Milí řešitelé a řešitelky!

Přinášíme vám vaše opravená a naše vzorová řešení čtvrté série. Letošní ročník se tak chýlí ke konci, zbývá už jenom vyřešit a opravit poslední pátou sérii. Těšíme se na vaše výtvory!

Termín odeslání šesté série jest pochopitelně neurčen, alébrž žádné šesté série není. Řešení byste bývali mohli odevzdávat jak elektronicky na *http://ksp.mff.cuni.cz/submit/,* tak na:

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

Aktuální informace o KSP můžete nalézt na stránkách *http://ksp.mff.cuni.cz/,* diskutovat můžete na fóru *http://ksp.mff.cuni.cz/forum/* a záludné dotazy organizátorům lze zasílat e-mailem na adresu *ksp@mff.cuni.cz.*

Vzorová řešení čtvrté série dvacátého ročníku KSP

20-4-1 Druidí nápisy

Tato úloha vám pravděpodobně přivodila značné bolesti hlavy, a proto došla všeho všudy dvě řešení. My si tedy budeme chvíli lámat hlavu společně, přičemž se pokusíme vyhnout zmíněným intelektuálním bolestem.

První úskok, kterého se dopustíme, bude odkaz na řešení úlohy 20-2-3. Jistě jste si povšimli, že zadání jsou si velmi podobná a věžte, že tak je tomu i u řešení. Proto se před dalším čtením ujistěte, že znáte zadání i řešení 20-2-3 a porozuměli jste dané problematice.

V tento moment můžeme předpokládat existenci několika částí našeho algoritmu a stavět na nich. Pro jistotu si však ještě shrňme nejdůležitější body, z nichž vyjdeme: k užitku přijde, že umíme vstupní slovník reprezentovat *trií*, kde navíc umíme pro každou posloupnost značek (značkami budeme rozumět např. tečky a čárky) okamžitě vypsat, jaká *všechna* slova může posloupnost kódovat; dalším stavebním kamenem bude postup, jakým jsme hledali nejkratší slovní reprezentaci řetězce – ten budeme upravovat pro naše potřeby a proto se na něho podíváme důkladněji.

Jak tedy vypadalo pole, které jsme pomocí *dynamického programování* naplnili, a co nám vlastně sdělovalo za informaci? Pole nám na indexu i pro každou podposloupnost vstupu od i do n (tedy jinak řečeno pro každý sufix vstupu začínající indexem i) poví, jaké slovo si máme vybrat jako první, aby výsledný rozklad vstupu na slova byl co nejkratší. To nás v důsledku odkazuje na další pozici v tomto poli atd. . . což už vedlo k řešení.

Nyní si představme, že naše pole B bude „chytřejší“ a poví nám více informací. Řekněme, že nám ke každému sufixu vstupu poví pro každé j, jestli je takovýto sufix rozložitelný na právě j slov. Takové pole je dvojrozměrné - $B[i, j] = 1$ pokud je sufix začínající na indexu i rozložitelný na j slov, v opačném případě je $B[i, j] = 0$. Konstrukce není nikterak složitá. Pro sufix nulové délky - tedy podposloupnost, která začíná za koncem řetězce a zároveň tam i končí - víme, že je rozložitelný pouze na 0 slov ($B[n + 1, 0] = 1$ ale všechna ostatní $B[n + 1, 1] = B[n + 1, 2] = \ldots = B[n + 1, n] = 0$. Podobně jako v 20-2-3 budeme postupovat od nejkratšího sufixu až po nejdelší (celá posloupnost). Tedy pole B vyplňujeme od indexu n do indexu 1. Pro každý sufix snadno vyplníme hodnoty pro různá j, když známe všechna taková j kratších sufixů. Podrobněji prozkoumejme část, kdy jsme v druhé sérii vyplňovali položky pole na indexu i. Díky trii jsme nalézali jednotlivá slova, která mohou na daném indexu začínat a podle nich přepisovali údaj o nejkratším možném rozkladu. Zůstaňme u toho, že nacházíme jednotlivá slova od daného počátku i. Pak tedy víme, kde má začínat další slovo patřičného rozkladu (totiž hned za koncem prvního slova) – označme si takové místo indexem q ($q = i + d$, kde d je délka prvního slova a zřejmě tedy $q > i$]). Protože postupujeme od nejkratších sufixů, tj. pole vyplňujeme od konce, tak hodnoty na pozici q již známe). Nyní nám stačí podívat se, na kolik slov umíme rozložit sufix vstupu začínající indexem q. Jestliže lze sufix začínající v q rozložit na j slov, potom sufix začínající i lze rozložit na $j + 1$ slov. Tj. pokud $B[q, j] = 1$ pak nastavíme $B[i, j + 1] = 1$. Zřejmě jen zkoušíme „dolepovat“ různá slova před začátek již existujících rozkladů – ačkoliv rozklady jako takové si nepamatujeme, pouze vedeme v patrnosti jejich existenci.

Tedy, právě jsme sestavili pole, podle kterého umíme říci, zda daný sufix lze rozložit na j slov. K čemu je nám to dobré? Umíme totiž nalézt všechny rozklady o daném počtu slov a vypsat je! Podívejme se na index 1, tedy na rozklad pro celý vstup. Vezměme si všechna j od nejmenšího k největšímu a uvažujme jen ta, pro která lze vstup rozložit na j slov (tedy $B[1, j] = 1$). Pro každé takové j můžeme zkusit s pomocí trie dohledat různá slova, kterými může rozklad začínat. Každé takové slovo někde končí a rozklad pokračuje na pozici q bezprostředně za ním následující. My se na tato q podíváme a pokud pro sufix začínající od q existuje rozklad na $j - 1$ slov, pak si slovo, jenž nás posunulo na index q, můžeme do rozkladu vybrat. Zkráceně: Vyzkoušíme všechna slova, která mohou být na začátku, a podíváme se, jestli si je můžeme opravdu vybrat, tedy jestli za ně můžeme „dolepit“ rozklad pokračující $j - 1$ slovy (jednička se zřejmě odečítá za již zvolené slovo). Jak dál? Jsme na indexu q a chceme rozložit sufix od q na $j - 1$ slov. To už je ale ten samý problém, jako když jsme zkoušeli rozložit celý vstup na j slov. Pouze se liší místo, kde má rozklad začít, a počet slov, které má rozklad mít. Opět zkusíme všechna možná slova pro pokračování rozkladu a budeme se dívat, kam dál.

Jistě vás tedy napadá myšlenka použít rekurzi a de facto nasadit backtrack. My to opravdu uděláme, ale neděste se, ukážeme totiž, že to bude „slušný“ backtrack, který díky předpočteným informacím z části s dynamickým programováním (konstrukce dvojrozměrného pole) poběží rozumně rychle.

Backtrackující funkce potřebuje ke své činnosti znát počet již vypsaných rozkladů a stav právě zkoumaného rozkladu. Stav je charakterizován slovy, které jsme si již vybrali pro začátek rozkladu, indexem i, na kterém končí poslední z těchto slov, a konečně počtem slov j, které ještě do úplného rozložení zbývá. Funkce pak vyzkouší všechna slova, která by mohla na zadaném indexu začínat, a pokud pro nějaké z nich zjistí, že lze rozložit zbytek vstupu na $j - 1$ slov (zkontroluje $B[q, j-1] = 1$, kde q značí index těsně za kontrolovaným slovem), zavolá sama sebe. Tomuto synovskému

volání přidá do rozkladu nalezené slovo a nový upravený stav – tedy index posunutý za konec takového slova a j snížené o 1. Pokud se po návratu ze synovského volání bude počet vypsaných rozkladů roven K, funkce svoji činnost ukončí. Tuto funkci spustíme postupně od nejkratšího na všechny rozklady začínající na indexu 1 – vyzkoušíme tak rozklady celého vstupu.

