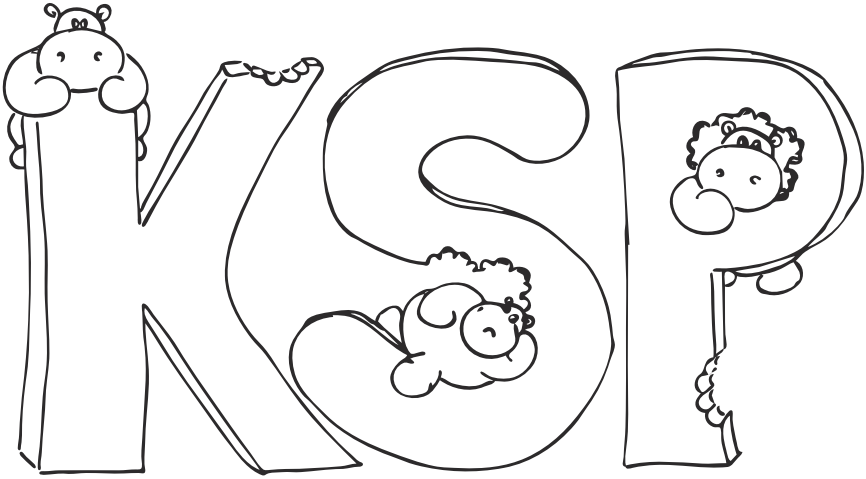
MILAN STRAKA A KOLEKTIV

Korespondenční seminář z programování

XVIII. ročník – 2005/2006



matfyzpress

VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

MILAN STRAKA A KOLEKTIV

Korespondenční seminář z programování

XVIII. ročník – 2005/2006

matfyzpress

Praha 2006

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji!

ISBN 80-86732-91-6

Úvod

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

KSP probíhá tak, že student od nás jednou za čas dostane poštou zadání obvykle šesti úloh, v klidu domácího krbu je (ne nutně všechny, počítají se nejlepší čtyři) vyřeší, svá řešení v přiměřeně vzhledné podobě sepíše a do určeného termínu zašle na naší adresu (ať už fyzickou či elektronickou). My je poté opravíme a spolu se vzorovými řešeními a výsledkovou listinou pošleme při vhodné příležitosti zpět na adresu studenta. Tento cyklus se nazývá *série,* resp. kolo.

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

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

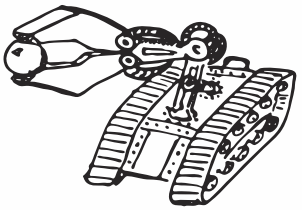
Velice rádi vám odpovíme na libovolné dotazy jak ohledně studia informatiky na naší fakultě, tak i stran jakýchkoliv informatických či programátorských problémů. Jakýkoliv problém, jakákoliv iniciativa či nabídka ke spolupráci je vítána na adrese:

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

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

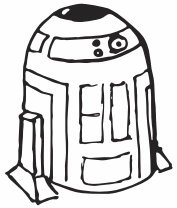
Zadání úloh

18-1-1 Dimenze X — 5 body



Vědci z ústavu teoretické fyziky MFF UK si usmysleli, že zachrání svět a jednou provždy zatočí s tajemnou a nebezpečnou Dimenzí X. Sestrojili přístupový portál a vyslali do něj dva roboty, z nichž každý nese slušný náklad plutonia. Jistě si sami dovedete představit za jakým účelem a sami tušíte, že oba kusy plutonia je potřeba dát na místě k sobě.

Jaké však bylo překvapení vědců, když po odeslání robotů zjistili, že Dimenze X je pouze jednorozměrná, tedy obyčejná přímka, navíc táhnoucí se prokazatelně od severu k jihu. Do výpočtů se navíc vloudila chybička a roboti dopadli na naprosto neznámé místo na přímce, každý někam jinam. Na obou místech přistání zůstala po robotech hromada šrotu, která z nich nárazem odpadla, mimo jiné komunikační a senzorová jednotka. Paměťový obvod se také částečně poškodil, kompas naštěstí vydržel. To znamená, že roboti se spolu nemůžou nijak na dálku domluvit a jejich setkání se tak značně komplikuje. Ale Vy to přeci tak nenecháte!



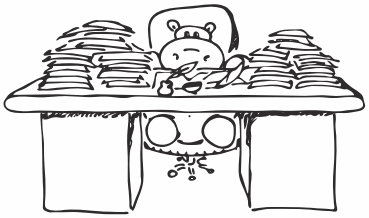
V každém kroku se robot může posunout buďto o 1 metr na sever nebo o 1 metr na jih. Robot naštěstí pozná, že právě přechází přes hromadu šrotu (ale už nepozná, jestli přes svou, nebo druhého robota). Kromě nákladu plutonia (který pochopitelně nesmí zahodit) už neunese vůbec nic dalšího. Paměťový obvod je poškozený, takže čím méně si toho bude robot muset pamatovat, tím lépe. Navrhněte nouzovou posloupnost povelů – tedy vlastně algoritmus (programovat ho ale nemusíte) – pro roboty takovou, aby do sebe roboti na přímce zaručeně po nějaké době narazili. Ale pozor, oba dva roboti dostanou jednu a tutéž posloupnost povelů.

18-1-2 Úřad — 6 bodů

Bylo nebylo, na jistém nejmenovaném úřadě jistého nejmenovaného města v jisté nejmenované republice byli velmi líní úředníci. Lidé přicházeli se svými žádostmi, přiznáními, potvrzeními, stížnostmi a dalšími písemnostmi za úředníky, kteří s otráveným výrazem vyslechli, co od nich kdo chce. Potom se slovy „Náš úřad to přezkoumá“ spis založili do šanonu, aby ho z něj už nevytáhli až do skonání světa. Ale to jim nestačilo a po čase začali zařazovat i samotné

šanony a pořadače do jiných šanonů, pořadačů, fasciklů, s deskami různých barev, a takové objemné svazky potom ukládali do police.

Jednou přišel do úřadu kontrolor. Ne ovšem proto, aby zjistil, jestli úředníci dobře vyřizují žádosti, nýbrž jestli správně archivují spisy. Stoupnul si k polici se spisy a ze zadní strany postavil nervózního ředitele, aby zleva doprava četl barvy desek a jestli je to otevírací deska nebo uzavírací. Úkolem kontrolora je zjistit, jestli jsou šanony do sebe dobře zabalené. Očíslujme si jednotlivé barvy a pokud je před číslem minus, je to deska uzavírací, jinak otvírací. Například tohle zjevně nejsou dobře zabalené desky: 4, 5, -4. Stejně tak tyto: 7, 8, -7, 8. Nicméně proti těmto nelze nic namítat: 1, 5, 6, -6, -5, -1, 6, -6.



Pomozte kontrolorovi zdeptat pana ředitele! Napište program, který odpoví, jestli desky jsou či nejsou dobře zabalené. Program dostane na vstupu přirozené číslo K, což je maximální počet barev šanonů, a počet nahlášených desek N (otevíracích i uzavíracích dohromady). Poté následuje N čísel c, $1 \leq |c| \leq K$ udávajících barvy šanonové desky, kladné číslo c značí otevírací desku barvy c, číslo $-c$ uzavírací desku barvy c. Program by měl vypsat ano nebo ne podle toho, jestli jsou desky správně zabalené. Jinými slovy, pokud si různé typy šanonů představíme jako různé typy závorek, pak je Vaším úkolem otestovat, zda je zadaná posloupnost závorek správně uzávorkovaná.

Příklad: Pro $K = 3$, $N = 8$ a desky $2, 1, 2, -2, -1, -2, 3, -3$ by měl Váš program vypsat odpověď *ano*, zatímco pro desky $1, 2, 3, 1, -1, -2, -3, -1$ je správná odpověď *ne*.

18-1-3 Keřík — 10 bodů

Housenka běláska je neuvěřitelně žravý tvor. Když se jednou tak potulovala po zahrádce plné salátu, kedluben, řepy a spousty dalších laskomin, narazila na velmi chutně vypadající keř.

Keř sestává z význačných bodů, na kterých rostou šťavnaté lístky, a větví mezi nimi, kde neroste nic. U každého bodu je zadáno, kolik lístků na něm roste, a seznam jiných význačných bodu, do kterých z něj vede větev. Dále víte, že keř je zkrátka strom, a tedy mezi každými dvěma význačnými body vede právě jedna cesta.

Housenka by ráda začala svou hostinu v nějakém význačném bodu keře a postupně se po větvích přesouvala na jiné body, nikdy však do místa, kde už

jednou byla. A chtěla by se co nejvíce najíst. Pomozte jí a napište program, který k zadanému keři najde nejvýživnější cestu v něm, tedy cestu mezi nějakými dvěma body, na které se neopakuje žádný význačný bod a která obsahuje největší možný počet lístků.

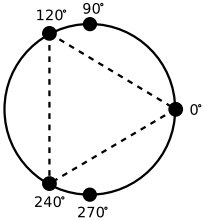
Program dostane na vstupu číslo N, což je počet význačných bodů, a N řádků popisujících jednotlivé vrcholy (očíslujeme si je 1 až N). První číslo na i-tém řádku udává počet lístků, druhé číslo počet větví v vedoucích z i-tého bodu, načež následuje v čísel udávajících, kam ty větve vedou. Program by měl vypsat na výstup nejvýživnější cestu, pokud je takových cest více, tak libovolnou z nich.

Příklad: Pro čtyři význačné body, které obsahují po řadě 1, 2, 3 a 4 lístečky, a pro větvičky $1 \leftrightarrow 2, 2 \leftrightarrow 3, 2 \leftrightarrow 4$ je nejvýživnější cesta $3 \rightarrow 2 \rightarrow 4$ s 9 lístečky.

18-1-4 Dortík — 11 bodů

Klub Sběratelů Pamětin právě slaví své K-té narozeniny. Proto jeho členové vytáhli ze svých nekonečných sbírek narozeninový dort s N svíčkami (jiný zrovna nedokázali najít). Rádi by na něm zapálili právě K svíček a jelikož sběratelé pamětin mají velmi vytříbené estetické cítění, musí tyto svíčky ležet ve vrcholech pravidelného K-úhelníku.

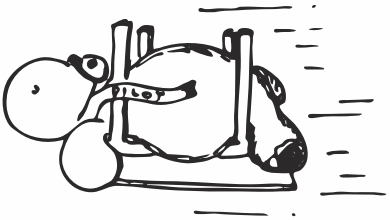
Napište pro ně program, který na vstupu dostane čísla N a K a dále polohy všech N svíček v libovolném uspořádání. Všechny svíčky leží na kružnici, takže jejich polohu můžeme jednoznačně popsat úhlem, který svírá spojnice středu kružnice a svíčky s osou x. Výsledkem programu by měl být nalezený pravidelný svíčkový K-úhelník, případně zpráva, že žádný takový neexistuje.



Příklad: 5 svíček na pozicích 0°, 240°, 270°, 120° a 90° obsahuje pravidelný svíčkový trojúhelník, ale neobsahuje pravidelný svíčkový čtyřúhelník.

18-1-5 Matlalové — 15 bodů

A nyní zprávy ze Země: „Institut SETI konečně potvrdil existenci mimozemského života! Frakce Marťanů a létajících talířů (MALTAL) oznámila zachycení vysílání potvrzující existenci skutečných Marťanů, kteří používají opravdové létající talíře!“ „Ty Matlalové jsou ale natvrdlí, jaké létající talíře? Budeme tam muset zaletět a objasnit to!“



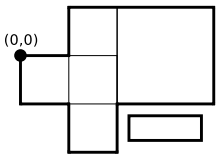
Na Zemi přistáli mimozemšťani přímo na zahradě před frakcí MALTAL. „Marťanové!“ „Matlalové! My vůbec nepoužíváme létající talíře! Dříve jsme létali na létajících kobercích, ale protože se na nich není jak držet, často jsme z nich ve vesmíru padali, a tak nyní používáme létající stoly!“

Na zahradě opravdu stály létající stoly, některé dokonce stály na ostatních (při pohledu shora se překrývaly). Ovšem díky marťanskému způsobu navigace byly hrany všech stolů rovnoběžné se světovými stranami. Protože se už ale seběhl dav, který si chtěl Marťany a jejich dopravní prostředky prohlédnout, rozhodli se Matlalové postavit kolem létajících stolů plot. Protože se ale některé ze stolů překrývají, není vůbec jasné, jak by měl být dlouhý. A protože Vás Matlalové už znají (vědí, jak jste zatočili s Dimenzí X), požádali Vás o pomoc.

Program dostane na vstupu N, počet marťanských létajících stolů, a jejich jednotlivé polohy. Každý marťanský stůl je zadán pomocí svého „levého horního“ rohu (při pohledu shora) a svou délkou a šířkou v metrech, přičemž jednotlivé stoly se mohou překrývat. Vaším úkolem je říci, kolik metrů plotu bude třeba k oplocení území, na kterém marťanské stoly přistály. Plot musí oplotit všechny zadané stoly, musí vést vždy po jejich hranici, ale nikdy nesmí vést uvnitř nějakého létajícího stolu (matematicky řečeno chcete zjistit obvod sjednocení všech létajících stolů). Počítejte s tím, že jak souřadnice, tak délky a šířky marťanských stolů mohou být reálná čísla.

Příklad: Pro $N = 4$ a létající stoly

- levý horní roh $(0, 0)$, délka 20 m a šířka 10 m,
- levý horní roh $(10, 10)$, délka 10 m a šířka 30 m,
- levý horní roh $(20, 10)$, délka 20 m a šířka 20 m,
- levý horní roh $(22.5, -12.5)$, délka 15 m a šířka 5 m,



je hledaná délka plotu 180 m.

18-1-6 Kompilované komplikátory — 11 bodů

V letošním ročníku seriálu si budeme povídat o kompilátorech. Samozřejmě toho nestihneme příliš mnoho (o kompilátorech existuje několik tlustých knih a stovky článků), ale měli bychom získat alespoň obecnou představu o tom, jak kompilátory fungují.

Začněme trochou historie. Před dávnými časy se počítače programovaly přímo ve strojovém kódu – tj. programátor psal (nebo děroval) řady čísel. Napsat takto jakýkoliv rozsáhlejší program bylo samozřejmě velmi obtížné, o hledání a opravování chyb ani nemluvě. Proto se brzy objevily nástroje, které tuto činnost usnadňovaly – assembler, který alespoň umožňoval zapisovat instrukce v čitelnější formě a pojmenovat si proměnné, a později i vyšší programovací jazyky.

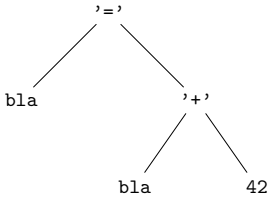
Jedním ze zásadních problémů překladačů programovacích jazyků byla rychlost (tedy spíše pomalost) jimi produkovaného kódu. Nějakou dobu trvalo, než optimalizace, které překladače provádí, dosáhly takové úrovně, aby výsledný kód byl srovnatelný s tím, co napíše programátor v assembleru. Po docela dlouhou dobu strašily v programátorských učebnicích poučky typu „místo $3*x$ pište $x+x+x$“, které měly kompilátoru zabránit, aby použil „drahou“ instrukci na násobení. Dnes již naštěstí takovéto obludnosti nejsou potřeba (a pouze učiní program těžším ke čtení) – dost často jsou přímo v assembleru napsané programy pomalejší než zkompilované.

Většina seriálu bude věnována tomu, jakými optimalizacemi se tohoto výsledku dosáhne. Nicméně nejprve bychom si měli popsat, jak kompilátor vypadá a s čím vlastně pracuje. Níže popsané schéma kompilátoru bylo obecně přijato a v nějaké podobě ho používají zřejmě všechny v současnosti existující kompilátory.

Prvním krokem při kompilaci je lexikální analýza. Jejím úkolem je vstup (což je nějaká posloupnost znaků) převést na posloupnost objektů, které nesou nějaký smysl. Například, pokud je na vstupu řetězec „`bla = bla + 42`“, pro překladač je nezajímavé, že je to znak 'b', následovaný znakem 'l', atd. Výsledek lexikální analýzy by nám v tomto případě řekl, že na vstupu je identifikátor proměnné (`bla`), následovaný operátorem přiřazení, dalším identifikátorem, operátorem sčítání, a číslem (42). Výstupem tedy je posloupnost tzv. *tokenů*. Každý z nich má typ udávající, o jaký objekt se jedná (zda to je identifikátor, číslo, atd.), a případně nějaké další informace (např. u identifikátorů řetězec, udávající jeho jméno, u čísel jejich hodnotu, apod.).

Druhým krokem je syntaktická analýza. Jejím úkolem je rozebrat strukturu programu. Výsledkem syntaktická analýzy výrazu „`bla = bla + 42`“ je to, že se jedná o přiřazení, na jehož levé straně je proměnná a na pravé straně je operátor sčítání aplikovaný na proměnnou a číslo. Povšimněte si, že v syntaktické analýze zpravidla záleží pouze na typech tokenů, nikoliv na další in-

formacích u nich uložených – nezáleží na tom, jak se proměnná jmenuje, nebo jaká je hodnota čísla (i když samozřejmě tyto informace nesmíme ztratit). Výsledek se obvykle reprezentuje jako *syntaktický strom*. Vrcholy tohoto stromu odpovídají tokenům, synové vrcholu pak operandům příslušného výrazu nebo příkazu. Například vrchol obsahující plus bude mít jako syny sčítané výrazy, vrchol obsahující příkaz `if` bude mít jako syny podmínku, `then` větev a `else` větev. Syntaktický strom výrazu „`bla = bla + 42`“ vypadá takto:



Syntaktická analýza se samozřejmě provádí podle syntaxe daného programovacího jazyka. Syntaxe bývá nejčastěji zadána pomocí bezkontextové gramatiky. Co to přesně znamená, se můžete dočíst v loňském seriálu, my si pouze ukážeme příklad syntaxe jednoduchého aritmetického výrazu (obsahujícího sčítání, odčítání, násobení, dělení a závorky), ze které bychom měli získat hrubou představu:

```
výraz     →  výraz '+' výraz_nás
          |  výraz '-' výraz_nás
          |  výraz_nás

výraz_nás →  výraz_nás '*' operand
          |  výraz_nás '/' operand
          |  operand

operand   →  číslo
          |  proměnná
          |  '(' výraz ')'
```

Zde `výraz_nás` je výraz, jehož operátor má alespoň tak vysokou prioritu, jako násobení. Tedy podle této gramatiky například `operand` výrazu je buď číslo, nebo proměnná, nebo uzávorkovaný výraz.

Existují nástroje, kterým se zadá takováto bezkontextová gramatika a automaticky vytvoří kód, který provádí syntaktickou analýzu popsaného jazyka. Druhou variantou je napsat si syntaktickou analýzu „ručně“, což je o něco pracnější, ale snáze se popisuje ošetřování chyb a jiných speciálních případů.

Dalším krokem bývají jednoduché optimalizace (například zjednodušování výrazů), které se snadno provádí na syntaktických stromech. Pro většinu optimalizací je však reprezentace pomocí syntaktických stromů poměrně nevhodná. Například operandy výrazu mohou být libovolně složité a mohou mít různé vedlejší efekty (změny hodnot proměnných, volání libovolných funkcí,

apod.), které činí práci s nimi dost nepohodlnou. Proto se program dále převádí do tzv. *mezikódu*, což bývá jazyk podstatně jednodušší, často podobný assembleru. To, jak přesně mezikód vypadá, se překladač od překladače dost liší, často se v jednom překladači používá i více mezijazyků, které se typicky čím dál tím víc blíží výslednému assembleru. Jednoduchý mezijazyk by mohl vypadat například takto:

Program je posloupnost příkazů. Příkazy jsou následující:

- `assign` *proměnná výraz* – přiřadí do proměnné hodnotu výrazu. Výraz musí být jednoduchý, tedy musí to být proměnná, číslo, nebo nějaká aritmetická operace, jejíž operandy jsou buď proměnné nebo čísla. Takže *výraz* může být třeba x, 42, $x + y$, ale už $x + y + 1$ je příliš složité.
- `label` *jméno* – označuje label, na který se dá skákat.
- `goto` *jméno* – skočí na label se jménem *jméno*.
- `if` *podmínka* $jméno_1$ $jméno_2$ – pokud je podmínka pravdivá, skočí na label se jménem $jméno_1$, jinak na label se jménem $jméno_2$. Podmínka musí být jednoduchá, tj. pouze porovnání dvou proměnných nebo čísel.

Například kousek programu v Pascalu

```
sum := 0;
for i := 1 to 10 do
  sum := sum + 2 * i;
```

se do mezikódu přeloží jako

```
assign sum 0
assign i 1
label loopbeg
assign tmp (2 * i)
assign sum (sum + tmp)
assign i (i + 1)
if (i <= 10) loopbeg loopend
label loopend
```

Nad takovýmto mezikódem se provádí většina optimalizací. Povšimněte si, že v mezikódu nezáleží na tom, jaký jazyk vlastně překládáme – stejný mezikód bychom dostali při překladu následujícího programu v C:

```
sum = 0;
for (i = 1; i <= 10; i++)
  sum += 2 * i;
```

Všechny optimalizace nad mezikódem tedy můžeme sdílet mezi překladači různých programovacích jazyků. Část překladače, která závisí na použitém programovacím jazyce a která končí typicky překladem do mezikódu, se nazývá

front-end. Část, v níž se provádí optimalizace víceméně nezávislé jak na překládaném jazyce, tak na assembleru, do nějž program překládáme, se nazývá *middle-end*.

Poslední částí překladače je *back-end*. Tato část přepisuje program z mezikódu do assembleru, a je tedy závislá na architektuře, pro kterou je výsledný kód určen (jiný back-end se používá pro počítače založené na procesorech typu x86, které asi všichni znáte, jiný pro další, „exotičtější" procesory). V back-endu se často provádí optimalizace specifické pro danou architekturu, například scheduling (přerovnávání instrukcí tak, aby se daly provádět paralelně), přiřazování proměnných do registrů a další.

Překladač se tedy skládá z jednoho či více front-endů pro různé programovací jazyky, middle-endu, a jednoho či více back-endů pro různé architektury. Samozřejmě ve skutečnosti rozdělení nebývá takto jasné. I v middle-endu musíme například znát časy provádění instrukcí, abychom mohli dobře optimalizovat, a tedy middle-end musí něco vědět o cílových architekturách. Pokud je dobře navržen popis architektury, lze také často sdílet některé části back-endů, a některé optimalizace se tak přesouvají spíše do middle endu.

Úloha:

Abychom si pouze nepovídali, zkusíme si v této sérii prakticky napsat jednoduchý front-end k překladači. Abychom se vyhnuli komplikacím, měl by umět pouze překládat výrazy popsané gramatikou uvedenou v textu seriálu, do popsaného mezikódu. Výsledek výrazu by měl být přiřazen do proměnné `result`. Například výraz

```
x * (x + y) - z / bla / neco * 5
```

by měl být přeložen jako

```
assign tmp1 (x + y)
assign tmp2 (x * tmp1)
assign tmp3 (z / bla)
assign tmp4 (tmp3 / neco)
assign tmp5 (tmp4 * 5)
assign result (tmp2 - tmp5)
```

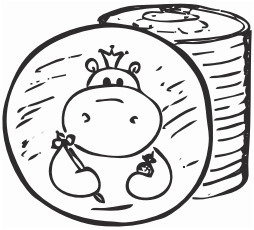
18-2-1 Potrhlý syslík — 6 bodů

Syslík Potrhlík dostal jednoho dne podle svého soudu přímo úžasný nápad, jak zbohatnout – bude chovat krokodýly a všechna ostatní zvířátka se na ně budou chodit dívat. A protože to byl velký Potrhlík, hned běžel za žabákem Skrblikvákem, který vlastnil nedalekou bažinu, a když doběhl, udýchaně volal: „*Krokoběh!* Půjč mi bažinu, postavíme *krokoběh* a budeme bohatí!"

„Cože, jaký kroko-běh? Ty chceš běhat nebo krokovat? Co to blábolíš za nesmysly? Ty už ses musel úplně zbláznit!" „Ale nezbláznil, já ti to vysvět-

lím.“ „Kdepak, s takovým potrhlým syslem se nebudu bavit.“ „Tak se na něco zeptej, ať víš, že jsem se nezbláznil, a já ti to pak vysvětlím.“

Skrblikvák se tedy zahloubal a vymyslel pro Potrhlíka následující zkoušku: Vzal Potrhlíka do zatemněné místnosti, ve které je na podlaze 50 mincí, z toho 18 leží nahoru lícem a zbytek rubem. Potrhlík je má za úkol rozdělit na dvě (ne nutně stejně velké) části tak, aby počet mincí ležících nahoru lícem byl v obou těchto částech stejný.



Potrhlík může kterékoliv mince libovolně otáčet, ale protože je tma a mince staré, nijak nemůže zjistit, které mince leží nahoru lícem a které rubem. Když tedy nějakou minci otáčí, sám neví, jestli bude rubem nebo lícem, ví jenom, že ji otočil. Sám si ale moc neví rady – pomůžete mu?

Vaším úkolem je vymyslet pro Potrhlíka *postup*, který vždy dokáže mince rozdělit na dvě části tak, aby počet mincí ležící lícem vzhůru byl v obou částech stejný. (Pro počet mincí ležících nahoru rubem to už platit nemusí.)

18-2-2 Kvakulátor — 5 bodů

„No, když jsi ty mince tak pěkně rozdělil, nebudeš úplně blázen, Potrhlíku. Co to tedy je ten kroko-běh?“ „Krokoběh je přece **kroko**dýlí vý**běh**! Postavíme ho v bažině a všichni se na něj budou chodit dívat.“ „Hm, špatný nápad to není. Ale budeme na to potřebovat určitě hodně peněz. Musíme koupit krokodýly a krokodýlí žrádlo, musíme výběh postavit, uplatit stavební komisi Bažinové Unie, koupit stánek se zmrzlinou, . . . “

Když si Skrblikvák sepsal, co všechno budou muset s Potrhlíkem zaplatit, zajímala ho celková výše jejich výdajů. Protože ale neměl při ruce svůj kvakulátor, poprosil Vás o něj.

Skrblikvák se snaží zapsat hodnotu zlomku a/b jako desetinné číslo. Zjistil, že někdy může být tento desetinný zápis nekonečný ($1/3 = 0.3333\ldots$), ale vždy, když se to stane, nějaká část čísla se pak musí opakovat ($1/3 = 0.\overline{3}$). Tato opakující část čísla se nazývá *perioda*.

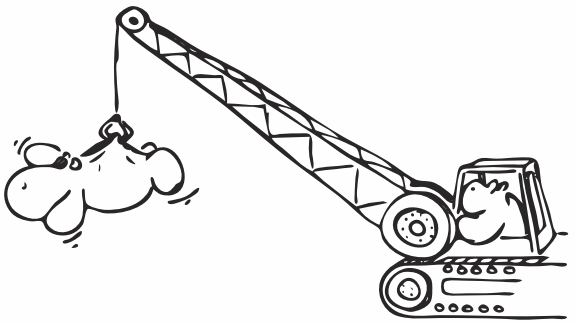
Skrblikvák by po vás chtěl program, který mu pro zadané celočíselná a a b řekne, jak je dlouhá perioda desetinného zápisu čísla a/b. Pokud zápis čísla a/b periodický není, je délka periody evidentně nula. Můžete počítat s tím, že čísla a a b se vejdou do `longintu`.

Příklad: Délka periody čísla 1/3 je jedna, u čísla 1/8 je nula a perioda čísla 123/456 má délku 18.

Bonus: Pokud váš program bude potřebovat jenom konstantně mnoho pomocných proměnných (žádné pole délky závislé na a a b) a nezhorší-li se tím jeho časová složitost, dostanete bonus až +3 body.

18-2-3 Jeřábník Evžen — 15 bodů

Poté, co se Skrblikvák a Potrhlík dohodli, začali hned stavět. „Budeme potřebovat jeřábníka. Nevěděl bys o někom?“ „A co Evžen, ten pelikán? Často stojí v bažině na svých vysokých nohách. Ovládat vysoký jeřáb pro něj bude hračka.“

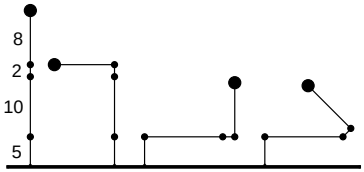


Brzy se ale ukázalo, že demoliční jeřáb je i na vysokého pelikána trochu moc komplikovaný (možná proto, že ovládat páky zobákem nebo volant křídly moc dobře nejde). Protože určitě nechcete, aby Evžen zdemoloval celou bažinu (nebo sebe v kabině), měli byste mu pomoci.

Jeřáb si představte jako N segmentů, každý je libovolné délky. První segment je zapuštěný v zemi a stojí stále vzhůru a mezi každými dvěma sousedními segmenty je ohebný kloub. Na konci posledního segmentu je těžká ocelová koule (je to *demoliční* jeřáb). Na začátku je jeřáb celý vzpřímený. V každém kroku si Evžen vybere nějaký kloub 1 až $N-1$ a ten otočí o zadaný počet stupňů. Zajímá ho, kam přesně se po každém kroku ocelová koule dostane.

Napište program, který dostane N (počet segmentů jeřábu), dále $l_1, \ldots, l_N$ (délky segmentů 1 až N) a M (počet otočení, které Evžen s jeřábem udělá). Následuje M dvojic (k_i, u_i), každá z nich znamená otočit kloub číslo k_i (je mezi segmenty k_i a $k_i + 1$) o u_i stupňů ve směru hodinových ručiček. Navíc *ihned* po načtení každé dvojice (k_i, u_i) musí váš program s přesností na dvě desetinná vypsat, kam se ocelová koule (konec jeřábu) přesune. Na začátku je celý jeřáb vzpřímený a bod $(0, 0)$ je spodek jeřábu.

Příklad: Jeřáb má 4 segmenty délky 5, 10, 2 a 8, $M = 3$.



Po provedení operace $(3, -90°)$ se dostane koule na souřadnice $[-8.00; 17.00]$, po $(1, 90°)$ na $[12.00; 13.00]$ a po provedení $(2, -45°)$ na $[5.76; 12.07]$.

18-2-4 Stavbyvedoucí — 9 bodů

Hned, jak byla příslušná část bažiny Evženem zdemolována (ať už podle plánu nebo ne), mohla začít stavba krokoběhu. Stavbyvedoucím se stal Potrhlík a všichni ostatní zvířecí stavitelé si u něj mohli objednávat materiál.

Když si konečně všichni nadiktovali, co chtěli, měl už Potrhlík pěkně dlouhý seznam. U každého předmětu ze seznamu si Potrhlík pamatuje N údajů a každý předmět má zapsaný v seznamu na jedné řádce. Celý seznam je tedy tabulka, která má N sloupců a tolik řádek, kolik je předmětů.

Všichni stavitelé si ovšem (stejně jako učitelé) myslí, že jejich předměty jsou ty nejdůležitější, a tak každý chce, aby byly všechny řádky tabulky setříděny podle jejich požadavku. Každý požadavek je číslo údaje (sloupce), podle kterého by se měly všechny řádky setřídit. (Neboli je to číslo sloupce, podle kterého bychom měli setřídit celou tabulku.)

Chudák Potrhlík nakonec obdržel M požadavků, tedy M žádostí o setřídění dle určitého sloupce. Rozhodl se, že řádky setřídí nejprve podle 1. požadavku, potom podle 2., ..., až M-tého požadavku. Navíc když budou v nějakém kroku dvě řádky podle zpracovávaného požadavku stejné, jejich vzájemné pořadí zůstane stejné jako před tímto tříděním.

Počet třídících požadavků je ale opravdu velký a často se v něm opakují čísla sloupců, takže Potrhlíka napadlo, že byste mu mohli pomoci jeho úkol zjednodušit. Zajímalo by ho, jestli by nemohl provést setřídění řádků podle menšího počtu třídících požadavků. Tato kratší posloupnost třídících požadavků by měla být s původní posloupností *ekvivalentní*, čili ať je seznam předmětů na začátku uspořádán libovolně, setřídění podle původní posloupnosti požadavků a podle kratší posloupnosti požadavků musí dát vždy stejné výsledky (stejně setříděný seznam).

Zkuste napsat program, který dostane na vstupu N (počet sloupců seznamu), M (počet třídících požadavků) a jednotlivé třídící požadavky a najde nejkratší posloupnost třídících požadavků, která je zadané posloupnosti ekvivalentní. Pokud je minimálních posloupností více, stačí vypsat libovolnou z nich.

Příklad: Pro $N = 3$ a $M = 7$ požadavků 3, 3, 1, 1, 2, 3, 3 je hledaná nejkratší posloupnost požadavků třeba 1, 2, 3.

18-2-5 Krokoběh — 11 bodů

Jakmile byl potřebný materiál nakoupen a staviteli rozebrán (takzvaně rozkraden), mohlo se začít se stavbou krokoběhu. Krokoběh se skládá z několika jezírek, ve kterých mohou krokodýli odpočívat, a kanálů mezi nimi. Kanály jsou obousměrné, vedou vždy mezi dvěma jezírky a žádné dva kanály se mimo jezírka neprotínají (mimoúrovňově ale mohou).

Nějaká jezírka a kanály jsou již postaveny. Pokud se ovšem stane, že se krokodýl nemůže dostat do nějakého jezírka (nevede k němu žádná cesta), je velmi nerudný a žere vše kolem. (*Kvák!*) Protože Potrhlík nechce dopadnout stejně jako Skrblikvák, chtěl by dostavět potřebné kanály tak, aby byli krokodýli spokojení. Ti budou spokojení, pokud i když jeden libovolný kanál vyschne, pořád se budou moci dostat z každého jezírka do kteréhokoliv jiného. A protože stavba krokoběhu Potrhlíka finančně velmi vyčerpala, chtěl by postavit nových kanálů co nejméně.

Váš program dostane na vstupu popis už existujícího krokoběhu. Ten se skládá z $N > 2$ jezírek a M kanálů, každý kanál spojuje dvojici jezírek. Vaším cílem je zjistit, kolik nejméně kanálů je třeba přidat, aby i když libovolný jeden kanál vyschne, bylo pořád možné dostat se z každého jezírka do každého.

Příklad: Pro $N = 6$ jezírek a $M = 4$ kanály vedoucí mezi jezírky $(1, 2)$, $(2, 3)$, $(3, 1)$ a $(4, 5)$ je třeba postavit alespoň další 3 kanály. (Jsou to například $(1, 4)$, $(5, 6)$, $(6, 2)$.)

18-2-6 Dominující komplikátory — 10 bodů

V prvním díle seriálu jsme si popsali a vyzkoušeli, jak funguje front-end překladače. Ve zbytku seriálu si budeme povídat o optimalizacích kódu v překladačích a o tom, jak se tyto optimalizace implementují.

První překladače zpracovávaly programy po jednotlivých příkazech. V rámci jednoho příkazu se dají provést pouze jednoduché optimalizace – lze například vyhodnotit aritmetické výrazy s konstantními operandy a zjednodušit je použitím algebraických identit. Produkovaný kód však nebude příliš dobrý, mimo jiné proto, že některé hodnoty (třeba adresy proměnných) budeme zbytečně počítat opakovaně v každém příkazu.

Aby se vyřešil tento problém, začaly se optimalizace provádět nad většími kusy programu. Některé optimalizace lze provádět nad *basic blocky*. Basic block je úsek programu, který se vždy vykonává sekvenčně, tj. neobsahuje skoky, a nedá se skočit do jeho vnitřku, tj. neobsahuje labely. Nad basic blocky se provádí například propagace konstant a kopií a nahrazování společných podvýrazů.

Většina moderních překladačů pracuje převážně nad jednotlivými funkcemi. Na této úrovni lze provádět navíc optimalizace typu mazání výpočtů, jejichž hodnota není použita, odstraňování invariantů ze smyček a mnoho dalších.

V posledních 5–10 letech se začaly prakticky používat optimalizace pracující nad celými programy, například změny v rozložení dat v paměti či interprocedurální propagace konstant.

Jednou ze základních datových struktur pro práci s celou funkcí je *Control Flow Graph* (CFG). CFG je orientovaný graf, jehož vrcholy jsou basic blocky a hrany odpovídají skokům. Z vrcholu tedy většinou vedou jedna nebo dvě hrany (podle toho, zda příslušný basic block končí nepodmíněným nebo podmíněným skokem). Občas se vyskytnou vrcholy s větším počtem výstupních hran, například pokud basic block končí příkazem `case` v Pascalu či `switch` v C. Jeden z vrcholů CFG je počáteční, v jemu odpovídajícím basic blocku začíná vykonávání funkce.

Některé optimalizace jsou přímo manipulace s hranami a vrcholy CFG, například odstraňování nedosažitelného kódu spočívá v odstranění vrcholů, do nichž nevede cesta z počátku. Mnohé další optimalizace používají CFG při analýzách nutných pro jejich provedení.

Úloha

Jednou z vlastností vrcholů CFG, kterou mnohé optimalizace potřebují znát, je *dominance*. Říkáme, že basic block A *dominuje* basic block B, jestliže každá cesta z počátečního vrcholu do B prochází přes A. Speciálně počáteční vrchol dominuje všechny vrcholy CFG. Podívejme se na následující příklad:

```
(BB0)
b := 0;

if y = 5 then
  begin (BB1)
    a := 1;
  end
else
  begin (BB2)
    a := 2;
    b := 6;
  end;

(BB3)
test (b);
b := 8;

if y > 5 then
  begin (BB4)
    a := 3;
  end
else
  begin (BB5)
    a := 4;
  end;
```

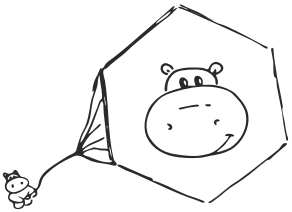
```
(BB6)
test (a);
test (b);
```

Zde basic block *BB0* dominuje všechny blocky a *BB3* dominuje *BB4*, *BB5* a *BB6*. Ostatní blocky dominují pouze sebe sama. Jedno z použití dominance je toto: Když máme v programu použití proměnné v basic blocku B, je často potřeba zjistit, kde byla do této proměnné přiřazena hodnota. Nejjednodušší případ nastává, když je jen jedno takové přiřazení (v nějakém blocku A) – například pro použití `b` v *BB6* je toto přiřazení v *BB3*; všimněte si, že přiřazení do `b` v *BB0* a *BB2* nás nezajímají, jelikož jejich hodnoty se do *BB6* nikdy nedostanou. Aby toto mohlo nastat, block A musí dominovat block B. Tato podmínka je nutná, ale ne postačující, například použití `b` v *BB3* má dvě definice (v *BB0* a *BB2*), přestože definice v *BB0* dominuje *BB3*.

Efektivní algoritmy pro určení dominance jsou ovšem velmi komplikované, proto se jimi nebudeme podrobněji zabývat. Předpokládejte, že už máte nějak relaci dominance pro všechny dvojice vrcholů spočtenu. Vaším úkolem v této úloze je navrhnout datovou strukturu, v níž lze tento výsledek reprezentovat. Chceme, aby tato datová struktura zabírala co nejméně místa, a přitom umožnila rychle pro libovolné dva basic blocky A a B rozhodnout, zda A dominuje B.

18-3-1 Trávník — 4 body

Hrošík Seček si na své zahrádce pěstuje trávník. Čas od času trávník poseče a z výtěžku uspořádá hostinu pro své přátele. Předtím, než trávník poseká, ho musí samozřejmě nechat pěkně narůst. A tak Seček potřebuje napsat program, který mu navrhne ideální den pro posekání trávníku.



Trávník se dá popsat maticí $M \times N$, kde každý prvek matice je celé číslo udávající velikost stébla v trávníku v centimetrech. Toto číslo také udává rychlost růstu stébla v centimetrech za den. Rychlost růstu se nemění. Aby Seček mohl přátele pohostit, potřebuje alespoň K centimetrů stébel trávy.

Napište tedy program, který na vstupu dostane čísla M, N a K a dále matici $M \times N$ nezáporných celých čísel, kde každý prvek odpovídá délce stébla trávníku v daném místě první den od posledního posekání. Tato matice udává

zároveň rychlost růstu jednotlivých stébel. Váš program by měl vypsat, kolik dní má Seček ještě počkat s posekáním trávníku.

Příklad: Pro $M = 2$, $N = 3$, $K = 79$ a matici

$$\begin{pmatrix} 1 & 0 & 9 \\ 8 & 9 & 3 \end{pmatrix}$$

Seček počká ještě 2 dny. Matice trávníku pak bude:

$$\begin{pmatrix} 3 & 0 & 27 \\ 24 & 27 & 9 \end{pmatrix}$$

Z tohoto trávníku lze získat 90 centimetrů trávy.

18-3-2 Duel — 5 bodů

Hrošík a prasátko jsou soutěživá zvířátka. Kde mohou, tam se snaží navzájem trumfovat. Jednou objevili zajímavou hru: Na stůl se na papírcích umístí čísla od 1 do 9. Oba hráči se potom střídají v odebírání jednotlivých čísel na svou hromádku. Vyhraje ten hráč, v jehož hromádce se jako prvnímu nachází trojice čísel dávající součet 15.

Hrošík začíná jako první a chce vyhrát. Jenže vůbec neví jak na to... Poraďte hrošíkovi, jak to zařídit, aby když začíná jako první, vždy nad prasátkem vyhrál.

18-3-3 Vrah — 10 bodů

Hugo se právě vrátil ze soustředění KSP, kam byl pozván jako účastník, a protože se nyní ve škole nudil, rozhodl se naučit své spolužáky zábavnou hru „na vraha“.

Do pytlíku se vhodí N papírků se jmény a každý si potom vylosuje jméno své oběti. Cílem hry je nikým jiným neviděn sáhnout zezadu své oběti na krk a sám se nestát obětí. Po úspěšném „zabití“ oběť předává vrahovi svůj papírek se jménem a vrah má nový cíl. Hráč končí, když dostane do ruky svůj vlastní papírek.

Jenže hráči za chvíli zjistili, že pro jistá rozlosování některé dvojice vůbec nedostanou šanci se spolu utkat (například když si hráč již na začátku vylosuje sám sebe, ale i jindy). Naštěstí měli osvícenou paní učitelku, která podporovala kulturní vyžití svých žáků a která se zeptala, kdo s kým chce mít šanci se ve hře potkat. Pak se podívala na jinak tajné rozlosování a některým dvojicím hráčů oznámila, aby si mezi sebou vyměnili své papírky, navíc tak, že počet prohození byl nejmenší možný.

