Výsledková listina třetí série dvacátého sedmého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*3-1*	*3-2*	*3-3*	*3-4*	*3-5*	*3-6*	*3-7*	*série*	*celkem*
0.					14	12	13	12	10	9	15	66,0	182,0
1.	Jan Špaček	G Wicht	4	8	13	10	13	12	10	9	13	62,5	173,9
2.	Stanislav Lukeš	GPísnickáPH	2	4	13	9	13		4	7	7,5	55,5	161,7
3.	Marek Černý	G Chrudim	4	8	6	12	3	3	10	9	10	49,7	160,7
4.	Richard Hladík	GOAMarLaz	2	13	7	10	10	12	10		12,5	52,9	160,3
5.	Martin Scheubrein	G MNám Třb	3	3	8	5		2	2	1	11,5	39,3	140,6
6.	Jan Tománek	GPelhřimov	4	4	10	7	3	0	5	5	2	34,2	135,9
7.	Jakub Tětek	CírkG Plzeň	1	3	6,5	6			0	8	0,5	27,2	133,6
8.	Václav Šraier	GČeskoliPH	2	3	8	6			10	4	6	46,3	127,0
9.	Štěpán Hudeček	G Litovel	3	3			12		10	4	11,5	42,7	125,4
10.	Přemysl Šťastný	GŽamberk	2	6					9	3	7,5	21,9	121,3
11.	Lukáš Ulrich	SSŠVTPraha	4	3		4	0	0	8	2	10	31,9	114,5
12.	Jan Kočur	G Wicht	4	3	1		0,5			5	9,5	23,3	113,4
13.	Adrián Goga	SPŠNitra	4	3			0				3	6,0	110,8
14.	Václav Volhejn	GKepleraPH	2	13		10		4	2	5		19,3	109,2
15.	Jan Knížek	G Strakon	4	16						6	11	13,5	107,7
16.	Jakub Zárybnický	GTomkovaOL	4	8		6				6	7	22,5	102,7
17.	Michal Převrátil	GKlatovy	2	3		8		3	10	2		28,3	102,4
18.	Anna Gajdová	GFPValMez	4	4								0,0	97,3
19.	Róbert Selvek	G KošiceS	3	3					2			2,0	96,4
20.	Michal Töpfer	G DrJPekMB	2	3	0,5	2			2	4	5,5	19,6	96,3
21.	Jan Gocník	GJŠkodyPŘ	3	3					10			10,0	74,1
22.	Jiří Sejkora	GVoděraPH	3	3							5	8,7	68,7
23.	Pavel Turinský	G Brandýs	2	3		6					14	23,7	65,4
24.	Jiří Vozár	G UherBrod	3	3				1			8	13,3	57,5
25.	David Cholewa	GMatOS	4	2								0,0	46,2
26.	Martin Zoula	GNadKavaPH	3	3						1		2,2	38,6
27.	Jan Bouček	GKepleraPH	2	2								0,0	33,2
28.	Václav Končický	GSOŠ FrMís	4	2								0,0	32,9
29.	Jan Pokorný	G_Bučovice	3	5								0,0	31,5
30.	Barbora Sedláková	GKonštanPV	4	2								0,0	27,4
31.	Jan Soukup	GKlatovy	4	2				12	10			22,0	22,0
32.	Filip Bialas	GOpatovPHA	2	5								0,0	20,0
33.	Vít Macura	GOAMarLaz	2	2								0,0	18,9
34.	Jakub Lukeš	GNAlejíPH	2	1								0,0	14,0
35.	Václav Rozhoň	GJirsíkaČB	4	8				12	0			12,0	13,6
36.	Dalimil Hájek	GKepleraPH	4	15								0,0	13,4
37.	Martin Kubeša	GJŠkodyPŘ	3	1								0,0	12,8
38.	David Juřica	GNadŠtolPH	2	2								0,0	10,9
39.	Jan Kaifer	GČesBrod	−1	1								0,0	10,6
40.–41.	Matěj Konečný	GJírovcČB	4	2								0,0	10,0
	Jakub Matěna	GČeskoliPH	3	1					10			10,0	10,0
42.	Jan Burda	G_Holice	1	1								0,0	9,0
43.	Václav Steinhauser	GDačice	1	1								0,0	7,9
44.	Josef Vávra	SJec	4	1								0,0	5,7
45.	František Dostál	VSPŠEOc	4	1								0,0	4,0
46.	Michael Novák	SSŠVTPraha	4	1								0,0	2,0

Chcete-li s námi komunikovat šifrovaně a bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: `0E:D9:B6:E5:6F:B0:51:D9:66:EB:E9:29:E4:58:AB:5F:99:D6:FD:A3.`

Korespondenční Seminář z Programování

27. ročník KSP Únor 2015

Milí řešitelé a řešitelky!

Věříme, že jste pololetí zakončili stejně dobře, jako orgové zvládají zkouškové (ne-li lépe), a proto si od školních testů rádi odpočinete a zapřemýšlíte nad některými z úloh této série. Čeká vás další příběh na téma fatální programátorské chyby, seriál o UNIXu a programátorská kuchařka.

Připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme propisku, blok a tužku. To vše s logem KSP.

Za úspěšné řešení KSP je možno být přijat na MFF UK bez přijímacích zkoušek. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % bodů (tedy alespoň 150 z maxima 300 bodů). Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě.

Termín série: **Pondělí 30. března 2015 v 8:00 SELČ**
Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*
Informace: Další podrobnosti o fungování KSP naleznete na *http://ksp.mff.cuni.cz/*. Pokud budete mít jakoukoliv otázku, neváhejte se nás zeptat na *ksp@mff.cuni.cz* nebo na našem fóru. Přejeme hodně štěstí! ;-)
Odměna série: **Čokoládu** pošleme každému řešiteli, který vyřeší **tři úlohy** této série tak dobře, že **za každou z nich získá alespoň jedenáct bodů**.