Podívejme se, jak dlouho trvá jedno volání našeho backtracku. Jsme na pevně zvoleném indexu a máme zadaný počet slov, na který máme vstup ještě dorozložit. Chceme přitom vypsat všechny možné rozklady dané délky. Projdeme až L indexů, kde může nějaké vybrané slovo končit (L označuje délku nejdelšího slova ve slovníku). Avšak pro každé slovo už vykonáme dotaz v konstantním čase do našeho zkonstruovaného pole. Pak už opět voláme sami sebe.

U backtracků obecně nám časovou složitost zhoršuje veliké množství větví výpočtu. Průběh výpočtu nějaké backtrackující funkce si totiž můžeme představovat jako strom, kde každý vrchol odpovídá jednotlivým voláním funkce, a hrany znázorňují vztah volající-volaný. Pokud se strom v každém vrcholu větví třeba jen dvakrát a hloubka stromu je řekněme 100, pak celková velikost stromu bude 2^100. V našem případě vyzkoušíme až L indexů. Ještě poznamenejme, že určitě $L \leq n$, protože kdyby tomu tak nebylo, museli bychom mít nějaké slovo delší než celý vstup. Takové slovo ale není k ničemu a můžeme ho zahodit. Mohlo by se zdát, že náš algoritmus poběží v čase $\mathcal{O}(n^n)$, což se tváří dosti beznadějně.

Nyní si připomeňme, že nám stačí K nejkratších rozkladů, tedy můžeme backtrack po vypsání těchto rozkladů ukončit. Dalším důležitým postřehem je, že nikdy nevstoupíme do „slepých“ větví výpočtu – tedy nezkoušíme něco, co nevede k výsledku. To nám zajistilo právě dynamickým programováním zkonstruované pole B, kterého se tak v důsledku ptáme, zda máme vůbec nějaký rozklad zkoušet. Spojením těchto pozorování je fakt, že strom volání naší backtrackující funkce má právě K listů. Celková hloubka stromu je nejvýše n (připomeňme, že za n rozumíme délku vstupní posloupnosti)– úroveň v hloubce h odpovídá částečnému rozkladu s h slovy, ale protože každé slovo má alespoň jedno písmeno, nemůžeme nikdy ve stromu být hlouběji než v hloubce n. Tedy celkový počet volání backtrackující funkce je nejvýše $K \cdot n$. Pro každý list pak musíme ještě celý rozklad vypsat, což ale stíháme rovněž v $K \cdot n$. To už dává pro funkci rozumně vypadající časový odhad $\mathcal{O}(K \cdot n^2)$, kde pro každé volání funkce zohledňujeme režiji $\mathcal{O}(n)$ na vyzkoušení slov.

Celková paměťová složitost programu je nejvíce zatížena použitým dvojrozměrným polem B, protože vše ostatní je lineárně velké vzhledem k délce vstupu, výsledkem je tedy $\mathcal{O}(n^2)$. Časová složitost je ovlivněna konstrukcí trie v $\mathcal{O}(P)$, kde P označme celkovou velikost slovníku. Dále počítáme ono dvojrozměrné pole B – ke každému z n indexů vyzkoušíme až n dalších. Pro každý z těchto n indexů vykonáme až n přepisování údajů (vyplňování hodnot rozkladů). Tedy dynamický výpočet je v $\mathcal{O}(n^3)$. Backtrack potom v $\mathcal{O}(K \cdot n^2)$. Celkem tedy $\mathcal{O}((K \cdot n^2 + n^3)$.

Zrychlujeme... Pozorní čtenáři si jistě všimli, že s časem poměrně plýtváme a jistě lze úlohu vyřešit lépe. Konkrétně budeme zrychlovat backtrack.

Podívejme se, zda nevykonáváme příliš mnoho kroků v každém z volání naší funkce. Zkoumáme totiž často až zbytečně mnoho indexů, jestli z nich nemůže náš rozklad pokračovat. Tedy ke každému i zkoušíme až n indexů. Kdyby se nám podařilo vyhnout se testování zbytečných indexů, znatelně si pomůžeme. Označmě $q_1, q_2 \ldots, q_m$ všechny indexy, pro něž se náš výpočet větví a kterými pokračuje rozklad. Za zbytečné indexy pak budeme považovat ty, které leží za q_m. Na první pohled se zdá, že si pranic nepomůžeme, ale opak je pravdou. Pohleďme na problém šalamounsky a „naúčtujme“ procházení indexů až do q_m někomu jinému, v našem případě oním šťastlivcem bude výpis rozkladu.

Uveďme na příkladu: Ocitáme se právě uprostřed výpočtu. Řekněme, že náš rozklad má již h slov a nacházíme se na indexu 100. Ve slovníku je celkem W slov, ale pouze pro w znich existují rozklady od aktuálního indexu. Délka nejdelšího z těchto w slov je řekněme 42. Délka nejdelšího slova ve slovníku je třeba 123. Pak za zbytečné indexy pokládáme všechny od 143 dál (až do $223 = 100 + 123$). Tedy $B[143, 0 \ldots m] \ldots B[223, 0 \ldots m]$ už nezkoušíme. Oněch prvních 42 znaků musíme vypsat tak jako tak, což znamená, že tyto kroky není nutné započítávat.

Pro každé z použitých $q_1, \ldots q_m$ budeme stejně muset vypsat celé slovo a proto můžeme s čistým svědomím všechny kroky potřebné k jejich určení započítávat do výpisu těchto slov (víceméně násobíme konstantou 2). Problematické jsou pouze zbytečné indexy, které nemáme komu naúčtovat, ale právě proto je vynecháme. Ve skutečnosti jsme tak schovali veškerou časovou režiji naší funkce do výpisu, o kterém ale víme, že spotřebuje $\mathcal{O}(K \cdot n)$.

Co k tomuto úskoku budeme potřebovat? Postačí nám, když budeme znát poslední index q_m a to pro každou dvojici $(index, počet slov rozkladu)$ – to znamená navíc ke každému $B[i, j]$. Pro rozklady o různém počtu slov se totiž $q_1, \ldots q_m$ obecně liší. Během výpočtu našeho dvojrozměrného pole B si budeme počítat i nějaké další pomocné pole $Q[i, j] = q_m$. V backtracku postačí se dívat do tohoto pole a nezkoušet indexy za $Q[i, j]$.

Snížili jsme odhad časové složitosti funkce na $\mathcal{O}(K \cdot n)$ a celkovým výsledkem je $\mathcal{O}(K \cdot n + n^3)$.

Ještě poznámka na závěr: Náš algoritmus jsme se záměrně optimalizovali na paměťovou složitost nezávislou na K a to za cenu dynamického výpočtu v $\mathcal{O}(n^3)$. Důvodem je možná velikost K. Představme si, že vstup bude pouze z teček a ve slovníku budou pouze dvě slova. Jejich zápisem bude jedna a dvě tečky. Počet možných rozkladů vstupu délky n pak bude F_n, kde F_n je n-té Fibonacciho číslo. Ty snadno odhadneme zdola na $2^{n/2}$, protože z $F_{n+2} = F_{n+1} + F_n$ plyne, že F_{n+2} je alespoň dvakrát větší než F_n. A exponenciální paměťovou složitost si opravdu dovolit nemůžeme.

Josef Pihera & Martin Mareš

20-4-2 Stonehedge

Napřed uděláme několik pozorování.

Z každého rohu vedou právě dvě úsečky. Méně nedává smysl a více dává křížení.

Jedna z těchto úseček bude vodorovná a jedna svislá (jinak by to nebyl roh, ale průchod bodem). Tedy, každý vrchol má svého vodorovného a svislého souseda.

Nyní si vezměme například vodorovné sousedy (pro svislé to bude obdobné). Vodorovný soused má stejnou y-ovou souřadnici.

