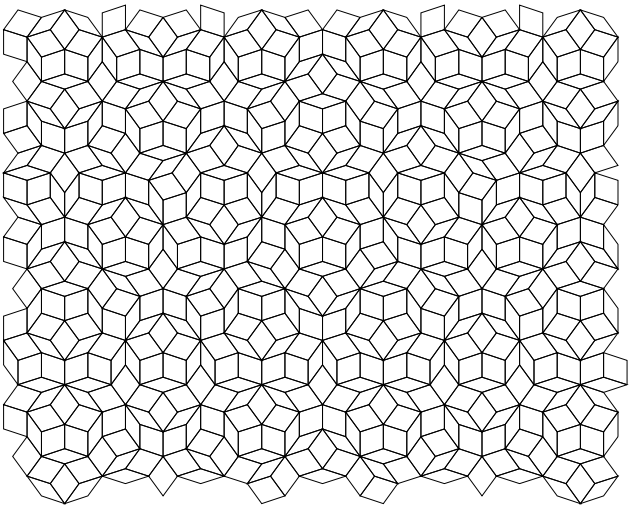
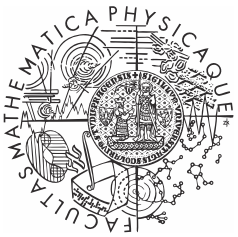
MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

XIV. ročník – 2001/2002





Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

XIV. ročník – 2001/2002

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož čtrnáctý ročník se vám dostává do rukou, patří k nejznámějším aktivitám pořádaným MFF 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 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 – existují korespondenční semináře z fyziky a matematiky při MFF, jakož i jiné programátorské semináře (kupříkladu bratislavský). Rozhodně si však nekonkurujeme, ba právě naopak – každý seminář nabízí něco trochu jiného, řešitelé si mohou vybrat z bohaté nabídky úloh a najdou se i takoví nadšenci, kteří úspěšně řeší několik seminářů najednou.

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

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

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

Zadání úloh

14-1-1 Kostky — 11 bodů

Frantík si rád hrál s kostkami. Nejvíce ho bavilo si z kostek stavět věže. Čím vyšší věž byla, tím větší měl Frantík radost. Jednoho dne Frantíka napadla záludná otázka, na kterou neznal odpověď ani jeho tatínek: Jaká nejvyšší věž jde z jeho kostek postavit? A protože tatínek před synkem nechtěl ukázat, že něco neví, obrátil se na vás, abyste mu pomohli otázku zodpovědět.

Váš program dostane na vstupu počet kostek N. Pak následuje popis kostek – pro každou kostku tři čísla (šířka, tloušťka a výška). Pro jednoduchost předpokládejte, že kostky se nedají otáčet. Dvě kostky lze postavit na sebe, pokud šířka resp. tloušťka spodní kostky je větší nebo rovná šířce resp. tloušťce horní kostky (to znamená, že kostky se na sebe dají postavit tak, že horní kostka stojí celou svou podstavou na spodní kostce). Vaším úkolem je vypsat výšku nejvyšší postavitelné věže.

Příklad: Ze tří kostek o rozměrech $1 \times 1 \times 1, 1 \times 5 \times 3, 5 \times 1 \times 2$ lze postavit věž výšky 4.

14-1-2 Orientační běh — 9 bodů

Orientační běh probíhá tak, že na trať vytyčenou v terénu v určitých intervalech postupně vybíhají závodníci. Na trati je několik stanovišť, které by měl každý závodník navštívit. Organizátoři mají ale problémy s ověřováním, zda každý závodník proběhl všechna stanoviště, a tak se rozhodli využít počítače.

Vaším úkolem je napsat program, který dostane na vstupu počet stanovišť N a pro každé stanoviště vzestupně setříděný seznam čísel závodníků, kteří jím proběhli. Vaším úkolem je napsat program, který vypíše čísla závodníků, kteří proběhli všechna stanoviště.

Příklad: Na trati jsou 4 stanoviště. Prvním stanovištěm proběhli závodníci 1, 6, 8, 10; druhým závodníci 6, 8, 9, 10; třetím závodníci 1, 6, 9, 10 a čtvrtým závodníci 1, 6, 10. Všemi stanovišti tedy proběhli závodníci 6 a 10.

14-1-3 Elektrická vedení — 10 bodů

Na planetě Aleton započala kolonizace. Aby měli kolonisté zajištěny alespoň základní životní potřeby, je nutné do celé země zavést elektřinu. Inženýři z Úřadu pro vesmírné plánování již navrhli úspornou síť rozvodných stanic a vedení mezi nimi. Je ale ještě třeba do některých rozvodných stanic umístit servisní střediska, která se budou starat o údržbu vedení. Aby byla údržba rychlá a jednoduchá, každé vedení musí mít alespoň na jednom svém konci servisní

středisko. Jak inženýři zjistili, vystačit si s malým počtem servisních středisek není vůbec jednoduché, a tak vás požádali, abyste jim s rozmísťováním pomohli svými programátorskými schopnostmi.

Vaším úkolem je napsat program, který na vstupu dostane počet rozvodných stanic N a dále seznam $N-1$ vedení mezi jednotlivými stanicemi. Každé vedení je určeno dvojicí čísel rozvodných stanic, mezi kterými vede. Předpokládejte, že se lze po vedení dostat mezi každými dvěma rozvodnými stanicemi (tzn. pro každé dvě stanice s_1 a s_k existuje posloupnost stanic $s_1, s_2, \ldots, s_k$ taková, že mezi stanicemi s_i a s_{i+1} vede přímé vedení pro každé $1 \leq i < k$) a že vedení nikde netvoří okruh. Na výstup má váš program vypsat nejmenší možný počet servisních středisek a čísla stanic, ve kterých mají být servisní střediska umístěna.

Příklad: Na planetě je 6 rozvodných stanic. Propojení jsou: $(1,2)$, $(2,3)$, $(2,4)$, $(4,5)$, $(4,6)$. Na planetě musí být alespoň 2 servisní střediska. Mohou být například ve stanicích 2 a 4.

14-1-4 k-Klidná posloupnost — 10 bodů

O posloupnosti řekneme, že je k-klidná, pokud se žádné dva její po sobě jdoucí prvky neliší o více než k. Vaším úkolem je napsat program, který pro dané k, n a posloupnost celých čísel $a_1, a_2, \ldots, a_n$ nalezne a vypíše nejdelší k-klidnou vybranou podposloupnost dané posloupnosti. Pokud je takových vybraných podposloupností více, stačí vypsat libovolnou z nich. Vybraná podposloupnost posloupnosti $a_1, \ldots, a_n$ je posloupnost čísel $a_{i_1}, a_{i_2}, \ldots, a_{i_l}$, kde $1 \leq i_1 < i_2 < \ldots < i_l \leq n$.

Příklad: Nejdelší 2-klidná podposloupnost posloupnosti 10, 7, 9, 12, 14, 15, 11, 13 je 10, 12, 14, 15, 13.

14-1-5 Turingovy stroje — 10 bodů

V letošním ročníku jsme se rozhodli uvést seriál na téma „Turingovy stroje“. Chtěli bychom vám v něm představit jeden z výpočetních modelů (výpočetní model je vlastně jakási abstrakce reálného počítače) a jeho rozličné modifikace, ukázat, jaký mají jednotlivé úpravy vliv na rychlost a výpočetní sílu modelu.

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

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

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

Pokud budeme v zadání mluvit o Turingově stroji *nad abecedou* Σ', myslíme tím, že vstup stroje bude zapsán pomocí písmen z abecedy Σ'. Samotný stroj ovšem může mít svou abecedu Σ, kterou bude používat při výpočtu, podstatně bohatší.

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

	0	**1**	Λ
q_0	$q_0, \mathbf{0}, R$	$q_0, \mathbf{1}, R$	q_1, Λ, L
q_1	$q_1, \mathbf{0}, L$	$q_2, \mathbf{0}, R$	—
q_2	$q_2, \mathbf{0}, R$	—	q_3, Λ, L
q_3	q_1, Λ, L	—	—

Vaším úkolem v této úloze bude navrhnout dva Turingovy stroje. Součástí vašeho řešení by samozřejmě neměla být pouze tabulka Turingova stroje, ale i popis, který osvětlí jeho funkci a správnost. Také by neměl chybět asymptotický odhad počtu kroků vašeho stroje a asymptotický odhad počtu použitých políček (tedy vlastně analogie časové a paměťové složitosti). Za počet použitých políček považujeme číslo nejzazšího políčka, na které vstoupila hlava Turingova stroje.

A nyní slíbené úlohy:

- Navrhněte stroj nad abecedou $\Sigma = \{\mathbf{0}, \mathbf{1}, \Lambda\}$, který vstupní slovo složené ze znaků **0** a **1** otočí (tzn. první znak bude poslední, druhý předposlední atd.).

- Navrhněte stroj nad abecedou $\Sigma = \{\mathbf{1}, \Lambda\}$, který ze vstupního slova sudé délky složeného ze znaků **1** odmaže (tzn. přepíše znakem Λ) jeho druhou polovinu.

Příklad stroje 1: Váš první stroj by měl slovo **01101** přepsat na slovo **10110**.

Příklad stroje 2: Druhý stroj by měl slovo **111111** přepsat na slovo **111**.

14-2-1 Kyvadlo — 10 bodů

V daleké Tramtárii žije král Trdlo. Tento král se velmi vyžívá v různých hříčkách a hlavolamech, ale hlavně v hazardních hrách. Jednou z jeho oblíbených her, kterou hraje se svými šlechtici, je hra „Kyvadlo". Hra se hraje na svisle umístěné obdélníkové desce. Na přední stranu desky vždy bankéř připevní několik kolíků a k nejvýše upevněnému kolíku přiváže provázek se závažím. Poté, co si každý z hráčů vsadí na nějaký kolík, bankéř natáhne provázek se závažím doprava do vodorovné polohy a závaží hodí směrem dolů. Závaží letí, provázek se otáčí okolo různých kolíků, až nakonec bude volná část provázku tak krátká, že nedosáhne k žádnému dalšímu kolíku a provázek se jen bude namotávat okolo jednoho kolíku. Hráč, který vsadil na tento kolík, vyhrává a bere vše. Pokud nikdo na kolík nevsadil, vyhrává bankéř. Jeden ze šlechticů na této hře prohrál již úctyhodné jmění a rozhodl se, že takhle to dál nejde. Proto si najal vás, abyste mu napsali program, který mu v sázení pomůže.

Váš program dostane na vstup počet kolíků N a souřadnice těchto kolíků. To jsou nějaké dvojice reálných čísel $(X_1, Y_1), \ldots, (X_N, Y_N)$. Pak ještě dostane délku provázku D. Na výstup má váš program vypsat číslo kolíku, okolo kterého se bude nakonec provázek namotávat. Pro účely naší úlohy předpokládejte, že kolíky mají nulový průměr a že bankéř hodí závaží dostatečnou silou (tedy že závaží bude mít dostatečnou rychlost na obtočení okolo libovolného kolíku).

14-2-2 Cenzoři — 11 bodů

Není to tak dávno, co se v Banánistánu ujal vlády další z řady moudrých panovníků (tedy alespoň tak to píší v novinách). Tento moudrý panovník brzy zjistil, že o něm někteří nerozumní novináři píší nepěkné věci, které poškozují jeho pověst. Proto se v zájmu své dobré pověsti rozhodl zavést v zemi cenzuru. Spolu se svými nejbližšími spolupracovníky vytvořil seznam slov, která se prostě v tisku nesmí objevit. Každý z cenzorů dostal seznam slov a měl za úkol z cenzurovaného textu každé slovo ze seznamu vyškrtnout. Protože ovšem tiskovin je velmi mnoho a cenzoři jsou nespolehliví a drazí, rozhodl se panovník po čase celou tuto proceduru mechanizovat. A to je již úkol pro vás.

Váš program dostane na vstupu seznam slov, která mají být z textu vyškrtána. V naší úloze považujeme text prostě za posloupnost znaků a na takové detaily, jako že slovo bývá ohraničeno mezerami, nehledíme. Dále dostane váš

program na vstupu text k cenzuře. Na výstup má pak váš program vypsat text s vyškrtanými zakázanými slovy. Pozor! Vyškrtnutím nějakého slova vám může opět vzniknout zakázané slovo!

Příklad: Pro zakázaná slova `voda`, `vodojem` a cenzurovaný text

`prasklvodovodajemuneteklavoda`

má váš program vypsat text `praskluneteklа`. Bylo totiž vypuštěno slovo `voda`, čímž vzniklo slovo `vodojem`, které bylo následně také vypuštěno. Nakonec bylo ještě vypuštěno slovo `voda` na konci textu.

14-2-3 Dekomprese — 10 bodů

Jak jistě víte, na pevném disku se dá ušetřit hodně místa kompresí souborů. Jednou z metod komprese je i metoda LZW. Nebudeme zde popisovat, jak tato metoda funguje. Postačí nám, když si řekneme, že soubor zakomprimovaný touto metodou obsahuje jednak normální data a jednak odkazy. Každý z odkazů ukazuje na nějaký předchozí úsek souboru (máme dánu jeho pozici a délku) a označuje, že při dekompresi se místo tohoto odkazu má vložit příslušný úsek souboru. Například soubor `abc`*(0,3)(0,6)(10,2)*`d` bude po dekompresi obsahovat `abcabcabcabcbcd`. Vaším úkolem bude rychle spočítat počet výskytů zadaného písmena v zakomprimovaném souboru.

Váš program dostane na vstupu počítané písmeno C a zakomprimovaný soubor (konkrétní formát vstupu necháme na vás, měl by ale přibližně odpovídat formátu v příkladu). Na výstup má vypsat počet výskytů C v odkomprimovaném souboru.

Příklad: V souboru `abc`*(0,3)(0,6)(10,2)*`d` je 5 výskytů písmena `b`.

14-2-4 Seznamování — 10 bodů

Na soustředění KSP přijelo mnoho účastníků. Ačkoliv někteří se již znali z předchozích let, jiní byli na soustředění poprvé, a tak byly na počátek soustředění naplánovány seznamovací hry. Jelikož účastníků je poměrně hodně, je potřeba je rozdělit na dvě skupiny. Nemá ale samozřejmě smysl seznamovat účastníky, kteří se již znají, a proto musí být rozdělení na skupiny takové, aby se každý účastník znal ve své skupině nejvýše s tolika lidmi, s kolika se zná ve skupině druhé. A to je úkol pro vás.

Váš program dostane na vstupu počet účastníků N a dále seznam známostí mezi účastníky (tedy seznam dvojic účastníků, kteří se navzájem znají – účastníky si pro jednoduchost očíslujeme od jedné do N). Na výstup má váš program pro každého účastníka vypsat, do které skupiny je třeba ho zařadit. Pokud je možných rozdělení více, stačí vypsat libovolné z nich.

Příklad: Pro 5 účastníků a známosti $(1,2)$, $(2,3)$, $(3,4)$, $(4,5)$, $(2,5)$, $(1,4)$ může váš program například vypsat rozdělení do skupin 1 2 1 2 1.

14-2-5 Turingovy stroje — 10 bodů

Ve druhé sérii budeme pracovat s vícepáskovými Turingovými stroji. Jak už název napovídá, vícepáskový stroj nemá pásku jednu, ale pásek k, kde $1 \leq k$ je nějaké pevné číslo nezávislé na vstupu. Každá páska je opět jednosměrně nekonečná, rozdělená na políčka a na každém políčku je nějaké písmeno z abecedy Σ. Nad každou páskou pracuje jedna hlava.

Řídící jednotka stroje je v každém okamžiku v nějakém stavu z množiny Q a rozhoduje se podle přechodové funkce $f(q,(z_1,\ldots,z_k))$. Ta pro každou kombinaci stavu q a k-tice znaků $(z_1,\ldots,z_k)$, které jsou pod jednotlivými hlavami, dává trojici $(q',(z'_1,\ldots,z'_k),(m_1,\ldots,m_k))$, kde q' je stav, do kterého řídící jednotka přejde v dalším kroku, $(z'_1,\ldots,z'_k)$ jsou znaky, kterými hlavy přepíší znaky $(z_1,\ldots,z_k)$, a $(m_1,\ldots,m_k)$ jsou pohyby, které mají provést hlavy. Mimo v první sérii zavedených pohybů L (doleva) a R (doprava) může být hlavě ještě předepsán pohyb N (zůstat na místě).

Výpočet vícepáskového Turingova stroje tedy probíhá obdobně jako výpočet stroje jednopáskového, jen stroj najednou pracuje na několika páskách. Práce stroje končí, pokud libovolná z hlav narazí do levého okraje pásky. Za výstup stroje se pak považuje obsah první pásky.

Protože u vícepáskových strojů je již občas trochu nepřehledné vše zapisovat pomocí tabulky, ukážeme vám v následujícím příkladu zápis ve tvaru „programu“.

Příklad: Následující 2-páskový Turingův stroj pracující nad abecedou $\Sigma = \{\mathbf{0},\mathbf{1},\Lambda\}$ převrátí slovo zadané na vstupu (první pásce):

$$\begin{array}{ll}
(q_0,(x,?)) \to (q_1,(\Lambda,x),(R,R)) & x \in \{\mathbf{0},\mathbf{1}\} \\
(q_1,(x,?)) \to (q_1,(x,x),(R,R)) & x \in \{\mathbf{0},\mathbf{1}\} \\
(q_1,(\Lambda,?)) \to (q_2,(\Lambda,\Lambda),(L,N)) & \\
(q_2,(x,?)) \to (q_2,(x,?),(L,N)) & x \in \{\mathbf{0},\mathbf{1}\} \\
(q_2,(\Lambda,?)) \to (q_3,(\Lambda,?),(N,L)) & \\
(q_3,(?,x)) \to (q_3,(x,\Lambda),(R,L)) & x \in \{\mathbf{0},\mathbf{1}\}
\end{array}$$

Poznámka: ? v levé části znamená libovolný znak, v pravé části pak znak, který byl zastoupen ? v levé části.

Vaším úkolem v této úloze bude navrhnout vícepáskový (počet pásek si zvolte sami) Turingův stroj nad abecedou $\Sigma = \{\mathbf{0},\mathbf{1},\Lambda\}$, který číslo zapsané v jedničkové soustavě na první pásce převede do dvojkové soustavy (výstup má být též na první pásce). Cifry s nejnižší vahou by měly být na levém konci pásky.

Příklad: Číslo **1111111111** by měl váš stroj převést na **0101**.

14-3-1 Ježíškův problém — 11 bodů

Vždy před Vánoci má Ježíšek problém, jak rozdělit mezi děti dárky. V průběhu roku z nebe Ježíšek pečlivě sleduje, jak je které dítě hodné, a podle toho pro něj udržuje *index zlobivosti*. Před Vánoci by pak chtěl mezi děti rozdělit dárky tak, aby když dítě A má vyšší index zlobivosti než dítě B, tak A dostane nejvýše tolik dárků, kolik dostane B (předpokládejte, že žádné dvě děti nemají stejný index zlobivosti). Může se přitom stát, že hodně zlobivé děti nedostanou žádný dárek a hodné děti dostanou dárků více. Dárky musí být rozděleny všechny. Aby se Ježíšek mohl dopředu připravit na rozhodování, chce po vás napsat program, který dostane počet dětí N a počet dárků M a na výstup vypíše počet způsobů, kterými lze mezi děti dárky rozdělit. Sami si rozmyslete, že jednotlivé indexy zlobivosti vlastně nepotřebujete znát.

Příklad: Pro čtyři děti a čtyři dárky je pět možností, jak dárky mezi děti rozdělit: (0,0,0,4), (0,0,1,3), (0,0,2,2), (0,1,1,2), (1,1,1,1).

14-3-2 Cestářův problém — 10 bodů

S novým rokem vešel v platnost i nový rozpočet v Agranetánii. V něm se našlo pouze velmi málo prostředků na údržbu cest a silnic. Ministr Poslů a cest rozhodl, že je třeba co nejvíce snížit délku udržovaných cest, a tím pochopitelně i náklady na jejich údržbu. Na druhou stranu je ovšem třeba zajistit, aby se po udržovaných cestách stále dalo dojet z libovolného města do libovolného jiného (jinak by se totiž král mohl zlobit, že jeho luxusní kočár příliš kodrcá, a ministra by sesadil). Protože úkol to není jednoduchý, rozhodl se ministr najmout vás, abyste mu napsali program, který problém vyřeší.

Váš program dostane na vstupu počet měst N a dále současný počet cest M. Pak následuje popis M cest. Pro každou cestu dostane program čísla dvou měst (města si očíslujeme od jedné do N), mezi kterými cesta vede, a dále délku příslušné cesty. Na výstup má váš program vypsat cesty, které je třeba zachovat, aby se stále ještě šlo dostat mezi každými dvěma městy, a přitom byl součet délek cest nejmenší možný.

Příklad: Pro osm měst a deset cest (cesty jsou zapsány ve tvaru (a, b, d), kde a a b jsou města a d je délka cesty) (1,2,1), (2,3,4), (1,4,3), (2,4,2), (2,5,2), (3,6,3), (5,6,1), (6,8,4), (4,7,1), (5,7,2) jsou hledané cesty (1,2), (2,4), (4,7), (7,5), (5,6), (6,3), (6,8).

Poznámka: Pokud se vám předchozí zadání zdá příliš triviální, zkuste se zamyslet, jak vaše řešení zrychlit za předpokladu, že zadaný graf lze nakreslit do roviny bez křížení hran. Za kvalitní řešení tohoto složitějšího problému je možno získat až pět bonusových bodů.

14-3-3 Králův problém — 10 bodů

Král Trdlo z Tramtárie vás poté, co se doslechl o vašich řešeních hry Kyvadlo, požádal o pomoc s další oblíbenou hazardní hrou. Jedná se o hru „Kamínky“. Tato hra se hraje na hracím plánu ve tvaru stromu (tedy grafu, kde mezi každými dvěma vrcholy vede právě jedna cesta). Bankéř vždy podle určitých pravidel vytvoří strom, na kterém se bude hrát. Pak v něm zvolí jeden významný vrchol (nazvěme ho *kořen*) a všechny hrany ve stromu zorientuje směrem ke kořeni. Nyní může hra začít. V každém tahu může hráč položit jeden hrací kámen na takový vrchol, jehož všichni předchůdci (tedy vrcholy, z nichž vede do onoho vrcholu hrana) již na sobě kameny mají, nebo může z libovolného vrcholu kámen odebrat. Speciálně na vrcholy, do kterých nevede ze žádného vrcholu hrana, lze položit kámen kdykoliv. Hráč vyhrává, pokud se mu podaří umístit dle pravidel hry kámen do kořene stromu. Problémem v této hře je, že hráč má pouze tolik kamenů, kolik si od bankéře na počátku hry koupil. Je proto třeba co nejlépe odhadnout potřebný počet kamenů. A to je přesně problém, o jehož řešení vás král Trdlo požádal.

Váš program dostane na vstupu počet vrcholů stromu N. Dále dostane číslo vrcholu, který byl bankéřem vybrán za kořen (vrcholy si očíslujeme od jedné do N). Nakonec dostane popis $N-1$ hran ve stromu. Pro každou hranu dostane dvojici čísel vrcholů, mezi nimiž hrana vede. Na výstup má váš program vypsat minimální počet kamínků potřebný k vyhrání hry.

