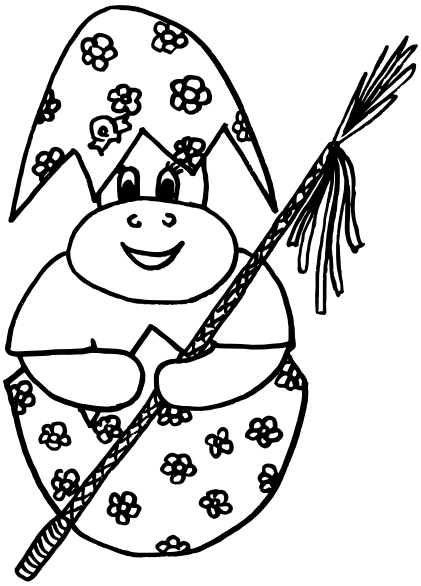
Milí řešitelé a řešitelky!



Je tady finále! Finále! Čtete zadání poslední, páté série 23. ročníku KSP. Připomínáme, že jako tradičně obsahuje 7 úloh, ze kterých se do celkového bodového hodnocení započítávají 4 nejlépe vyřešené.

Tentokrát jsou však úlohy o něco těžší a také bodované o něco mastněji. Asi třiceti nejlepším z vás přijde (snad už) na konci června pozvánka na podzimní soustředění, které by mělo být v druhé polovině září, takže s chutí do toho a soustředko je v kapse!

Zároveň se také rozhodně těsný souboj o čestné funkce tří králů[1] tohoto ročníku. Vysoké bodové hodnocení této série může ještě stále nečekaně zamíchat kartami.

Nezapomeňte, že k vyřešení některých úloh stačí prostudovat vhodné kuchařky.

Termín odevzdání páté série je stanoven na pondělí 30. května v 8:00 SELČ, což znamená, že papírové řešení byste měli podat na poštu do středy 25. května.

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

Na případné dotazy vám rádi odpovíme také na adrese *ksp@mff.cuni.cz* a v diskusním fóru na našem webu.

Pátá série třiadvacátého ročníku KSP

John von Neumann se narodil v Maďarsku (vlastně v Rakousku-Uhersku) v roce 1903 a zemřel v roce 1957 ve Spojených státech. Byl to velmi univerzální vědec – pracoval na matematice čisté i aplikované a významnou měrou přispěl k rozvoji počítačů.

S jeho jménem se jde občas potkat dokonce i ve středoškolské výuce informatiky, kde je často jmenována „von Neumannova architektura“, jejíž hlavní rys tkví v jediné paměti pro program i data.

Něco takového je skutečně dobrý nápad, který stojí za dnešní univerzálností počítačů, ale tehdy to byl velmi pokrokový koncept, neboť první výpočetní stroje se programovaly přepojováním drátů.

Vymyslel také jednoduchý a ne zcela nepoužitelný způsob generování pseudonáhodných čísel, který funguje tak, že předchozí vygenerované náhodné číslo umocníme na druhou a vezmeme z jeho desítkového zápisu prostřední část, kterou ohlásíme jako nové „náhodné“ číslo.

Von Neumann si byl vědom omezení podobných (pseudonáhodných) metod pro generování čísel a známý je jeho výrok „každý, kdo chce generovat náhodné číslice aritmetickými prostředky, je hříšník“. On sám jich však potřeboval neobvykle hodně pro náhodné simulace vodíkové bomby, a tak vzal zavděk takovýmto hříšným způsobem.

Společně se Stanisławem Ulamem je uváděn jako průkopník buněčných automatů, zjednodušených modelů vývoje fyzikálních systémů. Podařilo se mu v jednom takovém modelu vytvořit sebereplikující stroj a napsal na toto téma celou knihu. Navrhoval podobné sebesestavující stroje použít pro rozsáhlé těžební operace, kde by obvyklá tovární výroba potřebných strojů stála lidstvo přílišné úsilí.

Podobné myšlenky posledních několik desítek let budí hrůzu technologických nadšenců, kteří předvídají, že nanotechnologie umožní stavbu tak dobrých sebereplikujících jednotek, že na sebe přemění celou planetu. Ostatně o tom byl jeden z posledních proužků na xkcd.[2]

23-5-1 Boj s nanoboty — 11 bodů

Přeneseme se teď do fiktivního světa knihy Diamantový věk, ve které nejrůznější nanoboti vesele poletují po světě a zkázu světa to nevyvolá.

Existují totiž certifikační organizace, které pomocí svých bojových nanobotů vynucují, aby měl každý stroj, který se chce pohybovat v ovzduší, jisté nanorazítko, které zaručuje jeho bezpečnost.

Pokud se zloduch rozhodne, že zaplaví svět necertifikovanými nanoboty, nastane „tonerová válka“ – nanoboti se do sebe pustí, lidé z jejich zbytků dostanou rakovinu plic a vymřou.

Mějme kousek světa zadaný jako čtvercovou síť. Na prvním řádku dostaneme M; na každém z následujících M řádků dostaneme souřadnice města x_i a y_i, počet obyvatel C_i a čas příchodu padoucha T_i.

Náš hrdina chce předejít tonerové válce a z ní vyplývajícím zdravotním rizikům pro obyvatele daného města. Proto cestuje po mapě a snaží se v tom padouchovi zabránit. Povede se mu to vždy tehdy, když je ve správný čas ve správném městě.

Hrdina začíná na políčku $(0, 0)$ v čase 0 a za jednotku času se může přemístit na sousední políčko, nebo zůstat stát.

Vaším úkolem je najít posloupnost měst, které má navštívit, aby zachránil co nejvíce lidí.

---

[1] `http://ksp.mff.cuni.cz/zaciname/kral.html`

[2] `http://xkcd.com/865/`

Například pro vstup

```
4
1 6 1000 18
2 7 300 16
3 3 100 11
6 5 500 11
```

program odpoví `4 1` (zachrání 1500 lidí).

Podílel se na konstrukci atomové bomby. Na rozdíl od většiny ostatních významných vědců projektu Manhattan se dokonce zapojil do navazujícího programu pro vývoj bomby vodíkové.

Označoval své názory za „militantnější, než je obvyklé“ a prosazoval kupříkladu preventivní jaderný úder na Sovětský svaz předtím, než si obstará vlastní jaderné zbraně.

Postupně se tak zapojovoval do různých armádních poradních sborů. Přišel na to, že je efektivnější nechat detonovat atomovou nálož vysoko nad zemí, než při dopadu.

Zúčastnil se tedy třeba komise pro výběr japonských měst, nad kterými bude bomba použita, aby pomohl spočítat případné japonské ztráty.

23-5-2 Zjednodušení situace — 13 bodů

Když je mapa bitevního pole moc nepřehledná, odstraní se méně významné jednotky, popř. se sdruží pod souhrnnou vlaječku. Mohli bychom ale v takové situaci chtít zazoomovat na část bitevního pole tak, aby se nezměnil zobrazený poměr sil.

Na vstupu dostaneme $2N$ pozic vojáků naší strany a $2M$ pozic vojáků nepřítele; $2N, 2M \in [2, 600]$. Pozice budou dvojice nezáporných celých čísel v intervalu $[0, 100\,000]$.

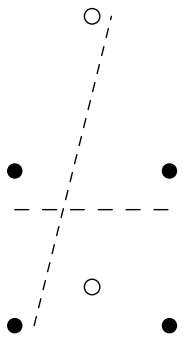
Úkolem bude najít takovou přímku, která rozdělí bojiště na dvě části, kde v každé z části leží právě N našich vojáků a M nepřátel. Přímku vypište jako trojici desetinných čísel (a, b, c) (ve výstupu oddělených mezerou), což jsou koeficienty v rovnici přímky $ax + by + c = 0$.

Můžete předpokládat, že se žádné tři body na vstupu nenacházejí na jedné přímce. Na dělicí přímce nesmí ležet žádný ze zadaných bodů. Pokud přímek bude několik, stačí vypsat libovolnou z nich.

Na vstupu jsou na prvním řádku čísla $2N$ a $2M$, následuje $2N + 2M$ řádků, na každém jsou dvě celá čísla oddělená mezerou – souřadnice vrcholů. Nejdříve jsou uvedeny naše pozice, poté pozice nepřátel.

Příklad vstupu:

```
4 2
0 0
0 4
4 0
4 4
2 1
2 8
```



Dvě možná řešení jsou na obrázku vpravo. `4 -1 -2` (krátké čárky) nebo `0 1 -3` (dlouhé čárky).

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[3] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

V oblasti čisté matematiky pracoval na jejích základech – dnešní způsob množinové definice přirozených čísel pochází právě od něj.

Možná znáte algoritmus Minimax, kterým se dá naučit počítač hrát piškvorky, ale i mnohé další hry. Von Neumann stál u zrodu teorie her a formuloval a dokázal tzv. minimaxovou větu.

Mimimaxová věta využívá principu „své tahy volím jako nejlepší, u tahů nepřítele počítám s nejhorším pro mě“ k tomu, aby u her dvou hráčů (s nulovým součtem)[4] prohlásila, že jeden hráč může nejlépe vyhrát stejnou měrou, jako může druhý hráč nejlépe (tj. nejméně) prohrát.

23-5-3 Hra pro jednoho hráče — 9 bodů

Hry pro dva hráče známe z běžné zkušenosti – piškvorky jsou jejich nejběžnějším zástupcem. Řekneme-li „hra pro jednoho hráče“, napadne nás nejspíš něco jako solitaire.

Zajímavá věc – existuje i pojem „hra pro žádné hráče“ a myslí se tím třeba již zmíněný buněčný automat (například Game of Life Johna Conwaye), kdy je celý průběh jeho vývoje určen počátečním stavem a jde se na něj jen koukat.

Hra, kterou budeme v této úloze uvažovat, počítá s hráčem jedním. Jmenuje se Hanojské věže a spočívá v tom, že dostanete tři tyče A, B a C a na tyči A máte navlečeny disky se zmenšujícím se průměrem (nahoře je nejmenší).

Vaším úkolem je přemístit při zachování pořadí disky z tyče A na tyč C. Jediný tah, který máte povolen, je přemístění svrchního disku z libovolné tyče na jinou, ale pouze tehdy, pokud je disk menší, než svrchní disk na cílové tyči.

Pro tuto známou hru existuje jedinečná vyhrávající strategie, která má $2^n - 1$ tahů a na kterou není těžké přijít. Pro tři disky vypadá kupříkladu tak,[5] že se ten nejmenší přesune na tyč C, střední na tyč B, nejmenší na tyč B, největší na tyč $C\ldots$ a pak už je to jasné.