Rozdělme si tedy všechny body do skupin podle y. Podívejme se na jednu takovou skupinku a setřiďme si ji, řekněme,

odleva. Ten úplně vlevo musí mít svého souseda (a to v této skupince), ale jediný, který připadá v úvahu je ten nejbližší napravo od něj. Takže je spojíme. Tím jej samozřejmě použijeme (může mít jen jednoho souseda) a stejná situace tedy nastává se třetím a čtvrtým a tak dále.

Všimněme si, že v každé skupince musí být sudý počet vrcholů. Kdyby nebylo, tak to zřejmě není kámen popsaný v zadání (poslední nemá s čím sousedit).

Takto můžeme zpracovat všechny skupinky. A obdobným způsobem můžeme spárovat i svislé dvojice.

Jak to ale udělat rychle? Pokud si setřídíme body lexikograficky podle (y, x), pak budou vždy všechny body se stejnou y-ovou souřadnicí za sebou, seřazené dle x-ové. A protože mají skupinky sudé počty, můžeme takto setříděnou posloupnost prostě začít spojovat po dvojicích od začátku (a nestarat se, kde začíná jedna skupinka a druhá končí).

Nyní již máme ke každému bodu jeho svislého a vodorovného souseda (svislé sousedy uděláme stejně, jen setřídíme dle (x, y)). Musí tvořit uzavřený cyklus (neboť v kameni nejsou díry a všechny body musí být na kameni a obvod je jistě souvislý). Takže jej stačí jen vypsat a jediný problém je, kterým směrem začít.

Pokud si vybereme některý vrchol na „horní vrstvě“ a ten má svého pravého souseda, pak je to zajisté po směru hodinových ručiček. A nejlevější vrchol svého pravého souseda bude mít určitě. Mimochodem, tento vrchol skončil jako první při setřídění dle (y, x).

Jak rychle to dokáže běžet? Načtení zvládneme lineárně, výpis také (to jen procházíme kolem dokola, než narazíme opět na ten první). Rozdělení na dvojice jde také lineárně – každý bod zapojíme jednou vodorovně a jednou svisle. Jediný problém je tedy s tříděním, které (v obecném případě) nezvládneme rychleji, než v $\mathcal{O}(N \cdot \log N)$.

Paměťová složitost je lineární, stačí si pamatovat jednotlivé body a jejich sousedy.

Michal „vorner“ Vaner

20-4-3 Mince

Do zadání se nám vloudila jedna drobná nejednoznačnost. Není zřejmé, zda v případě, kdy jsou na počátku 4 mince lícem vzhůru trpaslík hned ukončí hru, či ne. Pokud tomu tak nebude, přidáme na začátek „tah“, kdy neotočíme žádnou minci a vše převedeme na první případ.

Nejprve si uvědomme, že všechny možnosti, jak jsou mince otočeny (nazveme toto stavem mincí), je možno rozdělit na šest skupin (matematicky se mluví o kongruenci), které se při otočení nemění (tj. pokud vezmeme stav z nějaké skupiny a otočíme soudek, bude stav patřit do té samé skupiny). Popišme je vzájemnou polohou a počtem mincí, které jsou lícem vzhůru. To mohou být všechny (a tedy končíme), tři mince, dvě mince úhlopříčně, dvě mince vedle sebe, jedna mince, nebo žádná. Pokud tedy zjistíme, kterou skupinu převádí daný tah (tj. otočení mincí) na kterou, můžeme najít jejich posloupnosti, které každou z pozic převedou na první z nich a tím jsme hotovi.

Jak na to? Pro zjednodušení uvažujme, že každý lichý tah otočíme na soudku všechny mince a navrhujeme jen sudé. Ten nám převádí skupinu se čtyřmi a žádnou mincí lícem nahoru mezi sebou. Stejně tak mezi sebou přechází trojice, resp. samotná mince. Zbylých dvou skuipn, odpovídajících dvěma mincím lícem vzhůru, se tento tah nedotkne. Slučme tedy příslušně skupiny mezi sebou a označme si je jako (4) v případě, že jsou všechny mince stejnou stranou vzhůru, (2U) když jsou dvě mince úhlopříčně lícem vzhůru a dvě ne, (2V) případ, kdy jsou dvě mince vedle sebe lícem vzhůru a dvě ne a (1) stavy, je jedna mince nějak otočena a zbylé tři opačně.

Další užitečné pozorování je, že při otočení sudého počtu mincí se nezmění parita počtu mincí lícem vzhůru (tj. pokud byl lícem vzhůru lichý počet mincí, zůstane lichý, a pokud sudý zůstane sudý). Tedy otočením sudého počtu mincí převedeme stav ze skupiny (1) na stav ze skupiny (1) a stavy skupin ((4),(2U),(2V)) na stavy z té samé trojice skupin. Pokud otočíme jednu (nebo libovolný lichý počet mincí), převede to (1) na stav z některé ze skupin ((4),(2U),(2V)) (do které skupiny přesně se dostaneme záleží na otočení soudku a kterou přesně minci bereme) a stavy ze skupin ((4),(2U),(2V)) na stavy skupiny (1). Pokusme se tedy najít posloupnost otáčení sudého počtu mincí, kterou jsme schopni dostat stavy ze „sudých“ skupin ((4),(2U),(2V)) dostat na (4). Pokud nám nevyřeší problém, tak musel stav patřit (a stále patří) do skupiny (1). Otočením jedné mince ho převedeme na nějaký stav ze „sudých“ skupin a, jelikož už víme, že stav ze skupiny (1) nemůžeme mít, stejnou posloupností tahů trpaslíka porazíme.

Zbývá tedy podívat se na to, jak vyřešit skupiny (4), (2U) a (2V). Uvažujme pro tento odstavec, že víme, že lícem nahoru byl sudý počet mincí. (4) ani řešit nemusíme - tu nám zastaví trpaslík. (2U) je jednoduché - otočením dvojice mincí úhlopříčně na soudku dostaneme (4) (a tedy trpaslík nás zastaví) a z (2V) se nám stane (2V). Po „prvním“ tahu, pokud nás trpaslík nezastavil, víme, že stav soudku je ze skupiny (2V). Jako „druhý“ tah otočíme dvojici mincí vedle sebe. Podle toho, jak byl zrovna natočen soudek, dostaneme stav ze skupiny (4) nebo (2U). Pokud nás stále trpaslík nezastavil, patří do skupiny (2U) a tedy po otočení dvojice mincí úhlopříčně budou všechny rubem (nebo lícem) vzhůru.

Tím jsme tedy nastínili, jak získat vyhrávající strategii. Konkrétně bude vypadat:

```
otoč všechny 4 mince
otoč 2 mince úhlopříčně
otoč všechny 4 mince
otoč 2 mince vedle sebe
otoč všechny 4 mince
otoč 2 mince úhlopříčně
otoč všechny 4 mince
otoč jednu minci
otoč všechny 4 mince
otoč 2 mince úhlopříčně
otoč všechny 4 mince
otoč 2 mince vedle sebe
otoč všechny 4 mince
otoč 2 mince úhlopříčně
otoč všechny 4 mince
```

Malá poznámka na závěr. Rychlejší strategie, než zde uvedené ani neexistuje. Pro začátek uvažujme, že máme jen zavázané oči (a nevíme, jaké je počáteční otočení mincí), ale trpaslík netočí se soudkem. Uvažujme všechny přípustné stavu soudku (tj. takové, kdy nám trpaslík ještě nezastavil hru). Každým tahem můžeme maximálně jednu z těchto pozic převést na tu, kdy jsou všechny lícem vzhůru a tím jí vyloučit (buď nás trpaslík zastaví, nebo to tenhle stav mincí

nebyl). Zbylé pozice se nám vzájemně jednoznačně převedou na (možná) jiné stavy. Vzájemně jednoznačně znamená, že po provedení tahu budeme schopni určit jaká byla původní pozice (např. provedením toho samého tahu podruhé se vrátíme zpět) a tedy i že počet různých přípustných pozic se nám nezmenší, resp. zmenší o jednu, kterou jsme vyloučili. Jelikož na začátku máme jeden z 15-ti stavů a nevíme který, potřebujeme alespoň 15 tahů abychom libovolný z nich otočili lícem vzhůru. Pokud trpaslík „začne“ točit soudkem, určitě strategie řešící tuhle úlohu nebude kratší než v případě, kdy neotáčí. A protože nalezená strategie 15 tahů má, musí být nejkratší možná.