Příklad: Máme strom na sedmi vrcholech. Kořen je ve vrcholu jedna. Hrany ve stromu jsou: (2,1), (3,1), (4,1), (5,2), (6,2), (7,3). Na vyhrání hry jsou třeba čtyři kamínky. Můžeme táhnout například: $+5$, $+6$, $+2$, -5, -6, $+7$, $+3$, -7, $+4$, $+1$ (‘+’ před číslem znamená, že do daného vrcholu přidáváme kámen; ‘$-$’ před číslem znamená, že z daného vrcholu kámen odebíráme).

14-3-4 Hammingův problém — 10 bodů

Pro dvě čísla si nadefinujeme *Hammingovu vzdálenost* jako počet bitů, ve kterých se čísla liší. Například pro čísla 7 a 9 je jejich Hammingova vzdálenost tři, protože 7 je ve dvojkové soustavě 0111 a 9 je 1001, a čísla se tedy liší ve spodních třech bitech. Vaším úkolem je napsat program, který pro dvě čísla (předpokládejte, že se vejdou do standardního celočíselného typu) co nejrychleji zjistí jejich Hammingovu vzdálenost.

14-3-5 Turingův problém — 10 bodů

V minulých dvou sériích jste pracovali s jednopáskovým a vícepáskovým Turingovým strojem. Asi je každému jasné, že co jde spočítat na jednopáskovém stroji, půjde spočítat i na stroji vícepáskovém. Navíc vícepáskový stroj

nám umožňuje mnohé úlohy spočítat rychleji než stroj jednopáskový – například otočení slova zvládneme na dvoupáskovém stroji snadno v lineárním čase. Je poměrně zajímavým faktem (i když důkaz není úplně jednoduchý), že dvoupáskový stroj sice umožňuje spočítat některé úlohy rychleji než stroj jednopáskový, ale třípáskový stroj nám už oproti dvoupáskovému další zrychlení neposkytne. Další zajímavou skutečností je, že pokud má alespoň dvoupáskový Turingův stroj vyšší než lineární časovou složitost (tedy například $n \log n$), tak pak pro každé přirozené číslo $c > 1$ lze vytvořit Turingův stroj, který bude c-krát rychlejší. Tato skutečnost je také jedním z důvodů, proč v teoretické informatice nehledíme na konstanty. Na důkazu tohoto tvrzení si teď ukážeme některé techniky, které se při podobných důkazech používají.

Náš nově vytvářený stroj bude pracovat ve třech fázích. V první fázi zakomprimuje vstup, v druhé fázi bude konstantním počtem svých kroků simulovat alespoň c-krát více kroků původního Turingova stroje a ve třetí fázi bude dekomprimovat výstup.

Vstup budeme komprimovat tak, že si vstupní pásku rozdělíme na d-tice (d zvolíme dostatečně velké – řekněme jako $20 \cdot c$). Do abecedy si přidáme nové znaky – pro každou možnou d-tici jeden. Například pro abecedu $\{\mathbf{0}, \mathbf{1}, \Lambda\}$ a $d = 2$ do abecedy přidáme písmena $(\Lambda, \Lambda), (\Lambda, \mathbf{0}), (\Lambda, \mathbf{1}), (\mathbf{0}, \Lambda), \ldots, (\mathbf{1}, \mathbf{1})$. Protože jak d, tak velikost abecedy jsou pevné, přidáme tím do abecedy pouze konstantní počet znaků, a tedy jsme se nedostali do rozporu s definicí Turingova stroje (velikost abecedy nesmí záviset na velikosti vstupu). Komprese pak probíhá tak, že si stroj vždy „do stavu“ načte jednu d-tici a znak odpovídající d-tici zapíše na druhou pásku. Když takto stroj zakomprimuje celý vstup, přepíše původní vstupní slovo lambdami a přejde do druhé fáze.

Druhou fázi si ukážeme pouze pro jednu pásku. Pro více pásek by konstrukce fungovala úplně stejně, pouze by byla technicky komplikovanější. V této fázi budeme pěti kroky našeho stroje simulovat alespoň d kroků původního stroje. Ve stavu našeho stroje si mimo stavu původního stroje budeme udržovat obsah políčka pod hlavou a políček sousedních (nezapomeňte, že již pracujeme nad zakomprimovanou páskou, takže si vlastně ve stavu pamatujeme $3d$ políček). Krajní políčko na pásce má pouze jedno sousední políčko, takže pro toto políčko si musíme vytvořit ještě speciální stavy. Snadno si můžete ověřit, že stavů jsme opět přidali pouze konstantní (i když značné) množství. Přechody nového stroje ze stavu q, kde q kóduje $3d$-tici písmen $(\alpha_1, \ldots, \alpha_{3d})$ a nějaký stav q' původního stroje, vytvoříme následujícím způsobem: Vezmeme ona políčka, která máme uložena ve stavu, a necháme původní stroj nad nimi běžet (bude začínat ve stavu q'), dokud z nich jeho hlava nevyběhne. Přechod nyní nadefinujeme do stavu, který odpovídá obsahu oněch $3d$ políček po běhu stroje, a stavu, ve kterém stroj z $3d$ políček vyběhl. Přechod nebude ve skutečnosti tak jednoduchý – ještě než přejdeme do stavu, který jsme si vyhlédli,

musíme také změněná políčka přepsat na pásku. To je ale pouze technická komplikace, kterou snadno vyřešíme tak, že každý přechod se bude odehrávat přes čtyři stavy pro tento přechod speciálně vytvořené (takovýchto stavů budeme opět potřebovat pouze konstantní množství, takže dodržíme požadavky v definici Turingova stroje). V prvním ze stavů zapíšeme na políčko pod hlavou nový obsah prostředních d políček a přejdeme hlavou doleva. Tam zapíšeme obsah prvních d políček a přesuneme se dvakrát doprava, kde zapíšeme obsah posledních d políček. Nakonec se přesuneme nad políčko obsahující d-tici, nad kterou momentálně má stát hlava. Pokud původní stroj vyběhnutím z $3d$ políček vyběhl z pásky a tím skončil, přejdeme po předchozích operacích do třetí fáze. Protože původní stroj začíná buď nad d-tým nebo $2d$-tým políčkem z oněch $3d$ zapamatovaných, bude mu vyběhnutí trvat alespoň d kroků, a tedy našimi pěti kroky nasimulujeme alespoň slibovaných d kroků původního stroje.

Ve třetí fázi už jen dekomprimujeme výstup podobným způsobem, jakým jsme ho v první fázi komprimovali. Všimněte si, že naše konstrukce skutečně nutně potřebuje alespoň dvě pásky, protože jinak bychom kompresní a dekompresní fázi nemohli dělat v lineárním čase.

Nyní, když jsme si ukázali jeden z obtížnějších důkazů v této oblasti teorie složitosti, přichází čas na úlohu pro vás. Ukázali jsme si, že jednopáskové a vícepáskové mají některé dosti odlišné vlastnosti – například dvoupáskové stroje není problém konstantně zrychlovat. Vyvstává proto přirozená otázka: Pokud máme vícepáskový Turingův stroj, dokážeme sestrojit jednopáskový Turingův stroj, který bude dávat stejné výsledky (tzn. na jeho pásce bude na konci stejný výstup, jako na první pásce vícepáskového stroje)? Vaším úkolem je takovouto konstrukci vymyslet (samozřejmě se zdůvodněním, proč by měla fungovat).

14-4-1 Povodeň — 11 bodů

Karchamon byl prostým rolníkem hospodařícím na břehu Nilu. Každé jaro jeho políčka zaplavovala řeka a když voda ustoupila, zbyla na jeho políčkách různě velká jezírka vody. Karchamona jednoho dne napadlo, že zachycenou vodu by mohl využít k zavlažování alespoň pro několik následujících týdnů, a nemusel by tak nosit vodu přímo z řeky. Aby ale zjistil, jestli se mu vůbec takovéhle zavlažování vyplatí, potřeboval by vědět, kolik vody se zachytí na jeho políčkách. A to je problém, se kterým si Karchamon nevěděl rady. Budete si vědět rady vy?

Vaším úkolem je navrhnout algoritmus a napsat program, který dostane na vstupu N a matici přirozených čísel $N \times N$ (hodnoty v matici zachycují výšku terénu na jednotlivých čtverečcích) a spočítá kolik jednotek vody (čtverečky odpovídající prvkům matice mají jednotkové hrany) se v popsaném terénu

zachytí (předpokládejte, že terén byl na počátku celý zatopen a že krajina „okolo“ matice má výšku nula).

Příklad: Pro $N = 4$ a terén

```
4444
3113
3213
4444
```

je objem zadržené vody 7 jednotek.

14-4-2 Posloupnost — 10 bodů

Na vstupu je dána uspořádaná posloupnost celých čísel $a_1, \ldots, a_n$. Navrhněte algoritmus a napište program, který zjistí, zda se ve vstupní posloupnosti nachází tři čísla x, x^2, x^3 a pokud ano, tak je vypíše.

Příklad: Pro vstupní posloupnost -5, -1, 2, 4, 5, 7, 8, 10 jsou hledaná čísla 2, 4, 8.

14-4-3 Koordinátor — 12 bodů

Inženýři společnosti IBM (Interconnected Bloody Machines) vytvořili nový neomylný počítač. Ten se skládá z N výpočetních jednotek (procesorů). Všechny jednotky provádějí ten samý program a vždy na konci výpočtu své výsledky porovnají. Pokud se nějaká z jednotek odchýlí, tak je označena jako vadná a nadále se nebude účastnit výpočtů. Problémem ale je, že v síti musí být jedna „vedoucí“ jednotka – takzvaný *koordinátor*. Je ovšem poněkud nepraktické, aby, když se koordinátor porouchá, selhal celý počítač. Proto je třeba navrhnout nějaký postup, jakým se jednotky v síti dohodnou na koordinátorovi. A to je již úkol pro vás.

Máte dáno N – počet výpočetních jednotek (jednotky jako takové ale počet jednotek nevědí!). Každá z jednotek má své unikátní identifikační číslo, které má uložené ve speciální proměnné *ID*. Jednotky jsou spojeny do kruhové sítě a každá jednotka může komunikovat pouze se svým levým a pravým sousedem. Všechny jednotky začnou ve stejný okamžik vykonávat vámi zadaný program (ten musí být pro všechny jednotky identický). Ten mimo standarních příkazů programovacího jazyka může ještě obsahovat příkazy `SendLeft(message)`, `SendRight(message)`, které pošlou levému resp. pravému sousedovi zprávu `message`. Dále může obsahovat příkaz `Receive(buffer)`, který první došlou a nezpracovanou zprávu zkopíruje do bufferu. Předpokládejte, že zprávy se neztrácí, dochází v tom pořadí v jakém byly odeslány a že u každé jednotky jsou neomezené fronty na příchozí zprávy. O vzájemné rychlosti výpočetních jednotek však již nemůžete dělat žádné předpoklady. Navrhněte postup (přímo

program psát nemusíte), jakým mezi sebou jednotky mají zvolit koordinátora tak, aby bylo celkově vysláno co nejméně zpráv.

14-4-4 Triramidy — 10 bodů

Slavný archeolog Bedřich Šílený přišel na stopu jedné dosud nepoznané civilizace. Příslušníci této civilizace byli velmi vyspělí stavebníci. Zajímavé je, že všechny stavby, které byly civilizací postaveny, měly půdorys pravoúhlého rovnoramenného trojúhelníku, jehož odvěsny byly rovnoběžné s triběžkami a trileďníky (takto pan Šílený pojmenoval souřadnice odpovídající našim rovnoběžkám a poledníkům). Proč měly stavby takový neobvyklý tvar, se zatím nikomu nepodařilo zjistit. Bedřich má jednu teorii hodnou svého proslulého jména, ale neodvažuje se ji zveřejnit... Aby si pan Šílený mohl svou teorii ověřit, potřeboval by nalézt největší stavbu. A to je již úkol pro vás.

Navrhněte algoritmus a napište program, který dostane na vstupu letecký snímek oblasti a na výstup vypíše souřadnice rohů největší stavby. Snímek je popsán dvěma čísly M a N (rozměry snímku) a maticí $M \times N$ jejíž hodnoty jsou `x` (místo s troskami budovy) nebo `-` (volný prostor). Předpokládejte, že snímek je vyfocen tak, že jeho strany jsou rovnoběžné s triběžkami respektive trileďníky. Pozor! Jednotlivé budovy se mohou překrývat.

Příklad: Na snímku 8×5

```
--x-x---
--xxx-x-
-xxxxxx-
--xxxxx-
--xxxx--
```

má největší stavba vrcholy v bodech $(1, 3)$, $(4, 3)$ a $(4, 6)$.

14-4-5 Turingovy stroje — 10 bodů

V minulé sérii jste dokazovali, že Turingův stroj s jednou páskou dokáže spočítat přesně ty úlohy, které dokáže spočítat Turingův stroj s více páskami. Co když ale zkusíme možnosti našeho Turingova stroje omezit ještě více? Ukazuje se, že například omezení počtu stavů na dva stále sílu (tedy množinu řešitelných úloh) Turingova stroje nezmění. Také když Turingovu stroji dovolíme v každém kroku udělat pouze jednu z obvyklých činností (tzn. přepsat písmeno, změnit stav, posunout hlavu), zůstává jeho síla nezměněna (pokud bychom ale stroji povolili pouze dva stavy a současně v každém kroku pouze jednu z činností, byla by už síla zmenšena).

Nyní si představme, že Turingův stroj pracuje nad skutečnou děrnou páskou. Ta se vyznačuje tím, že pokud na nějaké políčko zapíšeme znak A, tak

na toto políčko už můžeme zapsat pouze znak $\star$ (tomuto znaku se říká „kaňka“). Tedy jediné povolené přepisy jsou $\Lambda \to A$ a $A \to \star$, kde $A \in \Sigma$. Otázkou pro vás je: Může takto omezený jednopáskový stroj vyřešit stejné úlohy jako standardní jednopáskový Turingův stroj? Protože takovýto stroj pochopitelně nemůže zapisovat výstup na místo původního vstupu, bude nám stačit, pokud se výstup vyskytne někde na pásce a ostatní políčka budou zakaňkovaná.

14-5-1 Nové slunce — 12 bodů

V Kocourkově se rozhodli, že již nadále nebudou snášet tyranii slunce, které když se mu chce, tak svítí, a když se mu nechce, tak se zaběhne schovat za mraky. Městská rada se usnesla, že město si pořídí slunce vlastní a lepší, které bude svítit tehdy, kdy to kocourkovští potřebují. Záhy ale ctihodný kocourkovský občan pověřený stavbou nového slunce zjistil, že osvítit celou zem není tak jednoduché a idea s nošením světla v pytlích se při dřívějších pokusech neodsvědčila. Proto se po delším uvažování rozhodl, že bude bohatě stačit, když jeho slunce osvítí celý Kocourkov. I pak ale zjistil, že oblast, kterou musí osvítit, není tak malá, a proto by potřeboval nalézt takové umístění pro slunce, aby poloměr osvícené oblasti mohl být co nejmenší. A s tímto problémem se obrátil na vás.

Na vstupu máte dáno N, počet domů v Kocourkově, a dále souřadnice jednotlivých domů x_i, y_i, kde $1 \leq i \leq N$ (pro naše účely budeme domy považovat za body). Vaším úkolem je nalézt střed kružnice s nejmenším poloměrem, která obsahuje všechny zadané domy. Pokud je takových kružnic více, stačí nám libovolná z nich.

Příklad: Pro 4 domy ležící na souřadnicích $(1, 0)$, $(1, 1)$, $(1, -1)$, $(-1, -1)$ je nejlepší umístit lampu na souřadnice $(0, 0)$.

14-5-2 Tramvaje — 10 bodů

Dvoukvítek při jednom ze svých turistických dobrodružství přepadl přes Okraj a padal a padal až dopadl na Zem. A nedopadl jen tak ledaskam, ale přímo do San Francisca, kde zrovínka dostavěli novou síť tramvajových tratí. Tratě ve městě tvoří čtvercovou síť o rozměrech $M \times N$. Každá linka tramvaje tedy jezdí buď severo-jižním nebo východo-západním směrem z kraje města až na kraj (pochopitelně v obou směrech). Po každé trati jezdí v intervalu i minut za sebou tramvaje a jízda okolo jednoho bloku (tj. jízda mezi dvěma křiženími) jim trvá d minut. Dvoukvítka by nyní zajímalo, jak se má z křižovatky o souřadnicích (a, b) ($(0, 0)$ je severozápadní roh města) co nejrychleji dostat na křižovatku o souřadnicích (c, d). A to je již úkol pro vás.

Navrhněte algoritmus a napište program, který dostane na vstupu čísla M, N, i, d, p, t (p je počet minut, které Dvoukvítek potřebuje na přestup mezi

dvěma tramvajemi a t je čas (počet minut od půlnoci), kdy Dvoukvítek vyráží na cestu) a dále souřadnice počáteční křižovatky (a, b) a souřadnice cílové křižovatky (c, d) a vypočte pro Dvoukvítka nejrychlejší cestu mezi těmito dvěma křižovatkami. Předpokládejte, že o půlnoci všechny tramvaje vyjíždí z okraje města.

Příklad: Pro město 3×5, interval tramvají $i = 22$, dobu jízdy $d = 20$, čas přestupu $p = 2$ a čas startu $t = 19$ je pro Dvoukvítka pro cestu z $(1, 1)$ do $(2, 5)$ nejlepší z křižovatky $(1, 1)$ jet na východ na $(1, 5)$, tam přestoupit a pokračovat na jih až do cílové zastávky $(2, 5)$.

14-5-3 Zapeklitý kabel — 9 bodů

Jednoho dne Lucifer usoudil, že je třeba jít s dobou a neposkytovat své služby pouze prostřednictvím osobních kontaktů, ale i na základě telefonických objednávek. Ke splnění tohoto cíle bylo ale třeba vybudovat telefonní linku do pekla. Tento náročný úkol byl svěřen některým řešitelům dobře známé společnosti Shumm & Brumm. Když byl již celý kabel do pekla položen, zjistil technik, který měl telefony zapojit, že neví, který z konců $a_1, \ldots, a_k$ na jedné straně kabelu patří ke kterému konci $b_1, \ldots, b_k$ na druhé straně kabelu. Aby mohl telefony správně zapojit, potřeboval by technik znát, který konec drátu patří ke kterému. Naštěstí s sebou měl dostatek drátu a zkoušečku, a tak mohl na jedné straně libovolně mnoho drátů propojit a na druhé straně pak změřit, na kterých drátech bude napětí, když přivede napájení na nějaký drát. Protože ale cesta do pekel není vůbec příjemná, byl by technik rád, abyste mu poradili, jak zjistit na co nejméně cest mezi konci kabelu, který konec drátu ke kterému patří.

Navrhněte algoritmus a napište program, který dostane na vstupu počet drátů N a vždy vypíše, jaké dráty má technik na jedné straně propojit, a pak mu poradí, jaké dvojice drátů na druhé straně proměřit. Po zadání výsledků měření pak program technikovi poradí nové zapojení a další měření a tak dále, dokud si nebude jistý, který konec patří ke kterém. Toto přiřazení pak vypíše.

Příklad: Pro 6 drátů by rady mohly vypadat následovně: Propojit na jedné straně a_1–a_2, a_2–a_3 a a_4–a_5. Proměřit na druhé straně všechny dvojice. Technik zadá, že při připojení na b_1 bylo napětí na b_6, při připojení na b_2 bylo napětí na b_4 a b_5 a při připojení na b_3 nebylo napětí nikde (výsledky dalších měření jsou již těmito určeny a proto je nebudeme uvádět). Navrhneme technikovi propojit a_3–a_4, a_2–a_5, a_2–a_6 a proměřit všechny dvojice. Po technikově sdělení, že při připojení na b_1 bylo napětí na b_3 a b_5, po připojení na b_2 nebylo napětí nikde a po připojení na b_4 bylo napětí na b_6 už víme, že kabely jsou propojeny následovně: a_1–b_2, a_2–b_5, a_3–b_4, a_4–b_6, a_5–b_1 a a_6–b_3.

14-5-4 Turingovy stroje — 10 bodů

Váš úkol v této úloze bude poněkud netradiční: zkuste rozhodnout, zda existuje algoritmus (ten případně alespoň v hrubých rysech popište), který dostane na vstupu jednopáskový Turingův stroj T, nějaký vstup pro něj V a číslo k a rozhodne, zda Turingův stroj na daném vstupu bude potřebovat více než k políček na pásce. Uvědomte si, že T se při práci nad vstupem V může zacyklit a nemusí nikdy skončit! Zkuste váš algoritmus vylepšit tak, že dostane na vstupu pouze T a V a vypíše počet políček, která stroji stačila ke zpracování vstupu (počet políček může být i nekonečný). O funkčnosti vašich řešení (popřípadě neexistenci řešení) se nás pokuste přesvědčit co nejlepším důkazem.

Vzorová řešení

14-1-1 Kostky — Zdeněk Dvořák

Tuto úlohu (jako ostatně skoro všechny v KSP) jde řešit hrubou silou, tzn. probráním všech možností; bohužel se také touto cestou mnoho řešitelů vydalo. 1 bod za takováto řešení jim budiž výstrahou.

Povšimneme-li si toho, že jestliže kostičky k_1 a k_2 nemají stejnou podstavu a k_1 se dá postavit na k_2, pak k_2 se nedá postavit na k_1, vzklíčí v nás podezření, zda by kostičky nešlo nějak uspořádat. Toto podezření snadno potvrdíme ověřením toho, že pokud kostička k_1 jde postavit na kostičku k_2, a ta jde postavit na kostičku k_3, tak jde postavit i přímo k_1 na k_3. Kostičky si tedy můžeme seřadit za sebe tak, že kostička k půjde postavit pouze pod předešlé kostičky (ne nutně všechny). Nyní se nabízí řešení dynamickým programováním:

Kostičky si setřídíme podle šířky podstavy a v nerozhodných případech podle hloubky. Dále si budeme postupně brát kostičky od nejmenší a budeme si počítat, jakou největší věž můžeme postavit tak, aby tato námi zvolená kostička k byla úplně dole. Taková věž je buď k samotná, nebo k, na níž leží nějaká věž v. Kostičku úplně vespod v si označme l. Nyní provedeme dvě jednoduchá pozorování:

1) v musí být největší věž, kterou lze s l ve spodu postavit (jinak bychom si mohli místo ní vzít tu větší).
2) l musí být menší než k, tedy vzhledem k našemu uspořádání již pro ni máme spočítanou výšku této maximální věže.

Postup je tedy jednoduchý – pro kostičku k si projdu všechny kostičky l, které jsou v našem uspořádání před ní, ověřím si, zda se l dá postavit na k a z těch, pro které to jde, si vyberu tu, která je ve spodu největší věže.