Vaším úkolem v této úloze není tuto strategii zahrát. Dostanete na vstupu počet disků D a číslo N a vypíšete, jak bude vypadat stav hry s D disky po odehrání N kroků této optimální strategie.

Třeba pro $D = 3$ a $N = 3$ je na tyči A disk největší, na tyči B disk střední a nejmenší a na tyči C není disk žádný.

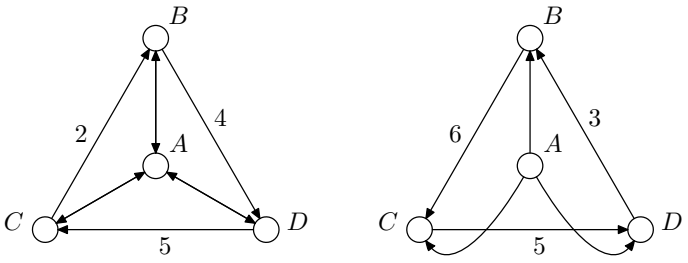
I o von Neumannovi se vypráví řada veselých příhod. Byl prý špatný, ale vášnivý řidič a často si za volantem četl.

23-5-4 Model čtoucího řidiče — 11 bodů

Mějme řidiče, jenž kvůli kvalitní beletrii nemá dost pozornosti na sledování informačních tabulí, které by mu pověděly, kam která odbočka vede. Jezdí po silniční síti, která je zadaná ohodnoceným orientovaným grafem (tj. každá silnice je jednosměrka), kde budeme každý vrchol chápat jako kruhový objezd a dostaneme s ním na vstupu i cyklické pořadí hran.

Protože řidič nesleduje cedule, vyjede vždy na nejbližším možném následujícím výjezdu. Vaším úkolem je najít pro něj orientovanou cestu s nejnižším možným součtem ohodnocení, která prochází každým vrcholem právě jednou.

[3] http://ksp.mff.cuni.cz/zaciname/codex.html
[4] http://cs.wikipedia.org/wiki/Hra_s_nulov%C3%BDm_sou%C4%8Dtem
[5] http://commons.wikimedia.org/wiki/File%3ATower_of_Hanoi.gif



Pro případ vlevo řešení neexistuje, pro případ vpravo je řešením *A–C–D–B*. Hrany bez čísel chápejte jako ohodnocené 1.

Stephen Wolfram o von Neumannovi k stému výročí narození napsal,[6] *že se oddaně držel aktuálních trendů vývoje matematiky a že je překvapivé, že člověk tak inteligentní jako on nikdy nepřišel s žádným skutečně originálním a nečekaným výsledkem, ke kterým měl ve své době opravdu blízko.*

Gödelovy věty či Turingův stroj mohly velmi dobře pocházet i od něj. „Von Neumannova architektura" se sice stala velmi vlivným konceptem, jistě však nešlo o první myšlenku svého druhu, o několik let ho s ní předběhnul kupříkladu Turing.

Vysvětluje si to jednak tím, že mu současné metody šly aplikovat tak hladce, že k odvážným výletům za hranice obvyklého necítil potřebu, druhak tím, že byl konformní člověk, který ctil autority a stejně jako byl zadobře s americkou vládou a armádou, tak nechtěl zpochybňovat velká Hilbertova přesvědčení, proti kterým zmíněné originální výsledky šly.

Podle pamětníků to byl velmi společenský a přátelský člověk. Každý týden uspořádal dva večírky a měl rád děti. Rád vyprávěl obhroublé vtipy. Jeho relativně brzká smrt nebyla tak tragická (dá-li se srovnávat) jako v případě zmíněného Gödela nebo Turinga – měl rakovinu a do poslední chvíle pracoval.

23-5-5 Kuchařková — 12 bodů

Následující problém si pojmenujeme Metr a vaším úkolem je dokázat, že je NP-úplný.

Jistě znáte skládací metry. Mají typicky pět článků po dvaceti centimetrech. Mějme metr nepravidelný, jehož jednotlivé články jsou různě dlouhé. Tyto délky dostaneme v pořadí na vstupu, stejně tak délku pouzdra, do kterého bychom chtěli metr uložit.

Podaří se nám to? Pro články délek 6, 3, 3 a pouzdro délky 6 odpověď jistě zní ANO, pro vstup 6, 3, 4 a stejně dlouhé pouzdro to už ale NEpůjde.

23-5-6 Předposlední — 13 bodů

Dostanete na vstupu orientovaný graf s kladně celočíselně ohodnocenými hranami. Dále tam bude pro každý vrchol dvojice kýžených limitů – minimální součet vstupních hran a maximální součet výstupních hran.

Vaším úkolem je najít nové nezáporné celočíselné ohodnocení každé hrany, které nebude větší než to původní a které bude dohromady se všemi ostatními novými ohodnoceními respektovat dané limity.

23-5-7 Perlím, Perlíš, Perlíme — 15 bodů

Tento text navazuje na předchozí série, některé pasáže nemusí být lehce pochopitelné bez jejich znalosti.

V závěrečném díle seriálu si ukážeme, kam došlo všemožné rozšiřování regexů v Perlu – jazyce určeném zvláště na zpracování textu. Nebudeme probírat kompletní PerlRe, kdybyste se o nich chtěli dozvědět více, přečtěte si přehlednou dokumentaci.[7]

První kroky v Perlu

Nejprve je potřeba Perl nějak získat. Pokud máte Linux, máte jej už pravděpodobně dávno nainstalovaný, jen o něm nevíte. Kdybyste jej náhodou neměli, nainstalujte si jej z balíčkovače, mají jej snad všechny rozumné distribuce.

Na Windows si nainstalujte Cygwin,[8] někteří jej asi máte už od minulé série. V něm by měl Perl jít nainstalovat.

Do Cygwinu se balíčky doinstalovávají spuštěním setupu, člověk si vybere nějaký server a dostane se na seznam balíčků. Tam dá vyhledat, co potřebuje, zaškrtne, odklikne a při příštím spuštění Cygwinu může nové balíčky používat.

Perl je programovací jazyk mnoha zajímavých vlastností, zájemci o více informací si přečtou rozsáhlou dokumentaci,[9] případně se zeptají na fóru.

My si ukážeme jenom minimální program, který můžete používat k testování svých pokusů s PerlRe.

```
#!/usr/bin/perl
use strict; use warnings; # hlídací pes

# cyklus přes všechny řádky vstupu
while (<>) {
    # sem pište své příkazy
    # nahraď všechny bagry za kombajny
    s/bagr/kombajn/g;

    # sem už své příkazy nepište
    # výpis
    print;
}
```

Tenhle program čte vstup po řádkách, pro každý řádek provede zadané příkazy a pak jej vypíše. Každý příkaz končí středníkem; komentáře jsou uvozeny znakem `#` (kriminál), od něj dál se všechno ignoruje.

Jak spustit program? V terminálu dojděte do příslušné složky (pomocí `cd`) a napište `perl minimal.pl`, pokud jste pojmenovali svůj minimální program `minimal.pl` (obvyklá přípona programu v Perlu je `.pl`).

Pak na vstup píšete podobně jako u `sedu`; ukončit vstup je možné stiskem Ctrl+D. Uvedený program by tedy převedl vstup (vlevo) na výstup (vpravo):

```
Žlutý bagr                  Žlutý kombajn
Modrý kombajn               Modrý kombajn
Mám dva bagry.              Mám dva kombajny.
bagrbagrbagr                kombajnkombajnkombajn
```

Nuže, po technickém úvodu přijde konečně něco o regexech (které také budeme v tomto textu označovat jako PerlRe).

[6] http://www.stephenwolfram.com/publications/recent/neumann/
[7] http://perldoc.perl.org/perlre.html
[8] http://www.cygwin.com/
[9] http://perldoc.perl.org/perlintro.html

Základní regexy

Existují dva typy regexových příkazů. Prvním z nich je regex ve stylu `grep`, který zjišťuje, jestli je na vstupním řádku vyhovující podřetězec. Vypadá třeba takhle: `m#cosi#`.

První písmenko je vždycky `m`, druhé písmenko je oddělovač, pak následuje hledaný regex (ve kterém je případně oddělovač nutno escapovat, viz níže) a pak zase oddělovač. Používá se třeba v podmínkách:

```
print "obsahuje bagr\n" if m/bagr/;
```

Další z mnoha využití je při parsování vstupu:

```
# $1 = hodiny, $2 = minuty, $3 = vteřiny
m/(..):(..):(..)/; print "Je $3. vteřina";
```

Takovéhle programy už ale psát nebudeme, zůstaneme jen u samotných regexů.

Druhý typ je nahrazovací – `s#bagr#kombajn#g`. Písmenko `s`, oddělovač, regex, oddělovač, čím nahradit, oddělovač, modifikátory (ty se ostatně mohou objevit i u prvního typu). Příkaz nalezne první výskyt regexu a nahradí jej zadaným řetězcem. Je-li uveden modifikátor `g`, nalezne příkaz všechny nepřekrývající se výskyty a pak je najednou nahradí.

Příkaz `s/rr/tr/g;` tedy převede vstup (vlevo) na výstup (vpravo):

`rr`	`tr`
`rrr`	`trr`
`rrrr`	`trtr`
`rrrrr`	`trtrr`

Jak vypadá oddělovač? Může to být nějaký ze znaků

`!"#$&%)*+,-./:;=>?@\]^‘|}~`,

případně je možné použít konstrukci `s(a)(b)g` (a analogicky s `{}`, `<>` a `[]`).

Jak vypadá výraz? Regexy, které jsme si definovali v prvním díle, by měl Perl přijmout bez řečí. Backreference fungují také stejně, jen jich může být libovolně mnoho. Navíc pokud není reference použita přímo ve výrazu, neodkazuje se na ni `\4`, ale `$4` (nebo třeba `$2078`). Takže například takhle: `s/(.*)(.*)\2\1/$2$1$1$2/`

Rozhlížecí předpoklady

Anglický název je „Look-around assertions“, kdybyste si o tom chtěli přečíst v manuálu.

Taková typická konstrukce je `s/bagr(?=b)/rýč/g`. Té vyhovuje `bagr`, pokud za ním je písmeno `b`. To však již není zahrnuto do onoho řetězce, takže z řetězce `bagrbagrbagry` udělá `rýčrýčbagry`.