Pavel Čížek

20-4-4 Skupinky pro chytré

Operace nad skupinkami přesně odpovídají operacím nad haldou z kuchařky 4. série, a naše rešení bude opravdu vycházet z haldy. Protože nechceme hledat jedince s minimálním IQ (těch je dost :-), ale s maximálním, bude zapotřebí otočit porovnávání.

Větší rozdíl spočívá v tom, že operace potřebujeme provádět "nedestruktivně" – nesmíme měnit již existující skupinky. Na principu fungování haldy se nic nemění, ale data nebudeme moci ukládat do pole jako v kuchařce. Strom uložíme jako sadu prvků (struct Node) pospojovaných pointery na levého a pravého syna. Operace nad haldou tak bude jednoduché dělat nedestruktivně: místo modifikace prvku naalokujeme nový prvek, zkopírujeme hodnoty ze starého prvku, a upravíme co je potřeba upravit. Při modifikaci syna budeme muset vždy vyrobit i nového otce a dál až ke kořeni.

Pro haldové operace potřebujeme efektivně umět pracovat s nejpravějším prvkem ve spodní hladině, a potřebujeme umět určit otce daného prvku. V poli je situace jednoduchá, požadovaný prvek je v poli tolikátý, kolik je prvků v haldě, a otec je na pozici i/2 (kde i je pozice syna).

Jednoduché řešení by bylo přidat do každého prvku ukazatel na jeho otce, a držet si ukazatel na nejpravější prvek ve spodní hladině; ale toto řešení použít nemůžeme, protože ukazatel na otce by nám neumožnil pracovat "nedestruktivně".

Naštěstí je možné i-tý prvek najít pomocí bitového zápisu i, stačí postupovat od kořene a dle hodnoty bitu jít do levého nebo pravého podstromu. Pokud si do pomocného pole schováme prvky, které jsme prošli, odpadne též problém s hledáním předchůdců.

Časová složitost operací `insert` a `delete_best` je $\mathcal{O}(log(n))$, časová složitost `find_best` je $\mathcal{O}(1)$. Operace `find_best` nepotřebuje žádnou dodatečnou paměť, `insert` i `delete_best` naalokují $\mathcal{O}(log(n))$.

(S diky Petru Ondrúškovi.)

Pavel Machek

20-4-5 Roboti na útěku

Řešitele této úložky lze rozdělit do dvou skupin. Skupina první (která měla mohutnost pouze 3) přišla, viděla a s přehledem vyřešila. Skupina druhá (značně početnější) přišla, viděla, nevěřila svým očím, a tak raději vůbec neřešila. Pojďme se nyní podívat, jak si s roboty poradit ...

K řešení našeho problému použijeme procházení stavového prostoru do šířky. U všech procházení je nejtěžší poznat, jak vypadá zmíněný stavový prostor. Obecně je stavový prostor orientovaný graf, kde vrcholy představují jednotlivé stavy a hrany přechody mezi nimi (jednotlivé kroky). V našem případě je stav definován polohou obou robotů a všech stráží. Z každého stavu pak vedou nejvýše čtyři hrany – jedna pro každý potencionální příkaz, který mohou roboti obdržet (stráže se pohybují automaticky, takže není potřeba jejich pohyb dále řešit). Některé hrany (případně i stavy) mohou být z prostoru vyloučeny, protože v nich dojde k zajmutí některého z robotů.

Bohužel nemůžeme stavy reprezentovat přímočaře, jak bylo popsáno. Pejsek je zakopaný v počtu stráží. Kdyby jich v obou bludištích byl maximální počet (tedy 10), potřebovali bychom vhodně ukládat informace o $(x_1y_1x_2y_2)^11$ stavech. Naštěstí víme, že stráže neustále chodí po stejných trasách a délka jejich tras může být pouze 2, 3 nebo 4. Podle toho se pozice stráže opakuje vždy po 2, 4 nebo 6 krocích. Nejmenší společný násobek těchto čísel je 12, takže každých 12 kroků se pozice všech stráží opakuje. Pokud víme, ve které z 12-ti možných pozic se právě stráže nachází, jsme schopni spočítat její polohu jen ze vstupních dat.

Jak již bylo naznačeno, prostor budeme prohledávat do šířky. Pro každý stav si potřebujeme pamatovat informaci, zda jsme v něm už byli, a také z jakého stavu jsme se do něho dostali, abychom pak mohli zrekonstruovat výslednou cestu. Maximální počet stavů bude $12 \times (20 \times 20)^2$, protože existuje 12 možných pozic stráží a maximální rozměry bludiště jsou 20×20, celkem tedy 2560000 stavů. Stav dokážeme bez problému zakódovat do 32-bitového integeru, takže celkem spotřebujeme necelých 10 MB paměti na stavy a nejvýš tři čtvrtiny této hodnoty na frontu. S těmito čísly už se do CodExu vejdeme.

Samotný algoritmus pak přesně odpovídá tomu, co již známe z kuchařky. Na počátku vložíme do fronty výchozí stav a označíme jej za použitý. V každém kroku vybereme jeden stav z fronty, spočítáme všechny stavy, do kterých se z něj dá dostat, ty si označíme a vložíme do fronty. Algoritmus končí v okamžiku, kdy narazíme na stav, ve kterém jsou oba roboti venku z bludiště.

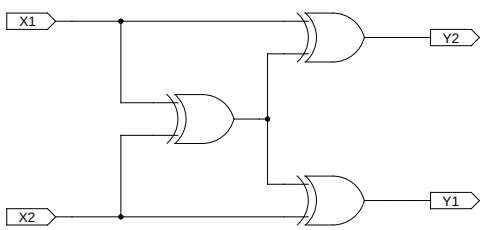
Výslednou cestu pak zrekonstruujeme tak, že postupujeme od koncového stavu zpět k výchozímu podle informací, které jsme si ke stavům uložili. Jediný zádrhel spočívá v tom, že cestu musíme vypsat v opačném pořadí, takže si ji musíme nejprve uložit a pak teprve vypsat.

Voila. Nyní už je dochutíme podle libosti a servírujeme kompilátoru.

Martin „Bobřík“ Kruliš

20-4-6 Hrady, hrádky, hradla

a) První podúloha, tedy nalezení „křižítka“, bude snadná. Budou nám totiž stačit tři hradla XOR:



Proč tento obvod dělá to, co potřebujeme? Pokud jsou oba vstupy stejné, odpoví prostřední hradlo nulou, takže krajní hradla jen zkopírují vstup na výstup, což je správně. Pokud

jsou naopak vstupy různé, prostřední hradlo odpoví jedničkou, takže krajní hradla vstup negují, a to je opět správně.

Dokázat bychom to mohli i algebraicky ($\oplus$ je XOR):

$$Y_2 = X_1 \oplus (X_1 \oplus X_2) = (X_1 \oplus X_1) \oplus X_2 = 0 \oplus X_2 = X_2.$$

A jak se dá na něco takového přijít? Zkusme uvažovat takto: Máme najít obvod se vstupy X_1 a X_2 a výstupy Y_1 a Y_2, který bude vždy počítat $Y_1 = X_1$ a $Y_2 = X_2$. Navíc tento obvod má být rovinný a pokud ho vepíšeme do kružnice, mají vstupy a výstupy po obvodu této kružnice dávat pořadí X_1, X_2, Y_1, Y_2. Jistě můžeme předpokládat, že hradlo, ze kterého vystupuje Y_2, leží přímo na kružnici, takže do něj můžeme podél kružnice přivést vstup X_1. Analogicky hradlo pro Y_1 může znát Y_2. Jak tedy spočítáme z X_1 hodnotu $Y_2 = X_2$? K tomu je potřeba informace o tom, zda se X_1 a X_2 liší či nikoliv. Tu lze snadno počítat uvnitř kružnice. A naše konstrukce je na světě.

