Korespondenční Seminář

z Programování

Milí chlapci a děvčata, jako každý rok je tu
opět **Korespondenční Seminář z Programování.**
Že jste o něm ještě neslyšeli? V tom případě
si zkuste odpovědět na následující kvíz:

- Zajímáš se o počítače?
- Rád soutěžíš?
- Chceš se dozvědět něco nového?
- Chceš poznat nové lidi?
- Chceš užitečně vyplnit volný čas?
- Hledáš výzvu pro svoji hlavu?

Odpověděl sis alespoň jednou „ano“? Pak hledáme
právě Tebe. Do KSP se může zapojit každý.
Máš-li chuť, otoč list …

Základní fakta o KSP:

- KSP je celostátní a celoroční soutěž v programování pro studenty středních i základních škol.
- Jeden ročník je rozdělen na 5 sérií.
- V každé sérii účastníci obdrží zadání úloh (buď poštou nebo po Internetu, jak chtějí).
- Úlohy vyřeší v teple domácího krbu a svá řešení nám zašlou.
- My jim vrátíme opravené úlohy společně se vzorovými řešeními, zpravidla se zadáním další série.
- Na vyřešení jedné série je několik týdnů času.
- Série obsahuje šest až sedm programátorských úložek a každému řešiteli jsou započítány čtyři nejlépe vyřešené.
- Úlohy jsou čistě algoritmického rázu. Rychlejší a lépe popsané algoritmy mají přednost před programy hýřícími barvami.
- Každá úloha je bodována, body ze všech úloh ze všech sérií se sčítají a tvoří celkové hodnocení.
- Pro nejlepší řešitele pořádáme na začátku dalšího školního roku (obvykle v říjnu) **soustředění,** na kterém se nejen dozví užitečné věci z programování, ale také si protáhnou tělo i mysl při ryze neinformatických činnostech.
- Pro řešitele začínající naopak pořádáme jarní soustředění, kde se lze naučit základy programování.
- Tři nejlepší řešitelé každého ročníku budou odměněni titulem **Král KSP,** ke kterému se váže mnoho dalších výsad.
- Další informace a **přihlášku** nalezneš na *http://ksp.mff.cuni.cz/*, dotazy (ale ne řešení úloh) můžeš posílat na *ksp@mff.cuni.cz*.
- Hodně štěstí!

Milí řešitelé a řešitelky!

Držíte v ruce první leták 23. ročníku KSP. Každá série i v letošním roce obsahuje 7 úloh, z toho 4 nejlépe vyřešené se započítávají do celkového bodového hodnocení.

Termín odevzdání první série je stanoven na pondělí 18. října v 8:00 SELČ, což znamená, že papírové řešení byste měli podat na poštu do středy 13. října.

Nejlepší tři řešitelé celého ročníku budou odměněni titulem **Král KSP,** o kterém se můžete dočíst na konci zadání.

Řešení přijímáme elektronicky na stránce *https://ksp.mff.cuni.cz/submit/*. Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš šifrovací certifikát – zde je jeho SHA1 hash: `7F:53:E7:00:60:F2:24:93:8F:52:51:EC:1E:A8:34:54:86:69:32:7D`.

Také nám řešení můžete poslat klasickou poštou na adresu

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

Před tím ale vyplňte přihlášku (a to i tehdy, když jste se KSPčka účastnili loni) na *http://ksp.mff.cuni.cz/,* kde najdete i další informace o tom, jak KSP funguje. Na případné dotazy vám rádi odpovíme na adrese *ksp@mff.cuni.cz.*

První série třiadvacátého ročníku KSP

Milý deníčku,

oproti minulému týdnu, kdy jsem slavila osmé narozeniny a dostala hříbě, byl tenhle strašný. Matka mi sehnala nového učitele matematiky, který je ještě ošklivější než ten předchozí. Navíc se jí prý mám věnovat o dvě hodiny týdně víc.

Už takhle skoro nedělám nic jiného. Moc ráda bych teď, když umím flétnu, začala hrát na klavír a slečna Jane, naše komorná, mi nabídla, že mě to naučí, ale matce se to nelíbilo, že prý mi sežene pořádného učitele, ale že mám na hudbu dost času.

Včera se mi povedlo utéct z domu a zašít se s Charliem do dílny. Měl tam schovanou žábu. Povídal mi, že od svého táty, našeho zahradníka, slyšel, že můj táta válčí v Řecku. Přišlo mi to divný, ale prý to ví fakt určitě.

Zítra mě čekají nerovnice. Netěším se.

23-1-1 Básníkův deník — 9 bodů

Zahradníkův syn měl pravdu: otec naší hrdinky, slavný romantický básník lord Byron, se na začátku roku 1824 skutečně účastnil Řecké války za nezávislost na Otomanské říši. Jako záminku (či, chcete-li, motivaci) pro první úlohu nového ročníku našeho semináře si vezmeme jeho deník, ve kterém popisoval své výlety a cesty Řeckem předtím, než 19. dubna umřel.

