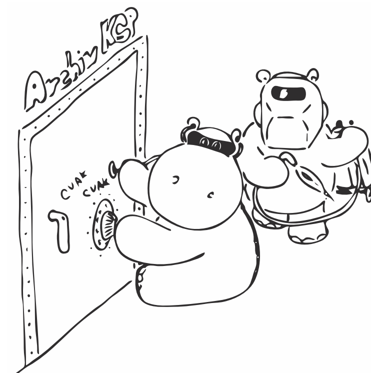
Milí řešitelé a řešitelky!

Vítejte u dalšího vysílání pořadu KSP. Pátý díl má název „Kterak řešitel k bodům přišel“. Za devatero horami, devatero řekami, devatero routery a devatero firewally … bzzzzt … Přerušujeme vysílání, abychom vám přinesli aktuální zprávy. Včera byla z přísně střeženého archívu KSP odcizena řešení úlohy 18-3-4 „Pochoutka pro prasátko“. Neznámý pachatel, nebo pachatelé, si úlohy odnesli z dosud neznámého důvodu. Policie po nich v současné době pátrá. Vedení KSP se tímto omlouvá všem řešitelům za vzniklé potíže. „Jako nejspravedlivější řešení se nám jeví prodloužení termínu odevzdání úlohy 18-3-4 do 20. března, kdy je také termín odevzdání 4. série,“ uvedl pro HTK Milan Foxík. Tiskový mluvčí KSP dále potvrdil, že úlohu 18-3-4 mohou odevzdat i řešitelé, kteří ji původně neodevzdali, tedy kdokoliv. Řešení odevzdaná elektronicky není třeba posílat znovu. Tímto se s vámi loučím a vracím slovo zpět do studia … bzzzzt … a jestli nezemřeli, žijí tam dodnes.



Termín odeslání páté série jest tentokrát dne 8. května 2006. Řešení můžete odevzdávat jak elektronicky na *http://ksp.mff.cuni.cz/submit/*, tak klasickou poštou na známou adresu:

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

Aktuální informace o KSP naleznete na stránkách *http://ksp.mff.cuni.cz/*. Dotazy ohledně zadání můžete posílat na adresu *ksp@mff.cuni.cz*, nebo se ptát přímo na diskusním fóru KSP (*http://ksp.mff.cuni.cz/forum/*).

Zadání páté série osmnáctého ročníku KSP

18-5-1 Účetní — 8 bodů

V nedávné době došlo k jedné z největších změn na trhu počítačového hardware. Společnost IBM (Inevitable Bureau Meltdown) byla prodána koncernu Le Nouveau (to znamená … to je … to je jedno). Všichni tento převod nadšeně uvítali, až na tři povedené pány účetní z IBM. Doteď poctivě tunelovali celou společnost a převodní audit by odhalil jejich nekalou činnost. Proto se rozhodli, že už se svou profesí (tunelováním) skoncují a odejdou do důchodu na Seychely. Problém je v tom, že se nemohou dohodnout, jak nahromaděný majetek spravedlivě rozdělit na tři díly. Majetku je opravdu velké množství a jeho hodnota se nedá objektivně změřit (např. zlatem, nebo penězi). Hodnota některých částí (např. pozemky nebo podílové listy) se každý den mění. Prodat všechen majetek také nemohou, protože audit je za dveřmi a na prodej nezbývá čas. Pokud se vašemu gustu nepříčí ekonomicko-logické úložky, můžete jim zkusit pomoci.

Účetní nemají mnoho možností. Každý z nich umí rozdělit majetek na několik (libovolně mnoho) stejných částí. Případně umí i část majetku rozdělit na několik menších částí. Dělení vždy probíhá subjektivně z pohledu toho, kdo dělení provádí. Když jeden účetní rozdělí majetek na (podle něho) stejné tři části, druhém účetnímu se mohou části zdát různě velké, a proto se nespokojí s libovolnou částí, ale bude chtít jen ty, které podle něj odpovídají alespoň třetině majetku.

Navrhněte algoritmus, který rozdělí majetek mezi účetní tak, aby byli všichni účetní spokojeni. Účetní je spokojen se svým podílem, pokud má (podle svého mínění) alespoň třetinu majetku.

Příklad: První rozdělí majetek na 3 části, o kterých tvrdí, že jsou stejné. Druhý si vybere část, která se mu zdá největší. Třetí si vybere část, která se mu zdá největší, a na prvního zbyde poslední část. První a druhý jsou spokojeni. První si myslí, že všichni mají spravedlivé třetiny, a druhý si mohl vybrat ze všech dílů ten, který se mu zdál největší. Třetí ale nemusí být spokojený, protože se mu může zdát, že části, které na něj zbyly, jsou obě příliš malé (ani jedna z nich není vetší než třetina celého majetku). Toto řešení tedy není správné.

Hint: Zkuste nejprve vyřešit situaci pouze pro dva účetní. I za to už bude nějaký bodík.

Bonus: Pokud úlohu vyřešíte obecně pro N účetních, sladká (bodová) odměna vás nemine.

18-5-2 Permutovat se musí legálně! — 7 bodů

Ministerstvo všech permutací (to je to ministerstvo, na kterém pracuje $\mathcal{O}(N!)$ úředníků) se rozhodlo, že je nejvyšší čas obměnit (tedy zpermutovat) vyhlášky a ostatní legislativní usnesení týkající se regulace všech permutačních živností. Od této chvíle musí všichni živnostníci generovat svým zákazníkům permutace pouze v lexikografickém uspořádání. Jako vždy zpermutovaní úředníci na něco zapomněli. Neuvědomili si, že někteří živnostníci mohou být právě uprostřed generování nějaké řady permutací a tohle jim může zkazit všechnu práci, kterou do této chvíle udělali. Proto vydali ještě doplňující vyhlášku. Ta nařizuje všem živnostníkům, kteří mají právě rozdělanou nějakou tu řadu permutací, aby vzali poslední vygenerovanou permutaci a od ní dále generovali permutace v lexikografickém pořadí, bez ohledu na to, které permutace již vygenerovali a které ne. Vy, coby zkušení programátoři, jste si okamžitě všimli potencionální marketingové trhliny na permutačním trhu a začali jste vyvíjet nový software, na kterém hodláte strašlivě zbohatnout.

Napište funkci, která na vstupu dostane permutaci alfanumerických znaků (zadanou jako řetězec) a vrátí následující permutaci podle lexikografického uspořádání.

V řetězci se mohou vyskytovat znaky `0`-`9` a `a`-`z` (nerozlišujeme velká a malá písmena) a každý znak se v permutaci vyskytuje nejvýše jednou. Uspořádání je definováno tak, že $\mathtt{0} < \mathtt{1} < \ldots < \mathtt{9} < \mathtt{a} < \ldots < \mathtt{z}$, a permutace $p_1 p_2 \ldots p_k$ je lexikograficky menší než $q_1 q_2 \ldots q_k$ právě tehdy, když je $p_1 < q_1$ (podle našeho uspořádání) nebo když je $p_1 = q_1$ a permutace $p_2 p_3 \ldots p_k$ je lexikograficky menší než $q_2 q_3 \ldots q_k$. Vzhledem k tomu, že váš program má udělat díru do světa (ehm … pokud možno jen obrazně), měli byste tuto funkci napsat tak, aby fungovala co nejrychleji.

Příklad: Máme permutaci `32a9`, lexikograficky následuje `392a`, `39a2`, `3a29`, `3a92`, `923a` atd.

18-5-3 Číňanské volby — 10 bodů

Už jste to slyšeli? V Číně padla vláda a číňané, číňanky i malá číňančata jsou zvědaví, jaké to bude mít demokracii. Celá nová Čínská republika se připravuje na nadcházející volby čínského prezidenta. Vzhledem k tomu, kolik číňanů je, panují ohledně voleb veliké zmatky. Kandidátem se totiž může stát každý číňan a každý číňan odevzdává jeden hlas. Vyhodnotit takové množství hlasů dá přeci jen práci, takže jste to dostali na starosti vy.

Volby již proběhly a výsledky máte načtené v počítači. Máte pole A o velikosti N, kde N je gigantické číslo, a v tomto poli jsou uložena čísla kandidátů, na jejichž velikost však není *žádné omezení*. Hodnota A_i tedy udává číslo kandidáta, kterého volil i-tý číňan. Máte za úkol zjistit, zda volby někdo vyhrál, tedy zda má nějaké číslo (kandidát) v poli A nadpoloviční počet výskytů (tj. $> N/2$) a pokud ano, vypsat které.

Číňani jsou celí dychtiví mít už už ve vládním paláci svého oblíbeného disidenta, takže byste měli raději vymyslet algoritmus pracující v čase $\mathcal{O}(N)$. Pole A je ale opravdu obrovské, takže na ostatní proměnné si budete muset vystačit pouze s konstantním množstvím paměti (čili $\mathcal{O}(1)$). Aby toho pořád nebylo málo, tak do pole A (z archivních důvodů) nesmíte v žádném případě zapisovat a to ani kdybyste ho na závěr vrátili zpět do původního stavu (v počítačové řeči je A read-only). Pomalejší či paměťově náročnější algoritmy budou mít příslušně snížená bodová hodnocení.

Dodáme jednu nápovědu z Říše Středu: zkuste tentokrát nepoužívat nejrůznější rafinované programátorské triky, fígle a datové struktury – jděte na to s logickým myšlením a matematikou. Nezapomeňte však, že vaše řešení by mělo obsahovat také důkaz správnosti, aby vám číňani věřili.

Příklad: Pro $N = 6$ a pole A obsahující $(2, 666, 2, 1000, 2, 2)$ je správná odpověď, že vyhrál kandidát číslo 2, pro pole A obsahující $(1, 1, 1, 2, 3, 4)$ volby nemají vítěze.

18-5-4 Detektýv — 10 bodů

Slavný detektiv Šérlok Houmles je na stopě vážného zločinu. A to doslova a do písmene. S lupou až u země právě prohlíží stopy, které by ho měly dovést k pachateli. Stop je ale příliš mnoho a jeho zajímají jen určité podezřelé sekvence stop. Práce je to velmi zdlouhavá, takže je téměř jisté, že mu zločinec zatím unikne. Pomůžete detektivovi s jeho případem?

Stopy jsou uspořádány do řady. Navíc každou stopu lze označit nějakým písmenem, nebo jiným znakem a těchto „typů“ stop není mnoho (desítky až stovky).

Dále má Šérlok k dispozici Knihu Stopování Pachatelů, ve které jsou popsány všechny podezřelé výskyty stop.

Např: Kniha praví, že stopy `LPLPO` – znamenají Levá, Pravá, Levá, Pravá a Otočení, což je velice podezřelé, neboť člověk, který takové stopy udělal, normálně šel a z ničeho nic se prudce otočil.