Čtvrtá série dvacátého sedmého ročníku KSP

Stejně jako v předchozích sériích, i v této budeme věnovat pozornost programátorské chybě, která měla, i přes svoji zdánlivou nevinnost, nedozírné důsledky. Situaci, o níž bude dnes řeč, nahrála i lidská nedbalost a kvůli tomu se mohla projevit jedna z nejzákeřnějších softwarových chyb, jež je oříškem i pro ostřílené programátory.

Celý příběh se odehrál na severovýchodě USA, v jednom horkém srpnovém dni roku 2003. Přesuňme se teď do dispečinku firmy FirstEnergy, amerického dodavatele elektřiny, kdesi v severním Ohiu…

***

Frank, jeden z operátorů na směně, odběhl z řídicí místnosti do kuchyňky a vzal lahev s vodou nejen pro sebe, ale také pro svého kolegu. „Díky moc," vydechl lehce obtloustlý Denis a otřel si z čela pot. Třicetistupňové teploty nebyly nic pro něj a celou směnu se snažil nastavit ventilátor tak, aby vanul přímo do jeho tváře.

Snad každá kancelář na severovýchodě Států teď měla zapnutou klimatizaci, čemuž odpovídala zvýšená spotřeba energie. Bylo potřeba zajistit její stabilní přísun. Frank už od rána mnohokrát upravoval parametry rozvodné sítě a několikrát žádal jižněji položené elektrárny o jejich nevyužitý výkon. Výstražný systém ohlašující každý problém se teď naštěstí na chvíli odmlčel a Frank měl čas se podívat na úkoly od svého nadřízeného. Díky novému systému, který ve firmě zavedli, jich naštěstí nebylo tolik.

27-4-1 Zadávání úkolů — 10 bodů

Ve FirstEnergy je přesně určena organizační struktura. Každý zaměstnanec v pozici vedoucího má právě dva podřízené (kteří mohou, ale nemusí být vedoucími), ostatní zaměstnanci na nikoho nedohlížejí. Jeden zaměstnanec může mít více nadřízených, v celém schématu však existuje právě jeden ředitel. Ředitel je vedoucí, který nemá žádné nadřízené a všichni ostatní zaměstnanci mu jsou (alespoň nepřímo) podřízeni. Hierarchii bychom tedy mohli označit jako souvislý acyklický hranově orientovaný graf (DAG).

Pouze ředitel může zadávat úkoly, každý úkol předá jednomu ze svých podřízených. Ten, pokud není vedoucím, musí úkol provést, jinak jej opět předá jednomu ze svých podřízených, a tak dále. Aby byly úkoly rozdělovány rovnoměrně, každý vedoucí (ředitele nevyjímaje) je musí předávat střídavě jednomu a pak druhému podřízenému.

O každém vedoucím víte, kterému ze svých dvou podřízených předá první úkol, který k němu dorazí. Ředitel zadá celkový počet N úkolů, které jsou postupně předávány celou hierarchií. Váš úkol se týká momentu, kdy jsou všechny tyto úkoly vykonány. Určete pro každého vedoucího, kterému ze svých podřízených by předal další úkol, který by k němu dorazil. Pokuste se, aby vaše řešení bylo efektivní i pro velké hodnoty N (velkou hodnotou myslíme například bilión).

Na obrázcích vidíte příklad takové hierarchie. Šipky vyznačují, komu bude nový úkol předán. Na prvním obrázku je znázorněna výchozí situace, na tom druhém stav po 99 zadaných úkolech:

99

Lehčí varianta (za 5 bodů): Navrhněte řešení pro případ, kdy má každý zaměstnanec jen jednoho nadřízeného (grafem hierarchie je tedy strom).

Místností se rozezněl zvuk telefonu. Frank se natáhl a zvedl sluchátko. „Nazdar človeče, tady je MISO," ozvalo se. „Teď jsme zaznamenali výpadek vedení Star-South Canton. Jenom na malou chvíli, už zase běží. Všimli jste si toho?"

Organizace MISO koordinovala tok elektřiny mezi sítěmi jednotlivých společností. Frank sjel pohledem na obrazovku počítače a potřásl hlavou. „Tady nic nevidíme, alespoň chvíli tu máme klid. Není někde u vás chyba?“ zvědavě se zeptal. „Hm… Máme tu nový software. Ještě se na to podívám,“ řekl operátor nejistě a zavěsil.*

Není čeho se bát, pomyslel si Frank. Před dvěma hodinami přestalo fungovat 345kV vedení Stuart-Atlanta, směřující na jih – kvůli velkému průtoku proudu se dráty mezi sloupy začaly prověšovat a protože společnost nechala pod vedením přerůst stromy, vodiče se dotkly jejich špiček a došlo ke zkratu. Zátěž se však rozložila na jiná vedení a vše stále fungovalo stabilně. Rozvodná síť v USA byla od počátku navržená tak, aby ji takový výpadek nemohl rozhodit.

27-4-2 Čtverce v síti — 11 bodů

Projektanti rozvodné soustavy se rozhodují mezi několika návrhy rozmístění vedení. Aby určili míru spolehlivosti návrhu, potřebují zjistit, kolik se v něm dá najít různých čtverců, které jsou složené z vodičů.

Navrhněte algoritmus, jenž na vstupu obdrží přímky představující vedení, zadané v některém z běžných tvarů (např. obecnou rovnicí, případně pomocí dvou bodů), a určí, kolik navzájem různých čtverců tyto přímky tvoří.

Počítáme i vzájemně se překrývající čtverce, tudíž přímky na obrázku tvoří celkem osm čtverců.