Řekněme, že si každý večer před spaním zaznamenal, o kolik metrů níže či výše za uběhlý den sestoupil či vystoupil. Váš program dostane seznam těchto údajů (jedno *celé číslo* za každý den) na vstupu a vaším úkolem je vypsat dvě čísla:

- V jaké výšce se lord Byron na konci každého dne nacházel nejčastěji? Je-li více možných odpovědí, vypište libovolnou z nich.
- V jaké největší nadmořské výšce jeho lordstvo přenocovalo?

Předpokládejte, že putování začalo u mořské hladiny (této výšce přiřadíme, jak je zvykem, nulu), a nikdy pod mořskou hladinu nesestoupilo.

Příklad vstupu: `103 20 -20 50 -82`
Odpovídající výstup: `nejčastěji: 103, nejvýše: 153`

Vážený pane de Morgane,

jsem vám velice vděčna, že jste si i na cestách našel čas a sepsal mi obsáhlý dopis plný úloh, jejichž řešením se poslední dva týdny těším. Věřte prosím, že matematiku studuji stejně pilně jako pod Vaším laskavým dozorem a že až na den mých patnáctých narozenin nebyla má pozornost odvedena od počtů ničím zásadním.

V příloze vám zasílám některé výsledky a upřímně doufám, že v nich nejsou ony trapné numerické chyby, které mne poslední dobou tak pronásledují.

S netrpělivostí v srdci vyhlížím váš návrat,

Ada

23-1-2 Jedna geometrická — 10 bodů

Z množství fiktivních úloh od pana de Morgana vybíráme dvě takové, které dává smysl zpracovávat moderní výpočetní technikou.

Mějme v kartézské soustavě souřadnic zadány body, jejichž souřadnice obecně nemusí být celočíselné. Ptáme se na nejmenší kruh, který je všechny obsahuje – tedy kde je jeho střed a jaký má poloměr.

Příklad vstupu: `[0;0] [0;1] [1;0] [2;2]`
Odpovídající výstup: `S=[1;1], r=1.4142135`

23-1-3 Jedna maticová — 11 bodů

Na vstupu dostaneme matici, tj. dvojrozměrné pole celých čísel, která má navíc tu zvláštní vlastnost, že jsou čísla v každém jejím řádku a sloupci ostře rostoucí. Potřebovali bychom rychle zjistit, zdali v ní neexistuje nějaké políčko v i-tém řádku a j-tém sloupci, které by mělo hodnotu přesně $i + j$.

Pokud hledaných políček existuje víc, můžete vypsat libovolné z nich. Pár bodíků navíc si můžete vysloužit, pokud vymyslíte, jak rychle spočítat, kolik takových políček je.

Při zvažování časové složitosti nepočítejte dobu načítání: představujte si, že už máte matici v paměti. Zkuste zdůvodnit, proč nelze dosáhnout rychlejšího řešení.

Příklad vstupu:

```
-3  1  4
 4  5  6
 7  9 11
```

Odpovídající výstup: `1. řádek, 3. sloupec`

Příklad vstupu:

```
 3  4  5
 4  5  6
 5  6  7
```

Odpovídající výstup: `žádné takové políčko není`

23-1-4 Ale co trapné numerické chyby? 10 bodů

Ada však evidentně s matematikou pomoct nepotřebuje: trápí ji trapné numerické chyby. Mohli byste jí pomoci s tím? Třeba… s dělením?

Na vstupu dostanete dělence a dělitele a váš program by měl na výstup vypsat celý desetinný rozvoj podílu. Pokud je rozvoj nekonečný, vyznačte nejkratší možnou periodu.

Příklad vstupu: `1 2`
Odpovídající výstup: `0.5`

Jiný příklad: `1 3`
Výstup: `0.(3)`

Ještě jeden příklad: `143 56`
Výstup: `2.553(571428)`

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx: *http://codex2.ms.mff.cuni.cz/ksp*. Pokud jste zatím žádnou praktickou úlohu neřešili, přečtěte si stručný úvod: *http://ksp.mff.cuni.cz/about/codex.html*.

Vážený pane Babbagei,

mrzí mne, že Vás zdržuji od důležité práce, ale naše setkání na zahradní slavnosti u pana Dickense mi nedá spát. Byla jsem Vám představena paní Somervillovou jako Ada, plným jménem jsem Augusta Byronová.

Je mi teprve sedmnáct let, ale má matka mi zajistila dobré matematické vzdělání a Vaše myšlenka Diferenciálního stroje, který by zautomatizoval výpočty matematických a technických tabulek, mi přijde úchvatná a podivně samozřejmá.

V příloze Vám zasílám seznam dokumentů a knih, o kterých od paní Somervillové vím, že je vlastníte, a o kterých doufám, že byste mi je, samozřejmě za patřičnou protislužbu, mohl zapůjčit.

Velice ráda bych se s Vámi též opět setkala osobně.

Zdraví

Augusta „Ada“ Byronová

23-1-5 Adina knihovna 10 bodů

Velká zásilka knih od pana Babbageho co nevidět přijde, Ada si proto musí udělat pořádek v knihovně a uvolnit pro ni zvláštní police. Trvá jí to samozřejmě předlouho, každé zařazení publikace do správné poličky se neobejde bez zběžného projití až pročtení jejího obsahu.