Vymyslete algoritmus a napište program, který to bude dělat za paní učitelku. Na vstupu dostanete počet hráčů N, které si místo jmen očíslujeme čísly

1 až N. Následuje N čísel, kde i-té udává číslo hráče, jehož má zabít hráč číslo i. Poté číslo K a za ním K dvojic čísel určujících, kteří hráči se spolu chtějí potkat. Zdůrazněme, že jeden hráč se může chtít potkat i s více než jedním jiným hráčem. Program by měl odpovědět nejmenším možným počtem pokynů k prohození papírků, aby všech K dvojic zaručeně mělo šanci se ve hře potkat (nemusí se potkat v jedné hře, ale pro každou dvojici musí existovat průběh hry, při kterém se potkají).

Příklad: Pro $N = 6$ hráčů, rozlosování $6, 2, 4, 3, 1, 5$ a $K = 3$ dvojice $(1, 6), (1, 3), (4, 5)$, které se chtějí potkat, stačí jediné prohození papírků, a to mezi lidmi 4 a 6.

18-3-4 Pochoutka pro prasátko — 10 bodů

V lese sousedícím s poklidným rybníčkem našich hrošíků se objevilo hladem šilhající prasátko. Zaslechlo totiž zvěsti o Velké Bukvici, která si tou dobou lebedila v podzemí lesíku. Začalo tedy bez rozmyslu rejdit mezi stromy, leč brzy mu došly síly – byl to už přeci jenom nějaký čas od poslední mňamky. Budete umět prasátku poradit ?

Les si představme jako čtvercovou síť, v jednom políčku prasátko, v jiném bukvice. Aby to nebylo tak jednoduché, prasátko se v lese může pohybovat jen podle určitých pravidel a každé z nich stojí nějaké kladné množství námahy.

Konkrétně – na vstupu dostanete rozměry lesa a pozici prasátka a bukvice spolu s pravidly, podle kterých se prasátko může pohybovat. Každé pravidlo obsahuje trojici čísel $x\ y\ z$, kde x a y je povolený posun v mřížce (o kolik se změní pozice prasátka ve čtvercové síti), zatímco z je úsilí, které prasátko musí vynaložit pro daný přesun. Vaším úkolem je najít a vypsat cestu od prasátka k bukvici. Na své cestě nesmí prasátko opustit lesík. A aby milý čuník hlady nepošel, musí být vynaložené úsilí nejmenší možné.

Příklad: Les má rozměry 6×6, poloha prasátka je $[3, 3]$ a poloha bukvice $[1, 5]$. Pohyb prasátka se řídí třemi pravidly 2 2 3, 1 1 1 a -4 0 5. Potom je pro prasátko nejvýhodnější použít dvakrát pravidlo 2 ($\rightarrow [5, 5]$) a pak jednou pravidlo 3 ($\rightarrow [1, 5]$). Vynaložená námaha je pak $2 \cdot 1 + 5 = 7$.

18-3-5 Hroší lov — 13 bodů

Do hrošího království vtrhlo šílenství – divoké prase začalo zběsile rozrývat rozsáhlé části lesa. Marné bylo domlouvání ostatních obyvatel polesí, kvičící střela zvolna proměňovala hluboké hvozdy v důlní centrum pro těžbu bukvic.

Hrochům nakonec došla trpělivost a rozhodli se, že nezbedné prase polapí, upečou a sní. Teprve nyní prasátko dostalo strach – ale ouha, bylo příliš pozdě. Stádo nasupených hrochů pročesávalo les a dalo se zastavit leda až večerní tmou. Pomůžete prasátku uniknout ještě dříve, než nastane večer?

Podobně jako v předešlém případě je možné les popsat jako čtvercovou síť, po které je pohyb všech zvířat možný pouze podle předepsaných pravidel a nijak jinak.

Na vstupu tedy dostanete rozměry lesa a pozici prasátka a hrochů společně s neohodnocenými pravidly pohybu pro každého z nich (i pro prasátko). Dále dostanete čas t, který zbývá do setmění. Vaším úkolem je najít a vypsat únikovou cestu pro prasátko. Ta se vyznačuje tím, že se prasátko celou dobu pohybuje po bezpečných polích, a buď uteče z lesa ven (dosáhne hranice lesa), nebo uplyne doba t (pak jej totiž hroši přestanou hledat). Bezpečné pole je takové, na které v čase pobytu prasátka nemůže dostat žádný hroch. Ještě dodáváme, že více hrochů smí v jeden moment stoupnout na jedno políčko a čas chápeme jako diskrétní kroky, v nichž každé zvíře musí udělat pohyb podle nějakého svého pravidla (čili nemůže zůstat stát na místě).

Příklad: Les má rozměry 3×3 a do setmění zbývá čas $t = 5$. Prasátko je na souřadnicích $[2, 2]$ a smí se pohybovat podle jediného pravidla $-1\ 0$. Hroši jsou dva – první je na $[3, 2]$ a pohybuje se podle jednoho pravidla $-1\ 0$, druhý je na $[2, 3]$ a pohybuje se podle dvou pravidel $0\ -1$ a $-1\ 0$.

Hledaná cesta pro prasátko je dvakrát použít pravidlo jedna (čili $-1\ 0$), čímž se dostane na $[0, 2]$, což jest ven z lesa.

18-3-6 Komplikovanější komplikátory — 10 bodů

Jedním z problémů, které je často nutné při optimalizacích v kompilátorech řešit, je *analýza toku dat* (*dataflow*). Základní optimalizací, která se pomocí dataflow provádí, je globální propagace konstant. Její úlohou je rozhodnout, které proměnné mají vždy konstantní hodnotu, a nahradit jejich použití touto hodnotou. Například následující kód (v mezijazyce popsaném v první sérii):

```
assign a 0
assign b 1
if (c = 0) 1 2

label 1
assign c (a + b)
goto 3

label 2
assign c 1
assign a 2

label 3
assign x (c + a)
```

bude po propagaci konstant vypadat takto:

```
assign a 0
assign b 1
if (c = 0) 1 2

label 1
assign c 1
goto 3

label 2
assign c 1
assign a 2

label 3
assign x (1 + a)
```

Povšimněme si, že nestačí jen určit, která proměnná je konstanta, protože to se může v průběhu programu změnit. Například `a` je v prvních třech basic blocích konstanta 0, ale v posledním může mít hodnotu 0 nebo 2. Budeme si proto chtít určit, zda je proměnná konstantní, zvlášť na začátku a na konci každého basic bloku – lokální propagace uvnitř každého basic bloku je jednoduchá, prostě projdeme postupně všechny příkazy a odsimulujeme si je. Pro každou proměnnou `x` a každý basic blok b budeme mít proměnné x_b^+ a x_b^-, které určují stav `x` na začátku a na konci bloku b. Ohodnocení proměnných bude nabývat jedné z následujících hodnot:

- `Top` – tato hodnota znamená, že `x` může být konstantní, ale ještě nevíme, jakou by ta konstanta měla mít hodnotu.
- Nějaké číslo c – to znamená, že `x` je buď vždy rovno c, nebo není konstantní.
- `Bottom` – tato hodnota znamená, že `x` není konstantní.

Tyto hodnoty si uspořádejme tak, že Bottom $< c <$ Top pro libovolnou konstantu c (konstanty jsou navzájem neporovnatelné). Toto uspořádání je důležité pro důkaz konečnosti algoritmu dataflow.

Jestliže nějak určíme hodnoty těchto proměnných na konci všech basic bloků, určit je na začátku libovolného bloku b je snadné, prostě je potřeba slít hodnoty příslušných proměnných na konci předchůdců b. Pravidla pro slévání jsou tato:

- `Bottom` slité s libovolnou hodnotou je `Bottom` – pokud se hodnota může v některém z předchozích bloků měnit, na začátku b nemůže být konstantní.
- `Top` slité s libovolnou hodnotou v je v – `Top` nám říká, že o hodnotě proměnné nic nevíme, takže pokud je na konci některého z předchozích bloků rovna konstantě c, může to být pravda i na začátku b (ale nemůže být vždy rovná libovolné jiné konstantě).

- Slití dvou různých konstant je `Bottom` – taková proměnná nabývá alespoň dvou různých hodnot.
- Slití dvou stejných konstant c je zase c – výsledek pořád může být tato konstanta.

Naopak, pokud bychom znali hodnoty proměnných na začátku basic bloku, hodnoty na konci určíme odsimulováním příkazů v basic bloku s tím, že operace s konstantami vyhodnocujeme. Výsledky operací s `Top` jsou skoro vždy `Top` a operací s `Bottom` zase `Bottom`, až na drobné výjimky – například 0 krát cokoliv je vždy 0, bez ohledu na hodnotu výrazu, který počítáme. Je potřeba si dávat maličko pozor, aby toto vyhodnocování operací bylo monotónní, tj. pokud x op a vyhodnotíme jako a', x op b vyhodnotíme jako b' a $a \leq b$, pak i $a' \leq b'$. Tedy například pokud chceme, aby $\texttt{Bottom} \times 0 = 0$, pak $\texttt{Bottom} \times \texttt{Top}$ může být `Top` nebo 0, ale nic jiného. Tato podmínka je nutná pro zajištění konečnosti níže popsaného algoritmu. Také je vhodné, abychom nevraceli `Top`, pokud alespoň jeden z operandů nebude `Top`. To už není nutné pro konečnost, jen to většinou nedává moc smysl – taková operace by tvrdila, že její výsledek je nezávisle na vstupech nějaká neznámá konstanta.

Algoritmus dataflow funguje takto: Na začátku algoritmu nastavíme všechny proměnné na `Top`, kromě těch na začátku prvního bloku, které nastavíme na `Bottom`. Poté budeme opakovat operace popsané v minulých odstavcích tak dlouho, dokud se něco mění. Operace lze provádět v libovolném pořadí, prakticky se to dělá tak, že si udržujeme seznam bloků, pro které se změnily hodnoty proměnných na konci některého z jejich předchůdců. Z něj si odebereme libovolný blok b, slijeme hodnoty z jeho předchůdců, vyhodnotíme si výrazy uvnitř b a v případě, že se ohodnocení proměnných na konci b změnilo, přidáme do seznamu všechny následníky b.

Stav proměnných poté, co dosáhneme ustáleného stavu, je řešením problému. K tomu je potřeba dokázat, že se algoritmus vždy zastaví a že je korektní, tj. že pokud proměnná není konstantní, pak její hodnota na konci bude `Bottom`. Pro důkaz konečnosti si povšimneme, že ohodnocení libovolné proměnné se může změnit nejvýše třikrát – na začátku je `Top`, pak se můžeme nějakou dobu domnívat, že by její hodnota mohla být konstantní, a nakonec se její hodnota může stát `Bottom`, pokud dokážeme, že konstantní není. Protože proměnných, jejichž ohodnocení určujeme, je nejvýše N^2, kde N je délka programu, algoritmus se časem jistě zastaví.

Zajímavější je korektnost. Nejprve si ukážeme, že na konci žádná proměnná nebude mít hodnotu `Top`. Předpokládejme, že tomu tak není a že například x_b^+ je `Top`. Vezměme si libovolnou cestu p z počátku do bloku b a vracejme se z p po b. V každém okamžiku musíme mít alespoň jednu proměnnou ve stavu `Top` – když přecházíme přes hranu CFG, `Top` na jejím konci mohl vzniknout jedině z `Top`u na jejím začátku, případně slitím `Top`ů z ostatních předchůdců.

Vyhodnocením příkazu také `Top` mohl vzniknout, jen pokud některý z operandů byl `Top`. Tedy `Top` by musel existovat i na začátku úplně prvního bloku, což ale není možné – ohodnocení všech proměnných na začátku jsme nastavili na `Bottom`.

Předpokládejme nyní, že algoritmus je chybný a nějaké proměnné x_b^+ přiřadí ohodnocení c, i když `x` na začátku basic bloku b může nabývat i jiné hodnoty d. Buď p cesta z počátku do b, která odpovídá výpočtu, jenž způsobí, že `x` je na konci rovno d. Někde na této cestě je první místo, kde se námi nalezené ohodnocení rozchází s tímto výpočtem. Nemůže to být úplně na začátku, neboť tam je ohodnocení všech proměnných `Bottom`, čili o hodnotách proměnných nic netvrdíme. Nemůže to také být na začátku jiného basic bloku, protože pokud o nějaké proměnné tvrdíme, že má hodnotu c, museli jsme to tvrdit i na konci předchozího basic bloku a na hraně CFG se hodnota proměnné nemohla změnit. Čili bychom museli někde mít výraz, jehož operandy mají správné ohodnocení, ale jeho výsledek chybné. To ale nemůže nastat, protože jsme používali pouze korektní pravidla pro vyhodnocování výrazů. Čili všechny proměnné na konci budou ohodnoceny korektně.

S několika drobnými triky se tento algoritmus se dá implementovat s časovou složitostí $O(N \cdot E)$, kde E je počet hran CFG. Naše implementace je pro lepší čitelnost o něco hloupější a nesnažili jsme se ji příliš optimalizovat, takže její časová složitost je o něco horší, $O(N^2 \cdot E)$. Můžete ji najít za kuchařkou, před řešením příkladů první série.

Úloha

Jedním z problémů, které se dají pomocí dataflow analýze řešit, je mazání mrtvého kódu, tedy výrazů, jejichž hodnota není k ničemu použita. Výraz je živý, pokud má nějaké efekty, které nejdou smazat (například volání funkce, skok, přiřazení do globální proměnné), nebo pokud je jeho hodnota použita v živém výrazu. Například v následujícím příkladě jsou živá jen přiřazení do i (protože hodnota i je použitá v podmínce skoku) a druhé přiřazení do k (které je použito v návratové hodnotě funkce):

```
assign i 0
assign j 0
assign k~10
assign k~15

label 1
assign i (i + 1)
assign j (j + 1)
if (i < 100) 1 2

label 2
assign result k
```

Vaším úkolem je navrhnout algoritmus pro nalezení mrtvých výrazů založený na dataflow analýze.

Ukázková implementace algoritmu propagace konstant z 18-3-6

```
#include <stdio.h>
#define MAX_BLOKU 100
#define MAX_PROM 100

  /* Typy pro reprezentaci programu. */
unsigned n_prom;                          /* Proměnné očísl. od 0 do N_PROM - 1. */

struct operand {                          /* Operand výrazu. Pokud je PROM, */
    enum typop {PROM, CISLO} typ;         /* je hodnota číslo proměnné, */
    unsigned hodnota; };                  /* jinak to je hodnota čísla. */

struct vyraz{
    char operator;
    struct operand operandy[2]; };

  /* Pro ASSIGN je v data[0] proměnné, kam se přiřazuje, ve vyraz přiřazovaný výraz. */
  /* Pro LABEL je v data[0] číslo labelu. */
  /* Pro IF je ve vyraz podmínka, v data[0]/data[1] label, kam se skáče při true/false. */
  /* Pro GOTO je v data[0] label, na který skáče. */
struct prikaz {
    enum prikazy {ASSIGN, LABEL, IF, GOTO} prikaz;
    struct vyraz vyraz;
    unsigned data[2];
    struct prikaz *dalsi, *predchozi;     /* Příkazy jsou uloženy v seznamu. */

struct hrana {                            /* CFG. */
    struct basic_block *z, *k;            /* Hrana z bloku Z do bloku K. */
    struct hrana *nasl_dalsi, *pred_dalsi; };  /* Předchůdci a následníci bloku. */

struct basic_block {
    unsigned index;                       /* Číslo bloku, od 0 do N_BLOKU. */
    struct hrana *pred, *nasl;            /* Seznam předchůdců a následníků. */
    struct prikaz *prvni, *posledni; };   /* Příkazy v bloku. */

unsigned n_bloku;                         /* CFG se skládá z N_BLOKU bloků.
                                             Počáteční blok má číslo 0. */
struct basic_block cfg[MAX_BLOKU];

struct hodnota {                          /* Hodnoty pro propagaci konstant. */
    enum hod { TOP, KONST, BOTTOM } hod;
    unsigned konstanta; };

  /* Ohodnocení proměnných na začátku a na konci bloku. */
struct hodnota ohodn_zac[MAX_BLOKU][MAX_PROM];
struct hodnota ohodn_kon[MAX_BLOKU][MAX_PROM];

void slij_hodnoty  (struct hodnota *h, struct hodnota h1) {   /* Slije H a H1 do H. */
    if (h->hod == BOTTOM || h1.hod == TOP) return;
    if (h->hod == TOP || h1.hod == BOTTOM) { *h = h1; return; }
    if (h->konstanta != h1.konstanta) h->hod = BOTTOM;
}

void slij  (struct basic_block *bb) {     /* Slití hodnot na začátku basic bloku */
    struct hrana *p;                      /* z hodnot na koncích předch. bloků. */
    struct basic_block *pred;
    unsigned i;
```

```
    for (i = 0; i < n_prom; i++) ohodn_zac[bb->index][i].hod = TOP;
    for (p = bb->pred; p; p = p->pred_dalsi) {
      pred = p->z;
      for (i = 0; i < n_prom; i++) slij_hodnoty (&ohodn_zac[bb->index][i],
          ohodn_kon[pred->index][i]);
    }
}

  /* Odsimuluje přiřazení PRIK. */
void vyhodnot_vyraz (struct basic_block *bb, struct prikaz *prik) {
    unsigned vysl = prik->data[0];
    struct hodnota hod[2], vys;
    unsigned i;
    struct operand *op;

    for (i = 0; i < 2; i++) {
      op = &prik->vyraz.operandy[i];
      if (op->typ == PROM) hod[i] = ohodn_kon[bb->index][op->hodnota];
      else { hod[i].hod = KONST; hod[i].konstanta = op->hodnota; }
    }

    /* Pokud je alespon jeden z operandů TOP, je bezpečné vrátit TOP (byť v praxi je
       výhodnější konvergovat k BOTTOMu rychleji). */
    if (hod[0].hod == TOP || hod[1].hod == TOP) vys.hod = TOP;
    else {
      vys.hod = KONST;
      switch (prik->vyraz.operator) {
      case '+': if (hod[0].hod==BOTTOM||hod[1].hod==BOTTOM)vys.hod=BOTTOM;
                else vys.konstanta = hod[0].konstanta + hod[1].konstanta; break;
      case '-': if (hod[0].hod==BOTTOM||hod[1].hod==BOTTOM)vys.hod=BOTTOM;
                else vys.konstanta = hod[0].konstanta - hod[1].konstanta; break;
      case '*': if ( (hod[0].hod == KONST && hod[0].konstanta == 0)
                  || (hod[1].hod==KONST && hod[1].konstanta==0)) vys.konstanta=0;
                else if (hod[0].hod == BOTTOM || hod[1].hod == BOTTOM)
                    vys.hod = BOTTOM;
                else vys.konstanta = hod[0].konstanta * hod[1].konstanta; break;
      case '/': if (hod[0].hod == KONST && hod[0].konstanta == 0) vys.konstanta = 0;
                else if (hod[0].hod == BOTTOM || hod[1].hod == BOTTOM)
                    vys.hod = BOTTOM;
                else vys.konstanta = hod[0].konstanta / hod[1].konstanta; break;
      default: vys.hod = BOTTOM;
      }
    }
    ohodn_kon[bb->index][vysl] = vys;
}

int vyhodnot (struct basic_block *bb) {          /* Vyhodnotí výrazy v bloku BB, vrátí */
    struct prikaz *prik;                         /* 1 změnilo-li se ohodn. na jeho konci. */
    unsigned i;
    struct hodnota ohodn_stare[MAX_PROM];

    for (i = 0; i < n_prom; i++) {               /* Uložíme staré ohodnocení, */
      ohodn_stare[i] = ohodn_kon[bb->index][i];  /* nové budou hodnoty ze začátku. */
```

```
      ohodn_kon[bb->index][i] = ohodn_zac[bb->index][i];
    }
    for (prik = bb->prvni; prik; prik = prik->dalsi) if (prik->prikaz == ASSIGN)
        vyhodnot_vyraz (bb, prik);
    for (i = 0; i < n_prom; i++)                    /* Porovnáme se starým ohodnocením. */
      if (ohodn_stare[i].hod != ohodn_kon[bb->index][i].hod) return 1;
    return 0;
}
void cprop_dataflow (void) {                        /* Vyplní ohodnocení pro prop. konst. */
    unsigned i, b;
  /* Zmenene je seznam bloků, jejichž ohodn. se změnilo a je třeba ho přepočítat. Chováme
       se k němu pro jednoduchost jako k zásobníku, buď bývá lepší používat jiné pořadí. */
    struct basic_block *zmenene[MAX_BLOKU], *bb;
    unsigned pocet_zmenenych;
    char je_zmeneny[MAX_BLOKU];
    struct hrana *n;
    for (b = 0; b < n_bloku; b++)
      for (i = 0; i < n_prom; i++)
        ohodn_zac[b][i].hod = ohodn_kon[b][i].hod = TOP;
    for (i = 0; i < n_prom; i++) ohodn_zac[0][i].hod = BOTTOM;
    for (b = 0; b < n_bloku; b++) je_zmeneny[b] = 0;    /* Na začátku je změněný jen
                                                          počáteční blok. */
    je_zmeneny[0] = 1;
    zmenene[0] = &cfg[0];
    pocet_zmenenych = 1;
    while (pocet_zmenenych > 0) {
      bb = zmenene[--pocet_zmenenych];
      je_zmeneny[bb->index] = 0;
      slij (bb);
      if (!vyhodnot (bb)) continue;
      /* Hodnoty na konci se změnily, vložíme následníky BB do seznamu, pokud už v něm
           nejsou. */
      for (n = bb->nasl; n; n = n->nasl_dalsi) if (!je_zmeneny[n->k->index]) {
          je_zmeneny[n->k->index] = 1;
          zmenene[pocet_zmenenych++] = n->k;
      }
    }
}
```

18-4-1 HP — 5 bodů

Výpočetní centrum HP (Hippo Programmers) se obrátilo vzhůru nohama. Zmatení hrošíci pobíhají sem a tam a každou chvíli se některý z nich zastaví a usne. Není také divu. Hroši nejsou zvyklí pracovat, natož řešit složité výpočetní úlohy. Jedna taková úloha teď sužuje celou společnost HP, a protože si s tím hrošíci sami neporadí, budete jim muset poradit vy. Výpočetní centrum dostalo následující úkol.

Na vstupu máte libovolně dlouhé číslo ve dvojkovém zápisu a máte za úkol zjistit, kolik nul bude mít takové číslo na konci, když ho přepíšeme do desítkového zápisu. A protože hroši chtějí mít na všechno dost času (zejména na spánek a jídlo), měl by váš algoritmus pracovat co nejrychleji. Můžete předpokládat, že se dané číslo vejde do `integeru`.

Vaším cílem je vyprodukovat *algoritmus*, ne program, stačí tedy, když váš postup dostatečně podrobně popíšete. Na druhou stranu, za pěkný program by vás mohla neminout i nějaká prémie :–)

Příklad: Číslo 11111010 končí v desítkovém zápisu na 1 nulu. Ještě složitější příklad – číslo 1010010000010000 končí na 3 nuly.

18-4-2 Elektronické hrátky — 10 bodů

Malý Martínek se jednoho dne velmi nudil. Nudil se tak moc, že už se víc ani nudit nemohl. Bloumal po bytě a hledal, čím by se zabavil. Televize byla rozbitá, hračky ho již omrzely a všechny knížky o diferenciálním počtu Martínek dávno přečetl. Při svém bloumání narazil na tatínkovu krabici plnou zvláštních broučků a při bližším prozkoumání zjistil, že se jedná o integrované elektronické obvody. Martínek si tedy vzal tatínkovo kontaktní pole a začal si hrát. Když už se mu na kontaktní pole nic nevešlo, rozhodl se, že vyzkouší, co vlastně postavil. Ale ouha. V té obrovské změti drátů a obvodů se skoro nevyznal, natož aby zjistil, jak vlastně jeho síť funguje. Už začínal natahovat moldánky, ale pak si vzpomněl, že má spoustu přátel, kteří řeší KSP, a ti mu jistě pomohou. Aby vám s tím Martínek alespoň trochu pomohl, přepsal schéma své sítě na papír a přitom si všiml, že mezi obvody není žádný cyklus. Všechny použité obvody jsou dvouvstupová logická hradla NAND. Hradlo NAND (negované AND) je hradlo s dvěma vstupy a jedním výstupem, které se chová dle tabulky. Síť má ještě několik nezapojených drátků, na které se zapojí vstup sítě, a několik drátků, které představují výstup sítě.

Hradlo NAND:	*vstup 0*	*vstup 1*	*výstup*
	0	0	1
	0	1	1
	1	0	1
	1	1	0

Napište program, který dostane na vstupu schéma sítě a pak bude schopen odpovídat na dotazy typu: „Když bude tohle na vstupu sítě, co bude na jejím výstupu?“ Svoji síť vám Martínek popsal takto:

Máte celkem N hradel, k_i bitů na vstupu a k_o bitů na výstupu. Hradla jsou číslována od 1 do N. U každého hradla máte napsáno, kam jsou zapojeny jeho dva vstupy. Každý vstup může být zapojen buď na výstup jiného hradla, nebo na některý vstupní bit celé sítě. Dále máte seznam k_o hradel, jejichž výstupy jsou zapojeny na výstupní bity celé sítě.

Příklad: Máme hradlovou síť se 4 hradly, 2 bity vstupu a 1 bitem výstupu.

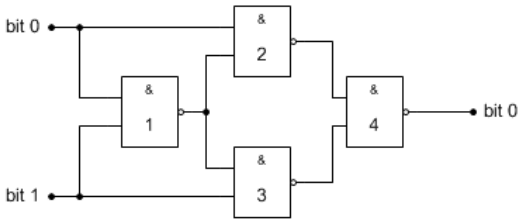
Popis hradel:

hradlo	*vstup 1*	*vstup 2*
1.	bit 0	bit 1
2.	bit 0	hradlo 1
3.	hradlo 1	bit 1
4.	hradlo 2	hradlo 3

Popis výstupu:

výstup	*připojen na*
bit 0	hradlo 4

Pro přehlednost ještě obrázek této sítě:



Program nyní spočítá ze zadaného vstupu výstup celé sítě:

vstup	*výstup*
00	0
01	1
10	1
11	0

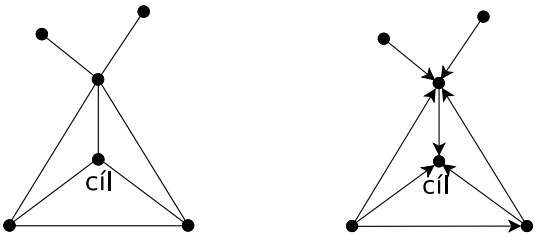
Pozn: Pořadí bitů vstupu a výstupu je stejné jako u klasického binárního zápisu. Tedy 0. bit představuje jednotky, 1. bit dvojky, 2. bit čtyřky atd.

18-4-3 Běh městem — 9 bodů

V Kocourkově se rozhodli uspořádat velkou oslavu na počest výročí města. Městská pokladna však zeje prázdnotou, a tak museli radní vymyslet finančně nenáročný způsob oslavy. Radní si dlouho lámali hlavu, až je napadlo uspořádat pro obyvatele města závod. Zpráva se rychle roznesla do širokého i dalekého okolí a na oslavy se začali sjíždět zvědavci i z jiných měst. Jak už to ale v Kocourkově bývá, radní zapomněli domyslet trať závodu. Ulice města jsou velice zamotané, takže i místní obyvatelé mají problémy trefit tam, kam zrovna chtějí. Tajemník městské rady si je tohoto problému vědom, ale má na práci spoustu jiných věcí (koneckonců dělá většinu práce za celou městskou radu), a tak se obrátil s prosbou o pomoc na vás.

Máte danou mapu města jako neorientovaný graf, kde každá hrana představuje ulici a každý vrchol je křižovatka, do které ústí alespoň 3 ulice. Jeden z vrcholů je označen jako cíl celého závodu. Tajemník vás požádal, abyste z každé ulice udělali jednosměrku, a to tak, aby každý závodník, který bude dodržovat pravidla jednosměrek, doběhl po konečném počtu kroků do cíle. Závodníci mohou vybíhat z libovolného místa a na každé křižovatce se libovolně rozhodnout, kam poběží (samozřejmě jen pokud vede z dané křižovatky více ulic; pokud z křižovatky nevede ulice žádná, závodník zde musí čekat do skonání věků). Zároveň by bylo rozumné, aby závod někdy skončil, takže každý závodník se musí dostat do cíle v konečném počtu kroků, ať se na křižovatkách rozhodne pro libovolnou jednosměrku. Pokud existuje víc možných řešení, program vypíše libovolné z nich.

Příklad: Na obrázku vidíte mapu města s vyznačeným cílem a jednu z možných správných orientací:



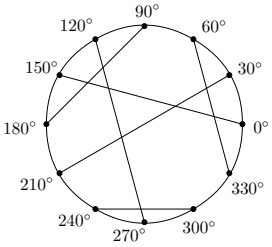
18-4-4 Metro pro krtky — 10 bodů

Baron von Maulwurfshaufen měl okolo svého sídla překrásnou zahradu v raně euklidovském slohu, které vévodil obrovský kruhový záhon plný macešek a okrasných okurek. Leč staleté prokletí rodu von Maulwurfshaufenů na sebe nedalo dlouho čekat. Na okraji záhonu se začaly objevovat zlověstné krtiny: první, druhá, třetí, čtvrtá, ... až N-tá. Nedosti na tom, krtci vedou velice bohatý společenský život a chtějí své příbuzné na druhém konci záhonu navštěvovat, a tak se rozhodli, že si mezi krtinami vyvrtají metro.

Všechny tunely metra vedou po úsečkách a v téže hloubce, proto se nejspíš mnohé z nich budou protínat a v místech průsečíků bude zapotřebí postavit výhybky a zaměstnat žížaly, které je budou obsluhovat. Krtci se ovšem obávají, že v okolí nebude dostatek žížal. Jelikož krtci jsou na rozdíl od cholerického pana barona milá a přátelská zvířátka (černý sametový kožíšek, drápy jako vývrtky atd., však to znáte), jistě jim rádi pomůžete zjistit, kolik žížal budou potřebovat najmout.

Napište program, který dostane zadané polohy krtin (měřené ve stupních, podobně jako jsme popisovali svíčky v úloze 18-1-4) a dvojice krtin, mezi kterými povedou trasy metra, a odpoví počtem průsečíků tras. Můžete předpokládat, že v každé krtině končí právě jedna trasa.

Napište program, který dostane zadané polohy krtin (ve stupních, viz obrázek) a seznam dvojic krtin, mezi nimiž povedou trasy metra. Program pak odpoví počtem průsečíků tras (pro náš obrázek je to 8). Navíc můžete předpokládat, že v každé krtině končí právě jedna trasa a že se v žádném bodě neprotnou více než dvě trasy.



18-4-5 Datokopci — 15 bodů

Moderní obor nazývaný Data Mining alias Datokopectví pokračuje ve svém vítězném tažení světem. Proniká už i do vydavatelského průmyslu. Známá tiskařská firma U-tisk se rozhodla jít s dobou a místo zaměstnávání spousty redaktorů shánějících v terénu potřebné informace se jala zprávy raději dolovat.

Za tímto účelem zakoupili dvojrozměrné pole, rozdělili jej na jednotková políčka a prozkoumali, kolik lze na kterém z nich vytěžit dobrých a špatných zpráv (obojí je dáno nějakým přirozeným číslem). Vytěžené zprávy chtějí dopravovat do redakcí, kde se budou zpracovávat: dobré zprávy do redakce pohádek ležící podél celého západního okraje pole, špatné do redakce zpravodajství ležící podél celého severního okraje.

Zprávy se mají dopravovat pomocí pásových dopravníků. Na každém políčku může stát buďto pás běžící z jihu na sever, nebo z východu na západ a naloží na něj vše, co je z příslušného políčka vytěženo. Každý pás musí vést buďto přímo do redakce (je-li políčko u okraje) nebo pokračovat pásem na sousedním políčku, který vede ve stejném směru, čímž se zprávy přidají ke zprávám ze sousedního políčka.

Vaším cílem je napsat program, který po zadání výtěžností jednotlivých políček navrhne rozmístění dopravníků, aby se do redakcí dostalo co nejvíce zpráv správného druhu (dobré zprávy se ve zpravodajství nehodí, podobně jako jsou nežádoucí špatné v pohádkách). Oba druhy zpráv jsou ceněny stejně.

Příklad: Pro pole 4×4 s dobrými zprávami rozmístěnými podle prvního obrázku a špatnými podle druhého je jedno ze správných řešení na obrázku třetím (vydoluje se 98 zpráv).

$$\begin{pmatrix} 0 & 0 & 10 & 9 \\ 1 & 3 & 10 & 0 \\ 4 & 2 & 1 & 3 \\ 1 & 1 & 20 & 0 \end{pmatrix} \qquad \begin{pmatrix} 10 & 0 & 0 & 0 \\ 1 & 1 & 1 & 30 \\ 0 & 0 & 5 & 5 \\ 5 & 10 & 10 & 10 \end{pmatrix} \qquad \begin{pmatrix} \leftarrow & \leftarrow & \leftarrow & \uparrow \\ \leftarrow & \leftarrow & \leftarrow & \uparrow \\ \leftarrow & \leftarrow & \leftarrow & \uparrow \\ \leftarrow & \leftarrow & \leftarrow & \uparrow \end{pmatrix}$$

18-4-6 Komplikátorový φgl — 11 bodů

V minulé sérii jsme si ukázali, jak se dělá globální propagace konstant pomocí dataflow. Nepříjemnou vlastností tohoto algoritmu je jeho kvadratická paměťová (a tedy i časová) složitost. Jejím důvodem je to, že si hodnotu každé proměnné určujeme na začátku a konci každého basic bloku. To ovšem vypadá jako bezdůvodné plýtvání – drtivá většina proměnných nemění své hodnoty příliš často (zejména pomocné proměnné, které si vytváří kompilátor, jsou často nastavovány jen na jednom místě v programu).

Úplně ideální by bylo, kdyby toto byla pravda pro všechny proměnné. Pokud se proměnná nastaví jen jednou a pak už se nemění, stačí si pro ni pamatovat jen jednu hodnotu, protože jestliže ji na tomto jediném místě nastavíme na konstantu, musí být této konstantě rovná úplně všude. Samozřejmě, že ne všechny programy tuto podmínku splňují. Občas si dokážeme pomoct přejmenováním proměnných. Například kód

```
assign a 0
assign b (a + 1)
assign a b
assign c (a + 1)
```

ve kterém do a přiřazujeme dvakrát, lze přepsat na program

```
assign a_1 0
assign b (a_1 + 1)
assign a_2 b
assign c (a_2 + 1)
```

kde se do každé proměnné přiřazuje jen jednou. Ovšem existují programy, kde to udělat nelze. Třeba

```
if (a < b) 1 2

label 1
assign x 1
goto 3

label 2
assign x 2

label 3
assign i 0
assign s x
```

```
label 4
assign s (s + i)
assign i (i + 1)
if (i < 100) 4 5

label 5

assign result s
```

nejde přepsat tak, abychom do proměnných x, i a s nepřiřazovali dvakrát. Toto nastane tehdy, jestliže chceme někdy použít proměnnou, do které jsme předtím mohli dosadit různé hodnoty. Abychom se s tím vypořádali, přidáme si do programu takzvané φ funkce. Funkce φ se nacházejí vždy jen na začátcích basic bloků, a to v místech, kde se sbíhá víc různých definic jedné proměnné. Argumenty φ funkce odpovídají hranám, které do bloku vedou, a její výsledek je vždy roven tomu argumentu, z jehož hrany přijde provádění programu. Například výše uvedený kód vypadá po doplnění φ funkcí takto:

```
if (a < b) 1 2

label 1
assign x 1
goto 3

label 2
assign x 2

label 3
assign x φ(x,x)
assign i 0
assign s x

label 4
assign s φ(s,s)
assign i φ(i,i)
assign s (s + i)
assign i (i + 1)
if (i < 100) 4 5

label 5
assign result s
```

Nyní již můžeme proměnné přejmenovat tak, aby se do každé z nich přiřazovalo jen jednou:

```
if (a1 < b2) 1 2

label 1
assign x3 1
goto 3

label 2
assign x4 2

label 3
assign x5 φ(x3,x4)
assign i6 0
```

```
assign s7 x5

label 4
assign s8 φ(s7,s10)
assign i9 φ(i6,i11)
assign s10 (s8 + i9)
assign i11 (i9 + 1)
if (i11 < 100) 4 5

label 5
assign result s8
```

Výsledku této transformace se říká *SSA forma* (kde SSA znamená „Static Single Assignment“). Nyní vás možná napadlo několik otázek, na které je nutné si odpovědět:

- Proč jsme přejmenovali i proměnné a a b, a přiřadili i proměnným s různými jmény různá čísla? Důvod je ten, že nyní můžeme na původní jména proměnných zapomenout a pracovat jen s jejich novými *verzemi*. Je samozřejmě praktické, aby každá verze měla vlastní číslo, protože pak se jím dají indexovat tabulky, do nichž si optimalizace ukládají pomocné hodnoty vztahující se k dané verzi.
- Přestože jsme vám to v předchozím textu zatajili, nebylo by dobré, abychom používali neinicializované proměnné. SSA forma by tedy měla splňovat to, že každé použití verze proměnné je dominováno její definicí. Na první pohled by se mohlo zdát, že například proměnná s_{10} toto nesplňuje (má použití ve φ funkci, která je před její definicí). Vzpomeňme si ale, že tuto hodnotu použijeme pouze tehdy, pokud přijdeme z hrany z konce bloku s návěštím 4, a v tomto okamžiku je jistě hodnota s_{10} definována. Často je praktické se dívat na použití ve φ funkcích tak, jako by se ve skutečnosti nacházela na hranách, jimž odpovídají (tedy použití s_{10} a i_{11} ve φ funkcích se chovají tak, jako by byly na hraně z konce bloku s návěštím 4 na jeho začátek).
- Jak SSA formu vytvořit? Existuje několik různých algoritmů, všechny však jsou poměrně netriviální a nebudeme se jimi v tomto seriálu zabývat. Pouze poznamenejme, že časová složitost těchto algoritmů bývá v nejhorším případě kvadratická (převod do SSA formy může v nejhorším případě způsobit kvadratické prodloužení kódu programu), nicméně pro běžné programy, které neobsahují mnoho složitě se chovajících proměnných, je skoro lineární.
- Jak se SSA formy zbavit? Na konci kompilace se potřebujeme zbavit φ funkcí a verzí proměnných, abychom mohli program přeložit do assembleru. Jedna varianta, která vás asi napadne, je prostě zahodit čísla verzí a vrátit se k původním proměnným. To ovšem nemusí fungovat. Například kód

```
assign a x
assign x y
assign b a
```

po převedení do SSA formy vypadá takto:

```
assign a1 x2
assign x3 y4
assign b5 a1
```

Zatím je vše v pořádku, pokud zahodíme verze proměnných, dostaneme původní program. Může se ale stát, že optimalizace nazývaná propagace kopií změní tento kód na

```
assign a1 x2
assign x3 y4
assign b5 x2
```

Zahodíme-li nyní verze proměnných, dostáváme program

```
assign a x
assign x y
assign b x
```

v němž má proměnná b má na konci chybnou hodnotu. Další jednoduchá idea je prostě považovat verze proměnných za nové proměnné, a φ funkce nahradit přiřazeními, které přidáme na příslušné hrany. Toto řešení je korektní, ale není příliš vhodné – do programu typicky přidáme mnoho přiřazení, a navíc budeme mít podstatně víc proměnných. Proto se používá „něco mezi“ – verze proměnných, které spolu navzájem nekolidují (tj. není místo v programu, kde bychom zároveň potřebovali znát hodnotu obou) slepíme do jedné proměnné. Tím se zbavíme většiny přiřazení a zbylé kolidující verze proměnných pak prohlásíme za nové proměnné.

Na závěr si naznačme, jak se dá SSA forma využít. Typicky se podstatně zjednoduší úlohy, které se dříve řešily pomocí dataflow analýzy. Například v propagaci konstant si stačí hodnotu pamatovat pro každou verzi (není třeba rozlišovat, na kterém místě). Algoritmus pak vypadá takto:

1) Hodnoty všech verzí nastav na Top a vlož je do fronty.
2) Z fronty odeber verzi v. Projdi všechny přiřazení tvaru `assign x něco`, v nichž je v použito.
   a) pokud `něco` je φ funkce, slij hodnoty ze všech hran
   b) jinak vyhodnoť `něco`

   Pravidla slévání hodnot a vyhodnocování výrazů jsou stejná, jaká jsme si popsali v minulé sérii. Pokud je získaná hodnota jiná, než jakou jsme měli u x uloženou, nastavíme x novou hodnotu a přidáme x do fronty (pokud tam už není).
3) Opakujeme 2) dokud fronta není prázdná.

4) Verze, jejichž hodnota je nějaká konstanta, nahradíme v programu touto konstantou a přiřazení do nich smažeme.

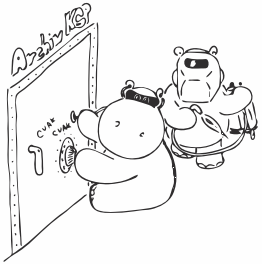
Protože hodnota přiřazená proměnné se změní nanejvýš dvakrát (z Top na konstantu a z konstanty na Bottom), je časová složitost tohoto postupu lineární ve velikosti programu (v SSA formě), což většinou bývá podstatně lepší, než složitost, jíž jsme dosáhli klasickou dataflow analýzou. Také implementace bývá o dost jednodušší a snáze se přidávají různá vylepšení.

Úloha

Navrhněte algoritmus pro mazání mrtvého kódu (co to znamená viz zadání minulé série) pracující s programem v SSA formě.

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 úlohu nejprve vyřešit pro 2 účetní, i za to 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 $O(N!)$ 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 dostane permutaci alfanumerických znaků (zadanou jako řetězec) a vrátí následující permutaci dle 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 `0` $<$ `1` $< \ldots <$ `9` $<$ `a` $< \ldots <$ `z`, a permutace $p_1 p_2 \ldots p_k$ je lexikograficky menší než $q_1 q_2 \ldots q_k$, pokud je $p_1 < q_1$ (podle našeho usp.) 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: Za permutací `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 $O(N)$. Pole A je ale opravdu obrovské, takže na ostatní proměnné si budete muset vystačit pouze s konstantním množstvím paměti (čili $O(1)$). 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 (například jako znak) a ještě 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 $O(N + P)$ (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 $O(N \cdot k)$, 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 $O(N \cdot k + P)$ je daleko hodnotnější než řešení se složitostí $O(N \cdot P)$.