Jak vypadají tyto konstrukce formálně? `(?=`*`Výraz`*`)` vyhovuje, pokud na tomto místě začíná kus řetězce, který mu vyhovuje. Pak se Perl **vrátí na jeho začátek** a tváří se, jako by tím kusem řetězce ještě nikdy neprošel. Třeba výrazu `^(?=(..)*$)(...)*$` vyhovují všechny řádky, na kterých je počet znaků dělitelný šesti.

Konstrukce `(?!`*`výraz`*`)` dělá téměř totéž, akorát v negaci. Vyhovuje, pokud se Perlu nepodaří najít žádný vyhovující řetězec začínající na tomto místě. Po tomto zjištění se Perl **vrátí na jeho začátek** a pokračuje, jako by se nechumelilo.

Ještě existují podobné konstrukce v obráceném směru. Předešlé dvě byly „koukni dopředu“, následující budou „koukni zpátky“.

Chcete-li tedy Perlu říct „Tady zastav a zjisti, jestli na tomto místě končí řetězec vyhovující výrazu,“ použijete konstrukci `(?<=`*`výraz`*`)`; pokud chcete negaci, neboli zajistit, aby na tomto místě **nekončil** žádný vyhovující řetězec, napíšete `(?<!`*`výraz`*`)` – například výrazu `.*(?<!bagr)` vyhovují všechny řetězce, které nekončí „bagr“.

Výrazy typu „koukni dozadu“ mají jedno nepříjemné omezení. Musí mít fixní délku. To znamená, že v nich nejen nesmí být kvantifikátory, ale ani třeba `(a|bb)`– dvě varianty různé délky.

Rekurze

Držte si klobouky, jedeme s kopce. Jak vyrobit výraz, kterému by vyhovoval palindromický řádek? Tenhle to je!

`^((.)(?1)\2|.?)$`

Co je na něm tak zajímavého? Ten malinký kousek `(?1)`, který říká něco takového: „Najdi závorku, na kterou by ses normálně odkazoval pomocí `$1`. Ten výraz, který je uvnitř, jako bys spustil na tomto místě...“

Nejlepší bude asi rekurzi předvést na příkladech. Onen první je velice jednoduchý. Celý se skládá ze dvou alternativních větví – v první probíhá rekurzivní krok, druhá funguje jako ukončovací podmínka.

První větev vezme první znak, spustí rekurzi a zbytek pak zase musí být první znak. Rekurze končí ve chvíli, kdy dojde doprostřed testovaného řetězce – zbývá tam prázdný řetězec (v palindromu sudé délky) nebo prostřední znak, který je sám sobě párovým. Přesně o to se stará druhá větev.

Za připomenutí stojí, že závorky se číslují podle pozice jejich **otevírací** závorky zleva doprava.

Ještě si uvedeme jeden rekurzivní příklad – regex, kterému vyhovují všechna správná uzávorkování. Prozkoumejte si jej sami.

`(\((?1)*\))*`

Úkol 1 [9b]: Na řádku jsou vždy dvě různá kladná celá čísla zapsaná ve dvojkové soustavě oddělená mezerou. Napište (jeden) substituční výraz, který na řádku zanechá to větší z nich.

Tedy na vstup (vlevo) dostanete výstup (vpravo):

`1001 1101`	`1101`
`11100 11`	`11100`
`100 1111`	`1111`

Úkol 2 [6b]: Na každém řádku vstupu je pseudoXML. Jsou povolené jen párové tagy bez atributů a prostý text libovolně mezi nimi a okolo. Název tagu (řetězec nenulové délky mezi `<` a `>`) smí být složen pouze z `[[:alnum:]]` znaků. Prostý text nesmí obsahovat znaky `<` a `>`.

Váš (jeden) substituční výraz zkontroluje validitu každého řádku na vstupu, tedy jestli má každý tag příslušný uzavírací a jestli se tagy nekříží. Pokud je řádek validní, nechá jej být, v opačném případě jej smaže (nahradí prázdným řetězcem).

Ze vstupu

```
xx<a>ehm<b>sgfd</b>dfsd</a>
xx<a>ehm<b>sgfd</a>dfsd</b>
```

nechá program jen první řádek.

Připomínám, že k řešení patří zdůvodnění. Zvláště u této série si dejte záležet na popisu jednotlivých částí regexů, stejně jako na zdůvodnění správnosti.

Používejte k řešení jen tu část PerlRe, kterou jsme si zde vyložili, ať mají všichni stejné podmínky.

Hroší relax. Dejte si taky pauzu.

Recepty z programátorské kuchařky

Těžké problémy

V této kuchařce konečně vysvětlíme, cože je to onen velmi známý problém za milion dolarů – *Je P rovno NP?*. Než se k němu dostaneme, budeme si muset ujasnit, které problémy jsou v informatice vlastně „těžké“.

Kuchařka není nijak komplikovaná, ale doporučujeme si aspoň oprášit, co to znamená lineární a exponenciální časová složitost.

S mapou v bludišti

Představme si, že jsme v bludišti a hledáme (náš algoritmus hledá) nejkratší cestu ven. Rychle nás napadne, že bychom mohli použít prohledávání do šířky[10] a cestu najít v čase lineárním ku velikosti bludiště. To je asymptoticky nejlepší možné řešení, v nejhorším případě bude totiž bludiště jedna dlouhá nudle a i nejkratší cesta bude dlouhá lineárně vůči velikosti bludiště.

Ve skutečném životě však „kulišáci“ znají lepší řešení – podvádět! Prostě si od kamaráda půjčíme mapu bludiště s vyznačenou nejkratší cestou a pak poběžíme hned tou nejkratší cestou, aniž bychom kdekoli ztráceli čas.

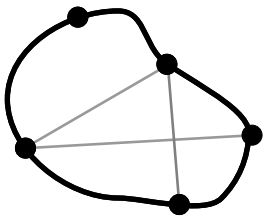
V nudlovém bludišti (nejkratší cesta má zhruba stejně vrcholů jako celý graf) jsme si vůbec nepomohli (takže je řešení asymptoticky stejně dobré). V alespoň trochu spletitém bludišti už budeme v cíli dříve než náš kamarád, který bloudí (prohledává) do šířky.

Existují tedy problémy, kde by se i v nejhorším možném případě vyplatilo podvádět pomocí taháku? Ano, zde je příklad – opět jsme v labyrintu, ale tentokrát jsou na všech stanovištích umístěny koláčky. Labyrint je to zvláštní, cesty se v něm nekříží, ale je tam plno nadchodů a podchodů.

Naším cílem je najít okružní cestu ze startovního místa zpátky na start, abychom každé stanoviště s koláčkem prošli právě jednou (protože víc než jeden koláček nám nedají).

Kdybychom tady chtěli použít procházení do šířky, bylo by to opět možné – ale tentokrát bychom se museli mnohokrát vracet, protože posloupnost stanovišť (začátek, první, druhé) může být špatná, zatímco posloupnost (začátek, druhé, první) už může být dobrá.

Přesněji řečeno, už by neplatilo, že při prohledávání do šířky každé stanoviště navštívíme nejvýše jednou, ale každou *posloupnost* stanovišť navštívíme nejvýše jednou. Projít všechny nám potrvá, matematicky řečeno, exponenciálně mnoho času vůči velikosti bludiště.



Kamarád s tahákem je na tom pořád dobře – prostě si pořídí jiný plán, na kterém bude mít vyznačenou cestu, po které má jít, aby vyhrál.

Ta cesta má stejně křižovatek jako bludiště samo, a tak bude jeho nápověda lineárně velká vůči velikosti bludiště a průchod napovězenou cestou bude trvat také lineárně. Podvodník tedy vyhrává i asymptoticky. Bídák!

Všechny by nás zajímalo, jestli by bylo možné najít tu nejlepší cestu bez podvádění v rozumně krátkém (řekněme polynomiálním) čase. Tato otázka je ekvivalentní otázce P vs. NP. Pojďme ta tajemná písmena přesně definovat.

Podvádíme s certifikáty

V teorii složitosti se často omezujeme jen na jeden typ problému, takzvaný *rozhodovací problém*. To je vlastně otázka, na kterou existují dvě možné odpovědi: ANO, nebo NE. Například „Existuje cesta z bludiště délky k?“ nebo „Je součet čísel $8 + 3$ roven 5?“

Ve zbytku kuchařky už budeme pracovat jen s nimi – skoro

[10] `http://ksp.mff.cuni.cz/tasks/20/cook3.html`

vždy se rychlé řešení rozhodovacího problému dá převést na rychlé řešení příslušného vyhledávacího problému, jako *Nalezněte nejkratší cestu z bludiště*.

Rozhodovací problém (dále už jen problém) bude náležet do *třídy problémů P* (třída je zde jen pomocné označení pro nekonečnou množinu), pokud existuje polynomiální algoritmus, který pro zadaný vstup odpoví korektně ANO nebo NE na výstupu.

Taháku z předchozí kapitolky se v literatuře říká *certifikát*. Formálně to je jen jakási polynomiálně velká informace. Můžeme si jej představit jako data, která náš program nalezne v „našeptávajícím" vstupním souboru, ke kterému program z třídy P nemá přístup.

Problém bude náležet do *třídy problémů NP* (nepoctivci), pokud existuje algoritmus a ke každé odpovědi ANO vhodný certifikát tak, že algoritmus je schopen pomocí certifikátu ověřit, že odpověď je skutečně ANO. Čili má-li ten program správný tahák, musí být schopen bludištěm projít rychle.

Zde si dejme pozor na to, že definice nedovoluje „podvádět na druhou" – nemůžeme si do pomocného souboru prostě uložit ANO a pak jej vypsat. Tak by se pak dal řešit libovolně složitý problém, i problémy mimo třídu NP! Jen tak na okraj – takové opravdu existují.

Onen algoritmus musí být schopen řešení ověřit, tedy odpovědět ANO tehdy a jen tehdy, pokud mu to napověděl certifikát a odpověď je správná. Kdyby byla skutečná odpověď NE a certifikát chybně tvrdil, že ANO, algoritmus musí být napsán tak, aby oznámil NE.

Co přesně bude certifikát, záleží na zadané úloze – často to bývá právě ono nejlepší možné řešení, kterého se stačí držet a najdeme hledanou odpověď (nebo zjistíme, že úloha nemá řešení).

Asi vám bylo hned jasné, že každý program z P patří také do NP – jakmile známe polynomiální řešení bez nápovědy, certifikátem může být i třeba prázdný soubor! Horší je to s problémy, pro které potřebujeme pro polynomiální vyřešení nějaký certifikát a zatím to lépe neumíme.