Mimochodem, vystačili bychom si i s hradly NAND: Vzpomeňte si na konstrukci hradla XOR ze čtyř NANDů – byla rovinná. Můžeme ji tedy dosadit do našeho schématu a získat křížící obvod z NANDů.

b) Druhá podúloha je zdánlivě dočista triviální. Pokud nějaký obvod není rovinný, je to proto, že se v něm vodiče kříží. Tak křížení nahradíme naším křižítkem a získáme obvod, v němž je o jedno křížení méně. To nám stačí provést konečně-krát a obvod bude rovinný. Jenže ...

1. V jednom místě grafu by se mohlo křížit i více hran. Pokud se to stane, určitě existuje nějaké malé okolí tohoto bodu, kde nejsou žádná hradla ani další křížení. Tak všechny hrany, které se křížily, „rozšoupneme“ do tohoto okolí a z násobného křížení tím vyrobíme několik obyčejných.

2. Kdybychom křižítko umístili nešikovně, mohli bychom vyrobit nová křížení a celý proces zroviňování by se nikdy nezastavil. Pomůžeme si podobně jako od násobných křížení: najdeme dostatečně malou kružnici okolo místa křížení, kde se vyskytují jenom ty vodiče, které se křížení účastní (taková určitě existuje), a obvod vlepíme do ní. Víme přeci, že se do kružnice dá vepsat a jistě ho můžeme „ohnout“ tak, aby měl vývody na správných místech.

3. Mohli bychom v obvodu vytvořit cyklus – třeba tehdy, když se kříží vodič vedoucí na vstup nějakého hradla s vodičem vedoucím z výstupu téhož hradla. Cykly jsme sice v definici hradel v první sérii výslovně nezakázali (a jeden z ukázkových obrázků dokonce cyklus obsahuje), ale není nijak jasné, jak se takové obvody chovají. Teprve v páté sérii se něco takového pokoušíme zavést. Proto cykly v obvodech, kde dříve nebyly, při zroviňování nechceme vytvářet.

To se zařídí snadno: nakreslíme si schéma tak, aby bylo *topologicky setříděné* podle x-ové souřadnice. Na přímce $x = 0$ budou ležet ta hradla, která závisí pouze na vstupech obvodu; poskládáme je tam libovolně. Na přímku $x = 1$ umístíme ta, která závisí na vstupech obvodu a na výstupech již umístěných hradel, a tak dále. Pokud v obvodu není cyklus, postupně takto umístíme všechna hradla. Všimněte si, že nyní jsou v každém křížení oba vodiče naprosto nezávislé – hodnota na jednom nijak nezávisí na tom, jaká hodnota putuje druhým. Přidáváním křižítek tedy už nemohou cykly vznikat.

Martin Mareš

Úloha 20-4-1 – Druidí nápisy – program

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

#define MAX 256
#define INFTY 10000             // Nekonečno

struct vrchol {                 // Vrchol trie
    struct vrchol *syn[2];      // syn[0] pro tečku, syn[1] pro čárku
    struct slovo *slova;        // Seznam slov, která tu končí
    int hloubka;                // Délka morseovkového zápisu (hloubka v trii)
};
struct vrchol koren;

struct slovo {                  // Takhle si ukládáme seznamy slov
    struct slovo *dalsi;        // Vlastně by si stačilo pamatovat jen jedno slovo,
    char s[1];                  // ale třeba se to bude hodit v následující úloze :)
};

// Tabulka kódů:   A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
char morse[26] = { 6,17,21, 9, 2,20,11,16, 4,30,13,18, 7, 5,15,22,27,10, 8, 3,12,24,14,25,29,19 };

void preloz(char *co, char *kam)                                // Přeloží slovo do morseovky
{
    while (*co) {
        int k = morse[*co++ - 'a'];                             // Ďábelský kód písmenka
        while (k > 1) {                                         // Postupně rozkládáme na značky
            *kam++ = ".-"[k%2];
            k /= 2;
        }
    }
    *kam = 0;
}

void nacti_slovnik(void)
{
    char slovo[MAX], mslovo[MAX];
    while (fgets(slovo, sizeof(slovo), stdin) && slovo[0] != '\n') {
        int len = strlen(slovo);
```

```
        slovo[len-1] = 0;                                       // Smažeme konec řádku
        preloz(slovo, mslovo);                                  // Přeložíme do morseovky
        struct vrchol *v = &koren;                              // Přidáváme do trie
        for (char *c=mslovo; *c; c++) {
            int z = (*c == '-');                                // Aktuální značka
            if (!v->syn[z]) {                                   // Kam dál? Není-li kam, založíme nový vrchol
                v->syn[z] = (vrchol*) malloc(sizeof(struct vrchol));
                memset(v->syn[z], 0, sizeof(struct vrchol));
                v->syn[z]->hloubka = v->hloubka + 1;
            }
            v = v->syn[z];                                      // Vydáme se tím směrem
        }
        struct slovo *s = (struct slovo*) malloc(sizeof(struct slovo) + len);
                                                                // Stojíme ve vrcholu, který odpovídá konci slova,
        s->dalsi = v->slova;                                    // tak tam slovo přidáme
        v->slova = s;
        strcpy(s->s, slovo);
    }
}

int B[MAX+1][MAX+1];    /* B[i][j]==1 pokud lze pokrýt [i...N-1] přesně j slovy */


//Zde pomocí dynamického programování naplníme B
void SpoctiB(char *vstup, int N)
{
        B[N][0] = 1;
        for (int i=N-1; i>=0; i--) //Postupujeme od konce vstupu
        {
                struct vrchol *v = &koren; //a procházíme v trii postupně od kořene
                int j = i;
                while (v && j < N)
                {
                        v = v->syn[vstup[j++] == '-'];
                        if (v && v->slova)  //ano, nějaké slovo zde končí
                                for (int k=0; k<N; k++)
                                        if (B[j][k]) //Od indexu j umíme rozložit k slovy
                                        {
                                                B[i][k+1] = 1; //tedy zde (od i) umíme rozložit k+1 slovy
                                        }
                }
        }
}


int K; //počet rozkladů, které máme vypsat
int K_hotovo; //kolik rozkladů jsme již vypsali

/* vypiš rozklad od i...N-1 s j slovy, kde v *rozklad jsou již uložena slova
   z rozkladu od 1..i-1 a *p udává, kde v řetězci *rozklad máme pokračovat */
void PisReseni(char *rozklad, char *p, char *vstup, int i, int N, int j)
{
        if (i >= N) //Rozklad této větve výpočtu je již dokončen
        {
                *p = 0;
                puts(rozklad);
                K_hotovo++;
                return;
        }

        struct vrchol *v = &koren;
        *p++ = ' ';
        while (v && i < N)
        {
                v = v->syn[vstup[i++] == '-'];
                if (v && v->slova && B[i][j-1]) //končí-li zde  nějaké slovo
                        for (struct slovo *slovo = v->slova; slovo; slovo=slovo->dalsi)
                        {                   //pak projdeme všechna taková slova
                                int delka = strlen(slovo->s);
                                memcpy(p, slovo->s, delka); //dopíšeme si je k částečnému rozkladu
                                PisReseni(rozklad, p + delka, vstup, i, N, j-1); //a rekurzivně pokračujeme
                                if(K_hotovo == K) //a jestli jich už máme dost, skončíme
                                        return;
                        }
        }
}


int main(void)
{
    char z[MAX];                                                // Vstupní řetězec
```