Na vstupu dostanete všechny podezřelé sekvence stop a dále řetězec stop, které detektiv sleduje. Tento řetězec je velice dlouhý a nevejde se do operační paměti. Pro jednoduchost předpokládejte, že existuje funkce `GetFootprint`, která vrací právě přečtenou stopu (např. jako znak) a procedura `RewindFootprint`, která vrátí detektiva na začátek stop. Váš program by měl zjistit ke každé sekvenci podezřelých stop, kolikrát se vyskytla během stopování. Zároveň si uvědomte, že času je málo, a tak by váš program měl pracovat ideálně v čase $\mathcal{O}(N+P)$ (N je délka stopovaného řetězce a P je součet délek všech podezřelých stop), bez ohledu na počet výskytů podezřelých sekvencí, přestože jejich počet může být až $\mathcal{O}(N \cdot k)$, kde k je počet podezřelých sekvencí. Detektiv také nemůže stále běhat sem a tam, takže váš program by měl funkci `RewindFootprint` volat co nejméně (ideálně vůbec).

Nicméně i řešení v čase $\mathcal{O}(N \cdot k + P)$ je daleko hodnotnější než řešení se složitostí $\mathcal{O}(N \cdot P)$.

Příklad: Podezřelé sekvence stop jsou `LPBBLP`, `BBBBO`, `OSSO`. Prohledávané stopy buďtež `OSSOSSOLPBBLPBBLPBBBBO`. Výstup programu by měl být

podezřelý vzorek	*počet jeho výskytů*
`LPBBLP`	2
`BBBBO`	1
`OSSO`	2

Vysvětlivky pro zvídavé čtenáře: `L`, `P` a `O` – již známe (viz výše), `B` – Běh (rozmazaná stopa), `S` – Stání (stopa bez náznaků pohybu).

18-5-5 Do vysokých kruhů — 13 bodů

Zchudlý šlechtic hrabě Karl von Quadrat se celý život toužil dostat do vyšších kruhů. Pilně se účastnil všech večírků, dýchánků, plesů a jiných společenských událostí, avšak nikdy se mu nepodařilo seznámit se s někým vlivným, mocným, nebo alespoň bohatým. A protože se věnoval jen samé zábavě, stával se chudším a chudším, až si nemohl dovolit chodit na společenské události vůbec. Karl však neztrácel hlavu. Za zbylé peníze si pořídil kružítko a spoustu papíru a rozhodl se, že když nepronikl do kruhů společenských, pronikne alespoň do tajemství kruhů geometrických.

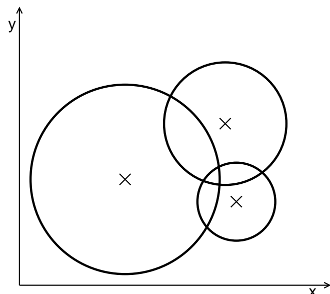
Celé hodiny vysedával za stolem a rýsoval kruhy malé, kruhy větší, kruhy tečné i sečné, a když byl v dobrém rozmaru, dokonce i kruhy soustředné. Jednou si takhle navečer opět hrál s kružítkem a pokreslil celý papír spoustou kružnic. Chtěl si vzít nový papír a starý zahodit, když v tom ho napadlo, že všechny ty kružnice vlastně rozdělili rovinu papíru na spoustu dílů. Protože byl zvědavý a stejně neměl nic lepšího na práci, rozhodl se, že ty díly spočítá. Počítání dílů roviny má velice podobné účinky, jako počítání oveček (nebo hrošíků), takže unavený hrabě brzy usnul. V několika následujících dnech se o to pokusil ještě několikrát, avšak vždy se stejným účinkem. Sotva začal počítat, víčka mu ztěžkla a po chvíli již dřímal spánkem spravedlivých.

Hrabě již nemá žádné služebnictvo, a tak poprosil vás, zda byste mu s tím mohli pomoci. Napište program, který dostane na vstupu N kružnic zadaných souřadnicemi středu a poloměrem (souřadnice a poloměry jsou reálná čísla), a na výstup vypíše, na kolik částí dělí tyto kružnice rovinu. Můžete předpokládat, že se žádné tři kružnice neprotínají v jednom bodě. Rovina bez kružnic se považuje za jeden díl, jedna kružnice rozdělí rovinu na dva díly (vnitřek a vnějšek kružnice) atd.

Příklad: Na vstupu jsou 3 kružnice:

x	y	r
1	1	0,9
2	0,8	0,4
1,9	1,5	0,6

Rovina je pak rozdělena na 8 částí.



18-5-6 Proφlování 10 bodů

V posledním díle seriálu o kompilátorech si povíme něco o optimalizacích řízených profilem. Ve většině programů bývá spousta kódu, který se neprovádí příliš často – občas se uvádí, že zhruba 90% času se stráví v 10% kódu. Takovému často prováděnému kódu se říká *horký* a zbytku *studený*. Vědět, kterých 10% kódu je horkých, by bylo pro kompilátor velmi užitečné – optimalizací zbylého studeného kódu nemusí ztrácet čas, a přitom dostane skoro stejně dobrý výsledek. Případně může studený kód optimalizovat jinak, třeba na velikost místo rychlosti, a tím dostat skoro stejně dobrý výsledek, který je navíc podstatně menší.

Zmenšení kódu může samo o sobě vést k jeho zrychlení. Důležitou součástí každého moderního procesoru jsou keše – velmi rychlé paměti, do kterých si ukládá kusy dat a programů, které právě zpracovává, aby k nim nemusel neustále přistupovat do poměrně pomalé hlavní paměti. Velikosti keší ovšem bývají poměrně malé (typicky řádově desítky až stovky kB), takže zatímco malé programy se do nich vejdou, u velkých to může být docela problém.

Kompilátor navíc může kód přeskládat tak, aby horké části byly blízko sebe. Keše fungují tak, že pokud jsou plné a je potřeba přistoupit k novému kusu paměti, nějaký jiný se z keše vyhodí. Je mnoho algoritmů, které se snaží nějak rozumně volit, který kus z keše vyhodit; typicky se navzájem nebudou vyhazovat kusy paměti, které jsou blízko u sebe, aby se rozumně chovaly programy, které čtou paměť víceméně sekvenčně. Takže pokud horký kód bude naměstnán do jednoho místa, nejspíš se cely vejde do keší (a jen málo často se stane, že nějaký kus z něj vypudí studený kód).

Znalost toho, jak často se které kusy kódu (typicky basic bloky, nebo hrany v CFG) budou provádět (takovému popisu se říká *profil* programu), můžeme využít i pro další optimalizace. Například si můžeme spočítat, s jakou pravděpodobností budou splněny podmínky v programu a přeuspořádat je tak, aby procesor dokázal uhodnout, zda se skok provede či ne: Procesor většinou zpracovává několik instrukcí zároveň. Když narazí na podmíněný skok, musel by čekat, než se vyhodnotí jeho podmínka, aby věděl, kudy dál. Místo toho si tipne, jaký bude výsledek. Pokud později zjistí, že se spletl, musí zahodit vše, co doteď spočítal, a vrátit se zpět. Je samozřejmě výhodné, aby se tohle dělo co nejméně. Nejjednodušší z heuristik, které se používají, je, že skoky zpět se provedou, zatímco skoky dopředu ne. Je tedy výhodné, aby kompilátor podmínky testoval tak, aby výsledky odpovídaly této heuristice.

Existuje ještě mnoho dalších využití pro znalost profilu. Nicméně otázkou je, jak by kompilátor mohl profil získat, když program, jehož se má týkat, teprve vyrábí. Používají se zejména následující postupy:

1) *Měření profilu.* Program nejprve zkompilujeme a řekneme kompilátoru, aby do něj přidal kód pro měření profilu. Například z programu

```
var i: integer;

begin
for i := 1 to horní_mez do
  begin
    if test(i) then
      vlevo;
    else
      vpravo;
  end;
end.
```

vznikne

```
var i: integer;
    ctr: array[0..5] of integer;

begin
vynuluj (ctr);
inc (ctr[0]);

for i := 1 to horní_mez do
  begin
    inc (ctr[1]);
    if test(i) then
      begin
        inc (ctr[2]);
        vlevo;
      end
    else
      begin
        inc (ctr[3]);
        vpravo;
      end;
    inc(ctr[4]);
  end;

inc(ctr[5]);
ulož (ctr);
end.
```

Takto zkompilovaný program spustíme. Spočítají se hodnoty čítačů `ctr`, které udávají, kolikrát se provedla která hrana CFG, a uloží se do souboru. Pak program zkompilujeme ještě jednou, a řekneme kompilátoru, aby si profil načetl z tohoto souboru.

2) *Hádání profilu.* Výše popsaný postup je přesný, ale mírně nepraktický (program musíme kompilovat dvakrát s různými volbami pro překladač, a mezitím ho musíme spouštět na nějaká rozumná data, což třeba u interaktivních programů jde poměrně těžko). O dost pohodlnější, ale také podstatně méně přesnou možností, je nechat kompilátor profil uhodnout. Kompilátor přitom vychází z typického chování programů – například toho, že většina čísel bývá nezáporná, ukazatele většinou nebývají `nil` apod. Pomocí podobných pravidel si pro každou podmínku určíme, s jakou pravděpodobností bude splněna, a z těchto pravděpodobností pak dopočteme, kolikrát se která hrana CFG provede.

3) *JIT (Just In Time) kompilace.* Další, poněkud zběsile vypadající variantou, je program překompilovávat za běhu. Program na začátku zkompilujeme jen velmi rychle, s minimem optimalizací, a připojíme k němu navíc ještě kompilátor. Za běhu programu se pak měří, kolikrát se která část provede, a když zjistíme, že některá část je horká, překompilujeme ji s vyšší úrovní optimalizací.

Úlohy:

1) Povšimněte si, že ve výše uvedeném programu jsme nemuseli mít všechny čítače. Například na konci programu bude vždy platit, že `ctr[5]=ctr[0]` a `ctr[1] = ctr[2] + ctr[3] = ctr[4]`, tedy stačí čítače `ctr[0]`, `ctr[2]` a `ctr[3]`. Mějme program a jeho CFG. Na kolik nejméně

hran CFG musíme umístit inkrementaci čítače, abychom dokázali vždy určit, kolikrát se která hrana provedla? Popište také algoritmus, který dopočítá počty provedení hran, na nichž nejsou čítače.