Příkladem buď problém z povídání o bludišti. Říká se mu *Hamiltonovská kružnice*.

Název problému: Hamiltonovská kružnice

Vstup: Neorientovaný graf.

Problém: Existuje v zadaném grafu kružnice procházející všemi vrcholy právě jednou?

Certifikát: Posloupnost vrcholů hamiltonovské kružnice.

Ověření v polynomiálním čase s certifikátem: Projdeme postupně vrcholy a ověříme, že jsou opravdu zapojeny do kružnice a kružnice je správné délky. Vrátíme NE, pokud tomu tak není.

Zatím nikdo nepřišel s řešením, které by nepoužívalo vůbec žádný certifikát. Dokonce zatím nikdo nenalezl problém, který by byl v NP, ale bez certifikátu už jej nelze řešit v polynomiálním čase. Kdyby takový neexistoval, třídy P a NP by se rovnaly. To je jádro otevřeného problému P vs. NP.

Převoditelnost a NP-úplnost

Když řešíme nějakou algoritmickou úlohu, obvykle přijdeme na nějaké přímé řešení využívající základních technik (prohledávání do šířky, dynamické programování, zametací přímka). Vzácně se může i stát, že v problému rozpoznáme problém jiný – občas lze geometrický problém převést na třídění čísel nebo umíme popsat situaci nějakým vhodným grafem.

Ukazuje se, že se ve třídě NP často vyplatí problémy převádět, neboť přímá řešení jsou zde vzácná. Dokonce tak můžeme i zjistit, do které z probíraných tříd problém patří.

Převodem budeme rozumět polynomiální algoritmus, který upraví vstup jednoho problému na vstup jiného problému. Musí navíc problémy převést tak, aby správná odpověď (ANO nebo NE) na vstup prvního problému byla tatáž, jako správná odpověď na vstup druhého problému.

Jednoduchým převodem je úprava problému *Existuje cesta z bludiště ze zadaného políčka délky d?* na *Existuje cesta v grafu délky c začínající v zadaném vrcholu?*.

Do výstupního grafu za každou křižovatku dáme vrchol, za každou cestu mezi křižovatkami hranu a ke hraně si poznamenáme, jak dlouhá byla. Hodnotu c pak můžeme nechat stejně velkou, jako d.

Pokud najdu správnou cestu v tomto grafu, pak nutně podobná cesta je i v bludišti, a pokud cesta v grafu není, pak není ani v bludišti. Převod je tedy korektní.

Zadefinujme si nyní pojem, který nám bude sloužit jako zkratka za to, že problém je ve třídě NP, ale není zároveň lehký (v P). Nemůžeme jen tak ledabyle říci „je v NP a není v P", protože to nevíme. To je právě ta slavná otázka.

Uděláme tedy krok stranou – budeme říkat, že problém je *NP-úplný*, pokud onen problém je v NP a zároveň jdou všechny ostatní problémy v NP převést na tento problém.

Všechny problémy v NP na něj jdou převést? Pokud tuto definici vidíte poprvé, asi to působí dost zvláštně – je těžké si představit, že všechny grafové, geometrické, počítací problémy, o kterých víte, že jsou v P (a tedy i v NP) jdou převést na nějaký NP-úplný superproblém.

Ale je to správně, ba co víc, Cookova věta[11] říká, že existuje alespoň jeden takový problém. (Samotná definice NP-úplného problému nezaručuje, že takový problém vůbec existuje.)

Ukazuje se však, že není sám, jsou jich stovky. Dokazovat existenci dalších NP-úplných problémů je však o dost lehčí, než dokázat Cookovu větu! Stačí totiž jen najít následující dva kroky:

- Dokázat, že problém je v NP – najít certifikát a polynomiální algoritmus, co jej využívá.
- Převést zadání libovolného NP-úplného problému na zadání našeho problému tak, že náš algoritmus vlastně vyřeší onen NP-úplný problém.

To postačí, protože pak libovolný jiný problém v NP nejprve převedeme na zvolený NP-úplný problém a pak pustíme námi vymyšlený převod. Zřetězení dvou polynomiálních algoritmů (převodů) je opět polynomiální algoritmus, takže podmínka převoditelnosti je splněna.

Ukážeme si důkaz NP-úplnosti jednoho problému na příkladu, pokud nám uvěříte, že již probíraný problém *Hamiltonovská kružnice* je NP-úplný. Nejprve zadefinujme jiný problém:

Název problému: Hamiltonovská cesta.

Vstup: Neorientovaný graf, dva speciální vrcholy x a y.

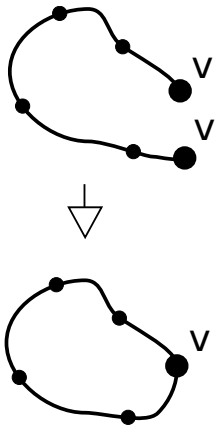
[11] `http://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem`

Problém: Existuje cesta z x do y (posloupnost vrcholů, ve které se žádné dva neopakují), která prochází každým vrcholem právě jednou?

Certifikát: Posloupnost vrcholů tvořící správnou cestu.

Řešení v NP: Projděme cestu z certifikátu a ověřme, že vrcholy jdou za sebou, je jich správný počet a žádný jsme nevynechali.

Důkaz NP-úplnosti: Převedeme předchozí problém (hamiltonovskou kružnici) na hledání hamiltonovské cesty. Uvažme graf G, ve kterém chceme najít hamiltonovskou kružnici.



Vyberme si libovolný vrchol v a vytvořme vrchol v', který bude kopií vrcholu v – do grafu přidáme hranu mezi u a v', pokud už v něm je hrana mezi u a v.

Na upravený graf zavoláme řešení problému *Hamiltonovská cesta* mezi vrcholy v a v'. Pokud taková cesta existuje, tak nutně v původním grafu G existuje hamiltonovská kružnice.

Cesta z vrcholu v' přesně odpovídá pokračováni kružnice poté, co přijde do vrcholu v.

Pseudopolynomiální algoritmy

Znáte problém batohu? Jeho varianty jsou oblíbené na programovacích soutěžích. Zadat se může třeba takto: mějme na vstupu seznam n dvojic kladných přirozených čísel, kde každá dvojice označuje váhu a cenu nějakého předmětu. Nakonec dostaneme na vstupu ještě číslo b, které udává nosnost našeho batohu.

Otázka zní: Jaký je nejcennější možný náklad, který přesto nepřesahuje váhový limit batohu?

Možná víte, že úloha jde řešit dynamickým programováním – vytvořím si pole `podbatoh[]` od 1 do b, kde `podbatoh[`i`]` je maximální hodnota, kterou bych si odnesl v batohu o nosnosti i. Postupně od první věci do poslední pak projdu celé pole `podbatoh[]` „zprava doleva“ od b do 1 a zkusím, jestli je výhodnější do batohu vložit novou věc a volné místo doplnit starými (optimální volné místo pro předchozí věci máme napočítané), nebo si nechat jen ty staré. Tuto hodnotu pak zapíšeme jako aktuální pro váhu i na místo `podbatoh[`i`]`.

Po n průchodech tohoto pole dostaneme řešení pro všechny věci dohromady na políčku `podbatoh[`b`]`. Celková složitost je $\mathcal{O}(nb)$, to je polynom, algoritmus je tedy polynomiální.

Světe div se, toto řešení je ve skutečnosti exponenciální. Kde jsme v řešení udělali chybu? Nikde – naše složitost závisela na b, ovšem když se podíváme do vstupních dat, tak pokud jsou zapsána v binárním (nebo ternárním a vyšším) tvaru, tak zápis čísla b byl veliký $\mathcal{O}(\log_2 b)$, ale naše složitost závisela na $b = 2^{\log_2 b}$, tedy exponenciálně vůči velikosti vstupu.

Problém batohu, respektive jeho rozhodovací verze, je dokonce NP-úplný problém.

Algoritmům, které řeší nějakou úlohu a jsou polynomiální oproti *hodnotě* čísel na vstupu, ale exponenciální ve *velikosti zápisu* těchto čísel, říkáme *pseudopolynomiální algoritmy*. Některé další NP-úplné problémy mají pseudopolynomiální řešení (jako například *Dva loupežníci* níže), ale dá se dokázat, že na jiné problémy pseudopolynomiální algoritmus neexistuje (pokud P $\neq$ NP).

Mimochodem: pokud bychom na vstupu zapisovali čísla v unárním zápisu, každý pseudopolynomiální problém by ležel v P.

Poznámky na závěr

Otázku „Je třída P rovna NP?“ se již snažilo rozlousknout mnoho matematiků a informatiků. Tato teorie přinesla spoustu zajímavých výsledků, například už se podařilo dokázat, že některými technikami tuto domněnku nelze nikdy dokázat, ani vyvrátit.

Kdyby platilo P = NP, pak by mnoho lidi zajásalo – mnoho přirozených problémů, které nastávají i v reálném životě, by najednou byla řešitelná rychle. Navíc by krachlo dosavadní šifrování a bylo by možné najít rychle důkaz ke každému pravdivému tvrzení výrokové logiky.

Tato rovnost by se dala hypoteticky ukázat velice snadno – stačilo by najít jeden polynomiální algoritmus pro libovolný NP-úplný problém! Většina informatiků studujících složitost se však domnívá, že se třídy nerovnají.

To ale neznamená, že si to nemáte zkusit dokázat! Naopak, bojovat s NP-úplnými problémy je užitečné i v reálném světě – například jde mnohdy vymyslet dobrá aproximace NP-úplného problému.

Například nenajdeme hamiltonovskou kružnici v polynomiálním čase, ale nalezneme nějakou relativně dlouhou kružnici, která nám v praxi může stačit, pokud podle ní třeba chceme vést náročný cyklistický závod.

O aproximacích je toho v literatuře napsáno mnoho zajímavého, pokud byste si o nich chtěli přečíst více v češtině, zkuste třeba výpisky z předmětu na Matfyzu.[12]

O NP-složitosti můžete něco najít na stejné adrese, nebo zkuste vynikající anglicky psanou knížku *Algorithms* od profesorů exotických jmen Dasgupta, Papadimitriou a Vazirani.

Existují i problémy, které jsou mimo P i NP, a dokonce existuje spousta různých dalších tříd problémů. Je jich celá zoologická zahrada – můžete ji najít na internetu.[13]

[12] `http://mj.ucw.cz/vyuka/0910/ads2/12-apx.pdf`

[13] `http://qwiki.stanford.edu/index.php/Complexity_Zoo`

Seznam NP-úplných problémů

Sedíte-li nad zatím nevyřešenou úlohou, kterou jste nalezli jinde než v KSP, pak se klidně může stát, že bude NP-úplná. Abyste mohli mezi NP-úplnými úlohami převádět, tak je dobré znát jich aspoň hrstku, podle toho, je-li problém grafový, rovnicový, a tak dále.