```
    int n;                                                  // Jeho délka
    struct vrchol *s[MAX+1];

    nacti_slovnik();                                        // Přečteme slovník a vstup
    fgets(z, MAX, stdin);                                   // Pozor, indexujeme od 1
    n = strlen(z)-1;

        scanf("%d", &K);    //Načteme, kolik rozkladů se po nás chce
        K_hotovo = 0;

        SpoctiB(z, n);
        char rozklad[MAX];
        for(int j=0; j<=n && K_hotovo<K; j++)  //ptáme se na všechny rozklady celého
        {                                      //vstupu, od nejkratších počínaje
                if(B[0][j])
                {
                        PisReseni(rozklad, rozklad, z, 0, n, j);  //a každý nalezený vypíšeme
                }                                       // (resp. všechny rozklady délky j)
        }

    if (K_hotovo < K)                                       // Pokud žádané řešení není, dáme o tom vědět
        {
                if(K_hotovo)
                        printf("Existuje jen %d rozkladů.\n", K_hotovo);
                else
                        puts("Řešení neexistuje");
        }

                return 0;
}
```

Úloha 20-4-2 – Stonehedge – program

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

#define VODOROVNE 0
#define SVISLE 1
#define X 0
#define Y 1

typedef struct bod_t {
        int souradnice[ 2 ];
        struct bod_t *sousedi[ 2 ];
} bod_t;

int co[ 2 ] = { X, Y };//Kterým směrem se porovnává

int porovnej( const void *_1, const void *_2 ) {//Lexikografické porovnání podle pořadí definovaného v co
        bod_t * const *a = _1, * const *b = _2;
        if( (*a)->souradnice[ co[ 0 ] ] == (*b)->souradnice[ co[ 0 ] ] )
                return (*a)->souradnice[ co[ 1 ] ] - (*b)->souradnice[ co[ 1 ] ];
        else
                return (*a)->souradnice[ co[ 0 ] ] - (*b)->souradnice[ co[ 0 ] ];
}

bod_t *sparuj( int n, bod_t *body ) {//Utvoří dvojice podle nastavení v co
        bod_t *pozice[ n ];
        for( int i = 0; i < n; ++ i )
                pozice[ i ] = &body[ i ];
        qsort( pozice, n, sizeof *pozice, porovnej );
        for( int i = 1; i < n; i += 2 ) {
                pozice[ i ]->sousedi[ co[ 1 ] ] = pozice[ i - 1 ];
                pozice[ i - 1 ]->sousedi[ co[ 1 ] ] = pozice[ i ];
        }
        return pozice[ 0 ];
}

int main( void ) {
        int n;
        scanf( "%d", &n );
        bod_t body[ n ];
        for( int i = 0; i < n; ++ i ) {
                scanf( "%d%d", &body[ i ].souradnice[ X ], &body[ i ].souradnice[ Y ] );
        }
        sparuj( n, body );//Svisle
        co[ 0 ] = Y;
        co[ 1 ] = X;
        bod_t *akt = sparuj( n, body ), *start = akt;//Vodorovně a ulož první
```

```
        int smer = VODOROVNE;
        do {//Obejdi
                printf( "%d %d\n", akt->souradnice[ X ], akt->souradnice[ Y ] );
                akt = akt->sousedi[ smer ];
                smer = ( smer + 1 ) % 2;
        } while( akt != start );
        return 0;
}
```

Úloha 20-4-4 – Skupinky pro chytré – program

```
#include <cstdio>
#include <cstring>

#define MAX_NUM_QUEUES 1000

// Struktura osoby. Alokujeme ji jen jednu, ostatni jsou odkazy.

struct Person {
        int IQ;
        char *Name;
        Person (int newIQ, char *newName) {
                IQ = newIQ;
                Name = strdup(newName);
        }
};

#define SWAP(A, B) do {  \
        Person *Tmp = A; \
        A = B;           \
        B = Tmp;         \
} while (0)

// struktura jednoho vrcholu v halde
struct Node {
        Node *Left, *Right;
        Person *P;
};


struct Queue {
        Node Root;
        int Size;
} Queue[MAX_NUM_QUEUES];

int Queues = 2;

// najde nejvetsi prvek - ten je vzdy ve vrcholu haldy
char *
find_best(int ID)
{
        return Queue[ID].Root.P->Name;
}

// odebere koren z haldy
int
delete_best (int ID)
{
        Queue[Queues].Size = Queue[ID].Size - 1;
        Node *Root;
        Node *A = &Queue[ID].Root;
        Node *B = &Queue[Queues].Root;
        int Pos = Queue[ID].Size;
        int log = 0;
        while (Pos >> log)
                log++;

        /* Pujdeme po strome dolu, smerem k poslednimu prvku (Pos), a
           budeme prvky posunovat nahoru */

        *B = *A;
        for (int i = log - 2; i > 0; i--) {
                if ((Pos >> i) & 1) {
                        puts ("right");
                        B->Right = new Node;
                        A = A->Right;
                        B = B->Right;
                } else {
                        puts ("left");
                        B->Left = new Node;
                        A = A->Left;
```

```
                        B = B->Left;
                }
                *B = *A;
        }
        Root = &Queue[Queues].Root;

        /* presunout nejpravejsi prvek v dolni hladine do vrcholu */

        if (Pos & 1) {
                Root->P = B->Right->P;
                B->Right = NULL;
        } else {
                Root->P = B->Left->P;
                B->Left = NULL;
        }

        /* bublat dolu a zabezpecit invariant IQ rodice > IQ synu */

        A = Root;
        while (1) {
                Node *Left = A->Left;
                Node *Right = A->Right;
                if (Right != NULL && Right->P->IQ > Left->P->IQ) {
                        if (Right->P->IQ > A->P->IQ) {
                                A->Right = new Node;
                                *(A->Right) = *Right;
                                SWAP(A->P, A->Right->P);
                                A = A->Right;
                        } else
                                break;
                }
                else if (Left != NULL) {
                        if (Left->P->IQ > A->P->IQ) {
                                A->Left = new Node;
                                *(A->Left) = *Left;
                                SWAP(A->P, A->Left->P);
                                A = A->Left;
                        }
                        else
                                break;
                }
                else
                        break;
        }
        return Queues++;
}

// vlozi prvek do haldy
int
insert (int ID, int IQ, char *Name)
{
        Queue[Queues].Size = Queue[ID].Size + 1;
        Node *A = &Queue[ID].Root;
        Node *B = &Queue[Queues].Root;
        int Pos = Queue[Queues].Size;
        int log = 0;
        while (Pos >> log)
                log++;

        /* odkazy na vrcholy na ceste k nejpravejsimu vrcholu ve
           spodni vrstve */

        Node *Path[log];
        Path[log - 1] = B;

        /* Sejdeme po strome smerem dolu k nejpravejsimu vrcholu */

        for (int i = log - 2; i >= 0; i--) {
                *B = *A;
                if ((Pos >> i) & 1) {
                        B->Right = new Node;
                        if (i)
                                A = A->Right;
                        B = B->Right;
                } else {
                        B->Left = new Node;
                        if (i)
                                A = A->Left;
                        B = B->Left;
                }
                Path[i] = B;
```

```
        }

        // pridat novy vrchol

        B->Left = B->Right = NULL;
        B->P = new Person (IQ, Name);

        // vybublat nahoru a zabezpecit rovnovahu

        for (int i = 0; i < log - 1; i++)
                if (Path[i]->P->IQ > Path[i + 1]->P->IQ)
                        SWAP(Path[i]->P, Path[i + 1]->P);
        return Queues++;
}

int
main(void)
{
        Queue[1].Size = 1;
        Queue[1].Root.P = new Person (0, "UNDERFLOW");
        insert(1, 130, "Ales");
        insert(2, 110, "Petr");
        insert(1, 140, "Jana");
        printf("%s\n", find_best(2));
        printf("%s\n", find_best(3));
        printf("%s\n", find_best(4));
        delete_best(3);
        printf("%s\n", find_best(5));
        return 0;
}
```