Jakmile to Ada dodělá, napadne ji následující logická hříčka, nad kterou stráví zbytek večera:

Mějme řadu N knih správně seřazenou podle svého názvu (který má každá kniha různý). Nyní ji přeskládáme tak, aby každá knížka byla na pozici právě o K větší nebo menší, než byla po seřazení. Pro jaká K v závislosti na N jde něco takového udělat a kolika způsoby?

(adresováno Charlesi Babbageovi)

Drahý příteli,

v příloze vám zasílám finální text překladu Menabreaova textu společně s poznámkami, na kterých jsme se dohodli. Věnujte prosím pozornost znění popisu výpočtu Bernoulliových čísel, opět jsem do toho hrábla, doufám, že nyní už bez újmy na matematické přesnosti.

Mým dětem se daří dobře, děkuji za optání. Byron už dorůstá do věku, kdy se musím rozhodnout, zdali ho hodlám obtěžovat matematikou, nebo jeho vzdělání nechám obvyklejší humanitní tvar. Je to těžké rozhodování a budu potřebovat Vaši radu.

Ráda bych Vás tu v Ockhamu zase někdy viděla. S mou finanční podporou samozřejmě můžete nadále počítat. Oba víme, kde by peníze, které bych Vám upřela, skončily.

Vaše Ada

23-1-6 Babbageova cesta 10 bodů

Z posledního dopisu vidíme, že Babbage nemá peněz nazbyt. Samozřejmě se chce za paní Adou, hraběnkou z Lovelace, dostat v co nejkratším čase, ale ze všech možností, ze všech tras, které mu dosažení tohoto nejkratšího času nabízejí, potřebuje vybrat takovou, která je nejlevnější.

Na vstupu dostaneme dopravní mapu Anglie zadanou jako seznam spojení (železničních tras), které vedou mezi různými městy (vždy oběma směry). Pro jednoduchost předpokládejme, že použití takového spojení trvá jednotkový čas. U každého spojení je na vstupu také napsáno přirozené číslo, kolik pana Babbage jeho použití stojí. Také dostanete napsáno, ve kterém městě Babbage začíná a kde bydlí Ada.

Na výstupu vypište posloupnost spojení (neboli *cestu*), která ze všech cest z Babbageho stanoviště do Adina bydliště, které používají nejméně spojení, stojí nejméně peněz, tj. součet ohodnocení všech spojení na cestě je nejmenší. Pokud je takových cest víc, vypište libovolnou z nich.

Ada zemřela v 36 letech na rakovinu dělohy. Nechala za sebou tři děti, které už jsou samozřejmě také dávno mrtvé, a první počítačový algoritmus, který by prý na Analytickém stroji, Babaggeově vylepšené verzi stroje Diferenciálního, skutečně běžel a generoval Bernoulliho čísla.

Vedou se neplodné diskuse o tom, nakolik byl program jejím dílem a nakolik šlo o práci Babbage, který toliko obdivoval její slohové schopnosti. Ada sice jeho snažení podporovala značnými částkami, stejně nemalé peníze ale prohýřila v sázkách na koně. Ke svým dětem prý měla ambivalentní vztah – Diferenciální stroj však považovala za „přítele“.

Sterling s Gibsonem kolem její postavy sepsali steampunkový román Mašina zázraků. Po Adě se jmenuje relativně použitelný programovací jazyk. Ada je skvělá. Ada je krásná. Ada je děvče všech matfyzáků bez děvčat.

23-1-7 Regulární výrazy 14 bodů

V letošním ročníku si budeme povídat o regulárních výrazech. Už se vám jistě někdy stalo, že jste potřebovali nějak zběsile přejmenovat soubory, nahradit v textu všechny výskyty jména Markéta za jméno Dominika nebo jednoduše najít všechna slova začínající velkým písmenem. Dokud jsem nevěděl o regulárních výrazech, dělal jsem veškerou takovou práci ručně, což není od deseti stran nic příjemného, nehledě na to, že lidské oko často nějaký výskyt opomene…

K nalezení všech pádů jména Markéta v jednotném čísle by tedy například sloužil výraz `Markét(a|y|ě|u|o|ou)`.

Regulární výraz umožňuje definovat *množinu řetězců*, které mu vyhovují. Když pak pomocí něj vyhledáváte, najdete všechny řetězce z té množiny.

Jak takový regulární výraz vypadá? Je to řetězec poskládaný z obyčejných a speciálních znaků. Typickým obyčejným znakem je písmeno: výrazu `ab` vyhovuje jen řetezec `ab`. Obyčejný znak je obecně „všechno ostatní“, tedy všechny znaky, o kterých si neřekneme nic zvláštního.

První důležitá skupina speciálních znaků, kterou si uvedeme, jsou znaky, které určují, kolikrát se předchozí znak bude opakovat:

`*`	libovolný počet ($0 \ldots \infty$)
`+`	alespoň jednou ($1 \ldots \infty$)
`?`	jednou nebo vůbec
`{`n`}`	právě n-krát, kdy n je přirozené číslo
`{`a`,`b`}`	a- až b-krát, přičemž musí platit, že $a \leq b < K$, kde K je číslo závislé na systému, obvykle 256, ale může být i víc. Pravá mez (b) může být i vynechaná. Pak je považována pravá mez za nekonečnou, nebo se to chová, jako by tam bylo $K - 1$: to také záleží na systému.