V následujícím seznamu najdete několik úloh, které jsou zaručeně NP-úplné. Převody se nám sem sice nevešly, ale mnoho z nich (ne-li všechny) zvládnete vymyslet sami – zkuste si to!

Název problému: Hamiltonovská kružnice

Vstup: Neorientovaný graf.

Problém: Existuje v zadaném grafu kružnice procházející všemi vrcholy právě jednou?

Název problému: Hamiltonovská cesta.

Vstup: Neorientovaný graf, dva speciální vrcholy x a y.

Problém: Existuje cesta z x do y (posloupnost vrcholů, ve které se žádné dva neopakují), která prochází každým vrcholem právě jednou?

Název problému: Splnitelnost

Vstup: Logická formule. Tu tvoří proměnné a logické spojky negace $\neg$, konjunkce $\wedge$ a disjunkce $\vee$. Například

$$(x \wedge (\neg y)) \vee z.$$

Problém: Můžeme proměnným přiřadit hodnoty 0 nebo 1 tak, že výsledná vyhodnocená formule má hodnotu 1?

Název problému: Součet podmnožiny

Vstup: Seznam nezáporných celých čísel, speciální číslo k.

Problém: Existuje podmnožina čísel, jejíž součet je přesně k?

Název problému: Batoh

Vstup: Seznam dvojic nezáporných čísel, kde dvojice označuje hodnotu a váhu předmětu. Přirozené číslo b – nosnost batohu, přirozené číslo k.

Problém: Umíme vložit do batohu předměty o hodnotě alespoň k, aniž bychom přešli přes limit váhy b?

Název problému: Dva loupežníci

Vstup: Seznam nezáporných celých čísel.

Problém: Existuje rozdělení seznamu na dvě hromádky tak, že každé číslo bude v právě jedné hromádce a v každé hromádce bude stejný součet čísel?

Název problému: Klika

Vstup: Neorientovaný graf, číslo k.

Problém: Existuje v grafu úplný podgraf o velikosti k, tedy k vrcholů takových, že mezi každými dvěma z nich vede hrana?

Název problému: Nezávislá množina

Vstup: Neorientovaný graf, číslo k.

Problém: Existuje v grafu prázdný podgraf o velikosti k, tedy k vrcholů, že žádné dva z nich nejsou spojeny hranou?

Název problému: Trojbarevnost grafu

Vstup: Neorientovaný graf.

Problém: Lze vrcholy tohoto grafu obarvit třemi barvami tak, že každá hrana sousedí s vrcholy dvou různých barev?

Název problému: Rozparcelování roviny

Vstup: Seznam bodů v rovině, kde každý má navíc přiřazenu jednu z b barev, číslo k.

Problém: Umíme rozdělit rovinu pomocí k přímek tak, že v každé oblasti jsou jen body té samé barvy?

Název problému: 3D párování

Vstup: Seznam mužů, žen a zvířátek, následovaný seznamem kompatibilních trojic tvaru $\{\text{muž}, \text{žena}, \text{zvířátko}\}$. Tyto trojice říkají, která trojice muž, žena a zvířátko by se dohromady snesla.

Problém: Můžeme všechny muže, ženy a zvířátka z prvního seznamu rozdělit do trojic tak, že každá trojice je kompatibilní a každá bytost je právě v jedné trojici?

Kuchařku sepsal Martin Böhm.

Vzorová řešení čtvrté série

23-4-1 Studenti a profesoři

Všichni, kdo se odvážili odevzdat řešení, argumentovali převoditelností na problém maximálního toku – kuchařka v tomhle směru napovídala dost jasně. Nikdo pod devět bodů nedostal (vždyť také nikdo celé řešení pouhým poukazem do mého textu neodbyl), ve zbývajícím rozsahu jsem tak hodnotil úvahy o časové složitosti a nuance.

Je docela jasné, že si budeme uzpůsobovat první ze dvou zmíněných aplikací, která mluví o tom, jak pomocí toku najít maximální párování. Postavíme si ze zadání bipartitní graf, zorientujeme v něm hrany k profesorům, vrcholy studentů a profesorů pak napojíme na studentský zdroj a profesorský stok.

Protože chceme, aby měl student právě K profesorů, nastavíme váhu každé z hran ze studentského zdroje na K – to samé uděláme hranám do profesorského stoku, to aby měl každý profesor právě K studentů. Hranám uvnitř někdejšího bipartitního grafu nastavíme jedničky.

Povšimněme si tu, že kdyby zadání nezakazovalo, aby si některý student vybral profesora pro několik svých prací, vyrovnali bychom se s tím jednoduše – hraně, která by mezi příslušnými vrcholy vedla, bychom nastavili kapacitu na povolenou maximální násobnost.

Samozřejmě by ani nebyl problém mít rozdílný počet profesorů a studentů, či dokonce zavést individuální požadavky na počet vedených prací. Zadání bylo tak jednoduché předně proto, aby neděsilo.

Vraťme se k původní úloze. Na popsaný graf pustíme tokový algoritmus zachovávající celočíselnost a získáme z něj výsledek. Pokud není nalezený tok velký právě NK, řešení, které by každého plně uspokojilo, není. Pokud ano, vypíšeme páry profesor-student, jejichž hrana má jednotkový tok.

Důvod, že postup funguje, můžeme načrtnout třeba skrze fakt, že tok větší než NK v grafu existovat nemůže. Svědčí o tom řez na hranách mezi studentským zdrojem a studentskými vrcholy, kde je N hran, každá o kapacitě K.

Z toho vidíme, že pokud nám algoritmus vrátí takto velký

tok, musí vést z každého studentského vrcholu k profesorům K jednotkových hran (a podobně ze strany profesorů), tedy jde o skutečné řešení našeho původního problému.

Zároveň se nemůže stát, aby postup řešení (maximální tok) nenašel a ono by existovalo – vždyť z každého řešení sestavíme tok o maximální velikosti.

Co časová složitost? Smířit se s tím, že má Edmondsův-Karpův algoritmus složitost $\mathcal{O}(M^2N)$, je přístup lenivý. Nicméně si můžeme všimnout, že zlepší-li každá cesta výsledek alespoň o jednotku, nenajdeme takových cest víc než KN.

Z toho plyne složitost $\mathcal{O}(KMN)$, což je lepší, protože pro $K > N$ úloha zřejmě není zajímavá.

Vysloveně akční přístup je začít se poohlížet po nekuchařkovém algoritmu. (To ale k získání maximálního počtu bodů potřeba nebylo.) Můžeme buď přemýšlet o tom, jestli není možné vzít Dinice či Goldberga a vzhledem k jisté speciálnosti našeho grafu vylepšit odhady časové složitosti, nebo zkusit najít specializovaný postup.

Vtip tkví v tom, že při zkoumání druhé možnosti nejspíše narazíme na Hopcroftův-Karpův algoritmus pro nalezení maximálního párování v bipartitním grafu běžící v čase $\mathcal{O}(M\sqrt{N})$, který je však jen dobře odhadnutý a přeříkaný Dinic.

My tu sice nechceme bipartitní párování, leč každé naše řešení (K-regulární bipartitní podgraf) se skládá z K takových disjunktních množin hran (1-regulárních bipartitních podgrafů). To není úplně vidět, ale je to hezká a užitečná pravda.

Můžeme tedy K-krát spustit Hopcrofta-Karpa a pokud nějaké řešení existuje, získáme ho v čase $\mathcal{O}(KM\sqrt{N})$. Pořád tak netrumfneme škálu rozličných moderních algoritmů pro hledání maximálního toku na obecném grafu, jde však o celkem srozumitelné a snadno naprogramovatelné řešení.

Lukáš Lánský

23-4-2 Paralelní profesoři

Tuto úlohu se pokoušelo vyřešit jen 8 z vás a k mému zklamání jen jedno řešení bylo úplně správně. Gratulace patří Vojtěchu Hlávkovi.

Nejčastější chybou bylo, že jste úlohu vyřešili pro $N = 2^k$ a zobecnili pro všechna N. Proč je tato úvaha špatná, je dobře vidět například pro $N = 3$.

Jak to tedy mělo být? Pokud $N = 2^k$, tak v prvním kroku profesory rozdělíme do dvojic a tím získáme dvojice profesorů se stejnými informacemi. Ve druhém kroku k sobě posadíme různé dvojice profesorů a tím získáme čtveřice profesorů se stejnými informacemi atd.

Až se dostaneme k jedné skupince o velikosti 2^k. Bude nám tedy stačit $\log_2 N$ sezení. Problém s dělením nikde nenastane, počet skupinek bude vždy sudý.

Pro jiné počty profesorů ale tento algoritmus aplikovat nemůžeme, protože v nějakém kroku dostaneme lichý počet skupinek a ten už neumíme jednoduše spárovat.

Úlohu vyřešíme zvlášť pro sudá a lichá čísla. U lichých čísel si můžeme všimnout, že při každém sezení bude alespoň jeden z profesorů lichý. Spočítáme si tedy, za kolik nejméně sezení se můžou všichni profesoři dozvědět informaci od toho, který byl lichý při prvním sezení.

Po prvním sezení ví onu informaci pouze on sám a po každém dalším sezení se množství profesorů se znalostí této informace může maximálně zdvojnásobit. Z toho vyplývá, že všichni profesoři můžou tuto informaci znát nejdřív po $\lceil \log_2 N \rceil + 1$ sezeních.

$\lceil x \rceil$ je takzvaná horní celá část, nejmenší celé číslo větší nebo rovné x.

Nyní, když najdeme obecný algoritmus pro lichá čísla, který řeší úlohu v $\lceil \log_2 N \rceil + 1$ krocích, tak máme vyhráno. Každé číslo si můžeme napsat jako $2^k + l$, kde k a l jsou přirozená čísla a k je nejvyšší možné.

Pak při prvním sezení sprárujeme l profesorů s některými z 2^k, těchto 2^k už umíme vyřešit v k krocích a nakonec opět zbylých l profesorů spárujeme s některými z 2^k.

Situaci se nám tedy povedlo vyřešit na

$$k + 2 = \lfloor \log_2 N \rfloor + 2 = \lceil \log_2 N \rceil + 1 \text{ kroků},$$

a to jsme chtěli.