V dozorně organizace MISO se Leonard, vedoucí směny, nedůvěřivě podíval na elektronickou mapu. Před chvílí došlo k neplánovanému odstavení jednoho z bloků jaderné elektrárny a mnoho linek se začalo barvit ze zelené do jemně žluté barvy. Jeden z pracovníků se po telefonu bavil s jiným dispečinkem o nejlepším řešení situace.

Mapa v reálném čase vykreslovala data vypočtená stavovým estimátorem. Program zachycoval údaje ze senzorů na vedení a počítal zatížení linek. Šlo o velice užitečný systém, ušetřující pracovníky od spousty výpočtů, ovšem za pouhé dva týdny, po které byl nainstalovaný, neměl vychytané všechny mouchy. Některé ze senzorů na něj ještě nebyly přímo napojené, a jejich stav bylo třeba aktualizovat ručně. Je to stejně daleko jednodušší než před dvaceti lety, utěšoval se Leonard. Vzpomněl si na jednoho ze starších techniků, s nímž se před týdnem bavil o martýriu při zjišťování napětí v síti.

27-4-3 Vysoké napětí — 11 bodů

Pracovníci dispečinku potřebují zjistit, jaké napětí se vyskytuje na různých bodech sítě. Sice ví, že v uzlových bodech soustavy se vyskytují jen tři různé hladiny napětí – 0, 100 a 200 kilovoltů, ale informace mají pouze ze senzorů na vodičích, které tyto body spojují a které měří rozdíl napětí mezi koncovými body (nevíme však, kde je napětí větší, jinými slovy, ze senzorů vyčteme pouze absolutní hodnotu rozdílu).

K dispozici jsme dostali mapu sítě (tvořenou uzlovými body a vodiči – jinde než v bodech se vodiče nekříží) a pro každý vodič hodnotu rozdílu napětí mezi uzly, které spojuje (buď 0, 100, nebo 200 kV). Máme za úkol určit hodnoty napětí na koncových bodech, aby rozdíly na vodičích odpovídaly (pokud je více možných řešení, stačí nalézt jedno z nich), nebo zjistit, že došlo k chybě a žádné takové řešení neexistuje.

Příklad: Vrcholům grafu na obrázku (hodnoty jsou ve stovkách kilovoltů) napětí přiřadit lze. Jedno možné přiřazení napětí vrcholům je zakreslené přímo do obrázku.

Lehčí varianta (za 7 bodů): Vyřešte úlohu pro dvě napěťové hladiny v bodech, 0 a 100 kV.

Nepříjemně hlasitý zvuk alarmu přerušil tok Leonardových vzpomínek. Okamžitě přiběhl k nejbližšímu ovládacímu pultu. „Vypadlo nám jedno z hraničních vedení, 345 kilovoltů,“ ukázal mu jeden z dispečerů na obrazovce místo poruchy. „Jak to?“ Leonard překvapením pozvedl obočí. „Běželo na osmdesáti procentech zátěže…“ rychle uvažoval, jakým přičiněním k tomu mohlo dojít.

A náhle mu na mysl přišla jedna událost z dopoledne. Stuart-Atlanta, nyní vypojené vedení a jedno z páteřních v celé oblasti, patřilo k těm spojením, se kterými estimátor počítal nepřímo, jen přes manuální zadávání stavů! „To snad ne,“ zamumlal a rychle se přepnul na obrazovku systému. A navzdory sytě zelené barvě, kterou bylo Stuart-Atlanta značené, ho polil mráz.

„Máme chybu v systému,“ zvolal vyděšeně. „Stavový estimátor nám počítá nesmysly!“ Dispečerům okamžitě došlo, jakou chybu udělali, jeden z nich usedl k počítači a zadal správné hodnoty. Během asi minuty, kdy probíhaly nové výpočty, přemýšlel Leonard, jakým způsobem by šlo síť, teď už notně oslabenou, odlehčit. Tak jako je tomu při řešení hlavolamů, byla v tuto chvíli klíčová schopnost vhodně kombinovat…

27-4-4 NP-úplný hlavolam — 11 bodů

Hlavolam, se kterým si budeme hrát v této úloze, sestává z $R \times S$ políček uspořádaných do mřížky s R řádky a S sloupci. Některá z políček jsou průhledná, ostatní jsou neprůhledná.

Pod sloupce mřížky můžeme zasouvat barevné proužky. Zasuneme-li proužek pod zvolený sloupec, pak všechna průhledná políčka tohoto sloupce uvidíme nyní jako barevná. U každého řádku je číslo – nula nebo jednička, které udává, kolik barevných políček má být v tomto řádku vidět.

Na následujícím obrázku vidíte dva hlavolamy. Průhledná políčka jsou znázorněna bíle, neprůhledná políčka šedivě. Řešením prvního hlavolamu je zasunutí proužků do druhého, třetího a pátého sloupce. Druhý hlavolam řešení nemá.

* Midcontinent Independent System Operator

ným způsobem velký soubor známé velikosti. Ve velkém počtu bloků by se i jedničkový rozdíl nasčítal, takže by počet bloků neseděl přesně.

Doplníme, že jeden diskový blok často mívá velikost 4 kB. Poučení z úlohy mělo být, že reálná velikost diskového bloku a velikost bloku používaná utilitami spolu nemají nic společného. Není blok jako blok.

Jenže se to zkomplikovalo, když jste začali hledat a používat utility, které nepracují s diskem přes souborový systém, ale koukají na něj přímo. A tak se přestaňme schovávat za představu ideálního světa, kde existuje něco jako diskový blok, a povězme si, jak se to s bloky na disku má doopravdy.