Takže výrazu `ab*` vyhovuje řetězec, který začíná `a` a následuje libovolné množství `b`; výrazu `a+b?c+` vyhovuje řetězec, který začíná alespoň jedním `a`, pak v něm možná je `b` a končí alespoň jedním `c`. Výraz `d{3,5}` je ekvivalentní s výrazem `dddd?d?`, neboť oběma vyhovují právě řetězce obsahující 3, 4, nebo 5 `d`.

Opakovat jen jeden konkrétní znak by však bylo hloupé. Můžeme tedy nějaký kus výrazu uzávorkovat do kulatých závorek a k němu celému se pak tenhle *opakovací operátor* bude vztahovat: výrazu `(aa)*` vyhovuje řetězec obsahující sudý počet `a` (ve stejném smyslu se dá použít `a{2}*`). Výrazu `b?(ab)*a?` vyhovuje řetězec, ve kterém se `a` a `b` pravidelně střídají. Výrazu `b?(a+b)*a*` pak vyhovuje řetězec složený z `a` a `b`, ve kterém se nikde nevyskytuje dvojice `b` vedle sebe.

Pomocí závorek můžeme také dát více variant za použití znaku `|` – výrazu `(bagr|kombajn)` vyhovuje jak řetězec `kombajn`, tak řetězec `bagr` (a nic jiného). Pozor, pokud napíšete `(a|b){2}`, budou výrazu vyhovovat řetězce `aa`, `bb`, ale i `ab` a `ba`. Opakovací operátory musíme brát jen jako zkratku za nakopírování toho, co opakují, vedle sebe do výrazu.

Úkol 1 [2b]: Napište jiný výraz, kterému budou vyhovovat přesně stejné řetězce jako výrazu `b?(a+b)*`.

Úkol 2 [2b]: Určete, které všechny řetězce vyhovují výrazu `((ab?)+|(ba?)+)*`, a případně nalezněte kratší ekvivalentní výraz.

Další trik, který si v tomto díle ukážeme, jsou hranaté závorky. Do nich uvedeme množinu znaků, které se na tomto místě mohou objevit. Tedy výrazu `[aZ!]*` vyhovují řetězce jakékoli délky, které jsou složené pouze ze znaků `a`, `Z` a `!`. Do závorek je možno uvést i rozsah: Výrazu `[a-z]` vyhovuje jakékoli malé písmeno. Také je dovnitř možno uvést *třídu znaků*: třeba `[[:digit:]]` jsou všechny číslice – standardní je následující dvanáctice:

`alnum`	písmeno nebo číslice
`alpha`	písmeno
`blank`	prázdný znak (obvykle mezera nebo tabulátor)
`cntrl`	řídící znak (znaky s ASCII kódem menším než 32 a ještě pár dalších)
`digit`	číslice (0–9)
`graph`	tisknutelný znak kromě mezery (písmena, čísla, interpunkce, ...)
`lower`	malé písmeno
`print`	tisknutelný znak včetně mezery
`punct`	tisknutelný znak mimo písmena, čísla a mezeru
`space`	bílý znak (mezera, nový řádek, tabulátor, ...)
`upper`	velké písmeno
`xdigit`	hexadecimální číslice (`[0-9a-fA-F]`)

Třídy znaků je možno kombinovat:
Výrazu `[[:digit:][:lower:]XYZ.]` vyhovují všechny číslice, malá písmena, `X`, `Y`, `Z` a tečka. Třídu můžeme také znegovat, uvedeme-li před výčet znaků stříšku `^`: výrazu `[^abc]` vyhovují všechny znaky kromě `a`, `b`, `c`.

Takovou perličkou je pak výraz, kterému vyhovují znaky `]`, `^` a `-`. Pomlčka totiž musí být na začátku nebo na konci, jinak signalizuje výčet; stříška nesmí být na začátku, jinak signalizuje negaci, a `]` musí být na začátku, jinak signalizuje konec závorky: `[]^-]` je správné řešení problému.

Může se hodit ještě jedna zkratka: tečka `.` symbolizuje *jakýkoli znak*. Oblíbený výraz `.*` pak symbolizuje *libovolný řetězec*.

Úkol 3 [4b]: Napište výraz, kterému budou vyhovovat právě desítkové zápisy čísel dělitelných osmi (takže 0, 8, 256, 344 vyhovují, ale 42 ani 37 ne).

Úkol 4 [2b]: Určete, které řetězce vyhovují výrazu `(00)*[01](11)*`.

Úkol 5 [4b]: Tomuto výrazu měly původně vyhovovat všechny řetězce, které mají sudý počet nul i sudý počet jedniček (a neobsahují jiné znaky). Nalezněte protipříklad a zkuste výraz opravit tak, aby dělal to, co má:
`(00|11)*(0(11|00)*1(11|00)*){2}`

Tip: Existují dva typy protipříkladů na regulární výrazy: *false positive* je řetězec, který vyhovuje, i když by neměl; *false negative* je řetězec, který nevyhovuje, ale měl by. Druhý z nich je obvykle závažnější, první z nich se špatně hledá, zvláště v rozsáhlejších výrazech.