Zbývají jen sudá čísla. Obdobně jako u lichých čísel ukážeme, že minimální nutný počet sezení je $\lceil \log_2 N \rceil$.

Profesory očíslujeme $0, 1, \ldots, N-1$. První sezení spárujeme $(0, 1), (2, 3), \ldots, (N-2, N-1)$, tím každý zná dvě informace.

Pro druhé sezení vytvoříme dvojice $(0, 3), (2, 5), (4, 7), \ldots$ Tím všichni profesoři se sudým číslem s mají informace $s \ldots (s + 3) \bmod N$, po k-tém sezení mají analogicky informace $s \ldots (s + 2^k - 1) \bmod N$.

Lichá čísla jsou k sudým párována symetricky, takže až sudí budou znát vše, tak i liší. Celkem nám tedy bude stačit $\lceil \log_2 N \rceil$ sezení.

Obecně N je tedy optimální počet sezení

$$\lceil \log_2 N \rceil + (N \bmod 2).$$

Jedinou výjimku tvoří $N = 1$, kde nepotřebujeme žádné sezení.

Karel Tesař

23-4-3 Zabugovaný program

Dva zlatokopové, neboli ve známější verzi loupežníci, zvolili hladový algoritmus. Předložený program setřídil vstupní hodnoty a potom je hladově rozdělil mezi zlatokopy.

Hladově, to znamená tak, že se podíval, který z nich má zrovna méně, a tomu nuget přidělil. Začínal od největšího, skončil nejmenším.

Rychle jste odhalili, že potřebujete najít *false negative*, tedy vstup, u kterého program nenalezne správné řešení, byť by existovalo. Když totiž program ohlásí řešení, je zjevně správně.

Nejmenší vstup, na kterém se program zachoval chybně, byl `3 3 2 2 2`, kde bylo správným řešením dát jednomu ze zlatokopů `3 3` a druhému `2 2 2`. Program si nicméně tvrdošíjně mlel svou a po rozdělení `3 2 2` a `3 2` prohlásil, že řešení neexistuje.

Vstup byl nejmenší co do počtu nugetů. Řešitelé, kteří to dokázali, získali body navíc.

Důkaz byl docela jednoduchý rozbor případů. Vstup s jedním nugetem nemá řešení. Vstup se dvěma nugety a_1, a_2 může mít řešení jen pro $a_1 = a_2$, což náš program najde. Vstup se třemi nugety $a_1 \le a_2 \le a_3$ může mít řešení jedině pro $a_1 + a_2 = a_3$, což náš program zase bez problému najde.

V případě čtyř nugetů na vstupu ($a_1 \le a_2 \le a_3 \le a_4$) bylo potřeba vyřešit několik možných případů. Program vždycky rozdělil nugety a_1 a a_2 na dvě různé hromádky. Kdyby

platilo $a_1 = a_2$, muselo by také platit $a_3 = a_4$, jinak by řešení neexistovalo (dokažte za domácí úkol). V takovém případě ale náš program funguje.

Tudíž $a_1 > a_2$, a tedy náš program dá a_3 na hromádku k a_2. Nakonec odloží a_4 na menší z obou hromádek. Pokud platí $a_4 = |a_1 - a_2 - a_3|$, program vydá správné řešení; rozmyslete si, že to platí vždy.

Základem důkazu je na tomto místě úvaha, za jakých podmínek by ve všech správných řešeních musely být a_1 a a_2 nebo a_1 a a_3 na společných hromádkách, nebo a_2 a a_3 na různých.

Vstup byl i nejmenší co do celkové hodnoty. Za důkaz jsme taktéž udělovali bonusové body. Taktéž byl nejrozumnějším přístupem rozbor případů.

Někteří řešitelé si nevšimli, že program bral vstupní hodnoty sestupně. Pythoní kód to řešil metodou `pop`, která odebírá z konce seznamu, program v C třídil obráceně a procházel pole od začátku.

Za řešení s touto chybou jsme udělovali 2 body. Jeden za správnost, druhý za nápad, jak si úlohu výrazně zjednodušit (nejmenším protipříkladem by byl vstup `1 1 2`).

Martin „Medvěd“ Mareš & Jan „Moskyto“ Matějka

23-4-4 Závorky v TEXu

Nejprve se podívejme na rozpoznávání správného uzávorkování. Řetězec se závorkami `{` a `}` budeme procházet zleva doprava a počítat si, kolik neuzavřených levých závorek nám zbývá (tento počet označme k). Mohou nastat pouze dva případy znamenající, že řetězec není správně uzávorkovaný:

- k je 0 a přečteme uzavírací závorku (počet neuzavřených klesne pod nulu),
- k bude na konci řetězce větší než 0.

Jak poznat, že lze řetězec změnit na správně uzávorkovaný pouhými změnami znaku? Je zřejmé, že pro lichý počet to učinit nelze a pro sudý naopak vždy lze, protože můžeme jednoduše změnit všechny znaky na řetězec `{}{}{}{}`...

Nyní přejděme k algoritmu, který řeší naši úlohu a zajišťuje minimální počet změn znaků. Stejně jako při rozpoznávání, jestli je uzávorkování správné, budeme procházet závorky zleva doprava a počítat si neuzavřené levé.

Když k klesne pod 0 na pozici i, musíme změnit nějakou uzavírací závorku na pozici menší nebo rovno i (na pozici větší než i už to nepomůže). Je celkem jedno kterou, jde nám jen o počet změn. Také nesmíme zapomenout aktualizovat počet neuzavřených závorek (z -1 na 1).

Po přečtení poslední závorky mohou nastat 3 případy:

- k je 0 – pak už máme správně uzávorkovaný řetězec a vypíšeme počet dosud provedených změn,
- k je liché – pak i celkový počet závorek je lichý a řetězec nelze správně uzávorkovat,
- k je sudé – musíme tedy nějaké otevírací závorky změnit na uzavírací a nesmí to být libovolné, protože bychom mohli dostat špatně uzávorkovaný řetězec (při čtení zleva by klesl počet neuzavřených levých závorek pod 0).

Určitě nic nepokazíme, pokud budeme otevírací závorky měnit zprava. Počet změn je $k/2$ (každou změnou klesne počet neuzavřených levých závorek o 2).

Časová složitost je zjevně lineární a lépe to nejde (musíme se podívat na každou závorku). Část řešitelů prohlásila i paměťovou složitost za lineární, což kupodivu jde zlepšit. Kdo četl zadání pozorně, všiml si, že úkolem bylo najít pouze počet změn.

Stačilo tedy číst znaky ze vstupu (např. z obrovského souboru) a vůbec je neukládat, což dává konstantní paměťovou složitost. Kdo chtěl vracet správně uzávorkovaný řetězec, musel si pamatovat alespoň část řetězce od poslední změny znaku `}` na `{` nebo od posledního nulového počtu neuzavřených levých závorek, takže nejhůře celý řetězec.

Možná se zcela správně ptáte, proč náš algoritmus dává minimální počet změn. Je zřejmé, že pro správně uzávorkovaný řetězec vypíše 0. Pro špatně uzávorkovaný žádnou ze změn provedených při kontrole, jestli k nekleslo pod 0, nemůžeme vrátit. Na konci také musíme změnit nějakých $k/2$ otevíracích závorek.

Navíc nelze na žádné pozici provést změnu z `{` na `}` a potom zpět na `{` (tj. obě změny by byly zbytečné), protože by nám opět kleslo k na té pozici pod 0. Algoritmus tedy dává minimální počet změn.

Vzorový program je na konci letáku.

Pavel „Paulie“ Veselý

23-4-5 Palindromnásobky

Zkusme řešit jednoduše – projdeme všechna čísla délky D dělitelná K a započítáme ta z nich, která jsou palindromem. Časová složitost tohoto řešení je $\mathcal{O}(D \cdot 10^D/K)$, protože čísel, která testujeme, je $\mathcal{O}(10^D/K)$ a pro otestování, zda je číslo palindromem, musíme projít všech jeho D číslic.

Co takhle zkusit to naopak, procházet všechny palindromy a určit, které z nich jsou dělitelné K? Palindromy projdeme tak, že začneme nejmenším z nich (jeho první a poslední číslice jsou 1, všechny ostatní 0) a vezmeme první polovinu jeho číslic, začínajíce od největšího řádu a včetně prostřední číslice v případě lichého D.

Toto číslo zvětšíme o jedna a zrcadlíme zpět, abychom získali palindrom odpovídající délky. Tedy například $13931 \to 139 \to 140 \to 14041$. Stejný postup opakujeme, dokud se nedostaneme k číslu obsahujícímu samé devítky, čímž jsme u konce.

Abychom nemuseli v každém kroku palindrom půlit a pak zase zrcadlit zpátky, můžeme pracovat přímo s palindromem, jenom začneme uprostřed a případný přenos šíříme na obě strany. Takto dostaneme časovou složitost $\mathcal{O}(10^{D/2})$.

Exponenciální časové složitosti jsou ale hodně ošklivé. Copak tahle úloha nejde vyřešit v (pseudo-)polynomiálním čase? (Proč pseudo? Obvykle se složitost měří vzhledem k *délce* vstupu, pokud je ale složitost polynomiální vzhledem k *hodnotě čísel* na vstupu, říká se takovému algoritmu pseudopolynomiální.)

Jistěže jde, jenom je potřeba se trochu zamyslet. Každý palindrom můžeme jednoznačně rozložit na $\lceil D/2 \rceil$ podpalindromů stejné délky jako celý palindrom tak, že i-tý podpalindrom má nenulové cifry pouze na i-té pozici od začátku a i-té pozici od konce. Tyto podpalindromy mohou mít, na rozdíl od běžných palindromů, nuly na začátku. Například 10301 rozložíme na 10001, 00000 a 00300.

Jak tohoto rozkladu využijeme? Vytvoříme tabulku zbytků – pro každou možnou hodnotu zbytku po dělelní číslem K (tedy pro čísla 0 až $K-1$) si budeme pamatovat, kolika

různými způsoby umíme vytvořit palindrom s daným zbytkem.

V prvním kroku projdeme podpalindromy, které mají nenulovou číslici na prvním (a tedy i na posledním) místě. Pro každý z nich určíme jejich zbytek a tyto počty si poznamenáme.

Ve druhém (a obdobně v každém dalším) kroku postupujeme tak, že nejdříve vytvoříme novou tabulku zbytků zkopírováním té staré, protože všechny palindromy, které jsme uměli vytvořit v předchozím kroku, umíme vytvořit stále (rozklad takového palindromu by měl na odpovídajícím místě podpalindrom ze samých nul).