Tento postup má časovou složitost $O(n^2)$. Jde to lépe? Místem, které nás zdržuje při výpočtu výšky věže, kterou lze postavit na k, je průchod přes všechny kostky před ní, při němž hledáme tu, kterou na k postavit. Co to přesně znamená? Hledáme kostičku, která je nanejvýš tak široká i hluboká jako k. Nicméně všechny kostičky před k jsou vzhledem ke zvolenému setřídění nanejvýš tak široké jako k. Tedy se tato podmínka redukuje na to, aby tato kostička byla nanejvýš tak hluboká jako k. Zajímá nás tedy maximum nějaké hodnoty (výšky věže) přiřazené číslům (šířkám kostiček) na nějakém intervalu (od hloubky nejmělčí kostičky po hloubku k). Existuje datová struktura, která nám umožňuje na takovéto dotazy odpovídat v logaritmickém čase (a také jsme schopni do ní přidat hodnotu v logaritmickém čase). Jejím použitím dosáhneme optimální časové složitosti $O(n \log n)$. Jak tato struktura vypadá:

Bude nám stačit, aby čísla, jimž budeme přiřazovat hodnoty, byla v rozsahu od 0 do $n-1$ (vzhledem k tomu, že u hloubek nás zajímá pouze uspořádání, si je můžeme snadno přečíslovat). Naši strukturu bude tvořit vyvážený binární strom. Jeho listy si zleva doprava očíslujeme od 0 do $n-1$ (obecně nám nějaké zbydou, pokud n nebude mocnina 2, ale to nám nevadí – zvolíme-li hloubku stromu na $\log n$ (zaokrouhleno nahoru), bude jich nejvýše n). Do těchto listů si budeme ukládat hodnoty odpovídající příslušným číslům. Vnitřní uzly pak budou odpovídat intervalům – konkrétně těm číslům, která leží v jejich podstromu (tedy např. kořen bude odpovídat všem číslům v rozsahu od 0 do $n-1$) a bude obsahovat maximum z jejich hodnot. Nejprve se podívejme, jak bude vypadat změna hodnoty v k-tém prvku; zřejmě tím ovlivníme pouze hodnoty přiřazené listu k a uzlům na cestě z něj ke kořeni. Těch je pouze logaritmický počet a jsme každou z nich schopni spočítat v konstantním čase (jako maximum z hodnot přiřazených synům daného uzlu), tedy nám to zabere slibovaný logaritmický čas.

Nechť tedy chceme zjistit maximum z hodnot v nějakém intervalu I. Začneme v kořeni a podíváme se, zda I leží celý v jednom z intervalů odpovídajících jeho synům. Je-li tomu tak, ten druhý nás nezajímá – přesuneme se tedy do toho prvního a postup opakujeme. Zajímavější je případ, kdy I protíná oba tyto intervaly. Stále ale ještě máme šanci na jednoduché řešení: pokud I obsahuje jeden z těchto intervalů, maximum z hodnot na této části známe rovnou (je uloženo v příslušném uzlu). Stačí se tedy přesunout do druhého intervalu, opakovat postup a na konci vzít maximum ze získaných hodnot. Nicméně se nám může stát, že toto nenastane; v té chvíli nám nezbývá nic jiného, než interval rozdělit na dva a popisovaný postup použít na oba (a samozřejmě na konci vzít větší z výsledků). Mohlo by se zdát, že toto nám může zkazit časovou složitost, ale není tomu tak – snadno nahlédneme, že jakmile jsme jednou provedli rozdělení intervalu, vždy už budou nastávat pouze první dvě možnosti; tedy k rozdělení dojde nanejvýš jednou. Vzhledem k tomu, že první dvě možnosti nás vždy přesunou ve stromu o úroveň níž a potřebují na to konstantní čas, je časová složitost této operace $O(\log n)$.

Ve skutečnosti v programu zneužívám toho, že dotazy jsou vždy speciálního tvaru (na interval od začátku někam). Díky tomu nemůže dojít na třetí případ (dělení intervalu), což program trochu zjednodušuje.

Zbývá se podívat na paměťovou složitost. Ta je lineární (jediným místem, které by nám mohlo něco pokazit, je strom použitý pro zjišťování maxim; nicméně počet vnitřních uzlů binárního stromu je roven počtu listů bez jedné, a počet listů je menší než $2n$.

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

#define MAX 1024
```

```
typedef struct
{
    int x, y, z;
} kostka;

int n, npt;
kostka kostky[MAX];
int itree[2*MAX-1];
/* vraci maximum z hodnot prirazenych cislum v intervalu 0 ... sir-1 */
int best_v (int sir)
{
    int asir=npt/2, ain=0, abest=0;
    while (sir>0)
      {
        if (sir>=asir)
          {
            if (itree[2*ain+1]>abest) abest=itree[2*ain+1];
            ain=2*ain+2;
            sir-=asir;
          }
        else
          ain=2*ain+1;
        asir/=2;
      }
    return abest;
}
/* priradi cislu sir hodnotu vys */
void set_v (int sir, int vys)
{
    int asir=npt, ain=0;
    while (asir>0)
      {
        asir/=2;
        if (itree[ain]<vys) itree[ain]=vys;
        if (sir>=asir)
          {
            ain=2*ain+2;
            sir-=asir;
          }
        else
          ain=2*ain+1;
      }
}

int cmpyx (const void *a, const void *b)
{
    const kostka *aa=a, *bb=b;
    return aa->y-bb->y ?:aa->x-bb->x;
}

int cmpxy (const void *a, const void *b)
{
    const kostka *aa=a, *bb=b;
```

```
    return aa->x-bb->x ?:aa->y-bb->y;
}
int main (void)
{
    int i;
    scanf ("%d", &n);
    for (npt=1; npt<n; npt*=2);
    for (i=0; i<n; i++)
      scanf ("%d%d%d", &kostky[i].x, &kostky[i].y, &kostky[i].z);
    qsort (kostky, n, sizeof (kostka), cmpyx);
    for (i=0; i<n; i++) kostky[i].y=i;
    qsort (kostky, n, sizeof (kostka), cmpxy);
    for (i=0; i<n; i++)
      {
        int mv=best_v (kostky[i].y);
        set_v (kostky[i].y, mv+kostky[i].z);
      }
    printf ("Vyska nejvetsi veze je %d.\n", itree[0]);
    return 0;
}
```

14-1-2 Orientační běh — Miroslav Trmač

Jak bylo možné poznat z bodového ohodnocení, patřila tato úloha k těm snazším, což potvrzuje fakt, že většina došlých řešení úlohu více či méně efektivně řešila (což se bohužel nedá říci o určování časové a paměťové složitosti).

Je evidentní, že chceme udělat průnik ze seznamů ze všech stanovišť. Budeme postupovat takto: Načteme závodníky, kteří proběhli prvním stanovištěm do lineárního spojového seznamu. Pak pro každé další stanoviště projdeme „náš“ seznam a seznam ze stanoviště a z „našeho“ seznamu odstraníme závodníky, kteří neproběhli daným stanovištěm. Protože po zpracování k stanovišť jsou v našem seznamu právě ti závodníci, kteří proběhli všemi prvními k stanovišti, získáme po zpracování všech stanovišť hledaný seznam poctivých závodníků.

Při zpracovávání jednoho stanoviště se „postavíme“ na začátek „našeho“ seznamu, a pak vždy načteme číslo závodníka ze vstupu (A) a porovnáme ho s číslem na aktuální pozici „našeho“ seznamu (B). Mohou nastat tři případy:

1) $A > B$: závodník B neproběhl aktuální stanoviště (protože seznam závodníků ze stanoviště je setříděný), bez milosti ho z „našeho“ seznamu vyřadíme, vezmeme následujícího závodníka z „našeho“ seznamu a znovu jej porovnáváme s A, dokud nenastane některý z následujících případů (tj. dokud z „našeho“ seznamu neodstraníme všechny závodníky s čísly menšími než A).
2) $A = B$: vše je v pořádku, závodník A dosud proběhl všechna stanoviště, v „našem“ seznamu se posuneme na následující pozici.

3) $A < B$, nebo jsme na konci „našeho“ seznamu: závodník A neproběhl předchozí stanoviště, takže na něj můžeme zapomenout a zpracovávat dalšího.

Když takto zpracujeme celý seznam z aktuálního stanoviště, ještě nezapomeneme z „našeho“ seznamu odstranit všechny závodníky od aktuální pozice dále, protože ti také neproběhli aktuálním stanovištěm (toto je také odstraňování prvků, takže dále jej zahrneme pod první případ).

Tím jsme získali průnik dvou vzestupně setříděných seznamů. Jednodušeji se to dá říci tak, že máme-li dva vzestupně setříděné seznamy a jejich první prvky se nerovnají, menší z těchto prvků v druhém seznamu jistě není a můžeme jej tedy odstranit, stejně jako odstraníme prvky v jednom seznamu, je-li druhý prázdný. Ten složitější popis se nám bude ale hodit pro určování časové složitosti získání tohoto průniku:

Má-li seznam z k-tého stanoviště celkem N_k závodníků, načetli jsme N_k čísel a s každým z nich jsme prošli právě jednou případem 2) nebo 3). Případem 1) jsme mohli obecně projít vícekrát, ale při každém výskytu případu 1) rušíme jeden prvek z „našeho“ seznamu, tedy případ 1) se za celou dobu běhu programu může vyskytnout nejvýše tolikrát, kolik v něm na začátku bylo prvků, to je N_1-krát.

Celková časová složitost je $O(N_1)$ pro zpracování prvního stanoviště a $O(N_2 + \cdots + N_M)$ pro případy 2) a 3) (je-li M počet stanovišť) a $O(N_1)$ pro případy 1). Ještě nesmíme zapomenout, že pro každé stanoviště musíme udělat trochu práce (přejít na začátek „našeho“ seznamu), i když je seznam ze stanoviště prázdný (což by se také mohlo stát), a započítat proto ještě $O(M)$. Pokud označíme P počet čísel na vstupu, přičemž za číslo považujeme i -1, kterou ukončujeme seznam každého stanoviště, je časová složitost $O(P)$. Paměťová složitost je zjevně $O(N_1)$.

Vzorové řešení používá jediný netriviální obrat: potřebujeme odebírat prvky z aktuální pozice v seznamu. K tomu pro aktuální pozici potřebujeme vědět, odkud na ni ukazuje předchozí prvek nebo hlava seznamu. Dá se použít obousměrný seznam nebo si pamatovat prvek před aktuální pozicí (pak bychom museli začátek seznamu ošetřit zvlášť nebo na začátek seznamu vložit „falešný“ prvek), vzorové řešení používá ukazatel na ukazatel. Proměnná `lp` ukazuje na ukazatel na aktuální prvek v seznamu, `*lp` je tedy ukazatel na aktuální prvek.

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

struct entry
{
    struct entry *next;
    int val;
```

```
};
int
main (void)
{
  unsigned seqs;
  struct entry *list, **lp;

  scanf ("%u", &seqs);
  lp = &list;
  for (;;)                                  /* Prvni posloupnost nacteme celou */
    {
      int val;
      struct entry *e;

      scanf ("%d", &val);
      if (val < 0)
        break;
      e = malloc (sizeof (*e));
      e->val = val;
      *lp = e;
      lp = &e->next;
    }
  *lp = NULL;
  while (--seqs != 0)
    {
      lp = &list;
      for (;;)
        {
          int val;

          scanf ("%d", &val);
          if (val < 0)
            break;
          while (*lp != NULL && (*lp)->val < val)
            *lp = (*lp)->next;
          if (*lp != NULL && (*lp)->val == val)
            lp = & (*lp)->next;
        }
      *lp = NULL;                           /* Zahodime zbyly konec seznamu */
    }
  while (list != NULL)
    {
      printf ("%d%c", list->val, list->next != NULL ? ' ' : '\n');
      list = list->next;
    }
  return EXIT_SUCCESS;
}
```

14-1-3 Elektrická vedení — Pavel Šanda

Zločinecký syndikát opět zasáhl a k naprostému zděšení inženýrů AEZu se z aletonských podzemních krčem rozšířil po planetě drb o plánovaném rozšiřování servisních stanic na kulábrová centra. Není se pak čemu divit, že velké množství došlých návrhů na rozmisťování SS si počínalo poněkud neskromným způsobem. Největší experti navrhovali dokonce umisťování po celém Aletonu krom okrajových stanic; pozůstatek pozemské sofistické školy započal vychytrale obdobné návrhy vylepšovat rozpustilými heuristikami, kdy ze kterého vrcholu SS odstranit. Když vedení AEZu přešly první záchvaty entuziastického vymýšlení protipříkladů, rozhodlo se brát opravdu vážně jen návrhy, které zdůvodnily, proč navržená heuristika najde v skutku optimum (mnozí se zjevně řídili pravidlem *Je-li to kopyto – platí to.* Ku podivu však poté zapomněli kopyto dokázat.)

Vzrůstající tradice foliantismu si i pro tentokrát našla své příznivce, kteří bohům díky podnes nechávají heuristikám spát. Listy si rozdělili na tré druhů – listy nepokryté (LN), listy pokryté sousedním servisem (LP) a listy servisní (LS).

Je-li LN, můžeme ho pokrýt pouze dvojím způsobem – vložením SS do LN nebo do jeho souseda – je však patrné, že varianta se sousedem je výhodnější. Po vložení je pro nás list nezajímavým (v řešení už nebude hrát roli) – můžeme ho proto odtrhnout.

Je-li LP, je taktéž nezajímavý a rovnou trháme.

Je-li LS, je třeba pouze dohlédnout na to, aby jeho sousedé nebyli označeni za nepokryté (to budeme dělat při umisťování SS) a dál už nás také nezajímá.

A teď přijde kouzlo – protože máme zadáním zaručeno, že graf je stromem, odtržením listu vznikne nový list, se kterým víme, co si počít – nemusíme už tedy vymýšlet nic dalšího.

Postup je krom starání se o pokrytost téměř identický s úlohou 13-1-2 – naleznu všechny listy. S každým naložím podle postupu uvedeného výše. Vzniknou nové listy a celý cyklus bude opakován, dokud je co trhat. Výsledkem pak pro nás není zbylé centrum stromu, nýbrž všechny vrcholy označené za servisní.

Implementace: Fronta, do které si postupně ukládám listí určené k utrhnutí. Při každém utržení vkládám do fronty nově vzniklé listy. Ke každému vrcholu si navíc vedu záznam, jakého je typu (*pokryt*).

Čas a paměť: Na každý vrchol sáhnu jednou – celkem tedy $O(n)$ – a u každého vrcholu hledám sousedy – stromeček má v **celém** grafu lineární počet sousedů, tedy $+O(n) = O(n)$. Paměťová složitost je kvadratická, leč jako obvykle bychom byli schopni ji převést na lineární. A přesně v ten slavný den, kdy k nám byl zaveden elektrický proud, byla ukázána konečnost, ježto v každém kroku ubereme jeden z konečného počtu vrcholů.

```
#include <stdio.h>
#define Max 20

int v[Max][Max];                        /* Sousede vrcholu */
int deg[Max];                           /* Stupne vrcholu */
int ndeg[Max];                          /* Nove stupne vrcholu (po odebrani) */
int queue[Max];                         /* Odtrzeni=inkremenatace na nulu */
int pokryt[Max];                        /* 0=nepokryto,1=pokryto,2=SS */

int pokryj (int x) {                    /* vlozi servisni stredisko a pokryje sousedy */
  if (pokryt[x]==2)
    return 1;
  else
    pokryt[x]=2;
  for (int i=0; i<deg[x]; i++)
    if (!pokryt[v[x][i]]) pokryt[v[x][i]]=1;
}                                       /* pokryj */

int main (void) {
  int n, first=0, last=0, i, x, y;      /* pocet vrcholu,ukazatele do fronty */
  scanf ("%d", &n);

  for (i=0; i<n-1; i++){                /* E=V-1, vrcholy cislovany od 0  */
    scanf ("%d %d", &x, &y);
    v[x][deg[x]++, ndeg[x]++]=y;        /* nastav sousedy a stupne vrcholu */
    v[y][deg[y]++, ndeg[y]++]=x;
  }                                     /* for */

  for (i=0; i<n; i++) if (deg[i]==1) queue[last++]=i;    /* vsechny listy do fronty */

  while (last>first) {
    x=queue[first++]; ndeg[x]--;        /* dalsi list na utrzeni */
    if (!pokryt[x])
      for (i=0; i<deg[x]; i++)
        if (ndeg[v[x][i]])
          pokryj (v[x][i]);

    for (i=0; i<deg[x]; i++)
      if (ndeg[v[x][i]])                /* neodtrzeno? */
        if (--ndeg[v[x][i]] == 1)
          queue[last++]=v[x][i];
  }                                     /* while */
  for (i=0; i<n; i++) if (pokryt[i]==2) printf ("%d", i);
  printf ("\n");
  return 0;
}                                       /* main */
```

14-1-4 k-Klidná posloupnost — Tomáš Valla

Hledejme nejdříve délku nejdelší k-klidné podposloupnosti. Řešení si ukážeme ve třech krocích; začneme poměrně jednoduchým algoritmem, který budeme postupně vylepšovat.

Krok 1. Použijeme myšlenku dynamického programování. Zavedeme si celočíselné pole P délky n, jehož i-tý prvek bude udávat délku nejdelší vybrané k-klidné podposloupnosti začínající i-tým prvkem vstupní posloupnosti. Poslední prvek vstupu je sám o sobě podposloupností, tedy P_n nastavíme na 1. Teď se budeme postupně za každý prvek $a_{n-1}, a_{n-2}, \ldots, a_1$ pokoušet napojovat co možná nejdelší k-klidnou podposloupnost. Když už ale máme spočítané nejdelší podposloupnosti od a_i napravo, prostě vybereme maximum z těch P_j od P_i napravo, kde se a_i a příslušné a_j neliší více než o k, (když nic napojit nejde, maximum vyjde 0) a uložíme ho zvýšené o jedničku.

Po konci výpočtu najdeme v P maximum, které je řešením. Správnost algoritmu je zřejmá už z popisu, šťouralové si ho mohou formálně dokázat indukcí. Jak je to s časovou složitostí: provádí se $1 + 2 + 3 + \cdots + (n-1)$ operací, což se napočítá na $O(n^2)$. Paměťová složitost je lineární, pamatujeme si P a vstup.

Krok 2. Jak je vidět, zbytečně jsme testovali i prvky, jejichž rozdíl je příliš velký. Když si nyní vstupní posloupnost setřídíme, budeme mít prvky lišící se o méně než k hned vedle sebe. Zase od konce vyhledáme aktuální a_i půlením intervalů v setříděné posloupnosti S a podíváme se nalevo i napravo tak daleko, dokud se nám prvky nezačnou lišit o více než k, a z nich vybereme maximum.

Na začátku si P nainicializujeme na nuly; maximum budeme zkoumat z P_k nalevo a napravo od vyhledaneho P_j včetně P_j, to když už se hodnota a_i někde vyskytla. S vynulovaným P se nemusíme ani starat o to, jestli je zkoumaný prvek skutečně napravo od a_i.

Časová složitost je $O(nk + n \log n)$, posloupnost setřídíme v $O(n \log n)$, n-krát v logaritmickém čase vyhledáme číslo v poli a prozkoumáme až $2k$ sousedů. (Při použití tohoto algoritmu je ještě třeba nechat v S jen jeden prvek téže hodnoty, trochu se pak zkomplikuje výpis posloupnosti.)

Krok 3. Pro jednoduchost předpokládejme, že n je mocninou dvojky a postavme si nad S dokonale vyvážený binární strom. Jeho listy tvoří hodnoty z P, ale v pořadí posloupnosti S, v jeho vrcholech je maximum celého podstromu, tedy větší ze dvou čísel o hladinu níže. Všimněte si nyní, že každý interval v S lze rozdělit na úseky, jejichž délka je mocninou dvojky, které přesně odpovídají velikostem podstromů jednotlivých uzlů. Protože nejhorší možné rozdělení intervalu na úseky (tyto úseky nazveme *pokrytím intervalu*) je $1, 2, 4, \ldots, 4, 2, 1$, je počet úseků (tedy vlastně počet vrcholů, z nichž získáme maximum celého intervalu) úměrný logaritmu velikosti intervalu.

Binárním hledáním v S nalezneme první prvek intervalu $\langle a_i - k; a_i + k\rangle$ a poslední prvek intervalu (z S není třeba vyhazovat duplikáty, binární hledání najde k hodnotě vždy jen jeden index); promítneme si je jako l a r do spodní hladiny stromu T. Nyní budeme postupně interval ořezávat tak, že visí-li l na pravé větvi, zahrneme ho do pokrytí a posuneme v hladině o prvek doprava, visí-li r na levé větvi, zahrneme ho do pokrytí a posuneme se doleva. Pak se posuneme s l a r po příslušných hranách o hladinu výše a celý proces opakujeme, dokud se nám l s r nepotkají.

Po logaritmicky mnoha krocích tedy nalezneme maximum z intervalu. Po updatu P_i je ještě třeba správně obnovit strom, to však zvládneme postupným průchodem od odpovídajícího listu ke kořeni taktéž v logaritmickém čase. Celkový čas bude tím pádem $O(n \log n)$. Strom má $2n$ uzlů, paměť zůstane $O(n)$.

Ještě zbývá vypsat nejdelší podposloupnost. Pokud jsme plnili P_j odzadu pro j od n k jedné, bude mít první prvek nejdelší podpousloupnosti maximální hodnotu v P. Všimněte si, že následující prvek je libovolný takový, který se liší o méně než k a hodnotu v P má o jedna nižší. Nepotřebujeme si pamatovat pole předchůdců ani obracet výstup, postačí jediný průchod polem P.

Pár slov k programu: n si zarovnáme na nejbližší vyšší mocninu dvojky. Pak budeme moci strom uložit do pole a indexovat ho podobně jako haldu, tj. od 1, na začátku pole budou vyšší hladiny, ke konci nižší. Princip ořezávání je stejný jako v popisu, ale l je index prvního prvku intervalu a r je index prvního prvku *za* intervalem. l a r jsou indexy v S a protože vrcholy stromu bez dolní hladiny včetně nultého nepoužívaného prvku zabírají právě n míst, přičtením n dostanu index v poslední hladině stromu. S tímto číslováním, jsou-li l a r lichá, je třeba zahrnout je do pokrytí. O hladinu výše se posuneme vydělením l a r dvojkou. Binární hledání funguje na principu nahazování bitů od nejvyššího k nejnižšímu.