Pokud chcete použít jakýkoli speciální znak v jeho obyčejném významu, předřaďte před něj `\` – výrazu `\.\*\\` vyhovuje tečka následovaná hvězdičkou a pak zpětným lomítkem.

Ještě by se hodila poznámka, že tohle všechno zde vyložené jsou *POSIX extended regular expressions*, neboli rozšířené regulární výrazy. Existuje totiž mnoho dalších dialektů regulárních výrazů, něco víc si o nich povíme příště.

Mnoho programů vyhledává v textu po řádkách, typicky `grep` vypisuje všechny řádky, na nichž našel vyhovující řetězec. Existují tedy speciální znaky pro *přišpendlení* řetězce na začátek (`^`) nebo konec (`$`) řádku, takže výrazu `^...$` vyhovují právě tříznakové řádky a výrazu `^a` vyhovují všechna `a` na začátku řádku. Tyto znaky mohou být použity jen na úplném začátku, resp. konci výrazu, kdekoli jinde jsou použity chybně. Takže výraz `(a|^b$|c)` je chybný...

Nakonec, kde se vlastně s regexy (což je zkratka z *regular expressions*) setkáme? S jejich pomocí vyhledávají slavné editory Vim i Emacs, na Windowsech třeba PSPad. Denním chlebem jsou pro programátora v Perlu a taktéž program `grep` a další využívají regexy ke svojí běžné činnosti. Konkrétně pak `egrep` používá přesně ten formát regexů, kterým jsme se tu zabývali.

Regexy můžete používat i v drtivé většině programovacích jazyků od C přes C# až k Perlu a Pythonu – někde jsou přímo součástí jazyka, jindy k dispozici jako knihovna…

Zbrusu nové značení

„Jedna úloha jak druhá… Achjo, to abych si hodil kostkou, co zkusím vyřešit.“

Koukáš na zadání a samou radostí nevíš, co začít řešit? Nepotkal(a) jsi ještě žádnou úlohu podobného rázu, tudíž nemáš tušení, co lze řešit snadno a u čeho by ses pravděpodobně zasekl(a)? Potom se Ti budou hodit následující značky u úloh:

Takto označenou úlohu považujeme za řešitelnou i pro začátečníky, zkušení řešitelé ji jistě dají levou zadní. Bývá za ní však oproti ostatním úlohám o bod méně.

Aby si i pokročilí přišli na své, zařazujeme do zadání někdy těžkou úlohu, která se může stát leckomu noční můrou. Odvážlivci, kdož ji chtějí pokořit, musí být připraveni na zákeřné výpady, nesmí se moc rychle vrhnout do útoku, ale v klidu promyslet strategii, a hlavně by měli mít dostatek pořádného vybavení. Odměnou za vítězství bude poklad v podobě bodů navíc.

Této úloze říkáme *praktická*, jelikož není potřeba popsat algoritmus, jen ho naprogramovat a odevzdat přes internet. Bližší informace naleznete přímo v jejím zadání.

V každém ročníku KSP rozebíráme na pokračování nějaké zajímavé informatické téma do hloubky. Poslední úloha série je většinou pokračováním takového *seriálu* – obsahuje kromě samotného zadání ještě text, ve kterém se mužete dozvědět o tématu něco nového. Jelikož díly seriálu na sebe navazují, vyplatí se mít nastudované i předchozí série.

Protože chápeme, že k „uvaření“ vašich řešení jsou často potřeba znalosti základních algoritmů a datových struktur, obvykle také příkládáme do každé série tzv. *kuchařku,* ze které se můžete takové věci naučit. Často je také v zadání úloha, jíž lze řešit algoritmem z kuchařky. A pozor: další kuchařky najdete na našich stránkách.

Králové KSP

Rozhodli jsme se, že každý rok pasujeme řešitele na prvních třech místech závěrečné výsledkové listiny na Krále KSP. Tento čestný úřad se vyznačuje zvěčněním na stránkách semináře, ikonkou na fóru a dalšími výhodami, zvláště pak těmi během podzimního soustředění:

- Králové budou slavnostně představeni na začátku soustředění.
- První král získá právo vybrat si libovolnou, i cizojazyčnou knihu (sehnatelnou za rozumnou cenu). Tato mu bude slavnostně předána při vyhlašování výsledků na závěr soustředění.
- Další dva králové mají právo podobné, avšak s omezením na knihy české.
- Drobné bonusy v podobě občasných moučníků před některými jídly.
- Oslovování organizátory „Vaše veličenstvo.“
- Čestnou povinnost uvádět přednášky.

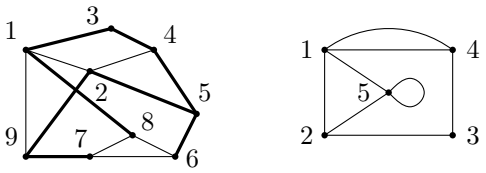
Navíc je pozveme k nám na Matfyz na svátek Tří králů. Za to přeci stojí bojovat, ne?