Dále projdeme podpalindromy, které mají nenulovou číslici na druhém (a tedy i na předposledním) místě. Pokud má podpalindrom zbytek r, přičteme do nové tabulky hodnoty ze staré, cyklicky posunuté o r míst. To proto, že pokud jsme v předchozím kroku uměli vytvořit n palindromů se zbytkem q, umíme s využitím aktuálního podpalindromu vytvořit n nových palindromů se zbytkem $(q + r) \bmod K$.

Po posledním kroku takto získáme v závěrečné tabulce zbytků na pozici 0 počet palindromů délky D, které mají zbytek po dělení číslem K rovný nule, což je přesně to, co jsme chtěli. Vzhledem k tomu, jak s palindromy a podpalindromy pracujeme (a s využitím předpočítaných zbytků mocnin desítky) si je dokonce ani nemusíme pamatovat celé, stačí vždy jejich zbytek po dělení K.

Celková časová složitost tohoto algoritmu je $\mathcal{O}(D \cdot 10 \cdot K)$, protože pro každý podpalindrom, kterých je $10 - 1$ v každé z $\lceil D/2 \rceil$ skupin, přičítáme K hodnot do tabulky zbytků (kromě podpalindromů s první číslicí nenulovou, které jsou jednodušší).

Možná by vás mohlo zarazit použití konstanty 10 v časové složitosti. Pokud bychom chtěli stejnou úlohu řešit v jiné soustavě, než je desítková, nahradili bychom toto číslo základem dané soustavy. Pokud ale nad takovou možností neuvažujeme, můžeme časovou složitost zapsat jako $\mathcal{O}(DK)$.

Vzhledem k tomu, že používáme předpočítané zbytky mocnin desítky, a vzhledem k tomu, že v každém kroku nám stačí dvě tabulky zbytků (aktuální a z předešlého kroku), je paměťová složitost $\mathcal{O}(D + K)$.

A ještě jeden dodatek na konec – zadané limity byly takové, že výsledek mohl být tak velký, že se nevešel do 32-bitového integeru, takže pro získání plného počtu bodů bylo potřeba použít 64-bitový integer.

Vzorový program je na konci letáku.

Petr Onderka

23-4-6 Knuthovy cesty po státech

Zkusme postupovat tak, že budeme následovat Knuthovu cestu a na každé křižovatce si pamatovat, kam až sahá nejdelší úsek cesty, který se nekříží a zároveň končí tam, kde zrovna jsme. Z takovýchto úseků pak vezmeme nejdelší a jsme hotovi.

Nyní předpokládejme, že známe nejdelší nekřížící se úsek končící i-tou křižovatkou (jeho začátek označme Z) a chceme najít takovou část cesty pro další křižovatku (nechť je to křižovatka K). Tam nám mohou nastat 2 případy:

1) Křižovatku K jsme navštívili před Z (popř. jsme ji nenavštívili vůbec). Pak můžeme nejdelší nekřížící se úsek cesty prodloužit o křižovatku K.

2) Křižovatku K jsme navštívili během nejdelší nekřížící se cesty pro i-tou křižovatku. Pak nastavíme nový začátek Z hned za minulou návštěvu křižovatky K a pokračujeme dál.

Zřejmě pokud bychom prodloužili aktuální cestu o jednu křižovatku zpět, dostali bychom se na nějakou podruhé, a tedy v každém kroku je nalezený úsek nejdelší nekřížící se.

Na to, aby výše uvedený postup fungoval efektivně, budeme potřebovat vědět, kdy jsme naposledy jakou křižovatku navštívili. To se udělá snadno pomocí pole o velikosti počtu křižovatek, kde si budeme příslušnou informaci udržovat.

Časová složitost je lineární a paměťová také.

Vzorový program je na konci letáku.

Pavel Čížek

23-4-7 Bratrstvo Seda a Grepa

Omluva na začátek. Formulace některých úkolů byly vágní a umožňovaly různé interpretace. Přesto jste je pochopili převážně tak, jak jsem je původně myslel.

Řešení **úkolu 1** bylo správně u všech, kdo jej poslali.

```
s/[ \t\r]+$//
```

Jeden výtečník zapomněl na dolar a místo něj použil chybné `\n`, za což byl nepatrně ztrestán (`sed` čte vstup po řádcích, takže `\n` na vstupu defaultně nikdy není). Hezké bonusové řešení předvedl Vojta Hlávka:

```
s/[[:space:]]*$//
```

Druhý úkol byl potvorný. Jedním ze správných řešení bylo napsat regex

```
s/(([ ~][KkOoSsUuVvZzIiA])|
            ([[:punct:]][[:space:]~]?a)) /\1~/g
```

a prohlásit, že jej spustíme dvakrát. Proč? Poprvé ovlnkuje liché, podruhé sudé výskyty. Vstup „`..., a i s nimi`“ se tedy nejprve změní na „`..., a~i s~nimi`“, aby po druhém průchodu přibyla i prostřední vlnka a vznikl kýžený výsledek „`..., a~i~s~nimi`“.

Druhá správná varianta se dala vymyslet s manuálem GNU `sed`u v ruce, kde jste se mohli dočíst o `\b`, což je *předpoklad nulové délky* s významem „hranice slova“.

Jinak řečeno, když `sed` přijde k `\b`, tak se podívá, jestli je na hranici slova. Pokud ano, pokračuje dál, jinak se vrátí a zkusí jinou variantu. Druhé možné řešení tedy bylo

```
s/((\b[KkOoSsUuVvZzIiA])|
            ([[:punct:]][[:space:]~]?a)) /\1~/g,
```

které už stačilo spustit jednou.

Většina řešitelů si nevšimla buďto problému s ovlnkováním řetězu předložek, nebo zapomněli na variantu se spojkou `a` a interpunkcí apod. Toleroval jsem neúplný výčet interpunkce i chybějící předložky v seznamu, nicméně v praktickém použití si na to dejte pozor.

Úkol 3 byl zadaný vágně. Nebyla totiž definována abeceda, nad kterou se problém řeší. Většina řešitelů předpokládala, že bude rozumná a nebude obsahovat speciální znaky. Za takových podmínek byl problém řešitelný.

Nabízí se řešení přímočaré, leč chybné:

```
egrep '^s(.).+\1.*\1g?$'
```

Vyhovuje mu totiž například i řetězec `saaaaaa`. Jak tomu předejít? Vykutálený trik některých řešitelů `[^\1]` nefunguje (`\1` se totiž uvnitř hranatic interpretuje jako dvojice znaků `\` a `1`).

Nuže, přilepíme k prvnímu regexu filtr, který zahodí nevyhovující řetězce (obsahují moc oddělovačů).

```
        | egrep -v '^s(.).*\1.*\1.*\1.*$'
```

To je sice hezké, ale tentokrát neprojde `sgaggg`. Musíme oddělit `g`. Zde je kompletní správné řešení.

```
egrep '^s(.).+\1.*\1g?$' |
egrep -v '^s(([^g]).*\1.*\1.*\1.*|
                    g[^g]*g[^g]*g([^g]+|g.+))' |
                              egrep -v '^s\1\1.*'
```

První regex vytáhne kandidáty, druhý z nich vyháže ty, které mají nadbytečný počet oddělovačů (dobře si prohlédněte druhou větev, ve které se řeší `g`) a třetí ještě smaže ty, které mají prázdný regex k vyhledání, neboť ty také nejsou validní.

Ještě by se dalo připustit v abecedě zpětná lomítka. S tím se úspěšně porval jeden člověk.

Jakub Zíka se pustil do složitějšího rozboru případu, kdy uvažoval obecnější abecedu, která by mohla obsahovat speciální znaky. Právem mu náleží dva bonusové body.

Pokud by abeceda obsahovala kulaté závorky, nemůžeme ani zkontrolovat jejich správné vnoření, ani zkontrolovat validitu backreferencí a v tom případě je úkol zadanými prostředky neřešitelný.

Třešničkou na dortu pak byl **úkol 4**. Objevil se nápad počítat si po 1, dokud nedojdeme k menšímu ze zadaných čísel (a vypsat pak to druhé). Má to jeden háček – napsat sčítačku je možná o něco těžší než tento úkol samotný...

Autorské řešení spočívalo v porovnání řetězců nejprve podle délky, pak už bylo přímočaré.

```
s/([01]+) ([01]+)/\1#\2#\1#\2/
s/#[01]([01]*)#[01]([01]*)$/#\1#\2/
s/([01]+)#([01]+)##[01]+/\2/
s/([01]+)#([01]+)#[01]+#$/\1/
s/([01]+)1([01]*)#\1[0]([01]*)##$/\11\2/
s/([01]+)0([01]*)#\1[1]([01]*)##$/\11\3/
```

První řádek zamezí vícenásobnému startu (změna oddělovače) a připraví půdu pro porovnání podle délky – vytvoří kopie, ze kterých budeme usekávat číslice.

Druhý řádek usekne z kopie vstupu po číslici. Třetí a čtvrtý řádek ošetřují případ, kdy je jedno číslo kratší než druhé.

Na pátém a šestém řádku jsme zjistili, že jsou čísla stejně dlouhá, takže z nich vybereme to větší a vypíšeme.

Kdyby mohly být na začátku čísel nuly, stačilo by doplnit na vhodná místa `0*`.

Ještě nabízím variantní řešení se sčítačkou.

```
s/^([01]+) ([01]+)$/\1-\2@0/
s/@([01]*)0$/#\11/
s/@([01]*)01(1*)$/:\110\2/
s/:([01]*0+)1(1*)$/:\10\2/
s/:([01]*0+)$/#\1/
s/^([01]+)-([01]+)#\1$/\2/
s/^([01]+)-([01]+)#\2$/\1/
s/#/@/
```

První řádek je vstupní, druhý řádek přičítá k sudému číslu, třetí až pátý k lichému. Významy oddělovačů jsou snad jasné. Pro ještě nezvýšené číslo používáme `@`, pro právě inkrementované číslo používáme `:` a pro hotové číslo k porovnání máme `#`.

Zde by se už hodila analýza časové složitosti. Předpokládejme, že vyhodnocení regexu trvá jednotkový čas. Reálný odhad to není a asi nikdy nebude, leč pro představu to stačí.

První řešení nejdřív postupně odsekává po číslici, což trvá N kroků (N číslic na vstupu). Porovnání stejně dlouhých čísel už proběhne v konstantním čase, takže složitost odsekávacího řešení je $\mathcal{O}(N)$ cyklů.