2) U hádání profilu jsme si řekli, že kompilátor uhodne pravděpodobnosti, s jakými jsou splněny jednotlivé podmínky, a pak dopočítá, kolikrát se která hrana provede. Jak to udělá? Popište algoritmus, který z pravděpodobností podmínek dopočítá profil. Zadání je CFG, v němž máme hranám na konci každého basic bloku určeno, s jakou pravděpodobností bude daná hrana zvolena, a víme, že počáteční basic blok bude proveden právě jednou.

Příklad: Mějme následující CFG: vstupní blok má číslo 0. Z BB 0 vede jediná hrana (s pravděpodobností 100%) do BB 1. Z BB 1 vedou dvě hrany – jedna z nich se vrací na začátek BB 1 (tato hrana je zvolena s pravděpodobností 90%) a druhá hrana vede do výstupního basic bloku 2 (s pravděpodobností 10%). Tato situace odpovídá programu se smyčkou, která se provede právě 10×. V tomto program se BB 0 a BB 2 provedou 1×, a BB 1 se provede 10× (hrana z BB 1 do BB 1 se provede 9× a hrana z BB 1 do BB 2 se provede 1×, tedy jejich pravděpodobnosti opravdu jsou 90% a 10%).

Recepty z programátorské kuchařky

V dnešním vydání kuchařky se podíváme na vyhledávání slov v textu. Náš úkol tentokrát zní: Máte seznam slov a *hodně dlouhý* text, vypište všechny výskyty těchto slov v textu. Ukážeme si řešení, kterému stačí jeden průchod textem a lineární čas na předzpracování slovníku.

Pro začátek si zavedeme několik pojmů:

- Mějme nějakou konečnou *abecedu* Σ, tedy množinu všech *znaků.* Klidně si představujte klasickou latinskou abecedu, ale může to být např. i množina $\{0, 1\}$.
- Σ^* je množina všech *slov*, která lze z naší abecedy utvořit. To jsou všechny konečné posloupnosti znaků z Σ. Takové slovo může tudíž být i posloupnost 01101. Slova budeme značit řeckými písmenky a zvláštní postavení mezi nimi má *prázdné slovo* ε.
- $|\alpha|$ pro $\alpha \in \Sigma^*$ je délka slova, tedy počet jeho znaků.
- $\alpha\beta$ pro $\alpha, \beta \in \Sigma^*$ je zřetězení slov α a β, tedy slovo, které vznikne zapsáním slov α a β za sebe.
- γ^k je slovo vzniklé k-násobným zopakováním slova γ. Tedy $\gamma^0 = \varepsilon$, $\gamma^{k+1} = \gamma^k\gamma$.
- Slovo α nazveme *podslovem* slova β, pokud je α obsaženo v β, čili pokud $\beta = \gamma\alpha\delta$ pro nějaká slova γ a δ.
- Řekneme, že slovo α je *prefixem* slova β, pokud slovo β začíná slovem α, čili $\beta = \alpha\delta$ pro nějaké slovo δ.
- Podobně α je *suffixem* slova β, pokud β končí slovem α, tedy $\beta = \delta\alpha$ pro nějaké slovo δ.
- Každé slovo je prefixem i suffixem sebe sama, takovému pre-/suffixu říkáme *vlastní;* všem ostatním *nevlastní.*
- Všimněte si, že prázdné slovo je podslovem, prefixem i suffixem každého slova včetně prázdného slova.

Po tomto teoretickém úvodu se konečně zamyslíme nad vlastním vyhledáváním. Ponejprv si úlohu trochu zjednodušme a zkoumejme případ, kdy hledáme všechny výskyty jednoho slova $\alpha \in \Sigma^*$ o délce $|\alpha| = p$ v textu $\beta \in \Sigma^*$, $|\beta| = n$. (Hledanému slovu se často říká jehla, textu kupka sena.)

Asi první algoritmus, který nás napadne, je procházet text β od začátku až do konce a pro každou pozici i v textu zkontrolovat, zda na této pozici nezačíná hledané slovo. Tak pro každou pozici provedeme až p porovnání znaků, čili celkem až np porovnání. To není nic pěkného, zkusme to lépe.

Všimněme si, že porovnávání slova s textem může skončit dvěma způsoby. Buď zjistíme, že se slovo s textem shoduje celé, nebo najdeme v textu znak, který ve slově není. Tehdy nestačí pokračovat novým vyhledáváním od místa, kde jsme skončili: např. pro slovo `instinkt` a text `instinstinkt` by algoritmus u druhého `s` zjistil, že se text liší, a pokud by pokračoval dále, již by nenalezl skutečný výskyt slova. Proto se vždy musíme vrátit o kousek zpět, v předchozím algoritmu jsme se vraceli vždy těsně za místo, kde se text začal se slovem shodovat.

Na druhou stranu, když se takto vrátíme, začneme znovu zpracovávat text, který už jsme jednou četli, takže je vlastně předem dáno, jak to dopadne. Pojďme toho využít. Říkejme *stavy* prefixům slova α. Pro každou pozici i v textu si označme $r[i]$ nejdelší stav, který je obsažen v textu tak, že v něm končí na pozici i (nebo vezměme nejdelší suffix prvních i znaků textu, který je stavem – to je totéž). Posuneme-li se v textu o pozici dále, další znak $\beta[i+1]$ buď prodlouží prefix $r[i]$, a tím určitě získáme nový nejdelší stav $r[i+1]$ (rozmyslete si, že nemůže existovat delší), nebo už prefix není možné prodloužit, a tehdy budeme muset najít jiný. Nahlédněme ale, že useknutím posledního písmenka stavu získáme zase stav, takže useknutím posledního písmenka stavu $r[i+1]$ získáme nějaký suffix stavu $r[i]$. Naše $r[i+1]$ tedy vznikne prodloužením co možná nejdelšího suffixu stavu $r[i]$ o písmenko $\beta[i+1]$ (některé suffixy prodloužit nejdou, vezměme nejdelší, který jde). Pro předchozí příklad a prefix `instin` to bude suffix `in`.

Jelikož nový stav získáme ze suffixů předchozího stavu, nemusíme vědět vůbec nic o předcházejících písmenech textu. Postačí nám předpočítat si pro každý stav σ jeho nejdelší vlastní suffix, který je také stavem – ten si označíme $f(\sigma)$ a funkci f budeme říkat *zpětná funkce.* Přechod od $r[i]$ k $r[i+1]$ budeme provádět tak, že zkusíme $r[i]$ prodloužit o znak $\beta[i+1]$ a když to nepůjde, zkrátíme si $r[i]$ pomocí zpětné funkce a opět zkusíme přidat tentýž znak, pokud to stále nejde, zkracujeme dál opětovným zavoláním zpětné funkce, dokud se nám prodloužení nezdaří nebo dokud nedostaneme prázdné slovo.

Když navíc během výpočtu narazíme na i, pro které je $r[i] = \alpha$, ohlásíme výskyt slova α.

Aby se nám se stavy v programu pohodlně pracovalo, očíslujeme si je – j-tý stav bude prefixem slova α o délce j. Zpětná funkce pak bude přiřazovat číslům čísla, takže si ji můžeme pamatovat v obyčejném jednorozměrném poli.

Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? Jak spočítat zpětnou funkci? Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až p-krát. Při každém volání však klesne délka aktuálního stavu alespoň o jedna, zatímco kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků lineární v délce textu.

Konstrukci zpětné funkce provedeme malým trikem. Všimněte si, že $f(i)$ je přesně stav, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec $\alpha[2 \ldots i]$, čili na i-tý prefix bez prvního písmenka. Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco $r[i]$ označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku. Takže f získáme tak, že spustíme vyhledávání na část samotného slova w. Jenže k vyhledávání zase potřebujeme funkci f. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $f(1) = \varepsilon$. Pokud již máme $f(i)$, pak výpočet $f(i+1)$ odpovídá spuštění automatu na slovo délky i a při tom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku – $(i+1)$-ní prefix je přeci prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na celý řetězec $\alpha[2 \ldots p]$ a sledovat, jakými stavy bude procházet, a to budou přesně hodnoty zpětné funkce. Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $p-1$, a proto poběží v čase $\mathcal{O}(p)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(n+p)$. Dodáme už jen, že tento algoritmus poprvé popsali pánové Knuth, Morris a Pratt a na jejich počest se mu říká KMP. Naprogramovaný bude vypadat následovně (čtení vstupu jsme si odpustili):

```
var
  Slovo: array[1..P] of Char;   { jehla }
  Text: array[1..N] of Char;    { seno }
  F: array[1..MaxS] of Integer; { zpětná fce }

  function Krok(I: Integer; C: Char): Integer;
  begin
    if (I < P) and (Slovo[I+1] = C) then
      Krok := I + 1
    else if I > 0 then
      Krok := Krok(F[I], C)
    else
      Krok := 0;
  end;

var
  I, R: Integer;  { pomocné proměnné }
begin
  { konstrukce zpětné funkce }
  F[1]:= 0;
  for I:= 2 to P do
    F[I]:= Krok(F[I-1], Slovo[I]);

  { procházení textu }
  R:= 0;
  for I:= 1 to N do
  begin
    R:= Krok(R, Text[I]);
    if R = P then
      writeln(I);
  end;
end.
```

Tento algoritmus můžeme také formálně popsat pomocí automatů:

Konečný automat nad abecedou Σ si můžeme představit jako stroj, kterému dáme slovo ze Σ^* a on ho buď odmítne nebo přijme. V průběhu práce je vždy v právě jednom *stavu* z nějaké pevné množiny stavů. Slovo zpracovává po jednotlivých znacích a podle přečteného znaku se rozhodne, do jakého stavu přejde. K tomu slouží *přechodová* funkce g, která dvojicím (*aktuální stav*, *nový znak*) přiřazuje nové stavy. Pokud vstupní slovo dojde, automat podle toho, v jakém stavu se právě nachází, odpoví, že je slovo přijato nebo odmítnuto.

Konečný automat můžeme formálně nadefinovat jako čtveřici (Q, g, q_0, F), kde:

- Q je konečná množina stavů automatu;
- $g : Q \times \Sigma \to Q$ je *přechodová funkce,* která pro daný stav automatu a znak na vstupu řekne, do jakého stavu má automat přejít;
- $q_0 \in Q$ je *počáteční stav,* v němž je automat na počátku výpočtu;
- $F \subset Q$ je množina *přijímacích stavů.*

Výpočte konečného automatu pak probíhá následovně:

1. Nastav aktuální stav s_0 na počáteční stav q_0.
2. Postupně čti znaky $x[i]$ ze vstupu a po přečtení každého přejdi ze stavu s_{i-1} do stavu $s_i = g(s_{i-1}, x[i])$.
3. Pokud skončíš v přijímacím stavu ($s_n \in F$), pak slovo přijmi.