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 spous-

tu papíru a rozhodl se, že když nepronikl do kruhů společenských, pronikne alespoň do tajemství kruhů geometrických.

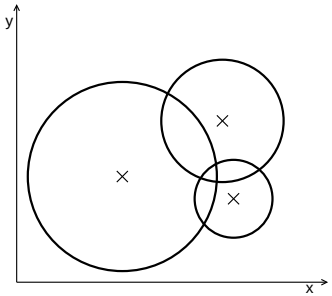
Celé hodiny vysedával za stolem a rýsoval kruhy malé, kruhy větší, kruhy tečné i sečné, a když byl v dobrém rozmaru, dokonce i kruhy soustředné. Jednou si takhle navečer opět hrál s kružítkem a pokreslil celý papír spoustou kružnic. Chtěl si vzít nový papír a starý zahodit, když v tom ho napadlo, že 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 `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 *BB0* vede jediná hrana (s pravděpodobností 100%) do *BB1*. Z *BB1* vedou dvě hrany – jedna z nich se vrací na začátek *BB1* (tato hrana je zvolena s pravděpodobností 90%) a druhá hrana vede do výstupního *BB2* (s pravděpodobností 10%). Tato situace odpovídá programu se smyčkou, která se provede právě desetkrát. V tomto program se *BB0* a *BB2* provedou jedenkrát a *BB1* se provede desetkrát (hrana z *BB1* do *BB1* se provede devětkrát a hrana z *BB1* do *BB2* se provede jedenkrát, tedy jejich pravděpodobnosti opravdu jsou 90% a 10%).

Programátorské kuchařky

16-1-K Kuchařka první série – třídění

I letos Vám kromě úloh budeme servírovat také recepty z programátorské kuchařky. Některé si vypůjčíme z dřívějších ročníků KSP, ale i k těm se budeme snažit připsat něco nového, aby si i zkušenější řešitelé přišli na své. V letošní první kuchařce si povíme o třídících algoritmech. Co to znamená? Pojem *třídění* je možná maličko nepřesný, nehodláme data (čísla, záznamy, řetězce a jiné) rozdělovat do nějakých tříd, ale přerovnat je do správného pořadí, protože se seřazenými údaji se mnohem lépe pracuje, například pokud v nich pak potřebujeme vyhledávat. Takové uspořádávání dat je denním chlebem každého programátora, a tak není divu, že třídící algoritmy jsou jedny z nejstudovanějších. My však nebudeme do nějakých velkých detailů a specialit příliš zabíhat. Zkrátka a dobře – budeme chtít třídit údaje rychle, úsporně a radostně.

Obvykle třídíme exempláře datové struktury typu pascalského záznamu. V takové datové struktuře bývá obsažena jedna význačná položka, *klíč*, podle které se záznamy řadí. Malinko si náš život zjednodušíme a budeme předpokládat, že třídíme záznamy obsahující pouze klíč, který je navíc celočíselný – budeme tedy třídit pole celých čísel. Pomocí počtu tříděných čísel N pak budeme vyjadřovat časovou (a paměťovou) složitost jednotlivých algoritmů, které si předvedeme.

Metody třídění můžeme rozdělit do dvou hlavních skupin, a to na *vnitřní třídění,* kdy si můžeme dovolit všechna data načíst do (rychlé) paměti počítače, a na *vnější třídění,* kdy již třídění musíme realizovat opakovaným čtením a vytvářením diskových souborů. V tomto dílu se omezíme pouze na algoritmy vnitřního třídění a tříděné pole si nadeklarujeme takto:

```
const N = 100;
type Pole = array[1..N] of integer;
```

Nejjednodušší třídící algoritmy patří do skupiny *přímých metod.* Všechny mají několik společných rysů: Jsou krátké, jednoduché a třídí přímo v poli (nepotřebujeme pomocné pole). Tyto algoritmy mají většinou časovou složitost $O(N^2)$. Z toho vyplývá, že jsou použitelné tehdy, když tříděných dat není příliš mnoho. Na druhou stranu pokud je dat opravdu málo, je zbytečně složité používat některý z komplikovanějších algoritmů, které si předvedeme později.

Stručně si přiblížíme tři nejznámější algoritmy pro třídění přímými metodami. *Třídění přímým výběrem (SelectSort)* je založeno na opakovaném vybírání nejmenšího čísla z dosud nesetříděných čísel. Nalezené číslo prohodíme s prvkem na začátku pole a postup opakujeme, tentokrát s nejmenším číslem na indexech $2, \ldots, N$, které prohodíme s druhým prvkem v poli. Poté postup opakujeme s prvky s indexy $3, \ldots, N$, atd. Je snadné si uvědomit, že když takto vybíráme minimum z menších a menších intervalů, setřídíme celé pole (v i-tém kroku nalezneme i-tý nejmenší prvek a zařadíme ho v poli na pozici s indexem i).

```
procedure SelectSort(var A: Pole);
var i,j,k,x: integer;
begin
  for i:=1 to N-1 do
    begin
      k:=i;
      for j:=i+1 to N do
        if A[j]<A[k] then k:=j;
      x:=A[k]; A[k]:=A[i]; A[i]:=x;
    end;
end;
```

Pro úplnost si ještě řekněme pár slov o časové složitosti právě popsaného algoritmu. V i-tém kroku musíme nalézt minimum z $N - i + 1$ čísel, na což spotřebujeme čas $O(N - i + 1)$. Ve všech krocích dohromady tedy spotřebujeme čas $O(N + (N - 1) + \ldots + 3 + 2 + 1) = O(N^2)$.

Třídění přímým vkládáním (InsertSort) funguje na podobném principu jako třídění přímým výběrem. Na začátku pole vytváříme správně utříděnou posloupnost, kterou postupně rozšiřujeme. Na začátku i-tého kroku má tato utříděná posloupnost délku $i - 1$. V i-tém kroku určíme pozici i-tého čísla v dosud utříděné posloupnosti a zařadíme ho do utříděné posloupnosti (zbytek utříděné posloupnosti se posune o jednu pozici doprava). Není těžké si rozmyslet, že každý krok lze provést v čase $O(N)$. Protože počet kroků algoritmu je N, celková časová složitost právě popsaného algoritmu je opět $O(N^2)$.

```
procedure InsertSort(var A: Pole);
var i,j,x: integer;
begin
  for i:=2 to N do
    begin
      x:=A[i]; j:=i-1;
      while (j>0) and (x<A[j]) do
        begin
          A[j+1]:=A[j];
          j:=j-1;
        end;
      A[j+1]:=x;
    end;
end;
```

(Upozornění: v našich příkladech předpokládáme, že máme v překladači zapnuto tzv. zkrácené vyhodnocování logických výrazu, třeba v předchozím while-cyklu se při `j=0` hodnoty `x` a `A[0]` již neporovnávají.)

Bublinkové třídění (BubbleSort) pracuje jinak než dva dříve popsané algoritmy. Algoritmu se říká „bublinkový“, protože podobně jako bublinky v limonádě „stoupají“ vysoká čísla v poli vzhůru. Postupně se porovnávají dvojice sousedních prvků, řekněme zleva doprava, a pokud v porovnávané dvojici následuje menší číslo po větším, tak se tato dvě čísla prohodí. Celý postup opakujeme, dokud probíhají nějaké výměny. Protože algoritmus skončí, když nedojde k žádné výměně, je pole na konci algoritmu setříděné.

```
procedure BubbleSort(var A: Pole);
var i,x: integer;
    zmena: boolean;
begin
  repeat
    zmena:=false;
    for i:=1 to N-1 do
      if A[i] > A[i+1] then
        begin
          x:=A[i]; A[i]:=A[i+1]; A[i+1]:=x;
          zmena:=true;
        end;
  until not zmena;
end;
```

Správnost algoritmu nahlédneme tak, že si uvědomíme, že po i průchodech while-cyklem bude posledních i prvků obsahovat největších i prvků setříděných od nejmenšího po největší (rozmyslete si, proč tomu tak je). Popsaný algoritmus se tedy zastaví po nejvýše N průchodech a jeho celková časová složitost v nejhorším případě je $O(N^2)$, neboť na každý průchod spotřebuje čas $O(N)$. Výhodou tohoto algoritmu oproti předchozím dvěma algoritmům je, že pokud je pole na začátku setříděné, tak algoritmus spotřebuje jen lineární čas, $O(N)$.

Lepší třídící algoritmy pracují v čase $O(N \log N)$. Jedním z nich je *Třídění slévánim (MergeSort),* založené na principu slévání (spojování) již setříděných posloupností dohromady. Představme si, že již máme dvě setříděné posloupnosti a chceme je spojit dohromady. Jednoduše stačí porovnávat nejmenší prvek z každé posloupnosti, který jsme dosud nedali do nově vytvářené posloupnosti, a menší z těchto prvků do nové posloupnosti přidat. Je zřejmé, že ke slití dvou posloupností potřebujeme čas úměrný součtu jejich délek.

My si zde popíšeme a předvedeme modifikaci algoritmu MergeSort, která používá pomocné pole. Algoritmus lze implementovat při zachování časové složitosti i bez pomocného pole, ale je to o dost pracnější. Existuje též modifikace algoritmu, která má počet fází (viz dále) v nejhorším případě $O(\log N)$, ale po-

kud je již pole na začátku setříděné, proběhne pouze jediná a v takovém případě má algoritmus časovou složitost $O(N)$. My si však zatajíme i tuto variantu.

Algoritmus pracuje v několika *fázích.* Na začátku první fáze tvoří každý prvek jednoprvkovou setříděnou posloupnost a obecně na začátku i-té fázi budou mít setříděné posloupnosti délky 2^{i-1}. V i-té fázi tedy vždy ze dvou sousedních 2^{i-1}-prvkových posloupností vytvoříme jedinou délky 2^i. Pokud N není násobkem 2^i, bude délka poslední posloupnosti zbytek po dělení N číslem 2^i. Zastavíme se, pokud $2^i \geq N$, tj. po $\lceil \log_2 N \rceil$ fázích. Protože v i-té fázi slijeme $\lceil N/2^i \rceil$ dvojic nejvýše 2^{i-1}-prvkových posloupností, je časová složitost jedné fáze $O(N)$. Celková časová složitost popsaného algoritmu je pak $O(N \log N)$.

```
procedure MergeSort(var A: Pole);
var P: Pole;        { pomocné pole }
    delka:integer;  { délka setříděných posl. }
    i: integer;     { index do vytvářené posl. }
    i1,i2: integer; { index do slévaných posl. }
    k1,k2: integer; { konce slévaných posl. }
begin
  delka:=1;
  while delka<N do
    begin
      i1:=1; i2:=delka+1; i:=1;
      k1:=delka; k2:=2*delka;
      while i<=N do
        begin
          if k2>N then k2:=N;
          while (i1<=k1) or (i2<=k2) do
            if (i1>k1) or
               ((i2<=k2) and (A[i1]<=A[i2]))
            then
              begin
                P[i]:=A[i1]; i:=i+1; i1:=i1+1;
              end
            else
              begin
                P[i]:=A[i2]; i:=i+1; i2:=i2+1;
              end;
          i1:=k2+1; i2:=k2+delka;
          k1:=k2+delka;
          k2:=k2+2*delka;
        end;
      A:=P;
      delka:=2*delka;
    end;
end;
```

V čase $O(N \log N)$ pracuje také algoritmus jménem *QuickSort.* Tento algoritmus je založen na metodě Rozděl a panuj. Nejprve si zvolíme nějaké číslo, kterému budeme říkat *pivot.* Více si o jeho volbě povíme později. Poté pole

přeuspořádáme a rozdělíme je na dvě části tak, že žádný prvek v první části nebude větší než pivot a žádný prvek v druhé části naopak menší. Prvky v obou částech pak setřídíme rekurzivním zavoláním téhož algoritmu. Musíme ale dát pozor, aby v každém kroku obě části byly neprázdné (a rekurze tedy byla konečná). Je zřejmé, že po skončení algoritmu bude pole setříděné.

Malá zrada spočívá ve volbě pivota. Pro naše účely by se hodilo, aby po přeházení prvků levá i pravá část pole byly přibližně stejně velké. Nejlepší volbou pivota by tedy byl *medián* tříděného úseku, tj. prvek takový, jenž by byl v setříděném poli přesně uprostřed. Přeuspořádání jistě zvládneme v lineárním čase a pokud by pivoty na všech úrovních byly mediány, pak by počet úrovní rekurze byl $O(\log N)$ a celková časová složitost $O(N \log N)$ (na každé úrovni rekurze je součet délek tříděných posloupnosti nejvýše N). Ačkoli existuje algoritmus, který medián pole nalezne v čase $O(N)$, v QuickSortu se obvykle nepoužívá, jelikož konstanta u členu N je příliš velká v porovnání s pravděpodobností, že náhodná volba pivota algoritmus příliš zpomalí. Většinou se pivot volí náhodně z dosud nesetříděného úseku – zkrátka se sáhne někam do pole a nalezený prvek se prohlásí za pivot. Dá se ukázat, že takovýto algoritmus s velmi vysokou pravděpodobností poběží v čase $O(N \log N)$. Důkaz tohoto tvrzení je trošičku trikový a lze jej nalézt např. v knize Kapitoly z diskrétní matematiky od pánů Matouška a Nešetřila. Je však třeba si pamatovat, že pokud se pivot volí náhodně, může rekurze dosáhnout hloubky N a časová složitost algoritmu až $O(N^2)$ – představme si, že se pivot v každém rekurzivním volání nešťastně zvolí jako největší prvek z tříděného úseku. V naší implementaci QuickSortu nebudeme pivot volit náhodně, ale vždy použijeme prostřední prvek tříděného úseku.

```
procedure QuickSort(var A:Pole; l,r:integer);
var i,j,k,x: integer;
begin
  i:=l; j:=r;
  k:=A[(i+j) div 2];
  repeat
    while A[i]<k do i:=i+1;
    while A[j]>k do j:=j-1;
    if i<=j then
      begin
        x:=A[i]; A[i]:=A[j]; A[j]:=x;
        i:=i+1;
        j:=j-1;
      end;
  until i >= j;
  if j>l then QuickSort(A, l, j);
  if i<r then QuickSort(A, i, r);
end;
```

Ještě si předvedeme dva třídící algoritmy, které jsou vhodné, pokud tříděné objekty mají některé další speciální vlastnosti. Prvním z nich je *třídění počítáním (CountSort).* To lze použít, pokud tříděné objekty obsahují pouze klíče a možných hodnot klíčů je málo. Tehdy stačí si spočítat, kolikrát se který klíč vyskytuje, a místo třídění vytvořit celé pole znovu na základě toho, kolik jednotlivých objektů obsahovalo pole původní. My si tento algoritmus předvedeme na příkladu třídění pole celých čísel z intervalu $\langle D, H \rangle$:

```
const D = 1;
      H = 10;
procedure CountSort(var A: Pole);
var C: array[D..H] of integer;
    i,j,k: integer;
begin
  for i:=D to H do C[i]:=0;
  for i:=1 to N do C[A[i]]:=C[A[i]] + 1;
  k:=1;
  for i:=D to H do
    for j:=1 to C[i] do
      begin
        A[k]:=i;
        k:=k+1;
      end;
end;
```

Časová složitost takovéhoto algoritmu je lineární v N, ale nesmíme zapomenout přičíst ještě velikost intervalu, ve kterém se prvky nacházejí ($K = H - D + 1$), protože nějaký čas spotřebujeme i na inicializaci pole počítadel. Celkem tedy $O(N + K)$.

Pokud by tříděné objekty obsahovaly vedle klíčů i nějaká data, můžeme je místo pouhého počítání rozdělovat do přihrádek podle hodnoty klíče a pak je z přihrádek vysbírat v rostoucím pořadí klíčů. Tomuto algoritmu se říká *přihrádkové třídění (BucketSort)* a my si popíšeme jeho víceprůchodovou variantu *(RadixSort),* která je vhodnější pro větší hodnoty K. V první fázi si čísla rozdělíme do přihrádek (skupin) podle nejméně významné cifry a spojíme do jedné posloupnosti, v druhé fázi čísla roztřídíme podle druhé nejméně významné cifry a opět spojíme do jedné posloupnosti, atd. Je důležité, aby se uvnitř každé přihrádky zachovalo pořadí čísel v posloupnosti na začátku fáze, tj. posloupnost uložená v každé přihrádce je vybranou podposloupností posloupnosti ze začátku fáze. Tvrdíme, že na konci i-té fáze obsahuje výsledná posloupnost čísla utříděná podle i nejméně významných cifer. Zřejmě i-té nejméně významné cifry tvoří rostoucí posloupnost, neboť podle nich jsme právě v této fázi rozdělovali čísla do přihrádek, a pokud dvě čísla mají tuto cifru stejnou, jsou uložena v pořadí dle jejich $i - 1$ nejméně významných cifer, neboť v každé přihrádce jsme zachovali pořadí čísel z konce minulé fáze. Na závěr ještě poznamenejme,

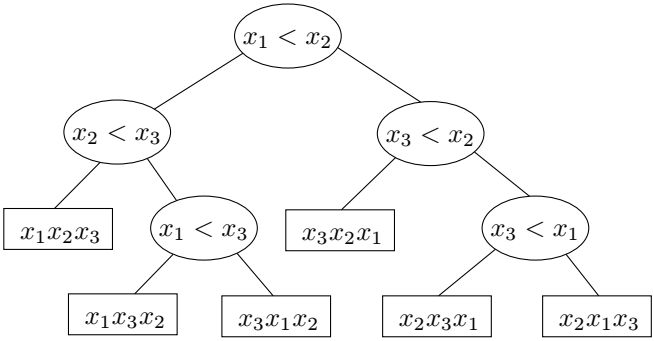
že místo čísel podle cifer lze do přihrádek rozdělovat též textové řetězce podle jejich znaků, atp.

Časová složitost této varianty RadixSortu, pokud třídíme celá čísla od 1 do K a v každém kroku je rozdělujeme do ℓ přihrádek, je $O((N+\ell)\log_\ell K)$, tedy $O(N)$, pokud K a ℓ jsou konstanty. My si předvedeme implementaci algoritmu pro $K = 255$ a $\ell = 2$ (čísla budeme roztřiďovat podle bitů v jejich binárním zápisu).

```
const K=255;
procedure RadixSort(var A: Pole);
var P0,P1: Pole;
    k1,k2: integer;
    i: integer;
    bit: integer;
begin
  bit:=1;
  while bit<=K do
    begin
      k1:=0; k2:=0;
      for i:=1 to N do
        if (A[i] and bit)=0 then
          begin
            k1:=k1+1; P0[k1]:=A[i];
          end
        else
          begin
            k2:=k2+1; P1[k2]:=A[i];
          end;
      for i:=1 to k1 do A[i]:=P0[i];
      for i:=1 to k2 do A[k1+i]:=P1[i];
      bit:=bit shl 1;
    end;
end;
```

Na závěr našeho povídání o třídících algoritmech si ukážeme, že třídit obecné údaje, se kterými neumíme provádět nic jiného, než je navzájem porovnávat, rychleji než $O(N \log N)$ nejen nikdo neumí, ale také ani umět nemůže. Libovolný třídící algoritmus založený na porovnávání a prohazování prvků totiž musí na některé vstupy vynaložit řádově alespoň $N \log N$ kroků. (RadixSort na první pohled tento výsledek porušuje, na druhý však už ne, když si uvědomíme, o jak speciální druh tříděných dat se jedná.)

Třídící algoritmus v průběhu své činnosti nějak porovnává prvky a nějak je přehazuje. Provedeme myšlenkový experiment. Pozměníme algoritmus tak, že nejdříve bude pouze porovnávat, podle toho zjistí, jak jsou prvky v poli uspořádány, a když už si je jistý správným pořadím, prvky najednou popřehází. Předpokládejme pro jednoduchost, že všechny tříděné údaje jsou navzájem různé. Porovnávací činnost algoritmu si můžeme popsat tzv. *rozhodovacím stromem.* Zde je příklad rozhodovacího stromu pro tříprvkové pole:



Každý vrchol obsahuje porovnání dvou prvků x a y, v levém podstromu daného vrcholu je činnost algoritmu pokud $x < y$, v pravém podstromu činnost při $x \geq y$. V listech je už jisté správné pořadí prvků.

Každému algoritmu odpovídá nějaký rozhodovací strom a každý průběh činnosti algoritmu odpovídá průchodu rozhodovacím stromem od kořene do nějakého listu. Naším cílem bude ukázat, že v libovolném rozhodovacím stromu (a tedy i libovolném odpovídajícím algoritmu) bude existovat cesta z kořene do nějakého listu (neboli výpočet algoritmu) délky $N \log N$.

Kolik maximálně hladin h, a tedy i jaká nejdelší cesta se v takovém stromu může vyskytnout? Náš strom má tolik listů, kolik je možných pořadí tříděných prvků, tedy právě $N!$. Různým pořadím totiž musí odpovídat různé listy, jinak by algoritmus netřídil (předpokládáme přeci, že to, jak má prvky prohazovat, může zjistit jenom jejich porovnáváním), a naopak každé pořadí prvků jednoznačně určuje cestu do příslušného listu. Na nulté hladině je jediný vrchol, na každé další hladině se oproti předchozí počet vrcholů nejvýše zdvojnásobí, takže na i-té hladině se nachází nejvýše 2^i vrcholů. Proto je listů stromu nejvýše 2^h (některé listy mohou být i výše, ale za každý takový určitě chybí jeden vrchol na h-té hladině). Z toho víme, že platí:

$$2^h \geq \textit{počet listů} \geq N!,$$

a proto:

$$h \geq \log_2(N!).$$

Logaritmus faktoriálu se těžko počítá přesně, ale můžeme si ho zdola odhadnout pomocí následující známé nerovnosti:

$$n^n \geq n! \geq n^{n/2}.$$

Dosazením získáme:

$$h \geq \log_2(N!) \geq \log_2(N^{N/2}) \geq \frac{N}{2} \log_2 N.$$

Vidíme tedy, že pro každý třídící algoritmus existuje vstup, na kterém se bude muset provést alespoň $N \log N$ kroků.

Poznámky na okraj:

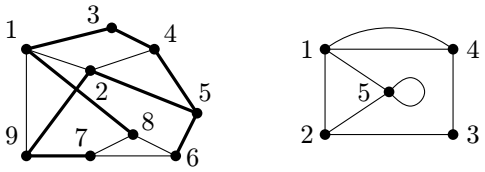
- Zkuste si též rozmyslet (drobnou modifikací předchozího důkazu), že ani *průměrný* čas třídění nemůže být lepší než $N \log N$.
- Odvodit průměrnou složitost QuickSortu vlastně není zase tak  těžké. Zkusme následující úvahu: Pokud by pivot nebyl přesně medián, ale alespoň se nacházel v prostřední třetině setříděného úseku, byla by složitost stále $O(N \log N)$, jen by se zvýšila konstanta v O-čku. Kdybychom pivot volili náhodně, ale po rozdělení prvků si zkontrolovali, jestli pivot padl do prostřední třetiny, a pokud ne (jeden z úseků by byl moc velký a druhý moc malý), volbu bychom opakovali, v průměru by nás to stálo konstantní počet pokusů (pozorování z řešení úlohy 16-1-5: pokud čekáme na událost, která nastává náhodně s pravděpodobností p, stojí nás to v průměru $1/p$ pokusů; zde je $p = 1/3$), takže celková složitost by v průměru vzrostla jen konstantně. Původní QuickSort sice žádné takové opakování volby neprovádí a rovnou se zavolá rekurzivně na velký i malý úsek, ale opět se po v průměru konstantně mnoha iteracích velký úsek zredukuje na nejvýše 2/3 původní velikosti a třídění malých úseků jednotlivě nezabere víc času, než kdyby se třídily dohromady.
- Kdybychom u QuickSortu použili rekurzivní volání jen na menší interval, zatímco ten větší bychom obsloužili přenastavením proměnných a skokem na začátek právě prováděné procedury, zredukovali bychom paměťovou složitost na $O(\log N)$, jelikož každé další rekurzivní volání zpracovává alespoň dvakrát menší úsek než to předchozí. Časové složitosti tím však nepomůžeme.
- Počet přihrádek u RadixSortu vůbec nemusí být konstanta – pokud např. chcete třídit N čísel v rozsahu $1 \ldots N^k$, stačí si zvolit $\ell = N$ a fází bude jenom k. Pro pevné k tak dosáhneme lineární časové složitosti.
- Nerovnost $n! \geq n^{n/2}$, kterou jsme použili v dolním odhadu složitosti třídění, můžeme dokázat snadným trikem: $n! = \sqrt{1^2 \cdot 2^2 \cdot \ldots \cdot n^2} = \sqrt{n \cdot 1} \cdot \sqrt{(n-1) \cdot 2} \cdot \ldots \cdot \sqrt{2 \cdot (n-1)} \cdot \sqrt{1 \cdot n} \geq \sqrt{n} \cdot \sqrt{n} \cdot \ldots \cdot \sqrt{n} = (n^{1/2})^n = n^{n/2}$. Pokud nevidíte, proč $\geq$, uvažte, že výraz pod odmocninou je tvaru $(n-k)(k+1) = nk + n - k^2 - k = n + k(n-k-1)$ a poslední závorka je pro $0 \leq k < n$ a $n \geq 1$ vždy nezáporná.

16-2-K Kuchařka druhé série – grafy

V dnešním vydání známého bestselleru budeme péci grafy souvislé i nesouvislé, orientované i neorientované.Řekneme si o základním procházení grafem, komponentách souvislosti, topologickém uspořádání a dalších grafových algoritmech. Abychom ale mohli začít, musíme si nejprve říci, s čím budeme pracovat.

Ingredience

Neorientovaný graf je určen množinou vrcholů V a množinou hran E, což jsou neuspořádané dvojice vrcholů. Hrana $e = \{x, y\}$ spojuje vrcholy x a y. Většinou požadujeme, aby hrany nespojovaly vrchol se sebou samým (takovým hranám říkáme *smyčky*) a aby mezi dvěma vrcholy nevedla více než jedna hrana (pokud toto neplatí, mluvíme o *multigrafech*). Obvykle také předpokládáme, že vrcholů je konečně mnoho. Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a multigraf

Podgrafem grafu G rozumíme graf G', který vznikl z grafu G vynecháním některých (a nebo žádných) hran a vrcholů.

Často nás zajímá, zda se dá z vrcholu x dojít po hranách do vrcholu y. Ovšem slovo „dojít“ by mohlo být trochu zavádějící, proto si zavedeme pár pojmů:

- *sled* je posloupnost vrcholů a hran $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$ taková, že $e_i = \{v_i, v_{i+1}\}$ pro každé i. Sled je tedy nějaká procházka po grafu. Délku sledu měříme počtem hran v této posloupnosti.
- *tah* je sled, ve kterém se neopakují hrany, tedy $e_i \neq e_j$ pro $i \neq j$.
- *cesta* je sled, ve kterém se neopakují vrcholy, čili $v_i \neq v_j$ pro $i \neq j$. Všimněte si, že se nemohou opakovat ani hrany.

Lehce nahlédneme, že pokud existuje sled z vrcholu x do y ($v_1 = x$, $v_n = y$), pak také existuje cesta z vrcholu x do vrcholu y. Každý sled, který není cestou, obsahuje nějaký vrchol u dvakrát, nechť $u = v_i = v_j$, $i < j$. Z takového sledu ale můžeme vypustit posloupnost $e_i, v_{i+1}, \ldots, e_{j-1}, v_j$ a dostaneme také sled spojující v_1 a v_n, který je určitě kratší než původní sled. Tak můžeme po konečném počtu úprav dospět až ke sledu, který neobsahuje žádný vrchol dvakrát, tedy k cestě.

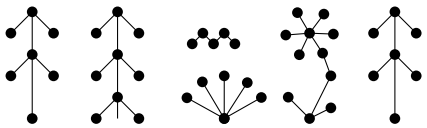
Kružnicí neboli *cyklem* nazýváme cestu délky alespoň 3, ve které oproti definici cesty platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

Ještě si ukážeme, že pokud existuje cesta z vrcholu a do vrcholu b a z vrcholu b do vrcholu c, pak také existuje cesta z vrcholu a do vrcholu c. To vyplývá z faktu, že existuje sled z vrcholu a do vrcholu c, který můžeme dostat například tak, že spojíme za sebe cesty z a do b a z b do c. A jak jsme si ukázali, když existuje sled z a do c, existuje i cesta z a do c.

V mnoha grafech (například v těch na předchozím obrázku) je každý vrchol dosažitelný cestou z každého. Takovým grafům budeme říkat *souvislé.* Pokud je graf nesouvislý, můžeme ho rozložit na části, které již souvislé jsou a mezi kterými nevedou žádné další hrany. Takové podgrafy nazýváme *komponentami souvislosti.*

Teď se podívejme na pár pojmů z přírody: *Strom* je souvislý graf, který neobsahuje kružnici. *List* je vrchol, ze kterého vede pouze jedna hrana. Ukážeme, že každý strom s alespoň dvěma vrcholy má nejméně dva listy. Proč to? Stačí si najít nejdelší cestu (pokud je takových cest více, zvolíme libovolnou z nich). Oba koncové vrcholy této cesty musí být nutně listy: kdyby z některého z nich vedla hrana, musela by vést do vrcholu, který na cestě ještě neleží (jinak by ve stromu byla kružnice), ale o takovou hranu bychom cestu mohli prodloužit, takže by původní cesta nebyla nejdelší.

Grafům bez kružnic budeme obecně říkat *lesy,* jelikož každá komponenta souvislosti takového grafu je strom.



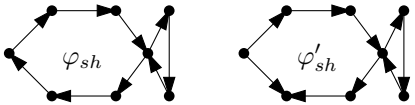
Les, jak ho vidí matematici

Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale matematici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny.*

Kostra souvislého grafu říkáme každému jeho podgrafu, který je stromem a spojuje všechny vrcholy grafu. Můžeme ji například získat tak, že dokud jsou v grafu kružnice, odebíráme hrany ležící na nějaké kružnici. Pro nesouvislé grafy nazveme kostrou les tvořený kostrami jednotlivých komponent. Na prvním obrázku je jedna z koster levého grafu znázorněna silnými hranami.

Orientované grafy

Často potřebujeme, aby hrany byly pouze jednosměrné. Takovému grafu říkáme *orientovaný graf.* Hrany jsou nyní uspořádané dvojice vrcholů (x, y) a říkáme, že hrana vede z vrcholu x do vrcholu y. Hrany (x, y) a (y, x) jsou tedy dvě různé hrany. Orientovaný graf většinou zobrazujeme jako body spojené šipkami. Většina pojmů, které jsme definovali pro neorientované grafy, dává smysl i pro grafy orientované, jen si musíme dát pozor na směr hran.



Silně a slabě souvislý orientovaný graf

Se souvislostí orientovaných grafů je to trochu složitější. Rozlišujeme slabou a silnou souvislost: *slabě souvislý* je graf tehdy, pokud se z něj zapomenutím orientace hran stane souvislý neorientovaný graf. *Silně souvislým* ho nazveme tehdy, vede-li mezi každými dvěma vrcholy x, y orientovaná cesta v obou směrech. Pokud je graf silně souvislý, je i slabě souvislý, ale jak ukazuje náš obrázek, opačně to platit nemusí.

Komponenta silné souvislosti orientovaného grafu G je takový podgraf G', který je silně souvislý a není podgrafem žádného většího silně souvislého podgrafu grafu G. Komponenty silné souvislosti tedy mohou být mezi sebou propojeny, ale žádné dvě nemohou ležet na společném cyklu.

Ohodnocené grafy

Další možností, jak si graf „vyzdobit“, je ohodnotit jeho hrany čísly. Například v grafu silniční sítě (vrcholy jsou města, hrany silnice mezi nimi) je zcela přirozené ohodnotit hrany délkami silnic nebo třeba mýtným vybíraným za průjezd silnicí. Přiřazeným číslům se proto často říká *délky* hran nebo jejich *ceny.* Pojmy, které jsme si před chvílí nadefinovali pro obyčejné grafy, můžeme opět snadno rozšířit pro grafy ohodnocené – např. délku sledu budeme namísto počtu hran sledu počítat jako součet jejich ohodnocení. Neohodnocený graf pak odpovídá grafu, v němž mají všechny hrany jednotkovou délku.

Podobně můžeme přiřazovat ohodnocení i vrcholům, ale raději si všechny operace s ohodnocenými grafy necháme na některé z dalších dílů Kuchařky. I tak budeme mít práce dost a dost.

Reprezentace grafů

Nyní už víme o grafech hodně, ale ještě jsme si neřekli, jak graf reprezentovat v paměti počítače. To můžeme udělat například tak, že vrcholy očíslujeme přirozenými čísly od 1 do N, hrany od 1 do M a odkud kam vedou hrany, popíšeme jedním z následujících tří způsobů:

- *matice sousednosti* – to je pole A velikosti $N \times N$. Na pozici $A[i,j]$ uložíme hodnotu 0 nebo 1 podle toho, zda z vrcholu i do vrcholu j vede hrana (1) nebo nevede (0). S maticí sousednosti se zachází velmi snadno, ale má tu nevýhodu, že je vždy kvadraticky velká bez ohledu na to, kolik je hran. Výhodou naopak je, že místo jedniček můžeme ukládat nějaké další informace o hranách, třeba jejich délky. Vpravo od tohoto odstavce najdete matici sousednosti grafu z prvního obrázku.

```
  123456789
1 011000011
2 100110001
3 100100000
4 011010000
5 010101000
6 000010110
7 000001011
8 100001100
9 110000100
```

- *seznam sousedů* je obvykle tvořen 2 poli: polem sousedů $S[1 \ldots M]$ obsahujícím postupně čísla všech vrcholů, do kterých vede hrana z vrcholu 1, pak z vrcholu 2 atd., a polem začátků $Z[1 \ldots N]$, v němž se pro každý vrchol dozvíme začátek odpovídajícího úseku v poli S. Pokud navíc do $Z[N+1]$ uložíme $M+1$, bude platit, že sousedé vrcholu i jsou uloženi v $S[Z[i]], \ldots, S[Z[i+1]-1]$. Tato reprezentace má tu výhodu, že zabírá pouze prostor $O(N+M)$ a sousedy každého vrcholu máme pěkně pohromadě a nemusíme je hledat. Pro graf z 1. obrázku:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$S[i]$	2	3	8	9	1	4	5	9	1	4	2	3	5	2	4	6	5	7	8	6	8	9	1	6	7	1	2	7

i	1	2	3	4	5	6	7	8	9	10
$Z[i]$	1	5	9	11	14	17	20	23	26	29

Reprezentace grafu seznamem sousedů

- *půlhranami* – tato reprezentace se používá tehdy, pokud potřebujeme během výpočtu graf složitě upravovat. Je univerzální, ale dost pracná na naprogramování. Spočívá v tom, že si každou hranu uložíme jako dvě půlhrany (začátek a konec hrany), každý vrchol bude obsahovat spojové seznamy přicházejících a odcházejících půlhran a každá půlhrana bude ukazovat na svou druhou polovici.

V následujících receptech budeme vždy používat seznamy sousedů, poli S budeme říkat `Sousedi`, poli Z `Zacatky` a nadeklarujeme si je takto:

```
var N, M: Integer;  { Počet vrcholů a hran }
    Zacatky: array[1..MaxN+1] of Integer;
    Sousedi: array[1..MaxM] of Integer;
```

Prohledávání do hloubky

Naše povídání o grafových algoritmech začneme dvěma základními způsoby procházení grafem. K tomu budeme potřebovat dvě podobné jednoduché datové struktury: *Fronta* je konečná posloupnost prvků, která má označený začátek

a konec. Když do ní přidáváme nový prvek, přidáme ho na konec posloupnosti. Když z ní prvek odebíráme, odebereme ten na začátku. Proto se tato struktura anglicky nazývá *first in, first out*, zkráceně *FIFO*. *Zásobník* je také konečná posloupnost prvků se začátkem a koncem, ale prvky přidáváme a odebíráme z konce zásobníku. Anglický název je (překvapivě) *last in, first out*, čili *LIFO*.

Algoritmus prohledávání grafu do hloubky:

1. Na začátku máme v zásobníku pouze vstupní vrchol w. Dále si u každého vrcholu v pamatujeme značku z_v, která říká, zda jsme vrchol již navštívili. Vstupní vrchol je označený, ostatní vrcholy nikoliv.
2. Odebereme vrchol ze zásobníku, nazvěme ho u.
3. Každý neoznačený vrchol, do kterého vede hrana z u, přidáme na zásobník a označíme.
4. Kroky 2 a 3 opakujeme, dokud není zásobník prázdný.

Na konci algoritmu budou označeny všechny vrcholy dosažitelné z vrcholu w, tedy v případě neorientovaného grafu celá komponenta souvislosti obsahující w. To můžeme snadno dokázat sporem: Předpokládáme, že existuje vrchol x, který není označen, ale do kterého vede cesta z w. Pokud je takových vrcholů více, vezmeme si ten nejbližší k w. Označme si y předchůdce vrcholu x na nejkratší cestě z w; y je určitě označený (jinak by x nebyl nejbližší neoznačený). Vrchol y se tedy musel někdy objevit na zásobníku, tím pádem jsme ho také museli ze zásobníku odebrat a v kroku 3 označit všechny jeho sousedy, tedy i vrchol x, což je ovšem spor.

To, že algoritmus někdy skončí, nahlédneme snadno: v kroku 3 na zásobník přidáváme pouze vrcholy, které dosud nejsou označeny, a hned je značíme. Proto se každý vrchol může na zásobníku objevit nejvýše jednou, a jelikož ve 2. kroku pokaždé odebereme jeden vrchol ze zásobníku, musí vrcholy někdy (konkrétně po nejvýše N opakováních cyklu) dojít. Ve 3. kroku probereme každou hranu grafu nejvýše dvakrát (v každém směru jednou). Časová složitost celého algoritmu je tedy lineární v počtu vrcholů N a počtu hran M, tedy $O(N + M)$. Paměťová složitost je stejná, protože si musíme hrany a vrcholy pamatovat.

Prohledávání do hloubky implementujeme nejsnáze rekurzivní funkcí. Jako zásobník v tom případě používáme přímo zásobník programu, kde si program ukládá návratové adresy funkcí. Může to vypadat třeba následovně:

```
var Oznacen: array[1..MaxN] of Boolean;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Oznacen[V] := True;
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if not Oznacen[Sousedi[I]] then
      Projdi(Sousedi[I]);
end;
```

Rozdělit neorientovaný graf na komponenty souvislosti je pak už jednoduché. Projdeme postupně všechny vrcholy grafu a pokud nejsou v žádné z dosud označených komponent grafu, přidáme novou komponentu tak, že graf z tohoto vrcholu prohledáme do hloubky. Vrcholy značíme přímo číslem komponenty, do které patří. Protože prohledáváme do hloubky několik oddělených částí grafu, každou se složitostí $O(N_i + M_i)$, kde N_i a M_i je počet vrcholů a hran komponenty, vyjde dohromady složitost $O(N + M)$. Nic nového si ukládat nemusíme, a proto je paměťová složitost stále $O(N + M)$.

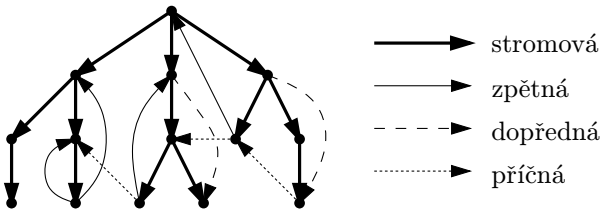
```
var Komponenta: array[1..MaxN] of Integer;
    NovaKomponenta: Integer;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Komponenta[V] := NovaKomponenta;
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if Komponenta[Sousedi[I]] = -1 then
      Projdi(Sousedi[I]);
end;

var I: Integer;
begin
  ...
  for I := 1 to N do Komponenta[I] := -1;
  NovaKomponenta := 1;
  for I := 1 to N do
    if Komponenta[I] = -1 then
      begin
        Projdi(I);
        Inc(NovaKomponenta);
      end;
  ...
end.
```

Průběh prohledávání grafu do hloubky můžeme znázornit stromem (říká se mu DFS strom). Z počátečního vrcholu w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslovat jako syny vrcholů, ze kterých jsme přišli. Syny každého vrcholu si uspořádáme v pořadí, v němž jsme je navštívili; tomuto pořadí budeme říkat *zleva doprava* a také ho tak budeme kreslit. Hranám mezi otci a syny budeme říkat *stromové hrany*. Protože jsme do žádného vrcholu nešli dvakrát, budou opravdu tvořit strom. Hrany, které vedou do již navštívených vrcholů na cestě, kterou jsme přišli z kořene, nazveme *zpětné hrany. Dopředné hrany* vedou naopak z vrcholu blíže kořeni do už označeného vrcholu dále od kořene. A konečně *příčné hrany* vedou mezi dvěma různými podstromy grafu.

Všimněte si, že při prohledávání neorientovaného grafu objevíme každou hranu dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, a nebo jednou jako zpětnou a podruhé jako dopřednou. Příčné hrany se objevit nemohou – pokud by příčná hrana vedla doprava, vedla by do dosud neoznačeného vrcholu, takže by se prohledávání vydalo touto hranou a nevznikl by oddělený podstrom; doleva rovněž vést nemůže: představme si stav prohledávání v okamžiku, kdy jsme opouštěli levý vrchol této hrany. Tehdy by naše hrana musela být příčnou vedoucí doprava, ale o té už víme, že neexistuje.

Prohledávání do hloubky lze tedy také využít k nalezení kostry neorientovaného grafu, což je strom, který jsme prošli. Rovnou při tom také zjistíme, zda graf neobsahuje cyklus: to poznáme tak, že nalezneme zpětnou hranu různou od té stromové, po níž jsme do vrcholu přišli.

Pro orientované grafy je situace opět trochu složitější: stromové a dopředné hrany jsou orientované vždy ve stromě shora dolů, zpětné zdola nahoru a příčné hrany mohou existovat, ovšem vždy vedou zprava doleva, čili pouze do podstromů, které jsme již prošli (nahlédneme opět stejně).



