnicari mohou nechat rozpadnout ',MinRez - 1,' silnice');
end.
```

9-4-6 !ʇǝɹɔǝs ɹoT — Jan Kotas

Opravdu náhodná čísla není možné generovat „čistým“ algoritmem. Každý takový algoritmus má totiž podle definice předem předvídatelné chování, a tedy jím vygenerovaná čísla nelze považovat za opravdu náhodná, protože jsou efektivně vyčíslitelná. (Je možné sestrojit algoritmus, který bude generovat posloupnost „náhodných“ čísel, která se nikdy nezačne opakovat, např. číslice čísla π. Ani takovou posloupnost ale není možné považovat za posloupnost opravdu náhodných čísel.) [Pozn. *MM:* Nikdo ovšem dosud nedokázal, že něco jako náhodná čísla vůbec může existovat – pokud je vesmír deterministický, náhodná čísla neexistují, ale nepředvídatelná čísla stále ještě ano – zkuste přijít na to, proč.]

Musíme si tedy vzít na pomoc něco z reálného světa. Budeme se zde zabývat pouze metodami, které jsou vhodné pro spojení s počítačem. (Vynecháme obligátní házení kostkou, losování předmětů, atp.)

Všechny následující metody jsou založeny na tom, že měří náhodnou složku nějakého jevu.

Jednoduše realizovatelné je měřit s velkou přesností, co člověk považuje za chvíli, a za náhodné číslo prohlásit zbytek po dělení této hodnoty malým číslem:

```
function Random256: Byte;  { Náhodné číslo v rozsahu 0..255 }
 label start;
 var x: longint;
 begin
start:
 writeln('Za chvíli stiskni klávesu.');
 x := 0; while not keypressed do inc(x);
 while keypressed do readkey;
```

```
 if x < 100000 then begin
  writeln('Tohle není žádný rychlokurs!');
  goto start;
  end;
 Random256 := Byte(x); {rychlejší 'x mod 256'}
end;
```

Modifikace tohoto postupu je použita ve velmi rozšířeném programu Pretty Good Privacy (PGP). Pro masové použití např. v bankách je ovšem tento postup nevhodný.

Brownův pohyb, rozpad radioaktivního materiálu a další (prozatím) náhodné fyzikální jevy, jsou v praxi těžko použitelné, jelikož k jejich automatickému měření je třeba drahé zařízení.

Nejsnáze realizovatelné je měření šumu nějaké elektrické veličiny:

* šum éteru – měří se s velkou přesností proud, který teče z antény.
* šum sítě – měří se s velkou přesností napětí v zásuvce.
* šum polovodičového přechodu (Snem většiny konstruktérů diod je vyrobit takovou, která vůbec nešumí. Existuje ale také malá, nicméně dobře placená skupina konstruktérů diod, jejichž snem je vyrobit diodu, která šumí co nejlépe.)

Je možné, že za pár let bude pro podporu šifrování standardní součástí PC obvod pro generování náhodných čísel. Zatím se vyrábějí pouze specializované desky.

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	3	20	220
2.	Aleš Přívětivý	G Dašická, Pardubice	4	20	205
3.	Kamil Toman	G Tábor	4	19	174
4.	Tomáš Horáček	SPŠE Ústí n. Labem	3	17	157
5.	Michal Šída	G Tanvald	3	20	135
6.	Pavel Šanda	G Jar. Vrchlického, Klatovy	2	18	109
7.	Zdeněk Dvořák	G Nové Město na Moravě	2	20	107
8.–9.	Pavel Nejedlý	G Vídeňská, Brno	2	14	94
	Jakub Ouhrabka	G Jeronýmova, Liberec	3	15	94
10.	Vladimír Šišma	G M. Koperníka, Bílovec	3	15	92
11.	Petr Zika	G Voděradská, Praha	2	15	89
12.	Antonín Slavík	G Národní, Karlovy Vary	3	17	87
13.	Tomáš Čejp	Jiráskovo G, Náchod	4	12	86
14.	Jan Tožička	G Jateční, Ústí n. Labem	4	14	85
15.	Jan Březina	G Jeronýmova, Liberec	4	9	78
16.–17.	Petr Šimeček	G Tř. kpt. Jaroše, Brno	3	9	77
	Jiří Vyskočil	G Lanškroun	3	11	77
18.	Roman Kašpar	G Trutnov	4	13	72
19.	Jan Vítek	G Slaný	3	8	66
20.	Karel Volný	G Bráfova, Třebíč	3	10	61
21.	Vlastimil Janda	G Humpolec	3	8	58
22.	Michal Svoboda	G Nad Alejí, Praha	3	9	53