Příklad: Mějme automat nad abecedou $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ se třemi stavy $s_1 \ldots s_3$, počátečním stavem $q_0 = s_1$, jedním přijímacím stavem $F = \{s_2\}$ a přechodovou funkcí g dle tabulky:

$$\begin{array}{ll} g(s_1,\mathtt{0}) = s_3 & g(s_2,\mathtt{1}) = s_3 \\ g(s_1,\mathtt{1}) = s_2 & g(s_3,\mathtt{0}) = s_3 \\ g(s_2,\mathtt{0}) = s_1 & g(s_3,\mathtt{1}) = s_3. \end{array}$$

Tento automat přijímá právě slova ve tvaru $(\mathtt{10})^k$, $k \geq 0$, tedy např. `101010` a prázdné slovo přijme, zatímco `1010101` odmítne.

Konečné automaty docela dobře popisují chod našeho algoritmu – ten také zpracovává text po znacích a přechází podle právě přečteného znaku mezi stavy. Jsou zde ale ještě některé rozdíly: předně KMP neodpovídá ano/ne, ale hlásí jednotlivé výskyty. K tomu můžeme automat upravit například tak, že množinu přijímacích stavů bude používat nejen na konci vstupu, ale v každém kroku. Druhá odlišnost tkví v tom, že přechodová funkce KMP (ta odpovídá prodlužování prefixu o další písmeno) není definována všude. Tam, kde definována není, nastupuje místo ní zpětná funkce, která nás přesouvá mezi stavy tak dlouho, než přechodová funkce definována je.

Tomuto rozšíření se obvykle říká *vyhledávací automat* a definuje se jako pětice $(Q, g, f, q_0, \mathit{out})$, kde:

- Q je konečná množina stavů automatu;
- $g : Q \times \Sigma \to Q$ je *přechodová funkce,* která je definovaná pouze pro některé dvojice (*stav*, *znak*);
- $f : Q \to Q$ je *zpětná funkce,* která říká, do jakého stavu se má automat přesunout, pokud přechodová funkce není definována;
- $q_0 \in Q$ je *počáteční stav,* v němž se automat nachází na začátku výpočtu;
- $\mathit{out} : Q \to \mathcal{P}(\Sigma^*)$ je *výstupní funkce,* která každému stavu přiřazuje, jaký se v něm má ohlásit výstup, což bude množina nalezených slov. (V případě KMP byla vždy buďto prázdná nebo jednoslovná, až budeme za chvíli hledat více slov, bude bohatší.)

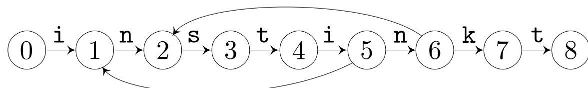
Výpočet vyhledávacího automatu pak probíhá následovně:

1. Nastav aktuální stav s na počáteční stav q_0.
2. Pro každý znak $c = x[i]$ vstupního textu proveď:
3. *Dokud* je $g(s, c)$ nedefinovaná, přejdi zpět do stavu $s \leftarrow f(s)$.

4. Přejdi do nového stavu $s \leftarrow g(s, c)$.
5. Vypiš všechna slova z $out(s)$.

Ještě doplňme, že aby se algoritmus vždy zastavil, musí být $g(q_0, c)$ definováno pro každý znak $c \in \Sigma$, obvykle opět jako q_0.

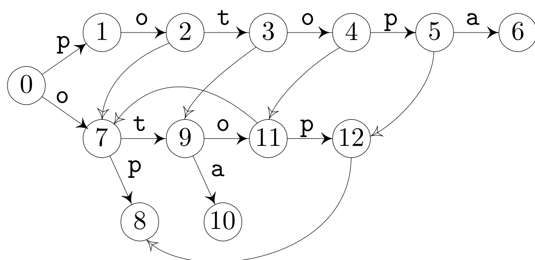
Příklad: Pro slovo `instinkt` by vyhledávací automat vypadal takto (zpětnou funkci jsme kreslili pouze tam, kde nevede do stavu 0):



Nyní algoritmus KMP rozšíříme, aby uměl hledat více slov. Mějme slovník K, což je konečná množina slov nad abecedou Σ, a prohledávaný text β. Vytvoříme vyhledávací automat, jehož výstupem bude výpis nalezených slov a jejich pozic v textu. Jeho stavy budou odpovídat prefixům všech slov ze slovníku a očíslujeme si je přirozenými čísly, počáteční stav $q_0 = 0$ bude odpovídat prázdnému prefixu. Výstupní funkce out pro prefix α ohlásí všechna slova ze slovníku, která jsou suffixem slova α.

Příklad: Jak takový vyhledávací automat může vypadat, si ukážeme pro latinskou abecedu a slovník

$$K = \{\texttt{potopa}, \texttt{op}, \texttt{ota}, \texttt{otop}\}.$$



Rovnými čarami je zobrazena přechodová funkce, kroucenými zpětná funkce. Nejsou zakresleny šipky do 0 u přechodové ani u zpětné funkce. Výstupní funkce je dána následující tabulkou:

$out(5) = \{\texttt{otop}, \texttt{op}\}$ $\quad out(10) = \{\texttt{ota}\}$
$out(6) = \{\texttt{potopa}\}$ $\quad out(12) = \{\texttt{otop}, \texttt{op}\}$
$out(8) = \{\texttt{op}\}$ $\quad out(\text{ostatní}) = \emptyset.$

Vyhledávání pomocí tohoto automatu bude probíhat stejně jako u KMP, $r[i]$ opět bude nejdelší stav, na který končí právě přečtená část textu, složitost vyhledávání bude opět $\mathcal{O}(n)$ až na vypisování výskytů, které poběží v čase $\mathcal{O}(\textit{počet výskytů})$, což může být více než lineárně, ale lépe to určitě nejde.

Pro pořádek dokážeme, že automat doopravdy vyhledává všechny výskyty: (i) Každé slovo, které oznámíme jako nalezené, se v textu opravdu vyskytuje ($r[i]$ se v textu vyskytuje podle své definice a všechna oznámená slova jsou suffixy $r[i]$). (ii) Všechny výskyty opravdu oznámíme. Pokud se na pozici i vyskytuje slovo $\alpha \in K$, pak je zajisté α jedním ze stavů, na něž $\beta[1 \dots K]$ končí a $r[i]$ musí být buďto tento stav nebo nějaký ještě delší, jehož je α suffixem.

Teď se podíváme na to, jak vyhledávací automat pro daný slovník sestrojit. Provedeme to ve dvou krocích. Nejprve sestrojíme množinu stavů Q, přechodovou funkci g a částečnou výstupní funkci o. Ve druhém kroku vytvoříme zpětnou funkci f a rozšíříme o na výstupní funkci out.

V prvním kroku založíme počáteční stav 0, postupně projdeme celý slovník K a každé slovo σ ze slovníku do automatu přidáme. To provedeme tak, že začneme ve stavu 0 a pustíme automat na σ. Jakmile ale v některém stavu s pro znak $\sigma[i]$ nebude přechodová funkce definována, přidáme nový stav q, nastavíme přechodovou funkci $g(s, \sigma[i]) = q$, přejdeme do stavu q a pokračujeme. Tím v lineárním čase vytvoříme strom stavů. Pokaždé, když dojdeme na konec slova, nastavíme také částečnou výstupní funkci $o(q)$ na $\{\sigma\}$.

Popíšeme tuto část formálně:

1. Začni s množinou stavů $Q \leftarrow \{0\}$.
2. Pro každé slovo σ ze slovníku K proveď kroky 3–7:
3. Nastav aktuální stav s na 0.
4. Pro každé písmeno $\sigma[i]$ slova σ proveď 5–6:
5. Pokud je $g(s, \sigma[i])$ nedefinované, založ nový stav q, nastav $Q \leftarrow Q \cup \{q\}$ a polož $g(s, \sigma[i]) \leftarrow q$.
6. Přejdi do nového stavu: $s \leftarrow g(s, \sigma[i])$.
7. Nadefinuj částečnou výstupní funkci: $o(s) \leftarrow \{\sigma\}$.

Zpětnou funkci vytvoříme podobně jako pro jedno slovo tak, že pustíme ještě nehotový automat na část vyhledávaného slova. Opět chceme využít toho, že je funkce definovaná pro všechna kratší slova. Vezměme si náš příklad. Při přidávání slova `potopa` bychom nastavili $f(1) = 0$, $f(2) = 7$, $f(3) = 9$, ale u druhého `o` bychom chtěli použít zpětnou funkci $f(9)$, která ještě není definovaná. Proto budeme postupovat pro všechna slova ze slovníku současně v pořadí podle rostoucí vzdálenosti od stavu 0.

Ještě vyřešíme výstupní funkci. Označme $\sigma(s)$ slovo, jehož cesta vede do stavu s. Pokud pro stav s platí $f(s) = 0$, znamená to, že neexistuje žádný nevlastní (neprázdný) suffix, který by byl prefixem některého ze slov ve slovníku. Proto v tomto stavu může skončit pouze slovo $\sigma(s)$. Nastavíme $out(s) = o(s)$. Pokud $f(s) \neq 0$ končí v tomto stavu také všechna slova, které jsou suffixem slova $\sigma(s)$. Tehdy je $out(s) = o(s) \cup out(f(s))$.

Opět formálně:

1. Založ frontu F, zatím prázdnou.
2. Nastav $f(0) \leftarrow 0$ a $out(0) \leftarrow \emptyset$.
3. Pro každý znak $c \in \Sigma$ proveď následující krok:
4. Pokud je stav $s \leftarrow g(0, c) \neq 0$ pak nastav $f(s) \leftarrow 0$, $out(s) \leftarrow o(s)$ a zařaď s na konec fronty F.
5. Dokud je nějaký stav ve frontě, prováděj následující:
6. Odeber první stav r z fronty F.
7. Pro každý znak $c \in \Sigma^*$, pokud je $g(r, c) \neq 0$, proveď:
8. Označ $s \leftarrow f(r)$. Dokud $g(s, c) = 0$, zvol $s \leftarrow f(s)$.
9. Nastav $f(s) \leftarrow g(s, c)$.
10. Nastav $out(s) \leftarrow o(s) \cup out(f(s))$.
11. Zařaď s na konec fronty F.