Strom prohledávání do hloubky a typy hran

Prohledávání do šířky

Prohledávání do šířky je založené na podobné myšlence jako prohledávání do hloubky, pouze místo zásobníku používá frontu:

1. Na začátku máme ve frontě pouze jeden prvek, a to zadaný vrchol w. Dále si u každého vrcholu x pamatujeme číslo $H[x]$. Všechny vrcholy budou mít na začátku $H[x] = -1$, jen $H[w] = 0$.
2. Odebereme vrchol z fronty, označme ho u.
3. Každý vrchol v, do kterého vede hrana z u a jeho $H[v] = -1$, přidáme do fronty a nastavíme jeho $H[v]$ na $H[u] + 1$.
4. Kroky 2 a 3 opakujeme, dokud není fronta prázdná.

Podobně jako u prohledávání do hloubky jsme se dostali právě do těch vrcholů, do kterých vede cesta z w (a označili jsme je nezápornými čísly). Rovněž je každému vrcholu přiřazeno nezáporné číslo maximálně jednou. To vše se dokazuje podobně, jako jsme dokázali správnost prohledávání do hloubky.

Vrcholy se stejným číslem tvoří ve frontě jeden souvislý celek, protože nejprve odebereme z fronty všechny vrcholy s číslem n, než začneme odebírat vrcholy s číslem $n+1$. Navíc platí, že $H[v]$ udává délku nejkratší cesty z vrcholu w do v. Že neexistuje kratší cesta, dokážeme sporem: Pokud existuje nějaký vrchol v, pro který $H[v]$ neodpovídá délce nejkratší cesty z w do v, čili vzdálenosti $D[v]$, vybereme si z takových v to, jehož $D[v]$ je nejmenší. Pak nalezneme nejkratší cestu z w do v a její předposlední vrchol z. Vrchol z je bližší než v, takže pro něj už musí být $D[z] = H[z]$. Ovšem když jsme z fronty vrchol z odebírali, museli jsme objevit i jeho souseda v, který ještě nemohl být označený, tudíž jsme mu museli přidělit $H[v] = H[z] + 1 = D[v]$, a to je spor.

Prohledávání do šířky má časovou složitost taktéž lineární s počtem hran a vrcholů. Na každou hranu se také ptáme dvakrát. Fronta má lineární velikost k počtu vrcholů, takže jsme si oproti prohledávání do hloubky nepohoršili a i paměťová složitost je $O(N+M)$. Algoritmus implementujeme nejsnáze cyklem, který bude pracovat s vrcholy v poli představujícím frontu.

```
var Fronta, H: array[1..MaxN] of Integer;
    I, V, Prvni, Posledni: Integer;
    PocatecniVrchol: Integer;
begin
  ...
  for I := 1 to N do H[I] := -1;
  Prvni := 1;  Posledni := 1;
  Fronta[Prvni] := PocatecniVrchol; H[PocatecniVrchol] := 0;
  repeat
    V := Fronta[Prvni];
    for I := Zacatky[V] to Zacatky[V+1]-1 do if H[Sousedi[I]] < 0 then begin
      H[Sousedi[I]] := H[V]+1;
      Inc(Posledni);
      Fronta[Posledni] := Sousedi[I];
    end;
    Inc(Prvni);
  until Prvni > Posledni;  { Fronta je prázdná }
end.
```

Prohledávání do šířky lze také použít na hledání komponent souvislosti a hledání kostry grafu.

Topologické uspořádání

Teď si vysvětlíme, co je *topologické uspořádání* grafu. Máme orientovaný graf G s N vrcholy a chceme očíslovat vrcholy čísly 1 až N tak, aby všechny hrany vedly z vrcholu s větším číslem do vrcholu s menším číslem, tedy aby pro každou hranu $e = (v_i, v_j)$ bylo $i > j$. Představme si to jako srovnání vrcholů grafu na přímku tak, aby „šipky“ vedly pouze zprava doleva.

Nejprve si ukážeme, že pro žádný orientovaný graf, který obsahuje cyklus, nelze takovéto topologické pořadí vytvořit. Označme vrcholy cyklu $v_1, \ldots, v_n$, takže hrana vede z vrcholu v_i do vrcholu v_{i-1}, resp. z v_1 do v_n. Pak vrchol v_2 musí dostat vyšší číslo než vrchol v_1, v_3 než $v_2, \ldots, v_n$ než v_{n-1}. Ale vrchol v_1 musí mít zároveň vyšší číslo než v_n, což nelze splnit.

Cyklus je ovšem to jediné, co muže existenci topologického uspořádání zabránit. Libovolný acyklický graf lze uspořádat následujícím algoritmem:

1. Na začátku máme orientovaný graf G a proměnnou $p = 1$.
2. Najdeme takový vrchol v, ze kterého nevede žádná hrana (budeme mu říkat *stok*). Pokud v grafu žádný stok není, výpočet končí, protože jsme našli cyklus.
3. Odebereme z grafu vrchol v a všechny hrany, které do něj vedou.
4. Přiřadíme vrcholu v číslo p.
5. Proměnnou p zvýšíme o 1.
6. Opakujeme kroky 2 až 5, dokud graf obsahuje alespoň jeden vrchol.

Proč tento algoritmus funguje? Pokud v grafu nalezneme stok, můžeme mu určitě přiřadit číslo menší než všem ostatním vrcholům, protože překážet by nám v tom mohly pouze hrany vedoucí ze stoku ven a ty neexistují. Jakmile stok očíslujeme, můžeme jej z grafu odstranit a pokračovat číslováním ostatních vrcholů. Tento postup musí někdy skončit, jelikož v grafu je pouze konečně mnoho vrcholů.

Zbývá si uvědomit, že v neprázdném grafu, který neobsahuje cyklus, vždy existuje alespoň jeden stok: Vezměme libovolný vrchol v_1. Pokud z něj vede nějaká hrana, pokračujme po ní do nějakého vrcholu v_2, z něj do v_3 atd. Co se při tom může stát?

- Dostaneme se do vrcholu v_i, ze kterého nevede žádná hrana. Vyhráli jsme, máme stok.
- Narazíme na v_i, ve kterém jsme už jednou byli. To by ale znamenalo, že graf obsahuje cyklus, což, jak víme, není pravda.
- Budeme objevovat stále a nové a nové vrcholy. V konečném grafu nemožno.

Algoritmus můžeme navíc snadno upravit tak, aby netratil příliš času hledáním vrcholů, z nichž nic nevede – stačí si takové vrcholy pamatovat ve frontě a kdykoliv nějaký takový vrchol odstraňujeme, zkontrolovat si, zda jsme nějakému jinému vrcholu nezrušili poslední hranu, která z něj vedla, a pokud ano, přidat takový vrchol na konec fronty. Celé topologické třídění pak zvládneme v čase $O(N + M)$.

Jiná možnost je prohledat graf do hloubky a všimnout si, že pořadí, ve kterém jsme se z vrcholů vraceli, je právě topologické pořadí. Pokud zrovna opouštíme nějaký vrchol a číslujeme ho dalším číslem v pořadí, rozmysleme si, jaké druhy hran z něj mohou vést: stromová nebo dopředná hrana vede do vrcholu, kterému jsme již přiřadili nižší číslo, zpětná existovat nemůže (v grafu by byl cyklus) a příčné hrany vedou pouze zprava doleva, takže také do již očíslovaných vrcholů. Časová složitost je opět $O(N + M)$.

```
var Ocislovani: array[1..MaxN] of Integer;
    Posledni: Integer;
    I: Integer;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Ocislovani[V] := 0; { zatím V jen označíme }
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if Ocislovani[Sousedi[I]] = -1 then
      Projdi(Sousedi[I]);
  Inc(Posledni);
  Ocislovani[V] := Posledni;
end;

begin
  ...
  for I := 1 to N do Ocislovani[I] := -1;
  Posledni := 0;
  for I := 1 to N do if Ocislovani[I] = -1 then Projdi(I);
  ...
end.
```

Hranová a vrcholová 2-souvislost

Nyní se podíváme na trochu komplikovanější formu souvislosti. Říkáme, že neorientovaný graf je *hranově 2-souvislý*, když platí, že:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolné hrany.

Hranu, jejíž odebrání by způsobilo zvýšení počtu komponent souvislosti grafu, nazýváme *most.*

Na hledání mostů nám poslouží opět upravené prohledávání do hloubky a DFS strom. Všimněme si, že mostem může být jedině stromová hrana – každá

jiná hrana totiž leží na nějaké kružnici. Odebráním mostu se graf rozpadne na část obsahující kořen DFS stromu a podstrom „visící“ pod touto hranou. Jediné, co tomu může zabránit, je existence nějaké další hrany mezi podstromem a hlavní částí, což musí být zpětná hrana, navíc taková, která není jenom stromovou hranou viděnou z druhé strany. Takovým hranám budeme říkat *ryzí zpětné hrany.*

Proto si pro každý vrchol spočítáme hladinu, ve které se nachází (kořen je na hladině 0, jeho synové na hladině 1, jejich synové 2, ...). Dále si pro každý vrchol v spočítáme, do jaké nejvyšší hladiny (s nejmenším číslem) vedou ryzí zpětné hrany z podstromu s kořenem v. To můžeme udělat přímo při procházení do hloubky, protože než se vrátíme z v, projdeme celý podstrom pod v. Pokud všechny zpětné hrany vedou do hladiny stejné nebo větší než té, na které je v, pak odebráním hrany vedoucí do v z jeho otce vzniknou dvě komponenty souvislosti, čili tato hrana je mostem. V opačném případě jsme nalezli kružnici, na níž tato hrana leží, takže to most být nemůže. Výjimku tvoří kořen, který žádného otce nemá a nemusíme se o něj proto starat.

Algoritmus je tedy pouhou modifikací procházení do hloubky a má i stejnou časovou a paměťovou složitost $O(N + M)$. Zde jsou důležité části programu:

```
var Hladina, Spojeno: array[1..MaxN] of Integer;
    DvojSouvisle: Boolean;
    I: Integer;

procedure Projdi(V, NovaHladina: Integer);
var I, W: Integer;
begin
  Hladina[V] := NovaHladina;
  Spojeno[V] := Hladina[V];

  for I := Zacatky[V] to Zacatky[V+1]-1 do begin
    W := Sousedi[I];
    if Hladina[W] = -1 then begin                    { stromová hrana }
      Projdi(W, NovaHladina + 1);
      if Spojeno[W] < Spojeno[V] then Spojeno[V] := Spojeno[W];
      if Spojeno[W] > Hladina[V] then DvojSouvisle := False; { máme most }
    end else                                  { zpětná nebo dopředná hrana }
    if (Hladina[W] < NovaHladina-1) and (Hladina[W] < Spojeno[V]) then
      Spojeno[V] := Hladina[W];
  end;
end;

begin
  ...
  for I := 1 to N do
    Hladina[I] := -1;
  DvojSouvisle := True;
  Projdi(1, 0);
  ...
end.
```

Další formou souvislosti je *vrcholová souvislost*. Řekneme, že graf je *vrcholově 2-souvislý*, právě když:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolného vrcholu.

Artikulace je takový vrchol, který když odebereme, zvýší se počet komponent souvislosti grafu.

Algoritmus pro zjištění vrcholové 2-souvislosti grafu je velmi podobný algoritmu na zjišťování hranové 2-souvislosti. Jen si musíme uvědomit, že odebíráme celý vrchol. Ze stromu procházení do hloubky může odebráním vrcholu vzniknout až několik podstromů, které všechny musí být spojeny zpětnou hranou s hlavním stromem. Proto musí zpětné hrany z podstromu určeného vrcholem v vést až *nad* vrchol v. Speciálně pro kořen nám vychází, že může mít pouze jednoho syna, jinak bychom ho mohli odebrat a vytvořit tak dvě nebo více komponent souvislosti. Algoritmus se od hledání hranové 2-souvislosti liší jedinou změnou ostré nerovnosti na neostrou, sami zkuste najít, které nerovnosti.

16-3-K Kuchařka třetí série – halda, Dijkstrův algoritmus

Dnešní povídání o algoritmech a datových strukturách se bude zabývat jedním z nejznámějších algoritmů: Dijkstrovým algoritmem pro hledání nejkratších cest v grafech. K tomu (a nejenom k tomu) se nám bude hodit šikovná datová strukturka zvaná halda, tak si předvedeme nejdříve ji.

Halda

Halda je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných prvků, na kterých máme definováno uspořádání, tj. umíme pro každou dvojici prvků říci, který z nich je menší). Tato datová struktura obvykle podporuje následující operace: Přidání nového prvku, nalezení nejmenšího prvku a odebrání nejmenšího prvku. My si ukážeme jednoduchou implementaci haldy, která bude při uložení N prvků potřebovat čas $O(\log N)$ na přidání či odebrání jednoho prvku a $O(1)$ (tj. konstantní) na zjištění hodnoty nejmenšího prvku.

Naše implementace bude vypadat následovně: Pokud halda obsahuje N prvků, uložíme její prvky do pole na pozice 1 až N. Prvek na pozici k bude mít dva *následníky,* a to prvky na pozicích $2k$ a $2k + 1$; samozřejmě, pokud je k velké, a tedy např. $2k + 1 > N$, má takový prvek jen jednoho či dokonce žádného následníka. Naopak prvek na pozici $\lfloor k/2 \rfloor$ nazveme *předchůdcem* prvku na pozici k. Ti z vás, kteří znají binární stromy, v tomto jistě rozpoznali způsob, jak v poli uchovávat úplné binární stromy (následníci jsou synové a předchůdci otcové v obvyklé stromové terminologii, prvek č. 1 je kořen stromu).

Prvky haldy však v poli neuchováváme v úplně libovolném pořadí. Chceme, aby platilo, že každý prvek je menší nebo roven všem svým následníkům. Naše halda tedy může vypadat např. takto:

1	2	3	4	5	6	7	8	9
5	6	20	25	7	21	22	26	27

Z toho, co jsme si právě popsali, je jasné, že nejmenší prvek je uložen na pozici s indexem 1, a tedy můžeme snadno v konstantním čase zjistit jeho hodnotu. Ještě prozradíme, jak lze prvky do haldy rychle přidávat a odebírat:

Jestliže halda obsahuje N prvků, pak nový prvek, říkejme mu třeba x, přidáme na konec pole, tj. na pozici s indexem N. Nyní x porovnáme s jeho předchůdcem. Pokud je jeho předchůdce menší, je vše v pořádku a jsme hotovi. V opačném případě x s jeho předchůdcem prohodíme. Tím jsme problém napravili, ale nyní může být x menší než jeho nový předchůdce. To lze napravit dalším prohozením a tak budeme pokračovat dále, než se buďto dostaneme do situace, kdy už je x větší nebo rovno svému předchůdci, nebo „vybublá“ až do kořene haldy, kde už žádného předchůdce nemá. Protože se v každém kroku pozice, na níž se prvek x právě nachází, zmenší alespoň na polovinu, provedeme dohromady nejvýše $O(\log N)$ výměn, a tedy spotřebujeme čas $O(\log N)$.

Odebírání nejmenšího prvku probíhá podobně: Prvek z poslední pozice (tj. z pozice N) přesuneme na pozici 1, tedy místo minima. Místo s předchůdci jej však porovnáme s jeho následníky a v případě, že je větší než některý z jeho následníků, opět je prohodíme (pokud je větší než oba následníci, prohodíme ho s menším z nich). A protože se nám v každém kroku index „bublajícího“ prvku v poli alespoň zdvojnásobí, opět spotřebujeme čas $O(\log N)$.

Jako cvičení si rozmyslete, že v čase $O(\log N)$ lze z haldy smazat dokonce libovolný prvek, pokud si ovšem pamatujeme, kde se v haldě nachází. Také můžeme prvek ponechat a jen změnit jeho hodnotu.

Ještě si předvedeme program:

```
var halda: array[1..MAX] of integer;
    N: integer;            { počet prvků v haldě }

function nejmensi: integer;
begin
  nejmensi:=halda[1]
end;

procedure vloz(prvek: integer);
var i, x: integer;
begin
  i:=N; N:=N+1;
  halda[i]:=prvek;
  while (i>1) and (halda[i div 2]>halda[i]) do begin
    x:=halda[i div 2];
    halda[i div 2]:=halda[i];
```

```
    halda[i]:=x;
    i:=i div 2
  end
end;

procedure smaz_nejmensi;
var i, j, x: integer;
begin
  halda[1]:=halda[N];
  N:=N-1; i:=1;
  while 2*i<=N do begin
    j:=i;
    if halda[j]>halda[2*i] then j:=2*i;
    if (2*i+1<=N) and (halda[j]>halda[2*i+1]) then j:=2*i+1;
    if i=j then break;
    x:=halda[i]; halda[i]:=halda[j]; halda[j]:=x;
    i:=j
  end
end;
```

HeapSort

Když už máme k dispozici haldu, můžeme pomocí ní například snadno třídit čísla. Máme-li N čísel, která chceme setřídit, vytvoříme si z nich nejprve haldu o N prvcích (například postupným vkládáním do prázdné haldy), načež z ní budeme postupně N-krát odebírat nejmenší prvek. Tím získáme prvky původního pole v rostoucím pořadí. Celkově provedeme N vložení, N nalezení minima a N smazání. To vše dohromady stihneme v čase $O(N \log N)$.

Než si ukážeme program, přidáme ještě dva triky, které nám implementaci značně usnadní. Předně si vše uložíme do jednoho pole – to bude při plnění haldy obsahovat na svém začátku haldu a na konci zbytek vstupního pole, přitom zbytek pole se bude postupně zmenšovat a uvolňovat tak místo haldě; naopak v druhé polovině algoritmu budeme zmenšovat haldu a do volného prostoru ukládat setříděné prvky. K tomu se nám bude hodit získávat prvky v opačném pořadí, proto si upravíme haldu tak, aby udržovala nikoliv minimum, nýbrž maximum.

Druhý trik spočívá v tom, že nebudeme haldu vytvářet postupným vkládáním, nýbrž naopak zabubláváním prvků (podobným, jako děláme při mazání minima) od konce. Všimněte si, že takto také získáme správné nerovnosti mezi prvky a jejich následníky, a dokonce tak zvládneme celou haldu vytvořit v lineárním čase [proč to tak je, si zkuste dokázat sami, stačí si uvědomit, kolikrát zabubláváme které prvky]. Zbytek třídění bohužel nadále zůstává $O(N \log N)$.

Tomuto algoritmu se říká *HeapSort* (třídění haldou) a je jedním z mála známých rychlých třídících algoritmů, které nepotřebují pomocnou paměť.

```
type Pole = array[1..MAXN] of Integer;

procedure HeapSort(var A: Pole);
var i, x: integer;
```

```
   procedure bublej(m, i: integer); { "zabublání" prvku }
   { m je velikost haldy, i je index zabublávaného prvku }
   var j, x: integer;
   begin
      while 2*i<=m do begin
         j:=2*i;
         if (j<m) and (A[j+1]>A[j]) then j:=j+1;
         if A[i]>=A[j] then break;
         x:=A[i]; A[i]:=A[j]; A[j]:=x;
         i:=j;
      end;
   end;
begin
   for i:=N div 2 downto 1 do bublej(N,i); { bublej }
   for i:=N downto 2 do begin { vybírej maximum }
      x:=A[1]; A[1]:=A[i]; A[i]:=x;
      bublej(i-1, 1);
   end;
end;
```

Dijkstrův algoritmus

Nyní již konečně k slíbenému *Dijkstrovu algoritmu.* Tento algoritmus dostane orientovaný graf s hranami ohodnocenými nezápornými čísly (viz kuchařka v minulé sérii) a nalezne v něm nejkratší cestu mezi dvěma zadanými vrcholy. Ve skutečnosti tento algoritmus dělá o malinko více: Najde totiž nejkratší cesty z jednoho zadaného vrcholu do všech ostatních.

Nechť v_0 je vrchol grafu, ze kterého chceme určit délky nejkratších cest. Budeme si udržovat pole délek zatím nalezených cest z vrcholu v_0 do všech ostatních vrcholů grafu. Navíc u některých vrcholů budeme mít poznamenáno, že cesta nalezená do nich je už ta nejkratší možná. Takovým vrcholům budeme říkat *definitivní.* Na začátku inicializujeme v poli všechny hodnoty na ∞ kromě hodnoty odpovídající vrcholu v_0, kterou inicializujeme na 0 (délka nejkratší cesty z v_0 do v_0 je 0). V každém *kroku* algoritmu pak provedeme následující: Vybereme vrchol w, který ještě není definitivní, a mezi všemi takovými vrcholy je délka zatím nalezené cesty do něj nejkratší možná. Vrchol w prohlásíme za definitivní. Dále otestujeme, zda pro nějaký vrchol v cesta z vrcholu v_0 do w a pak po hraně z w do v není kratší, než zatím nalezená cesta z v_0 do v, a je-li tomu tak, upravíme délku zatím nalezené cesty do v. Toto provedeme pro všechny takové vrcholy v. Celý algoritmus skončí, pokud jsou už všechny vrcholy definitivní nebo všechny vrcholy, co nejsou definitivní, mají délku cesty rovnou ∞ (v takovém případě se graf skládá z více nesouvislých částí).

Předtím než dokážeme, že právě představený algoritmus opravdu nalezne délky nejkratších cest z vrcholu v_0, se zamysleme nad jeho časovou složitostí.

Pro každý z N vrcholů si délku dosud nalezené cesty uchováme v poli. Celý algoritmus má nejvýše N kroků, protože v každém kroku nám přibude jeden

definitivní vrchol. Ten vybíráme z jako minimum z délky aktuální cesty přes všechny dosud nedefinitivní vrcholy, kterých je $O(N)$. V každému kroku musíme zkontrolovat tolik vrcholů v, kolik hran vede z vrcholu w. Počet takových změn pro všechny kroky dohromady je pak nejvýše $O(M)$, kde M je počet hran vstupního grafu. Z toho vyjde časová složitost $O(N^2 + M)$, čili $O(N^2)$, jelikož M je nejvýše N^2. Tuto implementaci Dijkstrova algoritmu najdete na konci naší kuchařky.

K uchovávání délek dosud nalezených nejkratších cest můžeme ovšem použít haldu. Ta bude na začátku obsahovat N prvků a v každém kroku se počet jejích prvků sníží o jeden: Nalezneme a smažeme nejmenší prvek, to zvládneme v čase $O(\log N)$, a případně upravíme délky nejkratších cest do sousedů právě zpracovávaného vrcholu. To pro každou hranu trvá rovněž $O(\log N)$, celkově za všechny hrany tedy $O(M \log N)$. Z toho vyjde celková časová složitost algoritmu $O((N + M) \log N)$, a to je pro „řídké“ grafy (tedy grafy s $M \ll N^2$) výrazně lepší.

Vraťme se nyní k důkazu správnosti Dijkstrova algoritmu. Ukážeme, že po každém kroku algoritmu platí následující tvrzení: Nechť A je množina definitivních vrcholů. Pak délka dosud nalezené cesty z v_0 do v (v je libovolný vrchol grafu) je délka nejkratší cesty $v_0 v_1 \ldots v_k v$ takové, že všechny vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Tvrzení dokážeme indukcí dle počtu kroků algoritmu, které již proběhly. Tvrzení zřejmě platí před a po prvním kroku algoritmu. Nechť w je vrchol, který byl v předchozím kroku prohlášen za definitivní. Uvažme nejprve nějaký vrchol v, který je definitivní. Pokud $v = w$, tvrzení je triviální. V opačném případě ukážeme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w. Označme D délku cesty z v_0 do v přes vrcholy A bez vrcholu w. Protože v každém kroku vybíráme vrchol s nejmenším ohodnocením a ohodnocení vybraných vrcholů v jednotlivých krocích tvoří neklesající posloupnost (váhy hran jsou nezáporné!), tak délka cesty z v_0 do w přes vrcholy z A je alespoň D. Ale potom délka libovolné cesty z v_0 do v přes w používající vrcholy z A je alespoň D. Z volby D pak víme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w.

Nyní uvažme takový vrchol v, který není definitivní. Nechť $v_0 v_1 \ldots v_k v$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Pokud $v_k = w$, pak jsme ohodnocení v změnili na délku této cesty v právě proběhlém kroku. Pokud $v_k \neq w$, pak $v_0 v_1, \ldots, v_k$ je nejkratší cesta z v_0 do v_k přes vrcholy z množiny A a tedy můžeme předpokládat, že žádný z vrcholů $v_1, \ldots, v_k$ není w (podle toho, co jsme si rozmysleli na konci minulého odstavce). Potom se ale délka cesty do v rovnala správné hodnotě už před právě proběhlým krokem.

Vzhledem k tomu, že po posledním kroku množina A obsahuje právě ty vrcholy, do kterých existuje cesta z vrcholu v_0, dokázali jsme, že náš algoritmus

funguje správně.

Na závěr ještě poznamenejme, že Dijkstrův algoritmus funguje je možné snadno upravit tak, aby nám kromě délky nejkratší cesty i takovou cestu našel: U každého vrcholu si v okamžiku, kdy mu měníme ohodnocení, poznamenáme, ze kterého vrcholu do něj přicházíme. Nejkratší cestu do nějakého vrcholu pak zrekonstruujeme tak, že u posledního vrcholu této cesty zjistíme, který vrchol je předposlední, u předposledního, který je předpředposlední, atd.

Poznámka pro zvídavé: existují i jiné druhy hald, například k-regulární haldy, v nichž má každý prvek k následníků (rozmyslete si, jaká je v takové haldě časová složitost operací a jak nastavit k v závislosti na M a N, aby byl Dijkstrův algoritmus co nejrychlejší), nebo tzv. Fibonacciho halda, která dokáže upravit hodnotu prvku v konstantním čase. S tou pak umíme hledat nejkratší cesty v čase $O(M + N \log N)$.

Implementace Dijkstrova algoritmu

```
var N: word;                                  { počet vrcholů }
    vahy: array[1..MAX, 1..MAX] of integer; { váhy hran, -1 = hrana neex. }
    delky: array[1..MAX] of integer;          { délky zatím nalezených cest, }
                                              {   -1 = nekonečno }
    def: array[1..MAX] of boolean;            { definitivní? }

procedure Dijkstra(odkud: word);
var i, w, v: word;
begin
  for i:=1 to N do begin
    def[i]:=false;  delky[i]:=-1;
  end;
  def[odkud]:=true;
  delky[odkud]:=0;
  repeat
    w:=0;
    for i:=1 to N do
      if not def[i] and ((w=0) or (delky[i]<delky[w])) then w:=i;
    if w<>0 then begin
      def[w]:=true;
      for i:=1 to N do
        if (vahy[w][i]<>-1) and (delky[w]+vahy[w][i]<delky[i]) then
          delky[i]:=delky[w]+vahy[w][i]
    end
  until w=0;
end;
```

16-4-K Kuchařka čtvrté série – vyhledávací stromy

V nedávném vydání programátorské kuchařky jsme se zabývali tříděním dat, tentokráte si povíme, jak v uspořádaných datech něco efektivně najít a jak si data udržovat stále uspořádaná. K tomu se nám bude hodit zejména binární vyhledávání a různé druhy vyhledávacích stromů.

Binární vyhledávání. Představte si, že jste k narozeninám dostali obrovské pole setříděných záznamů (to je, pravda, trochu netradiční dárek, ale proč ne – může to být třeba telefonní seznam). Záznamy mohou vypadat libovolně a to, že jsou setříděné, znamená jen a pouze, že $x_1 < x_2 < \ldots < x_N$, kde $<$ je nějaká relace, která nám řekne, který ze dvou záznamů je menší (pro jednoduchost předpokládáme, že žádné dva záznamy nejsou stejné).

Co si ale s takovým polem počneme? Zkusíme si v něm najít nějaký konkrétní záznam z. To můžeme udělat třeba tak, že si nalistujeme prostřední záznam (označíme si ho x_m) a porovnáme s ním naše z. Pokud $z < x_m$, víme, že se z nemůže vyskytovat „napravo“ od x_m, protože tam jsou všechny záznamy větší než x_m a tím spíše než z. Analogicky pokud $z > x_m$, nemůže se z vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně zmenšovat interval, ve kterém se z může nacházet, až buďto z najdeme nebo vyloučíme všechny prvky, kde by mohlo být.

Tomuto principu se obvykle říká *binární vyhledávání* nebo také *hledání půlením intervalu* a snadno ho naprogramujeme buďto rekursivně nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r\rangle$, ve kterém se hledaný prvek může nacházet:

```
function BinSearch(z : integer):integer;
var l,r,m : integer;
begin
  l := 1;               { interval, ve kterém hledáme }
  r := N;
  while l <= r do begin { ještě není prázdný }
    m := (l+r) div 2;   { střed intervalu }
    if z < x[m] then
      r := m-1          { je vlevo }
    else if z > x[m] then
      l := m+1          { je vpravo }
    else begin          { Bingo! }
      hledej := m;
      exit;
    end;
  end;
  hledej := -1;         { nebyl nikde }
end;
```

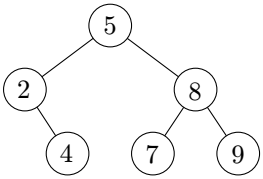
Všimněte si, že průchodů cyklem `while` může být nejvýše $\lceil \log_2 N \rceil$, protože interval $\langle l, r\rangle$ na počátku obsahuje N prvků a v každém průchodu jej zmenšíme

na polovinu (ve skutečnosti ještě o jedničku, ale tím lépe pro nás). Proto po k průchodech bude interval obsahovat nejvýše $N/2^k$ prvků a jelikož pro $N/2^k < 1$ se algoritmus zastaví, může být k nejvýše $\log_2 N$. Proto je časová složitost binárního vyhledávání $O(\log N)$. [Základ logaritmu nemusíme psát, protože $\log_a b = \log_c b / \log_c a$, čili logaritmy o různých základech se liší jen konstantou, která se „schová do O-čka.“]

Hledání půlením intervalu je tedy velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, potážeme se se zlou: buďto budeme mít záznamy uložené v poli, a pak nezbývá než při zatřiďování nového prvku ostatní „rozhrnout“, což může trvat až N kroků, a nebo si je budeme udržovat v nějakém seznamu, do kterého dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni najít tolik potřebnou polovinu.

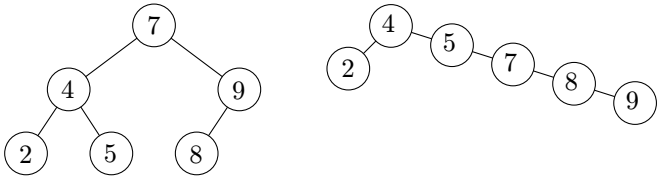
Zkusme ale provést jednoduchý myšlenkový pokus:

Vyhledávací stromy. Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáváme s prostředním prvkem a podle výsledku se vydáme jednou ze dvou možných cest (nebo rovnou zjistíme, že se jedná o hledaný prvek, ale to není moc zajímavý případ). Na každé cestě nás zase čeká porovnání se středem příslušného intervalu a to nás opět pošle jednou ze dvou dalších cest atd. To si můžeme přehledně popsat pomocí stromu:



Jeden vrchol stromu prohlásíme za kořen a ten bude odpovídat celému poli (a jeho prostřednímu prvku). K němu budou připojené vrcholy obou polovin pole (opět obsahující příslušné prostřední prvky) a tak dále. Ovšem jakmile známe tento strom, můžeme náš půlící algoritmus provádět přímo podle stromu (ani k tomu nepotřebujeme vidět původní pole a umět v něm hledat poloviny): začneme v kořeni, porovnáme a podle výsledku se buďto přesuneme do levého nebo pravého podstromu a tak dále. Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu.

Teď si ale všimněte, že aby hledání hodnoty podle stromu fungovalo, strom vůbec nemusel vzniknout půlením intervalu – stačilo, aby v každém vrcholu platilo, že všechny hodnoty v levém podstromu jsou menší než tento vrchol a naopak hodnoty v pravém podstromu větší. Hledání v témže poli by také popisovaly následující stromy (např.):



Hledací algoritmus podle jiných stromů samozřejmě už nemusí mít pěknou logaritmickou složitost (když bychom hledali podle „degenerovaného“ stromu z pravého obrázku, trvalo by to dokonce lineárně). Důležité ale je, že takovéto stromy se dají poměrně snadno modifikovat a že je při troše šikovnosti můžeme udržet dostatečně podobné ideálnímu půlení intervalu. Pak bude hloubka stromu stále $O(\log N)$, tím pádem i časová složitost hledání a, jak za chvilku uvidíme, mnohých dalších operací.

Definice. Zkusme si tedy pořádně nadefinovat to, co jsme právě vymysleli:

Binární vyhledávací strom (po domácku BVS) je buďto prázdná množina nebo *kořen* obsahující jednu hodnotu a mající dva *podstromy* (levý a pravý), což jsou opět BVS, ovšem takové, že všechny hodnoty uložené v levém podstromu jsou menší než hodnota v kořeni, a ta je naopak menší než všechny hodnoty uložené v pravém podstromu.

Úmluva: Pokud x je kořen a L_x a R_x jeho levý a pravý podstrom, pak kořenům těchto podstromů (pokud nejsou prázdné) budeme říkat *levý a pravý syn* vrcholu x a naopak vrcholu x budeme říkat *otec* těchto synů. Pokud je některý z podstromů prázdný, pak vrchol x příslušného syna nemá. Vrcholu, který nemá žádné syny, budeme říkat *list* vyhledávacího stromu. Všimněte si, že pokud x má jen jediného syna, musíme stále rozlišovat, je-li to syn levý nebo pravý, protože potřebujeme udržet správné uspořádání hodnot. Také si všimněte, že pokud známe syny každého vrcholu, můžeme rekonstruovat všechny podstromy.

Každý BVS také můžeme popsat velmi jednoduchou strukturou v paměti:

```
type pvrchol = ^vrchol;
     vrchol = record
       l, r : pvrchol;  { levý a pravý syn }
       x    : integer;  { hodnota }
     end;
```

Pokud některý syn neexistuje, zapíšeme do příslušné položky hodnotu *nil.*

Find. V řeči BVS můžeme přeformulovat náš vyhledávací algoritmus takto:

```
function TreeFind(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu. Vrátí vrchol s danou hodnotou, příp. nil.}
begin while (v<>nil) and (v^.x<>x) do begin
        if x<v^.x then v := v^.l
        else           v := v^.r
      end;
      TreeFind := v;
end;
```

Funkce `TreeFind` bude pracovat v čase $O(h)$, kde h je hloubka stromu, protože začíná v kořeni a v každém průchodu cyklem postoupí o jednu hladinu níže.

Insert. Co kdybychom chtěli do stromu vložit novou hodnotu (aniž bychom se teď starali o to, zda tím strom nemůže degenerovat)? Stačí zkusit hodnotu najít a pokud tam ještě nebyla, určitě při hledání narazíme na odbočku, která je *nil*. A přesně na toto místo připojíme nově vytvořený vrchol, aby byl správně uspořádán vzhledem k ostatním vrcholům (že tomu tak je, plyne z toho, že při hledání jsme postupně vyloučili všechna ostatní místa, kde nová hodnota být nemohla). Naprogramujeme opět snadno, tentokráte si ukážeme rekurzivní zacházení se stromy:

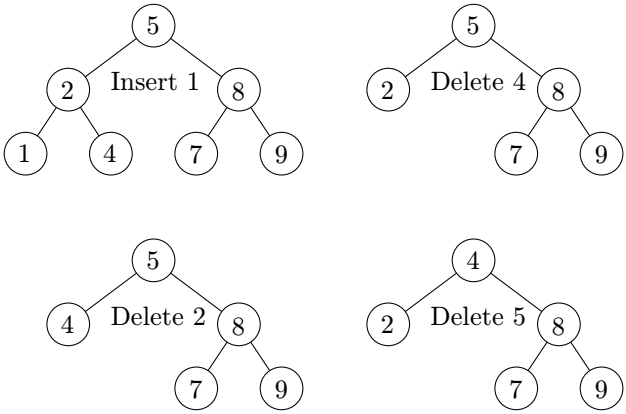
```
function TreeIns(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu ke vložení, vrátí nový kořen. }
begin
  if v=nil then begin          { prázdný strom => založíme nový kořen }
    new(v);
    v^.l := nil; v^.r := nil; v^.x := x;
  end else if x<v^.x then v^.l := TreeIns(v^.l, x)   { vkládáme vlevo }
  else if x>v^.x then     v^.r := TreeIns(v^.r, x); { vkládáme vpravo }
  TreeIns := v;
end;
```

Delete. Mazání bude o kousíček pracnější, musíme totiž rozlišit tři případy: Pokud je mazaný vrchol list, stačí ho vyměnit za *nil*. Pokud má právě jednoho syna, stačí náš vrchol v ze stromu odstranit a syna přepojit k otci v. A pokud má syny dva, najdeme největší hodnotu v levém podstromu (tu najdeme tak, že půjdeme jednou doleva a pak pořád doprava), umístíme ji do stromu namísto mazaného vrcholu a v levém podstromu ji pak smažeme (což už umíme, protože má 1 nebo 0 synů). Program následuje:

```
function TreeDel(v:pvrchol; x:integer):pvrchol;
{ Parametry stejně jako TreeIns }
var w:pvrchol;
begin
  TreeDel := v;
  if v=nil then exit                              { prázdný strom }
  else if x<v^.x then v^.l := TreeDel(v^.l, x) { ještě hledáme x }
  else if x>v^.x then v^.r := TreeDel(v^.r, x)
  else begin                                      { našli jsme }
    if (v^.l=nil) and (v^.r=nil) then begin       { mažeme list }
      TreeDel := nil; dispose(v);
    end else if v^.l=nil then begin               { jen pravý syn }
      TreeDel := v^.r; dispose(v);
    end else if v^.r=nil then begin               { jen levý }
      TreeDel := v^.l; dispose(v);
    end else begin                                { má oba syny }
      w := v^.l;                                  { hledáme max(L) }
      while w^.r<>nil do w := w^.r;
```

```
      v^.x := w^.x;                    { prohazujeme a mažeme původní max(L)}
      v^.l := TreeDel(v^.l, w^.x);
    end;
  end;
end;
```

Když do stromu z našeho prvního obrázku zkusíme přidávat nebo z něj odebírat prvky, dopadne to takto:



Jak vkládání, tak mazání opět budou trvat $O(h)$. Ale pozor, jejich používáním může h nekontrolovatelně růst – sami zkuste najít nějaký příklad, kdy h dosáhne až N.

Procházení stromu. Pokud bychom chtěli všechny hodnoty ve stromu vypsat, stačí strom rekursivně projít a sama definice uspořádání hodnot ve stromu nám zajistí, že hodnoty vypíšeme ve vzestupném pořadí: nejdříve levý podstrom, pak kořen a pak podstrom pravý. Časová složitost je, jak se snadno nahlédne, lineární, protože strávíme konstantní čas vypisováním každého prvku a prvků je právě N. Program bude opět přímočarý:

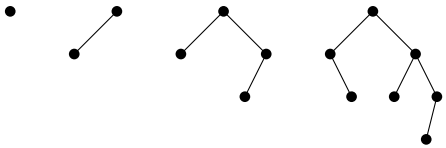
```
procedure TreeShow(v:pvrchol);
begin
  if v=nil then exit;                    { není co dělat }
  TreeShow(v^.l);
  writeln(v^.x);
  TreeShow(v^.r);
end;
```

Vyvážené stromy. S binárními stromy lze dělat všelijaká kouzla a prakticky všechny stromové algoritmy mají společné to, že jejich časová složitost je lineární v hloubce stromu. (Pravda, právě ten poslední byl výjimka, leč všechny prvky rychleji než lineárně s N opravdu nevypíšeme.) Jenže jak jsme viděli, neopatrným insertováním a deletováním prvků mohou snadno vznikat všelijaké degenerované stromy, které mají lineární hloubku. Abychom tomu zabránili,

musíme stromy *vyvažovat.* To znamená definovat si nějaké šikovné omezení na tvar stromu, aby hloubka byla vždy $O(\log N)$. Možností je mnoho, my uvedeme jen ty nejdůležitější:

Dokonale vyvážený budeme říkat takovému stromu, ve kterém pro každý vrchol platí, že počet vrcholů v jeho levém a pravém podstromu se liší nejvýše o jedničku. Takové stromy kopírují dělení na poloviny při binárním vyhledávání, a proto (jak jsme již dokázali) mají vždy logaritmickou hloubku. Jediné, čím se liší, je, že mohou zaokrouhlovat na obě strany, zatímco náš půlící algoritmus zaokrouhloval polovinu vždy dolů, takže levý podstrom nemohl být nikdy větší než pravý. Z toho také plyne, že se snadnou modifikací půlícího algoritmu dá dokonale vyvážený BVS v lineárním čase vytvořit ze setříděného pole. Bohužel se ale při Insertu a Deletu nedá v logaritmickém čase strom znovu vyvážit.

AVL stromy. Zkusíme tedy vyvažovací podmínku trochu uvolnit a vyžadovat, aby se u každého vrcholu lišily o jedničku nikoliv velikosti podstromů, nýbrž pouze jejich hloubky. Takovým stromům se říká *AVL stromy* a mohou vypadat třeba takto:



Každý dokonale vyvážený strom je také AVL stromem, ale jak je vidět na předchozím obrázku, opačně to platit nemusí. To, že hloubka AVL stromů je také logaritmická, proto není úplně zřejmé a zaslouží si to trochu dokazování:

Věta: AVL strom o N vrcholech má hloubku $O(\log N)$.

Důkaz: Označme A_d nejmenší možný počet vrcholů, jaký může být v AVL stromu hloubky d. Snadno vyzkoušíme, že $A_1 = 1$, $A_2 = 2$, $A_3 = 4$ a $A_4 = 7$ (příslušné minimální stromy najdete na předchozím obrázku). Navíc platí, že $A_d = 1 + A_{d-1} + A_{d-2}$, protože každý minimální strom hloubky d musí mít kořen a 2 podstromy, které budou opět minimální, protože jinak bychom je mohli vyměnit za minimální a tím snížit počet vrcholů celého stromu. Navíc alespoň jeden z podstromů musí mít hloubku $d-1$ (protože jinak by hloubka celého stromu nebyla d) a druhý hloubku $d-2$ (podle definice AVL stromu může mít $d-1$ nebo $d-2$, ale s menší hloubkou bude mít evidentně méně vrcholů).