Bloků existuje víc různých typů, konkrétně jste narazili na tři. Fyzický sektor, logický sektor a alokační jednotka (neboli cluster). Nakonec jako bonus přidám ještě IO blok.

Začněme od nejnižších vrstev abstrakce, u fyzického sektoru. To je blok, který interně používá pevný disk a jeho firmware. Vůbec by nás jako uživatele nemusel zajímat, kdyby jeho velikost nesouvisela s výkonem programů, které s diskem intenzivněji pracují. U starších disků má velikost 512 B, u novějších (od roku 2011) 4096 B. Formátu novějších sektorů se říká Advanced Format (AF) – pod tímto termínem vygooglíte víc.

O vrstvu abstrakce výš leží logický sektor. V nich disk přemýšlí, když komunikuje se zbytkem počítače, typicky přes svůj ovladač v operačním systému. U starších disků se nelišil od fyzického, AF podle něj definuje dvě kategorie zařízení. První je 512e (e jako emulace) s 512B logickým sektorem, druhá 4Kn (n jako nativní) s 4096B logickým sektorem. Kategorie 512e se snaží být aspoň navenek zpětně kompatibilní, uvnitř už ale vyzvedává a ukládá celé 4kB fyzické sektory.

Teprve nad ovladačem disku sedí souborový systém a diskový prostor dělí na alokační jednotky, čili clustery. Po takových blocích přiděluje paměť jednotlivým souborům, a právě na ně se snažil první úkol seriálu mířit. Velikost alokační jednotky už není zadrátovaná v hardware, bývá možné ji nastavit při vytváření souborového systému, ale zejména kvůli efektivitě se obvykle volí jako celočíselný násobek velikosti sektoru.

Pro úplnost uvedeme ještě čtvrtý typ bloku, na který jste už nenarazili. Operační systém má nějaké povědomí o tom, jak by se na disk mělo přistupovat, s jak velkými bloky by se ideálně mělo pracovat najednou. Říká se jim IO bloky (I je input, O output). Dosud jsme psali především o klasických pevných discích. Pro ně mívají stejnou velikost jako alokační jednotka souborového systému, vnitřní struktura SSD disků ale vyžaduje IO bloky zpravidla mnohem větší, aby se zbytečně nepřepisovaly velké kusy paměti a nezkracovala se tak životnost zařízení. Program intenzivněji pracující s diskem si pak může přečíst, po jakých blocích je dobré číst, nebo si to může nechat nadiktovat od uživatele, jako to dělá třeba utilita `dd` pro čtení a zápis po blocích. Je prastará a na nestandardní syntaxi jejích parametrů je to vidět, nenechte se vyděsit. :-)

Když se začnete vrtat ve specifikách jednotlivých souborových systémů, najdete džungli specialit, na které běžně ve svém počítači nenarazíte. Více detailů si v zájmu zachování vašeho duševního zdraví dovolíme nerozebírat. Pokud si ho nevážíte, koukněte na pojmy block suballocation, sparse file, inlining nebo na souborový systém jménem ZFS.

Další debatu s vámi rád povedu na fóru, ať už se bude týkat vašich řešení, tohoto textu, UNIXu nebo disků a souborových systémů.

Samé příjemné zážitky s UNIXem přeje

Tomáš „Palec“ Maleček

Ve složitějším programu by bylo nepříjemné, že se ve smyčce nejde rozumně podívat na tokeny před a za aktuálně zpracovávaným. To by se dalo napravit použitím jiné smyčky. Příkaz `read` může opakovaně odkousávat první token; smyčka se zastaví, když zbytek řádku ke zpracování už je prázdný.

```
while [ -n "$line" ]; do
        IFS=' ' read -r token line <<EOF
$line
EOF
        # tělo cyklu
done
```

Tuto syntaktickou vlastnost shellu možná vidíte poprvé. Slovo `<<EOF` je přesměrování vstupu. Vstup se bere z následujících řádek, až do slova `EOF`, které je úplně samo na řádku (nesmí předcházet ani bílé znaky!). Říká se tomu *here document*. Místo `EOF` je možné použít i jiný řetězec. Na začátku (za `<<`) může být v uvozovkách, v tom případě může obsahovat i mezery, na konci je ale vždycky bez uvozovek. Při použití jednoduchých uvozovek (`<<'EOF'`) se celý here document chová jako řetězec v jednoduchých uvozovkách, tedy se v něm neexpandují proměnné ani se neprovádí substituce příkazů.

Kdybychom se chtěli použití here document vyhnout, jde to, ale není to pěkné:

```
r() { IFS=' ' read -r token line; }
token="$(echo "$line" | { r; echo "$token"; })"
line="$(echo "$line" | { r; echo "$line"; })"
```

Opět si uvědomte, co se stane při vynechání složených závorek, co je za zádrhel s readem v pipeline, jak se v pipeline chová `exit`.

Trochu potíží nadělá ještě vynechávání prázdných řádků. Projděte si vzorový kód a rozmyslete si, že a jak funguje. Všimněte si použití podmínek a příkazů `true` a `false`. Podotýkám, že pro vypsání stringu bez konce řádku na `echo -n` všude spoléhat nejde. Příkaz `printf` je portabilní, chová se předvídatelně a POSIX ho doporučuje používat místo `echo`.

Úkol 6 – Ztabulkování /etc/passwd

O formátu `/etc/passwd` jste si měli přečíst v `man 5 passwd` nebo analogickém zdroji. Pomocí standardních prostředků si s úlohou můžete snadno poradit za použití `while`, `read`, `IFS` a `printf`.