Aby algoritmus fungoval rychle, musíme zvolit šikovnou reprezentaci výstupní funkce. Kdyby si každý stav pamatoval svou vlastní množinu, mohly by tyto množiny dohromady být víc než lineárně velké (zkuste vymyslet příklad slovníku, pro který tomu tak je) a museli bychom se vzdát naděje, že stihneme automat zkonstruovat v lineárním čase. Proto použijeme trik: všimneme si, že $out(s)$ je pro každý stav buďto rovna $out(f(s))$ nebo se od ní liší přidáním slova $o(s)$. Stačí si proto pamatovat $o(s)$ a ještě nějakou funkci $z(s)$, která řekne, ve kterém stavu máme najít zbytek množiny $out(s)$. Krok 9 proto upravíme takto:

9. Pokud je $o(f(s)) = \emptyset$, polož $z(s) \leftarrow z(f(s))$, jinak $z(s) \leftarrow f(s)$.

Podobně upravíme vypisování nalezených slov: vypíšeme $o(s)$ a pokud je $z(s) \neq 0$, pokračujeme ve vypisování ve stavu $z(s)$.

Ještě se zamysleme nad časovou složitostí. Označme P velikost celého slovníku. První část algoritmu provede maximálně $\mathcal{O}(P)$ kroků, pokud považujeme velikost abecedy za konstantu. Ve druhé fázi se každý stav dostane do fronty právě jednou, takže vše je lineární až na průchody zpětnou funkcí. Můžeme si ale všimnout, že podobně jako u KMP i zde vlastně spouštíme vyhledávací automat na všechna hledaná slova bez prvního písmene, až na to, že místo jedno po druhém je zpracováváme na přeskáčku a že společné části výpočtů (než se strom rozvětví) počítáme jen jednou. Celkem to tedy bude trvat nejvýše tolik, kolik vyhledání všech slov dohromady, což je $\mathcal{O}(P)$.

Celkové tedy vyhledávací algoritmus běží v čase $O(P+n+v)$, kde n je délka textu, P celková velikost slovníku a v počet nalezených výskytů. Na závěr dodejme, že tento algoritmus vymysleli pan Aho a paní McCorasicková, a předveďme program:

```
var
  N: Integer;                  { délka textu }
  W: Integer;                  { počet slov}
  Slova: array[1..MaxW, 1..MaxK] of Char;
  Delky: array[1..MaxW] of Integer;
  Text: array[1..MaxN] of Char;
  Q: Integer;                  { počet stavů }
  { přechodová a zpětná funkce: }
  g: array[1..MaxQ, Char] of Integer;
  f: array[0..MaxQ] of Integer;
  { obě části výstupní funkce: }
  o: array[0..MaxQ] of Integer;
  z: array[0..MaxQ] of Integer;

  procedure NulujStav(State: Integer);
  var
    C: Char;
  begin
    o[State]:= 0;
    for C:= #0 to #255 do
      g[State, C]:= 0;
  end;

  function Krok(S: Integer; C: Char): Integer;
  begin
    while (S > 0) and (g[S, C] = 0) do S:= f[S];
    Krok:= g[S, C];
  end;

var
  S, I, J, L: Integer;
  C: Char;
  FI, FC: Integer;
  Fronta: array[1..MaxQ] of Integer;
begin
  { vložíme všechna slova }
  Q:= 0;
  NulujStav(Q);
  for I:= 1 to W do
  begin
    S:= 0;
    for J:= 1 to Delky[I] do
    begin
      if g[S, Slova[I, J]] = 0 then
      begin
          Q:= Q + 1;
          NulujStav(Q);
          g[S, Slova[I, J]]:= Q;
      end;
      S:= g[S, Slova[I, J]];
    end;
    o[S]:= I;
  end;

  { zkonstruujeme zpětnou a výstupní fci }
  f[0]:= 0;
  z[0]:= 0;
  FC:= 1;
  FI:= 1;
  Fronta[FC]:= 0;
  while FI <= FC do
  begin
    for C:= #0 to #255 do
    if g[Fronta[FI], C] <> 0 then
    begin
      S:= g[Fronta[FI], C];
      I:= Krok(f[Fronta[FI]], C);
      if Fronta[FI] = 0 then f[S]:= 0
                         else f[S]:= I;
      if o[f[S]] <> 0 then z[S]:= f[S]
                      else z[S]:= z[f[S]];
      FC:= FC + 1;
      Fronta[FC]:= S;
    end;
    FI:= FI + 1;
  end;

  { hledáme }
  S:= 0;
  for I:= 1 to N do
  begin
    S:= Krok(S, Text[I]);
    L:= S;
    while outc[L] <> 0 then
    begin { hlásíme výskyty }
      write(I, ': ');
      for J:= 1 to Delky[o[L]] do
        write(Slova[o[L], J]);
      writeln;
      L:= z[L];
    end;
  end;
end.
```

Dnešní menu Vám servírovali
Martin Mareš a Petr Škoda

Vzorová řešení třetí série osmnáctého ročníku KSP

18-3-1 Trávník

Příprava na mírumilovnou hroší hostinu se zvrhla v divý boj dvou nesmiřitelných zahrádkářských skupin, které se do sebe pustily rýči, hráběmi a jinými zemědělskými stroji. Nedosti na tom, k arzenálu mimo lopatek vytáhli mnohem strašlivější zbraně sestávající z prapodivně zpotvořených tvrzení z teorie složitosti, nesčetných mýtů o rychlosti dělení a jiných zbraní hromadného ničení.

Zahrádkářská frakce *S iterací na věčné časy* aneb *Dělení jen přes naše mrtvoly* zvolila slogan *Vše je konstantní*. Jejich oponenti, partaj *Divisoři*, nechvalně proslulá jako *Zběsilé vzorečky*, přistoupila na taktiku *Co je vyděleno, je správně*.

Pojďme se podívat na jejich programy:

Obě skupinky si správně povšimly, že součet čísel v matici udává přesně množství centimetrů, o které pažit denně poroste.

Konstantníci se rozhodli posčítat si tedy čísla v matici a přičítat denní přírůstek trávníku tak dlouho, až dosáhnou požadovaného množství. Jak dlouho takový přístup trvá? Inu, představme si tu nejhorší situaci, zahradníkovu noční

můru, totiž že by trávník rostl hrozně pomalu a my bychom čekali na nějaké hodně velké množství trávy. Třeba kdybychom měli trávník 1×1 s hodnotou 1 a chtěli bychom 10^9 trávy (máme velmi pažravé přátele). Nejhůř tedy budeme čekat $\mathcal{O}(K)$ času, kde K je požadované množství. A tady se konstantníci zaradovali, protože K je konstanta, tudíž máme program s konstantní časovou složitostí. Ale K není přeci žádná konstanta, kterou bychom dopředu (při psaní programu znali), nýbrž číslo, které nám přijde na vstupu. Správné je tedy říci, že *program má časovou složitost lineární vzhledem ke* K, tím pádem exponenciální k délce vstupu.

Divisoři si povšimli, že jestliže trávník poroste denně o s a my chceme K trávy, stačí provést triviálně K/s. Přitom velkoryse pominuli, že celočíselné dělení vrací *dolní celou část* a začaly se dít věci. Pro $K = 10$ a $s = 10$ pak program radil čekat 1 den, pro $K = 11$ a $s = 10$ také radil čekat 1 den (plus minus jednička, pokud někdo počítal počáteční den), prostě zmatek. Správný postup je udělat normální (neceločíselné) dělení a poté výsledek zaokrouhlit nahoru (neboli vzít horní celou část). Protože jsme informatici a neradi dělíme reálná čísla, je třeba tuto operaci nějak nasimulovat, třeba takhle (s díky Mirku Klimošovi za pěknou formulaci):

Zřejmě potřebujeme, aby tráva vyrostla ještě o $(K - s)$. Jelikož každý den povyroste o s centimetrů, tak počet dnů, které musíme počkat, je $\lceil (K - s)/s \rceil$. Značka $\lceil\ \rceil$ znamená horní celou část. To lze při celočíselném dělení napsat jako $((K - s - 1)/s) + 1$, což se rovná $(K - 1)/s$. Tohle trvá $\mathcal{O}(N \cdot M)$, neboť musíme napřed posčítat prvky v matici a pak už jen jedno dělení.

Co je tedy rychlejší? Navzdory oblíbenému argumentu "dělení je strašně pomalé" je rychlejší udělat jedno dělení oproti ohromnému množství sčítání, o kterých ani nevíme, kolik jich bude.

Někteří *Divisoři* pak ještě s oblibou tvrdili, že jejich program má konstantní časovou složitost. Načítání vstupu se přece nepočítá a pak už se dělá jen to jedno dělení. Nicméně i kdybychom načítání vstupu nepočítali, což se v mnohých analýzách opravdu dělá, tak stejně při výpočtu součtu prvků matice musíme sáhnout na každý prvek matice. A protože tento výpočet je neoddělitelnou součástí algoritmu, nemůžeme ho oddiskutovat jako součást vstupu. Časová složitost algoritmu je tedy $\mathcal{O}(N \cdot M)$. Paměťová je tentokrát skutečně konstantní, protože nás nikdo nenutí pamatovat si celou matici, ale jenom její součet.

Jana Kravalová

18-3-2 Duel

Hrošík vám velice děkuje za došlá řešení, avšak ať se snažil hrát kteroukoli z vámi zaslaných strategií, vždy s prasátkem remizoval. Naštěstí někteří z vás přišli na to, že hra nemá vyhrávající strategii ani pro jednoho z hráčů, ale zato existuje neprohrávající strategie pro oba hráče. Prasátko také není hloupé, a proto hrošík vždy jen remízuje. Nyní se podíváme, proč tomu tak je.

Čísla 1..9 lze jednoduše uspořádat do magického čtverce 3×3. V magickém čtverci platí, že součty trojic čísel jsou pro každý řádek, každý sloupec a obě diagonály rovny číslu 15 (tedy našemu hledanému číslu). Existuje celkem 8 různých možností, jak rozmístit čísla do čtverce, ale všechny lze vygenerovat z rozmístění na obrázku pomocí zrcadlení a otáčení čtverce o 90°.

2	7	6
9	5	1
4	3	8

Představme si, že obě zvířátka nebudou čísla odebírat, ale místo toho si budou zaškrtávat políčka v magickém čtverci. Jistě již tušíte kam tím mířím. Zvířátka budou vlastně hrát piškvorky na mřížce 3×3 (tzv. Tic Tac Toe). Pokud se některému ze zvířátek podaří vytvořit piškvorku (tj. zaškrtnout 3 políčka v řadě, sloupci nebo na diagonále), odpovídá to situaci, kdy se jim v duelu podařilo ukořistit čísla se součtem 15.

Tím jsme ukázali, že obě hry jsou ekvivalentní. Nyní použijeme všeobecně známé tvrzení, že piškvorky na ploše 3×3 nemají vyhrávající strategii pro žádného z hráčů, a protože jsou obě hry ekvivalentní, bude toto tvrzení platit i pro náš Duel. Důkaz spočívá v rozboru všech možných tahů (s ohledem na symetrii čtverce). Podrobnější informace o tomto problému můžete nalézt třeba na stránce *http://en.wikipedia.org/wiki/Tic_tac_toe*.