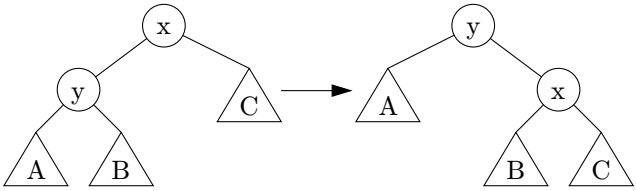
Spočítat, kolik přesně je A_d, není úplně snadné. Nám však postačí dokázat, že $A_d \geq 2^{d/2}$. To provedeme indukcí: Pro $d < 4$ to plyne z ručně spočítaných hodnot. Pro $d \geq 4$ je $A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2} \cdot (2^{-1/2} + 2^{-1}) > 2^{d/2}$ (součet čísel v závorce je ≈ 1.207).

Jakmile už víme, že A_d roste s d alespoň exponenciálně, tedy že $\exists c : A_d \geq c^d$, důkaz je u konce: Máme-li AVL strom T na N vrcholech, najdeme si nejmen-

ší d takové, že $A_d \leq N$. Hloubka stromu T může být maximálně d, protože jinak by T musel mít alespoň A_{d+1} vrcholů, ale to je více než N. A jelikož A_d rostou exponenciálně, je $d \leq \log_c N$, čili $d = O(\log N)$. *Q.E.D.*

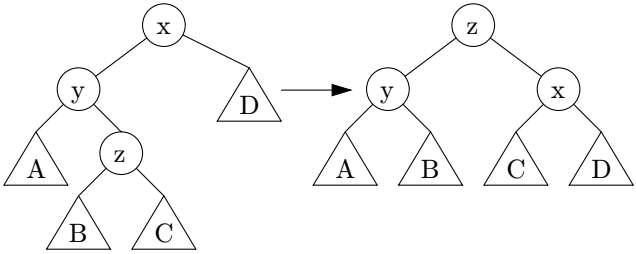
AVL stromy tedy vypadají nadějně, jen stále nevíme, jak provádět Insert a Delete tak, strom zůstal vyvážený. Nemůžeme si totiž dovolit strukturu stromu měnit libovolně – stále musíme dodržovat správné uspořádání hodnot. K tomu se nám bude hodit zavést si nějakou množinu operací, o kterých dokážeme, že jsou korektní, a pak strukturu stromu měnit vždy jen pomocí těchto operací. Budou to:

Rotace. Rotací binárního stromu (respektive nějakého podstromu) nazveme jeho „překořenění“ za některého ze synů kořene. Místo formální definice ukažme raději obrázek:



Strom jsme překořenili za vrchol y a přepojili jednotlivé podstromy tak, aby byly vzhledem k x a y opět uspořádané správně (všimněte si, že je jen jediný způsob, jak to udělat). Jelikož se tím okolí vrcholu y „otočilo“ po směru hodinových ručiček, říká se takové operaci *rotace doprava.* Inverzní operaci (tj. překořenění za pravého syna kořene) se říká *rotace doleva* a na našem obrázku odpovídá přechodu zprava doleva.

Dvojrotace. Také si nakreslíme, jak to dopadne, když provedeme dvě rotace nad sebou lišící se směrem (tj. jednu levou a jednu pravou nebo opačně). Tomu se říká *dvojrotace* a jejím výsledkem je překořenění podstromu za vnuka kořene připojeného „cikcak“. Raději opět předvedeme na obrázku:



Znaménka. Při vyvažování se nám bude hodit pamatovat si u každého vrcholu, v jakém vztahu jsou hloubky jeho podstromů. Tomu budeme říkat *znaménko* vrcholu a bude buďto 0, jsou-li oba stejně hluboké, − pro levý podstrom

hlubší a + pro pravý hlubší. V textu budeme znaménka, respektive vrcholy se znaménky značit $\oplus$, $\ominus$ a $\odot$.

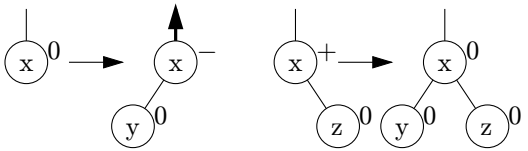
Pokud celý strom zrcadlově obrátíme (prohodíme levou a pravou stranu), znaménka se změní na opačná ($\oplus$ a $\ominus$ se prohodí, $\odot$ zůstane). Toho budeme často využívat a ze dvou zrcadlově symetrických situací popíšeme jenom jednu s tím, že druhá se v algoritmu zpracuje symetricky.

Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat, a nebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, a celou cestu z kořene si zapamatujeme v nějakém zásobníku a postupně se budeme vracet.

Tím jsme si připravili všechny potřebné ingredience, tož s chutí do toho:

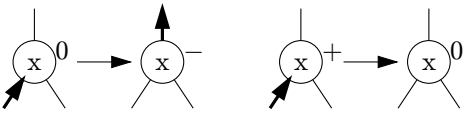
Vyvažování po Insertu. Když provedeme Insert tak, jak jsme ho popisovali u obecných vyhledávacích stromů, přibude nám ve stromu list. Pokud se tím AVL vyváženost neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže porušila, musíme s tím něco provést, konkrétně ji šikovně zvolenými rotacemi opravit. Popíšeme algoritmus, který bude postupovat od listu ke kořeni a vše potřebné zařídí:

Nejprve přidání listu samotné:



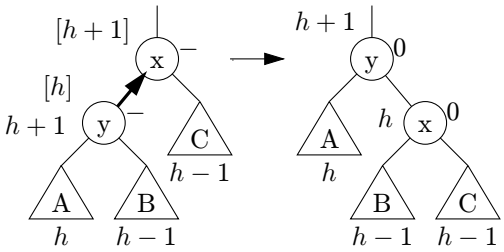
Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem $\odot$, změníme znaménko na $\ominus$ a pošleme o patro výš informaci o tom, že hloubka podstromu se zvýšila (to budeme značit šipkou). Přidali-li jsme list k $\oplus$, změní se na $\odot$ a hloubka podstromu se nemění, takže můžeme skončit.

Nyní rozebereme případy, které mohou nastat na vyšších hladinách, když nám z nějakého podstromu přijde šipka. Opět budeme předpokládat, že přišla zleva; pokud zprava, vyřešíme zrcadlově. Pokud přišla do $\oplus$ nebo $\odot$, ošetříme to stejně jako při přidání listu:



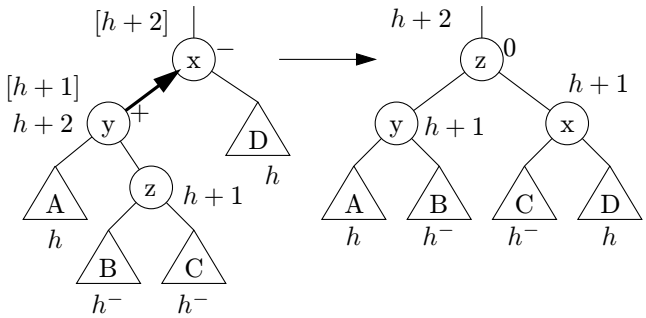
Pokud ale vrchol x má znaménko $\ominus$, nastanou potíže: levý podstrom má teď hloubku o 2 vyšší než pravý, takže musíme rotovat. Proto se podíváme

o patro níž, jaké je znaménko vrcholu y pod šipkou, abychom věděli, jakou rotaci provést. Jedna možnost je tato (y je $\ominus$):



Tehdy provedeme jednoduchou rotaci vpravo. Jak to dopadne s hloubkami jsme přikreslili do obrázku – pokud si hloubku podstromu A označíme jako h, B musí mít hloubku $h-1$, protože y je $\ominus$, atd. Jen nesmíme zapomenout, že v x jsme ještě $\ominus$ nepřepočítali (vede tam přeci šipka), takže ve skutečnosti je jeho levý podstrom o 2 hladiny hlubší než pravý (původní hloubky jsme na obrázku naznačili [v závorkách]). Po zrotování vyjdou u x i y znaménka $\odot$ a celková hloubka se nezmění, takže jsme hotovi.

Další možnost je y jako $\oplus$:

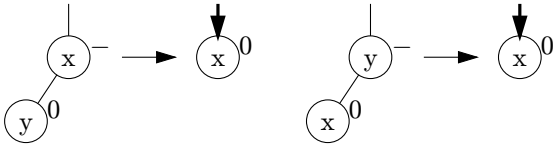


Tehdy se podíváme ještě o hladinu níž a provedeme dvojrotaci. (Nemůže se nám stát, že by z neexistovalo, protože jinak by v y nebylo $\oplus$.) Hloubky opět najdete na obrázku. Jelikož z může mít libovolné znaménko, jsou hloubky podstromů B a C buďto h nebo $h-1$, což značíme h^-. Podle toho pak vyjdou znaménka vrcholů x a y po rotaci. Každopádně vrchol z vždy obdrží $\odot$ a celková hloubka se nemění, takže končíme.

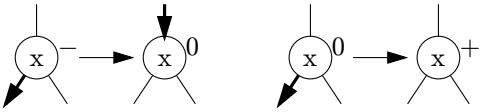
Poslední možnost je, že by y byl $\odot$, ale tu vyřešíme velmi snadno: všimneme si, že nemůže nastat. Kdykoliv totiž posíláme šipku nahoru, není pod ní $\odot$. (Kontrolní otázka: jak to, že $\oplus$ může nastat?)

Vyvažování po Deletu. Vyvažování po Deletu je trochu obtížnější, ale také se dá popsat pár obrázky. Nejdříve opět rozebereme základní situace: odebíráme

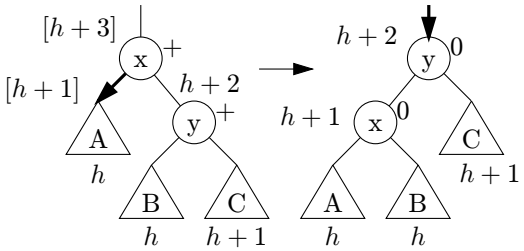
list (BÚNO levý) nebo vnitřní vrchol stupně 2 (tehdy ale musí být jeho jediný syn listem, jinak by to nebyl AVL strom):



Šipkou dolů značíme, že o patro výš posíláme informaci o tom, že se hloubka podstromu snížila o 1. Pokud šipku dostane vrchol typu $\ominus$ nebo $\odot$, vyřešíme to snadno:

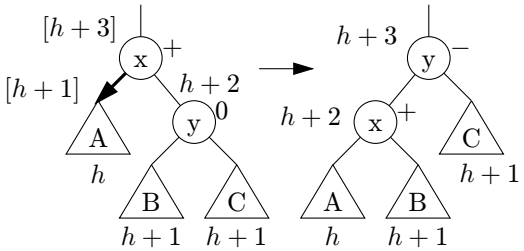


Problematické jsou tentokráte ty případy, kdy šipku dostane $\oplus$. Tehdy se musíme podívat na znaménko *opačného* syna a podle toho rotovat. První možnost je, že opačný syn má $\oplus$:



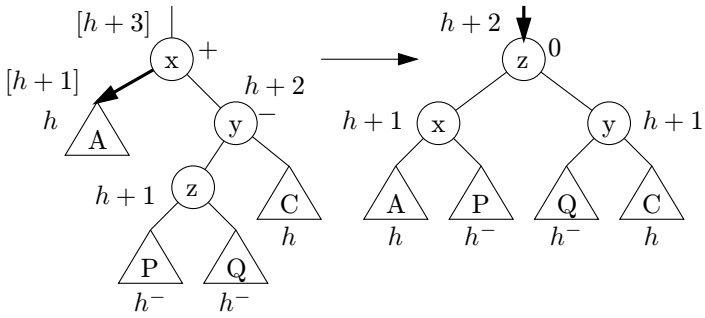
Tehdy provedeme rotaci vlevo, x i y získají nuly, ale celková hloubka stromu se sníží o hladinu, takže nezbývá, než poslat šipku o patro výš.

Pokud by y byl $\odot$:



Opět rotace vlevo, ale tentokráte se zastavíme, protože celková hloubka se nezměnila.

Poslední, nejkomplikovanější možnost je, že by y byl $\ominus$:



V tomto případě provedeme dvojrotaci (z určitě existuje, jelikož y je typu $\ominus$), vrcholy x a y obdrží znaménka v závislosti na původním znaménku vrcholu z a celý strom se snížil, takže pokračujeme o patro výš.

Happy end. Jak při Insertu, tak při Deletu se nám podařilo strom upravit tak, aby byl opět AVL stromem, a trvalo nám to lineárně s hloubkou stromu (konáme konstantní práci na každé hladině), čili stejně jako trvá Insert a Delete samotný. Jenže o AVL stromech jsme již dokázali, že mají hloubku vždy logaritmickou, takže jak hledání, tak Insert a Delete zvládneme v logaritmickém čase (vzhledem k aktuálnímu počtu prvků ve stromu).

Další typy stromů. AVL stromy samozřejmě nejsou jediný způsob, jak zavést stromovou datovou strukturu s logaritmicky rychlými operacemi. Další jsou třeba:

- *Červeno-Černé stromy* – ty si místo znamének vrcholy barví, každý je buďto červený nebo černý a platí, že nikdy nejsou dva červené vrcholy pod sebou a že na každé cestě z kořene do listu je stejný počet černých vrcholů. Opět je hloubka stromu logaritmická, po Insertu a Deletu barvy opravujeme přebarvováním na cestě do kořene a rotováním, jen je potřeba rozebrat podstatně více případů než u AVL stromů. (Za to jsme ale odměněni tím, že nikdy neděláme více než 2 rotace.)
- *2-3-stromy* – v jednom vrcholu nemáme uloženu jednu hodnotu, nýbrž jednu nebo dvě (a synové jsou pak 2 nebo 3, odtud název). Hloubka opět logaritmická, vyvažování řešíme pomocí spojování a rozdělování vrcholů.
- *Splay stromy* – nezavádíme žádnou vyvažovací podmínku, nýbrž definujeme, že kdykoliv pracujeme s nějakým vrcholem, vždy si jej vyrotujeme do kořene a pokud to jde, preferujeme dvojrotace. Takové operaci se říká Splay a dají se pomocí ní definovat operace ostatní: Find hodnotu najde a poté na ni zavolá Splay. Insert si nechá

vysplayovat předchůdce nové hodnoty a vloží nový vrchol mezi předchůdce a jeho pravého syna. Delete vysplayuje mazaný prvek, pak uvnitř pravého podstromu vysplayuje minimum, takže bude mít jen jednoho syna a můžeme jím tedy nahradit mazaný prvek v kořeni.

Jednotlivé operace samozřejmě mohou trvat až lineárně dlouho, ale dá se o nich dokázat, že jejich *amortizovaná* složitost je vždy $O(\log N)$. Tím chceme říci, že provést t po sobě jdoucích operací začínajících prázdným stromem trvá $O(t \cdot \log N)$ (některé operace mohou být pomalejší, ale to je vykoupeno větší rychlostí jiných). To u většiny použití stačí – datovou strukturu obvykle používáte uvnitř nějakého algoritmu a zajímá vás, jak dlouho běží všechny operace dohromady – a navíc je Splay stromy daleko snazší naprogramovat než nějaké vyvažované stromy. Mimo to mají Splay stromy i jiné krásné vlastnosti: přizpůsobují svůj tvar četnostem hledání, takže často hledané prvky jsou pak blíž ke kořeni, snadno se dají rozdělovat a spojovat atd.

- *Treapy* – randomizovaně vyvažované stromy: něco mezi stromem (tree) a haldou (heap). Každému prvku přiřadíme *váhu,* což je náhodné číslo z intervalu $\langle 0, 1\rangle$. Strom pak udržujeme uspořádaný stromově podle hodnot a haldově podle vah (všimněte si, že tím je jeho tvar určen jednoznačně). Insert a Delete opravují haldové uspořádání velmi jednoduše pomocí rotací. Časová složitost v průměrném případě je $O(\log N)$.
- *BB-α stromy* – zobecnění dokonalé vyváženosti jiným směrem: zvolíme si vhodné číslo α a vyžadujeme, aby se velikost podstromů každého vrcholu lišila maximálně α-krát (prázdné podstromy nějak ošetříme, abychom nedělili nulou; dokonalá vyváženost odpovídá $\alpha = 1$ [až na zaokrouhlování]). V každém vrcholu si budeme pamatovat, kolik vrcholů obsahuje podstrom, jehož je kořenem, a po Insertu a Deletu přepočítáme tyto hodnoty na cestě zpět do kořene a zkontrolujeme, jestli je strom ještě stále α-vyvážený. Pokud ne, najdeme nejvyšší místo, ve kterém se velikosti podstromů příliš liší, a vše od tohoto místa dolů znovu vytvoříme algoritmem na výrobu dokonale vyvážených stromů. Ten, pravda, běží v lineárním čase, ale čím větší podstrom přebudováváme, tím to děláme méně často, takže vyjde opět amortizovaně $O(\log N)$ na operaci.

Cvičení. Několik věcí, které se do kuchařky už nevešly, ale můžete si je zkusit vymyslet:

1. jak konstruovat dokonale vyvážené stromy
2. jak pomocí toho naprogramovat BB-α stromy

3. algoritmus, který k prvku ve stromu najde jeho následníka, což je prvek s nejbližší vyšší hodnotou (zde předpokládejte, že ke každému prvku máte uložený ukazatel na jeho otce)
4. jak vypsat celý strom tak, že začnete v minimu a budete postupně hledat následníky (i když nalezení následníka může trvat až $O(h)$, všimněte si, že projití celého stromu přes následníky bude lineární)
5. jak do vrcholů stromu ukládat různé pomocné informace, jako třeba počet vrcholů v podstromu každého vrcholu, a jak tyto informace při operacích se stromem udržovat (při Insertu, Deletu, rotaci)
6. že libovolný interval $\langle a, b\rangle$ lze rozložit na logaritmicky mnoho intervalů odpovídajících podstromům
7. a že zkombinováním předchozích dvou cvičení lze odpovídat i na otázky typu „kolik si strom pamatuje hodnot ze zadaného intervalu“ v logaritmickém čase...

Několik poznámek na závěr.

- Pokud záznamy můžeme jenom porovnávat, je binární vyhledávání nejlepší možné. Libovolné hledání založené na porovnávání lze totiž popsat binárním stromem a binární strom s N vrcholy musí mít vždy hloubku alespoň $\lfloor \log_2 N \rfloor$.
- Pokud bychom ale předpokládali, že se záznamy můžeme zacházet i jinak, dají se některé operace provádět i v konstantním čase (alespoň průměrně). K tomu se hodí například *hashování,* a to si popíšeme v některé z kuchařek v příštím ročníku KSP. Jeho nevýhodou ovšem je, že udržuje jenom množinu prvků, nikoliv uspořádání na ní, takže například nelze najít k zadanému prvku nejbližší vyšší.
- Pokud bychom připustili, že se mohou vyskytnout dva stejné záznamy, budou stromy stále fungovat, jen si musíme dát o něco větší pozor na to, co všechno při operacích se stromem může nastat.
- Jakpak přišly AVL stromy ke svému jménu? Inu, podle svých objevitelů pánů Adeľsona-Veľského a Landise.
- Rekurenci $A_d = 1 + A_{d-1} + A_{d-2}$, $A_1 = 1$, $A_2 = 2$ pro velikosti minimálních AVL stromů je samozřejmě možné vyřešit i přesně. Žádné překvapení se nekoná, objeví se totiž stará známá Fibonacciho čísla: $A_n = F_{n+2} - 1$.

16-5-K Kuchařka páté série – vyhledávání slov v textu

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 $O(p)$. Časová složitost celého algoritmu tedy bude $O(n+p)$. 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]);

  R:= 0;                                   { procházení textu }
  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 formálně nadefinujeme 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, 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*);

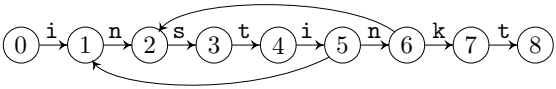
- $f : Q \to Q$ je *zpětná funkce,* která říká, do jakého stavu se má automat přesunout, pokud přechodová funkce není definována;
- $q_0 \in Q$ je *počáteční stav,* v němž se automat nachází na začátku výpočtu;
- $out : Q \to \mathcal{P}(\Sigma^*)$ je *výstupní funkce,* která každému stavu přiřazuje, 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.

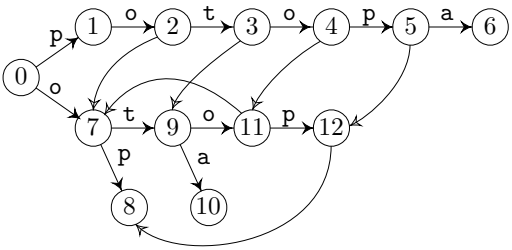
Příklad: Pro slovo `instinkt` by vyhledávací automat vypadal takto (zpětnou funkci jsme kreslili pouze tam, kde nevede do stavu 0):



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:

$$\begin{array}{ll} out(5) = \{\texttt{otop}, \texttt{op}\} & out(10) = \{\texttt{ota}\} \\ out(6) = \{\texttt{potopa}\} & out(12) = \{\texttt{otop}, \texttt{op}\} \\ out(8) = \{\texttt{op}\} & out(\text{ostatní}) = \emptyset. \end{array}$$

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 $O(n)$ až na vypisování výskytů, které poběží v čase $O(\textit{počet výskytů})$, 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 $\beta[1 \ldots K]$ 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 $O(P)$ 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 $O(P)$.

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 }
  g: array[1..MaxQ, Char] of Integer;      { přechodová a zpětná funkce: }
  f: array[0..MaxQ] of Integer;
  o: array[0..MaxQ] of Integer;            { obě části výstupní funkce: }
  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
  Q:= 0;                                   { vložíme všechna slova }
  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;

  f[0]:= 0; z[0]:= 0;              { zkonstruujeme zpětnou a výstupní fci }
  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;

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

Vzorová řešení

18-1-1 Dimenze X — Zbyněk Falt

Roboti se v naprosté většině došlých řešení šťastně shledali. Ne vždy však po setkání došlo ke zničení Dimenze X, neboť občas jim hledání trvalo tak dlouho, že se jejich zásoba plutonia rozpadla až na bismut, který se k výbuchu již tolik neměl.

Došlá řešení lze rozdělit na dvě zhruba stejně velké skupiny. V první prohledávali roboti své okolí do stále se zvětšující vzdálenosti a když narazili na hromadu šrotu toho druhého, tak na něj buď počkali, nebo v lepším případě mu šli naproti. Myšlenka je to správná, bohužel implementace pokulhávala. Většina zvětšovala amplitudu prohledávání o konstantu, což dává složitost $O(N^2)$. Pouze menšina amplitudu zvětšovala konstantněkrát, čehož výsledkem je pro Dimenzi X nepříznivější složitost $O(N)$.

Druhá skupina řešila úkol tak, že vyslala roboty nižší rychlostí libovolným směrem. Jeden z robotů tak zákonitě musel narazit na hromadu druhého z robotů, což pochopil jako povel zrychlit na plnou rychlost a dohnat tak robota, který se nerušeně vzdaloval stále nízkou rychlostí. Myšlenka je to opět správná, bohužel i tentokrát nebyla implementace vždy v pořádku. Většina totiž řešila snížení rychlosti tak, že robot dělal mezi posuny pauzu. V zadání však bylo, že robot umí udělat v jednom kroku pouze posun o 1 metr na sever nebo na jih (o zastavení tam řeč nebyla). Řešitelné to však je. Například udělat dva posuny jedním směrem a jeden posun zpět, atd. Obecně je ale jakékoliv řešení založené na této myšlence v čase $O(N)$.

Všechny algoritmy si vystačily s maximálně 3 proměnnými, což dává velmi příznivou paměťovou složitost $O(1)$. Bohužel paměťový obvod nemusel mít poruchou omezenou pouze kapacitu, ale i spolehlivost uchovávání informací a proto byla nejcennější řešení ta, které si kromě ukazatele pozice v programu nemusela pamatovat vůbec nic.

Abych byl konkrétní, tak uvádím příklad možného postupu, který pracuje v čase $O(N)$, nepotřebuje žádné pomocné proměnné a za které bylo možné získat plný počet bodů:

```
{ posun na sever nižší rychlostí dokud nenajdu druhou hromadu  }
repeat
  Posun_Na_Sever; Posun_Na_Sever; Posun_Na_Jih;
until Stojím_Na_Hromadě;

{ narazil jsem na hromadu druhého robota => zrychlím a doženu ho }
while true do
  Posun_Na_Sever;
```

18-1-2 Úřad — Jan Bulánek

U mnoha řešitelů byla znát vcelku oprávněná zášť vůči úřednímu šimlovi, jež mnohdy vyústila v zákeřné chyby, které měly znemožnit další bujení byrokracie. Ve snaze znemožniti kontrolorovi jejich odhalení je důvtipně skrývali v moři rekurze a záplavě cyklů. Někteří z vás se dokonce uchýlili k zákeřné fintě známé jako vypouštění komentářů a popisů řešení. Kvílení kontrolorů a otázky proč to děláš? a jak to funguje?, budiž jim odměnou za dobře odvedenou práci. Ale zpět k úloze.

Jak většina z vás správně poznala, použijeme zásobník. Pokud ze vstupu načteme otevírací závorku, pak ji uložíme na zásobník. Na vrcholu zásobníku tak bude vždy poslední nespárovaná otevírací závorka O. Pokud načtu zavírací závorku Z, mohou nastat tyto tři situace:

1) Barvy si odpovídají. Pak je vše v pořádku. Navíc je tato závorka použitá a už nikdy ji nebudu potřebovat. Můžeme ji tedy zezásobníku odebrat.
2) Barvy si neodpovídají. Takové uzávorkování nemůže být správné, protože O může být uzavřena až za Z. Zároveň ale i otevírací závorka pro Z se může nacházet jedině před O. Tím by se nám zkřížily dva šanony a to je chybné.
3) V zásobníku nic není. Velmi často opomíjená možnost nám říká, že nalevo od uzavírací závorky není žádná nepoužitá otevírací závorka. V tomto případě samozřejmě končíme s výpisem „ne“.

Pokud program dojde na konec vstupu, tak zjistí, zda je zásobník prázdný. Pokud ano, tak to znamená, že všechny otevírací závorky byly zavřeny a uzávorkování je korektní. V opačném případě jsme něco neuzavřeli a končíme s „ne“. Časová složitost je stejně jako paměťová $O(N)$, s každým prvkem provedeme konstantní počet operací a prvků v pomocném poli je nejvýše $N/2$.

Někteří z vás zbytečně načítali vstupní data do pole, i když to nebylo nutné. Naštěstí pro ně si tím asymptotickou složitost nezhoršili. Za chybějící nebo špatné popisy, odhady složitosti a hlavně zdůvodnění správnosti se dle míry provinění daly dohromady ztratit až dva body. Pokud se v programu vyskytla chyba, která způsobila nefunkčnost programu, tak jsem strhával dle nefunkčnosti až 4 body, podobně pro pomalá (typicky $O(N^2)$) řešení docházelo ke srážkám úměrným pomalosti.

```
program Kontrolor;
const MAX_N=1000;
var K,N:integer;
    vstup,akt:integer;
    zasobnik:array[1..MAX_N] of integer;
```

```
begin
     writeln('zadej počet barev: ');   readln(K);
     writeln('zadej počet šanonů: ');  readln(N);
     akt:=1; zasobnik[1]:=0;        { trik, aby zásobník nikdy nebyl prázdný }
     if (N mod 2=1) then N:=-1;     { ošetření lichého počtu šanonů}
     while(N>0)do begin
          writeln('zadej barvu šanonu: ');
          readln(vstup);
          if (vstup>0) then begin   { přidává nový šanon do zásobníku}
               akt:=akt+1;
               zasobnik[akt]:=vstup;
          end else begin            { uzavírá sanon}
               if (zasobnik[akt]=-vstup) then akt:=akt-1 { uzavírá se správný }
               else N:=-1;          { neuzavírá se správný, konec a odpověď ne }
          end;
          N:=N-1;
     end;
     if (N<>0) then writeln('ne')
     else writeln('ano');           { jinak je vstup bez chyby}
end.
```

18-1-3 Keřík — Jana Kravalová

Velká lístečková žranice skončila ve většině případů dle libosti nenasytné píďalky. Kdo už jednou vymyslel ten správný výpočet pohybu housenky, neudělal už většinou žádnou chybu, takže bodové zisky od spokojeně nadládbnuté žížalky byly hojné. Zbytek řešitelů se více či méně úspěšně snažil spouštět z každého význačného bodu prohledávání do hloubky, případně generovat všechny možné dvojice význačných bodů (dále je budeme v souladu s grafovou teorií nazývat vrcholy), za což zaplatil zvýšením časové složitosti. Taková řešení pak získala maximálně 5 bodů. Jak tedy nakrmit žravou housenku v lineárním čase?

Nejprve je třeba si ujasnit, jaký typ grafu náš keřík vlastně je. Víme, že se jedná o obecný strom s neorientovanými hranami. Někteří předpokládali, že máme dán zakořeněný strom s hranami orientovanými a snažili se hledat vrchol, do nějž nevede žádná hrana, aby z něj pak vesele začali počítat. Nikdo ale neříkal, že graf má hrany orientované, právě naopak, logicky hrany musí být obousměrné. Navíc nám nikdo nedal záruku, že takový vrchol, ze kterého nevede žádná hrana, je právě jeden!

Mějme tedy náš obecný neorientovaný strom. Tento strom si „zakořeníme“, čili jeden vrchol prohlásíme za kořen, jeho sousedy za jeho syny, jejich sousedy za syny synů, atd. Uvidíme, že náš algoritmus tím nijak nepoškodíme, děláme to jenom proto, abychom si vše mohli lépe představit. Navíc uvidíme, že kořenem může být libovolný vrchol.

Jistě mi budete věřit, že nejvýživnější cesta musí začínat a končit v listu, to jest ve vrcholu stupně jedna. Kdyby tomu tak nebylo a cesta by končila

v nějakém vrcholu stupně > 1, pak bychom cestu mohli z tohoto vrcholu ještě prodloužit až do nějakého listu.

Dobře, ale pak cesta musí vypadat tak, že začíná v nějakém listu, pak míří nahoru směrem ke kořeni, dorazí do nějakého vrcholu, ve kterém se jakoby láme a míří zase dolů až do nějakého jiného listu a skončí.

Představme si, že se nacházíme v nějakém vrcholu v a chtěli bychom vědět, jaká nejlepší cesta přes něj prochází, čili láme se v něm. Hodilo by se nám, kdybychom u každého jeho syna měli spočítáno, jaká nejlepší cesta vede od nějakého listu v jeho podstromě až do něj. Pak bychom si ze všech synů vrcholu v vybrali ty dva nejvýhodnější, sečetli jejich hodnoty, přičetli bychom počet lístečků ve vrcholu v a kýženou výživnost cesty lámající se ve vrcholu v bychom znali. Je pro nás těžké spočítat tyto hodnoty u všech synů? Kdepak, uděláme to úplně stejným způsobem, tedy rekurzí.

Algoritmus tedy funguje tak, že pro každičký vrchol ve stromě spočítáme dvě hodnoty: Jak výhodná cesta se v něm láme (tuto informaci získáme rekurzivním voláním synů) a jaká nejlepší cesta vedoucí z nějakého listu v některém z jeho podstromů v něm končí (tuto informaci předáme nahoru jeho otci).

Nakonec stačí projít všechny vrcholy a najít ten s nejvyšší hodnotou lámající se cesty a z něj pak spustit výpis na obě strany. Samozřejmě, že jedna strana nemusí existovat, to například když má strom tvar cesty (např. $1 \to 2 \to 3 \to 4$).

Jak rychlá je tato rekurze? Do každého vrcholu přijdu jenom jednou, tedy $O(N)$, a na každou hranu se podívám také jenom jednou, tedy $O(M)$. My ale víme, že pracujeme se stromem, tedy pro něj platí, že $M = N - 1$, takže složitost rekurze je opravdu $O(N)$. Načtení hodnot, závěrečné vyhledání nejvýhodnějšího vrcholu a výpis cesty nám též trvá $O(N)$. Paměťová složitost je při reprezentaci seznamem sousedů také lineární.

```
Program Kerik;
const MaxN=100;
var Hrany: array[1..MaxN] of integer;    { reprezentace grafu seznamem sousedů }
    Vrcholy: array[1..MaxN+1] of integer;
    Listky: array[1..MaxN] of integer;          { počet lístků v daném vrcholu }
    NejCesta: array[1..MaxN] of integer;   { nejlepší cesta z listu do vrcholu }
    N,max_cesta, max_vrchol, max_syn1, max_syn2: integer;

procedure Ohodnot(v,o:integer);         { najde vrchol, kde se láme nej. cesta }
var max1,max2,p_max_syn1,p_max_syn2,i: integer;
begin
   max1:=0; max2:=0; p_max_syn1:=-1; p_max_syn2:=-1;{ hledáme 2 nej. podstromy }
  for i:=Vrcholy[v] to Vrcholy[v+1]-1 do { ohodnoť všechny podstromy vrcholu v }
    if Hrany[i]<>o then begin                       { nebudeme se vracet zpátky }
      Ohodnot(Hrany[i],v);             { zjisti výživnost cesty končící v synu }
       if NejCesta[Hrany[i]]>=max1 then begin                 { zatím nej. syn }
        max2:=max1; max1:=NejCesta[Hrany[i]];
```

```
        p_max_syn2:=p_max_syn1; p_max_syn1:=Hrany[i];
      end else if NejCesta[Hrany[i]]>=max2 then begin      { zatím 2. nej. syn }
        max2:=NejCesta[Hrany[i]]; p_max_syn2:=Hrany[i];
      end;
    end;
   NejCesta[v] := Listky[v] + max1;  { nej. cesta z nějakého listu končící zde }
  if Listky[v] + max1 + max2 > max_cesta then begin         { máme lepší cestu }
    max_cesta:=Listky[v] + max1 + max2;
    max_vrchol:=v; max_syn1:=p_max_syn1; max_syn2:=p_max_syn2;
  end;
end;

procedure Vypis(v,o,smer:integer);
var i,max,max_syn:integer;
begin
  if smer = 1 then write(v,' ');         { jdeme z vrcholu dolů, chceme prefix }
  max:=-1;
  for i:=Vrcholy[v] to Vrcholy[v+1]-1 do begin       { najdeme nejlepšího syna }
    if (Hrany[i]<>o) and (NejCesta[Hrany[i]]>max) then begin
      max:=NejCesta[Hrany[i]]; max_syn:=Hrany[i];
    end;
  end;
  if max<>-1 then Vypis(max_syn,v,smer);        { pokud existuje, vypíšeme jej }
  if smer = -1 then write(v,' ');       { jdeme zespoda nahoru, chceme postfix }
end;

begin
  { Zde se načítají data... }
  if N = 0 then begin
    writeln('Chudinka housenka, asi se moc nenažere...');  exit;
  end;
  max_syn1:=-1; max_syn2:=-1;  max_cesta:=-1;
  Ohodnot(1,-1);
  if max_syn1 <> -1 then Vypis(max_syn1,max_vrchol,-1);     { výpis cesty }
  write(max_vrchol,' ');
  if max_syn2 <> -1 then Vypis(max_syn2,max_vrchol,1);
  writeln;
end.
```

18-1-4 Dortík — Tomáš Valla

Sfoukněte svíce, uchopte do ruky nůž, nakrojte dort a vydatně se posilněte na následující vzorové řešení.

Nejprve několik poznámek k došlým řešením. Někteří řešitelé si bohužel do zadání domysleli některé dodatečné podmínky, které jsme ve skutečnosti nikde netvrdili. Například to, že úhly jsou vždy jen celá čísla a že se nikdy nevyskytnou dvě svíčky na stejném místě. Takové podmínky ovšem úlohu zjednodušují, proto jsem za takové věci strhával body. Pokud si příště nebudete jisti, řešte raději tu těžší variantu, případně není problém se nás zeptat.

Ale teď již k věci. Ukážeme si řešení, které kdyby dostalo na vstupu úhly svíček vzestupně setříděné, seběhlo by v lineárním čase vzhledem k počtu svíček. Nejprve si uvědomíme, že mezi každými dvěma sousedními svíčkami v K-úhelníku musí být úhel $360/K$. Další naše pozorování bude, že každá svíčka může ležet maximálně v jednom K-úhelníku.

Mějme tedy na vstupu setříděné úhly U. Dle našich úvah z nich teď můžeme klidně vyházet duplikáty, tedy ze svíček na stejné pozici nechat jen jednu. Poté projdeme seznam svíček od nejmenšího úhlu k největšímu a budeme si pro i-tou svíčku s úhlem U_i počítat následující věci: Kde (a jestli vůbec) má nějakého „předchůdce“ P_i na potenciálním K-úhelníku (tedy svíčku na úhlu $U_i - 360/K$), a kolikátá svíčka C_i s rozestupem $360/K$ v řadě to je (čili kde se v potenciálním K-úhelníku nachází).

Když budeme mít tato data spočítána, stačí se podívat, jestli existuje svíčka s právě $K-1$ pravidelnými předchůdci. Jestliže ano, potom touto svíčkou končí K-úhelník a proskáčeme skrz zpětné odkazy P po svíčkách a výsledek vypíšeme.

Zbývá si rozmyslet, jak efektivně počítat žádaná data. Použijeme k tomu metodu dvou posuvných indexů. Mějme indexy i a j, kde j bude vždycky ten více vepředu. Na začátku nastavíme i na 1 a j na 2. Pokud je rozdíl úhlů U_i a U_j roven $360/K$, j-tá svíčka má předchůdce i a je $(C_i + 1)$-ní v pořadí, nastavíme tedy příslušně hodnoty P_j a C_j. Pokud je $U_j - U_i > 360/K$, tedy svíčky jsou příliš daleko, zvýšíme i o 1, čímž je k sobě přiblížíme, pokud $U_j - U_i < 360/K$, zvýšíme o 1 j, čímž je oddálíme. Všimněte si, že pokud jsou úhly setříděné, náš postup skutečně najde všechny dvojice správně vzdálených svíček.

Vyházení duplikátů zvládneme jedním průchodem přes U v lineárním čase, stejně tak výpis výsledků proskákáním pole P a C. Při posouvání dvou indexů oba pouze rostou, tedy se nad každým prvkem vykonají maximálně dvě operace, máme tedy časovou složitost $O(N)$. Ještě je potřeba započítat čas na třídění na počátku, to umíme například použitím nějakého rychlého kuchařkového algoritmu v čase $O(N \log N)$, dohromady tak máme časovou složitost $O(N \log N)$. Všimněte si, že třídění je nejpomalejší částí našeho algoritmu. Paměťová složitost je $O(N)$, pamatujeme si tři pole délky N.

Ještě poznámka k programu: ve skutečnosti bychom se obešli bez jednoho z polí P či C, pro větší srozumitelnost však v programu používáme obě.

```
program dortik;
const max = 1000;
var U: array[1..max] of real;              {seznam úhlů svíček}
    P, C: array[1..max] of integer;        {čísla předchůdců a jejich počet}
    i, j, N, K: integer;
    dif: real;

begin
   read(N); read(K);
```

```
   for i:=1 to N do begin
      read(U[i]);
      P[i]:=0; C[i]:=0
   end;

   {setřiď úhly v U a vyházej duplikáty - vynecháme}
   i:=1; j:=2;
   while j <= N do begin
      dif:=U[j] - U[i] - 360/K;
      if abs(dif) < 0.00001 then begin    {našli jsme vhodného následníka}
         P[j]:=i;
         C[j]:=C[i]+1;
         inc(j)
      end else if dif > 0 then inc(i)
      else inc(j)
   end;
   for i:=1 to N do if C[i] = K-1 then begin      {našli jsme konec K-úhelníku}
         j:=i;
         writeln('Svíčkový k-úhelník:');
         repeat writeln(U[j]); j:=P[j]
         until j=0
      end
end.
```

18-1-5 Matlalové — Petr Škoda

Matlalové jsou již za vysokým drátěným plotem a jen občas hromada přebytečného pletiva zaclání ve výhledu. Spíše byl problém postavit plot dříve, než Matlalové zase odletí.

Nebudu déle zdržovat a rovnou přejdu ke vzorovému řešení. Stoly si rozdělíme na dvě vodorovné a dvě svislé hrany stolu. Nejprve zjistíme, které vodorovné hrany tvoří obvod stolů. Pak můžeme stoly otočit o 90° a použít stejný algoritmus na vertikální hrany. Proto se zabývejme pouze horizontálními hranami.

Představme si horizontální přímku, kterou posunujeme přes naši soustavu stolů, hledanému sjednocení stolů budeme říkat *oblast*. Jak přecházíme přímkou do vnitřku oblasti a ven z ní, připočítáváme tyto hrany přechodu k celkovému obvodu. Horizontální část obvodu se může změnit pouze na místě, kde začíná nebo končí nějaký stůl vodorovnou hranou. To je hlavní idea algoritmu.

Mějme tedy pole $2n$ vodorovných hran. Zvolíme si směr průchodu podle rostoucí y-nové souřadnice. U každé hrany si zapamatujeme

- y – y-ovou souřadnici
- `left` – souřadnici x počátku hrany
- `right` – souřadnici x konce hrany
- `open` – zda je hrana otevírající, tedy projdeme jí dříve než druhou hranou stolu

Nyní pole setřídíme podle dvou kritérií. Prvním je y-nová souřadnice. Pokud ji mají dvě hrany stejnou, pak upřednostníme tu, která je otevírající. To proto, abychom mezi stoly stojícími těsně vedle sebe nepostavili plot.

Při průchodu přímkou přes oblast můžeme na přímce zobrazit oblast jako několik intervalů. Pokud si budeme tyto intervaly pamatovat, změna intervalu znamená konec nebo začátek oblasti čili příspěvek do obvodu. Je vidět, že intervaly mohou začínat a končit pouze na místě, kde začínají nebo končí stoly. Těchto bodů je maximálně $2n$. Místo celé přímky si stačí pamatovat pouze $2n-1$ bloků. Intervaly se pak mohou skládat až z $O(n)$ těchto bloků. Jak budeme aktualizovat intervaly? Pokud projdeme otevírající hranou, znamená to, že jsme uvnitř nějakého stolu, pokud projdeme ukončující hranou, právě jsme stůl opustili. Stolů ale může na sobě ležet více, a proto si pro každý blok zapamatujeme číslo c_i, kolik stolů je právě na jeho místě. Projdeme hrany jednu po druhé tak, jak je máme setříděny a pokud je to hrana otevírající, přidáme interval na místě hrany, jinak interval odebereme. Přidání a odebrání intervalu znamená přičtení nebo odečtení jedničky od c_i všech bloků, do kterých hrana zasahuje. Pokud se přitom změní c_i některého bloku z 0 na 1 nebo z 1 na 0, znamená to, že jsme na tomto bloku vstoupili do oblasti nebo ji opustili, a proto délku tohoto bloku přičteme k obvodu. Říkáme, že blok je pokryt, pokud má $c_i > 0$.