Recepty z programátorské kuchařky

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* budeme říkat takové posloupnosti vrcholů a hran tvaru $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$, 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, totiž obsahuje nějaký vrchol u dvakrát. Existuje tedy $i < j$ takové, že $u = v_i = v_j$. Pak ale můžeme z našeho sledu 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.

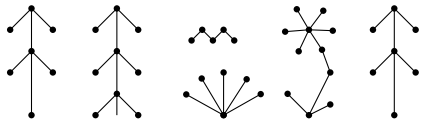
Kružnicí neboli *cyklem* nazýváme cestu délky alespoň 3, ve které oproti definici cesty platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

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

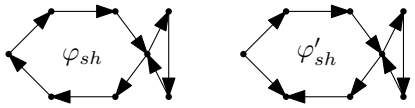
Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale matematici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny.*

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.

Cvičení: Zkuste si dokázat, že stromy jsou právě grafy, které jsou souvislé a mají o jedna méně hran než vrcholů.

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 dvěma 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 $\mathcal{O}(N+M)$ 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

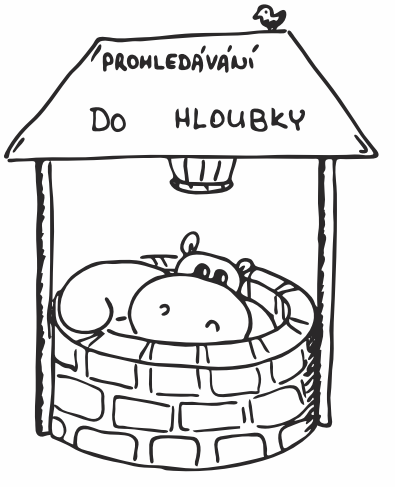
obsahovat spojové seznamy přicházejících a odcházejících půlhran a každá půlhrana bude ukazovat na svou druhou polovici a na vrchol, ze kterého vychází.

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 zatímco prvky přidáváme také na konec, odebíráme je z téhož konce. 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 do zásobníku 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, čili $\mathcal{O}(N+M)$. Paměťová složitost je stejná, protože si tak jako tak musíme hrany a vrcholy pamatovat a zásobník není větší než paměť na vrcholy.

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 $\mathcal{O}(N_i + M_i)$, kde N_i a M_i je počet vrcholů a hran komponenty, vyjde dohromady složitost $\mathcal{O}(N+M)$. Nic nového si ukládat nemusíme, a proto je paměťová složitost stále $\mathcal{O}(N+M)$.

```
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.
```

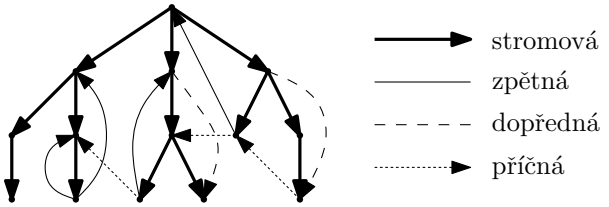
Průběh prohledávání grafu do hloubky můžeme znázornit stromem (říká se mu DFS strom (podle anglického názvu Depth-First Search pro prohledávání do hloubky)). Z počátečního vrcholu w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslovat jako syny vrcholů, ze kterých jsme přišli. Syny 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 $\mathcal{O}(N + M)$. 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 může existenci topologického uspořádání zabránit. Libovolný acyklický graf lze uspořádat následujícím algoritmem:

1. Na začátku máme orientovaný graf G a nastavíme proměnnou p na 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 $\mathcal{O}(N + M)$.

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 $\mathcal{O}(N + M)$.

```
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 $\mathcal{O}(N+M)$. 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*. 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.

Dnešní menu Vám servírovali
Martin Mareš, David Matoušek a Petr Škoda

Často kladené dotazy

„U Elektrona Svatýho, co myslel tímhle?“

„A časovou složitost má kde?“

„Proč by tohle mělo fungovat?“

Takové a mnohé další dotazy si my, organizátoři KSP, klademe při opravování došlých řešení. Bohužel, někdy na ně odpovědi nenajdeme, a proto ze zvědavosti strhneme několik bodů. Sice se mnozí řešitelé časem naučí, co dělat mají a co ne, aby získali co nejvíce bodů, ale chvíli to trvá a některé při tom ztratíme.

Pro ulehčení nováčkům jsme tedy připravili tento návod. Představme si modelovou úložku. Úkolem je spočítat největšího společného dělitele dvou čísel (předstírejte na chvíli, že jste to ještě nikdy neřešili). Na ní si ukážeme postup, jak dojít ke správnému řešení, a také pár tipů, co nedělat.

Napřed je, samozřejmě, potřeba řešení vymyslet. První, co nás pravděpodobně napadne, je zkusit vydělit obě čísla tím menším z nich. Pokud po dělení je nějaký nenulový zbytek, pak to není největší společný dělitel a my zkusíme číslo o 1 menší. A tak dále, dokud nepotkáme takové, které beze zbytku dělí obě. To je samozřejmě společný dělitel a je první, kterého jsme potkali, takže je největší. Takový postup má mnohé výhody – je jednoduchý, zcela očividně vrátí správný výsledek, a navíc máme jistotu, že někdy skončí (zastavíme se určitě nejpozději u jedničky). Ale jde to i rychleji (co znamená „rychleji“, to si povíme za chvíli).

Zapomeneme na rozklad na prvočísla, který je na první pohled příliš komplikovaný, než aby měl šanci na úspěch. Vezmeme dvě zadaná čísla. Pokud se rovnají, pak jsou (obě) největším společným dělitelem. Pokud ne, to větší z nich zmenšíme o to menší a pokračujeme stejně. Navíc pokud bude jedno číslo obrovské a druhé maličké, budeme to maličké odečítat opakovaně, až získáme ... zbytek po dělení většího menším. To by mohlo výpočet ještě zrychlit.

Dobrá, postup máme. Co teď? Nyní je vhodné napsat vlastní program v nějakém jazyce. My organizátoři ho sice nepožadujeme „povinně“, ale pomůže odhalit nedostatky (či nedomyšlené „zrady“) v algoritmu. Například na našem příkladě bychom zjistili, že zatímco odečítací metoda funguje, metoda zbytková se bude pokoušet dělit nulou (alespoň tak, jak jsme ji popsali). V nějakou chvíli již bude v menším čísle uložený výsledek. Při odčítání dojdeme postupně ke stejnému číslu – ale při dělení získáme zbytek 0 jedinou operací. V takovou chvíli je třeba skončit. Navíc nám program pomůže pochopit méně jasné části popisu.

Program by vypadal třeba takto (zapíšeme ho v Pascalu):