Martin „Bobřík“ Kruliš

18-3-3 Vrah

Zadání „obětí“ papírky je vlastně permutace a také, pro naši představu asi vhodnější, orientovaný graf. V něm vede z každého vrcholu (hráče) právě jedna hrana (k oběti) a do každého také právě jedna hrana, takže graf se skládá z několika (*kruz*) kružnic. Potkat se určitě mohou jen ti, co jsou na společné kružnici, kružnice ale není problém po dvou spojovat tak, že si libovolní dva lidé z různých kružnic vymění papírky. Stačilo by nám tedy $kruz - 1$ výměn na spojení do jediné kružnice, tolik ale většinou ani nebude potřeba.

Pokud budeme do (neorientovaného) grafu s kružnicemi postupně přidávat některé *zájmové* (vyjadřující zájem o možnost potkání) hrany (kde přidání hrany mezi kružnice zároveň symbolizuje jejich spojení) s tím, že hranu přidáme jen tehdy, když mezi zájemci ještě nevede cesta (vede-li, už leží na kružnici – původní nebo nově vzniklé), získáme snadno postup spojování, a tedy i počet přehozů, ale za cenu složitosti $\mathcal{O}(KN)$.

Zkusíme to vylepšit: Uvažujme právě popsaný graf s některými zájmovými hranami. Pokud bychom do něj přidali i zájmové hrany, které jsme v předchozím algoritmu vynechali, počet komponent se nezvýší – stačí tedy pracovat s grafem, do kterého přidáme všechny hrany. Nechť má tento graf *komp* komponent. Navíc víme, že původní graf bez zájmových hran měl *kruz* komponent. Protože jedno prohození papírků může snížit počet komponent právě o jedna, hledaný počet prohození je $kruz - komp$. Potřebujeme tedy spočítat obě tyto hodnoty, počty komponent dvou grafů. Zvládneme to pomocí dvou prohledávání do hloubky s časovou složitostí $\mathcal{O}(K + N)$.

V programu využiji malý trik, jak vložit informace o K hranách do pole s velikosti $2K$ jako seznam sousedů: prvky na pozici $1 = s_1 + 1$ až s_2 budou sousedé vrcholu 1, od $s_2 + 1$ do s_3 sousedé vrcholu 2 atd. Nejdříve si spočtu počet sousedů každého vrcholu p_v probráním hran a pak jen nasčítám: $s_1 = 0$, $s_v = -1 + \sum_{i=1}^{v-1} p_i = s_{v-1} + p_{v-1}$, což zvládnu lineárně. Pak projdu hrany znovu a přidávám

sousedy vrcholu v na pozice $s_v + p_v$ a pokaždé snížím p_v o jedna. V programu to ale dělám v jediném poli *poc_hran*.

Navíc mohu původní hrany v kružnicích nechat jen jednosměrně orientované, při prohledávání do hloubky je jedno, jakým směrem kružnice projdeme.

Tomáš Gavenčiak

18-3-4 Pochoutka pro prasátko

Vzhledem k tomu, že nám byly úlohy ukradeny a termín odeslání této úlohy se posouvá na 20. března, se zde nenachází vzorové řešení. To vám pošleme až s řešením čtvrté série. Spolu s ním také pošleme finální verzi výsledkové listiny, která je zatím pouze dočasná.

KSP

18-3-5 Hroší lov

Pomóc! Pomóóóc! Lesem zní vyplašené kvíkání prasátkovo, tak nešťastné, že ani jediné oko ostříleného programátora nezůstává suché a ani jediné srdce neustrnuté. Nezbývá nám tedy, než co nejrychleji navrhnout nějaký alespoň trochu funkční a alespoň trochu efektivní algoritmus a teprve pak se pokoušet o zlepšování.

Rychle si do bločku načrtneme základní značení: les bude mít rozměry $M \times N$, velké H bude znamenat počet velkých hrochů, čas budeme měřit v krocích od nuly a v čase T se nebesa slitují a setmí se. Zvířátka si očíslujeme: 0 bude prasátko, 1 až H hroši; přitom každé zvířátko má svou sadu pravidel pro pohyb, jejich počet si pro i-té zvířátko označíme p_i; konečně P bude celkový počet pravidel, čili $P = p_0 + \ldots + p_H$. V programu takto:

```
typedef struct { int x, y; } xy;  // políčko
int M, N, H, T;         // parametry hry
xy S[MAXH+1];           // počáteční polohy
int NP[MAXH+1];         // počty pravidel
xy Pr[MAXH+1][MAXP];  // pravidla
```

S hrochy v zádech budeme pro každý čas t a každé políčko (x, y) zjišťovat, kteří hroši se tam mohou vyskytovat a zda se tam může vyskytovat prasátko. Tomu budeme říkat *stav lesa v čase t* a budeme ho značit S_t.

```
// typ pro stav; pozor, je to pole,
// takže se předává vždy odkazem
typedef char stav[MAXM][MAXN][MAXH+1];
```

Jednotlivé stavy sestrojíme snadno:

V čase $t = 0$ muže být každý hroch jen na svém počátečním políčku a prasátko jakbysmet. Pokud víme, kde může kdo být v čase t, snadno to spočteme i pro čas $t + 1$: hroch může být na nějakém políčku v čase $t + 1$ právě tehdy, existuje-li políčko, na němž může být v čase t a z něhož se lze na nové políčko dostat nějakým jeho povoleným pohybem. Analogicky pro prasátko, jen prasátku nedovolíme vstupovat na políčka, pro která jsme už zjistili, že na nich může být některý z hrochů.

Až toto všechno spočítáme, rozlišíme následující případy:

- V nějakém čase $t < T$ se prasátko může vyskytovat na políčku, které je venku z lesa. Tehdy prasátko uteče a my vypíšeme cestu, která ho na toto políčko zavedla. To zařídíme třeba tak, že se budeme vracet v čase zpět a vždy zjistíme, ze kterého políčka, na kterém se prasátko mohlo vyskytnout v předchozím kroku, lze doskočit na políčko, kde stojí teď.
- V nějakém čase $t < T$ už neexistuje políčko, na kterém by se mohlo prasátko vyskytovat. Běda, hroší spravedlnost vyhrála!
- V čase T stále existují políčka s potenciálním prasátkem. Tehdy musí jít hroši do postýlek a prasátko vyhrává timeoutem. Najdeme si tedy libovolné takové políčko a stejně jako v prvním případě sestrojíme cestu, kudy se prasátko má vydat.

Tento algoritmus se i snadno naprogramuje: budeme si pamatovat stavy S_t pro $t = 0, \ldots, T$. Přitom $S_t[x, y, z]$ bude nula nebo jednička podle toho, zda na políčku (x, y) může být v čase t zvířátko z (tedy z-tý hroch nebo pro $z = 0$ prasátko). Stav S_0 inicializujeme podle počátečních poloh, načež provedeme T *dopředných kroků,* z nichž každý spočítá z S_t stav S_{t+1}. V každém kroku stačí probrat všechna políčka, na každém všechna zvířátka a všechny jejich pohyby. Jeden krok tedy trvá čas $\mathcal{O}(MN \cdot (p_0 + p_1 + \ldots + p_H)) = \mathcal{O}(MNP)$.

Když objevíme, jak může prasátko uniknout (nejpozději po T krocích), začneme se vracet zpět a hledat konkrétní cestu. To bude probíhat ve *zpětných krocích:* každý takový krok dostane polohu prasátka v čase t a podle této polohy a známého stavu S_{t-1} nalezne polohu v čase $t - 1$. Takto se po nejvýše T zpětných krocích, z nichž každý trvá $\mathcal{O}(MNp_0) = \mathcal{O}(MNP)$, dostaneme do počáteční polohy, a tím je cesta ukončena.

Celkem tedy náš algoritmus doběhne za $\mathcal{O}(MNPT)$ kroků a k zapamatování stavů spotřebuje paměť $\mathcal{O}(MNHT)$. [Náš laskavý čtenář si také jistě všimne, že zapomínáme, že všech stavů není T, ale $T + 1$, a zvířátek $H + 1$ namísto H. Ovšem pro $T > 0$ je $T + 1 = \mathcal{O}(T)$ a případy s $T = 0$ můžeme ošetřit zvláštní výjimkou. V zájmu zachování duševní rovnováhy budeme všelijaké ± 1 přehlížet i nadále.]

Následuje náčrt programu:

```
void start(stav s)
{
  // vyplní počáteční stav
  for (int z=0; z<=H; z++)
    for (int x=0; x<M; x++)
      for (int y=0; y<N; y++)
        s[x][y][z] =
          (x == S[z].x && y == S[z].y);
}

int vpred(stav a, stav b, xy *out)
{
  // provede jeden krok vpřed ze stavu a do b
  // pokud je možné utéci, zapíše do out, kam
  for (int z=0; z<=H; z++)
    for (int x=0; x<M; x++)
      for (int y=0; y<N; y++)
        b[x][y][z] = 0;
  for (int z=0; z<=H; z++)
    for (int x=0; x<M; x++)
      for (int y=0; y<N; y++)
        if (a[x][y][z])
          for (int p=0; p<NP[z]; p++)
          {
            int xx = x + Pr[z][p].x;
            int yy = y + Pr[z][p].y;
            if (xx >= 0 && xx < M &&
                yy >= 0 && yy < N)
              b[xx][yy][z] = 1;
```

```
            else if (!z && out)
              {
                // prasátko může utéci
                out->x = xx, out->y = yy;
                return 1;
              }
          }
  // smaže prasátko z políček s hrochy
  for (int x=0; x<M; x++)
    for (int y=0; y<N; y++)
      for (int z=1; z<=H; z++)
        if (b[x][y][z])
          b[x][y][0] = 0;
  return 0;
}

int najdi_prase(stav s, xy *out)
{
  // zjistí, zda se někde vyskytuje prasátko
  for (out->x=0; out->x<M; out->x++)
    for (out->y=0; out->y<N; out->y++)
      if (s[out->x][out->y][0])
        return 1;
  return 0;
}

void vzad(stav a, xy kam, xy *od)
{
  // provede jeden krok vzad
  for (int x=0; x<M; x++)
    for (int y=0; y<N; y++)
      if (a[x][y][0])
        for (int p=0; p<NP[0]; p++)
          if (x + Pr[0][p].x == kam.x &&
              y + Pr[0][p].y == kam.y)
            od->x = x, od->y = y;
}

void pomoz_prasatku(void)
{
  // hlavní program
  stav S[MAXT+1];
  xy cesta[MAXT+1];
  start(S[0]);
  int t = 1;
  while (t <= T &&
         !vpred(S[t-1], S[t], &cesta[t]))
    t++;
  if (t <= T)
    puts("Prasátko uteklo:");
  else if (najdi_prase(S[T], &cesta[T])) {
      puts("Prasátko běhalo do setmění:");
      t = T;
  }
  else {
      puts("Hroši hodují.");
      return;
  }
  // rekonstrukce cesty
  for (int s=t; s > 0; s--)
    vzad(S[s-1], cesta[s], &cesta[s-1]);
  for (int s=0; s <= t; s++)
    printf("(%d,%d)\n",
           cesta[s].x, cesta[s].y);
}
```

Právě předvedené řešení by sice prasátku v nouzi stačit mohlo, ale zejména s paměťovou náročností se ještě pokusíme něco udělat, jeť obludná.