Popsaný algoritmus spočte horizontální obvod jako součet délek bloků při změně jejich pokrytí. Jak dlouho mu to bude trvat? Třídění nám bude trvat čas $O(n \log n)$. Pak pro každou hranu aktualizujeme $O(n)$ bloků. Celkem tedy $O(n^2)$. Paměti spotřebujeme pouze lineárně – $O(n)$.

Pokud si budeme intervaly uchovávat trochu chytřeji, dá se časová složitost trochu zlepšit. Použijeme k tomu tzv. *intervalový strom*. Intervalový strom je binární vyvážený strom, který v našem případě vypadá následovně. Každému uzlu přiřadíme interval. Listům přiřadíme naše bloky, jak je známe, seřazené zleva doprava. Vnitřnímu uzlu přiřadíme interval daný sjednocením intervalů jeho synů. Kořen stromu má tedy přiřazen celý interval. Protože strom je vyvážený a má $O(n)$ listů, je jeho hloubka $O(\log n)$. Každý uzel má tyto vlastnosti:

- `left` – začátek intervalu
- `right` – konec intervalu
- `covered` – pokrytí intervalu
- `tables` – počet stolů, které ho úplně překrývají

Potřebujeme umět přidat do stromu a odebrat z něj interval v lepším čase než $O(n)$. Jak asi u stromu očekáváte, bude to $O(\log n)$. Protože se interval skládá až z $O(n)$ bloků, nemůžeme si dovolit při jeho přidání zaktualizovat všechny bloky, z kterých se skládá. Rozmyslíme si, že se každý interval dá rozložit do $O(\log n)$ intervalů reprezentovaných uzly našeho stromu. Tento roz-

klad najdeme podobně jako při přidávání intervalu, označme ho I. Budeme ho realizovat rekurzivní funkcí `find`. Začneme v kořeni stromu. Aktuální uzel označme u, interval uzlu označme I_u. Pokud I a I_u mají prázdný průnik, skončíme. Pokud I_u je podinterval I, I_u je jeden z hledaných intervalů. Jinak se částečně překrývají a zavoláme funkci `find` na oba syny. Průběh volání funkce bude vypadat takto. Z kořene projdeme po synech až k uzlu, kde interval I zasahuje do obou synů. Zde se průchod rozdělí na dvě větve. Jedna jde po levé straně intervalu, druhá po pravé. Všechny intervaly mezi nimi spadají zcela do intervalu I. Zkuste si nakreslit obrázek. Protože při zavolání funkce `find` na kořen projdeme maximálně dvě cesty z kořene k listům, je časová složitost této operace $O(\log n)$.

Vkládání a odebírání intervalu bude podobné funkci `find`. V každém z uzlu, na který se interval rozložil, aktualizujeme vlastnosti `tables` a `covered`. Vlastnost `tables` je počet stolů, jejichž rozklad intervalu obsahuje tento uzel. Pokrytí intervalu, `covered`, udává v reálných souřadnicích, kolik z intervalu je pokryto stoly. Pokrytí je větší než nula i v případě, kdy `tables` je rovno nula a některé intervaly v jeho synech jsou pokryty. Tyto vlastnosti se vztahují pouze k podstromům daného uzlu, nikoli k jeho rodičům. Rozmyslete si, že je dokážeme při vkládání a odebíraní intervalu aktualizovat. Funkce pro vkládání a odebírání bude vracet změnu pokrytí v daném podstromě. Proto pokud je interval uzlu pod stolem a uvnitř jeho podstromu se změní pokrytí, tento uzel ho znuluje. Jinak se propaguje až do kořene a nakonec je celková změna pokrytí přičtena k počítanému obvodu.

Protože přidání a odebrání intervalu nám trvá čas $O(\log n)$ a stálé máme celkem $2n$ hran, celkový čas algoritmu je $O(n \log n)$. Paměťová složitost zůstala lineární.

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 1000

struct Edge {
    double main, left, right;
    int open;
};
struct Node {
    double covered;
    int tables;
    double left, right;
    int leaf;
};
int edgecmp (const void * a, const void * b) {
    struct Edge * p = a, * q = b;
    if (p->main != q->main) return p->main - q->main;
    if (p->open != q->open) return p->open ? -1 : 1;
    return 0; }
```

```
int n, s;
double perim;
struct Edge ve[2 * MAXN], he[2 * MAXN];
struct Node tree[8 * MAXN + 1];

void buildTree (struct Edge * f, int p, int l, int r) {
  tree[p].covered = 0; tree[p].tables = 0;
  tree[p].left = f[l].main; tree[p].right = f[r].main;
  tree[p].leaf = 1;

  if (r - l == 1) return;
  int m = (l + r) / 2;
  tree[p].leaf = 0;
  buildTree (f, 2*p, l, m);
  buildTree (f, 2*p+1, m, r);
}

double update (int p, struct Edge * edge) {
  if (edge->right <= tree[p].left || edge->left >= tree[p].right) return 0;

  double sum = 0;
  if (edge->left <= tree[p].left && edge->right >= tree[p].right) {
    if (edge->open && !tree[p].tables++)
      sum = tree[p].right - tree[p].left - tree[p].covered;
    else if (!edge->open && !--tree[p].tables) {
      sum = tree[p].right - tree[p].left;
      if (!tree[p].leaf) sum -= tree[2*p].covered + tree[2*p+1].covered;
    }
  } else {
    sum = update (2*p, edge) + update (2*p+1, edge);
    if (tree[p].tables) sum = 0;
  }

  if (tree[p].tables) tree[p].covered = tree[p].right - tree[p].left;
  else tree[p].covered = tree[p].leaf ? 0 : tree[2*p].covered + tree[2*p+1].covered;
  return sum;
}

void makeEdge (struct Edge * edge, double main, double left, double right, int open) {
  edge->main = main; edge->open = open;
  edge->left = left; edge->right = right;
}

int main (void) {
  scanf ("%d", &n);
  s = 2 * n;

  int i;
  for (i = 0; i < n; i++) {
    double x, y, w, h;
    scanf ("%lf%lf%lf%lf", &x, &y, &w, &h);
    makeEdge (he + 2 * i , y - h, x, x + w, 1);
    makeEdge (he + 2 * i + 1, y, x, x + w, 0);
    makeEdge (ve + 2 * i , x, y - h, y, 1);
    makeEdge (ve + 2 * i + 1, x + w, y - h, y, 0);
  }
```

```
    qsort (he, s, sizeof (struct Edge), &edgecmp);
    qsort (ve, s, sizeof (struct Edge), &edgecmp);

    buildTree (ve, 1, 0, s - 1);
    for (i = 0; i < s; i++) perim += update (1, &he[i]);

    buildTree (he, 1, 0, s - 1);
    for (i = 0; i < s; i++) perim += update (1, &ve[i]);

    printf ("%2.2f\n", perim);
    return 0;
}
```

18-1-6 Kompilované komplikátory — Zdeněk Dvořák

Jak si mnoho řešitelů uvědomilo, tuto úlohu šlo řešit i podstatně přímočařeji než v textu seriálu naznačeným postupem, například konstrukci syntaktického stromu lze přeskočit a generovat rovnou požadovaný mezikód. My si nejprve implementujeme jedno takové jednoduché řešení a poté si ukážeme, jak si ho vylepšit – kvůli tomu již bude nutné se přidržet postupu popsaného v seriálu. Abychom šetřili naše lesy a nervy méně otrlých řešitelů, asi 700-řádkový program k tomuto rozšířenému řešení zde netiskneme. Můžete ho nalézt na adrese *http://ksp.mff.cuni.cz/tasks/18/ksp1816b.c*.

Lexikální analýza je v našem případě triviální – načteme znak ze vstupu a rozhodneme, zda je to jeden z operátorů. Je-li tomu tak, rovnou vrátíme jemu odpovídající token. Další možností je začátek jména identifikátoru nebo číslo, pak načteme jeho zbytek.

V jednoduchém řešení bude sémantická analýza spojena se syntaktickou a místo stavby syntaktického stromu budeme rovnou vypisovat výpočet v mezikódu. Pro syntaktickou analýzu se prakticky se používají dva hlavní přístupy. Jeden z nich je zkonstruovat si zásobníkový automat, který rozpoznává danou gramatiku. Tento postup je vysvětlen například v řešení úlohy 9-3-3. Druhý přístup je složit analyzátor ze vzájemně rekurzivních funkcí, které odpovídají symbolům gramatiky. Implementace tohoto postupu bývá pro člověka o něco čitelnější a dají se v ní lépe ošetřovat chyby a jiné speciální případy. My se přidržíme druhého postupu. Syntaktickou analýzu budou zajišťovat funkce `cti_vyraz`, která zpracovává celý výraz nebo jeho podvýraz uvnitř závorek, `cti_faktor`, která zpracovává podvýraz, jehož operátory jsou násobení či dělení, a `cti_term`, která vyhodnotí podvýraz tvořený identifikátorem, číslem, nebo uzávorkovaným výrazem. Každá z těchto funkcí přečte co nejdelší kus kódu, který jí odpovídá, vypíše příkazy nutné pro jeho vyhodnocení a vrátí identifikátor proměnné, v níž je uložena jeho hodnota. Například funkce `cti_vyraz` pomocí funkce `cti_faktor` čte postupně kusy výrazu oddělené znaménky plus a minus, dokud nenarazí na konec výrazu či závorky, a sčítá či odčítá odpovídající hodnoty. Funkce `cti_faktor` se chová podobně, volá `cti_term` a kontroluje,

zda po nich následuje krát či děleno. Funkce `cti_term` se podívá, zda následuje proměnná či číslo (pak ho rovnou vrátí), nebo otevírací závorka, na jejíž vnitřek zavolá `cti_vyraz`. Každá z těchto funkcí také posune ukazatel ve vstupu na první znak, který nezpracovala.

Celý tento postup lze realizovat s paměťovou i časovou složitostí lineární v délce zadaného výrazu. Pro časovou složitost stačí nahlédnout, že je omezená počtem volání funkce `cti_term`, a ta vždy načte alespoň jeden token ze vstupu.

Nyní si popíšeme možná vylepšení tohoto postupu. U každé fáze si řekneme něco k tomu, co jde udělat lépe. Mimo jiné se budeme zabývat zotavením se z chyb. Pokud se v kódu programu vyskytne chyba (v našem případě třeba nespárované závorky), je poněkud nešikovné s překladem okamžitě skončit, například proto, že už se nevypíšou hlášení pro další chyby. Nejde tedy opravit všechny chyby naráz a je nutné po každé z nich program znovu kompilovat, což může být dost pomalé. Je zřejmě lepší se s chybou nějak vypořádat a pokračovat v překladu.

Smysluplné ošetření chyb při lexikální analýze je obtížné, protože v této fázi toho o vstupu mnoho nevíme. Chyby se proto řeší prostým zahozením nerozpoznaných znaků. Dojde-li k chybě v syntaktické analýze (například proto, že po sobě následují dva operátory a podobně), příslušná funkce si domyslí nějaký token, který se jí hodí, nebo ten aktuální zahodí, podle toho, co dává víc smysl.

Co se týče vylepšení lexikální analýzy, všimneme si, že syntaktická analýza čte vstup postupně a nikdy se nevrací. Je tedy zbytečné si vstup rozložený na tokeny pamatovat celý. Lexikální analýzu proto budeme realizovat funkcí, která ze vstupu načte a vrátí další token (`dalsi_token`), a sémantická analýza si ji bude volat podle potřeby. Ve skutečnosti se občas hodí se ve vstupu o jeden token vrátit – například když `cti_faktor` narazí na plus, ukončí se, ale toto plus by měla zpracovat funkce `cti_vyraz`. Proto `cti_faktor` nejdřív vrátí plus zpět do vstupu. K tomu slouží funkce `vrat_token`.

Dalším drobným trikem je, že si udržujeme tabulku identifikátorů jménem `hodnota_na_promennou`, a když načteme dvakrát identifikátor se stejným jménem, vrátíme místo něj jeho pořadí v tabulce, takže nemusíme nikde dál testovat, zda jsou dva identifikátory stejné (`promenna_na_hodnotu` je vlastně hešovací tabulka, která nám umožní záznamy v tabulce `hodnota_na_promennou` hledat rychle – víc k hešování viz kuchařka druhé série sedmnáctého ročníku).

Syntaktickou analýzu oddělíme od sémantické. Syntaktická analýza už nebude přímo generovat mezikód, ale místo toho každému zpracovanému podvýrazu přiřadí nějaké číslo. Tato čísla budou taková, že výrazy s různou hodnotou dostanou vždy různé číslo, zatímco výrazy se stejnou hodnotou dostanou stejné číslo – např. `x + y` a `x - y` dostanou různá čísla, protože jejich hodnoty se mohou lišit, zatímco výrazům `x - x` a `0` bude přiřazeno stejné číslo. Uděláme to tak, že

si udržujeme tabulku hodnot výrazů, které jsme již viděli (`hodnota_na_vyraz` a `vyraz_na_hodnotu`, opět používáme hešování), v níž si pamatujeme, jak se každé číslo hodnoty spočítá. Pokud narazíme na výraz, který už v tabulce je, vrátíme jeho číslo, jinak mu přidělíme nové číslo a přidáme ho do tabulky. Podrobnější vysvětlení tohoto postupu viz řešení úlohy 17-2-1. Snadno nahlédneme, že toto je v podstatě jen jiná reprezentace syntaktického stromu – čísla hodnot odpovídají vrcholům a abychom určili syny vrcholu, podíváme se do tabulky `hodnota_na_vyraz`. Číslování hodnot nám ale umožní zajistit, že stejnou hodnotu nebudeme počítat dvakrát – když na ni narazíme podruhé, budeme místo ní používat pomocnou proměnnou, do níž jsme ji poprvé spočítali.

Navíc se nám toto očíslování hodnot hodí při zjednodušování výrazů. Budeme chtít rovnou vyhodnocovat konstantní výrazy a také aplikovat jednoduché algebraické identity typu `x - x = 0`. Abychom mohli tuto optimalizaci provést, je potřeba zjistit, zda oba operandy minusu jsou stejné. Vzhledem k tomu, jak si výrazy reprezentujeme, stačí porovnat čísla jejich hodnot, není potřeba procházet stromy výrazů a ověřovat, zda si odpovídají (to by nám mohlo zhoršit časovou složitost na kvadratickou). Zjednodušování výrazů provádíme tak, že kdykoliv vytváříme vrchol stromu, který odpovídá nějakému operátoru, podíváme se na jeho operandy a určíme, zda ho můžeme nějak zjednodušit. Toto provádí funkce `postav_strom`. Například pokud vyhodnocujeme výraz $(x + 1) + 2$, `postav_strom` dostane plus, jehož parametry mají čísla hodnot h_1 a h_2, a z tabulek zjistíme, že h_2 je ve skutečnosti konstanta 1, a že h_1 je součet hodnot h_3 a h_4, kde h_4 je konstanta 2. Konstanty sečteme, dostaneme 3 a zjistíme si, že číslo hodnoty pro 3 je h_5. Zjednodušený výraz tedy bude $h_3 + h_5$, což odpovídá $x + 3$. Tomuto výrazu přidělíme nové číslo hodnoty h_6, dáme ho do tabulek a h_6 vrátíme.

Jednou z komplikací, které se v tomto řešení vyhýbáme, ale prakticky je nutné se jí zabývat, je provedení vedlejších účinků zjednodušovaných výrazů. Například máme-li výraz `funkce () * 0`, jeho hodnota je vždy 0, ale přesto je nutné funkci zavolat. Prakticky tedy nestačí pouze vracet hodnotu zjednodušeného výrazu, ale je nutné zajistit, aby se také provedly tyto vedlejší akce. V našem případě jediný takový problém je dělení 0 – například výraz `x/y - x/y` by mohl způsobit chybu, pokud $y = 0$, ale zjednodušený výraz 0 chybu způsobit nemůže. Tento problém pro jednoduchost řešit nebudeme – konec konců, v platném programu se dělení nulou vyskytnout nesmí (až na výjimky).

Vedlejší účinky by navíc mohly měnit hodnoty proměnných, které se ve výrazu používají. Například při vyhodnocování výrazů v C je nutné při dosažení *sequence pointu* (což je místo, na kterém je zaručeno, že se vedlejší účinky vykonají) upravit čísla hodnot ovlivněných proměnných.

Poslední fází je sémantická analýza, tj. expanze do mezikódu. V ní vypíšeme výrazy nutné pro spočtení hodnoty, která odpovídá číslu hodnoty celého

výrazu. Podíváme se tedy do tabulek, jak jsme toto číslo dostali, rekurzivně vyhodnotíme čísla hodnot podvýrazů a vypíšeme příslušnou operaci, která z nich spočte výsledek. Abychom nepočítali nějaký výraz dvakrát, u každého čísla hodnoty si pamatujeme, zda jsme ho už počítali, a když ho potřebujeme podruhé, použijeme místo něj příslušnou pomocnou proměnnou. Přidělování jmen pomocným proměnným řešíme jednoduše, k prefixu `tmp` připojíme číslo hodnoty.

Je snadné si rozmyslet, že všechna tato rozšíření fungují v lineárním čase.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#define MAX_DELKA 1000

  /* Typ tokenu – +, -, *, /, (, ), 'a' pro proměnnou, '0' pro číslo. */
typedef char token;

char vstup[MAX_DELKA];
char tokeny[MAX_DELKA];
char *val[MAX_DELKA];                          /* Řetězce hodnot proměnných a čísel. */

void lex (void) {                              /* Lexikální analýza */
    unsigned index = 0, atok = 0, zac;
    char znak;

    while (1) {
        znak = vstup[index++];

        while (isspace (znak)) znak = vstup[index++];    /* Přeskočíme mezery. */
        switch (znak) {
          case 0:                              /* Na konec výrazu si přidáme ')', */
            tokeny[atok++] = ')'; return;      /* abychom ho nemuseli ošetřovat. */
          case '+': case '-': case '*': case '/': case '(': case ')':
            tokeny[atok++] = znak; break;
          default:
            zac = index - 1;
            if (isdigit (znak)) {
                tokeny[atok] = 'a';
                while (isdigit (znak = vstup[index])) index++;
            } else if (isalpha (znak)) {
                tokeny[atok] = '0';
                while (isalnum (znak = vstup[index])) index++;
            } else abort ();

            vstup[index] = 0;
            val[atok++] = strdup (vstup + zac);
            vstup[index] = znak;
            break;
        }
    }
}
```

```
char *vyhodnot (char *levy, char znam, char *pravy) {  /* Syntaktická */
    static unsigned pom = 0;                           /* a semantická analýza */
    char *prom = malloc (20);

    sprintf (prom, "_tmp%d", pom++);
    printf ("assign %s (%s %c %s)\n", prom, levy, znam, pravy);
    return prom;
}

unsigned pos;                                          /* Aktuální pozice ve vstupu. */

char *cti_term (void);
char *cti_faktor (void);

char *cti_vyraz (void) {
    char *levy, *pravy, znam;
    levy = cti_faktor ();
    while (1) {                                        /* Čteme zbývající faktory */
        if (tokeny[pos] == ')') return levy;
        znam = tokeny[pos++];
        pravy = cti_faktor ();
        levy = vyhodnot (levy, znam, pravy);
    }
}

char *cti_faktor (void) {
    char *levy, *pravy, znam;
    levy = cti_term ();
    while (1) {
        znam = tokeny[pos];
        if (znam != '*'&& znam != '/') return levy;
        pos++;
        pravy = cti_term ();
        levy = vyhodnot (levy, znam, pravy);
    }
}

char *cti_term (void) {
    char *hodnota, token = tokeny[pos++];
    if (token == '(') {
        hodnota = cti_vyraz ();                        /* Přeskočit zavírací závorku. */
        pos++;
    } else hodnota = val[pos - 1];
    return hodnota;
}

int main (void) {                                      /* A teď to celé spojíme dohromady */
    char *hodnota;

    fgets (vstup, MAX_DELKA, stdin);
    lex ();
    hodnota = cti_vyraz ();
    printf ("assign _result %s\n", hodnota);
    return 0;
}
```

18-2-1 Potrhlý syslík — David Matoušek

Většina z vás si s tímto úkolem hravě poradila. Je však nutné dodat, že samotný postup není dostačující řešení. Vždy je třeba říci, proč zvolený postup funguje. Ale i to se většině řešitelů povedlo, a tak se v rychlosti koukněme, jak by mohlo vypadat vzorové řešení.

Máme 50 mincí, 18 je otočeno lícem, zbylé jsou otočeny rubem. Oddělme libovolných 18 mincí a označme je jako první hromádku. Ostatní mince tvoří hromádku druhou. Všechny mince v první hromádce otočíme a nahlédneme, že tím je úkol splněn. Z původních 18 mincí otočených lícem se do první hromádky dostalo právě k z nich. A tedy $18 - k$ je počet rubem otočených mincí v první hromádce. Po otočení všech mincí v této hromádce se počty rubem a lícem otočených prohodí. V první hromádce tak bude $18 - k$ lícem otočených mincí. Ve druhé hromádce je ale již od začátku $18 - k$ lícem otočených mincí, protože celkem jich bylo 18 a k z nich připadlo do první hromádky. Počty v obou jsou tak shodné a úloha je vyřešena.

Jak mnozí z vás také poznali, princip řešení není dán tím, že mincí bylo právě 50 a 18 bylo lícem otočených a že řešení se dá spolu se zadáním úlohy zobecnit a stále bude fungovat. Dodejme ještě, že k takovýmto úlohám není třeba psát zdrojový kód. V některých došlých řešeních se zbytečně otáčely mince tam a zase zpět, za což byl strháván jeden bodík, protože taková řešení jsou vlastně pomalejší.

18-2-2 Kvakulátor — Tomáš Gavenčiak

Vzpomenu si na ZŠ a písemné dělení "pod sebou". Tam vše záleželo na zbytcích – vyšel-li nám někdy po vyčerpání cifer čitatele zbytek 0, nemělo číslo periodu, pokud se nějaký zbytek opakoval potom, co nám došly cifry čitatele, vznikla tak automaticky perioda. Ještě předtím ale zajistím, aby $a < b$ tak, že provedu $a' = a \bmod b$, kde mod je zbytek po dělení. Počítáním s a' místo a se mi perioda určitě nezmění. První zbytek si tedy nastavím rovnou jako $z_1 = a \bmod b$ (zbavím se tak všech cifer a najednou a potom při dělení pod sebou "připisuji" už jen 0). Další zbytky získám postupně jako $z_{i+1} = (10 z_i) \bmod b$. Všimněte si, že cifry výsledku nás vůbec nezajímají, po nalezení opakování zbytků se budou opakovat i cifry, neboť cifra závisí jen na předchozím zbytku a b, které je pevné. Délka periody je nanejvýš $b - 1$, protože po více než $b - 1$ číslech $0..b-1$ potkám buď 0 nebo nějaký zbytek dvakrát.

Jednodušší cesta, jak najít periodu, je pamatovat si zbytky z_i v poli a pokaždé pole prohledat, zda už zbytek máme. Toto má časovou složitost $O(b^2)$ a paměťovou $O(b)$.

Trochu chytřejší je nepamatovat si pole z_i, ale zda a kde jsem již potkal zbytek z v poli a_z. Potom zjišťuji, zda jsem již zbytek potkal v konstantním

čase a dostanu časovou i paměťovou složitost $O(b)$.

Situaci mi komplikuje jen možná před-perioda, bez ní bych si mohl zapamatovat první zbytek a prostě si na něj počkat až ho potkám znovu. Před-perioda má ale délku nanejvýš b, stačí tedy provést prvních b dělení a z_b buď bude 0 (pak je perioda 0) nebo si ho zapamatuji a až ho potkám na pozici z_{b+p}, našel jsem periodu délky p. Časová složitost je tak pořád $O(n)$, paměťová $O(1)$.

Existuje i rychlejší řešení pracující v čase $O(\sqrt{b})$ nebo i lepším (hlavně podle zběsilosti použité faktorizace), ale to je příliš složité na to, abych ho zde uspokojivě popsal.

```
program kvák-ulator;
var a,b,z,i:integer;

begin
  readln(a,b);
  for i:=1 to b do a:=(10*a) mod b;
  if a=0 then writeln('perioda delky 0')
  else begin
    z:=a;
    i:=0;
    repeat
      a:=(10*a) mod b;
      inc(i);
    until a=z;
    writeln('perioda delky ',i);
  end;
end.
```

18-2-3 Jeřábník Evžen — Petr Škoda

Z počtu došlých řešení je jasné, že vás demolice bažiny zaujala, ale poradit Evženovi, jak ovládat jeřáb nejen spolehlivě, ale také rychle, byl pro vás oříšek. Pojďme ho tedy rozlousknout.

Jednoduché řešení problému je zapamatovat si úhel natočení každého segmentu a po každé změně výslednou pozici demoliční koule spočítat. Pokud víme počáteční bod i-tého segmentu a jeho absolutní úhel α v rovině, spočítáme z něj pozici $(i+1)$-ního segmentu posunutím o délku segmentu ve směru α. Takto postupným počítáním koncové pozice segmentů dojdeme až k pozici koule. Pokud máme n segmentů a dostaneme m dotazů, tento algoritmus poběží v čase $O(m \cdot n)$. Podobný algoritmus poslala většina z vás.

My se ovšem nespokojíme s jednoduchým řešením. Potřebujeme si zapamatovat některé pozice, abychom je nemuseli pokaždé počítat znovu. To uděláme celkem klasickou metodou — postavíme nad segmenty *intervalový strom*.

Předpokládejme nyní, že n je mocninou dvojky. Potom si segmenty představíme jako listy dokonale vyváženého binárního stromu. Protože počet vrcholů

se na každé další hladině zdvojnásobí, je hloubka našeho stromu $\log_2 n$, neboli $O(\log n)$. Pokud spočteme tuto geometrickou posloupnost velikosti hladin, dostaneme $2n - 1$, a tedy náš strom má celkem $O(n)$ vrcholů.

Každý vnitřní uzel stromu u bude reprezentovat skupinu segmentů, které jsou listy podstromu s kořenem u. Tuto skupinu po sobě jdoucích segmentů si můžeme představit jako jeden dlouhý segment, který začíná na počátku prvního segmentu a končí na konci posledního segmentu.

Každý segment můžeme charakterizovat jeho délkou a relativním natočením vůči předchozímu segmentu. Pokud chceme podobně popsat i složený segment, musíme ještě přidat natočení posledního segmentu ze skupiny, protože právě ten určuje, jak bude natočen další segment. Jednoduchý segment pak má v tomto popisu oba úhly stejné.

Protože bychom ale museli pro složené segmenty složitě počítat jejich délku a vstupní úhel, zapamatujeme si rovnou souřadnice bodu, kde by tento segment končil, kdyby byl umístěn do počátku a předchozí segment mířil do kladného směru osy y. Díky tomu jsou data segmentu nezávislá na ostatních segmentech a nebudeme jich muset tolik měnit při otočení jeřábu.

Jak tedy vytvoříme složený segment? Souřadnice segmentu v označme $x(v)$ a $y(v)$, výstupní úhel $\alpha(v)$. Vezměme vnitřní uzel u, který má dva syny s_1 a s_2. Pak souřadnice u spočítáme jako otočení souřadnic s_2 o úhel $\alpha(s_1)$ a přičtení k souřadnicím s_1. Výstupní úhel u je součtem výstupních úhlů jeho synů.

$$\begin{aligned} x(u) &= x(s_1) + x(s_2) \cdot \cos\alpha(s_1) + y(s_2) \cdot \sin\alpha(s_1) \\ y(u) &= y(s_1) + x(s_2) \cdot \sin\alpha(s_1) + y(s_2) \cdot \cos\alpha(s_1) \\ \alpha(u) &= \alpha(s_1) + \alpha(s_2) \end{aligned}$$

Již umíme skládat segmenty a můžeme tedy ze segmentů v listech vygenerovat složené segmenty tak, že půjdeme po hladinách stromu od nejnižší k nejvyšší a budeme je popsaným způsobem plnit. Všimněte si, že odpověď na dotaz, kde je koule jeřábu, se schovává již v kořeni stromu, protože ten je složením všech dílčích segmentů.

Složitější je otázka, zda dokážeme také tuto strukturu aktualizovat po otočení nějakého segmentu. Podívejme se, kolik složených segmentů se změní při otočení jednoho segmentu. Jsou to pouze ty, které tento segment obsahují, a ty leží na cestě od listu ke kořeni. Stačí tedy při změně segmentu projít po cestě od segmentu ke kořeni a upravit všechny složené segmenty. To uděláme stejně, jako bychom je vytvářeli znova.

Jak dlouho nám to bude trvat? Již jsme si ukázali, že náš strom má hloubku $O(\log n)$ a tedy celková časová složitost je $O(m \cdot \log n)$. Paměťová složitost je $O(n)$, protože strom má také lineární velikost.

Ještě se podíváme na implementaci našeho algoritmu. Pokud vytváříme vyvážený strom, do kterého nechceme přidávat nebo z něj odebírat prvky, často

se na to hodí použít implementaci haldy v poli. Ta má kořen na pozici s indexem 1 a synové vrcholu s indexem i jsou (v případě binárního stromu) na pozici $2i$ a $2i+1$. Tím se často zjednoduší veškeré cestování po stromě.

Jako třešničkou na dortu se můžete inspirovat Martinovým (MJ) vzorovým řešením.

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

#define MAXN 16384                          /* musí být mocninou dvojky */
#define MAX (2*MAXN+1)

static struct seg {
    double x, y;
    int a;
} seg[MAX];

static void merge (int i) {
    struct seg *l = seg+2*i, *r = l+1;
    double a = 2*M_PI*l->a/360;
    seg[i].x = l->x + r->x*cos (a) - r->y*sin (a);
    seg[i].y = l->y + r->x*sin (a) + r->y*cos (a);
    seg[i].a = (l->a + r->a) % 360;
}

int main (void) {
    int i, j, k, n0, N;

    scanf ("%d", &n0);
    for (N=1; N<n0; N*=2);
    for (i=0; i<n0; i++) scanf ("%lf", &seg[N+i].y);
    for (i=N-1; i>=1; i--) merge (i);
    while (scanf ("%d%d", &j, &k) == 2) {
      j = N+j-1;
      seg[j].a = (180+k) % 360;
      while (j != 1) merge (j /= 2);
      printf ("%.2f %.2f\n", seg[1].x, seg[1].y);
    }

    return 0;
}
```

18-2-4 Stavbyvedoucí — Tomáš Gavenčiak & Martin Mareš

Ah, bezdůvodně čekáš, čtenáři drahý, ďábelské fígle, i když na obyčejný počátek prachobyčejného řešení stavbyvedoucího strastí tato věta vyznívá značně zvláštně. Demonstruje totiž jeden velice důležitý fakt: setřídit slova podle třetího písmenka, pak podle druhého (stabilně, tj. se zachováním původního pořadí, pokud jsou druhá písmenka nějakých dvou slov stejná) a nakonec podle prvního, dopadne stejně jako nejdříve je setřídit podle prvního, pak zvlášť každou skupinu začínající stejným písmenem setřídit podle druhého a skupinky mající stejné i druhé písmeno ještě podle třetího. Totéž samozřejmě platí i

pro třídění tabulek podle sloupečků podle Potrhlíkových požadavků: ponejprv řádky třídíme podle sloupečku daného posledním požadavkem, skupiny se stejnou hodnotou tohoto sloupečku pak podle předchozího požadavku a tak dále, až se propracujeme k začátku seznamu požadavků nebo skončíme se skupinkami o jednom řádku. Z toho ovšem ihned plyne, že zabývat se tímtéž sloupečkem vícekrát je zhola zbytečné: pokud po prvním porovnání podle nějakého sloupečku zůstaly nějaké dva řádky v téže skupině, měly v příslušném sloupečku stejnou hodnotu, a proto nám je další porovnání musí ponechat ve stejném pořadí.

Pokud se tedy nějaký sloupeček v posloupnosti požadavků vyskytuje vícekrát, stačí ponechat jen jeho poslední výskyt. Tím určitě dostaneme posloupnost ekvivalentní se zadanou (takové budeme říkat *řešení*). Zbývá nám ještě dokázat, že žádné kratší řešení nemůže existovat. Kdyby existovalo, vezmeme si nejkratší takové. Určitě se v něm nebudou opakovat sloupečky (jinak by se naším algoritmem dalo ještě zkrátit) a ani v něm nebude žádný sloupeček navíc (to by byl sloupeček, podle kterého se netřídilo ani v zadané posloupnosti, takže bychom ho mohli škrtnout). Tudíž v ní musí nějaký sloupeček z našeho řešení chybět. Pak stačí vytvořit dva řádky, které se budou lišit pouze v chybějícím sloupečku, a takové musí obě řešení setřídit různě, což je evidentní podvod, totiž spor. Podobně můžeme dokázat i to, že naše řešení je nejen nejkratší, ale také jediné s touto délkou: jiné by se nutně lišilo pořadím nějakých dvou sloupečků i, j a mohli bychom sestrojit dva řádky podle i uspořádané opačně než podle j a jinak stejné a opět dojít ke sporu.

Zbývá si rozmyslet, jak naše řešení naprogramovat. Znalci Unixového shellu mohou navrhnout třeba toto:

```
nl -s:|tac|sort -t: -suk2|sort -n|cut -d: -f2
```

My si předvedeme jednoduchý (a přiznejme, že daleko efektivnější) program v Pascalu. Bude číst vstupní posloupnost po jednotlivých prvcích a ve frontě si udržovat řešení pro zatím přečtenou část vstupu. Přijde-li požadavek na třídění podle nějakého sloupečku, přidáme tento sloupeček na konec fronty a pokud se již ve frontě vyskytoval, předchozí výskyt odstraníme. Abychom to zvládli rychle, budeme si frontu pamatovat jako obousměrný spojový seznam, tj. pro každý sloupeček si uložíme jeho předchůdce a následníka. Tak nám celá fronta zabere paměť lineární s počtem sloupečků a na každou operaci si vystačíme s konstantním časem, celkově tedy s časem $O(M+N)$ (požadavků+sloupečků).

Ještě přidáme malý trik pro zkrácení programu: Abychom si nemuseli dávat pozor na případy, kdy je fronta prázdná, a udržovat si ukazatel na konec fronty, přidáme do fronty ještě sloupeček 0, který bude stále na začátku i na konci (fronta tedy bude zacyklená) a který pak nevypíšeme.

```
const MaxN = 1000;                                  { maximální počet sloupečků }
var M, N : integer;                                { počet požadavků a sloupečků }
```

```
    pred, next : array [0..MaxN] of integer; {před. a násl. sloupečků ve frontě}
    p, s : integer;                 { právě zpracovávaný požadavek a sloupeček }
begin
   read(N, M);
   pred[0] := 0; next[0] := 0;      { 0 je sama sobě předchůdcem i následníkem }
   for s := 1 to N do pred[s] := -1;   { ostatní sloupečky jsme ještě neviděli }
   for p := 1 to M do begin
      read(s);                                                { další požadavek }
      if pred[s] >= 0 then begin { pokud už jsme ho viděli, pryč s ním z fronty}
         next[pred[s]] := next[s];
         pred[next[s]] := pred[s];
      end;
      next[s] := 0; pred[s] := pred[0];  { a každopádně přidat na konec fronty }
      next[pred[0]] := s; pred[0] := s;
   end;
   s := next[0];                                { vypíšeme koncový stav fronty }
   while s <> 0 do begin
      writeln(s);
      s := next[s];
   end;
end.
```

18-2-5 Krokoběh — Pavel Čížek

Krokodýli šli dneska spát nalačno prakticky u všech došlých řešení a překvapivě u většiny došlých řešení byl spokojen i Potrhlík, který stavbu přežil bez bankrotu. No a nyní, jak se úloha měla řešit. Většina řešitelů (možná pod vlivem kuchařky) používala terminologii teorie grafů, proto jí i v tomto řešení použijeme.

O co tedy šlo. Zjistit, kolik nejméně hran je třeba přidat do grafu, aby se stal 2-souvislý. Hned na začátku si všimneme, že pokud najdeme v grafu komponentu, která je 2-souvislá, tak jí můžeme zkontrahovat (scvrknout) do jednoho vrcholu, aniž by se změnil počet potřebných hran. Takhle můžeme pokračovat tak dlouho, dokud se v grafu budou vyskytovat kružnice. Snadno se dá nahlédnout, že hrany tohoto zkontrahovaného grafu budou mosty v původním grafu (most není součástí žádné kružnice, proto nebude v žádném kroku zkontrahován, na druhou stranu pokud hrana není most, pak je součástí nějaké kružnice a proto bude dříve či později zkontrahována). Je také vidět, že takto zkontrahovaný graf bude les, jelikož neobsahuje kružnice. Dále budeme uvažovat tento les.

Nyní mohou nastat 2 situace. První, kterou dost řešitelů zapomnělo ve svých řešení ošetřit, je ta, že graf byl na počátku 2-souvislý, tj. že se zkontrahoval do bodu. Pak není třeba nic přidávat.

Druhá je zbytek. Kolik bude třeba hran dodat? Jelikož v 2-souvislém grafu má každý vrchol stupeň (tj. kolik hran do něj vede) alespoň 2 a hrana spojuje právě 2 vrcholy, musíme přidat alespoň $A + \lceil B/2 \rceil$ $(*)$ hran, kde A je počet

vrcholů stupně 0, B počet vrcholů stupně 1 (tj. listů) a zaokrouhluje se nahoru, jelikož v případě lichého B musíme tento lichý list také zapojit do nějaké kružnice, tedy tento lichý list zapojíme na libovolný vrchol.

Nyní, indukcí podle počtu vrcholů dokážeme, že tolik i stačí. Pro 2 vrcholy může les vypadat buď jako 2 vrcholy a pak je třeba přidat ještě 2 hrany (což splňuje vzorec $(*)$), nebo jsou tyto 2 vrcholy spojené hranou, a pak stačí přidat jednu (opět v souladu s $(*)$). Všimněme si také, že jsme v obou případech alespoň jednu hranu přidali.

Platí, že libovolný les jde vyrobit z jednoho vrcholu pomocí operací:

1) přidej vrchol (a s ničím ho nespojuj)
2) přidej vrchol a spoj ho s nějakým vrcholem, který už v lese je.

Nyní uvažujme, že pro N vrcholů máme již 2-souvislý les pomocí

a) $A + B/2$ hran (pro sudé B)
b) $A + (B-1)/2$ hran (pro liché B; 1 cesta nezesouvislena)

Přidejme vrchol X pomocí pravidla:

1) Vezmeme nějakou přidanou hranu (vedoucí $I \leftrightarrow J$), tu odstraníme a přidáme místo ní hrany $I \leftrightarrow X$ a $X \leftrightarrow J$. Tím nám stoupl počet přidaných hran do grafu o 1. Také A se zvětšilo o jedna, takže a), resp. b) stále platí.
2) Při připojování X mohou nastat tři situace:
   - α) Připojujeme ho hranou pod vrchol Y stupně 0. Pak ale od tohoto vrcholu vedou 2 přidané hrany. Vezmeme libovolnou z nich (nechť vede z I) a zrušíme ji. Místo ní zavedeme novou hranu $I \leftrightarrow X$. Touto operací se nám snížil počet vrcholů stupně 0 v grafu o 1, nicméně z X i Y se staly listy a proto je B o 2 vetší. Tedy a) příp. b) je stále splněno.
   - β) Připojujeme ho hranou za list L. Pokud je B liché a list L je konec naší volné cesty, není třeba nic dělat a indukční předpoklady máme splněny. Jinak do tohoto listu vede nějaké přidaná hrana (z nějakého vrcholu I). Pak ale stačí zrušit hranu $I \leftrightarrow L$ a zavést novou hranu $I \leftrightarrow X$. Tím zůstane počet přidaných hran zachován. L přestal být po tomto kroku listem, nicméně objevil se nový list X, tudíž A i B zůstalo a tedy a), resp. b) stále platí.
   - γ) Připojujeme-li ho hranou za vrchol stupně alespoň 2, B se nám zvýší o 1, A zůstane stejné. Pokud B bylo sudé, není třeba nic dělat. Po tomto přidání bude B liché a vrchol X bude konec nezesouvislněné cesty. Vzorec b) bude zřejmě platit. Nyní pokud je B liché, označíme si list na konci cesty Y. Pokud vrchol

X napojujeme za vrchol, který nebyl součástí cesty, pak stačí přidat hranu $X \leftrightarrow Y$. Pokud napojujeme X na cestu, pak vezmeme libovolnou přidanou hranu $I \leftrightarrow J$, tu z grafu odstraníme a přidáme 2 nové $I \rightarrow X$ a $J \rightarrow Y$. V obou případech stoupne počet přidaných hran v do lesa o 1, což je v souladu s a).

A je to. Pro sudé B jsme dostali rovnou 2-souvislý graf, pro liché musíme ještě konec cesty napojit na libovolný vrchol, který do téhle cesty nepatří, abychom dostali 2-souvislý graf. Tím se ale dostaneme na $A + (B-1)/2 + 1 = A + \lceil B/2 \rceil$ hran.

Program je implementací výše uvedeného. Pomocí algoritmu popsaného v kuchařce 2. série najde v zadaném grafu mosty a pak v každé komponentě 2-souvislosti spočítá, kolik mostů z ní vede. Nakonec spočte hrany, které je třeba přidat, pomocí (∗). Časová i paměťová náročnost programu je $O(M+N)$ (při každém průchodu do hloubky se algoritmus zřejmě na každou hranu podívá dvakrát).

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

int Vrcholu, Hran;

struct Hrana {
    int VedeDo;                          /* číslo vrcholu, kam tahle hrana vede */
    int Most;                            /* 1, je-li to most, jinak 0 */
    struct Hrana *DalsiHrana;            /* další hrana vedoucí z daného vrcholu */
    struct Hrana *OpacnaHrana;           /* hrana opačného směru */
};

struct Hrana **Hrany;                    /* pole seznamů hran z daného vrcholu */
int *Navstiveno;                         /* pro průchod do hloubky */

int HledejMosty (int Vrchol, int MinulyVrchol, int Hloubka) {
    /* průchod do hloubky z daného vrcholu, který v dané komp. souv. hledá mosty */
    struct Hrana *hrana;
    int minHloubka;
    int HloubkaDaneVetve;

    Navstiveno[Vrchol] = Hloubka;
    minHloubka = Hloubka;
    for (hrana = Hrany[Vrchol]; hrana != NULL; hrana = hrana->DalsiHrana)
    if (hrana->VedeDo != MinulyVrchol) {
      if (Navstiveno[hrana->VedeDo] == 0) {   /* nový vrchol, rekurze */
        HloubkaDaneVetve = HledejMosty (hrana->VedeDo, Vrchol, Hloubka+1);
        if (HloubkaDaneVetve > Hloubka) {
          hrana->Most = 1;
          hrana->OpacnaHrana->Most = 1;
        }
      } else HloubkaDaneVetve = Navstiveno[hrana->VedeDo];
      if (HloubkaDaneVetve < minHloubka) minHloubka = HloubkaDaneVetve;
    }
    return minHloubka; };
```

```
int SpocitejMosty (int Vrchol) {
    struct Hrana *hrana;
    int Mostu;

    Navstiveno[Vrchol] = 0;
    Mostu = 0;
    for (hrana = Hrany[Vrchol]; hrana != NULL; hrana = hrana->DalsiHrana)
      if (hrana->Most) Mostu++;
      else if (Navstiveno[hrana->VedeDo] != 0)
        Mostu+= SpocitejMosty (hrana->VedeDo);

    return Mostu;
};
int main () {
    int i, Z, Do;
    int Listu, Stupen0;
    int VidiMostu;
    struct Hrana *NovaHrana1, *NovaHrana2;

    scanf ("%d %d", &Vrcholu, &Hran);
    Hrany = malloc (Vrcholu * sizeof (struct Hrana *));
    for (i = 0; i < Vrcholu; i++) Hrany[i] = NULL;
    for (i = 0; i < Hran; i++) {
      scanf ("%d %d", &Z, &Do);
      Z--; Do--;                          /* indexujeme 0..Vrcholu-1 */
      NovaHrana1 = malloc (sizeof (struct Hrana));
      NovaHrana2 = malloc (sizeof (struct Hrana));
      NovaHrana1->OpacnaHrana = NovaHrana2;
      NovaHrana2->OpacnaHrana = NovaHrana1;
      NovaHrana1->VedeDo = Do;
      NovaHrana1->Most = 0;
      NovaHrana1->DalsiHrana = Hrany[Z];
      Hrany[Z] = NovaHrana1;
      NovaHrana2->VedeDo = Z;
      NovaHrana2->Most = 0;
      NovaHrana2->DalsiHrana = Hrany[Do];
      Hrany[Do] = NovaHrana2;
    };

    Navstiveno = malloc (sizeof (int)*Vrcholu);
    for (i = 0; i < Vrcholu; i++) Navstiveno[i] = 0;
    for (i = 0; i < Vrcholu; i++) if (Navstiveno[i] == 0) HledejMosty (i, -1, 1);
    Stupen0 = 0; Listu = 0;
    for (i = 0; i < Vrcholu; i++) if (Navstiveno[i] != 0) {
      VidiMostu = SpocitejMosty (i);
      if (VidiMostu == 0) Stupen0++;
      if (VidiMostu == 1) Listu++;
    };
    printf ("%d\n", ( (Stupen0 == 1) && (Listu == 0)) ? 0 : (Stupen0 + (Listu+1)/2));

    return 0;
};
```

18-2-6 Dominující komplikátory — Zdeněk Dvořák

Někteří řešitelé si prostě uložili matici relace dominance (tedy pole indexované čísly bloků, v níž na pozici $[A, B]$ je 1 pokud basic blok A dominuje basic blok B, a 0 jinak). Toto řešení je nepraktické – na dotaz, zda jeden blok dominuje druhý, jsme sice schopni odpovědět v konstantním čase, ale paměťová složitost je $O(N^2)$, kde N je počet bloků v programu, což často bude příliš mnoho. Navíc se s touto reprezentací špatně pracuje, pokud optimalizace změní CFG, často je jediná možnost celou matici přepočítat. Dominance je ovšem velmi speciální relace, kterou lze reprezentovat mnohem efektivněji.