Druhé řešení počítá po jedné. Byť stráví sčítáním od 1 do K jen $\mathcal{O}(K)$ cyklů, je to pořád $\mathcal{O}(2^N)$, neboť K může být s $\mathcal{O}(N)$ číslicemi na vstupu až exponenciálně veliké.

Není všechno zlato, co se třpytí, aneb přičítání jedničky vypadá lákavě, leč jeho rychlost není závratná. Naopak zdánlivě chlupaté řešení se sekáním číslic je výrazně rychlejší a efektivnější.

Jan „Moskyto“ Matějka

23-4-4 Program (Závorky v TEXu) — Python

```
import sys
k = 0 # počet neuzavřených levých závorek
zmen = 0
while True:
  znak = sys.stdin.read(1) # čte znak ze vstupu
  if znak == '{' :
    k = k + 1
  elif znak == '}' :
    k = k - 1
  else :
    break;
  if k < 0 :
    k = k + 2 # změna z otevírací na uzavírací
    zmen = zmen + 1

if k % 2 == 1 : # počet závorek je lichý
  print("Nelze správně uzávorkovat")
else :
  zmen = zmen + k / 2 # uzavři přebývající levé
  print("Minimální počet změn je " + str(zmen))
```

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

const int radix = 10;

unsigned long long solve(int K, int D)
{
    // předpočítané zbytky mocnin desítky
    int *powers = malloc(sizeof(int) * D);

    int remainder = 1;

    for (int i = 0; i < D; i++)
    {
        remainder = remainder % K;
        powers[i] = remainder;
        remainder = remainder * radix;
    }

    int baseCount = (D + 1) / 2;

    // předpočítané zbytky podpalindromů s nenulovými číslicemi rovnými 1
    // zbytky ostatních podpalindromů se z těchto snadno dopočítají
    int *bases = malloc(sizeof(int) * baseCount);

    for (int i = 0; i < baseCount; i++)
    {
        int j = D - 1 - i;
        int base;
        if (i == j)
            base = powers[i];
        else
            base = (powers[i] + powers[j]) % K;
        bases[i] = base;
    }

    // tabulka zbytků pro aktuální krok
    unsigned long long *newPalindromes = calloc(K, sizeof(unsigned long long));

    remainder = bases[0];

    for (int i = 1; i < radix; i++)
    {
        newPalindromes[remainder]++;
        remainder = (remainder + bases[0]) % K;
    }

    // tabulka zbytků pro předešlý krok
    unsigned long long *palindromes = malloc(sizeof(unsigned long long) * K);

    for (int i = 1; i < baseCount; i++)
    {
        for (int t = 0; t < K; t++)
            palindromes[t] = newPalindromes[t];

        remainder = bases[i];

        for (int j = 1; j < radix; j++)
        {
            for (int k = 0; k < K; k++)
                newPalindromes[(k + remainder) % K] += palindromes[k];

            remainder = (remainder + bases[i]) % K;
        }
    }

    return newPalindromes[0];
}

int main(void)
{
    int K, D;
```

```
    FILE *in = fopen("vstup.in", "r");
    fscanf(in, "%d %d", &K, &D);
    fclose(in);

    unsigned long long solution = solve(K, D);

    FILE *out = fopen("pocet.out", "w");
    fprintf(out, "%llu", solution);
    fclose(out);
}
```

23-4-6 Program (Knuthovy cesty po státech) — Pascal

```
const
   MaxK = 1000;
   MaxN = 1000;

var
   N:integer;
   Cesta:array[1..MaxN] of integer;
   Navstivena:array[1..MaxK] of integer;
   Krizovatka:integer;
   Zacatek,Pozice:integer;
   Nejdelsi_Zacatek,Nejdelsi_Delka:integer;
begin
   readln(N);
   for Pozice:=1 to N do begin
      read(Krizovatka);
      Cesta[Pozice] := Krizovatka;
      Navstivena[Krizovatka] := -1;
   end;                  {načtení vstupu a inicializace pole s posledními návštěvami křižovatek}
   Nejdelsi_Zacatek := 1;
   Nejdelsi_Delka := 0;
   Zacatek := 1;
   for Pozice:=1 to N do begin
      if Navstivena[Cesta[Pozice]] >= Zacatek then Zacatek := Navstivena[Cesta[Pozice]] + 1;
                                                   {je potřeba nejdelší úsek zkrátit?}
      if Pozice - Zacatek + 1 > Nejdelsi_Delka then begin
         {úprava nejdelšího nalezeného úseku je-li třeba}
         Nejdelsi_Zacatek := Zacatek;
         Nejdelsi_Delka := Pozice-Zacatek + 1;
      end;
      Navstivena[Cesta[Pozice]] := Pozice; {opravení poslední návštěvy aktuální křižovatky}
   end;
   for Pozice := Nejdelsi_Zacatek to Nejdelsi_Zacatek + Nejdelsi_Delka - 1 do write(Cesta[Pozice],' ');
end.
```

Výsledková listina dvacátého třetího ročníku KSP po čtvrté sérii

		Škola	*ročník*	*sérií*	*2341*	*2342*	*2343*	*2344*	*2345*	*2346*	*2347*	*série*	*celkem*
1.	Jakub Zíka	GNAlejíPH	4	4	11			10	11		18	50,6	185,5
2.	Vojtěch Hlávka	GŠlapanice	2	9	9	10	9	9,5	9	9	15,9	45,1	174,4
3.	Lukáš Folwarczný	GKomHavíř	3	5	11			10	11		13,5	47,5	170,5
4.	Juda Kaleta	GKlatovy	2	5	10		8		8		7,5	39,2	166,2
5.	Peter Zeman	GAnVra	4	4	12	4		9,5			9	40,5	149,3
6.	Matěj Kocián	GLesníZlín	4	6			10	9,5	11		14	45,8	145,4
7.	Martin Raszyk	G_Karvina	1	4		4	8		11	6	15,8	42,5	144,7
8.	Vojtěch Sejkora	SPSE_Pard	2	4			2	10	8	9	7,1	39,4	144,4
9.	Filip Hlásek	GMikulášPL	4	19					11			11,0	130,2
10.	Jan Bok	GJungmanLT	4	5	10		8		11	9		39,1	123,1
11.	Štěpán Šimsa	GJungmanLT	2	12	12	4			11		15,4	42,0	121,6
12.	Michal Anderle	GTim_Lučen	4	3								0,0	114,8
13.	Ondřej Hübsch	GArabskáPH	1	9			10	9,5	11			30,6	114,3
14.	Jindřich Pilař	GBroumov	3	5		3,7	10	8		9		33,7	113,9
15.	David Bernhauer	GZborovPH	3	3								0,0	109,9
16.–17.	Ondřej Fiedler	GJungmanLT	4	5	10		8		7	9		36,9	107,0
	Michal Pokorný	SŠkybernHK	3	4			2			9		12,6	107,0
18.	Ondřej Cífka	GNAlejíPH	2	6			8	10		9	14,9	42,5	102,0
19.	Jan Hadrava	GZborovPH	3	3								0,0	99,8
20.	Matouš Kozma	BiGyBBHK	4	3		6,7	10			9	6,6	38,4	88,6
21.	Ondřej Mička	GJírovcČB	2	8	10		8				0,7	19,6	85,4
22.	Jerguš Greššák	GRaymanaPV	2	4								0,0	82,6
23.	Vojtěch Kletečka	GHavlBrod	3	3								0,0	71,2
24.	Daniel Stahr	GJungmanLT	4	6			10					10,0	69,4
25.	Jiří Setnička	G25březnPH	4	14			2			9	6,3	15,4	62,3
26.	Jonatan Matějka	GJírovcČB	1	7			8	9			3	21,9	48,3
27.	David Krška	GJirsíkaČB	4	2					4			7,1	47,4
28.	Andrej Mariš	PriorPC	3	2								0,0	45,2
29.	Jiří Eichler	SlovanGOL	3	7								0,0	43,9
30.	Matěj Židek	GBroumov	3	5			2		4			9,4	43,4
31.	Jan Paštyka	SPSKutHora	2	2								0,0	39,8
32.	Rastislav Rabatin	GJHroncaBA	2	1								0,0	37,4
33.	Filip Matzner	GJirsíkaČB	4	2								0,0	34,5
34.	Jan Škoda	GMikulášPL	4	6	9		8					18,3	34,0
35.	Jitka Fürbacherová	GKlatovy	2	3			2			3		9,0	33,3
36.	Tereza Hulcová	GKlatovy	2	3			2					3,8	32,7
37.	Robin Mana	GValašKlob	4	2								0,0	32,3
38.	Milan Berka	G_Krumlov	4	1								0,0	29,8
39.	Mária Mrocková	GJHroncaBA	4	4								0,0	29,0
40.	Daniel Švec	SPŠERožnov	3	1								0,0	27,9
41.	Jakub Kulhan	G_Kralupy	3	1								0,0	27,3
42.	Michal Punčochář	GJírovcČB	1	1								0,0	24,2
43.	Dominik Smrž	GOhradníPH	1	6		4	8	10				23,7	23,7
44.	Tomáš Varga	GMost	-1	2								0,0	22,0
45.	Tomáš Jareš	PORGPha	0	1		1		6	7			20,8	20,8
46.–48.	Anna Dresslerová	GJHroncaBA	4	2								0,0	19,0
	Milan Mikuš	GĽŠtúraTN	3	2								0,0	19,0
	Filip Štědronský	GMikulášPL	4	1								0,0	19,0
49.	Tomáš Velecký	GBezručeFM	0	2								0,0	17,3
50.	Jiří Šebele	GArabskáPH	1	1								0,0	14,3
51.	Pavel Kratochvíl	VOŠGSvětlá	3	11								0,0	14,1
52.–53.	Martin Mach	GJírovcČB	3	4								0,0	10,0
	Alexander Mansurov	GNVPláníPH	2	4								0,0	10,0
54.	Josef Klesa	GKlatovy	3	1								0,0	9,5
55.	Martin Holec	GSlavičín	4	8								0,0	8,7
56.	Jan Lejnar	GKlatovy	1	1								0,0	8,6
57.	Tomáš Turlík	GRaymanaPV	2	1								0,0	8,4
58.	Petr Pecha	SPŠsVsetín	4	10								0,0	7,2
59.	Barbora Hourová	G_Brandýs	4	1								0,0	5,7
60.	Patrik Jung	GKlatovy	1	1								0,0	4,5
61.	Radim Cajzl	GNoMěsNMor	4	23								0,0	1,7