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

#define MAX 1024

static int N, n;                                /* N puvodni delka, n nejblizsi vyssi
                                                   mocnina 2 */
static int S[MAX], T[2*MAX], A[MAX], P[MAX];

static inline int max (int p, int q)
{
        return (p>q) ? p : q;
}

/* najde index posledniho prvku v S, ktery je <= v */
static int find_last_le (int v)
{
```

```
        int i=0, x=n;
        while ( (x /= 2) >= 1)
                if (S[i+x] <= v) i += x;
        return i;
}
static int cmp (const void *X, const void *Y)
{
        int x = *(int *)X;
        int y = *(int *)Y;
        return (x>y) - (x<y);
}
int main (void)
{
        int i, j, k, m, l, r;
        scanf ("%d%d", &k, &N);
        n=1;
        while (n < N) n *= 2;                   /* najit nejblizsi mocninu 2 */
        for (i=0; i<N; i++) scanf ("%d", &A[i]);
        memcpy (S, A, N*sizeof (int));
        qsort (S, N, sizeof (int), cmp);
        while (i<n) S[i++] = S[N-1];    /* zbytek S zaplnit velkym cislem */
        for (i=N-1; i>=0; i--) {
                l = find_last_le (A[i]-k) + n;
                if (S[l-n] < A[i]-k)
                        l++;                    /* dorovnat na prvni prvek */
                r = find_last_le (A[i]+k) + n+1;
                m = 0;
                while (l<r) {                   /* maximum z pokryti */
                        if (l&1) m=max (m, T[l++]);
                        if (r&1) m=max (m, T[--r]);
                        l/=2; r/=2;
                }
                P[i] = ++m;                     /* update stromu */
                for (j=find_last_le (A[i])+n; j; j/=2)
                        T[j]=max (T[j], m);
        }
        j=-1;
        for (i=j; i<N; i++)                     /* T[1] je max. delka podposloupnosti */
                if (P[i] == T[1] && (j==-1 || (A[j]-A[i] <= k && A[i]-A[j]
                    <= k))) {
                        printf ("%d ", A[i]);
                        j=i;
                        T[1]--;
                }
        putchar ('\n');
        return 0;
}
```

14-1-5 Turingovy stroje — Jan Kára

Většina z vás zvládla oba Turingovy stroje více či méně elegantně zkonstruovat v optimální složitosti. Na co jste občas zapomínali, bylo, že vstupní slovo může být i prázdné (tedy neobsahovat žádné znaky) a hlavu Turingova stroje jste v tom případě poslali hledat v nekonečnu pravý konec pásky, případně jste stroj donutili bezradně kroutit hlavou nad nedefinovaným přechodem.

Stroj obracející slovo z nul a jedniček bude pracovat následovně: Nejdříve hlava při průchodu zleva doprava posune celé slovo o jedno políčko vpravo (stavy q_0, q_{p0} a q_{p1}). Potom při návratu nabere poslední číslici (tu vlastně stavy q_{p0} a q_{p1} ani nikdy neuložily), umístí ji na první místo (přejezd zajišťují stavy q_{v0} a q_{v1}), přesune se nad druhé políčko a celý cyklus opakuje. Po prvním cyklu se tedy na správném místě ocitne poslední číslice, po druhém předposlední až po n-tém cyklu bude správně umístěna i první číslice. Stroj potřebuje $O(n^2)$ kroků (probíhá přes $n + (n-1) + (n-2) + \ldots + 1 = n \cdot (n+1)/2 = O(n^2)$ políček) a na pásce zabere $O(n)$ políček.

	0	**1**	Λ
q_0	q_{p0}, Λ, R	q_{p1}, Λ, R	q_k, Λ, L
q_{p0}	$q_{p0}, \mathbf{0}, R$	$q_{p1}, \mathbf{0}, R$	q_{v0}, Λ, L
q_{p1}	$q_{p0}, \mathbf{1}, R$	$q_{p1}, \mathbf{1}, R$	q_{v1}, Λ, L
q_{v0}	$q_{v0}, \mathbf{0}, L$	$q_{v0}, \mathbf{1}, L$	$q_0, \mathbf{0}, R$
q_{v1}	$q_{v1}, \mathbf{0}, L$	$q_{v1}, \mathbf{1}, L$	$q_0, \mathbf{1}, R$
q_k	$q_k, \mathbf{0}, L$	$q_k, \mathbf{1}, L$	—

A nyní stroj na půlení slova: Stroj bude pracovat tak, že vždy odmaže jednu jedničku z počátku slova, pak přejede na konec slova a zde též odmaže jednu jedničku. Pak se hlava vrací na počátek slova a odmazávání se opakuje. Po odmazání všech číslic bude hlava uprostřed slova a stačí tedy vlevo od ní dopsat jedničky. Stroj potřebuje $O(n^2)$ kroků a paměť $O(n)$. Tento algoritmus jsem v řešení zvolil proto, že ho lze velmi snadno upravit i na případ, kdy abeceda bude obsahovat více znaků, než jen 1 a Λ. V tom případě totiž můžeme znaky z počátku slova místo mazání „čárkovat“ – přidáme si do abecedy nová písmena `a', b', c'`... a znak `a` budeme přeznačovat na `a'`, `b` na `b'` atd.

	1	Λ
q_0	q_r, Λ, R	q_k, Λ, L
q_r	$q_r, \mathbf{1}, R$	q_m, Λ, L
q_m	q_l, Λ, L	—
q_l	$q_l, \mathbf{1}, L$	q_0, Λ, R
q_k	—	$q_k, \mathbf{1}, L$

14-2-1 Kyvadlo — Miroslav Trmač

Většina řešitelů se rozhodla cestu provázku prostě nasimulovat. Tento algoritmus si v každém kroku pamatuje délku zbylé části provázku, poslední zpracovaný kolík A a směr, ve kterém jsme do něj přišli. Pak mezi všemi kolíky, na které dosáhneme z A se zbylou částí provázku (s výjimkou A samotného), hledáme ten kolík B, který je z nich nejvíce vlevo vzhledem ke směru, ve kterém jsme přišli do A. Pokud je takových víc, vybereme ten vzdálenější od A. Délku zbylé části provázku zkrátíme o vzdálenost kolíků A a B a postup opakujeme, tentokrát ale za kolík A budeme považovat nalezený kolík B. Algoritmus končí, jakmile z aktuálního kolíku nedosáhneme na žádný jiný.

Tento algoritmus zřejmě řeší úlohu, protože přesně simuluje pohyb provázku, je konečný, protože v každém kroku se zmenší délka zbylé části provázku. Časová složitost je $O(KN)$, kde N je počet kolíků a K je celkový počet „zalomení“ provázku okolo kolíků, protože v každém z K kroků musíme projít všechny ostatní kolíky. Paměťová složitost je $O(N)$, protože právě tolik místa je třeba na uložení pozic všech kolíků.

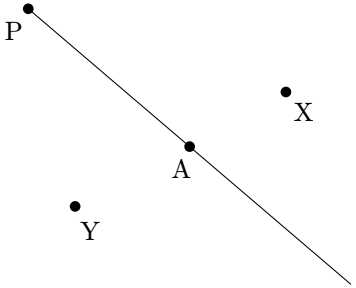
My si zde ukážeme dvě zlepšení tohoto řešení, díky nimž bude časová složitost v nejhorším případě $O(N^2)$. Zaprvé nebudeme opakovaně počítat pořád to samé a ke každému kolíku si budeme pamatovat jeho „následníka“. Jestliže přijdeme do některého kolíku a na jeho následníka už nedosáhneme, provedeme „opravu“ a najdeme nového následníka v dosahu. Podle tohoto algoritmu uděláme celkem K kroků po „cestě“ mezi kolíky, každému z až N kolíků najdeme poprvé následníka v čase $O(N)$.

Ještě spočítáme, kolikrát budeme následníka „opravovat“. V případě, že hledáme nového následníka ke kolíku A, protože jeho starý následník B je moc daleko, víme, že do kolíku B se už nikdy nedostaneme (neexistuje kratší cesta než po přímce), tedy kolík B navždy vypadává ze hry. Proto se při každé „opravě“ následníka počet „použitelných“ kolíků zmenší o jedna, tedy oprav bude nejvýše $O(N)$, každá z nich trvá $O(N)$. Časovou složitost jsme tedy zlepšili na $O(N^2 + K)$.

Druhé zlepšení taktéž eliminuje opakované výpočty. Je-li totiž provázek dost dlouhý na to, aby kolem kolíků „oběhl“ několikrát, tak náš algoritmus otrocky počítá každý krok. Tomu se vyhneme tak, že si ke každému kolíku poznamenáme délku zbylé části provázku v době, kdy jsme jej navštívili naposledy. Pokud se pak do tohoto kolíku vrátíme, můžeme snadno zjistit, jak dlouhý byl tento „cyklus“. Místo abychom tento cyklus opakovali, můžeme prostě zbylou délku provázku zkrátit o největší možný násobek délky cyklu. Tím získáme délku provázku, která je menší, než délka tohoto cyklu, tedy alespoň jeden z kolíků na tomto cyklu už dále nebude dosažitelný. Proto takových cyklů objevíme $O(N)$. Protože objevení jednoho cyklu nastane nejdéle po N krocích, je celkový počet kroků nejvýše $O(N^2)$.

Pro pořádek zrekapitulujme všechny členy výsledné časové složitosti: hledání prvního následníka ke každému kolíku bude trvat $O(N^2)$, všechny „opravy“ dohromady $O(N^2)$, všechny „kroky“ po kolících celkem $O(N^2)$. Výsledná časová složitost je evidentně $O(N^2)$.

Ještě bychom měli dokázat věc, kterou jsme používali při zlepšeních: totiž že si údaje (následníka a „čas“ poslední návštěvy) můžeme pamatovat ke každému kolíku a že tyto údaje nezávisí na tom, kterým směrem jsme do daného kolíku přišli (jinými slovy, v daném „cyklu“ se každý kolík vyskytuje právě jednou). Pokud si rozmyslíme případ tří kolíků na jedné přímce, zjistíme, že toto tvrzení neplatí. To ale můžeme snadno „opravit“: všechny vektory od daného kolíku A rozdělíme na dvě části tak, že opačné polopřímky s počátečním bodem A nikdy „nespadnou“ do stejné části a údaje si budeme pamatovat podle toho, z které části jsme přišli do kolíku A. Dá se již dokázat, že údaje si můžeme pamatovat pro každou dvojici kolík-část.



Pomůžeme si obrázkem. Nechť jsme do kolíku A přišli z kolíku P. Pak víme, že v polorovině PAX (s výjimkou přímky PA) není žádný kolík dosažitelný z A (jinak by byl dosažitelný i z P). Následující kolík B tedy musí být v polorovině PAY, a to buď mimo přímku PA, nebo na ní.

Jestliže B je mimo přímku PA, jsou všechny kolíky dosažitelné z A v úhlu BAP (včetně hraničních polopřímek), do kolíku A tedy můžeme znovu přijít jedině z tohoto úhlu. Pokud ale přijdeme z tohoto úhlu, je kolík následující po A kolík B, tedy jsme si jej zapamatovali správně.

Zbývá případ, kdy B je na přímce PA. Jestliže B leží na polopřímce AP, není co řešit, protože existuje jediný směr, jak můžeme přijít do A. Problém nastává právě v situaci, kdy B leží na polopřímce opačné k AP, kdy můžeme do bodu A přijít z bodu P nebo B a pokaždé máme jiného následníka. Protože jsme ale vektory od A rozdělili tak, že opačné polopřímky nejsou v jedné části, pamatujeme si pro každý z těchto směrů následníka zvlášť, a nic jsme tedy nepokazili.

Ještě pár slov k technické realizaci, protože analytická matematika je oblastí, kde se často programuje poměrně obtížně, i když to tak ze zadání nevypadá.

Při hledání kolíku „nejvíce vlevo“ od kolíku A bychom mohli určit úhel spojnice AK, kde K je zkoumaný kolík, od směru, kterým jsme přišli. K tomu se výborně hodí funkce `atan2(double y, double x)`, která spočítá $\arctan y/x$, ale ošetřuje případy, kdy $x = 0$, a podle znamének parametrů umístí výsledný úhel do správného kvadrantu.

Můžeme ale postupovat jednodušeji: budeme hledat kolíky, které jsou vlevo od směru, ve kterém jsme přišli, a zároveň jsou vpravo od našeho dosud nejlepšího kandidáta na následující kolík. To, jestli jde vektor v nalevo nebo napravo od vektoru u, zjistíme jednoduše pomocí vektorového součinu: Budeme-li u a v považovat za vektory v prostoru, bude souřadnice z součinu $u \times v$ menší než, větší než, resp. 0 podle toho, jestli je v vpravo nebo vlevo od u, resp. jestli jsou u a v rovnoběžné. Jsou-li u_1, u_2 souřadnice vektoru u a v_1, v_2 souřadnice vektoru v, pak hledaná souřadnice z jejich vektorového součinu je $u_1 \cdot v_2 - u_2 \cdot v_1$.

Poslední poznámka: počítání s čísly v plovoucí řádové čárce není přesné, takže je vhodné dát procesoru trošku „volnost“ ve výsledku, např. místo zjišťování, jestli je výsledek rozdílu 0, zjišťovat, jestli absolutní hodnota rozdílu je menší než zvolená (zvolená tak malá, že nenulový rozdíl tak „nemůže“ vyjít).

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

#define EPSILON 1.0e-15

#define MAXN 1024

struct point
{
    long double x, y;
    struct point *next[2];
    long double last_time[2];
};

#define NO_LAST_TIME -1.0L

struct point points[MAXN];

int num_points;

long double
dist (const struct point *a, const struct point *b)
{
    return hypotl (a->x - b->x, a->y - b->y);
}

long double
product (const struct point *s1, const struct point *e1,
         const struct point *s2, const struct point *e2)
{
    return (e1->x - s1->x) * (e2->y - s2->y) - (e1->y - s1->y) * (e2->x -
        s1->x);
}
```

```
struct point *
find_next (const struct point *prev, const struct point *last,
           long double length)
{
  struct point *p, *best;

  best = NULL;
  for (p = points + 1; p < points + num_points; p++)
    {
      if (p == last || dist (p, last) > length + EPSILON)
        continue;
      if (product (prev, last, prev, p) <= 0.0L)
        {
          if (best != NULL)
            {
              long double diff;

              diff = product (last, best, last, p);
              if (fabsl (diff) > EPSILON && diff < 0.0L)
                continue;
              if (fabsl (diff) <= EPSILON)
                {
                  /* last, best a p jsou na jedne primce, tedy je na teto primce i
                     prev. Pokud mozno chceme vybrat bod ve smeru prev-¿last,
                     vybereme tedy bod vzdalenejsi od prev. */
                  if (dist (prev, best) > dist (prev, p))
                    continue;
                }
            }
          best = p;
        }
    }
  return best;
}

int
main (void)
{
  struct point *p, *top;
  const struct point *prev;
  long double length;

  scanf ("%u", &num_points);
  num_points++;
  top = NULL;
  for (p = points + 1; p < points + num_points; p++)
    {
      scanf ("%Lf%Lf", &p->x, &p->y);
      p->next[0] = NULL;
      p->next[1] = NULL;
      p->last_time[0] = NO_LAST_TIME;
      p->last_time[1] = NO_LAST_TIME;
```

```
        if (top == NULL || top->y < p->y || (top->y == p->y && top->x <
            p->y))
          top = p;
      }
    scanf ("%Lf", &length);
    p = top;
    points[0].x = p->x - 1;
    points[0].y = p->y;
    prev = points;
    for (; ; )
      {
        struct point **next;
        int type;
        long double *last;

        if (p->y > prev->y || (p->y == prev->y && p->x > prev->x))
          type = 0;
        else
          type = 1;
        next = p->next + type;
        if (*next == NULL || dist (*next, p) > length + EPSILON)
          {
            *next = find_next (prev, p, length);
            if (*next == NULL)
              break;
          }
        length -= dist (*next, p);
        prev = p;
        p = *next;
        last = p->last_time + type;
        if (*last != NO_LAST_TIME)
          {
            long double cycle;

            cycle = *last - length;
            if (length >= cycle)
              length -= floorl (length / cycle) * cycle;
          }
        *last = length;
      }
    printf ("Provazek se namota na kolik na souradnicich %Lf, %Lf\n", p->x, p->y);
    return EXIT_SUCCESS;
}
```

14-2-2 Cenzoři — Pavel Nejedlý

V této úloze bylo potřeba řešit dva základní problémy – jednak zakázaná slova najít, a pak je nějak chytře odstranit – a to celé pokud možno co nejrychleji. Okamžitě se nabízí řešení používající funkce `pos` a `delete` v Pascalu, resp. `strstr` a `strcpy` v C, ale s jeho efektivitou to bude špatné – funkce `pos` pracuje v čase $O(n \cdot m)$, kde n je délka textu a m je délka slova, funkce `delete` v $O(n)$. Uvážíme-li, že „počet odstranění“ může být až n/m (všechna slova stejně dlouhá, resp. n pro nestejně dlouhá), dává nám to složitost $O(n/m \cdot (wnm+n)) = O(wn^2)$, kde w je počet slov. Jinými slovy, takové řešení si asi jedenáct bodů nezaslouží.

Zkušení řešitelé si buď pamatovali, nebo v ročence (úloha 12-2-1) či jiné moudré knize (aspoň podle toho, co do řešení napsali) našli algoritmus na řešení prvního problému pracující v čase $O(n+wm)$, což je o hodně lepší než $O(nwm)$ pro funkci `pos`. Tento algoritmus zde popisovat nebudu, neboť je již popsán u zmíněné úlohy (tak proč nosit dříví do lesa) a navíc konečným automatům a jejich konstrukci byla věnována pokračovací série osmého ročníku KSP.

Zbývá tedy jen dořešit, jak provádět vyškrtávání cenzurovaných slov – jednoduchý algoritmus typu najdi slovo, vyšktrni a začni zase od začátku vede na složitost $O(n^2/m)$, což stále ještě není to pravé. Pokud si uvědomíme, jak vyhledávací automat pracuje, zjistíme, že po odstranění slova pouze potřebujeme změnit jeho stav na stav, v němž byl předtím, než začal číst odstraněné slovo, a pokračovat dalšími znaky ve vstupu. Řešení je tedy jednoduché – pamatujeme si načtené nešktrnuté znaky, a pro každý z nich také stav, v němž se ocitl automat po jeho načtení. Pokud zjistíme, že jsme právě „donačetli“ zakázané slovo, jednoduše odstraníme posledních k znaků (k je délka slova), a pak nastavíme aktuální stav na stav u posledního z neodstraněných znaků. Tím se vyhneme zbytečnému kopírování zbytku řetězce, které provádí `delete`.

Časová složitost celého algoritmu pak je $O(n+wm+d)$, kde d je počet odstraněných znaků, protože ale každý znak může být odstraněn nejvýše jednou, je celková složitost $O(n+wm)$.

```
#include <stdio.h>

#define MAXSTAVU 100
#define MAXSLOVO 100
#define MAXTEXT 1000
#define ZNAKU 256

struct {
    int delka;
    int zpetna;
    int prechod[ZNAKU];
} stavy[MAXSTAVU];

int fronta[MAXSTAVU], fpos, fend;
```

```
struct {
    char pismeno;
    int stav;
} text [MAXTEXT];

char slovo [MAXSLOVO];

int pocetSlov;
int pocetStavu;
int delkaTextu;

int main (void) {

    int i;
    char *p;
    int stav;
    int c;

    gets (slovo);
    sscanf (slovo, "%d", &pocetSlov);

    pocetStavu = 1;                          /* Nulty stav je pocatecni */

    /* vytvoreni A-C stromu */
    for (i = 0; i < pocetSlov; i++) {
       gets (slovo);
       stav = 0;
       p = slovo;

       for (p = slovo; *p; p++)
          if (stavy[stav].prechod[*p])
             stav = stavy[stav].prechod[*p];
          else {
             /* pridavame novy stav */
             stavy[stav].prechod[*p] = pocetStavu;
             stav = pocetStavu;
             pocetStavu ++;
          }

       stavy[stav].delka = p - slovo;        /* delka slova */
    }

    /* pridame zpetne hrany */
    fpos = 0; fend = 0;

    /* prvni stav je zvlastni pripad */
    for (i = 0; i < ZNAKU; i++)
       if (stavy[0].prechod[i])
          fronta[fend++] = stavy[0].prechod[i];

    while (fpos < fend) {
       stav = fronta[fpos++];

       for (i = 0; i < ZNAKU; i++)
          if (stavy[stav].prechod[i]) {
             int zpetna = stavy[stav].zpetna;
             int novyStav = stavy[stav].prechod[i];

             while (!stavy[zpetna].prechod[i] && zpetna )
```

```
            zpetna = stavy[zpetna].zpetna;
          stavy[novyStav].zpetna = stavy[zpetna].prechod[i];
          if (stavy[novyStav].delka < stavy[stavy[zpetna].prechod[i]].delka)
            stavy[novyStav].delka = stavy[stavy[zpetna].prechod[i]].delka;
          fronta[fend++] = novyStav;
        }
    }

    /* a ted vlastni prace */
    stav = 0;
    delkaTextu = 1;
    text[0].stav = stav;
    while ( (c = getchar ()) != '\n') {
      while (stav && !stavy[stav].prechod[c])
        stav = stavy[stav].zpetna;
      stav = stavy[stav].prechod[c];
      if (stavy[stav].delka > 0) {
        /* odstran slovo a obnov stav */
        delkaTextu -= stavy[stav].delka - 1;
        stav = text[delkaTextu - 1].stav;
      } else {
        text[delkaTextu].stav = stav;
        text[delkaTextu].pismeno = (char) c;
        delkaTextu ++;
      }
    }
    for (i = 1; i < delkaTextu; i++)
      putchar (text[i].pismeno);
    putchar ('\n');
    return 0;
}
```

14-2-3 Dekomprese — Daniel Kráľ

Všechna řešení, co jsme obdrželi, byla založena na dekompresi zakódovaného textu, případně jeho částečné dekompresi, a spočítání počtu výskytů hledaného znaku v původním (nezakomprimovaném) textu. Takováto řešení však nejsou polynomiální ve velikosti vstupu, neboť původní (nezakomprimovaný) text může být až exponenciálně větší.

Naše řešení nebude vůbec vytvářet původní text. Nejprve si úlohu malinko zobecněme: LZW kód je tvořen posloupností bloků tvořených buď jedním písmenem nebo odkazem na úsek v předchozím textu. My každému bloku přiřadíme váhu w, která bude udávat, že jeden výskyt hledaného písmene v daném bloku se bude počítat jako w výskytů. Samotný algoritmus na určení počtu

výskytů hledaného písmena v textu by mohl vypadat následovně: Uvažme poslední blok LZW kódu. Pokud je tento blok tvořen písmenem, potom ho z kódu odebereme a pokud je písmeno tohoto bloku hledaným písmem, zvýšíme čítač výskytů hledaného písmene o váhu tohoto bloku. Pokud je tento blok odkazem do předchozího textu, pak zvýšíme o váhu posledního bloku váhu těch bloků kódu, které přesně tvoří interval, na který se poslední blok odkazuje. K tomu, abychom mohli zvýšit váhu přesně těch bloků, co tvoří interval, na který se poslední blok odkazuje, může být nutné některé bloky v kódu rozdělit. Nyní by se mohlo zdát, že jsme si příliš nepomohli, neboť nám může vznikat nekontrolovatelné množství nových bloků v kódu.

V našem řešení budeme postupovat tak, jak jsme si výše popsali, ale s trochou opatrnosti navíc. Bloky původního kódu a skupiny bloků vzniklých rozdělením jednoho bloku původního kódu budeme nazývat *pravé bloky*. V každém kroku našeho algoritmu vezmeme poslední pravý blok (v některých krocích vezmeme skupinu bloků vzniklých rozdělením jednoho pravého bloku), a ty „přičteme“ ke zbylému kódu. Nechť P je počet pravých bloků a Q je počet bloků v našem kódu. V každém kroku se P zmenší o 1. Řekněme, že poslední pravý blok je ve skutečnosti tvořen k bloky. V tomto kroku nejprve snížíme Q o k, a pak se možná počet bloků zvýší o $k+1$ rozdělením některých bloků v kódu. Celkově se tedy Q, počet bloků kódu, zvýší nejvýše v jednom kroku o 1. Protože kroků našeho algoritmu je tolik, kolik je délka původního kódu, délka kódu (měřena počtem jeho bloků), se kterým pracujeme, nikdy nepřevýší dvojnásobek délky kódu zadaného na vstupu. Každý krok algoritmu lze snadno provést v lineárním čase v délce právě zpracovávaného kódu. Časová složitost našeho algoritmu tedy bude $O(N^2)$ a jeho paměťová složitost bude $O(N)$, kde N je počet bloků LZW kódu na vstup. Všimněte si, že délka samotného textu může být až řádově 2^N, tj. náš algoritmus může být až exponenciálně rychlejší než algoritmus založený na dekompresi zadaného textu.