```
var x, y: integer;
begin
   read(x, y);
   while (x<>0) and (y<>0) do
      if x>y then x := x mod y
             else y := y mod x;
   writeln(x+y);
end.
```

(všimněte si malého triku: když je na konci jedno z čísel `x`, `y` nulové, tak `x+y` je rovno tomu nenulovému).

Nakonec je třeba vytvořit text řešení. Co by měl obsahovat? Určitě popis algoritmu, a to takový, aby kdokoliv, kdo umí jen trochu programovat, podle něho byl program schopný napsat. Musí z něho jít pochopit, co se děje. Při tomto popisu lze použít nějaký již existující algoritmus jako stavební kámen, například se odkázat na nějakou knížku nebo Programátorské kuchařky z webu KSPčka. Pokud tento popis bude nejasný nebo nejednoznačný, pokusíme se nějakou myšlenku vykoukat z přiloženého programu, avšak už za nedostatečný popis nejspíš pár bodů ztratíte.

Další částí by mělo být nějaké zdůvodnění, proč vlastně program počítá to, co se po něm chce. Určitě nám ještě nevěříte, že popsaná magie se zbytky funguje. My mnohým tvrzením, která nám dojdou, také ne (některému až tak moc, že si dáme práci ho vyvrátit). Co by bylo důkazem v tomto případě? Třeba následující textík (zapsaný opravdu důkladně, jako formální důkaz, obvykle však stačí myšlenka):

Tvrdíme, že v každém kroku algoritmu nahradíme větší z čísel x, y nějakým jiným, ale zachováme přitom všechny společné dělitele dvojice x, y, tím pádem samozřejmě i největšího společného dělitele. A jakmile se algoritmus zastaví (což zajisté učiní), držíme v ruce dvě čísla, z nichž jedno je nula (a ta je beze zbytku dělitelná čímkoliv), a tedy největší číslo, které dělí obě, je to druhé, nenulové.

Zbývá tedy dokázat, že jeden krok algoritmu zachovává všechny společné dělitele. Mějme nějakého společného dělitele d čísel x, y. Navíc předpokládejme, že $x > y$, takže x budeme nahrazovat číslem $z = x \bmod y$ (kdyby $x < y$, tak

jen prohodíme x s y). Co z toho víme:

$$x = x' \cdot d$$
$$y = y' \cdot d$$
$$z = x - t \cdot y$$

pro nějaká x', y', t. Číslo z tedy můžeme upravovat takto: $z = x - ty = x'd - ty'd = (x' - ty')d$. Takže z je také dělitelné číslem d.

Nechť naopak nějaké d dělí dvojici z, y. Pak víme:

$$z = z' \cdot d$$
$$y = y' \cdot d$$
$$x = z + t \cdot y,$$

takže $x = z'd + ty'd = (z' + ty')d$ je také dělitelné číslem d. Zjistili jsme tedy, že společní dělitelé dvojic x, y a z, y jsou titíž.

Nakonec je potřeba odhadnout, jak dobrý algoritmus jsme vymysleli. K tomu slouží odhady časové a paměťové složitosti. Co to je? Zajímá nás především, jakým způsobem rostou nároky na čas strávený algoritmem a paměť, kterou při běhu spotřebuje, v závislosti na velikosti vstupu. Například pokud chceme vybrat nejmenší ze zadaných čísel, pak při dvakrát větším vstupu to bude trvat $2\times$ déle, zatímco vypsat 3. číslo (dle pořadí, ne dle velikosti) vstupu nám bude trvat vždycky stejně dlouho – zbytek vstupu nepotřebujeme ani přečíst.

K tomuto používáme tzv. asymptotické odhady – najdeme funkci, která roste řádově alespoň tak rychle, jako naše nároky. Pro jednoduchost se zanedbají všechny konstanty a menší funkce. V případě výběru nejmenšího budeme psát, že čas je $\mathcal{O}(n)$ pro vstup velikosti n – roste tolikrát, kolikrát vyroste vstup, zatímco paměť bude vždy stejná – $\mathcal{O}(1)$, tj. konstantní (ano, mohli bychom psát třeba $\mathcal{O}(4)$, ale 1 je nejjednodušší konstantní funkce).

Do $\mathcal{O}$ se píše tzv. horní odhad, tedy „nikdy to nemůže být horší“ (ale když budeme mít štěstí, mohlo by být i menší). Existují i jiné, ale když se chceme chlubit kvalitou našeho řešení, těžko budeme chtít tvrdit: „určitě to nikdy nepoběží rychleji než n^{99}“.