Úloha 20-4-5 – Roboti na útěku – program

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

#define MAX_ROWS                20
#define MAX_COLS                20
#define MAX_PATROL_STATES       12
#define MAX_MAZE_SIZE           (MAX_ROWS * MAX_COLS)
#define MAX_CONFIGS             (MAX_PATROL_STATES * (MAX_MAZE_SIZE+1) * (MAX_MAZE_SIZE+1))

const char *moves = "NSEW";

// Koordináty robota v bludišti.
struct scoords {
        char row, col;
};

// Konfigurace robotů a stráží.
struct sconfig {
        unsigned char ticker;   // Tikač určuje, v jaké fázi se nacházejí stráže (počítá modulo 12).
        char move;                              // Tah (N, S, E nebo W), kterým jsme se do této konfigurace dostali.
                                                        // X je počáteční konfigurace a \0 je zatím neprozkoumaná konfigurace.
        struct scoords robot[2];
};

// Struktura zapouzdřující frontu konfigurací.
struct sfifo {
        struct sconfig data[MAX_CONFIGS];
        int first, last;
};

// Struktura zapouzdřující jednu stráž.
struct sguard {
        struct scoords start;   // Políčko, na kterém stráž začíná svou hlídku.
        unsigned char len;              // Délka hlídkovací trasy.
        char direction;                 // Směr, kterým se stráž na začátku dívá (N, S, E nebo W).
};

/*
 * Stavový prostor všech možných konfigurací. Každá konfigurace ukazuje na předchozí tak,
 * jak byly procházeny v BFS. Dosud nenavštívené konfigurace mají nastavený move na \0.
 */
struct sconfig configs[MAX_PATROL_STATES][MAX_MAZE_SIZE+1][MAX_MAZE_SIZE+1];

// Jednotlivá bludiště (hodnota 0 = volné pole, 1 = zeď).
char mazes[2][MAX_ROWS][MAX_COLS];
int rows[2], cols[2];
```

```
// Seznam stráží.
struct sguard guards[2][10];
int guardsCount[2];

// Fronta konfigurací.
struct sfifo fifo;

// Pole pro výsledky.
char results[MAX_CONFIGS];
int resultsCount = 0;

/*
 * Souřadnice a směry.
 */

// Převede znak reprezentující světovou stranu na relativní koordináty pro pohyb.
inline struct scoords getDirectionCoords(char direction) {
        struct scoords res;
        res.row = res.col = 0;
        switch(direction) {
                case 'N':       res.row = -1;   break;
                case 'S':       res.row = 1;    break;
                case 'E':       res.col = 1;    break;
                case 'W':       res.col = -1;   break;
        }
        return res;
}

// Otestuje, zda je robot venku z bludiště.
inline int isRobotOut(struct scoords robot) {
        return (robot.row < 0) ? 1 : 0;
}

// Vrací true, pokud jsou si koordináty rovny. Jinak vrací false.
inline int isEqual(struct scoords coords1, struct scoords coords2) {
        return (coords1.row == coords2.row) && (coords1.col == coords2.col);
}

// Posune robota daným směrem (pokud je to možné).
inline void moveRobot(struct scoords *robot, struct scoords direction, int mazeIdx) {
        if (isRobotOut(*robot)) return;
        struct scoords newPos = *robot;
        newPos.row += direction.row;
        newPos.col += direction.col;
        if ((newPos.row < 0) || (newPos.col < 0) || (newPos.row >= rows[mazeIdx]) || (newPos.col >= cols[mazeIdx]))
                robot->row = -1;
        else if (mazes[mazeIdx][(unsigned)newPos.row][(unsigned)newPos.col] == 0)
                *robot = newPos;
}

/*
 * Práce se stavovým prostorem konfigurací.
 */

// Zakóduje pozici robota do integeru. Bere ohled na to, že robot může být mimo bludiště.
inline int encodeCoords(struct scoords coords) {
        int res = coords.row * MAX_COLS + coords.col;
        if (coords.row < 0) res = MAX_MAZE_SIZE;        // Pokud je robot mimo bludiště.
        return res;
}

// Označí danou konfiguraci za zpracovanou.
inline void markConfigVisited(struct sconfig C, struct sconfig Prev, char move) {
        struct sconfig *config = &configs[C.ticker][ encodeCoords(C.robot[0]) ][ encodeCoords(C.robot[1]) ];
        *config = Prev;
        config->move = move;
}

// Podívá se na stav dané konfigurace.
inline unsigned char isConfigVisited(struct sconfig C) {
        return (configs[C.ticker][ encodeCoords(C.robot[0]) ][ encodeCoords(C.robot[1]) ].move != '\0');
}

// Nalezne následující konfiguraci když se roboti hýbou daným směrem.
inline struct sconfig getNextConfig(struct sconfig config, char move) {
        struct scoords direction = getDirectionCoords(move);
        struct sconfig res = config;
        res.ticker = (res.ticker + 1) % 12;
        for(int i = 0; i < 2; i++)
                moveRobot(&res.robot[i], direction, i);
        return res;
```

```
}

/*
 * Práce s FIFO.
 */

// Inicializuje frontu.
inline void initFifo(struct sfifo *F) {
        F->last = -1;
        F->first = 0;
}

// Pokud je fronta prázdná, vrací true, jinak false.
inline int isFifoEmpty(struct sfifo *F) {
        return (F->last < F->first);
}

// Vloží prvek C do fifo.
inline void pushFifo(struct sfifo *F, struct sconfig C) {
        F->data[ ++F->last ] = C;
}

// Vyjme prvek z fronty a uloží jej do C. Pokud se to povedlo, vrací true, jinak false.
inline int popFifo(struct sfifo *F, struct sconfig *C) {
        if (isFifoEmpty(F)) return 0;
        *C = F->data[ F->first++ ];
        return 1;
}

/*
 * Stráže.
 */

// Vypočítá pozici stráže podle časovače.
inline struct scoords getGuardPos(struct sguard guard, unsigned char ticker) {
        ticker %= (guard.len - 1) * 2;
        struct scoords res = guard.start;
        struct scoords direction = getDirectionCoords(guard.direction);
        int len = (guard.len-1) - abs(ticker - guard.len + 1);
        res.row += direction.row * len;
        res.col += direction.col * len;
        return res;
}

// Otestuje, zda došlo při přesunu do nové konfigurace k polapení některého z robotů.
inline int isCaptured(struct sconfig config, struct sconfig newConfig) {
        struct scoords pos1, pos2;
        for(int i = 0; i < 2; i++)
                if (!isRobotOut(config.robot[i]))
                        for(int g = 0; g < guardsCount[i]; g++) {
                                pos1 = getGuardPos(guards[i][g], config.ticker);
                                pos2 = getGuardPos(guards[i][g], newConfig.ticker);
                                if (isEqual(pos2, newConfig.robot[i]) ||
                                                (isEqual(pos1, newConfig.robot[i]) && isEqual(pos2, config.robot[i])))
                                        return 1;
                        }
        return 0;
}

/*
 * Hlavní procedury.
 */

// Načte z daného souboru jedno bludiště a jeho sttáže.
void loadMaze(FILE *fp, struct scoords *robot, int mazeIdx) {

        // Načteme rozměry.
        fscanf(fp, "%d %d\n", &rows[mazeIdx], &cols[mazeIdx]);

        // Načteme mapu bludiště.
        char ch;
        for(int i = 0; i < rows[mazeIdx]; i++) {
                for(int j = 0; j < cols[mazeIdx]; j++) {
                        fscanf(fp, "%c", &ch);
                        switch(ch) {
                                case '#':       // Nastavíme pole na stěnu.
                                        mazes[mazeIdx][i][j] = 1;
                                        break;