Samotný program je přímým přepsáním výše popsaného postupu. Kód je uchováván v proměnných speciálního datového typu `lzw` typu záznam. Nejdříve v proceduře `vstup` načteme kód (přidržujeme se formátu vstupu v zadání úlohy) a pomocí funkce `redukce` odebíráme poslední pravý blok. Funkce `redukce` vrací počet výskytů (vzhledem k váhám bloků) hledaného písmene v odebraných blocích. Pokud dochází k rozdělení bloku, dodržujeme pravidlo, že první (odkazový) blok pravého bloku má v položce `pismeno` hodnotu `#0` a ostatní odkazové bloky tohoto pravého bloku mají v položce `pismeno` hodnotu `#1`. Bloky odpovídající jednomu písmenu není potřeba nikdy rozdělovat, neboť délka jim odpovídajícího textu je jedna.

```
program lzw_count;
const MAX=1000; { maximální počet úseků v LZW kódu }
      LMAX=1000000000; { maximalni délka textu }
type lzw=record
```

```
          zaznamu: word;
          zaznamy: array[1..MAX] of
                   record
                     vaha: longint;
                     pismeno: char;    { #0/#1 = odkaz na předchozí část textu
                                         #1 = blok vzniklý rozdělením bloku }
                     zacatek: longint;
                     delka: longint;
                   end;
        end;
procedure vstup(var lzw0: lzw; var pismeno: char);
  { Načteme vstup - jenom samé nudné technické náležitosti ... }
  var c: char;
      i1,i2: longint;
  begin
    lzw0.zaznamu:=0;
    c:=readkey;
    while c<>'/' do
      begin
        case c of
        'a'..'z': begin
                    inc(lzw0.zaznamu);
                    lzw0.zaznamy[lzw0.zaznamu].vaha:=1;
                    lzw0.zaznamy[lzw0.zaznamu].pismeno:=c;
                    lzw0.zaznamy[lzw0.zaznamu].delka:=1;
                  end;
        '(': begin
               i1:=0;
               c:=readkey;
               repeat
                 i1:=10*i1+ord(c)-ord('0');
                 c:=readkey;
               until c=',';
               i2:=0;
               c:=readkey;
               repeat
                 i2:=10*i2+ord(c)-ord('0');
                 c:=readkey;
               until c=')';
               inc(lzw0.zaznamu);
               lzw0.zaznamy[lzw0.zaznamu].vaha:=1;
               lzw0.zaznamy[lzw0.zaznamu].pismeno:=#0;
               lzw0.zaznamy[lzw0.zaznamu].zacatek:=i1;
               lzw0.zaznamy[lzw0.zaznamu].delka:=i2;
             end;
        end;
        c:=readkey;
      end;
    pismeno:=readkey;
  end;
function redukce(var lzw1,lzw2: lzw; pismeno: char):longint;
  var nalezeno: longint;
      index2, odkud: word;
      index: word;
      delka: longint;
  begin
    nalezeno:=0;
    { Nejdříve odstraníme písmena na konci zakomprimovaného textu }
```

```
while (lzw1.zaznamu>0) and (lzw1.zaznamy[lzw1.zaznamu].pismeno>#1) do
  begin
    if lzw1.zaznamy[lzw1.zaznamu].pismeno=pismeno then
      nalezeno:=nalezeno+lzw1.zaznamy[lzw1.zaznamu].vaha;
    dec(lzw1.zaznamu);
  end;
redukce:=nalezeno;
if lzw1.zaznamu=0 then
  begin
    lzw2.zaznamu:=0;
    exit
  end;
{ Najdeme souvislý podrozdělený úsek na konci textu }
index2:=lzw1.zaznamu;
while lzw1.zaznamy[index2].pismeno<>#0 do dec(index2);
odkud:=index2;
lzw1.zaznamy[lzw1.zaznamu+1].zacatek:=LMAX;
{ A přeneseme do nového kódu }
index:=1;
delka:=0;
lzw2.zaznamu:=0;
while index<odkud do
  begin
    if delka+lzw1.zaznamy[index].delka+1<=lzw1.zaznamy[index2].zacatek then
      begin { ještě jsme před koncovým úsekem }
        inc(lzw2.zaznamu);
        lzw2.zaznamy[lzw2.zaznamu].vaha:=lzw1.zaznamy[index].vaha;
        lzw2.zaznamy[lzw2.zaznamu].pismeno:=lzw1.zaznamy[index].pismeno;
        lzw2.zaznamy[lzw2.zaznamu].zacatek:=lzw1.zaznamy[index].zacatek;
        lzw2.zaznamy[lzw2.zaznamu].delka:=lzw1.zaznamy[index].delka;
        delka:=delka+lzw2.zaznamy[lzw2.zaznamu].delka;
        inc(index);
        continue;
      end;
    if delka+1<lzw1.zaznamy[index2].zacatek then
      begin { částečný přesah; tady musí platit lzw1.zaznamy[index]=#0/#1 }
        inc(lzw2.zaznamu);
        lzw2.zaznamy[lzw2.zaznamu].vaha:=lzw1.zaznamy[index].vaha;
        lzw2.zaznamy[lzw2.zaznamu].pismeno:=lzw1.zaznamy[index].pismeno;
        lzw2.zaznamy[lzw2.zaznamu].zacatek:=lzw1.zaznamy[index].zacatek;
        lzw2.zaznamy[lzw2.zaznamu].delka:=lzw1.zaznamy[index2].zacatek-delka-1;
        delka:=delka+lzw2.zaznamy[lzw2.zaznamu].delka;
        { Rozdělíme interval, do kterého částečně zasahujeme. }
        lzw1.zaznamy[index].pismeno := #1;
        lzw1.zaznamy[index].zacatek :=
           lzw1.zaznamy[index].zacatek+lzw2.zaznamy[lzw2.zaznamu].delka;
        lzw1.zaznamy[index].delka :=
           lzw1.zaznamy[index].delka-lzw2.zaznamy[lzw2.zaznamu].delka;
        continue;
      end;
    { Úsek v lzw1 i v lzw2 nyní začíná stejně ... }
    if lzw1.zaznamy[index].delka=lzw1.zaznamy[index2].delka then
      begin
        inc(lzw2.zaznamu);
        lzw2.zaznamy[lzw2.zaznamu].vaha:=
            lzw1.zaznamy[index].vaha+lzw1.zaznamy[index2].vaha;
        lzw2.zaznamy[lzw2.zaznamu].pismeno:=lzw1.zaznamy[index].pismeno;
        lzw2.zaznamy[lzw2.zaznamu].zacatek:=lzw1.zaznamy[index].zacatek;
```

```
            lzw2.zaznamy[lzw2.zaznamu].delka:=lzw1.zaznamy[index].delka;
            delka:=delka+lzw2.zaznamy[lzw2.zaznamu].delka;
            inc(index);
            inc(index2);
            continue;
          end;
        if lzw1.zaznamy[index].delka<lzw1.zaznamy[index2].delka then
          begin
            inc(lzw2.zaznamu);
            lzw2.zaznamy[lzw2.zaznamu].vaha:=
                lzw1.zaznamy[index].vaha+lzw1.zaznamy[index2].vaha;
            lzw2.zaznamy[lzw2.zaznamu].pismeno:=lzw1.zaznamy[index].pismeno;
            lzw2.zaznamy[lzw2.zaznamu].zacatek:=lzw1.zaznamy[index].zacatek;
            lzw2.zaznamy[lzw2.zaznamu].delka:=lzw1.zaznamy[index].delka;
            delka:=delka+lzw2.zaznamy[lzw2.zaznamu].delka;
            lzw1.zaznamy[index2].zacatek:=
                lzw1.zaznamy[index2].zacatek+lzw1.zaznamy[index].delka;
            lzw1.zaznamy[index2].delka:=
                lzw1.zaznamy[index2].delka-lzw1.zaznamy[index].delka;
            inc(index);
            continue;
          end;
        if lzw1.zaznamy[index].delka>lzw1.zaznamy[index2].delka then
          begin
            inc(lzw2.zaznamu);
            lzw2.zaznamy[lzw2.zaznamu].vaha:=
                lzw1.zaznamy[index].vaha+lzw1.zaznamy[index2].vaha;
            lzw2.zaznamy[lzw2.zaznamu].pismeno:=lzw1.zaznamy[index].pismeno;
            lzw2.zaznamy[lzw2.zaznamu].zacatek:=lzw1.zaznamy[index].zacatek;
            lzw2.zaznamy[lzw2.zaznamu].delka:=lzw1.zaznamy[index2].delka;
            delka:=delka+lzw2.zaznamy[lzw2.zaznamu].delka;
            lzw1.zaznamy[index].pismeno:=#1;
            lzw1.zaznamy[index].zacatek:=
                lzw1.zaznamy[index].zacatek+lzw1.zaznamy[index2].delka;
            lzw1.zaznamy[index].delka:=
                lzw1.zaznamy[index].delka-lzw1.zaznamy[index2].delka;
            inc(index2);
            continue;
          end;
        { Sem bychom se nikdy neměli dostat ! }
      end;
    { lzw1.zaznamy[lzw1.zaznamu+1].zacatek= ???? }
  end;
var lzw1,lzw2: lzw;
    vyskytu: longint;
    pismeno:char;
begin
  vstup(lzw1,pismeno);
  vyskytu:=0;
  while lzw1.zaznamu<>0 do
    begin
      vyskytu:=vyskytu+redukce(lzw1,lzw2,pismeno);
      if lzw2.zaznamu=0 then break;
      vyskytu:=vyskytu+redukce(lzw2,lzw1,pismeno);
    end;
  writeln('Písmeno ',pismeno,' se v textu vyskytuje ',vyskytu,'-krát.');
end.
```

14-2-4 Seznamování — Miroslav Rudišin

Napriek tomu, že tento príklad nie je obtiažny, prišlo relatívne málo riešení. Ideou algoritmu je postupne premiestňovať účastníkov, ktorí majú vo svojej skupine viac známych, do druhej skupiny. Toto sa opakuje pokiaľ existuje účastník, ktorý má v druhej skupine menej známych. Z popisu ešte nie je zrejmé, či sa tento algoritmus nezacyklí na nejakom vstupe, preto to musíme dokázať.

Nech F je počet dvojíc, ktoré sa poznajú a sú v jednej skupine v nejakom rozdelení. Na začiatku nech sú všetci účastníci v prvej skupine, teda F_0 je M (počet známostí). Každým presunom sa hodnota F zmení na $F-k+(p-k)$, kde p je počet známych presúvaného účastníka a k je počet jeho známych v pôvodnej skupine. Pretože F je shora obmedzená a $p-2k$ je záporné číslo (vybrali sme účastníka pre ktorého platí $2k > p$), bude F klesať až sa raz zastaví. Potom musí platiť, že neexistuje niekto, kto pozná vo svojej skupine viac účastníkov (inak by sme ho ešte mohli preradiť). Časová zložitosť algoritmu je $O(mn)$, pamäťová $O(m+n)$, kde m je počet známostí a n počet účastníkov.

```
#include <stdio.h>
#define maxn 100
#define maxm 1000
/* pocet znamych, pocet znamych v skupine, znamosti */
int cnt[maxn], scnt[maxn], zn1[maxm], zn2[maxm];
/* skupina, zoznam znamych pre kazdeho ucastnika, index na zaciatok */
int skup[maxn], zn[maxm], idx[maxn+1];

int main () {
    int n, m, i, j, a, b, stop=0;
    int k;

    scanf ("%d %d", &n, &m);                    /* pocet ucastnikov, znamosti */
    for (i=0; i<m; i++) {
        scanf ("%d %d", &a, &b);
        cnt[--a]++; cnt[--b]++;
        zn1[i]=a; zn2[i]=b;
    }

    for (i=0; i<n; i++)                          /* rozdelenie zoznamu znamych pre
                                                    ucastnikov */
        idx[i+1]=idx[i]+cnt[i];

    for (i=0; i<m; i++) {                        /* naplnanie */
        a=zn1[i]; b=zn2[i];
        zn[--idx[a+1]]=b;
        zn[--idx[b+1]]=a;
    }

    for (i=0; i<n; i++) {
        idx[i+1]=idx[i]+cnt[i];                  /* obnovenie indexov */
        scnt[i]=cnt[i];                          /* na zaciatku su vsetci v 1. skupine */
    }

    do {
```

```
        stop=1;
        for (i=0; i<n; i++)
            if (2*scnt[i] > cnt[i]) {        /* presun do inej skupiny */
                stop=0;
                /* update poctu znamych v skupinach */
                for (j=idx[i]; j<idx[i+1]; j++)
                    /* ak boli rovnakej sk.: -1 inak: +1 (^ je xor) */
                    scnt[zn[j]] += -1+2*(skup[zn[j]]^skup[i]);
                skup[i] = 1-skup[i];         /* presun */
                scnt[i] = cnt[i]-scnt[i];    /* vymena znamych */
                break;                       /* aby bol dokaz jasnejsi, inak netreba */
            }
    } while (!stop);
    for (i=0; i<n; i++)
        printf ("%d.: %d\n", i+1, skup[i]+1);
    return 0;
}
```

14-2-5 Turingovy stroje — Jan Kára

K řešení této úlohy se dá přistoupit dvěma způsoby. Oba dva vedou stroj, který pracuje v čase i prostoru $O(N)$. První způsob bude využívat stroj se třemi páskami. První dvě pásky budeme používat jako pomocné, na třetí pásce si zkonstruujeme číslo ve dvojkové soustavě, a to nakonec překopírujeme na první pásku. Číslo ve dvojkové soustavě budeme konstruovat tak, že budeme postupně procházet přes jedničky na první pásce, každou druhou jedničku si zapíšeme na druhou pásku a na třetí pásce si budeme průběžně udržovat počet projitých jedniček modulo dva. Když dojdeme na konec jedniček na první pásce (řekněme, že jich tam bylo k), máme na třetí pásce k mod 2 a na druhé pásce k div 2. Nyní smažeme číslo na první pásce, nakopírujeme na jeho místo číslo z druhé pásky, a pak znovu procházíme posloupnost jedniček na první pásce (mohli bychom se pokusit stroj zrychlit tak, že bychom místo kopírování pouze zaměnili význam první a druhé pásky. To by nám ale žádné asymptotické zrychlení nepřineslo, a proto si touto optimalizací nebudeme komplikovat život). Takto postupujeme, dokud na první pásce zbývají nějaká čísla. Nakonec ještě překopírujeme výsledek z třetí pásky na první. Časová složitost našeho Turingova stroje je $O(N + N/2 + N/4 + \ldots + 1) = O(N)$, jeho paměťová složitost je zřejmě $O(N + N/2 + \log N) = O(N)$. Implementace tohoto stroje je čistě technickou záležitostí, a proto ji zde neuvádíme.

Druhý způsob má výhodu v jednodušší implementaci, ovšem jeho nevýhodou je komplikovanější analýza časové složitosti. Tento způsob využívá stroje se dvěma páskami. Z první pásky se vstupem budeme postupně odebírat jedničky a ty budeme přičítat k číslu ve dvojkové soustavě uloženému na druhé pásce. Přičítání jedničky k číslu ve dvojkové soustavě je jednoduché. Procházíme číslo od nejnižších bitů. Dokud jdeme po jedničkách, přepisujeme je na nulu

(dochází totiž k přenosu). Když narazíme na nulu nebo na Λ, přepíšeme ji na jedničku a vrátíme se na začátek čísla. Když nám dojdou jedničky na vstupní pásce, překopírujeme číslo z druhé pásky na první a skončíme. Stroj bude mít zřejmě paměťovou složitost $O(N + \log N) = O(N)$. S časovou složitostí je to ovšem komplikovanější, protože jedno přičtení jedničky nám může trvat až $O(\log N)$ a celkově bychom tedy dospěli k času $O(N \log N)$. Pokud chceme dosáhnout lepšího odhadu, musíme počítat přesněji – nebudeme počítat, kolik nám trvá jedno přičtení jedničky, ale jak dlouho nám bude trvat nasčítat po jedničce do N (tedy jak dlouho nám bude trvat provedení N operací přičtení jedničky). Můžeme klidně předpokládat, že $N = 2^K$ (jinak si dané číslo pro účely odhadu můžeme zvýšit na nejbližší vyšší mocninu dvojky – zvýšíme ho tak nejvýše dvakrát a asymptoticky si tedy neuškodíme). Počet operací v posloupnosti N přičítání, které se zastaví na první cifře, bude zřejmě $N/2$ – právě tolik čísel totiž končí na nulu. Obdobně počet operací, které se zastaví na druhé cifře, bude $N/4$, protože tolik je čísel z $0 \ldots N-1$, která končí na jedničku a na druhé cifře od konce mají nulu. Takto snadno spočteme, že počet operací, které projdou i cifer, bude $N/2^i$. Když nyní čas pro všechny operace sečteme, získáváme sumu:

$$\sum_{i=0}^{K} i \cdot N/2^i \leq N \cdot \sum_{i=0}^{K} 1/2^i \leq 2 \cdot N = O(N)$$

(První z nerovností plyne z následujícího rozpisu sumy

$$\sum_{i=0}^{K} i/2^i = \left(\sum_{i=0}^{K} 1/2^i\right) + \left(\sum_{i=1}^{K} 1/2^i\right) + \ldots + \left(\sum_{i=K}^{K} 1/2^i\right),$$

ve kterém se j-tá suma dá shora odhadnout jako $1/2^j$.)
Poznámka: Takovémuto počítání složitosti celé posloupnosti operací se říká „amortizovaná složitost“.

Stroj pro postupné přičítání vypadá takto:

$(q_0, (\Lambda, \Lambda))$	$\to (q_0, (\mathbf{0}, \Lambda), (L, L))$	Ošetříme prázdný vstup
$(q_0, (\mathbf{1}, \Lambda))$	$\to (q_1, (\mathbf{0}, \Lambda), (R, R))$	Označíme začátky pásek
$(q_1, (?, \Lambda))$	$\to (q_2, (?, \mathbf{1}), (N, N))$	Zapíšeme první nabranou jedničku
$(q_2, (1, ?))$	$\to (q_i, (\Lambda, ?), (R, N))$	
$(q_2, (\Lambda, ?))$	$\to (q_c, (\Lambda, ?), (L, N))$	Došly jedničky na vstupu?
$(q_i, (?, \mathbf{1}))$	$\to (q_i, (?, \mathbf{0}), (N, R))$	Procházíme přes jedničky
$(q_i, (?, \mathbf{0}/\Lambda))$	$\to (q_v, (?, \mathbf{1}), (N, L))$	Konec přetékání?
$(q_v, (?, \mathbf{0}/\mathbf{1}))$	$\to (q_v, (?, \mathbf{0}/\mathbf{1}), (N, L))$	Návrat na počátek
$(q_v, (?, \Lambda))$	$\to (q_2, (?, \Lambda), (N, R))$	
$(q_c, (\Lambda, ?))$	$\to (q_c, (\Lambda, ?), (L, N))$	Návrat na první pásce

$(q_c, (\mathbf{0}, x))$	$\to (q_{c'}, (x, \Lambda), (R, R))$	Začínáme kopírovat z druhé pásky
$(q_{c'}, (\Lambda, x))$	$\to (q_{c'}, (x, \Lambda), (R, R))$	
$(q_{c'}, (\Lambda, \Lambda))$	$\to (q_e, (\Lambda, \Lambda), (L, L))$	Dokopírováno
$(q_e, (?, ?))$	$\to (q_e, (?, ?), (L, L))$	Končíme

14-3-1 Ježíškův problém — Pavel Šanda

Můj nejmilejší Ježíšku,

minulé Vánoce byly na prosto skvělé, ale musím Ti napsat, že roztomilý kanárek skončil ve věčně nenažrané tlamě našeho Moura a rovněž potápěčské brýle doznaly patrných změn na levém předním skle v důsledku ztřeštěného chování mé sestry. Jelikož vím o Tvém zapeklitém problému, rozhodl jsem se pomoci. Drahnou dobu jsem si říkal, že postačujícím řešením by bylo postupně nagenerovat všechny uspořádané monotónní posloupnosti, zvláště poté, co se ukázalo, že rekurzivní funkci velmi často volám se shodnými parametry – pouhým zapamatováním jednou spočtených hodnot se vyhnu exponenciální časové složitosti. Když jsem si však uvědomil, že se mi rozdírá starotou zašlá kopací meruna, bylo mi jasné že by to pro Tebe chtělo něco lepšího.

První trik spočívá v tom, že budu při generování rozdělovat dárky postupně od nezlobivějšího děcka. Jestliže mu dám $i \geq 0$ dárků, musím dát alespoň i všem ostatním.

Nechť n je počet dětí a m počet dárků. Zjistit počet rozdělení pro i dárků u nejzlobivějšího děcka je úloha stvořená pro rekurzi – přidělím mu i dárků a rekurzivně vyřeším tutéž úlohu pro $n-1$ dětí a $m - n \cdot i$ dárků (ostatní d. byly přece mým výběrem pevně stanoveny). Zjistit počet všech rozdělení pak znamená zjistit počet rozdělení pro všechna *rozumná* i, formálněji

$$R(n, m) = \sum_{i=0}^{m} R(n-1, m - i \cdot n) \quad \text{pro } i \leq m/n.$$

Pro jediné dítě existuje pouze jedna možnost, kterak dárky rozdělit, neboli $R(1, *) = 1$.

Pro druhý trik využijeme dynamického programování: $R(i, *)$ je závislé pouze na $R(i-1, *)$, takže nám stačí generovat postupně jednotlivé řádky od $R(1, 0), \ldots, R(1, m)$ do $R(n, 0), \ldots, R(n, m)$. Tím se jednak zachránila paměťová složitost $O(m)$ (pro generování stačí dva řádky) a druhak je zaručeno, že se nic nepočítá na dvakráte.