Paměťová složitost našeho algoritmu je zcela očividně konstantní – máme jen dvě proměnné na čísla, v nich provádíme veškeré operace.

Časová bude chtít trošku odhadovat. Jednak, co je velikostí vstupu? Tou bude třeba součet velikostí obou čísel, tedy $n = x + y$ (délka výpočtu totiž nezávisí na počtu čísel na vstupu – tam je jich vždy stejně – ale na jejich hodnotách). V každém kroku se jedno z čísel sníží alespoň o 1 a nikdy se nedostaneme do záporných čísel. Takže bychom mohli klidně psát, že časová složitost je $\mathcal{O}(n)$ – určitě náš program nepoběží déle.

To je sice pravda, ale moc jsme se nevytáhli – stejnou časovou složitost měl i původní algoritmus se zkoušením všech potenciálních dělitelů. Tak co teď? Vymyslet lepší algoritmus? Ne, my na to půjdeme šalamounsky – vymyslíme lepší důkaz.

Opět předpokládejme, že přecházíme od dvojice x, y ke dvojici z, y, kde $z = x \bmod y$. Dokážeme, že $z \leq x/2$, takže každým krokem algoritmu se aspoň jedno z čísel zmenší aspoň dvakrát. Přitom kroků, kdy se dvakrát zmenšilo původní x, může být celkem nejvýš $\lfloor \log_2 x \rfloor$, a analogicky pro y. Proto je celková časová složitost $\mathcal{O}(\log_2 x + \log_2 y) = \mathcal{O}(\log n)$.

A proč je $z \leq x/2$? Rozebereme dvě možnosti: buďto je $y \leq x/2$, ale pak stačí využít toho, že zbytek po dělení je vždy menší než dělitel, tedy $z < y \leq x/2$. A nebo je $y > x/2$, ale pak $z = x - y \leq x - x/2 = x/2$.

Několik špatných pokusů

Minulá část obsahovala popis, jak vypadá správné řešení. Nyní zmíníme několik chyb, se kterými se celkem pravidelně při opravování setkáváme.

Jednou (a asi nejvážnější) z nich je, když nám přijde pouze zdrojový kód, který je občas (ale sporadicky) komentovaný a není k tomu žádný popis. Popis má být hlavní částí řešení, zdrojový kód pouze doplňkem.

Opačný extrém je příliš podrobné (a komplikované) vyprávění či slohová práce. Opravdu neplatí, že čím delší text, tím více bodů. Úlohy v KSP jsou dělané tak, aby se daly jednoduše popsat na stránku nebo dvě. Řešení o 20 stranách je tak dlouhé, že v něm prostě něco špatně být musí.

Další, celkem běžný, problém je špatně pochopitelný popis. Zkuste si text po sobě přečíst – s vědomím, že člověk, který ho bude číst, možná vaše řešení vůbec nezná a nic o něm neví. Nejlépe s odstupem několika hodin či dní. Do této oblasti patří i pravopis (někdy špatně umístěná (nebo chybějící) čárka ve větě může úplně změnit význam) a v případě psaní ručně i čitelnost rukopisu (za to, co nepřečteme, body nedáme).

A, samozřejmě, plný počet bodů nedáváme ani za řešení, které nefunguje.

Co dělat když...

Mnoho lidí KSP řešit nezačne, přestože by je třeba i bavilo. Většinou proto, že narazí na nějaký problém, který ale obvykle není tak neřešitelný, jak vypadá.

Pokud vám některá úloha přijde příliš těžká, nezoufejte. Snažíme se dávat i „šťavnaté“ úložky pro pokročilejší řešitele – ne však všechny, některé jsou lehké (od tohoto ročníku jsme zavedli značení, u některých je značka, že si myslíme, že by měla jít snadno vyřešit). Zkuste ty. Časem se vám úložky budou zdát lehčí.

A co v případě, když vás napadne jen pomalé řešení? Je na něm sice jasně vidět, že jsme při zadávání mysleli na něco rychlejšího, ale vy na to ne a ne přijít. Rozhodně napište alespoň to pomalé – zatímco za nefunkční řešení nedáváme skoro nic, za pomalejší řešení dáváme docela dost (tedy, podle toho, jak moc pomalejší je).

Nakonec malý tip. Zkuste začít řešit s předstihem. Velmi pomáhá, když je pár dní času na to, aby pěkné řešení „napadlo“.