Použití tabulátorů jako oddělovačů sloupců nestačí, protože pak se tabulka rozpadne, když se liší délky položek jednoho sloupce po vydělení osmi (rozestup tabulačních zarážek). Předpokládal jsem, že si víc přečtete o `printf` a jeho formátovacích direktivách, jak napovídalo i umístění úlohy. Najít jste potřebovali nastavení šířky, na jakou se string má doplnit mezerami. Na plný počet bodů stačilo maximální délku položky v každém sloupci odhadnout konstantou, obludně dlouhý login klidně smí tabulku rozbít. Šlo by šířku sloupců i počítat, ale to za druhý průchod souborem a dost kódu navíc nestojí.

```
tplt='%-13s\t%-1s\t%5s\t%5s\t%-35s\t%-25s\t%s\n'
while IFS=: read login passwd uid \
        gid name home shell; do
        printf "$tplt" \
                "$login" \
                "$passwd" \
                "$uid" \
                "$gid" \
                "$name" \
                "$home" \
                "$shell"
done < /etc/passwd
```

Backslash na konci řádku dovolí v příkazu pokračovat na dalším řádku a logický řádek tak fyzicky rozdělit. Nesmí za ním být do konce řádku už nic jiného, ani bílé znaky.

Vzorové řešení už znáte, ale zmíním ještě řešení, kterým jste mě překvapili. Není portabilní, zato na úlohu padne jako ulité a za celkem rozšířené se rozhodně dá považovat. Je jím příkaz `column`, který slouží k výpisu dat do tabulky. Bez parametrů považuje vstup za seznam jednotlivých položek a řádek může ukončit mezi libovolnými dvěma z nich, s přepínačem `-t` respektuje řádkové zlomy ve vstupu a může nechávat v tabulce na koncích řádků prázdné buňky. Přepínač `-s` nastavuje oddělovač sloupců na vstupu a `-n` zakazuje ignorování prázdných buněk.

```
column -t -s: -n /etc/passwd
```

Úkol 1 – Velikosti bloků

Jako poslední nám zbývá první úkol. Zadání se nepodařilo zformulovat tak, abyste se ho nesnažili řešit cestami, které vás protáhnou detaily současných disků. Seriál tím směrem nemířil, ale cestu do pekel si mnozí z vás našli sami.

Jak jsem doufal, že k řešení přistoupíte? Já bych si zakládal soubory různých velikostí a zkoumal na nich výstup příkazu `du` s přepínačem `-h` a bez něj. Vyrobit takový soubor velikosti N bytů můžeme třeba příkazem `head -c N /dev/zero > soubor`, jak bylo ukázáno v textu seriálu.

Pokud tedy nemáme zapnutý inlining, o kterém jsem také psal, vlastně stačí soubor velikosti 1: `echo > soubor`. Ten určitě bude zabírat nejmenší možné nenulové množství paměti, tj. jeden diskový blok. (Prázdný soubor by často zabral jen inode, který se do počtu zabraných bloků nepočítá.) S inliningem by bylo potřeba soubor postupně zvětšovat a někde ve větších velikostech hledat velikost, při které přidání jediného bytu zvětší počet bloků zabraných souborem. Pak by stačilo odečíst velikost před zvětšením.

Příkaz `du -h soubor` nebo lépe `du -B 1 soubor` nám ukáže, jak velký takový blok je. Když použijeme `du soubor` bez přepínačů, dozvíme se, kolik bloků soubor podle představ utility `du` zabírá, z čehož už snadno dopočítáme, jak velký blok utilita používá.

Na Linuxu to bývá 1024 B. Ovšem když nastavíme proměnnou prostředí `POSIXLY_CORRECT` (na libovolnou hodnotu), `du` se přizpůsobí požadavkům POSIXu a začne používat bloky o velikosti 512 B. Není bez zajímavosti, že tato proměnná mívala alias `POSIX_ME_HARDER`, což ilustruje názor autorů na některé části normy. Zajímavé čtení o této proměnné[16] najdete na stránce Waikatské skupiny uživatelů Linuxu.

Utilita `ls` tu vůbec nebyla potřeba. Spousta z vás se snažila využít její přepínač `-s` a zjišťovala tak velikost bloku používaného utilitou `ls` namísto toho používaného utilitou `du`.

Kdybychom chtěli vyloučit i možnost obskurní velikosti bloku, která není mocninou dvojky, můžeme pozorovat stej-

[16] `http://wiki.wlug.org.nz/POSIXLY_CORRECT`

		■		■	1
	■	■	■	■	0
■	■	■			1
■		■		■	1
■	■		■	■	1
	■	■	■		1

		■	■	■	1
■			■	■	1
■	■			■	1
■	■	■			1
	■	■	■		1
■	■	■	■	■	0

Po vás však nechceme návod na řešení hlavolamu. Raději dokažte, že úloha rozhodování toho, zda má zadaný hlavolam řešení, je NP-úplná. Pokud se vám to nebude dařit, dokažte alespoň NP-úplnost této úlohy pro obecnější hlavolamy, kdy u každého řádku může být požadován libovolný počet barevných políček, nikoliv pouze nula nebo jedna.

Na dispečink FirstEnergy mezitím dorazili technici. Jejich provinilý výraz ve tváři bylo snad to jediné, co Frankovi zabránilo, aby je popadl za límec košile a začal s nimi třást. Po téměř hodině od poslední zprávy výstražného systému a mnoha telefonátech od sousedících společností, které je upozorňovaly na blížící se nebezpečí kolapsu, byla zřejmá závažnost celé situace.

„Spadl nám server, asi před půlhodinou,“ přiznal jeden z techniků. „Moc jsme tomu nevěnovali pozornost, jenom jsme zkontrolovali, že běží záložní počítač. Ale… Ten před deseti minutami… spadl taky.“ „Takže proto nám všechny programy tady běží hlemýždí rychlostí?“ zeptal se Frank. Jindy celkem svižný systém najednou potřeboval na získání nových informací téměř minutu. „Jistě,“ přikývl technik, „ale je tu ještě jedna věc. Až po pádu toho druhého serveru jsme zjistili, že… totiž… nefunguje výstražný systém. Snad víc než hodinu,“ vyklopil ze sebe.