A jak to bude s časovou složitostí? Pro každé políčko v i-tém řádku je třeba spočítat m/i hodnot, což pro celý řádek znamená m^2/i. Tedy pro všechny řádky platí

$$\sum_{i=1}^{n} \frac{m^2}{i} = m^2 \sum_{i=1}^{n} \frac{1}{i} = m^2 \log n.$$

Zbývá jen dodat program:

```
#include <stdio.h>
#define MAX 10

int R[2][MAX];

int r (int n,  int m){
int i, j, k;
  for  ( i=0; i<=m; i++) R[1][i]=1;              /* Pro jedno dite jedna moznost */

  for  ( i=2; i<=n; i++ )
    for ( R[i%2][j=0]=0; j<=m; R[i%2][++j]=0 )   /* zapomen predchozi */
      for  ( k=0; k <= j/i; k++ ) R[i%2][j]+=R[ (i-1)%2][j-k*i];    /* suma */

  return R[n%2][m];
}

int main (void){
  int deti,  darku;
  scanf ("%d", &deti); scanf ("%d", &darku);
  printf ("%d moznosti.\n", r (deti, darku));
  return 0;
}
```

14-3-2 Cestářův problém — Tomáš Vyskočil

A jak by to vlastně v Agranetánii dopadlo, kdyby vás požádali o program, který určuje udržované silnice? Docela dobře. Většina z vás si všimla, že jde o klasický problém již tisíckrát řešený, tedy o hledání minimální kostry v grafu. Je pravda, že někteří neměli řešení zrovna optimální, ale povětšinou by alespoň vedlo ke správnému výsledku.

Jak se tedy s tímto problémem vypořádat? Já použiji Kruskalův algoritmus na hledání minimální kostry. Algoritmů je více – např. Borůvkův, Jarníkův – ale tento je nejjednodušší; na implementaci navíc využiji struktury *union-find.*

Úlohu si nejdříve převedeme do řeči grafů. Města budou vrcholy našeho grafu a silnice jeho hrany. Na začátku algoritmu máme graf bez hran – každý vrchol má tedy vlastní komponentu souvislosti. Postupně bereme hrany podle jejich délky (od nejkratší k nejdelší) a pokud hrana nespojuje vrcholy ve stejné komponentě, tak hranu přidáme do našeho grafu a komponenty, které hrana spojila, slijeme (takto přidáme $N-1$ hran (N je počet vrcholů), protože přidáním každé hrany se sloučí dvě komponenty). A tím je vlastně popsán celý algoritmus.

Implementace je v podstatě přesnou kopií algoritmu až na to, že zde používám algoritmus union find, který nám zaručuje, že celkem na slévání komponent a testování, do které komponenty jaký vrchol patří, spotřebujeme čas nejvýše $O(M \cdot \alpha(N))$, kde N je počet vrcholů a M počet hran ($\alpha(N)$ je funkce inverzní k Ackermanově funkci. Pro naše účely nám bude stačit, že $\alpha(N)$ roste skutečně velmi pomalu a pro praktické účely ji lze bez problémů považovat

za konstantní). Union find funguje tak, že každý vrchol si udržuje odkaz na „nadřízený“ vrchol v jeho komponentě souvislosti. Na počátku se každý vrchol odkazuje sám na sebe. Když chceme zjistit, zda dva vrcholy leží ve stejné komponentě, zjistíme si nejdříve „šéfy“ komponent, ve kterých leží – půjdeme po nadřízených vrcholech tak dlouho, až narazíme na vrchol, který se odkazuje sám na sebe – to je šéf komponenty. Jestli jsou vrcholy ve stejné komponentě, zjistíme pak snadno tak, že zjistíme, zda mají shodného šéfa. Propojení komponent pak realizujeme tak, že šéfovi menší komponenty dáme za nadřízeného šéfa větší komponenty (u každého šéfa komponenty si budeme proto udržovat velikost komponenty, které šéfuje). Jak si někteří pečlivější čtenáři všimli, takto by struktura ještě zdaleka neměla požadovanou složitost (cesty k šéfům komponent mohou vznikat dosti dlouhé). Složitost se nám ale zlepší na požadovanou, pokud uděláme do hledání šéfů drobné vylepšení: Vždy když k nějakému vrcholu nalezneme šéfa jeho komponenty, tak projdeme cestu k šéfovi ještě jednou a u každého vrcholu nastavíme odkaz na nadřízeného na šéfa komponenty. Výpočet časové složitosti takto vylepšeného algoritmu rozhodně není jednoduchý, a proto ho zde nebudeme uvádět.

A jakou má náš algoritmus celkovou složitost? To spočítáme jednoduše: třídění $O(M \cdot \log M)$, zkoušení a přidávání hran uděláme v $O(M \cdot \alpha(N))$. Dohromady to tedy dává $O(M \cdot \log M)$.

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

#define MAX_M 100000
#define MAX_N 1000

typedef struct{
        int from;
        int to;
        int length;
}T_EDGE;

int ud[MAX_N];                          /* Ukazatelé na nadřízené vrcholy */
int vn[MAX_N];                          /* Velikosti podřízených komponent */
T_EDGE h[MAX_M];

/* Nalezne šéfa komponenty a přepíše odkazy na nadřízené */
int find_chief (int a)
{
        int h, x=a;

        while (ud[a]!=a)
                a=ud[a];
        while (ud[x]!=x){
                h=ud[x];
                ud[x]=a;
                x=h;
        }
```

```
        return a;
}
/* Spojí dvě komponenty */
void join (int a, int b)
{
        a=find_chief (a);
        b=find_chief (b);
        if (vn[a] < vn[b]) {
                vn[b]+=vn[a];
                ud[a]=b;
        }
        else {
                vn[a]+=vn[b];
                ud[b]=a;
        }
}
int cmp (const void *a, const void *b)
{
        return ( (T_EDGE *)a)->length - ( (T_EDGE *)b)->length;
}
int main (void)
{
        int i, n, m, no_edge, sum;
        /* Inicializace */
        for (i=0; i<MAX_N; i++){
                ud[i]=i;
                vn[i]=1;
        }

        scanf ("%d %d", &n, &m);
        for (i=0; i<m; i++){
                scanf ("%d %d %d", &h[i].from, &h[i].to, &h[i].length);
        }

        /* Setřídíme si hrany podle délky */
        qsort (h, m, sizeof (T_EDGE), cmp);
        no_edge=0;
        i=0;
        sum=0;
        while (no_edge<n-1){            /* Ještě nemáme dost hran? */
                /* Vrcholy v různých komponentách? */
                if (find_chief (h[i].from) != find_chief (h[i].to)){
                        /* Přidáme hranu, spojíme komponenty */
                        join (h[i].from, h[i].to);
                        printf ("Udrzuj cestu z %d do %d\n", h[i].from, h[i].to);
                        sum+=h[i].length;
                        no_edge++;
                }
                i++;
```

```
        }
        printf ("Celkove se musi udrzovat %d km silnic.\n", sum);
        return 0;
}
```

14-3-3 Králův problém — Jakub Bystroň

Máme dán kořenový strom a naším úkolem je umístit kámen podle daných pravidel do kořene. Ukažme si jednoduché řešení pracující v čase $O(N \log N)$. Strom budeme prohledávat od kořene do hloubky. Řekněme, že chceme zjistit, kolik minimálně potřebujeme kamenů k obsazení vrcholu v. Pokud je v list, je hledaná hodnota rovna jedné. Jinak si vezměme všechny jeho syny $w_1, \ldots, w_k$. Rekurzivně pro každého syna w_i určíme minimální počet kamenů potřebný k jeho obsazení. Takto získané hodnoty setřiďme sestupně, vzniklou posloupnost označme p_i. Tvrdím, že minimální počet kamenů nutný k umístění kamene do vrcholu v je maximum z čísel $p_i + i, i = 1, 2, \ldots, k$. Je jasné, že nejlepší pro nás bude obsazovat kameny od těch vrcholů, které potřebují nejvíce kamenů na své obsazení. Po obsazení některého syna kamenem je však již tento kámen blokován a nemůžeme jej dále použít. Proto proměnná i. Tvrzení je velice jednoduché, samostatně si jej promyslete.

Poměrně častou chybou ve Vašich řešeních bylo, že jste si neuvědomili, že záleží na pořadí, v jakém budete jednotlivé vrcholy obsazovat kameny. Převládala však řešení používající stejnou myšlenku jako výše uvedené. K programu snad jen to, že se předpokládá, že na vstupu jsou hrany, podobně jako v zadání, vzestupně setříděny podle koncového vrcholu.

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

#define MAX 100

int h[MAX];                                  /* seznam hran  */
int l[MAX];
int N;

int cmp (const void *a,  const void *b)
{
    return * (int *) b - * (int *) a;
}

int p[MAX];

int solve (int v)
{
    int i,  j,  max=0;
    if  (l[v] == l[v+1])                     /* list  */
        return 1;
    for  (i=l[v]; i<l[v+1]; i++)             /* synove  */
        p[i] = solve (h[i]);
```

```
    qsort (p+l[v], l[v+1]-l[v], sizeof (*p), cmp);
    for (i=1; i<=l[v+1]-l[v]; i++) {
      j = i+l[v]-1;
      if (p[j] + i > max)                    /* maximum */
        max = p[j] + i;
    }

    return max;
}
int main (void)
{
    int i, j, x, y, z, koren;

    /* hrany jsou na vstupu setrideny */
    scanf ("%d %d", &N, &koren);
    koren--;
    for (i=j=0, z=-1; i<N-1; i++) {
      scanf ("%d %d", &x, &y);
      x--, y--;
      if (y!=z) { z=y; l[y]=j; }
      h[j] = x;
      j++;
    }
    for (y++; y<=N; y++)
      l[y]=j;
    printf ("Potrebujeme alespon %d kamenu.\n", solve (koren));
    return 0;
}
```

14-3-4 Hammingův problém — Pavel Machek

Tato úloha byla spíše praktického ražení. Proto byl pro zajímavost k jednotlivým způsobům řešení připojen i čas běhu na reálném počítači. Komentáře naleznete uvnitř vzorového programu:

```
/*
   Úlohu šlo řešit mnoha různými způsoby; uvedu několik z nich.

   Začátek je vždy stejný: provedeme XOR zadaných čísel. Tím úlohu převedeme
   na úlohu spočítat jedničky ve dvojkovém zápisu čísla.

   Triviálním řešením na spočtení jedniček je prostě jít po bitech, a
   za každý jedničkový si přičíst jedničku k výsledku. Toto řešení je
   nejmenší, ale také nejpomalejší, a vzhledem ke své rychlosti nebylo
   honorováno příliš mnoha body.

   Toto řešení je O(počet_bitů_v_int). Na reálném počítači (AMD Athlon 4 na 900MHz)
   15 sekund na 2 miliardy iterací.
*/

int bitcount_trivial(unsigned int a)
{
```

```
        int res = 0;
        while (a)
                res += a & 1,
                a >>= 1;
        return res;
}

/*
   Prvním zlepšením je brát bity ne po jednom, ale po nějakých větších
   skupinách, a počet bitů zjistit podíváním se do tabulky.

   Toto řešení je stále O(počet_bitů_v_int). Na reálném počítači 2.7 sekundy.
*/

char table[2048];

void table_init(void)
{
        int i;
        for (i=0; i<2048; i++)
                table[i] = bitcount_trivial(i);
}

int bitcount_table(unsigned int a)
{
        int res = 0;

        while (a)
                res += table[a & 0x7ff],
                a >>= 11;
        return res;
}

/*
   2.7 sekundy je trochu, moc, a mělo by to jít rychleji. Kompilátor
   zřejmě nepochopil, že počet průchodů while cyklem je maximálně
   3. Tak budu předpokládat 32-bitové slovo a řeknu mu to. 1.7
   sekundy.
*/

int bitcount_table2(unsigned int a)
{
        return table[a & 0x7ff] + table[(a >> 11) & 0x7ff] + table[a >> 22];
}

/*
   Ale ono to samozřejmě jde lépe. Počet bitů nastavených ve dvou
   bitech se vejde do dvou bitů. Počet bitů nastavených ve čtyřech
   bitech se vejde do čtyř bitů, a tak dále. Toto pozorování nám
   umožní sčítat bity paralelně; nejdřív skupiny po dvou, pak po
   čtyřech, ...

   Toto řešení je O(log_2(počet_bitů_v_int)), ale stále předpokládá
   délku slova 32 bitů. 3.4 sekundy.  */

int bitcount_smart(unsigned int n)
{
        n = ((n >>  1) & 0x55555555) + (n & 0x55555555);
```

```
        n = ((n >>  2) & 0x33333333) + (n & 0x33333333);
        n = ((n >>  4) & 0x0f0f0f0f) + (n & 0x0f0f0f0f);
        n = ((n >>  8) & 0x00ff00ff) + (n & 0x00ff00ff);
        n = ((n >> 16) & 0x0000ffff) + (n & 0x0000ffff);
        return n;
}

/*
   A teď jedna perlička, o kterou vás nemohu připravit. Přišla od jednoho z řešitelů
   a je tak rychlá, až je nečitelná. Používá stejný algoritmus jako bitcount_smart,
   ale poslední dva kroky spojí do jednoho násobení. Hezký nápad. A cenná položka
   do soutěže "co je to". 2.9 sekundy.
 */

int bitcount_obfuscated(unsigned int n)
{
        n = n - ((n >>  1) & 0x55555555);
        n = ((n >>  2) & 0x33333333) + (n & 0x33333333);
        n = (((n >>  4) + n) & 0x0f0f0f0f);
        n = (n * 0x01010101) >> 24;
        return n;
}

/*
   A teď jedno řešení teoreticky pěkné: v čase O(log_2(počet_bitů_v_int))
   a s kódem nezávislým na délce intu. Předpokládá ale, že unsigned int má
   nějakou velikost tvaru 2^n.

   Do tabulky array si připraví jednotlivé masky, které potom používá
   jako v předchozích algoritmech. 8.7 sekundy.
*/

unsigned int masks[1024];
int size = 0;

void theoretical_init(void)
{
        unsigned int mask = ~0U;
        int i;
        int numbits = bitcount_trivial(~0U);
        while (numbits) {
                mask = mask ^ (mask >> (numbits /= 2));
                masks[size++] = ~mask;
        }
        size --;
}

int bitcount_theoretical(unsigned int n)
{
        int i = size - 1;
        int shift = 1;
        while (i >= 0) {
                n = ((n >> shift) & masks[i]) + (n & masks[i]);
                i--;
                shift *= 2;
        }
        return n;
}
```

```
/*
   Existuje ještě jeden algoritmus, který s použitím násobení zvládne spočítat
   bity v konstantním čase bez ohledu na délku slova. Má jediný problém:
   já detaily neznám a v žádném z odevzdaných řešení nebyl. Takže snad někdy příště.
*/
```

14-3-5 Turingův problém — Jan Kára

Na tuto úlohu se nesešlo mnoho řešení, zato povětšinou obsahovala pouze drobnější chyby. A nyní k vzorovému řešení: Původní k-páskový Turingův stroj si označíme T, jeho abecedu Σ. Nově vytvářený jednopáskový stroj si označíme T' a jeho abecedu Σ'. K řešení lze přistupovat v zásadě třemi způsoby: buď pásky proložit (tj. i-té políčko z j-té pásky bude na pozici $i \cdot k + j$) nebo je naskládat za sebe (pak se ale musí při zápisu nového znaku pásky posouvat) nebo i-tá políčka na všech k páskách „zakomprimujeme“ do jedné k-tice na i-tém políčku jediné pásky nového stroje. Posledně jmenovaný způsob použijeme i v našem vzorovém řešení. Protože si také potřebujeme pamatovat pozice hlav nad jednotlivými páskami, bude abeceda stroje T' $\Sigma' = (\Sigma \times \{H, V\})^k \times \{Z, S\} \cup \{\omega\}$. Značka H u znaku znamená, že nad znakem je hlava; značka Z u k-tice znaků znamená, že k-tice je první na pásce. ω je speciální znak konce pásky. Znaky vyjadřující k-tice, kde na první pásce je nějaký znak z abecedy a na ostatních páskách jsou znaky Λ (nad žádným znakem není hlava), ztotožníme se znaky původní abecedy Σ. Tím se nám překódování vstupu zjednoduší na přepsání prvního znaku $\alpha \in \Sigma$ na pásce znakem $(\{\alpha, H\}, \{\Lambda, H\}, \ldots, \{\Lambda, H\}, Z)$ (všechny hlavy totiž na počátku stojí na začátku pásek). Uvědomte si, že překódování celého vstupu by nebylo vůbec jednoduché, protože stroj nemusí být schopen poznat, kde vlastně vstup končí – znak Λ může být korektní součástí vstupu.

Nyní jak bude stroj T' pracovat: Stroj vždy dojede hlavou na počátek pásky (ten pozná podle značky Z). Pak jede hlavou doprava, dokud nenarazí na značku odpovídající hlavě nad první páskou, zde si přečte znak pod hlavou na první pásce a znak si uloží do stavu. Pak se stroj opět přesune na počátek pásky a obdobným způsobem načítá znaky pro druhou až k-tou pásku. Nyní, když má stroj ve stavu uloženu celou k-tici znaků pod hlavami, není problém přejít do stavu, do kterého přejde původní stroj T a zároveň si ve stavu uložit znaky, které mají být zapsány a směry, kterými mají být posunuty hlavy. Pak stroj začne podobně jako při načítání jezdit po pásce, zapisovat nové znaky a posouvat značky pro hlavy. Když byly zapsány všechny znaky a posunuty všechny hlavy, začne stroj simulovat další krok původního stroje. Pokud stroj T' zjistí, že se nějaká hlava posunula nad znak ω, tak posune znak ω o jedno pole doprava a na jeho původní místo zapíše znak příslušné k-tice (tedy k znaků Λ s příslušnou značkou pro hlavu). Pokud stroj T' při posouvání hlav zjistí, že nějaká hlava narazila do levého okraje své pásky (to snadno pozná podle značky Z), tak projde celou pásku až po znak konce pásky ω a znaky na pásce přepíše znaky

odpovídajícími obsahu první pásky, a pak se ukončí nárazem hlavy do levého konce pásky. Všimněte si, že znak ω je nutný, protože jinak bychom při přepisování nepoznali, kde končí výstup. Takto ale máme jistotu, že za znakem ω už jsou pouze znaky Λ, které není třeba přepisovat. Počet stavů stroje T' bude zřejmě konstantní (řádově $c \cdot k \cdot 3^k \cdot |\Sigma|^k \cdot Q$ (Q je počet stavů původního stroje T) – tolik stavů potřebujeme, abychom si zapamatovali, do jakého stavu chceme přejít, kam posunout hlavy, jaké znaky zapsat a kolik pásek jsme již vyřídili).

14-4-1 Povodeň — Pavel Nejedlý

Při řešení této úlohy bylo klíčovým momentem spočítat, do jaké výšky na kterém čtverci vystoupí hladina. Jeden z efektivních algoritmů (mezi řešeními se vyskytly tři různé) je založen na Dijkstrově algoritmu: všem čtvercům na okraji pole přiřadím výšku hladiny rovnou maximu z jejich výšky a nuly (co kdyby to třeba byla prohlubeň), ostatním čtvercům nekonečno. Nyní vezmu z dosud nezpracovaných čtverců (na počátku jsou všechny čtverce nezpracované) ten s nejmenší výškou hladiny a všem dosud nezpracovaným čtvercům okolo nastavím výšku hladiny jako maximum z jejich výšky a výšky hladiny na onom vybraném čtverci. Druhý krok pak opakuji tak dlouho, dokud zbývají nějaké nezpracované čtverečky. Na vyhledání minima se použije halda, což nám zaručuje složitost $O(n^2 \cdot \log n)$. Všimněte si, že proti standardnímu Dijkstrovu algoritmu nastavujeme výšku hladiny na jenom čtverečku jen jednou (tj. pak už se nesnižuje), což nám umožňuje jisté zjednodušení práce s haldou.

```
#include <stdio.h>

#define MAXN 100
#define MOC 0x1000000
#define MAX (a, b) ( (a) < (b) ? (b) : (a))

int n;
int vyska[MAXN][MAXN];
int hladina[MAXN][MAXN];

struct {
    int i, j;
    int hladina;                                /* kopie hladina[i][j] */
} halda[MAXN*MAXN];

int hlen = 0;
int smery[4][2] = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

void swap (int p1, int p2) {
    int p;

#define SWAP (x, y) ( p = (x), (x) = (y), (y) = p )
    SWAP (halda[p1].i, halda[p2].i);
    SWAP (halda[p1].j, halda[p2].j);
    SWAP (halda[p1].hladina, halda[p2].hladina);
}
```

```
void extractMin () {
    int p;

    if (hlen < 2) {
        hlen = 0;
        return;
    }

    swap (1, hlen);
    hlen --;

    p = 1;
    while (2 * p <= hlen) {
        int k = 2 * p;

        if (k + 1 <= hlen && halda[k].hladina > halda[k + 1].hladina)
            k ++;

        if (halda[p].hladina <= halda[k].hladina)
            break;

        swap (p, k);
        p = k;
    }
}
void add (int i, int j) {
    int p;

    p = ++ hlen;
    halda[hlen].i = i;
    halda[hlen].j = j;
    halda[hlen].hladina = hladina[i][j];

    while (p > 1) {
        int k = p / 2;

        if (halda[k].hladina <= halda[p].hladina)
            break;

        swap (p, k);
        p = k;
    }
}
int spocti () {
    int objem = 0;
    int i, j;

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            hladina[i][j] = MOC;

    /* pridej kraje */

    for (i = 0; i < n; i++) {
        hladina[i][0] = MAX (0, vyska[i][0]); add (i, 0);
        hladina[i][n - 1] = MAX (0, vyska[i][n - 1]); add (i, n - 1);
    }
```

```
    for (j = n − 1; j > 0; j−−) {
      hladina[0][j] = MAX (0, vyska[0][j]); add (0, j);
      hladina[n − 1][j] = MAX (0, vyska[n − 1][j]); add (n − 1, j);
    }
    while (hlen > 0) {
      int i = halda[1].i;
      int j = halda[1].j;
      int h = halda[1].hladina;
      int k;
      extractMin ();
      for (k = 0; k < 4; k++) {
        int ii = i + smery[k][0];
        int jj = j + smery[k][1];
        if (ii >= n || ii < 0 || jj >= n || jj < 0)
          continue;                              /* mimo matici */
        if (hladina[ii][jj] != MOC)
          continue;                              /* uz nastaveno */
        hladina[ii][jj] = MAX (h, vyska[ii][jj]);
        add (ii, jj);
      }
    }
    /* spocti objem */
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        objem += hladina[i][j] − vyska[i][j];
    return objem;
}
void nacti () {
    int i, j;
    scanf ("%d", &n);
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        scanf ("%d", vyska[i] + j);
}
int main () {
    int objem;
    nacti ();
    if (n < 2)
      objem = 0;                                 /* trivialni pripady */
    else
      objem = spocti ();
    printf ("Mnozstvi zadrzene vody je %d\n", objem);
    return 0;
}
```

14-4-2 Posloupnost — Zdeněk Dvořák

Tahle úložka byla spíš z těch jednodušších, a přestože se vyskytla i řešení, která o sobě tvrdila, že pracují v čase $\Omega(n^3)$, žádné z nich o tom nedokázalo přesvědčit ani mě. Jak by tedy mohlo vypadat řešení:

Pro zjednodušení předpokládejme, že všechna čísla v posloupnosti jsou kladná. Budeme postupně probírat čísla posloupnosti a zjišťovat, zda se v ní nevyskytuje jejich 2. a 3. mocnina. Toto vyhledávání budeme provádět jednoduše postupným procházením posloupnosti podle velikosti. Tímto bychom dosáhli časové složitosti $O(n^2)$, což nechceme. Provedeme tedy drobné vylepšení – při hledání čísla se můžeme zastavit, když aktuálně probrané číslo je větší než hledané (posloupnost je rostoucí, tedy pak už nic nemůžeme najít). To nám ovšem na asymptotické složitosti nic nezmění.