1. pokus: možných stavů je pro každý les konečně mnoho, takže pokud bude T dostatečně velké, musí se stát, že pro nějaké dva časy t a t' bude $S_t = S_{t'}$. Co víc, stav S_{i+1} je pro každé i jednoznačně určen stavem S_i, pročež musí také stavy v časech $t+1$ a $t'+1$ být stejné a vše se začne opakovat. Tehdy je zbytečné počítat dál a můžeme rozhodnout rovnou. Tak bychom mohli paměťovou složitost omezit na $\mathcal{O}(MNHt')$, zaplatíme za to však zpomalením (zjišťovat, zda už jsme stav někdy viděli, jistě něco stojí) a hlavně se to celé vyplatí jen tehdy, pokud je $T > (MN)^{H+1}$ (tolik je totiž různých možných stavů). Zkusme to raději jinak.

2. pokus: všimneme si, že pro výpočet stavu lesa v čase $t+1$ nám stačí znát pouze stav v čase t. Není tedy zapotřebí pamatovat si všechny stavy v minulosti a postačí nám dvě trojrozměrná pole: jedno pro stav současný, druhé pro minulý. Ale ouha, z těchto polí pak nevyčteme zpáteční cestu! Na tu skutečně potřebujeme informace o možných polohách prasátka ve všech časech, jen polohy hrochů zde už nehrají roli. Takže ke dvěma polím pro hlavní výpočet přidáme ještě možné polohy prasátka pro všechny časy, čímž paměťovou složitost zlepšíme na $\mathcal{O}(MNH + MNT)$ a časovou nepokazíme, je stále $\mathcal{O}(MNPT)$.

3. pokus (trochu zoufalý): pokud použijeme jen dvě trojrozměrná pole z předchozího pokusu, nezjistíme sice celou cestu, ale alespoň z poslední polohy spočteme polohu předposlední. Pak bychom mohli celý výpočet spustit znovu, ale zastavit ho o krok dříve a podle výsledku se dostat k předpředposlední poloze a tak dále. Tím redukujeme paměť na $\mathcal{O}(MNH)$, ale čas zhoršíme T-krát na $\mathcal{O}(MNPT^2)$. To se stěží vyplatí.

4. pokus (vymyslel Peter Perešíni): zkusíme si v průběhu výpočtu zapamatovávat jen *některé* stavy a při zpětném průchodu dopočítávat ty zbývající mezi nimi. Řekněme, že si zapamatujeme jen každý k-tý stav, tj. S_0, S_k, S_{2k}, atd. Při zpětném průchodu si pak vždy z S_{jk} spočítáme $k-1$ dopřednými kroky stavy $S_{jk+1}, S_{jk+2}, \ldots, S_{(j+1)k-1}$.

Celkem si tedy potřebujeme zapamatovat T/k stavů při hlavním výpočtu a k stavů při dopočítávání. Časovou složitost hlavního výpočtu jsme nezhoršili, zpětný chod jsme zpomalili o $(T/k) \cdot (k-1) < T$ dopředných kroků, ale to je méně, než kolik potřebuje hlavní výpočet, takže si tím neuškodíme.

Zbývá si rozmyslet, pro jakou hodnotu k získáme nejlepší paměťovou složitost. Snadno zjistíme, že to bude $k = \sqrt{T}$ (až na konstantu) – pokud volíme menší k, převládá T/k, pokud větší, převládá k. Touto volbou k dostaneme paměťovou složitost $\mathcal{O}(MNH\sqrt{T})$.

4,5. pokus (aneb všechno jde trochu zlepšit): co kdybychom předchozí řešení udělali tříúrovňově: pamatovali bychom si jedno hrubé dělení a až dojde na zpětný chod, tak bychom si toto dělení vždy mezi dvěma časy trochu zjemnili a teprve mezi časy jemnějšího dělení si znovu spočítali všechno? Po troše žonglování s čísly bychom se tak dostali k paměti $\mathcal{O}(MNH\sqrt[3]{T})$ a času horšímu jen v multiplikativní konstantě. Ale proč se zastavovat u tří úrovní?

5. pokus (s díky Pepovi Piherovi a jednomu ne úplně vzorovému řešení z CEOI 2005): Představme si, že už známe stav

lesa S_t v nějakém čase t a chceme se v čase $s > t$ prasátkem dostat na určitou pozici (x, y). Zvolíme si čas u někde mezi s a t (nejlépe v polovině) a $u - t$ dopřednými kroky si spočítáme ze stavu S_t stav S_u. Pak rekurzivně zavoláme tentýž algoritmus pro počáteční čas u a koncovou polohu (x, y) v čase s, čímž se dozvíme, že v čase u máme vyrazit z nějaké pozice (x', y') a po jaké cestě půjdeme. A nakonec ještě jedním rekurzivním voláním zkonstruujeme cestu mezi s a u a vrátíme její počáteční vrchol.

Cestu délky T tak postupně rozkládáme na menší a menší úseky, až se dostaneme k úsekům délky 1, kde stačí provést jediný zpětný krok. Rekurzi tedy můžeme popsat binárním stromem, který v kořeni bude mít úsek délky T, v prvním patře úseky délky $T/2$ až v $\log_2 T$-tém patře úseky délky 1. Zpracovat úsek délky l nás stojí $l/2$ dopředných kroků, tedy čas $\mathcal{O}(MNPl)$, což v součtu přes všechny úseky na jednom patře dá $\mathcal{O}(MNPT)$, a tudíž v součtu přes všechna patra $O(MNPT \log T)$.

Paměť potřebujeme v každém rekurzivním volání pouze na dva stavy (počítáme jich sice více, ale zajímá nás jen ten poslední, takže můžeme průběžně zapomínat), celkem si tedy pamatujeme $\mathcal{O}(\log T)$ stavů zabírajících dohromady prostor $\mathcal{O}(MNH \log T)$.

Toto řešeni tedy dokáže výrazně zlepšit paměťovou složitost za cenu $\log T$-násobného zpomalení. S procedurami pro dopředné a zpětné kroky ho už naprogramujeme snadno:

```
void trick(xy *cesta, int t, int s, stav St)
{
  // rekurzivní fce pro sestrojení cesty
  if (t == s)
    return;
  if (t+1 == s)
    {
      vzad(St, cesta[s], &cesta[t]);
      return;
    }
  // najdeme stav v polovině
  stav S[2];
  memcpy(S[t%2], St, sizeof(stav));
  int u = t;
  while (u < (s+t)/2)
    {
      vpred(S[u%2], S[(u+1)%2], NULL);
      u++;
    }
  // a rekurzivně voláme pro obě části
  trick(cesta, u, s, S[u%2]);
  trick(cesta, t, u, St);
}

void nasel(int t, xy pos)
{
  // sestroj a vypiš cestu
  stav S0;
  xy cesta[t+1];
  start(S0);
  cesta[t] = pos;
  trick(cesta, 0, t, S0);
  for (int s=0; s<=t; s++)
    printf("(%d,%d)\n",
           cesta[s].x, cesta[s].y);
}
```

```
void pomoz_prasatku_levneji(void)
{
  stav S[2];  // střídavě minulý a nynější
  xy pos;
  start(S[0]);
  for (int t=1; t<=T; t++)
    if (vpred(S[(t-1)%2], S[t%2], &pos))
      {
        puts("Prasátko uteklo:");
        nasel(t, pos);
        return;
      }
  if (najdi_prase(S[T%2], &pos))
    {
      puts("Prasátko běhalo do setmění:");
      nasel(T, pos);
    }
  else
    puts("Hyeny hodují.");
}
```

Martin Mareš

18-3-6 Komplikovanější komplikátory

Mazání mrtvého kódu lze řešit několika způsoby, my si popíšeme variantu založenou zcela na dataflow analýze. Mějme nějaké místo m v programu. O proměnné x budeme říkat, že je na místě m živá, pokud může být použita v živém výrazu, tedy pokud existuje v CFG cesta z m k nějakému živému výrazu taková, že na ní není žádné přiřazení do x. Zřejmě stačí umět pro každou proměnnou rozhodnout, kde je živá – přiřazení `assign x` … je živé právě tehdy, pokud je proměnná x živá hned za ním.

Stejně jako při propagaci konstant si pro každou proměnnou x na každé pozici v programu (pro jednoduchost můžete uvažovat skutečně všechny pozice, tj. před a po každém příkazu, ale samozřejmě se stačí omezit jen začátky a konce basic bloků) pamatovat ohodnocení. Na rozdíl od propagace konstant nám budou stačit dvě hodnoty – Ž (hodnota x na daném místě je určitě živá) a M (proměnná x by zde možná mohla být mrtvá). Na začátku hodnoty všech proměnných nastavíme na M. Bude platit, že jakmile proměnná na dané pozici nabyde hodnoty Ž, už se nikdy nezmění – to nám zajistí konečnost algoritmu.

Nyní budeme postupně zpracovávat příkazy programu, a budeme se o nich snažit dokázat, že jsou živé. Může se nám stát, že se k jednomu příkazu budeme muset vrátit vícekrát, proto si budeme udržovat frontu příkazů, které je ještě potřeba zpracovat (na začátku do fronty vložíme všechny příkazy). V každém kroku z fronty odebereme příkaz p a zpracujeme ho tímto způsobem:

1) zjistíme, zda má p nějaké vedlejší efekty, pokud ano, prohlásíme ho za živý
2) pokud p je přiřazení do proměnné x, a x má za p ohodnocení Ž, prohlásíme p za živý
3) pro všechny proměnné, které mají za p ohodnocení Ž (kromě proměnné x, pokud je p přiřazení), nastavíme jejich ohodnocení před p také na Ž.
4) pokud je p živý, ohodnocení všech v něm použitých proměnných před p nastavíme na Ž
5) pokud jsme v některém z předchozích dvou kroků změnili ohodnocení proměnné z M na Ž, přidáme příkaz před p do fronty.