Denis mezitím skončil telefonický rozhovor. „Zase MISO,“ oznámil. „Nezbývá nám prý nic jiného, než snížit napětí u odběratelů. Nějak se to začíná hroutit!“ Frank přikývl, šlo o nouzové, ale stále poměrně rozumné řešení. Jenže – „jak se to dělá?“ zeptal se Denise. „Naposledy jsme to dělali – kdy vůbec?“ odpověděl Denis a poděšeně se podíval na techniky, kteří jen rezignovaně pokrčili rameny.

***

V MISu už bylo jasné, že se situace stává kritickou. Estimátor konečně spočítal skutečný stav sítě a ten byl daleko méně optimistický než předtím.

V jedné chvíli se těsně za sebou vypojilo pět vedení, jednoduše proto, že procházející proud byl až příliš velký. Další elektrárny hlásily odstavení a začala se objevovat první místa kompletně odpojená od elektřiny. Neustále se ozývalo zvonění telefonů, to jak se operátoři snažili odpojit nestabilní soustavu od ostatních částí.

A pak, několik minut po čtvrté hodině, se vše začalo hroutit jako dominové kostky. Během několika vteřin vypadly všechny zbývající linky dopravující elektřinu do Ohia – bylo jich tolik, že Leonard ani nestačil číst jejich označení na displeji – elektrický proud si našel cestu přes Pensylvánii a New York a během mžiku odstavil většinu tamější sítě, spolu s elektrárnami. Padesát milionů lidí se ocitlo bez elektřiny. Provoz vlaků a hromadné dopravy ve městech byl přerušen.

***

Přestože na některých místech ke zprovoznění stačilo několik hodin, celá rozvodná soustava fungovala až po několika dnech. Město New York, které na elektřinu čekalo do tří hodin ráno, se stalo symbolem celého výpadku. Fotografie davů lidí jdoucích pěšky po Brooklynském mostě, dopravních kolapsů nebo potemnělého Manhattanu, ozařovaného pouze svitem zapadajícího slunce, obletěly celý svět. Naštěstí se nevyskytly případy rabování, došlo ale k několika požárům vzniklých od zapálených svíček. Protože byl horký letní den, většina restaurací začala nabízet své jídlo komukoliv, kdo o ně požádal, protože by se bez chlazení stejně zkazilo. Když Leonard později v televizi sledoval reportáž se záběry na techniky opravující elektrickou síť, kteří přišli v montérkách do luxusně vypadající restaurace a konzumovali očividně drahé pokrmy, nemohl se tomu nezasmát.

27-4-5 Večeře pro opraváře — 12 bodů

Elektrotechnik Larry strávil celé odpoledne zjišťováním poruch vedení ve městě, a jelikož mu je jasné, že se práce dnes protáhnou do pozdních nočních hodin, chce se na takový úkol pořádně navečeřet. Do zjednodušené mapy Manhattanu (čtvercová síť s vyznačenými budovami, policisty na křižovatkách a ostatními věcmi, které nelze procházet; všechna ostatní pole ano) si proto zakreslil hospody, restaurace, ba i zmrzlinářství, kde se chce najíst. Je mu ale jasné, že času není nazbyt a že zásoby jídla nejsou bezedné.

Proto by potřeboval najít nejkratší trasu chůze po Manhattanu, během níž všechny zakreslené podniky navštíví. Můžete předpokládat, že podniků nebude více než dvacet. A pospěšte si, zmrzlina už teče!

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte. Přesný popis formátu společně s ukázkovými vstupy nalezenete ve webové verzi zadání.

Okamžitě po opětovném spuštění soustavy se rozběhlo vyšetřování, které odhalilo množství lidských i technických chyb. Dispečeři MISa byli kritizováni za spoléhání se na systém, jehož části byly teprve ve vývoji. FirstEnergy nedostatečně proklesťovala stromy pod vedením: to kvůli zkratům vypadávalo i při mírně nadprůměrném zatížení. Navíc nedostatečně školila obslužný personál: jinak by nedošlo k tomu, že by dispečeři nevěděli, jak ručně snížit napětí u odběratelů.

Ale co zhroucení serverů a výstražného systému? Aby vyšetřovatelé zjistili, v jakém stavu se programy nacházely před tím, než přestaly fungovat, začali procházet jejich záznamy. Ty se ve společnosti stále ještě ukládaly na magnetické pásky.

27-4-6 Stěhování pásek — 11 bodů

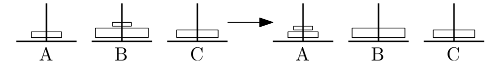
Pásky s důležitými daty jsou namotány na ploché kotouče, které však mají různý průměr. Ve FirstEnergy se k ukládání záznamů používá poměrně malá místnost, kde jsou kotouče položeny na sebe a seřazeny takovým způsobem, že ten největší je vespod a ten nejmenší navrchu.

Problém takovéhoto věžovitého uspořádání spočívá v obtížném stěhování. Před několika týdny museli pracovníci všechny kotouče přesunout, aby se do skladu vešla i část firemního archivu. Aby se pásky nepoškodily, museli postupovat podle striktních pravidel: v jedné chvíli mohli přesouvat pouze jeden kotouč (pokud je víc kotoučů na sobě, mohou přesunout pouze ten, který je navrchu), a aby se věž složená z kotoučů nezhroutila, nesměli položit kotouč s větším průměrem na kotouč s průměrem menším.