                                case 'X':       // Označíme počáteční pozici.
                                        robot->row = i;
```

```
                                        robot->col = j;
                                        // A tady propadneme do následujícího case...
                                case '.':       // Nastavíme pole jako volné.
                                        mazes[mazeIdx][i][j] = 0;
                                        break;
                        }
                }
                fscanf(fp, "\n");
        }

        // Načteme stráže.
        fscanf(fp, "%d\n", &guardsCount[mazeIdx]);
        for(int i = 0; i < guardsCount[mazeIdx]; i++) {
                int row, col, len;
                fscanf(fp, "%d %d %d %c\n", &row, &col, &len, &guards[mazeIdx][i].direction);
                guards[mazeIdx][i].start.row = row - 1;
                guards[mazeIdx][i].start.col = col - 1;
                guards[mazeIdx][i].len = len;
        }
}

// Načteme data ze vstupního souboru.
void load(void) {
        FILE *fp = fopen("robots.in", "r");
        if (!fp) exit(1);

        struct sconfig config;
        config.ticker = 0;
        for(int i = 0; i < 2; i++)
                loadMaze(fp, &config.robot[i], i);

        fclose(fp);
        pushFifo(&fifo, config);
        configs[0][ encodeCoords(config.robot[0]) ][ encodeCoords(config.robot[1]) ].move = 'X';
}

// Pustí nad daným bludištěm prohledávání do šířky.
void bfs(void) {
        struct sconfig config, newConfig;
        while(popFifo(&fifo, &config)) {
                for(int i = 0; i < 4; i++) {
                        newConfig = getNextConfig(config, moves[i]);
                        if (!isConfigVisited(newConfig) && !isCaptured(config, newConfig)) {
                                markConfigVisited(newConfig, config, moves[i]);
                                if (isRobotOut(newConfig.robot[0]) && isRobotOut(newConfig.robot[1]))
                                        return;
                                pushFifo(&fifo, newConfig);
                        }
                }
        }
}

// Zapíše výsledky do souboru.
void writeResults(void) {
        FILE *fp = fopen("robots.out", "w");
        if (!fp) return;

        // Projdeme možné koncové stavy.
        int i = 0;
        while((i < MAX_PATROL_STATES) && (configs[i][MAX_MAZE_SIZE][MAX_MAZE_SIZE].move == 0))
                i++;

        // Neexistuje korektní řešení.
        if (i == MAX_PATROL_STATES) {
                fprintf(fp, "-1\n");
                return;
        }

        // Projdeme cestu zpet k pocatecnimu stavu.
        struct sconfig *config = &configs[i][MAX_MAZE_SIZE][MAX_MAZE_SIZE];
        while(config->move != 'X') {
                results[resultsCount++] = config->move;
                config = &configs[config->ticker][ encodeCoords(config->robot[0]) ][ encodeCoords(config->robot[1]) ];
        }

        // Nalezenou cestu zapíšeme v opačném pořadí do souboru.
        fprintf(fp, "%d\n", resultsCount);
        while(resultsCount-- > 0)
                fprintf(fp, "%c\n", results[resultsCount]);
}
```

```
int main(void) {
        initFifo(&fifo);
        load();
        bfs();
        writeResults();
    return 0;
}
```

Výsledková listina dvacátého ročníku KSP po čtvrté sérii

		škola	*ročník*	*sérií*	*2041*	*2042*	*2043*	*2044*	*2045*	*2046*	*série*	*celkem*
1.	Peter Ondrúška	SPŠDubnica	4	4	10			10	15	12	47,0	176,2
2.	Jan Michelfeit	G HBrod	4	9	10	8		7		8	34,3	144,3
3.	Alena Skálová	GNaVPláni	4	5		8	7			8	24,9	134,8
4.	Filip Hlásek	GMikulášPL	1	4		7	0	5		8	24,9	132,9
5.	Radim Cajzl	G NMnMor	1	14						88	88,0	127,7
6.	Petr Malý	GSladNámPH	4	4			6			8	16,7	115,9
7.	Vlastimil Dort	GŠpitálsPH	2	9				5		8	14,4	105,9
8.	Štěpán Weber	GBuďánkaPH	3	3		8	7			8	25,4	104,3
9.	Filip Štědronský	GMikulášPL	1	4		7,5	0			10	19,0	100,1
10.	Libor Plucnar	GPBezruče	3	8							0,0	97,8
11.	Trung Ha duc	GMasarykPL	2	9		1					1,3	93,0
12.	Pavel Veselý	G Strakon	3	11		1	1	4		8	14,0	86,9
13.	Tomáš Toufar	G Bílovec	4	3							0,0	83,5
14.	Vojtěch Tůma	G Jihlava	4	7							0,0	81,3
15.	Stanislav Fořt	G Tábor	0	7							0,0	60,2
16.	Jan Škoda	GMikulášPL	1	3							0,0	59,1
17.	Jitka Novotná	G Bílovec	3	2							0,0	58,6
18.	Lukáš Kripner	G Litvínov	2	6							0,0	57,7
19.	Jakub Hrnčíř	GFXŠaldyLI	1	3							0,0	55,9
20.	Petr Babička	G SvětláNS	3	4		1				8	12,1	55,3
21.	Pavel Kratochvíl	ZŠSvětlá	0	4			6			8	16,7	51,0
22.	David Marek	SPŠ Zlín	4	4							0,0	49,2
23.	Jan Matějka	GJírovcoČB	3	4							0,0	35,4
24.	Milan Rybář	GJungmanLT	3	2				3			5,7	35,0
25.	Jakub Kaplan	GJKTylaHK	4	16							0,0	34,4
26.	Roman Smrž	GOhradníPH	4	2					15		15,0	34,0
27.	Jiří Zárevúcky	SŠInformFM	3	1							0,0	32,5
28.	Tomáš Sýkora	G VKlobou	4	11							0,0	30,8
29.	David Brázdil	G Zlín	3	6							0,0	29,8
30.	Martin Vlach	G Jihlava	4	1							0,0	29,7
31.	Karel Tesař	SPŠEPlzeň	2	2							0,0	28,4
32. – 33.	Jiří Keresteš	SPŠEPlzeň	2	4							0,0	28,2
	Petr Sokola	SPŠ Zlín	4	2							0,0	28,2
34.	Adam Streck	G Hořice	4	1							0,0	27,6
35.	Vojtěch Kolář	G Neratov	3	1							0,0	27,3
36.	Jakub Červenka	GŠpitálsPH	2	4							0,0	26,0
37.	Pavel Taufer	GArcibisPH	2	1							0,0	25,7
38.	Martin Patera	GArabská	2	1							0,0	25,5
39. – 40.	Jakub Suchý	GMikulášPL	1	2							0,0	24,1
	Jan Žák	G HBrod	3	6							0,0	24,1
41.	Alžběta Pechová	SPŠSVsetín	3	4		1					2,0	22,6
42.	Jiří Setnička	G25březnPH	1	1			2	4		8	21,8	21,8
43. – 44.	Tomáš Jakl	G MTřebová	4	4		1	5	4			14,4	19,4
	Miroslav Klimoš	G Bílovec	3	21					15		15,0	19,4
45.	Jan Vaňhara	G Holešov	3	1							0,0	17,8
46.	Dominik Smrž	GOhradníPH	0	1			7	4			14,1	14,1
47.	Petr Holášek	G Příbor	4	5							0,0	10,5
48.	Nikolas Zigmund	ZŠHavířov	1	1							0,0	8,9
49.	Lukáš Timko	G Tábor	0	2		1		1			4,7	8,6
50.	Peter Uhnák	GBBolzana	2	1		6					7,4	7,4
51.	Peter Smatana	EkoGLabsBO	3	1							0,0	6,4