Začneme tím, že si zavedeme následující značení: budeme psát $A \leq B$, pokud basic blok A dominuje basic blok B. Samozřejmě se tím snažíme naznačit, že relace dominance by se mohla chovat jako uspořádání. Je asi vhodné si to zdůvodnit:

- Pro každý blok A platí, že $A \leq A$, tedy že dominuje sám sebe.
- Jestliže $A \leq B$ a zároveň $B \leq A$, pak $A = B$: pokud by A a B byly různé bloky, pak si vezměme cestu $sv_1v_2 \ldots v_kA$ z počátku do A, která neprochází A (tj. $v_i \neq A$ pro všechna $1 \leq i \leq k$). Protože $B \leq A$, někde na této cestě musí být blok B, tedy $v_t = B$ pro nějaké t. Pak ale $sv_1v_2 \ldots v_t$ je cesta z počátku do B, která neobsahuje A, což je ve sporu s tím, že $A \leq B$.
- Jestliže $A \leq B$ a zároveň $B \leq C$, pak $A \leq C$: pokud každá cesta z počátku do C obsahuje B a každá cesta z počátku do B obsahuje A, pak každá cesta z počátku do C také musí obsahovat A.

Dominance tedy opravdu je uspořádání, ale nemusí to být uspořádání úplné – většinou existují bloky, které jsou v ní neporovnatelné, tj. $A \not\leq B$ a $B \not\leq A$. Tím jsme si tedy příliš nepomohli, protože obecné uspořádání v lepším než kvadratickém prostoru reprezentovat nelze. Povšimneme-li si ovšem ještě následující vlastnosti, situace se značně zjednoduší: Pokud se omezíme na bloky, které dominují nějaký pevně zvolený blok C, pak je uspořádání dominancí úplné, tedy pokud $A \leq C$ a $B \leq C$, pak buď $A \leq B$ nebo $B \leq A$. Dokažme si toto tvrzení. Předpokládejme, že $A \leq C$, $B \leq C$ a A a B jsou přitom neporovnatelné. Zvolme si libovolnou cestu $p = sv_1 \ldots v_kC$ z počátku do C. Na p se musí vyskytovat jak A, tak B. Nechť bez újmy na obecnosti p projde jako poslední A, tj. existuje t tak, že $v_t = A$ a $v_i \neq B$ pro $i > t$. Protože $B \not\leq A$, existuje cesta q z počátku do A, která neprochází přes B. Pak ale cesta q, za niž připojíme $v_{t+1} \ldots v_kC$, jde z počátku do C, aniž by prošla přes B, což je spor s tím, že $B \leq C$.

Pro každý blok C kromě počátku tedy existuje „nejpozdější“ blok, který ho dominuje (budeme mu říkat *přímý dominátor* bloku C a značit ho $d(C)$), tedy takový, že pokud $A \leq C$ a $A \neq C$, pak $A \leq d(C)$. Zřejmě $A \leq C$ právě

tehdy, pokud $A = d(d(\ldots(d(C))\ldots))$. Jestliže si nakreslíme orientovaný graf, jehož hrany jsou dvojice $(C, d(C))$ pro všechny bloky C, dostaneme strom, orientovaný směrem ke kořeni, jímž je počátek. Platí, že $A \leq B$, pokud vrchol B patří do podstromu, jehož kořen je A. Další možnost, jak si relaci dominance reprezentovat, je tedy uložit si tento strom a pak při dotazu procházet buď podstrom A, nebo následníky vrcholu B. Paměťová složitost je $O(N)$ – stačí mít uložen tento strom. Oba postupy mají ovšem v nejhorším případě lineární časovou složitost.

Tento strom si proto ještě dále zpracujeme. Provedeme jeho prohledání do hloubky a budeme si počítat čísla kroků (tedy budeme mít čítač, který si zvýšíme o 1 pokaždé, když vstoupíme do vrcholu nebo se z něj vracíme). Pro každý vrchol C si zapamatujeme čísla $i(C)$ a $o(C)$ – čísla kroků, kdy jsme vstoupili do C a kdy jsme se z něj vrátili. Zřejmě $A \leq B$ právě tehdy, když do B vstoupíme až po A, ale vrátíme se z něj předtím, než se vrátíme z A, tedy pokud $i(A) \leq i(B)$ a zároveň $o(B) \leq o(A)$. Paměťová složitost zůstane lineární, neboť pro každý vrchol si potřebujeme pamatovat pouze dvě čísla. Složitost dotazu bude konstantní, stačí provést dvě porovnání.

Na závěr poznamenejme, že většina algoritmů pro určení dominance přímo konstruuje strom přímých dominátorů, takže nikdy není nutné si pamatovat celou matici dominance. Zajímavá otázka je, jak popsanou datovou strukturu opravit, pokud některá optimalizace změní CFG. Strom přímých dominátorů se změní snadno, nicméně popsané očíslování se nám tím pokazí, proto je v praxi nutné použít složitější datové struktury.

18-3-1 Trávník — Jana Kravalová

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 $O(K)$ 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 $O(N \cdot M)$, 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 $O(N \cdot M)$. Paměťová je tentokrát skutečně konstantní, protože nás nikdo nenutí pamatovat si celou matici, ale jenom její součet.

```
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.
```

18-3-2 Duel — Martin „Bobřík“ Kruliš

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 remizuje. 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 osm 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 tři 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*.

18-3-3 Vrah — Tomáš Gavenčiak

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 $O(KN)$.

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 $O(K + N)$.

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.

```
const maxN=1000
      maxK=1000;

var K,N:integer;                                {zadání}
    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);                                      {načtu oběti}
    obeti[i]:=a; byl[i]:=false; poc_hran[i]:=0; {...a zároveň inicializuji}
  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.
```

18-3-4 Pochoutka pro prasátko — Jana Kravalová

Máme šachovnici o rozměrech $X \times Y$ a sadu K pravidel, podle nichž se prasátko umí pohybovat s určitou námahou. Krátké pozorování odhalí, že každé políčko šachovnice je jeden vrchol grafu a že mezi vrcholy vede hrana právě tehdy, pokud existuje pravidlo převádějící prasátko z jednoho vrcholu na druhý. Pak je hrana samozřejmě ohodnocena příslušným množstvím námahy. A jelikož jsou hrany kladně ohodnocené a my hledáme nejkratší cestu ze startovní pozice hladovějícího pašíka na naleziště Velké Bukvice, máme úlohu jako dělanou (ve skutečnosti opravdu dělanou) pro použití kuchařkového Dijkstrova algoritmu s haldou.

Ukázalo se ale, že naprogramovat takový algoritmus nemusí být až tak jednoduché. Někteří těžce válčili s haldou, jiní v boji podlehli a zaslali jen slovní popis algoritmu.

První otázka je, jak si vyrobit onen graf zobrazující prostor lesa. Odpověď je jednoduchá. Žádný graf není třeba vyrábět, budeme pracovat přímo nad políčky lesa a hledané hrany si budeme konstruovat přímo v okamžiku, kdybychom se v Dijkstrově algoritmu dívali na sousedy aktuálně zkoumaného vrcholu. Postupně použijeme všechna možná pravidla pro pohyb z daného políčka a podíváme se, jestli jsme se nedostali mimo les.

Druhý, horší problém, vzniká u haldy. V okamžiku, kdy v Dijkstrově algoritmu najdeme lepší cestu a přepočítáváme vzdálenost nějakému vrcholu, mění se samozřejmě jeho pozice v haldě vrcholů a haldu musíme přeskládat. Jak na to?

Můžeme si někde bokem pamatovat, kde přesně se každý vrchol v haldě nachází, a pustit na něj bublání. Pak ale musíme při jakékoli operaci s haldou každému vrcholu přepočítávat tento jeho index v haldě a to je trošku zmatek.

Jiné, jednodušší řešení je haldu nijak nepředělávat, a když nějakému vrcholu přepočítáme vzdálenost, prostě jej do haldy strčit znovu. Tak se nám některé vrcholy mohou v haldě opakovat, ale my dokážeme v Dijkstrově algoritmu při vytahování minimálního prvku z haldy snadno rozeznat, jestli jej máme zpracovávat, nebo jestli je to jen zopakovaný prvek. Poznáme to podle toho, jestli už má trvalou hodnotu.

Za jednodušší řešení ale zaplatíme. Zatímco v těžším, „přepočítávacím“ řešení se každý prvek dostane do haldy nejvýš jednou, takže halda může zabírat jen tolik místa, jaký je počet vrcholů grafu, u druhého řešení se prvky mohou dostat do haldy víckrát, konkrétně halda může být veliká jako počet hran grafu.

Dijkstrův algoritmus z kuchařky trvá $O((N + M) \cdot \log N)$, kde N je počet vrcholů, u nás $X \times Y$, a M počet hran, u nás XYK. Za každou operaci s haldou násobíme logaritmem velikosti haldy. Pokud tedy použijeme haldu s přepočítáváním, dostaneme časovou složitost $O(XYK \cdot \log(XY))$. Jednoduš-

ší halda dá časovou složitost $O((N+M)\cdot\log M) \le O((N+M)\cdot\log N^2) = O((N+M)\cdot 2\log N) = O((N+M)\cdot\log N)$, takže vlastně tutéž.

Paměťová složitost je u haldy s přepočítáváním $O(XY)$, protože si potřebujeme pamatovat jen les a haldu na vrcholy, ale u větší haldy až $O(XYK)$.

Jak si všiml Pepa Pihera, náš algoritmus jde ještě vylepšit. Malou úpravou dosáhneme toho, že v haldě bude vždy nejvýš K prvků, čímž stlačíme složitost na $O(XYK\cdot\log K)$. (Platí $K \le XY$, protože pokud by pravidel bylo více, na některé políčko by se dalo dostat pomocí více pravidel a my si můžeme nechat jenom to lepší z nich.)

V jednom kroku se Dijkstrův algoritmus pokouší najít vrchol s nejmenším dočasným ohodnocením. Jinak řečeno, hledá takový nezpracovaný vrchol spojený s už zpracovaným vrcholem, že součet ohodnocení zpracovaného vrcholu a hrany z něj vedoucí je co nejmenší. Navíc vrcholy zpracováváme (trvale ohodnocujeme) podle jejich vzdálenosti od výchozího místa, tedy v neklesajícím pořadí.

Zvolme si pro tento odstavec jediné pravidlo. Kromě krajních případů ho můžeme použít z každého vrcholu. Sledujme vrcholy, které pomocí tohoto pravidla dostanou trvalé ohodnocení. V průběhu algoritmu je ohodnocení těchto zpracovávaných vrcholů neklesající. Protože jsme ho získali přičtením hodnoty pravidla k ohodnocení výchozímu vrcholu, je i ohodnocení vrcholů, *ze* kterých toto pravidlo používáme, neklesající. Toto jediné pravidlo tedy používáme na vrcholy v tom pořadí, v jakém je trvale ohodnocujeme.

Když víme, že každé pravidlo používáme na vrcholy v tom pořadí, v jakém je trvale ohodnocujeme, zapamatujeme si u každého pravidla, ze kterého vrcholu jsme ho naposledy použili. Když potom hledáme vrchol s nejnižším dočasným ohodnocením, každé pravidlo už má určený vrchol, ze kterého ho použijeme. Vybereme si tedy tu nejlepší kombinaci vrchol-pravidlo, tím jsme našli další vrchol s trvalým ohodnocením, a použité pravidlo „posuneme“ k dalšímu vrcholu. Tím myslíme, že příště ho budeme používat z vrcholu, který jsme v Dijkstrovi trvale ohodnotili hned po tom vrcholu, ze kterého jsme teď pravidlo používali.

V každém kroku tedy potřebujeme najít minimum z K hodnot, toto minimum odstranit a přidat místo něj jinou hodnotu. K tomu je halda jako stvořená, všechny tyto operace zvládne v čase $O(\log K)$. Navíc každou hranu zpracujeme právě jednou, čímž se dostáváme na slibovanou složitost $O(XYK\log K)$.[M.S.]

```
#include <stdio.h>
#define MaxN 100
#define MaxM 100
#define MaxK 50

typedef struct policko_typ {
    int x, y, vzdal, je_docas, predx, predy;        /* souřad., vzdál., dočasnost, předchůdce */
} policko;
```

```
typedef struct pravidlo_typ {                    /* možné pohyby čuněte */
    int x, y, namaha;                            /* posun o x a y, vynaložená námaha */
} pravidlo;
typedef struct uzel_typ {                        /* uzel v haldě */
    int vzdal, x, y;                             /* dočasná vzdálenost, souřadnice */
}uzel;

int N, M, K;                                     /* rozměry lesa, počet pravidel */
pravidlo pravidla[MaxK];                         /* všechny pohyby čuněte */
uzel halda[MaxN*MaxM*MaxK];                      /* halda na políčka  */
int halda_len;                                   /* velikost haldy */
policko les[MaxN][MaxM];                         /* dvojrozměrný prostor = les */

void halda_vloz (int x,  int y,  int vzdal) {
    int i;
    uzel pom;

    i = halda_len++;
    halda[i].x = x; halda[i].y = y; halda[i].vzdal = vzdal;
    while (i > 1 && halda[i/2].vzdal > halda[i].vzdal) {
        pom = halda[i/2];
        halda[i/2] = halda[i];
        halda[i] = pom;
        i /= 2;
    }
}
void halda_odeber_min (void) {
    int i,  j;
    uzel pom;
    halda[1] = halda[halda_len--];
    i = 1;
    while (2*i<=halda_len) {
        j = i;
        if (halda[j].vzdal > halda[2*i].vzdal) j = 2*i;
        if (2*i+1 <= halda_len && halda[j].vzdal > halda[2*i+1].vzdal) j = 2*i+1;
        if (i == j) break;
        pom = halda[i]; halda[i] = halda[j]; halda[j] = pom;
        i = j;
    }
}
void dijkstra (int start_x,  int start_y,  int cil_x,  int cil_y) {
    int vx,  vy,  nx,  ny,  i,  j;

    for (i = 0; i < N; i++) {                    /* inicializace */
        for (j = 0; j < M; j++) {
            les[i][j].je_docas = 1;
            les[i][j].vzdal = -1;
            les[i][j].predx = les[i][j].predy = -1;
        }
    }
    les[start_x][start_y].vzdal = 0;             /* vzdálenost start-start je 0 */
```

```
    halda_vloz (start_x, start_y, 0);                    /* start na haldu */

    while (halda_len > 0) {                              /* dokud není halda prázdná */
      vx = halda[0].x;                                   /* vybereme políčko s nejmenší */
      vy = halda[0].y;                                   /* dočasnou vzdáleností */
      halda_odeber_min ();                               /* odebereme políčko z haldy */
      if (vx == cil_x && vy == cil_y) break;    /* bukvice nalezena, konec */
      if (!les[vx][vy].je_docas) {
        printf ("Tady uz jsme jednou byli.\n");
        continue;                                        /* tady uz jsme byli, jdeme dál */
      }
      les[vx][vy].je_docas = 0;                          /* políčko prohlásíme za trvalé */

      for (i = 0; i < K; i++) {
        nx = pravidla[i].x + vx;                         /* kam se umíme dostat s pravidlem i */
        ny = pravidla[i].y + vy;
        printf ("Zkousim nove policko [%d, %d]\n", nx, ny);
        if (nx < N && ny < M && nx >=0 && ny >= 0)   /* nevyběhli jsme z lesa */
          if ( (les[nx][ny].vzdal == -1) ||
               (les[nx][ny].vzdal > les[vx][vy].vzdal + pravidla[i].namaha)) {
            les[nx][ny].vzdal = les[vx][vy].vzdal + pravidla[i].namaha;
                                                         /* nová vzdálenost */
            les[nx][ny].predx = vx;                      /* nový předek */
            les[nx][ny].predy = vy;
            halda_vloz (nx, ny, les[nx][ny].vzdal);    /* vlož na haldu */
          }
      }
    }
}
int main () {
    int prase_x, prase_y, bukv_x, bukv_y, nx, ny, i;

    printf ("Rozměry lesa:␣"); scanf ("%d %d", &N, &M);
    printf ("Pozice prasete:␣"); scanf ("%d %d", &prase_x, &prase_y);
    printf ("Pozice bukvice:␣"); scanf ("%d %d", &buku_x, &buku_y);
    printf ("Počet pravidel pohybu prasete:␣"); scanf ("%d", &K);
    for (i = 0; i < K; i++)
      scanf ("%d %d %d", &pravidla[i].x, &pravidla[i].y, &pravidla[i].namaha);

    dijkstra (prase_x, prase_y, bukv_x, bukv_y);
    if (les[buku_x][buku_y].vzdal == -1) printf ("Cesta neexistuje.\n");
    else {
      printf ("Nejkratší cesta pozpátku: \n");
      nx = buku_x; ny = buku_y;
      while (nx != -1 && ny != -1) {
        printf ("[%d, %d]\n", nx, ny);
        nx = les[nx][ny].predx; ny = les[nx][ny].predy;
      }
    }
    return 0;
}
```

18-3-5 Hroší lov — Martin Mareš

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 $O(MN \cdot (p_0 + p_1 + \ldots + p_H)) = O(MNP)$.

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 $O(MNp_0) = O(MNP)$, dostaneme do počáteční polohy, a tím je cesta ukončena.

Celkem tedy náš algoritmus doběhne za $O(MNPT)$ kroků a k zapamatování stavů spotřebuje paměť $O(MNHT)$. [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 $T + 1 = O(T)$ 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 1 krok vpřed ze st. a do b; je-li 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, 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;
              }
          }
```

```
  for (int x=0; x<M; x++)          // smaže prasátko z˜políček s˜hrochy
    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) { // vyskytuje se někde 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;
  }
  for (int s=t; s˜> 0; s--)         // rekonstrukce cesty
    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 $O(MNHt')$, 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 $O(MNH + MNT)$ a časovou nepokazíme, je stále $O(MNPT)$.

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 $O(MNH)$, ale čas zhoršíme T-krát na $O(MNPT^2)$. 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 $O(MNH\sqrt{T})$.

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 $O(MNH\sqrt[3]{T})$ 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 $O(MNPl)$, což v součtu přes všechny úseky na jednom patře dá $O(MNPT)$, 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 $O(\log T)$ stavů zabírajících dohromady prostor $O(MNH \log T)$.

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
  if (t == s) return;                          //   sestrojení cesty
  if (t+1 == s) {
      vzad(St, cesta[s], &cesta[t]);
      return;
    }
  stav S[2];                                   // najdeme stav v~polovině
  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++;
  }
  trick(cesta, u, s, S[u%2]);           // a rekurzivně voláme pro obě části
  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.");
  }
```

18-3-6 Komplikovanější komplikátory — Zdeněk Dvořák

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 nabude 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 $O(NV)$, kde N je velikost programu (včetně jeho CFG). Paměťová složitost je také $O(NV)$, protože si pro každou proměnnou všude pamatujeme, zda je živá či mrtvá.

18-4-1 HP — Tomáš Valla

Až na drobné výjimky všechna došlá řešení fungovala. Svůj podíl na tom má to, že úloha patřila k těm nejjednodušším ze série. Někteří řešitelé si však přesto počínali poněkud neohrabaně a zbytečně složitě, za což je pochopitelně neminula záplava komentářů v jejich řešení.

Jak tedy na věc? Nejprve si z jednotlivých dvojkových cifer zadaných na vstupu vyrobíme číslo, v paměti počítače reprezentované uvnitř integeru. V prvním kroku načteme do proměnné `cislo` první cifru. V druhém kroku vynásobíme `cislo` dvojkou a přičteme druhou cifru. V třetím kroku opět vynásobíme `cislo` dvojkou a přičteme třetí cifru, a tak dále. Nyní si stačí uvědomit, že to, co na závěr zbylo v proměnné `cislo` je skutečně správně načtené číslo ze vstupu. Tomuto jednoduchému algoritmu se také říká *Hornerovo schéma.* A nepotřebovali jsme k němu žádná pomocná pole plná předpočítaných mocnin, jak se snažili tvrdit někteří řešitelé.

Nyní zbývá zjistit počet nul na konci v desítkovém zápisu. To lze udělat například tak, že budeme `cislo` neustále dělit desíti tak dlouho, dokud končí na 0 (což zjistíme pomocí operace modulo 10), a za každé vydělení si zapamatu-

jeme jedničku. Jak dlouho to celé poběží? Použijeme při tom, v tomto případě poněkud umělý, předpoklad, že všechny základní operace s integery probíhají v konstantním čase. Zjevně potřebujeme $O(N)$ kroků na načtení a zpracování N bitů ze vstupu a stejně tak $O(N)$ kroků na postupné dělení desíti, protože číslo může mít nejvýše tolik desítkových cifer kolik je jich dvojkových. Dle zadání se číslo vždy vejde do integeru, paměťová spotřeba tudíž bude konstantní. Pokud bychom však chtěli být naprosto přesní, museli bychom veškerou paměťovou složitost udávat v počtu použitých bitů, a stejně tak do času započítat dobu trvání elementárních operací (sčítání, násobení) nad několikabitovými čísly. Tak se to řeší ve formální teorii složitosti. My si tím však v této úloze nebudeme lámat hlavu.

Zdálo by se, že už můžeme skončit. Lépe než $O(N)$ zjevně algoritmus nenavrhneme, vstupní bity musíme alespoň načíst. Konečně, taková řešení jsem bral jako naprosto správná. To bychom však nebyli my, abychom nepředvedli nějakou vychytávku. Ukážeme si řešení, které by pracovalo rychleji, pokud bychom číslo již měli načtené (a samozřejmě za předpokladu jednotkové ceny za elementární aritmetické operace).

Nejprve budeme postupně mocnit na druhou číslo 10 tak dlouho, dokud ještě bude dělit `cislo`, tedy najdeme maximální 10^{2^i}, které dělí `cislo`. Správný počet nul na konci tedy bude ležet někde mezi 2^i a 2^{i+1}. K přesnému číslu se nyní dobereme upravenou metodou půlení intervalů. Zapamatujeme si 10^{2^i} do m. Zkusíme, je-li `cislo` dělitelné $m \cdot 10^{2^{i-1}}$. Pokud ano, nastavíme m na $m \cdot 10^{2^{i-1}}$. A testujeme znovu, tentokrát je-li `cislo` dělitelné $m \cdot 10^{2^{i-2}}$. To celé opakujeme, dokud i není 0.

První fáze bude trvat $O(\log N)$ kroků, neboť, jak už jsme si řekli, desítkových cifer je nejvýše tolik, kolik je dvojkových, a postupné mocnění na druhou tedy bude trvat maximálně logaritmický čas. Druhá fáze bude opět trvat logaritmicky vzhledem k počtu bitů, protože číslo i opět může nabýt maximálně hodnoty logaritmu z N. Výsledný výpis výsledku nás opět nezdrží, protože číslo udávající počet nul může mít samo maximálně $O(\log N)$ cifer. Celkem tedy čas $O(\log N)$. A pokud si budeme počínat šikovně, vystačíme si pouze s konstantní pomocnou pamětí. Konečně, pro detaily viz vzorový program.

```
program hp;
var cislo, m, pocet, s, sp: longint;
    c: char;
begin
   cislo:= 0;      {načti číslo ukončené tečkou Hornerovým schématem}
   c:= '0';
   while c<>'.' do begin
      cislo:= 2*cislo + ord(c)-ord('0');
      read(c);
   end;
```

```
   m:= 10;        {zjisti nejbližší nižší 10^{2^i} které dělí číslo}
   if cislo mod 10 = 0 then pocet:= 1
   else pocet:= 0;
   while cislo mod (m*m) = 0 do begin
      m:= m * m;
      pocet:= 2*pocet;
   end;

   {"půlením intervalů" mezi 10^(2^(i+1)) a 10^(2^i) zjisti přesný počet nul}
   s:= round(sqrt(m));
   sp:= pocet div 2;
   while sp > 0 do begin
      if cislo mod (m*s) = 0 then begin
         m:= m*s;
         pocet:= pocet+sp;
      end;
      s:= round(sqrt(s));
      sp:= sp div 2;
   end;
   writeln('počet nul je ', pocet);
end.
```

18-4-2 Elektronické hrátky — Martin „Bobřík“ Kruliš

Malý Martínek vám děkuje za došlá řešení. Moc mu pomohla. Jen byl nejprve trochu zmaten odlišným přístupem řešitelů. Někteří z vás se do problému pustili zepředu a výpočet prováděli od vstupních bitů průchodem grafu do šířky. Ostatní na to šli od lesa (tedy odzadu) a procházeli graf do hloubky od výstupních bitů. Avšak který postup je lepší? Trochu se nad oběma postupy zamysleme. Hradlová síť je zadána tak, že reference vedou směrem „zpátky“ (tedy ke vstupním bitům). Procházení od posledních (výstupních) bitů nám tedy nebude činit žádné potíže, ale opačný přístup si vyžádá předzpracování, při kterém "otočíme" reference (nebudeme si pamatovat kam jsou zapojeny vstupy jednotlivých hradel, ale zapamatujeme si, kam je zapojen výstup). Procházení odzadu je tedy výhodnější, protože nevyžaduje žádné předzpracování (byť bychom jej prováděli při načítání sítě). Procházení do hloubky od výstupu bude mít navíc tu výhodu, že spočítáme opravdu jen ta hradla, která budeme potřebovat, zatímco při procházení do šířky spočítáme úplně všechno a pak se nám může stát, že hodnoty výstupů některých hradel nebudeme vůbec potřebovat.

Náš algoritmus bude obsahovat funkci, která se zavolá na určité hradlo a spočítá jeho výsledek tak, že se rekurzivně zavolá na obě „předchozí“ hradla (to jsou hradla, na jejichž výstup jsou zapojeny moje dva vstupy) a z jejich výsledků spočítá operací NAND vlastní výsledek. Ano Ondro, vidím, že se hlásíš a chceš nám říci, že tenhle algoritmus by měl exponenciální časovou složitost. Inu ono to nebude tak docela pravda, ale každopádně nebude příliš efektivní,

protože bude počítat výstupy některých hradel vícekrát. Tento problém vyřešíme jednoduše tak, že si budeme hodnoty již spočtených hradel pamatovat. Naši rekurzivní funkci pak zavoláme pro každé hradlo, jehož výstup je zapojen na výstupní bit celé sítě.

Jakou to bude mít časovou složitost? Náš algoritmus je v podstatě speciální variantou procházení grafu do hloubky (DFS), tudíž jeho složitost je $O(M + N)$, kde N je počet vrcholů (v našem případě hradel) a M je počet hran. Naštěstí víme, že každé hradlo má právě 2 vstupy, tudíž počet hran bude roven dvojnásobku počtu vrcholů. Složitost pro jeden dotaz je tedy $O(2N + N)$, což je totéž jako $O(N)$. Graf budeme procházet pro každý dotaz úplně znovu, takže toto je časová složitost jednoho dotazu. Předzpracování si nevyžádá žádný čas a paměti spotřebujeme také pouze lineárně mnoho.

Ještě dvě poznámky ke vzorovému programu:

- Ve funkci `spocitejHradlo` a ve struktuře `THradlo` se používají celočíselné indexy. Hodnoty v rozsahu $1..N$ jsou indexy na hradla, zatím co hodnoty 0 a menší jsou odkazy na vstupní bity (tj. hodnota -2 je odkaz na 2. bit vstupu).
- Při výpočtu hodnoty hradla se používá malá optimalizace. Pokud vyjde hodnota prvního vstupu 0, pak bude výsledek vždy 1 a nemusíme počítat druhý vstup (viz. tabulka funkce NAND v zadání).

```
const MAX = 1000; { maximalní počet hradel ... to není důležité }

type THradlo = record
       in1, in2: Integer;   { indexy na vstupní hradla (nebo bity) }
       done: Boolean;       { je true, pokud bylo hradlo již spočteno }
       val: Boolean;        { spočtená hodnota (pouze pokud done = true) }
     end;

var PocetHradel, PocetVstupu, PocetVystupu: Integer;
    Hradla: array[1..MAX] of THradlo;   { pole všech hradel }
    Vystupy: array[0..MAX] of Integer;  { indexy hradel, zapojených na výstupy }
    Vstupy: array[0..MAX] of Boolean;   { hodnoty vstupních bitů }

{ rekurzivní funkce, která určí výstupní hodnotu hradla }
{ Pozn: Budu zde sahat na glob. proměnné (Hradla, Vstupy a Pocty) }
function spocitejHradlo(index: Integer): Boolean;
var val_tmp: Boolean;
begin
   if (index < 1) then begin  { index neukazuje na hradlo, ale na vstup. bit }
      spocitejHradlo := Vstupy[-index];
      Exit;
   end;

   if (not Hradla[index].done) then begin { hradlo ještě nebylo spočteno }
      Hradla[index].done = true;       { tak ho spočítáme }
```

```
      val_tmp := spocitejHradlo(in1);
      if (val_tmp) then Hradla[index].val := not (val1 and spocitejHradlo(in2))
      else     { optimalizace - nemusíme počítat dál}
         Hradla[index].val = true;  { 0 NAND s čímkoli je vždy true }
   end;
   spocitejHradlo := Hradla[index].val;   { sáhnem pro výsledek do cache }
end;

{ Připraví hradla pro další dotaz }
procedure vycistiHradla;
var i: Integer;
begin
   for i := 1 to PocetHradel do Hradla[i].done := False;
end;

{ Spočítá všechna výstupní hradla a vytiskne spočtené hodnoty }
procedure spocitejHradla;
var i: Integer;
begin
   for i := 0 to PocetVystupu-1 do
      Writeln("bit ", i, ".: ", spocitejHradlo(Vystupy[i]) );
end;

{ načte ze vstupu hodnoty do pole Vstupy - implentaci vynechávám }
procedure nactiVstup;
{ načte ze vstupu hodnoty do pole Hradla a Vystupy a konstanty Pocet... }
procedure nactiHradla;

begin
   nactiHradla;
   ...
   { při každém dotazu se zavolá posloupnost }
   vycistiHradla;
   nactiVstup;
   spocitejHradla;
end.
```

18-4-3 Běh městem — Zbyněk Falt

Tajemník by byl určitě zklamaný, neboť prakticky všechna došlá řešení byla málo Kocourkovská a fungovala správně. Bohužel ne vždy bylo z řešení zřejmé, proč by tomu tak mělo být.

Město si můžeme představit jako souvislý graf, jehož vrcholy jsou křižovatky a ulice jsou hranami. Řešením pak je graf vzniklý z původního zorientováním jeho hran. Co takový graf musí splňovat?

i) Z každého vrcholu musí existovat alespoň jedna cesta vedoucí do cíle
ii) V grafu nesmí být kružnice, aby se závodník nemohl zacyklit

A jak vypadá algoritmus, který vyhovující řešení najde? Nejjednodušší je graf projít do šířky nebo do hloubky směrem od cíle a u každého vrcholu si zapamatovat, v kolikátém kroku jsme ho navštívili. Nakonec projdeme všechny hrany a zorientuje je tak, aby vedly vždy z vrcholu s vyšším pořadovým číslem do vrcholu s číslem nižším. Podmínka (i) je tak splněna, neboť každá cesta je posloupnost vrcholů s klesajícím pořadím a platí, že libovolný vrchol kromě cíle má souseda s nižším číslem. To je zřejmé z toho, jak jsme grafem procházeli. Cíl je vrchol s nejnižším pořadovým číslem, a proto v něm všechny cesty musí končit. Podmínka (ii) je rovněž splněna, neboť každá cesta obsahuje vrcholy v klesajícím pořadí a taková cesta nemůže být uzavřená.

A jak vypadá implementace? Ve skutečnosti si nemusíme žádná čísla pamatovat, stačí, když pro dva vrcholy dokážeme určit, který z nich má menší pořadové číslo. Když jsme dospěli do nějakého vrcholu a procházíme všechny hrany z něj vedoucí, pak vrchol na druhém konci buď již navštíven byl a má tak pořadové číslo jistě nižší, nebo ještě navštíven nebyl. Potom bude jeho pořadové číslo zákonitě vyšší.

Je tedy možné hrany orientovat již během prvního průchodu a dokonce jejich orientaci ihned vypisovat tak, že vypisujeme pouze nově zorientované hrany. Přičemž nově zorientované hrany vždy směřují do zpracovávaného vrcholu. Hrany z něj vycházející totiž byly zorientovány již před jeho zpracováním. A to je celé.

Časová i paměťová složitost algoritmu je zřejmě $O(N+M)$, kde N je počet křižovatek a M je počet ulic. Program implementuje prohledávání do šířky a pro zpestření jsou seznamy následníků vrcholu reprezentovány spojovými seznamy, aby paměťová složitost byla opravdu $O(N+M)$.

```
program BehMestem;
const MAXN = 512;

type Uk = ^tUzel;
     tUzel = record
       info : integer;
       dalsi : Uk;
     end;
     tVrchol = record
       navstiven : boolean;
       naslednici : Uk;
       stupen : integer;
     end;
var FIFO : array [0..MAXN] of integer;
    zac,kon : integer;
    vrchol : array [1..MAXN] of tVrchol;
    v1,v2,i,j,x,N,M,cil : integer;
    P : Uk;
```

```
begin
  readln(N,M,cil);

  for x:=1 to N do begin
    vrchol[x].naslednici:=nil;
    vrchol[x].stupen:=0;
    vrchol[x].navstiven:=false;
  end;

  for x:=1 to M do begin
    readln(i,j);
    { vytváření seznamu následníků }
    new(P); P^.info:=i; P^.dalsi:=vrchol[j].naslednici; vrchol[j].naslednici:=P;
    new(P); P^.info:=j; P^.dalsi:=vrchol[i].naslednici; vrchol[i].naslednici:=P;
  end;

  FIFO[0]:=cil;                    { začínáme od cíle }
  zac:=0; kon:=1;
  while zac<kon do begin           { dokud je co zpracovávat }
    v1:=FIFO[zac];
    inc(zac);

    vrchol[v1].navstiven:=true;
    P:=vrchol[v1].naslednici;
    while P<>nil do begin           { projdeme všechny sousedy }
      v2:=P^.info;
      P:=P^.dalsi;
      if not vrchol[v2].navstiven then begin
        FIFO[kon]:=v2;
        inc(kon);
        writeln(v2,' -> ',v1);     { nově zorientované hrany vypisujeme }
      end;
    end;
  end;
end.
```

18-4-4 Metro pro krtky — Jan Bulánek

Ukončete nástup, dveře se zavírají.

Vítejte v krtčím metru. Naši krtečci byli velmi potěšeni, protože všichni poslali funkční řešení. Bohužel u velké části z vás by krtečci už pro několik desítek tisíc krtin (což je při třiceticentimetrových krtincích těsně vedle sebe krtčí Stonehenge o průměru pouhopouhý kilometřík) značně zešedivěli. I přesto vaše řešení v $O(N^2)$ získala až 6 bodů, protože krtčí správa v městečku Jablonec nad Krysou s nimi byla spokojena. Ovšem naše cíle jsou vyšší. Ano, míříme až do krtkopolis Krtečkov. A tady se s ničím menším (větším) než $O(N \log N)$ nespokojí.

Ale jak jen docílit této mety nejvyšší (nejnižší)? Představme si tunely jako intervaly omezené úhlem krtin. Abychom si situaci ulehčili, budeme za levý

okraj považovat menší z obou hodnot. Díky tomu si můžeme kružnici v nule přestřihnout, aniž bychom tak rozdělili nějaký interval na dva (to si rozmyslete!), a nahradit ji úsečkou od $0°$ do $360°$. Úlohu jsme si tím změnili na nalezení počtu průniku intervalů na úsečce.

Vezměme si interval s nejmenším levým okrajem. V tomto intervalu mohou ostatní tunely začínat nebo začínat a končit, ale hlavně v něm nemohou pouze končit. Proč? Kdyby v něm nějaký interval pouze končil, tak by musel mít počáteční krtinu ještě před tímto intervalem, ale to je spor, protože ten byl vybrán právě proto, že je jeho levý okraj nejmenší.

Našim cílem je tedy spočítat, kolik tunelů v tomto intervalu začíná a zároveň nekončí. To je možné třeba tak, že spočteme, kolik jich tam začíná, a od tohoto počtu odečteme, kolik jich tam končí. To vede na pěkné kvadratické řešení. Nebo ne?

Setřiďme si krtiny podle úhlu od nejmenší. Nyní v takto setříděném poli prohlašme krtinec tunelu s menším úhlem za počáteční, naopak s větším úhlem za koncový. Pokud si počátky označíme 1 a konce -1 tak si povšimněte, že součtem čísel na určitém úseku získám, o kolik je v daném úseku více nebo méně počátků než konců. Například pokud zde začne 9 tunelů a jen 5 skončí je součet čísel na daném úseku 4. Doplňme si počet krtin na nejbližší mocninu dvojky (nulami, tím nic nezkazím) a vystavme nad nimi binární strom a to tak, že v každém uzlu bude součet jeho levého a pravého syna. Jen tak mimochodem si uvědomte, že v kořeni tohoto stromu musí být 0. Navíc si pro každý začátek tunelu pamatujme, kde je umístěn jeho konec. V takovémto stromu je jednoduché jedním průchodem z listu, ve kterém je umístěn konec tunelu, do kořenu najít součet prvků nalevo nebo napravo od něj.

Tím jsme vyřešili první tunel, ale co ten druhý? Třetí? Chtělo by to první odebrat. Ale to je snadné, na umístění jeho krtin dáme nuly a strom opět, tentokrát dvěma průchody (za každou krtinu jeden) opravím. Obě tyto operace mají složitost $O(\log N)$. Tím se druhý tunel stane prvním a vše může pokračovat.

Teď ještě pár slov k těmto dvěma operacím. Začneme opravením. Nejprve změním daný list na nulu. Pak skočím do jeho otce a přiřadím mu součet jeho synů. Tím se opět změnila jeho hodnota a tak opět skočím o úroveň výš. To budu opakovat dokud nedojdu do kořene. Trochu složitější se mi zdá zjištění počtu počátečních a koncových krtin. Představte si nyní nějakou (pokud možno hodně klikatou) cestu z listu ke kořeni. To co my chceme získat, je součet všech listů nalevo od našeho. Teď se posuňme o úroveň výš. Mohly nastat dva případy

i) do otce jsme přišli zleva. Tak nic neřešíme, protože pravý list obsahuje jistě součet (nějakých) listů napravo od našeho.

ii) do otce jsme přišli zprava. To je mnohem zajímavější, v tom případě navýšíme počítadlo o hodnotu levého syna, protože je to součet úseku vlevo od našeho listu. Pokud si navíc nakreslíte obrázek, tak krásně

uvidíte, že tento postup skutečně pokryje celý interval vlevo od našeho vrcholu.

Výsledek v počitadle získáme v $O(\log N)$, protože v každém kroku skočíme o úroveň výš, ve stejném čase také opravíme strom. Protože tyto operace provedeme $N/2$-krát (tolik je počátečních krtin), je celková složitost $O(N \log N)$. Paměti potřebujeme na stromeček a umístění konců pouze lineárně mnoho, takže $O(N)$. Samozřejmě časová složitost je vzhledem k použití QuickSortu jen průměrná, ale použitím jiného sortu bychom si ji zajistili.

Ukončete výstup, dveře se zavírají.