Protože je sklad skutečně stísněný, kromě místa označeného `A`, kde se kotouče vyskytovaly nejdříve, a místa `B`, kam byly přesunuty, mohli pracovníci kotouče odkládat pouze

na jedno další místo C – všude je však museli skládat na sebe za dodržení uvedených pravidel.

Vyšetřovatel získal fotografie vzniklé při tomto stěhování a chce se ujistit, že během něj nedošlo k manipulaci s daty. Na vstupu dostanete každou fotografii popsanou jako řetězec znaků A, B a C, kde k-tý znak určuje polohu k-tého nejmenšího disku na fotce (protože jsou na každém místě disky seřazeny od největšího po nejmenší, je tím jejich poloha určena jednoznačně). Na obrázku vidíte rozmístění kotoučů odpovídající řetězci BACB a také rozmístění odpovídající řetězci AACB, které vzniklo z prvního rozmístění přesunutím nejmenšího kotouče na místo A.



Víme, že pracovníci přemístili všechny kotouče z místa A na místo B způsobem, při kterém byl počet přesunů kotoučů mezi místy nejmenší možný. Seřaďte zadané fotografie podle času jejich pořízení.

Ukázkový vstup:	*Ukázkový výstup:*
CCA	AAA
CCB	BCA
BCA	CCA
AAA	CCB

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[1] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

Podezření, že počítač byl napaden virem, se ze záznamů nepotvrdilo. I hardwarové selhání bylo téměř vyloučeno, a proto nezbylo nic jiného, než přikročit ke zkoumání zdrojového kódu. Tým programátorů se několik týdnů pokoušel simulovat různé situace a testoval program na nejrůznějších vstupech. A nakonec ji nalezli – téměř neviditelnou chybu projevující se jen při kombinaci různých podmínek, které se právě toho nešťastného odpoledne splnily.

Výstražný systém pro řazení událostí používal datovou strukturu, do níž mohlo simultánně zapisovat více vláken, tedy procesů, které spolu sdílejí data. V takovém případě se používají různé mechanismy zajišťující, aby se taková struktura nerozbila. Přestože systém toto obvykle zvládal dobře, při splnění podmínek – jednou z nich bylo i velké množství přicházejících výstrah – mohla vlákna zapisovat do stejného místa ve struktuře současně. Takový chybový stav se obecně nazývá race condition, a aby ve vícevláknové aplikaci nemohl nastat, musí být kódu věnována velká pozornost.

Poškození dat vedlo k chybám při přidávání nových zpráv, procesy se chovaly nekorektně, zacyklily se a přehlcený server následně spadl. Záložnímu serveru však nezměněná data předal, k chybovým situacím docházelo nadále a i ten nakonec přestal fungovat.

Ve FirstEnergy bohužel nefungovalo nic, co by dispečery na vypnutá varování včas upozornilo. Kvůli tomu je ani nenapadlo sledovat síť pomocí jiných prostředků a na to, co způsobili, začali přicházet až po samotném kolapsu.

Celou událost převyprávěl

Jakub Maroušek

27-4-7 Nástroj pro zpracování textu 14 bodů

Již v prvním dílu jsme si řekli, že Unix byl zpočátku z politických důvodů oficiálně vyvíjen jako „nástroj pro zpracování textu". Je pomalu načase představit si jeho schopnosti v této oblasti. Ale nejprve motivační příklad.

Příklad: Počasí

Svého času bývalo populární nechat si zobrazovat na ploše různé aktuální informace: například o počasí. Samozřejmě existovala spousta programů (zvláště na Windows), která dělala právě tohle: zobrazila na ploše informace o počasí. To je sice užitečné, ale jen omezeně. Co když si vedle toho chcete zobrazit aktuální zprávy, čas do odjezdu nejbližšího autobusu, aktuálně přehrávanou písničku, čas východu a západu slunce, kurz dánské koruny nebo cokoli jiného? A co když tyhle všechny věci chcete naopak zobrazit někde jinde, řekněme třeba na nástěnných hodinách?

Určitě chápete, že na každou z takovýchto věcí vám samostatný program nikdo nenapíše. Unixový svět se k tomu postavil trochu jinak. V něm najdeme programy, které umí zobrazit na ploše *cokoliv*,[2] a jak asi tušíte, tím *cokoliv* je v tomto případě výstup nějakého jiného programu či skriptu. To nám nabízí poměrně bohaté možnosti, neb na rozdíl od programu kreslícího na plochu, je snadné vytvořit shellový skript, který něco vypíše.

Zkusme si právě to: napsat skript, který na svůj výstup vypíše aktuální teplotu v nějakém městě. Pokud bydlíte v Praze, dobrým zdrojem informací o počasí je meteostanice Planetária ve Stromovce.[3] Relevantní kousek HTML kódu této stránky vypadá následovně:

```
<TD VALIGN="top" WIDTH="200">
<FONT [...]>aktuální teplota vzduchu</FONT>
</TD>
<TD VALIGN="top" WIDTH="150">
<FONT [...]>-2.3°C</FONT>
</TD>
```

Pokud jste nikdy o HTML neslyšeli, zkonzultujte např. Wikipedii. Pro účely seriálu bude stačit vědět, že každá webová stránka je ve skutečnosti textový soubor, který popisuje, co se má uživateli zobrazit, a právě tento textový soubor vám `curl` zmíněný níže vypíše.

Podtržený je údaj o teplotě, který bychom rádi získali, `[...]` jsou vynechané nezajímavé části. Jen vás upozorníme na obtíž se zpracováním speciálních znaků (závislých například na kódování, jako jsou diakritická písmenka, nebo třeba znak stupňů). Až s nimi budeme pracovat dále v různých příkazech, budou typicky nahrazeny otazníky.