Poslední krok je potřeba mírně modifikovat, pokud p je na začátku basic bloku b. V tomto případě musíme nejprve pro takové proměnné nastavit ohodnocení na Ž na konci všech basic bloků, z nichž vede hrana v CFG do b, a případně přidat do fronty poslední příkazy těchto bloků. Až se situace ustálí (což poznáme tak, že se nám vyprázdní fronta), ohodnocení proměnných přesně popisuje, kde jsou proměnné živé a kde mrtvé. Živé příkazy jsou pak právě ty, o nichž jsme to někdy v průběhu algoritmu prohlásili.

Formálně lze správnost tohoto algoritmu dokázat podobně, jako správnost algoritmu pro propagaci konstant. Nám by mohlo stačit následující intuitivní zdůvodnění: Když nějaký příkaz prohlásíme za živý, také za živé těsně před ním prohlásíme proměnné v něm použité. „Živost" těchto proměnných se pak šíří zpět po CFG, dokud nenarazí na přiřazení, které je definuje. Toto přiřazení pak prohlásíme za živé a celý postup opakujeme – časem se takto musíme dostat ke každému živému přiřazení.

Nyní ještě určíme časovou a paměťovou složitost tohoto algoritmu. Ohodnocení každé proměnné se na každé pozici změní nejvýše jednou, tedy každý příkaz budeme zpracovávat nejvýše $V\times$, kde V je počet proměnných v programu. Z trochou šikovnosti lze celý výše popsaný postup provést v čase $\mathcal{O}(NV)$, kde N je velikost programu (včetně jeho CFG). Paměťová složitost je také $\mathcal{O}(NV)$, protože si pro každou proměnnou všude pamatujeme, zda je živá či mrtvá.

Zdeněk Dvořák

Úloha 18-3-1 – Trávník – program

```
program Travnik;

var
        n, m : integer;                         { rozměry matice }
        k : integer;                            { požadované množství trávy }
        s : integer;                            { součet prvků v matici }
        c : integer;                            { načtené číslo }
        i : integer;                            { čítač }
begin
        writeln('Zadejte n m k:');
        readln(n,m,k);
        writeln('Zadejte matici: ');
        s := 0;
        for i := 1 to n*m do begin
                read(c);
                s := s + c;
        end;
        if s = 0 then writeln('Z takového trávníku nikoho nepohostíš.')
        else writeln('Na hostinu je potřeba počkat ještě ',(k-1) div s,' dní.');
end.
```

Úloha 18-3-3 – Vrah – program

```
const maxN=1000;
const maxK=1000;

var
  {zadání}
  K,N:integer;
  obeti:array [1..maxN] of integer;
  hrany:array [1..maxK,1..2] of integer;
  {pro průchody}
  byl:array [1..maxN] of boolean;             {značka pro DFS}
  sousedi:array [1..(2*maxK)] of integer;     {tabulka sousedů}
  poc_hran:array [1..maxN] of integer;        {počet zájmových hran z vrcholu}

procedure projdi(v:integer;sou:boolean);
var i:integer;
begin
  if byl[v] then exit;                        {byl jsem tu už?}
  byl[v]:=true;
  projdi(obeti[v],sou);                       {projdi se po kružnici...}
  if not sou then exit;                       {mám jít i podle pole sousedi?}
  for i:=poc_hran[v]+1 to poc_hran[v+1] do
    projdi(sousedi[i],sou);                   {rekurze...}
end;

var
  i,a,b:integer;
```

```
  kruznic,komponent:integer;
begin
  kruznic:=0; komponent:=0;

  read(N);                                  {nejprve načtu oběti (kružnice v grafu)}
  for i:=1 to N do begin
    read(a);
    obeti[i]:=a;                            {načtu oběti}
    byl[i]:=false;                          {...a zároveň inicializuji}
    poc_hran[i]:=0;
  end;

  for i:=1 to N do                          {projdu kružnice DFS}
    if not byl[i] then begin
      projdi(i,false);                      {projdi bez pole sousedi}
      inc(kruznic); end;
  for i:=1 to N do byl[i]:=false;           {uklid}

  read(K);                                  {a nyní se zájmovými hranami:}
  for i:=1 to K do begin
    read(a,b);                              {načtu další hrany}
    hrany[i,1]:=a; inc(poc_hran[a]);
    hrany[i,2]:=b; inc(poc_hran[b]);
  end;
  {z počtu hran vyrobím nasčítáním indexy do velkého pole sousedi}
  {to obsahuje před poc_hran[i] dost místa na všechny hrany z a do i}
  for i:=2 to N do
    inc(poc_hran[i],poc_hran[i-1]);
  poc_hran[N+1]:=poc_hran[N];
  for i:=1 to K do begin
    a:=hrany[i,1]; b:=hrany[i,2];
    {zapíšu souseda na volné místo do sousedi[] a posunu ukazovadlo}
    sousedi[poc_hran[a]]:=b; dec(poc_hran[a]);
    sousedi[poc_hran[b]]:=a; dec(poc_hran[b]);
  end;

  for i:=1 to N do                          {projdu kružnice i zájmové hrany DFS}
    if not byl[i] then begin
      projdi(i,true);                       {nyní i s polem sousedi}
      inc(komponent); end;
  writeln('Je potřeba ',kruznic-komponent,' přehození.');
end.
```

Dočasná výsledková listina osmnáctého ročníku KSP po třetí sérii

		škola	*ročník*	*sérií*	*1831*	*1832*	*1833*	*1835*	*1836*	*suma*	*celkem*
1.	Peter Perešíni	GJGTajov	4	11		5	10	14		29,0	119,4
2.	Josef Pihera	G Strakon	3	8			9	15	10	34,3	114,6
3.	Pavel Klavík	G Chrudim	3	12	5		10	12	10	36,9	107,5
4.	Miroslav Klimoš	G Bílovec	1	12	4		9	13		25,8	100,7
5.	Jakub Kaplan	GJKTyla	2	8	5		6	13		25,0	89,6
6.	Roman Smrž	GOhradní	2	8	4		10			14,3	88,1
7.	Zbyněk Konečný	GKptJaroš	3	10	5	5	10			20,0	83,3
8.	Jiří Maršík	GJKTyla	2	3	5		5	4		19,7	77,7
9.	Lukáš Lánský	GJKTyla	2	8	3		3	9		17,5	77,0
10.	Michal Pavelčík	G UBrod	3	6			8			8,9	71,9
11.	Petr Onderka	G VKlobou	3	3	5		3			10,4	69,2
12.	Michal Čudrnák	G Holešov	4	2						0,0	64,1
13. – 14.	Petr Kratochvíl	G SvětláNS	3	12	4	5	10			18,9	62,5
	Tomáš Zámečník	GJKeplera	3	3	4		3			10,1	62,5
15.	Michal Vaner	G Turnov	4	4	5					5,0	55,7
16.	Adam Zivner	G UBrod	4	8			5			6,0	54,0
17.	Jan Kohout	G Roudnice	3	3	4	2		8		18,8	48,8
18.	Kristýna Krejčová	G Tišnov	3	2	5		4	2		16,3	48,4
19.	Tomáš Herceg	G Třebíč	3	9	4			10		14,8	48,0
20.	Josef Špak	GJírovco	3	4	4	1	3			11,6	41,5
21.	Radim Pechal	SPŠ Rožnov	3	2						0,0	39,9
22.	Jan Hrnčíř	GFXŠaldy	4	12	5		7			11,7	39,3
23.	Daniel Marek	GZborov	4	6						0,0	38,0
24.	Pavel Veselý	G Strakon	1	2						0,0	37,4
25.	Kateřina Böhmová	G Rožnov	4	2						0,0	36,2
26.	Cyril Hrubiš	G Bílovec	4	9	3	2				5,6	36,0
27.	Jiří Machálek	G Holešov	4	4	5					5,0	35,5
28.	Drahoslav Viktorýn	G UBrod	3	2	4	1				6,8	34,3
29.	Ondřej Bílka	G Zlín	4	11	1	0	8	7		16,0	32,7
30. – 31.	Richard Jedlička	G Vlašim	2	3	3					4,1	31,0
	Tereza Klimošová	G Lanškr	4	3						0,0	31,0
32.	Jakub Pavlík jn.	G Kladno	3	3	3					4,1	28,1
33.	Vojtěch Molda	G Vsetín	4	1						0,0	26,9
34.	Ondrej Mikuláš	G Lučenec	3	2	5	5	6			18,4	25,7
35.	Ondřej Bouda	GKptJaroš	3	4	5		5			12,2	25,3
36.	Petr Trňák	G UHradi	3	3	2					3,3	24,3
37.	Ján Mikuláš	G Lučenec	4	1						0,0	21,7
38. – 39.	Martin Kahoun	GJNerudy	3	4	4			1		6,7	17,9
	Adam Ráž	GBudějo	3	5	5	5				10,0	17,9
40.	Jiří Cabal	SPŠ DvKrál	3	2						0,0	15,1
41.	Rudolf Rosa	G Kladno	3	2						0,0	15,0
42.	Matej Kollár	G PBystric	4	2	4					4,7	14,8
43.	Lukáš Moravec	GSRandyJN	2	1						0,0	12,7
44.	David Škorvaga	G Kralupy	3	1						0,0	12,4
45.	Radim Cajzl	G NMnMor	0	3	4					4,7	10,4
46.	Martin Majer	SPŠÚžlabin	1	1	3		3			10,3	10,3
47.	Tomáš Ehrlich	G Holešov	3	3						0,0	9,8
48.	Tomáš Sýkora	G VKlobou	2	2	2	1				5,5	9,0
49.	Marián Bazálik	G Košice	4	1						0,0	8,3
50.	Jan Musílek	G NBydžov	2	1						0,0	7,3
51.	Jan Tichý	GDašická	1	1						0,0	6,6
52.	Jiří Václavík	G Dobříš	4	2	3					4,2	6,5
53.	Vladimír Munzar	SPŠ Rožnov	1	1						0,0	5,8
54. – 58.	Jakub Balhar	GJNerudy	3	1						0,0	4,7
	Martin Fojtík	GSRandyJN	2	1						0,0	4,7
	Jan Krajdl	SPŠÚžlabin	1	1	4					4,7	4,7
	Dušan Rychnovský	G Hranice	2	1	4					4,7	4,7
	Radek Svoboda	G Roudnice	3	1	4					4,7	4,7
59.	Robert Brunetto	SPŠMasaryk	2	1	3					4,3	4,3
60. – 61.	Miroslav Jančařík	G UBrod	2	1						0,0	3,5
	Jakub Loucký	G Písek	3	1						0,0	3,5
62.	Jiří Keresteš	ZŠKostelní	0	1						0,0	0,0