```
program Metro;
const MAX=1024;

type node=record                          { uzel ve stromě }
       id:integer;
       val:real;
     end;
var tree:array[1..MAX] of node;
   place:array[1..MAX div 2] of integer;{ umístění konců úseček }
   N,N2,i,poc_prusec:integer;

procedure sort(a,b:integer);              { QuickSort }
var i,j:integer;
   pivot:real;
   pom:node;
begin
   i:=a;j:=b;pivot:=tree[(a+b) div 2].val;
   while(i<=j) do begin
      while(tree[i].val<pivot) do Inc(i);
      while(tree[j].val>pivot) do Dec(j);
      if (i<j) then begin
         pom:=tree[i];tree[i]:=tree[j];tree[j]:=pom;
      end;
      Inc(i); Dec(j);
   end;
   if (a<j) then sort(a,j);
   if (b>i) then sort(i,b);
end;

procedure repair(i:integer);              { opraví strom z daného listu }
begin
   while(i>1) do begin
      i:=i div 2;
      tree[i].val:=tree[2*i].val+tree[2*i+1].val;
   end;
end;

function value(i:integer):integer;{ vrací počet otevřených cest od začátku k i }
var pom:real;
```

```
begin
   pom:=tree[i].val;                    { krajní bod tam patří také }
   while (i>1) do begin
      if (i mod 2 = 1) then pom:=pom+tree[i-1].val;
      { je-li pravý syn, tak hodnota levého značí kolik konců nebo začátků leží
      uvnitř intervalu, tím vlastně získáme počet začátků,které nejsou uzavřeny.
      Protože začátků bude alespoň tolik, kolik konců, vyjde kladné číslo }
      i:=i div 2;
   end;
   value:=round(pom);
end;

begin
   writeln('Zadej pocet krtin');readln(fin,N);
   N2:=1;
   while N2<N do N2:=2*N2;              { nalezení nejbližší vyšší mocniny 2 }

   for i:= 1 to N div 2 do begin        { načtení cest }
      writeln('Zadej 1. krtinu cesty');readln(tree[N2].val);  { načítám rovnou }
      writeln('Zadej 2. krtinu cesty');readln(tree[N2+1].val); {do listu stromu}
      tree[N2].id:=i;                   { uložení indexu cesty }
      tree[N2+1].id:=i;
      Inc(N2,2);
   end;

   N2:=N2-N;                            { návrat na první list }
   sort(N2,2*N2-1);
   for i:=2*N2-1 downto N2 do begin
      if (tree[i].id<>0) then begin
         if (place[tree[i].id]=0) then begin { je to začátek tunelu }
            tree[i].val:=-1;{ jeho hodnota je nám už k ničemu, teď značí konec }
            place[tree[i].id]:=i;       { kde je konec }
         end else tree[i].val:=1;       { začátek tunelu }
      end;
   end;

   for i:= N2-1 downto 1 do             { naplnění stromu }
      tree[i].val:=tree[2*i].val+tree[2*i+1].val;

   for i:=N2 to 2*N2-1 do begin
      if tree[i].id<>0 then begin
         poc_prusec:=poc_prusec+value(place[tree[i].id]);
         tree[place[tree[i].id]].id:=0;
         tree[place[tree[i].id]].val:=0;{ neutralní hodnota nic neovlivní }
         repair(place[tree[i].id]);     { opravi strom v místě konce }
         tree[i].val:=0;
         repair(i);                     { oprava v místě začátku tunelu }
      end;
   end;
   writeln('Bude treba ',poc_prusec,' zizal.');
end.
```

18-4-5 Datakopci — Tomáš Gavenčiak

Nejprve bylo dobré povšimnout si, že protože každá zpráva bude doručena (to plyne přímo ze zadání – každý dopravník buď vede přímo do redakce nebo pokračuje pásem na sousedním políčku, který vede stejným směrem), stačí nám pamatovat si rozdíl počtu dobrých a špatných zpráv s tím, že znaménka množství doručených na severní okraj obrátíme. Také nám hodně pomůže fakt, že hranice mezi S-J a Z-V pásy bude lomená čára začínající v severozápadním (levém horním) rohu, vedoucí pouze na jih a východ a končící v jihovýchodním (pravém dolním) rohu. Jižně a západně od ní leží dopravníky ženoucí se na západ do redakce pohádek, na sever a západ ty vedoucí na sever do redakce zpravodajství. Proč nemůže mít i nějaký ten záhyb na sever či západ? Zprávy z něj by se už určitě nedostaly a to je zakázáno. Zkoušet všechny tyto dělící cesty není dobrý nápad, je jich totiž $n^n/n!$, ačkoliv by to byla teoretická možnost.

My úlohu vyřešíme lépe a to dynamickým programováním. Jakkoli hrozně vám některým může tento pojem znít, jde vlastně jen o to, že si pro každý řádek pro všechna $k = 0 \ldots n$ postupně spočtu, jaký nejlepší výsledek mohu dostat, pokud v něm vede právě k dopravníků na západ a zbylé na sever. Pro první řádek to zjistíme snadno a pro $j+1$-ní to závisí pouze na optimech pro j-tý řádek – když potřebuji zjistit optimum pro situaci, kdy k dopravníků vede na západ, vyzkouším to zkombinovat se situacemi, kdy dělící cesta pokračuje o řádek výše v pozici $0, 1, 2, \ldots, k$. Toto stihnu mnohem rychleji – kombinací k_{j+1} s možnými k_j je $O(n^2)$, každý z přepočtů mi bude trvat $O(n)$ a toto udělám pro $O(m)$ řádků, celkem tedy $O(n^3m)$ nebo $O(m^3n)$ pokud bych se místo po řádcích rozhodl jít po sloupcích, což by byla stejná jen zrotovaná úvaha.

Zkusme to ale zlepšit. Toho, že toto spočteme pro $O(m)$ řádků se nezbavíme, zrychlíme ale jednotlivé řádky. První zrychlení dosáhneme předpočítáním částečných součtů zleva na jednotlivých řádcích (při jeho rozdělení v jednotlivých místech by nám řádek vynesl $\sum_{i=1}^{k} d_i - \sum_{i=k+1}^{n} d_i = 2\sum_{i=1}^{k} d_i - \sum_{i=1}^{n} d_i = 2s_k - konst.$, konstantu nezávisející na místě rozdělení mohu ignorovat a počítat jen s $2s_k$ a to klidně polovičním. Tímto si trochu zjednoduším počítání s_k), když už si totiž vybereme konkrétní k_j a k_{j+1}, je nové optimum jen součtem tohoto částečného součtu $s_{k_{j+1}}$ a optima na k_j v minulém řádku. Tímto jsme dosáhli $O(n^2m)$ (předpočítat částečný součet umíme v čase $O(n)$ na řádek).

Další a poslední zlepšení: při zkoušení optimálního pokračování dělící cesty o řádek výše vždy vybereme maximum z možných mezivýsledků. Toto nalezené maximum pro k_{j+1} si ale můžeme schovat pro $k_{j+1}+1$ (další políčko na řádce) a jen ho třeba zlepšit na $s_{k_{j+1}}$, pokud je větší. Tento malý trik nám zlepší složitost až na $O(mn)$. Nejen, že už nezáleží na tom, zda jdeme po sloupcích či řádcích, navíc jsme určitě dosáhli optima – na každou hodnotu musíme v libo-

volném fungujícím algoritmu sáhnout alespoň jednou, mohlo by se tam ukrývat nějaké velmi velké množství pohádek, o které přece nechceme přijít...

Paměťová složitost je $O(mn)$.

```
#include <stdio.h>
#define M 4
#define N 4
int dobre[M][N], spatne[M][N];

  /* nultý sloupec je vždy 0 (pro usnadnění) */
int d[M][N+1];                          /* rozdíly počtu dobrých a špatných */
int s[M][N+1];                          /* částečné součty */
int o[M][N+1];                          /* optima */
  /* pro zjištění tvaru výsledné dělící cesty si pamatuji, */
  /* kde pokračovala o řádek výše pro všechna její pokračování tady */
int k[M][N+1];
int nazapad[M+1];                       /* kolik cest nakonec kde vede na Z */

int main () {
    int i, j, max;
    /* zde se načte vstup, neimplementováno */
    for (i=0; i<M; i++) {               /* spočtu rozdíly a částečné součty */
        d[i][0]=s[i][0]=o[i][0]=k[i][0]=0;
        for (j=1; j<=N; j++) {
            d[i][j]=dobre[i][j-1]-spatne[i][j-1];
            s[i][j]=s[i][j-1]+d[i][j];
        }
    }
    for (j=0; j<=N; j++) o[0][j]=s[0][j];   /* první řádek */
    for (i=1; i<M; i++) {               /* a ty další */
        max=0;                          /* pozice maxima */
        for (j=1; j<=N; j++) {
            if (o[i-1][max]<o[i-1][j]) max=j;
            o[i][j]=s[i][j]+o[i-1][max];
            k[i][j]=max;
        }
    }
    max=0;
    for (j=1; j<=N; j++) if (o[M-1][max]<o[M-1][j]) max=j;
    /* nyní mám v max optimální konec cesty, z něj, zpětně zjistím, kudy optimálně vedla */
    nazapad[M-1]=max;
    for (i=M-2; i>=0; i--) nazapad[i]=k[i+1][nazapad[i+1]];
    max=0;                              /* a ještě výpis */
    for (i=0; i<M; i++) {
        for (j=0; j<nazapad[i]; j++) { max+=dobre[i][j]; printf ("-"); }
        for (; j<N; j++) { max+=spatne[i][j]; printf ("|"); }
        printf ("\n");
    }
    printf ("Vydoluje se %d zpráv\n", max);
    return 0; }
```

18-4-6 Kompilátorový φgl — Zdeněk Dvořák

Jedním z možných řešení této úlohy bylo použít algoritmus popsaný v řešení úlohy 18-3-6 a modifikovat ho pro práci nad SSA formou. Existuje ovšem přímočařejší řešení – vyjdeme z definice živého kódu.

Výraz je živý, pokud má nějaké vedlejší efekty, nebo pokud je jeho hodnota použita v živém výrazu. Představme si následující orientovaný graf: jeho vrcholy budou příkazy v programu. V grafu bude hrana z příkazu p_1 do příkazu p_2, pokud p_1 používá hodnotu vypočtenou v příkazu p_2, tedy pokud p_2 je buď přiřazení `x = expr`, nebo `x = phi(...)`, a příkaz p_2 používá proměnnou x. Vrcholy odpovídající příkazům s vedlejšími efekty si označíme. Pak příkaz je živý právě tehdy, pokud do jemu odpovídajícímu vrcholu vede cesta z některého z označených vrcholů. Toto je ovšem dobře známá úloha na dosažitelnost v grafu, kterou můžeme vyřešit například procházením grafu do hloubky (viz třeba kuchařku z druhé série). Algoritmus tedy může vypadat zhruba takto:

foreach příkaz p:
  if p má vedlejší efekty **then**
    označ p a ulož ho na zásobník

while zásobník není prázdný **do**
  odeber ze zásobníku příkaz (p)
  pro všechny proměnné x použité v p:
    buď q příkaz, který definuje x
    if q není označený **then**
      označ q a přidej ho do zásobníku

foreach příkaz p:
  if p není označený **then**
    smaž p

V algoritmu využíváme toho, že x je definováno jen jedním příkazem, a tedy nemusíme zjišťovat, které z definic odpovídají danému použití. Pro každý příkaz v programu si musíme pamatovat značku a mít pro něj místo na zásobníku, tedy paměťová složitost je lineární. Každý příkaz se dostane do zásobníku nanejvýš jednou, proto je časová složitost také lineární.

18-5-1 Účetní — Martin „Bobřík“ Kruliš

Řešení došlo opravdu hodně a většina jich byla i správně. Finančnímu úřadu tedy nezbývá než doufat, že se z vás stanou programátoři, a ne třeba ... účetní. Pro ty z vás, kterým se úlohu náhodou vyřešit nepodařilo, posíláme řešení vzorové, abyste měli v kriminále o čem přemýšlet :)

Vězte, že k získání maxima bodů nestačil pouze správný popis řešení. Nezbytnou součástí musel být i důkaz, aby vám účetní věřili, že je nechcete podfouknout.

Situace pro dva účetní je velice jednoduchá. První rozdělí majetek na dvě poloviny podle svého mínění. Druhý si vybere polovinu, která se mu zdá větší. První je spokojený, protože dostal přesně polovinu majetku. Druhý je taktéž spokojený, protože si vybral větší díl ze dvou, takže má alespoň polovinu majetku (podle jeho mínění).

Ve třech je situace o trochu komplikovanější, protože musíme rozebrat několik případů. Dokonce existuje několik řešení, jak účetní spravedlivě podělit. Nyní si ukážeme jedno z nich: První rozdělí majetek na tři díly, o kterých tvrdí, že jsou to přesně třetiny (označím je A, B a C). Nyní druhý a třetí ukáží na hromádku, která se jim zdá největší. Zde mohou nastat dva případy. Buď oba ukázali na různé hromádky (řekněme, že 2. ukázal na B a 3. ukázal na C). Pak si tyto hromádky nechají a prvnímu zbude poslední hromádka (tj. A). Všichni budou spokojeni, protože 2. a 3. si myslí, že mají největší hromádku ze tří (což je určitě víc než 1/3 a 1. dostane právě třetinu (sám si to tak rozdělil). Komplikace nastane, pokud 2. a 3. ukáží na stejnou hromádku (řekněme na A). V takovém případě si tuto hromádku rozdělí půl na půl (předchozím algoritmem) a znovu ukážou na hromádku, která se jim zdá největší (ze zbývajících dvou - B a C). I zde mohou nastat dva případy. Pokud oba ukáží na stejnou hromádku (řekněme B), tak si ji rozdělí „fifty-fifty“ a poslední hromádku C nechají prvnímu. První bude spokojen určitě, protože je mu jedno, kterou hromádku dostane. Druhý a třetí budou spokojeni rovněž, neboť si mezi sebe spravedlivě rozdělili dvě „největší“ hromádky (lze jednoduše nahlédnout, že tyto hromádky obsahují alespoň 2/3 majetku). Druhá možnost je, že při druhém výběru si každý vybere jinou hromádku (řekněme 2. vybere B a 3. vybere C). V takovém případě se oba podělí o zvolenou hromádku s prvním účetním (algoritmem dělení pro dva). Opět budou všichni spokojeni. První dostal dvakrát alespoň 1/2 z 1/3, takže má nejméně $2 \cdot 1/2 \cdot 1/3 = 1/3$. Druhý a třetí dostali každý alespoň polovinu z dvou (podle nich) největších hromádek, takže mají každý alespoň 1/3.

Abychom si ukázali i jiný algoritmus na dělení, budeme řešit úlohu N účetních trochu jinak. Řešení i důkaz bude založené na rekurzi a matematické indukci. Předpokládejme, že umíme vyřešit úlohu pro $N-1$ účetních a chceme ji rozšířit na N účetních. Vezmeme prvního účetního a na chvíli jej postavíme stranou. Teď necháme zbývajících $N-1$ účetních, aby si rozdělili spravedlivě všechen majetek mezi sebe. Každý účetní kromě prvního má tedy svoji hromádku (označme je $H_2, H_3 ... H_N$) o které si myslí, že obsahuje alespoň $1/(N-1)$ majetku. Nyní poprosíme jednotlivé účetní, aby každý rozdělil svoji hromádku na N stejných podhromádek. První účetní pak obejde všechny své kolegy a od každého si vezme jednu podhromádku. Všichni by měli být spokojeni. První dostal z každé hromádky alespoň $1/N$-tinu. O jednotlivých hromádkách toho z pohledu prvního účetního moc tvrdit nemůžeme, ale víme, že jejich součet je 1

(dohromady tvoří celý majetek):

$$1/N \cdot H_2 + 1/N \cdot H_3 + \ldots + H_N = 1/N \cdot (H_2 + H_3 + \ldots + H_N) = 1/N \cdot 1 = 1/N$$

První má tedy alespoň $1/N$-tinu majetku. Ostatní by měli být také spokojeni. Při počátečním dělení dostal každý hromádku, na které bylo (podle jeho mínění) alespoň $1/(N-1)$ majetku. Následně jej první obral přesně o $1/N$-tinu této hromádky, takže každému zbyla přesně $(N-1)/N$-tina toho, co už dostal. Ve výsledku každému zbylo $1/(N-1) \cdot (N-1)/N = 1/N$ majetku.

Všichni jsou spokojeni (no, až na finanční policii) a tak hurá na Seychely...

18-5-2 Permutovat se musí legálně! Jan Bulánek & Zbyněk Falt

Pro „výzkumné“ účely si úlohu pozměňme: Chceme vypsat všechny permutace dané množiny lexikograficky setříděné. Jak na to? Zafixujeme nejmenší prvek z množiny na začátku a za něj postupně připojíme všechny lexikograficky setříděné permutace zbývajících prvků. Poté zafixujeme druhý nejmenší prvek a postup zopakujeme, atd. Takto by bylo možné napsat rekurzivní funkci, která by vždy fixovala prvky od nejnižšího k nejvyššímu (zafixovaný prvek označme jako prvek přilehlý) a pomocí sama sebe by vygenerovala všechny permutace zbylých prvků (jejich množinu označme jako zbytek).

Jak nyní vyřešíme původní úlohu? Stačí najít zbytek, který v zadané permutaci právě dopermutoval, a k němu přilehlý prvek. Nyní můžeme permutovat zbytek opět od začátku, s tím rozdílem, že pouze vyměníme přilehlý prvek s nejbližším vyšším prvkem z nalezeného zbytku (tedy to samé, co v modifikované úloze).

A implementace? Prvním úkolem je najít zbytek, který v zadané permutaci právě dopermutoval. To je ale snadné, neboť víme, že se nachází na konci permutace a ona dopermutovanost znamená, že zbytková posloupnost je sestupná a nejdelší možná. Začít permutovat zbytek od začátku je také jednoduché. Stačí si uvědomit, že jediné co je třeba udělat, je setřídit ji vzestupně. To bylo ale kamenem úrazu některých řešení, neboť posloupnost třídili některým z třídících algoritmů místo aby si všimli, že ze sestupné posloupnosti vytvoříme vzestupnou tak, že obrátíme pořadí jejích prvků.

Záměna přilehlého prvku je již triviální. Problém může nastat pouze tehdy, když žádný přilehlý prvek neexistuje. To se ale stane jenom tehdy, když jsme na vstupu dostali lexikograficky nejvyšší možnou permutaci.

Paměťová i časová složitost je $O(N)$.

```
program Permutace;

procedure swap(var a, b : char);
var p : char;
begin p:=a; a:=b; b:=p; end;
```

```
var perm : string;
    N, i, j : integer;

begin
   readln(perm);
   N := length(perm);

   i := N;
   while (i > 1) and (perm[i-1] > perm[i]) do dec(i);  { Hledáme zbytek }
   if i > 1 then begin
      for j := 0 to (N-i-1) div 2 do
         swap(perm[j+i], perm[N-j]);                   { Setřídíme zbytek }

      j := i;
      dec(i);
      while (perm[i] > perm[j]) do            { Najdeme první zlom v permutaci }
         inc(j);
      swap(perm[i], perm[j]);                 { Zaměníme je }
      writeln(perm);
   end
   else writeln('Dopermutovali jsme');          { Byla to poslední permutace }
end.
```

18-5-3 Číňanské volby — Zdeněk Dvořák

Představme si následující „paralelní“ algoritmus na vyhodnocení voleb: každý Číňan si najde do dvojice nějakého Číňana, který chce volit někoho jiného. Pokud má některý kandidát nadpoloviční většinu hlasů, někteří z jeho příznivců zůstanou nespárovaní; u takto nalezeného potenciálního vítěze si ještě ověříme, zda skutečně vyhrál (spočítáme si počet jeho hlasů).

Druhou část tohoto algoritmu jistě zvládneme v lineárním čase a s konstantní pamětí, zbývá si rozmyslet, jak realizovat tu první. Zjevně nás nezajímá, jak jsou Číňané spárováni, stačí nám vědět, kolik jich zůstane nespárovaných a pro koho nespárovaní hlasují. Budeme si tedy udržovat dvě čísla – K (číslo kandidáta, pro nějž hlasují nespárovaní Číňané), a C (počet nespárovaných Číňanů). Postupně procházíme všechny Číňany. Pokud aktuální Číňan hlasuje pro kandidáta K, nejde ho spárovat, a proto zvýšíme C o jedna. Pokud hlasuje pro někoho jiného (kandidáta X), rozlišíme dva případy: jestliže je $C > 0$, pak tohoto voliče spárujeme s jedním z voličů kandidáta K, tedy snížíme C o jedna. Jestliže je $C = 0$, nelze aktuálního Číňana spárovat, a proto dosadíme $K = X$ a $C = 1$.

Tento algoritmus používá konstantní množství paměti a seznam hlasů projde dvakrát, časová složitost je tedy $O(N)$. Pokud bychom chtěli být přesnější, je třeba vzít do úvahy to, že na uložení čísel potřebujeme $O(\log N)$ bitů, což pro velká N nelze zanedbat tedy paměťová složitost je $O(\log N)$ a časová $O(N \log N)$.

```
program Cina;
const N = 100;
var hlasy : array[1..N] of integer;
    i, x, k, c : integer;

begin
   k := 0; c := 0;                                  { Najdeme kandidáta. }
   for i := 1 to N do begin
      readln(x); hlasy[i] := x;
      if c = 0 then k := x;
      if k = x then inc (c) else dec (c);
   end;

   c := 0;                                          { Ověříme, zda vyhrál. }
   for i := 1 to N do begin
      x := hlasy[i];
      if k = x then inc (c);
   end;

   if 2 * c > N then writeln ('Vyhrál kandidát číslo ', k, '.')
   else writeln ('Nikdo nevyhrál.')
end.
```

18-5-4 Detektýv — Martin Mareš

S důkladností takřka šerlokovskou prozkoumáme několik možných řešení, až usvědčíme to nejrychlejší. Označme si (věrni písmenkům ze zadání) N délku stopovaného řetězce, k počet podezřelých sekvencí, $p_1, \ldots, p_k$ délky těchto sekvencí a $P = p_1 + \ldots + p_k$ jejich celkovou délku.

0. pokus (jak by ho vymyslel strážník Vopička): Budeme hledat každou sekvenci zvlášť, a to tak, že si po vstupu „pojedeme okénkem“ délky p_i a vždy porovnáme, jestli se okénko rovná i-té sekvenci. Kdybychom si okénko ukládali jako cyklické pole, zvládli bychom ho posunout v konstantním čase, ale stejně nás nemine čas $O(p_i)$ na porovnání. Celkově trvá $O(Np_1 + \ldots + Np_k) = O(NP)$ a navíc potřebujeme k-krát volat rewind.

1. pokus (inspektor Neverley): Damned, na hledání výskytů jednoho řetězce přeci můžeme použít algoritmus KMP z té vaší cookbook, takže jeden průchod zvládneme v timu $O(N + p_i)$, celkově tedy $O(Nk + P)$ s k rewindy. That's it.

2. pokus (policejní rada Žák): V kuchařce je přeci i algoritmus A-McC na hledání výskytů více slov najednou. Stačí, když hlášení výskytu nahradíme připočtením jedničky k počítadlu. (Na to praktikant Hlaváček:) Dobrý plán, pane rado, ale má jedno háčisko jak na sumce: jelikož se sekvence mohou překrývat, může jich v jednom místě končit až k, takže jsme opět na $O(Nk + P)$, i když tentokrát bez rewindů.

3. pokus (Šérlok osobně): Postavíme si vyhledávací automat jako v minulém pokusu, ale místo abychom počítali rovnou výskyty, budeme si pamatovat jen

to, kolikrát jsme prošli kterým stavem, a pak z toho výskyty dopočítáme. Well, ale jak?

Pokud máme nějaký stav α (o kterém víme, že je prefixem některého z vyhledávaných slov, takže mimo jiné mezi stavy najdeme všechny sekvence stop, které počítáme) a chceme zjistit, kolikrát se slovo α v textu vyskytlo, stačí sečíst počet průchodů tímto stavem a všemi dalšími stavy, které končí na α, což jsou přesně ty, ze kterých se do α lze dostat pomocí zpětné funkce (případně zavolané vícekrát).

Stačí tedy projít automat v opačném pořadí, než ve kterém jsme vytvářeli zpětnou funkci (nejlepší bude si během konstrukce automatu toto pořadí zapamatovat, třeba v poli, v němž jsme měli uloženu frontu). Pro každý stav α pak přičteme počítadlo odpovídající tomuto stavu k počítadlu stavu, do nějž vede z α zpětná funkce. (To se pak přičte podle další zpětné funkce atd., takže počítadlo stavu α se opravdu postupně popřičítá ke všem rozšířením stavu α.)

To vše zvládneme v čase $O(P + N + P)$ (konstrukce automatu + průchod textem + dopočítání), čili $O(P + N)$, a v paměti $O(P + N)$, bez jediného zavolání rewindu.

Program obnáší připsání cca čtyř řádků ke zdrojáku z kuchařky. Abychom z toho nevybruslili tak snadno, ukážeme si trochu jinou implementaci v Céčku.

It's a lemon tree, my dear Watson!

```
/* Counting words. MM scribebat me per III Id. Mai. MMDCCLIX AUC. */

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

struct state {                                  /* jeden stav automatu */
    struct state *fwd[256], *back;              /* hrany dopředné a zpětná */
    struct state *qnext, *qprev;                /* předchůdce a následník ve frontě */
    char *out;                                  /* které slovo v tomto stavu končí */
    int count;                                  /* počet průchodů stavem */
};

struct state root;                              /* počáteční stav (kořen stromu) */
struct state *head, *tail;                      /* první a poslední ve frontě */

struct state *step (struct state *s, int c) {   /* jeden krok automatu (včetně */
    while (s) {                                 /* vracení se po zpětných hranách) */
      if (s->fwd[c])                            /* hrr na ně! */
        return s->fwd[c];
      s = s->back;                              /* a když to nejde, tož cófnem */
    }
    return &root;
}

void insert (char *x) {                         /* vložení jednoho slova do stromu */
    struct state *s = &root;
    int c;
    for (char *y = x; c=*y++; ) {
      if (!s->fwd[c]) s->fwd[c] = calloc (1, sizeof (struct state));
```

```
        s = s->fwd[c];
    }
    s->out = x;
}
void build (void) {                          /* konstrukce zpětných hran přesně */
    struct state *s;                         /* podle kuchařky */
    head = tail = &root;
    while (head) {
        for (int c=0; c<256; c++)
            if (s = head->fwd[c]) {
                s->back = step (head->back, c);
                tail->qnext = s;
                s->qprev = tail;
                tail = s;
            }
        head = head->qnext;
    }
}
void run (void) {                            /* projití celého vstupu s počítáním */
    struct state *s = &root;                 /* průchodů stavy */
    for (int c; (c = getchar ()) != EOF; ) {
        s = step (s, c);
        s->count++;
    }
}
void count (void) {                          /* propagování počtů po */
    for (struct state *s=tail; s != &root; s=s->qprev) {   /* zpětných hranách */
        s->back->count += s->count;
        if (s->out) printf ("%6d %s\n", s->count, s->out);    /* ... a vypisování */
    }
}
int main (int argc, char **argv) {           /* main sweet main :) */
    for (int i=1; i<argc; i++)
        insert (argv[i]);
    build ();
    run ();
    count ();
    return 0;
}
```

18-5-5 Do vysokých kruhů — Tomáš Gavenčiak

Nejprve bylo potřeba oblasti převést na objekty, se kterými umíme manipulovat rozumněji než s obecnými množinami bodů v rovině. Velmi užitečné je představit si protínající se kružnice jako graf s průsečíky a dotyky kružnic jako vrcholy. Hrany budou oblouky mezi sousedními vrcholy. Tento graf je vlastně multigraf, což je graf, ve kterém může mezi dvěma vrcholy vést více než jedna hrana a z jednoho vrcholu do toho samého může vést více než jedna smyčka.

Takový graf je určitě jednoznačně zadán polohami a poloměry kružnic a je rovinný (původní rozmístění kružnic je jeho rovinné nakreslení). Bohužel se nám do něj nijak nepromítnou izolované kružnice, ty je třeba ošetřit jinak.

Nyní se nám z na první pohled neuchopitelného problému stal problém mnohem jednodušší – spočítat stěny rovinného grafu. K tomu se ideálně hodí Eulerova věta:

$$V + F = K + E + 1$$

Toto je vztah mezi počtem vrcholů (V), stěn (včetně té vnější) (F), komponent souvislosti (K) a hran (E). Tato věta platí pro rovinné grafy a platí i pro multigrafy, pokud si zvolím, že mezi „rovnoběžnými“ násobnými hranami jsou také stěny a že smyčka přidává jednu stěnu. Toto rozšíření přesně odpovídá naší představě toho, jak kružnice dělí rovinu na oblasti.

Věta se dokazuje indukcí podle složitosti grafu. Pro prázdný graf určitě platí ($0+1 = 0+0+1$). Přidáme-li nový vrchol, stoupnou V i K o jedna a rovnost zůstane zachována. Přidáme-li hranu a zvýšíme tak E o jedna, pak jsme buď spojili dvě komponenty souvislosti a snížili K o jedna, nebo přidali jednu stěnu rozdělením nějaké existující na právě dvě. V obou případech zůstane rovnost zachována a druhý případ navíc zahrnuje přidávání násobných hran a smyček. Každý rovinný (multi)graf lze postavit z prázdného přidáváním vrcholů a hran, takže pro něj věta musí platit.

Stačilo by tedy spočítat počet komponent, hran a průsečíků. Víme, že na každé kružnici je stejně vrcholů a hran. Vrchol je ale sdílen mezi dvěma kružnicemi, zatímco hrana patří právě jedné. Jinak řečeno je stupeň každého vrcholu 4. Z toho plyne, že $E = 2V$. Tedy:

$$F = K + 2V + 1 - V = K + V + 1.$$

Tento vzorec nám navíc zahrne i izolované kružnice, počítáme-li je jako jednu komponentu bez průsečíků. To nám trochu zjednoduší algoritmus.

Stačí tedy spočítat počet komponent a průsečíků, obojí zvládneme v čase $O(N^2)$ průchodem do hloubky (s hledáním sousedů vyzkoušením všech) a vyzkoušením všech dvojic. Zkoušení dvojic navíc zahrneme do toho průchodu. V programu je průchod do hloubky realizován rekurzivní funkcí `navstiv()`. Ta bude pro každý vrchol spuštěna určitě právě jednou, určitě projde celou komponentu a zároveň správně napočítá počet průsečíků. Jen je si třeba dát pozor, abychom nezapočítávali průsečíky dvakrát (za páry kružnic (k_i, k_j) a (k_j, k_i)).

Toto řešení má časovou složitost $O(N^2)$, paměťovou $O(N)$. Existuje ještě jiné o dost složitější řešení používající *zametací čáru* k dosažení složitosti $O((N + V) \log N)$, což je lepší než naše $O(N^2)$, pokud je počet průsečíků $V < N^2/\log N$, tedy pro dost „řídké“ konfigurace kružnic. Pro $V = O(N^2)$ má ale časovou složitost až $O(N^2 \log N)$. Paměťová složitost tohoto algoritmu je $O(N)$. Jeho popis by ale byl dost komplikovaný a proto ho neuvádím.

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

#define SQR (x)  ( (x) * (x) )                    /* Druhá mocnina */
#define EPSILON  (0.000001)                       /* Podovnání s tolerancí */
#define EQ (x, y)  ( ( (x)+EPSILON> (y))&& ( (x)-EPSILON< (y)))
#define MAX_N 10000
int N;                                            /* Počet kružnic */

double x[N],  y[N],  r[N];                        /* Kružnice */
int byl[N];                                       /* Navštíveno? */
int v=0;                                          /* Vrcholy (průsečíky) */
int k=0;                                          /* Komponenty */

int pruseciku (int a,  int b) {                   /* Kolik je mezi a a b průsečíků? */
    double d=sqrt ( SQR (x[a]-x[b]) + SQR (y[a]-y[b]) );    /* Vzdálenost středů */
    if  (d>r[a]+r[b]) return 0;                   /* Úplně mimo */
    if  ( (d+r[a]<r[b]) || (d+r[b]<r[a]) ) return 0;    /* Jedna ve druhé */
    if  ( EQ (d, r[a]+r[b]) ) return 1;           /* Vnější dotyk */
    if  ( (EQ (d+r[a], r[b])) || (EQ (d+r[b], r[a])) ) return 1;    /* Vnitřní dotyk */
    return 2;                                     /* Jinak mají právě dva průsečíky */
}

void navstiv (int koho) {                         /* Rekurze pro průchod komponent */
    int i, p;

    byl[koho]=1;
    for  (i=0; i<N; i++) {                        /* Teď projdi všechny sousedy */
      p=pruseciku (i, koho);
      if  ( (i!=koho) &&  (p>0)) {
        if  (i<koho) v+=p;                        /* Připočti průsečíky (ale ne dvakrát) */
        if  (!byl[i]) navstiv (i);                /* Navštiv */
      }
    }
}

int main () {
    int i;
    scanf ("%d", &N);                             /* Načteme */
    for  (i=0; i<N; i++) {
      scanf ("%lf %lf %lf", & (x[i]), & (y[i]), & (r[i]));
      byl[i]=0;
    }
    for  (i=0; i<N; i++)
      if  (!byl[i]) {                             /* V této komponentě jsme ještě nebyli */
        k++;
        navstiv (i);
      }
    printf ("Oblastí: %d\n", k+v+1);
    return 0;
}
```

18-5-6 Proφlování — Zdeněk Dvořák

Úloha 1: Buď G graf, který vznikne z CFG tak, že zapomeneme na orientaci hran, přidáme hranu mezi vstupním a výstupním blokem CFG, a zahodíme hrany, na které nedáme čítač. Graf G nemůže obsahovat cyklus – počty provedení hran na odpovídajícím cyklu (nebo cestě) v CFG by bylo možné zvyšovat a snižovat bez toho, že bychom ovlivnili libovolný čítač, tedy by nebylo možné pouze z čítačů určit profil. G je tedy les, a má nejvýše $N - 1$ hran, kde N je počet bloků v programu. Proto v původním CFG můžeme vynechat čítač na nejvýše $N - 2$ hranách.

Naopak, pokud vynecháme čítač na libovolných $N - 2$ hranách takových, aby G byl strom, dokážeme počty provedení zbývajících hran dopočítat: protože G je strom, má vrchol stupně jedna (vede do něj jen jedna hrana e). To odpovídá bloku, u nějž neznáme počty provedení pouze jedné z hran, které do něj vstupují nebo z něj vystupují. Součet počtů provedení hran vstupujících do bloku je roven součtu počtů provedení hran, které z něj vystupují, tedy dokážeme dopočítat počet provedení hrany e. Hranu e vyhodíme z G a tento postup opakujeme, dokud neurčíme počty provedení všech hran. Časová i paměťová složitost tohoto algoritmu je lineární.

Úloha 2: Zjevně stačí spočítat počty provedení bloků – pokud hrana vychází z bloku, který se provede n-krát, a provedeme ji s pravděpodobností p, pak výpočet touto hranou projde pn-krát. Mějme blok b, do nějž vstupují hrany e_1, e_2, …, e_k, které vycházejí z bloků b_1, b_2, …, b_k s pravděpodobnostmi p_1, p_2, …, p_k. Počet provedení bloku b je zjevně součtem počtu provedení hran, které do něj vstupují, tedy jestliže je blok b proveden n-krát a bloky b_i jsou provedeny n_i-krát, pak platí

$$n = \sum_{i=1}^{k} p_i n_i.$$

Pokud navíc položíme počet provedení vstupního bloku roven 1, dostáváme soustavu lineárních rovnic, jejímž řešením je hledaný profil (samozřejmě, kdybychom chtěli být zcela přesní, je třeba ještě dokázat, že tato soustava má jednoznačné řešení). Časová složitost závisí na tom, jak tuto soustavu budeme řešit. Pokud použijeme Gaussovu eliminaci, dostáváme kubickou časovou složitost, což je v praxi příliš mnoho. Proto se většinou používají algoritmy, které využívají toho, že CFG má speciální strukturu – pokud se nepoužije příkaz skoku `goto`, každý cyklus má právě jeden vstup. V případě, že `goto` použijeme a tuto podmínku porušíme, tyto algoritmy vrátí pouze přibližné řešení, zato však fungují v lineárním čase.

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
			max.	30	224
1.	Josef Pihera	G Strakon	3	20	208,1
2.	Miroslav Klimoš	G Bílovec	1	25	190,0
3.	Pavel Klavík	G Chrudim	3	25	183,5
4.	Jakub Kaplan	GJKTyla	2	21	161,9
5.	Zbyněk Konečný	GKptJaroš	3	24	154,2
6.	Jiří Maršík	GJKTyla	2	20	138,1
7.	Peter Perešíni	GJGTajov	4	11	119,4
8.	Michal Pavelčík	G UBrod	3	13	115,9
9.	Adam Zivner	G UBrod	4	16	113,0
10.	Petr Kratochvíl	G SvětláNS	3	17	106,2
11. – 12.	Lukáš Lánský	GJKTyla	2	16	98,2
	Petr Onderka	G VKlobou	3	16	98,2
13.	Roman Smrž	GOhradní	2	12	88,1
14.	Jan Kohout	G Roudnice	3	15	82,2
15.	Tomáš Herceg	G Třebíč	3	14	77,3
16.	Michal Vaner	G Turnov	4	10	72,7
17.	Michal Čudrnák	G Holešov	4	8	64,1
18.	Tomáš Zámečník	GJKeplera	3	11	62,5
19.	Drahoslav Viktorýn	G UBrod	3	10	61,6
20.	Kristýna Krejčová	G Tišnov	3	9	56,6
21.	Richard Jedlička	G Vlašim	2	8	55,0
22.	Ondřej Bouda	GKptJaroš	3	8	48,1
23.	Daniel Marek	GZborov	4	6	47,0
24.	Ondřej Bílka	G Zlín	4	16	44,1
25.	Josef Špak	GJírovco	3	7	41,5
26.	Radim Pechal	SPŠ Rožnov	3	6	39,9
27.	Jan Hrnčíř	GFXŠaldy	4	9	39,3
28.	Jan Dvořák	GZborov	3	5	38,3
29.	Pavel Veselý	G Strakon	1	7	37,4
30.	Kateřina Böhmová	G Rožnov	4	4	36,2
31.	Cyril Hrubiš	G Bílovec	4	6	36,0
32.	Jiří Machálek	G Holešov	4	6	35,5
33.	Roman Říha	G Prachat	2	5	31,1
34.	Tereza Klimošová	G Lanškr	4	4	31,0
35.	Radim Cajzl	G NMnMor	0	7	30,4
36.	Jakub Pavlík jn.	G Kladno	3	5	28,1

Pořadí řešitelů

37.	Vojtěch Molda	G Vsetín	4	5	26,9
38.	Martin Kahoun	GJNerudy	3	6	26,1
39.	Ondrej Mikuláš	G Lučenec	3	5	25,7
40.	Petr Trňák	G UHradi	3	5	24,3
41.	Martin Majer	SPŠÚžlabin	1	4	23,9
42.	Tomáš Sýkora	G VKlobou	2	5	22,8
43.	Ján Mikuláš	G Lučenec	4	5	21,7
44.	Adam Ráž	GBudějo	3	5	21,4
45.	Jan Krajdl	SPŠÚžlabin	1	3	17,3
46.	Jiří Cabal	SPŠ DvKrál	3	2	15,1
47.	Rudolf Rosa	G Kladno	3	3	15,0
48.	Matej Kollár	G PBystric	4	3	14,8
49.	Jiří Lekeš	G UBrod	2	2	14,4
50.	Miroslav Jančařík	G UBrod	2	3	13,5
51.	Lukáš Moravec	GSRandyJN	2	3	12,7
52.	David Škorvaga	G Kralupy	3	2	12,4
53.	Tomáš Ehrlich	G Holešov	3	2	9,8
54.	Jakub Balhar	GJNerudy	3	2	8,9
55.	Marián Bazálik	G Košice	4	2	8,3
56.	Jan Musílek	G NBydžov	2	2	7,3
57.	Jan Tichý	GDašická	1	2	6,6
58.	Jiří Václavík	G Dobříš	4	2	6,5
59.	Vladimír Munzar	SPŠ Rožnov	1	1	5,8
60. – 62.	Martin Fojtík	GSRandyJN	2	1	4,7
	Dušan Rychnovský	G Hranice	2	1	4,7
	Radek Svoboda	G Roudnice	3	1	4,7
63.	Robert Brunetto	SPŠMasaryk	2	1	4,3
64.	Jiří Keresteš	ZŠKostelní	0	2	4,2
65.	Jakub Loucký	G Písek	3	1	3,5

Obsah

Milan Straka a kolektiv

Korespondenční seminář z programování
XVIII. ročník

Autoři a opravující úloh:
Jan Bulánek, Pavel Čížek, Zdeněk Dvořák, Zbyněk Falt,
Tomáš Gavenčiak, Jana Kravalová, Martin Kruliš,
Pavel Machek, Martin Mareš, David Matoušek,
Milan Straka, Petr Škoda, Tomáš Valla

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 185. publikaci.

TEX-ová makra pro sazbu ročenky vytvořil Martin Mareš.
S jejich pomocí ročenku vysázel Milan Straka.
Ilustrace (včetně té na obálce) vytvořil Martin Kruliš.
Sazba byla provedena písmem Computer Modern v progamu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 158 stran
Náklad 300 výtisků
Praha 2006

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji!

ISBN 80-86732-91-6

ISBN 80-86732-91-6