23.	Daniel Kastner	G Šternberk	4	10	52
24.	Vlastimil Křápek	G Křenová, Brno	4	7	51
25.	Radek Sýkora	G M. Koperníka, Bílovec	3	10	44
26.	Peter Vasil	G Pavla Horova, Michalovce	4	7	40
27.–28.	Jan Mysliveček	G Tř. kpt. Jaroše, Brno	2	10	39
	Tomáš Vyskočil	G Lanškroun	1	7	39
29.	Milan Kryl	G Jana Opletala, Litovel	2	8	37
30.	Igor Szöke	G Tř. kpt. Jaroše, Brno	3	8	36
31.	Michal Kopřiva	G Jiřího z Poděbrad	3	8	35
32.	Martin Lasoň	G Bohumín	4	7	34
33.	Jan Malach	G Zastávka u Brna	4	6	31
34.	Tomáš Smlsal	G Jiřího z Poděbrad	3	5	30
35.	Šárka Štěpánová	G Rychnov nad Kněžnou	3	4	28
36.	Pavel Šedivý	G Ml. Boleslav	4	3	27
37.	Ondřej Přibyla	G Tř. kpt. Jaroše, Brno	2	7	25

38.	Ladislav Kavan	Gym Ústavní, Praha	3	4	23
39.–42.	Miroslav Kašpar	SPŠE Pardubice	2	4	22
	Roman Nosek	G Jaroměř	4	4	22
	Lukáš Petrů	G Děčín	4	2	22
	Jakub Skopal	G Na Vítězné pláni, Praha	3	4	22
43.–44.	Marek Hlaváček	G Tanvald	4	4	21
	Vilém Maršík	G Na Pražačce, Praha	3	4	21
45.	Michal Wokoun	G Tř. kpt. Jaroše, Brno	4	3	19
46.–47.	Filip Kábrt	G Litoměřická, Praha	4	4	18
	David Klimek	G Rožnov p. R.	4	4	18
48.	Iva Marhánková	G Příbram	2	4	17
49.	Michal Pták	G Jičín	3	3	15
50.–51.	Vladimír Dubský	G Jiřího z Poděbrad	4	4	14
	Vít Špinka	G F. X. Šaldy, Liberec	4	4	14
52.–54.	Vlastislav Hynek	G Žatec	4	3	13
	Martin Kozák	G Jar. Vrchlického, Klatovy	1	4	13
	Blanka Kurková	?	?	5	13
55.–57.	Martin Dráb	G U Libeň. zámku, Praha	4	2	12
	Jaromír Malenko	G M. Koperníka, Bílovec	2	4	12
	Pavel Surynek	G Vlašim	3	3	12
58.–62.	David Holec	G Tř. kpt. Jaroše, Brno	2	4	11
	Martin Jelen	?	?	1	11
	Petr Kadlec	G Voděradská, Praha	3	2	11
	Stanislav Kříž	G Ostrov	3	3	11
	Jan Měkota	G Rožnov p. R.	3	3	11
63.–64.	Miroslav Krhounek	G Kralupy	3	3	10
	František Němec	ZŠ Uhelný Trh, Praha	0	4	10
65.–67.	Tomáš Hrubý	G Jar. Vrchlického, Klatovy	2	2	9
	Filip Rychnavský	OA Mariánské Lázně	4	3	9
	Miron Tegze	G Sladkovského nám., Praha	2	2	9
68.–72.	Stanislav Čejka	G Jaroměř	4	3	8
	Jiří Krystýnek	G M. Koperníka, Bílovec	3	2	8
	Dalibor Rožník	G Tř. kpt. Jaroše, Brno	2	1	8
	Kateřina Slováčková	SPŠ Kratochvílova, Ostrava	4	1	8
	Václav Těšínský	G Tábor	4	1	8
73.–74.	Libor Dener	G M. Koperníka, Bílovec	3	2	7
	Jiří Franek	G M. Koperníka, Bílovec	2	2	7
75.–77.	Radomír Chabiniok	G M. Koperníka, Bílovec	2	1	6
	Lenka Kocmanová	G Tř. kpt. Jaroše, Brno	4	1	6
	Tomáš Kubeš	G Jižní město, Praha	1	1	6
78.	Jan Machala	G Holešov	?	1	5

79.–82.	Tomáš Holubec	G Vsetín	2	1	4
	Lenka Kučerová	G Jičín	3	1	4
	Jakub Nosek	G Jižní město, Praha	0	1	4
	Martin Wokoun	G Tř. kpt. Jaroše, Brno	2	1	4
83.–84.	Milan Kvasnica	G Bohumín	3	1	3
	Martin Ostapčuk	G Chomutov	3	1	3
85.–95.	Jiří Dvořák	?	?	0	0
	Milan Filo	G Habrmanova, Hradec Kr.	3	0	0
	Michal Finěk	G Strakonice	4	0	0
	Petr Frank	G Kadaň	?	0	0
	David Hartman	G Příbram	2	0	0
	David Karban	G Orlová	2	0	0
	Jan Micza	G Český Těšín	3	0	0
	Lukáš Neterda	G U Libeň. zámku, Praha	4	0	0
	Jiří Roubínek	G Ždár n. Sázavou	4	0	0
	Pavel Šimsa	G Praha-Radotín	4	0	0
	Miroslav Vitásek	SPŠ Kratochvílova, Ostrava	4	0	0

Obsah

Martin Mareš a kolektiv

Korespondenční seminář v programování IX. ročník

Autoři a opravující úloh:
Martin Bělocký, Jan Kotas, Michal Koucký, Pavel Machek
Martin Mareš, Robert Šámal, Robert Špalek

Vydala Matematicko-fyzikální fakulta University Karlovy
Oddělení vnějších vztahů a propagace
Ke Karlovu 3, 121 16 Praha 2
Praha 1997

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

76 stran, 4 obrázky
Sazba písmem Computer Modern v programu TEX
Vydání první
Náklad 400 výtisků