Nyní ovšem využijeme toho, že funkce x^2, x^3 jsou rostoucí; z toho je zřejmé, že je-li $x_1 < x_2$, nemá smysl hledat x_2^2 před číslem, na kterém jsme se zastavili při hledání x_1^2 (všechna tato čísla jsou menší než x_1^2, a tím spíše než x_2^2). Budeme si tedy pamatovat, kde jsme skončili pro minulý prvek posloupnosti, a hledat budeme až od této pozice.

Paměťová složitost je zjevně lineární. Časová složitost je zajímavější, protože v programu se nám vyskytují do sebe zanořené smyčky; přesto je i časová složitost lineární, protože v každé iteraci vnitřní smyčky se ukazatele pozice pro x^2 a x^3 zvětší o 1, což mohou udělat maximálně n krát.

```
#include <stdio.h>

#define N 1000

int posl[N];
int n;

int main (void)
  {
    int i, f_nneg;
    int x, x2, x3;
    int xna2, xna3;

    scanf ("%d", &n);
    for (i = 0; i < n; i++)
      scanf ("%d", posl + i);

    for (f_nneg = 0; f_nneg < n; f_nneg++)
      if (posl[f_nneg] >= 0)
        break;

    /* Nezaporna cisla. */
    for (x = x2 = x3 = f_nneg; x3 < n; x++)
      {
        xna2 = posl[x] * posl[x];
        xna3 = xna2 * posl[x];

        while (x2 < n && xna2 > posl[x2]) x2++;
```

```
        while (x3 < n && xna3 > posl[x3]) x3++;
        if (xna2 == posl[x2] && xna3 == posl[x3])
          goto found;
      }
    /* Zaporna cisla. */
    for (x = x3 = f_nneg - 1, x2 = f_nneg; x3 >= 0 && x2 < n; x--)
      {
        xna2 = posl[x] * posl[x];
        xna3 = xna2 * posl[x];
        while (x2 < n && xna2 > posl[x2]) x2++;
        while (x3 >= 0 && xna3 < posl[x3]) x3--;
        if (xna2 == posl[x2] && xna3 == posl[x3])
          goto found;
      }
    printf ("Takove x neexistuje.\n");
    return 0;
    found:
    printf ("Nalezena cisla %d, %d, %d.\n", posl[x], posl[x2], posl[x3]);
    return 0;
  }
```

14-4-3 Koordinátor — Miroslav Rudišin

Túto úlohu lze previesť na hľadanie minima v kruhovej sieti, v ktorej sa koordinátorom stane počítač s najmenším ID. Predpokladajme, že máme k dispozici funkcie `ReceiveLeft` a `ReceiveRight` (realizované napr. buffermi správ pre oba smery), ktoré vrátia prvú nespracovanú správu v požadovanom smere. Na začiatku výpočtu všetky počítače kandidujú na koordinátora. Ak sa nejaký počítač dozvie o inom počítači s menším ID ako je jeho, prestane kandidovať a prijaté správy preposiela ďalej v pôvodnom smere. Voľba koordinátora prebieha po kolách. V každom kole všetci kandidáti odošlú svoje ID na obe strany a následne dostanú ID najbližších kandidátov. Ak ich ID nie je najmenšie, prestanú kandidovať. Voľba končí, ak prijaté ID sú rovnaké ako lokálne ID. To znamená, že v kruhu ostal jediný kandidát. Ten to oznámi ostatným počítačom; napríklad obežníkom, ktorý po prejdení kruhu skončí u neho. V kruhu s N ($N > 2$) kandidujucími počítačmi je N susediacich dvojíc, z ktorých v každom kole vypadne aspoň jeden počítač (ten s vetším ID). Preto sa počet kandidátov každým kolom aspoň zpoloviční. Teda počet kôl je najviac $\log N$. V každom sa pošle najvyše $2N$ správ. Preto výsledna zložitosť, meraná počtom správ, je $O(N \log N)$.

```
stav = kandidat
do
       if stav == kandidat then
              SendLeft(<KAMPAN,ID>)
```

```
                <typ,rID> = ReceiveRight;
                SendRight(<KAMPAN,ID>)
                <typ,lID> = ReceiveLeft;
                if lID == ID && rID == ID then
                        stav = koordinator
                        SendLeft(<VYHLASKA,ID>);
                fi
                if lID < ID || rID < ID then
                        stav = neaktivny
                fi
        fi

        if stav == neaktivny then
                <typ,rID> = ReceiveRight;
                SendLeft(<typ,rID>)
                if typ == VYHLASKA then
                        kID = rID;
                        break;
                fi
                <typ,lID> = ReceiveLeft;
                SendRight(<typ,lID>)
        fi

        if stav == koordinator then
                <typ,rID> = ReceiveRight;
                kID = ID;
                break;
        fi
forever
```

14-4-4 Triramidy — Aleš Přívětivý

Úloha byla jednoduchá a většina z vás si s ní hravě poradila. Úkolem bylo najít největší jedničkový rovnoramenný pravoúhlý trojúhelník v matici $A = \{a_{i,j}\}$ složené z nul a jedniček, jehož odvěsny mají svislý a vodorovný směr.

Pro zjednodušení se budeme nyní zajímat pouze o trojúhelníky mající pravý úhel vpravo dole, zbylé tři případy se vyšetří analogicky. Nechť tedy $t_{i,j}$ označuje velikost odvěsny největšího jedničkového trojúhelníku majicího pravý úhel vpravo dole na prvku $[i, j]$. Zřejmě platí:

$$t_{i,j} = \begin{cases} 0 & \text{když } a_{i,j} = 0 \\ \min(t_{i-1,j}, t_{i,j-1}) + 1 & \text{když } a_{i,j} = 1 \end{cases}$$

Pro korektnost si dodefinujme $t_{0,j} = 0$ a $t_{i,0} = 0$.

Všimněme si, že pro výpočet prvku $t_{i,j}$ potřebujeme prvky, které mají alespoň jeden z indexů menší. Budeme-li počítat $t_{i,j}$ postupně po řádkách, budeme mít vždy při určování $t_{i,j}$ spočítané všechny dílčí výsledky, a tedy budeme znát i hodnotu $t_{i,j}$. Nalezmeme-li pak $t_{i,j}$ s maximalní hodnotou, víme, že maximální jedničkový rovnoramenný pravoúhlý trojúhelník má dolní pravý úhel na prvku $[i, j]$ a odvěsny velikosti $t_{i,j}$. Ostatní natočení trojúhelníku vyřešíme podobně.

Bude-li mít prohledávaná matice A velikost $m \times n$, budeme muset spočítat $m \cdot n$ hodnot $t_{i,j}$. Každou z hodnot $t_{i,j}$ počítáme v konstantním čase, tedy celková časová složitost algoritmu je lineární k velikosti matice A, tj. $O(mn)$. Paměťová složitost je rovněž $O(mn)$.

```
#include <stdio.h>

#define MAXN 100
int a[MAXN][MAXN], t[MAXN][MAXN];    /* mapa a pole pro pocitani t[i][j] */
int m, n;                            /* rozmery mapy */
int maxo, maxi1, maxi2, maxi3,
          maxj1, maxj2, maxj3;       /* parametry prozatimniho nejlepsiho reseni */

inline long min (long a, long b)
{
      return (a < b) ? a : b;
}

int main ()
{
      /* Nacteni vstupu */
      scanf ("%d %d", &m, &n);
      for (int i=0; i<m; i++)
            for (int j=0; j<n; j++)
                  scanf ("%d", &a[i][j]);

      /* Hledani maximalniho trojuhelnika – pravy dolni roh */
      for (int i=0; i<m; i++)
        for (int j=0; j<n; j++)
        {
              t[i][j] = i==0||j==0 ?a[i][j]:a[i][j]* (min (t[i-1][j], t[i][j-1])+1);
              if (t[i][j] > maxo)
                    maxo = t[i][j], maxi1 = maxi2 = i + 1, maxi3 = i - maxo + 2,
                    maxj1 = j - maxo + 2, maxj2 = maxj3 = j + 1;
        }

      /* Hledani maximalniho trojuhelnika – levy dolni roh */
      for (int i=0; i<m; i++)
        for (int j=n-1; j>=0; j--)
        {
             t[i][j] = i==0||j==n-1 ?a[i][j]:a[i][j]* (min (t[i-1][j], t[i][j+1])+1);
             if (t[i][j] > maxo)
                  maxo = t[i][j], maxi1 = maxi2 = i + 1, maxi3 = i - maxo + 2,
                  maxj1 = maxj3 = j + 1, maxj2 = j + maxo;
        }

      /* Hledani maximalniho trojuhelnika – pravy horni roh */
      for (int i=m-1; i>=0; i--)
        for (int j=0; j<n; j++)
        {
              t[i][j] = i==m-1||j==0 ?a[i][j]:a[i][j]* (min (t[i+1][j], t[i][j-1])+1);
              if (t[i][j] > maxo)
                    maxo = t[i][j], maxi1 = maxi2 = i + 1, maxi3 = i + maxo,
                    maxj1 = j - maxo + 2, maxj2 = maxj3 = j + 1;
        }
```

```
    /* Hledani maximalniho trojuhelnika – levy horni roh */
    for (int i=m−1; i>=0; i−−)
      for (int j=n−1; j>=0; j−−)
      {
          t[i][j] = i==m−1||j==n−1 ?a[i][j]:a[i][j]* (min (t[i+1][j], t[i][j+1])+1);
          if (t[i][j] > maxo)
              maxo = t[i][j], maxi1 = maxi2 = i + 1, maxi3 = i + maxo,
              maxj1 = maxj3 = j + 1, maxj2 = j + maxo;
      }

    printf ("Nejvetsi stavba ma vrcholy v bodech (%d,%d), (%d,%d) a (%d,%d).\n",
            maxi1, maxj1, maxi2, maxj2, maxi3, maxj3);
    return 0;
}
```

14-4-5 Turingovy stroje — Honza Kára

Myšlenka vedoucí k řešení této úlohy napadla téměř všechny řešitele. Technické záležitosti při implementaci myšlenky ale zvládl málokdo. Nejčastějšími nedostatky vašich řešení bylo, že jste při kopírování obsahu pásky nepoznali konec kopírované části a začali jste kopírovat již zkopírovanou část (čímž jste vašemu stroji zajistili práci až do konce světa), či že jste nezvládli kopírování znaku Λ, který může být klidně součástí vstupu. Nikdo si pak též nevšiml drobných nedostatků v zadání – stroj nijak nedokáže zjistit, kde končí vstup, a tedy ani to, kam má posouvat obsah pásky. Stroj má také problémy, pokud se znak Λ vyskytuje uvnitř textu na pásce. Proto by správná odpověď puntičkáře na původní zadání měla znít: „Stroje nejsou ekvivalentní.“. My si proto do zadání doplníme požadavek, aby za posledním znakem vstupního slova byl ještě připsán speciální znak `]` a na výstupu budeme místo znaku Λ používat znak Λ'.

A nyní již ke vzorovému řešení. Idea řešení je jednoduchá. Budeme na našem stroji s děrnou páskou simulovat původní stroj tak, že vždy, když původní TS zapisuje na nějaké políčko, tak obsah pásky celý zkopírujeme do volného místa, ale již se změněnou hodnotou zapisovaného políčka. Původní obsah pásky při kopírování zakaňkujeme. Tato myšlenka má ale několik technických háčků, které se nyní pokusíme rozřešit.

Protože si budeme potřebovat pamatovat, na jaké pozici stojí hlava, budeme mít od každého znaku dvě verze – s hlavou a bez hlavy. Dále budeme potřebovat mít nějak označený levý konec pásky (abychom nechtěně neukončili běh stroje). To nejsnáze uděláme tak, že celé vstupní slovo (tzn. až po znak `]`) zkopírujeme těsně za znak `]` a původní vstup zakaňkujeme. Kaňky nám budou značit levý konec pásky. Při tomto kopírování první znak zapíšeme ve formě „s hlavou“ a protože navíc budeme potřebovat odlišit znak Λ uvnitř vstupního slova a mimo něj, budeme při prvním kopírování znak Λ zapisovat jako

nějaký nový znak Λ'. Náš stroj se zřejmě může při simulaci chovat, jako by místo znaků Λ' četl znaky Λ. Jak nyní bude přesně probíhat kopírování s přepisem jednoho znaku? Do stavu stroje si zapamatujeme písmeno, které má být zapsáno, a posun hlavy. Nyní stroj jede doleva až k nejbližší kaňce (počátek „aktivní“ pásky), tam si načte do stavu znak a zakaňkuje ho. Znak pak přenese na první volné pole vpravo (nezapomeňte, že znaky Λ se díky jejich nahrazení uvnitř slova skutečně budou vyskytovat až za znakem `]`). Takto v kopírování pokračujeme s tím, že když kopírujeme znak, kde má nově být hlava (to umíme poměrně snadno zjistit – ve stavu si pamatujeme, jakým směrem se má pohnout hlava a vždy před kopírováním znaku jen zkontrolujeme, jestli náhodou není na příslušném sousedním políčku hlava), tak zapíšeme do kopie formu znaku s hlavou a pokud kopírujeme znak, který původně byl pod hlavou, tak místo něj zapíšeme znak zapamatovaný ve stavu. Kopírování končí zkopírováním znaku `]` (pokud na této pozici má nově ležet hlava, tak na ni místo `]` zapíšeme znak Λ' a znak `]` zapíšeme až na následující pozici). Simulace stroje pak končí v okamžiku, kdy by se hlava měla přesunout nad kaňku na levém okraji pásky (na původním stroji tedy přes levý okraj pásky) – v tomto okamžiku už nám stačí jen výstupní slovo naposledy zkopírovat, přitom odstranit označení znaku s hlavou a znak `]`, a pak jet hlavou neustále doleva, až narazí na okraj pásky.

14-5-1 Nové slunce — Jan Kára

Tato úloha patřila k těm dosti těžkým. Vaše nejlepší funkční řešení (a že jich mnoho nebylo) dosahovala složitosti $O(N^3)$. Za tato řešení bylo možno získat až 10 bodů z 12. Za řešení v čase $O(N^4)$ pak bylo možno získat až 9 bodů. Nefunkční řešení pak podle míry nefunkčnosti a kvality popisu mohla získat až 5 bodů.

A nyní již k řešení. Nejdříve si učiníme jednoduché pozorování: Na hranici nejmenší kružnice, která obsahuje všechny body, musí ležet buď dva nejvzdálenější body – označme si je A_1 a A_2 – a střed kružnice je ve středu úsečky A_1A_2 nebo alespoň tři body. Toto pozorování platí proto, že pokud na hranici kružnice neleží žádný nebo jeden bod, lze kružnici snadno zmenšit tak, že se žádný bod nedostane ven. Pokud na hranici kružnice leží dva body, lze kružnici zmenšit právě když střed kružnice neleží uprostřed spojnice těchto bodů.

Z výše uvedeného pozorování snadno odvodíme algoritmus běžící v čase $O(N^4)$ a s trochou snahy i algoritmus běžící v čase $O(N^3)$ – oba algoritmy naleznou dvojici nejvzdálenějších bodů v čase $O(N^2)$ a vyzkouší, zda kružnice se středem uprostřed mezi těmito body a obsahující tyto dva body na hranici neobsahuje všechny body. Pokud ano, máme zřejmě hledaný střed (menší kružnice totiž zřejmě existovat nemůže). Pokud ne, tak algoritmus prostě vyzkouší všechny trojice bodů. Ke každé trojici sestrojí kružnici opsanou oněm třem bodům, otestuje, zda obsahuje všechny body a pokud ano, tak ji porovná

s nejmenší dosud nalezenou kružnicí. Tento postup lze poměrně snadno naimplementovat v čase $O(N^3)$. Protože míříme k poněkud vyšším metám, nebudu se detaily implementace tohoto algoritmu zabývat.

Pro jednoduchost nadále předpokládejme, že žádné čtyři body neleží na jedné kružnici (mohli bychom se obejít i bez tohoto předpokladu, ale situace by se nám poněkud zkomplikovala). Náš algoritmus je založen na jednoduché rekurzivní funkci `MinKruh(V,H)`. Ta má jako parametry dvě množiny bodů – H je množina bodů, které mají ležet na hledané kružnici, V je množina bodů, které mají ležet uvnitř nebo na hranici. Funkce pak vrací nejmenší kružnici, která splňuje výše uvedené požadavky. Na počátku voláme funkci s parametry $M, \emptyset$ (M je množina bodů, pro které máme kružnici nalézt).

A jak funkce pracuje? Pokud už je množina V prázdná, jen sestrojíme kružnici opsanou bodům v H a jsme hotovi (protože v H budou vždy nejvýše tři body, nebude sestrojení kružnice problém). Jinak vybereme náhodný bod z V, odebereme ho a rekurzivně si necháme nalézt kružnici pro menší V. Pokud nalezená kružnice obsahuje i zvolený bod, nemusíme dále nic dělat a kružnici pouze vrátíme. Pokud kružnice bod neobsahuje, musí bod nutně ležet na hranici nejmenší kružnice pro množinu $V \cup H$. Zařadíme ho tedy do H a rekurzivně se zavoláme na menší V a větší H – kružnice, kterou nám vrátí toto volání, bude zaručeně naše hledaná nejmenší kružnice. Všimněte si, že v H nikdy nebudou více jak tři body – pokud nějaký bod dáme do H, tak už musí ležet na hranici nejmenší kružnice. Na té jsou ale nejvýše tři body.

Správnost algoritmu jsem se snažil vysvětlovat v popisu, takže nám už chybí jen odhad časové složitosti. S tou to bude trochu obtížnější. V nejhorším případě může být složitost výše uvedeného algoritmu až exponenciální (pokaždé si vybereme bod z hranice kružnice). V průměrném případě na tom ale budeme o mnoho lépe – časová složitost bude lineární ($O(n)$, kde n je počet bodů). A jak se to spočítá? Inu, zkusme to následovně: Označme $T(n,k)$ průměrný čas potřebný ke zpracování množiny V o velikosti n a množiny H o velikosti k. Zřejmě platí, že $T(0,k) = c$, kde c je vhodná konstanta odpovídající času na nalezení kružnice procházející třemi body. Dále platí:

$$T(n,k) \leq T(n-1,k) + (3-k)/n \cdot T(n-1,k+1) + d$$

První člen součtu je za první rekurzivní volání, druhý člen je za druhé volání. To ovšem proběhne pouze pokud jsme špatně zvolili bod, a to se stalo právě s pravděpodobností $(3-k)/n$ (v naší množině n bodů je právě $3-k$ špatných bodů z hranice). d je pak vhodná konstanta odpovídající času na testování, zda bod leží uvnitř kružnice a podobně. Když si nyní dáme trochu práce a rekurenci vyřešíme, vyjde nám skutečně, že $T(n,k) = O(n)$. Na závěr diskuse složitosti bych ještě poznamenal, že existují i algoritmy se zaručenou složitostí $O(n \log n)$. Ty už používají poněkud komplikovanějších technik.

Program je přímou implementací algoritmu. Pouze množiny jsou reprezentovány seznamem prvků v poli a pole jsou předávána odkazem (předávání polí hodnotou by nám zkazilo časovou složitost). Při testování, zda bod leží v kružnici, pak zbytečně neodmocňujeme, nýbrž porovnáváme druhé mocniny vzdáleností.

```
program Slunce;
const
  MAXB=200;
type
  Bod = record
          x,y : Real;
        end;
  PoleBodu = Array[1..MAXB] of Bod;
  PoleMna = Array[1..MAXB] of Integer;
  Kruznice = record
               x, y, r : Real;
             end;
var
  Bodu : Integer;       {Počet bodů}
  Body : PoleBodu;      {Pole s body}
  V, H : PoleMna;       {Pole se seznamem bodu patřících do množiny V a H}
  VN, HN : Integer;     {Počty prvků v množinách V a H}
  K : Kruznice;         {Výsledná nejmenší kružnice}

{Inicializace}
procedure Init;
var
  i : Integer;
begin
  Write('Pocet bodu: ');
  ReadLn(Bodu);
  for i := 1 to Bodu do begin
    Write('Souradnice: ');
    ReadLn(Body[i].x, Body[i].y);
  end;
  HN := 0;
  VN := Bodu;
  for i := 1 to Bodu do
    V[i] := i;
end;

{Vytvoří kružnici obsahující dané body na hranici}
function ProlozKruznici(N : Integer; var B : PoleMna) : Kruznice;
var
  K : Kruznice;
  s, t : Real;
  U, V : Bod;
begin
  if N <= 1 then begin
   K.x := 0;
   K.y := 0;
   K.r := 0;
  end
  else if N = 2 then begin
    K.x := (Body[B[1]].x+Body[B[2]].x)/2;
```

```
    K.y := (Body[B[1]].y+Body[B[2]].y)/2;
    K.r := sqrt(sqr(Body[B[1]].x-Body[B[2]].x)+sqr(Body[B[1]].y-Body[B[2]].y))/2;
  end
  else begin
    {Spočteme střed kružnice opsané}
    U.x := Body[B[2]].x-Body[B[1]].x;
    U.y := Body[B[2]].y-Body[B[1]].y;
    V.x := Body[B[2]].x-Body[B[3]].x;
    V.y := Body[B[2]].y-Body[B[3]].y;
    {Alespoň jedno z čísel U.x a U.y bude nenulové}
    if U.x <> 0 then begin
      s := ((Body[B[1]].x-Body[B[3]].x)/2-U.y/U.x*(Body[B[3]].y-Body[B[1]].y)/2)
           /(U.y*V.x/U.x-V.y);
      K.x := (Body[B[2]].x+Body[B[3]].x)/2-s*V.y;
      K.y := (Body[B[2]].y+Body[B[3]].y)/2+s*V.x;
    end
    else begin
      t := ((Body[B[1]].x-Body[B[3]].x)/2+V.y/V.x*(Body[B[1]].y-Body[B[3]].y)/2)
           /(U.y-V.y*U.x/V.x);
      K.x := (Body[B[2]].x+Body[B[1]].x)/2-t*U.y;
      K.y := (Body[B[2]].y+Body[B[1]].y)/2+t*U.x;
    end;
    K.r := sqrt(sqr(K.x-Body[B[1]].x)+sqr(K.y-Body[B[1]].y));
  end;
  ProlozKruznici := K;
end;

{Zjistí, zda daný bod leží uvnitř kružnice}
function LeziUvnitr(K : Kruznice; B : Integer) : Boolean;
begin
  if sqr(K.x-Body[B].x)+sqr(K.y-Body[B].y) <= sqr(K.r) then
    LeziUvnitr := True
  else
    LeziUvnitr := False;
end;

{Základní rekurzivní funkce}
function MinKruh(VN, HN : Integer; var V, H : PoleMna) : Kruznice;
var
  K : Kruznice;
  VBodI, VBod : Integer;        {Vybraný bod k vypuštění}
begin
  if VN = 0 then
    MinKruh := ProlozKruznici(HN, H)
  else begin
    VBodI := Trunc(Random*(VN-1)) + 1;
    VBod := V[VBodI];
    V[VBodI] := V[VN];
    Dec(VN);
    K := MinKruh(VN, HN, V, H);

    if not LeziUvnitr(K, VBod) then begin
      Inc(HN);
      H[HN] := VBod;
      K := MinKruh(VN, HN, V, H);
      Dec(HN);
    end;
    {Vrátíme pole do původního stavu}
```

```
    Inc(VN);
    V[VN] := V[VBodI];
    V[VBodI] := VBod;
    {Vrátíme výsledek}
    MinKruh := K;
  end;
end;

begin
  Init;
  K := MinKruh(VN, HN, V,H);
  WriteLn('Souradnice stredu: ', K.x:2:2, ' ', K.y:2:2);
  WriteLn('Polomer: ', K.r:2:2);
end.
```

14-5-2 Tramvaje — Tomáš Vyskočil

Tato úloha s veleznámým cestovatelem Dvoukvítem se dala řešit různě. Jistě by v tom každý z vás našel Dijkstru, mnozí hravější našli i jiné, mnohdy efektivnější, algoritmy, a tak vypadá i vzorové řešení.

A teď již k řesení. Nejdříve bylo dobré si pohrát se zadaným problémem a zjistit nějaké zákonitosti, které zde platí. Po chvilce uvažování přijdete na to, že se využívají jenom směry od startu ke konci (jet zpět se nikdy nevyplatí). Tedy pokud máme startovní vrchol v levém horním rohu a konec v pravém dolním, potom používáme pouze cesty dolů a doprava. Tím si značně ulehčíme práci. A abychom nemuseli řešit spousty okrajových situací, všechna uskupení si převedeme rotacemi na situaci, kdy je začátek vpravo nahoře a konec vlevo dole. A nyní již stačí projít síť zleva doprava a shora dolů trochu upraveným prohledáváním do šířky. U každého uzlu si budeme počítat nejmenší čas, ve kterém jsme schopni z daného uzlu vyrazit v x-ovém a y-ovém směru. Pak už jen stačí projít a vypsat cestu, která dosáhla nejkratšího času (tu snadno zrekonstruujeme tak, že půjdeme z cíle vždy na políčko, ze kterého šlo vyjet nejdříve).

Správnost algoritmu zřejmě plyne z toho, že směry, které má smysl používat, jsou pouze shora dolů a zleva doprava. Časová složitost algoritmu je $O(N \cdot M)$ a paměťová je $O(N \cdot M)$.

```
#include <stdio.h>

#define MAX_N   1000

int M, N, in, d, pr, ti;
int sx, sy, zx, zy, kx, ky;

/* Spočte čas, za který přijede nejbližší tramvaj */
int next_tram(int x, int t)
{
        int h=t-x*d;
        return (h<=0)?-h:((h%in)?in-h%in:0);
}
```

```
int rotatex(int x)
{
        return (sx?(N-x-zx):(x+zx))-1;
}

int rotatey(int y)
{
        return (sy?(M-y-zy):(y+zy))-1;
}

int main(void)
{
        int timex[MAX_N][MAX_N], timey[MAX_N][MAX_N];
        int x, y, h;

        scanf("%d %d %d %d %d %d", &M, &N, &in, &d, &pr, &ti);
        scanf("%d %d %d %d", &zx, &zy, &kx, &ky);

        if (sx=(zx>kx)){
                zx=N-zx;
                kx=N-kx;
        }
        if (sy=(zy>ky)){
                zy=M-zy;
                ky=M-ky;
        }

        timex[0][0]=next_tram(0, ti);
        timey[0][0]=next_tram(0, ti);

        for (x=1; x<kx-zx; x++){
                timex[x][0]=timex[0][0]+x*d;
                timey[x][0]=timex[x][0]+pr+next_tram(0, timex[x][0]+pr);
        }
        for (y=1; y<ky-zy; y++){
                timey[0][y]=timey[0][0]+y*d;
                timex[0][y]=timey[0][y]+pr+next_tram(0, timey[0][y]+pr);
        }

        for (x=1; x<=kx-zx; x++){
                for (y=1; y<=ky-zy; y++){
                        h=timey[x][y-1]+pr+next_tram(x, timey[x][y-1]+pr+d);
                        if (timex[x-1][y]<h){
                                timex[x][y]=timex[x-1][y]+d;
                        }else{
                                timex[x][y]=h+d;
                        }
                        h=timex[x-1][y]+pr+next_tram(y, timex[x][y-1]+pr+d);
                        if (timey[x][y-1]<h){
                                timex[x][y]=timex[x][y-1]+d;
                        }else{
                                timex[x][y]=h+d;
                        }
                }
        }

        x=kx-zx; y=ky-zy;
```

```
        printf("%d %d\n", rotatex(kx), rotatey(ky));
        while (x!=0 || y!=0){
                if (y>0 && timex[x-1][y]>timey[x][y-1])
                        y--;
                else
                        x--;
                printf("%d %d\n", rotatex(x+zx), rotatey(y+zy));
        }

        return 0;
}
```

14-5-3 Zapeklitý kabel — Daniel Kráľ

Nejprve krátce k bodování této úlohy: Nezbytnou součástí řešení této úlohy měl být i (rychlý) program, který radí technikovi firmy Shumm & Brumm, jak kabely spojovat. K získání plného počtu 9 bodů bylo tedy nejen nutné vymyslet optimální postup spojování kabelů (tedy s 2 návštěvami pekla), ale i program s lineární časovou složitostí, který optimální postup vygeneruje a pak vyhodnotí.

Nechť N označuje počet vodičů v položeném kabelu. Povšimněme si, že pro $N = 1$ je úloha triviální a pro $N = 2$ je úloha naopak neřešitelná. Omezme se tedy na ty případy, pro které platí $N \geq 3$. Řešme nejdříve případ, kdy N je liché. Při první návštěvě pekla spojíme prvních $\lceil N/2 \rceil$ dvojic kabelů, tj. a_1–a_2, a_3–a_4, a_5–a_6, …, a_{N-2}–a_{N-1}. Na druhé straně pak měřením zjistíme, které dvojice kabelů jsou pospojovány do dvojic a který kabel není v dvojici (a tedy odpovídá konci a_N v pekle). Při druhé návštěvě pekla vytvoříme opět $\lceil N/2 \rceil$ dvojic kabelů, ale nyní to budou dvojice a_2–a_3, a_4–a_5, a_6–a_7, …, a_{N-1}–a_N. Protože víme, který konec $b_?$ odpovídá konci a_N, můžeme určit, který konec $b_?$ odpovídá a_{N-1}. Z výsledků měření v prvním kroku pak můžeme určit, který konec $b_?$ odpovídá a_{N-2} (je to ten konec $b_?$, co byl s koncem $b_?$ odpovídajícím a_{N-1} spárován v prvním kroku). Nyní lze určit konec $b_?$ odpovídající a_{N-3} (je nyní spárován s koncem $b_?$ odpovídajícím a_{N-2}), a takto pokračujeme až ke konci $b_?$ odpovídajícímu a_1.

Pokud je N sudé, je postup potřeba malinko upravit, a to následovně: V prvním kroku ponecháme vodiče a_{N-1} a a_N nespárovány, tj. vytvoříme $N/2 - 1$ párů vodičů. V druhém kroku ponecháme nespárovány vodiče a_1 a a_N. Konec $b_?$, který byl nespárován v obou krocích, zřejmě odpovídá konci a_N. Druhý konec, který byl nespárován v prvním kroku, odpovídá a_{N-1}. Zbytek postupu je stejný jako v případě, že by N bylo liché.

Zbývá domyslet, jak rychlý může být program, který výše uvedený postup navrhne a vyhodnotí výsledky technikova měření. Samotné vypsání dvojic vodičů, které je třeba spojit v prvním a druhém kroku, je triviální. Při vyhodnocování měření v druhém kroku potřebujeme následující:

- Znát jeden či dva (v závislosti na paritě N) vodiče, které byly v prvním kroku nespárovány.
- Ke konci $b_?$ umět rychle (tj. v čase $O(1)$) najít druhý konec $b_?$, se kterým byl spárován v prvním kroku.

První bod je snadný – do speciálních proměnných `slepy` a `slepy2` si uložíme indexy i takové, že b_i v prvním kroku nebyl vodivě spojen s jiným koncem $b_?$. K dosažení druhého bodu, si v prvním kroku naplníme pomocné pole `pary` následovně: Hodnota `pary[i]` je rovna 0, pokud vodič b_i není spárován. Jinak obsahuje (jednoznačně určený) index j takový, že vodiče b_i a b_j jsou spárovány (přesněji jim odpovídající konce $a_?$ vodivě spojeny). S pomocí tohoto pole je snadné vyhodnotit měření v druhém kroku (postupem popsaným v druhém odstavci) v čase $O(N)$. Paměťová složitost navrženého algoritmu je též $O(N)$ (je třeba uchovat pole `pary`).

```
program peklo;
const MAXN=10000;
var N: longint;
    i: longint;
    pary: array[1..MAXN] of longint;
    spojen: array[1..MAXN] of longint;
    posledni, slepy, slepy2: longint;
begin
  write('Zadejte pocet kabelu: ');
  readln(N);
  if N=1 then { Případ N=1 }
    begin
      writeln('Uloha je trivialni.');
      halt;
    end;
  if N=2 then { Případ N=2 }
    begin
      writeln('Uloha je neresitelna.');
      halt;
    end;
  { 1. krok }
  for i:=1 to (N-1) div 2 do writeln('Propoj a_',2*i-1,' s a_',2*i,' !');
  for i:=1 to N do pary[i]:=0;
  slepy:=0;
  if (N mod 2)=0 then slepy2:=0;
  for i:=1 to N do
    if pary[i]=0 then
      begin
        writeln('Se kterym vodicem b_?? je vodive propojen b_',i,'? [0 = s zadnym]');
        readln(pary[i]);
        if pary[i]<>0 then
          pary[pary[i]]:=i
        else
          if slepy=0 then
            slepy:=i
          else
            slepy2:=i;
      end;
  { 2. krok }
```

```
  writeln('Zrus vsechna propojeni!');
  for i:=1 to (N-1) div 2 do writeln('Propoj a_',2*i,' s a_',2*i+1,' !');
  { for i:=1 to N do spojen[i]:=0; }
  if (N mod 2)=0 then
    begin
      writeln('S kolika vodici je vodic b_',slepy,' spojen? S 0 nebo 1?');
      readln(i);
      if i=0 then
        begin
          spojen[N]:=slepy;
          slepy:=slepy2;
        end
      else
        spojen[N]:=slepy2;
      posledni:=N-1;
    end
  else
    posledni:=N;
  { A jdeme po dvojicich zpatky ! }
  while posledni>1 do
    begin
      writeln('Se kterym vodicem b_?? je spojen b_',slepy,'?');
      readln(i);
      spojen[posledni]:=slepy;
      spojen[posledni-1]:=i;
      slepy:=pary[i];
      posledni:=posledni-2;
    end;
  spojen[posledni]:=slepy;
  { Vypiseme vysledek. }
  for i:=1 to N do
    writeln('Vodic a_',i,' je spojen s b_',spojen[i],'.');
end.
```

14-5-4 Turingovy stroje — Honza Kára

První části úlohy – tj. zjistit, zda Turingovu stroji bude k výpočtu stačit počet políček omezený nějakým daným k – se všichni zhostili úspěšně (ačkoliv více či méně efektivně). S druhou částí úlohy to ale bylo o dost horší a pouze jediné řešení bylo správně.

Ověřit, zda Turingovu stroji bude k výpočtu stačit k políček, je úloha poměrně jednoduchá. Budeme postupně simulovat Turingův stroj. Pokud někdy během simulace bude stroj chtít vstoupit na $k+1$-ní políčko, řekneme, že stroji k políček nestačilo. Pokud během simulace stroj skončí (hlava narazí do levého okraje pásky), stroji zjevně k políček stačilo. Jediná netriviální otázka je, jak vyřešit případ, kdy se Turingův stroj zacyklí. I to lze ale vyřešit poměrně elegantně – celkový stav stroje (a tím i celý následující výpočet) je zřejmě v každém okamžiku jednoznačně určen obsahem prvních k políček pásky, vnitřním stavem stroje a pozicí hlavy. Pokud má stroj q stavů a a písmen v abecedě, je možných celkových stavů stroje tedy $a^k \cdot k \cdot q$. Pokud tedy stroj vykoná alespoň $a^k \cdot k \cdot q + 1$ kroků, musel se zaručeně do nějakého celkového stavu dostat

alespoň dvakrát, a to znamená, že se musel zacyklit. Stačí tedy při simulaci počítat počet kroků stroje a po určitém počtu kroků již s jistotou víme, že se stroj musel zacyklit.

Otázka, zda stroji stačí nějaký omezený počet políček (ale tento počet nemáme dán), je neřešitelná. Pokud bychom tento problém uměli vyřešit, uměli bychom totiž vyřešit i tzv. Halting problém – tedy problém, zda se daný Turingův stroj na daném vstupu někdy zastaví. Mohli bychom se totiž zeptat, zda danému Turingovu stroji stačí omezená paměť (na to bychom právě použili algoritmus z předpokladů). Pokud je odpověď záporná, stroj se zjevně nikdy nemůže zastavit. Pokud je odpověď kladná, můžeme stroj simulovat s omezeným počtem políček, stejně jako je popsáno v předchozím odstavci. Pokud simulace skončí s tím, že stroj chtěl použít více políček, zkusíme ho pustit s prostorem o políčko větším. Jinak simulace může skončit buď tím, že se stroj zacyklil, nebo že doběhl. V obou případech pak stačí dát příslušnou odpověď (tedy stroj neskončí pro první případ, resp. skončí pro druhý). Protože počet políček, která stroj potřebuje, je omezený, zřejmě musí jednou simulace skončit buď oznámením o zacyklení či skončení.

Protože si nyní ukážeme, že Halting problém je neřešitelný, nemůže mít řešení ani náš problém. Pro spor předpokládejme, že Halting problém je řešitelný. V tom případě existuje nějaký Turingův stroj $H(S, V)$, který přijme, právě když se stroj S zastaví na vstupu V. My si nyní sestrojíme pomocný stroj $H'(S)$, který přijme právě když se stroj S zastaví na vstupu S (tedy když stroji S na vstup podstrčíme jeho vlastní zápis). Ze stroje $H'(S)$ pak můžeme snadno sestrojit stroj $A(S)$, který se zastaví, právě když H' odmítl S (tedy právě když se stroj S na vstup S nezastavil) – prostě necháme běžet stroj H' a pokud by chtěl skončit přijmutím, tak se zacyklíme, jinak skončíme. Nyní ale přichází záludná otázka a s ní i spor: Co udělá stroj A na vstup A? Z konstrukce se A zastaví právě když H' odmítl A, a to je právě když se A na vstup A nezastavil. A se tedy nemůže ani zastavit ani nezastavit. Nemůže tedy existovat, a proto nemůže existovat ani původní stroj H, ze kterého jsme stroj A snadno sestrojili.

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
			max.	24	296
1.	Lukáš Turek	G Zborovská, Praha	3	20	206
2.	Josef Cibulka	G Štěpánská, Praha	4	23	199
3.	Jiří Danihelka	SPOE Písek	3	18	169
4.	Zbyněk Falt	G Žďár nad Sázavou	1	23	150
5.	Martin Hamrle	G Pelhřimov	4	15	129
6.	Jiří Štěpánek	G Tř. kpt. Jaroše, Brno	2	14	126
7.	Milan Straka	G Strakonice	3	16	114
8.	Petr Škoda	G Ústavní, Praha	2	15	104
9.	Petr Soběslavský	G J. Heyrovského, Praha	1	23	99
10.-11.	Jindřich Flídr	G Lanškroun	2	15	96
	Jan Havlíček	G Zborovská, Praha	3	14	96
12.	Daniel Lessner	G B. Bolzana, Praha	3	17	93
13.	Jaroslav Havlín	G Sedlčany	2	14	89
14.	Peter Bella	G J. Hronca, Bratislava	4	11	84
15.	Martin Demín	G Nitra	2	21	76
16.	Marek Sterzik	SPŠ Ostrov	3	9	74
17.	David Matoušek	G Arcus, Praha	2	13	71
18.	Jiří Paleček	G Nám. E. Beneše, Kladno	3	8	69
19.	Zuzana Vydrová	G Tanvald	3	16	68
20.	Jozef Tvarožek	G J. Hronca, Bratislava	4	7	64
21.	Jan Křetínský	G M. Lercha, Brno	2	8	59
22.	Pavel Čížek	G Kralupy n. Vltavou	3	5	57
23.	Petr Turbek	G J. Barranda, Beroun	2	14	56
24.	Anton Repko	G Sv. Mikuláša, Prešov	1	14	55
25.-26.	Alexandr Kazda	G Nad alejí, Praha	2	7	54
	Peter Šufliarsky	G Nové Zámky	2	12	54
27.-28.	Martin Lopatář	G Tř. kpt. Jaroše, Brno	2	5	52
	Jan Matoušek	G J. Wolkera, Prostějov	2	9	52
29.	Marie Zachovalová	G L. Jaroše, Holešov	4	8	49
30.	Pavol Polačko	G Poštová, Košice	3	7	47
31.-32.	Vojtěch Kovář	G Břeclav	2	6	46
	Hana Kozelková	G Opava	3	13	46
33.	Martin Dobroucký	G Moravská Třebová	1	9	45
34.	Ondrej Hirjak	G J. A. Raymana, Prešov	4	11	44
35.	Václav Cvíček	G Frýdek-Místek	3	6	42
36.	Pavel Bazika	G Nad Kavalírkou, Praha	4	7	39

37.-38.	Josef Sedlačík	G Uherský Brod	3	7	36
	Pavel Troubil	G Tř. kpt. Jaroše, Brno	2	6	36
39.	Jan Bulánek	G J. Vrchlického, Klatovy	1	12	35
40.	Ján Mazák	G Poštová, Košice	4	4	31
41.-42.	Petr Baudiš	G Ad Fontes, Jihlava	2	8	30
	Michal Štefkovič	G Ľ. Štúra, Trenčín	3	10	30
43.-44.	Martin Kruliš	G Kolín	3	4	29
	Matěj Skopový	G Česká Lípa	3	6	29
45.	Jozef Matějička	G Sv. Františka, Žilina	3	7	28
46.	Tomáš Staněk	G Volgogradská, Ostrava	3	6	27
47.	Jakub Horák	G Frenštát pod Radhoštěm	1	5	26
48.-49.	Tomáš Dzetkulič	G P. Horova, Michalovce	4	3	25
	Michal Tekeľ	G Konštantínova, Prešov	4	5	25
50.-51.	Jakub Galgonek	G Frýdek-Místek	4	4	22
	Jan Matějek	G Nám. E. Beneše, Kladno	2	3	22
52.-53.	Petr Havlíček	G Voděradská, Praha	3	4	21
	Michal Potfaj	G Nové Mesto nad Váhom	2	6	21
54.	Jan Hladký	G Tř. kpt. Jaroše, Brno	3	3	19
55.	Stanislav Basovník	G Kroměříž	1	4	18
56.	Jakub Nezveda	SPŠ V Úžlabině, Praha	1	5	17
57.	Petr Paščenko	G Dašická, Pardubice	2	4	16
58.	Tomáš Kučera	G Voděradská, Praha	3	4	14
59.-60.	David Hauzar	G Vimperk	2	6	13
	Martin Salaj	G J. M. Hurbana, Čadca	3	6	13
61.-62.	Jan Doubek	G Vimperk	2	5	12
	Bejnamin Vejnar	G Nymburk	2	4	12
63.-64.	Jaroslav Kudlička	G Hodonín	3	4	11
	Oto Petřík	G Vrchlabí	1	2	11
65.-67.	Petr Los	G Hranice na Moravě	3	2	10
	Vojtěch Šádek	G Hranice na Moravě	2	3	10
	Pavel Zemek	G Humpolec	3	3	10
68.-69.	Vojtěch Barta	G Slezská Ostrava	0	4	9
	Peter Novotný	G Martin	4	3	9
70.-71.	Pavol Jusko	G Alejová, Košice	2	2	8
	Petr Pospíšil	SPŠ Bruntál	3	3	8
72.-79.	Jan Kaštil	G J. Škody, Přerov	3	2	7
	Martin Komoň	G Valašské Meziříčí	1	3	7
	Vladimír Lhotský	G Svitavy	2	3	7
	Lukáš Okoun	Městské G, Bruntál	3	2	7
	Kamil Paulíny	G Poštová, Košice	4	1	7
	David Sulaiman	G Pelhřimov	1	3	7

Pořadí řešitelů

	Alexander Šimko	SPŠE Prešov	2	1	7
	Radek Vystrčil	G Táborská, Brno	2	2	7
80.	Radek Frolík	G Nám. E. Beneše, Kladno	3	1	6
81.	Kateřina Bambušková	SPŠ Bruntál	3	2	5
82.	Tomáš Hubálek	G Kralupy n. Vltavou	-1	1	4
83.-85.	David Irschik	G Ledeč nad Sázavou	1	1	3
	Michal Novotný	G Svitavy	2	4	3
	Jaromír Vojíř	G Ledeč nad Sázavou	1	1	3
86.-87.	Pavel Vlašánek	SPŠ Bruntál	1	4	1
	Lukáš Vlk	SOŠT Glaverbel, Teplice	3	1	1
88.-102.	Lukáš Cedrych	ZŠ U Roháč. kas., Praha	-1	0	0
	Jonáš Fiala	G B. Bolzana, Praha	3	0	0
	Lukáš Hron	G Dašická, Pardubice	3	0	0
	Martin Kaman	G Velké Meziříčí	2	0	0
	Lucie Kučerová	G Hořovice	2	0	0
	Vladimír Lazor	G Bardejov	2	0	0
	Marek Lipták	G Přípotoční, Praha	3	0	0
	Daniel Marek	ZŠ Filosofská, Praha	-1	0	0
	Marek Pavlů	SPŠ Litovel	?	0	0
	Jakub Plšek	G Tř. kpt. Jaroše, Brno	2	0	0
	Daniel Sedláček	ZŠ 1. máje, Havířov	-2	0	0
	Filip Sedlák	G Řečice, Brno	1	0	0
	Zdeněk Softič	G Vídeňská, Brno	3	0	0
	Jaromír Šimek	G Kojetín	3	0	0
	Jan Vlastník	G Dašická, Pardubice	4	0	0

Obsah

Martin Mareš a kolektiv

Korespondenční seminář z programování XIV. ročník

Autoři a opravující úloh:
Jakub Bystroň, Zdeněk Dvořák, Pavel Machek,
Pavel Nejedlý, Jan Kára, Daniel Kráľ,
Aleš Přívětivý, Miroslav Rudišin, Pavel Šanda,
Miroslav Trmač, Tomáš Valla, Tomáš Vyskočil

Vydala Univerzita Karlova v Praze, Matematicko-fyzikální fakulta
Oddělení vnějších vztahů a propagace
Ke Karlovu 3, 121 16 Praha 2
Praha 2002

Písmem Computer Modern v programu TEX vysázel Martin Mareš

Korektury provedla Karolína Šimková

Titulní obrázek Penroseova dláždění vygeneroval program Roberta Babilona

Vytiskla Tiskárna P. Flodr

84 stran, 2 obrázky
Vydání první
Náklad 300 výtisků