Potřebovali bychom stránku stáhnout, vybrat z ní správný řádek a z něj oddělit číselnou hodnotu teploty. A nepříliš překvapivě na každou z činností poslouží jiný nástroj.

Všechny ty internety…

Již známe spoustu zajímavých unixových nástrojů pro zpracování informací (a na konci tohoto dílu budeme znát ještě víc), ale zatím jediné, s čím umí pracovat, jsou soubory na našem disku. Kdybychom jim tak dokázali „podstrčit" data získaná z Internetu, otevře se nám nepřeberné množství nových možností, stále s těmi stejnými nástroji.

[1] `http://ksp.mff.cuni.cz/viz/codex`
[2] Třeba prográmek jménem `conky`. Ten sice počasí už umí zobrazovat sám, ale i tak je zajímavé cvičení ho to naučit po svém.
[3] `http://www.planetarium.cz/meteo/PL_meteo.htm`

27-3-7 UNIXové déjà vu

S třetím dílem seriálu o UNIXu jste se poprali statečně. Nejvíc zádrhelů bylo paradoxně v nejméně hodnoceném prvním úkolu, jehož vzorové řešení si necháme až na konec, abyste se nelekli a neutekli hned na začátku. Trochu jste také zápasili se skriptovacími úkoly, kde se projevily jednak nedostatek zkušeností s efektivním kombinováním shellových utilit, jednak znalosti získané jinde. Ty některým z vás dovolily napsat neportabilní kód, který by v jiném shellu než Bashi neběžel. Používejte shell častěji, zkuste si s jeho pomocí šetřit práci, hrajte si s ním. Stručné, elegantní a efektivní vyjadřování vám v něm pak půjde snáz a nebude vás to tolik svádět k používání céčkových konstrukcí v shellu.

Ukol 2 – Hardlinky na adresáře

Pohledem do `man ls` nebo do předchozího dílu seriálu můžeme zjistit, že `ls -ld adresar` vypíše podrobnosti o daném adresáři, nikoliv o souborech v něm umístěných, jako by to udělal bez přepínače `-d`. Počet linků je první číslo vpravo od práv. Experimentální pozorování na všech slušných systémech prozradí, že toto číslo je vždycky aspoň 2. Porovnáváním adresářů s nízkým a vysokým počtem linků přijdeme na to, že se o jedničku zvětší za každý podadresář. Po troše přemýšlení a přečtení výkladu v seriálu by mělo být jasné, že jeden link vede z nadřazeného adresáře, druhý z adresáře samotného (jmenuje se `.` – tečka) a zbytek jsou dvě tečky (`..`) z podadresářů.

Mezi slušné systémy tu nemůžeme počítat Cygwin, který o každém adresáři tvrdí, že má jediný link. Tečku a dvě tečky implementuje úplně speciálně, a to hlavně kvůli podpoře windowsových souborových systémů. Při hlubším pohledu do POSIXu jsem s podivem zjistil, že norma takové chování dovoluje – konkrétně v definici prázdného adresáře.

Tenhle úkol se ukázal být celkem jednoduchým na slušných i neslušných systémech.

Úkol 3 – Mazání obsahu a linku

Příkaz `cat </dev/null >soubor` zkrátí délku souboru na nulu. Pokud před jeho zavoláním soubor smažeme příkazem `rm` (té operaci se říká také unlink – maže se jenom jeden link), původní data se nemění a vytvoří se nový prázdný soubor. Důsledkem je, že původní data jsou pořád přístupná přes zbývající linky, pokud nějaké existují. Pokud se soubor předem neunlinkuje, přepíše se původní obsah a změna se projeví při přístupu přes libovolný link.

Na tomto místě bychom rádi připomněli, že inode je datová struktura, která si pamatuje metadata souboru a odkazy na jeho datové bloky. Link ukazuje na inode. Je to záznam v adresáři, který má jméno (jméno souboru) a číslo inode souboru. Data adresáře se vlastně skládají jen z takových záznamů. V těchto pojmech jste měli v řešení dost zmatku.

Úkol 4 – Nuceně privátní home

Při klasickém umístění domovských adresářů, tedy v `/home/<uživatel>`, je úloha představenými prostředky neřešitelná. Vlastník souboru totiž vždycky může měnit jeho práva, takže bychom museli buď měnit práva adresáři `home` a odstřihnout tak přístup do vlastních domovských adresářů ostatním uživatelům, nebo používat nějaký speciální mechanismus. Norma POSIX ale žádný neposkytuje, jen na souborovém systému ext2 (a vyšších) jste objevili příkaz `chattr`, který si dovolím ignorovat jako těžce neportabilní.

Nemusíme se ale tak moc omezovat, v `/etc/passwd` je možné cestu k domovskému adresáři nastavit. To je dobré vědět, abyste se nikdy na klasické umístění domovského adresáře nespoléhali a vždycky ve skriptech používali expanzi vlnky (`~`, `~hroch`, …) nebo obsah proměnné prostředí `HOME` (`$HOME` expanduje na cestu k domovskému adresáři aktuálního uživatele).

Potřebný nápad je použití nadřazeného adresáře pro kontrolu přístupu. Root si může přivlastnit adresář nadřazený domovskému adresáři hrocha, sebrat práva světu (others), jako skupinového vlastníka nastavit skupinu obsahující jen hrocha a této skupině dát právo k prohledávání (x). Nikdo jiný než root a hroch tak nemá možnost se k hrochovu domovskému adresáři dostat, ať se snaží sebevíc.

Budeme tedy potřebovat skupinu, ve které je hroch sám. Často taková skupina již existuje a jmenuje se `hroch`. Pokud neexistuje, snadno ji vytvoříme:

```
groupadd hroch
usermod -aG hroch hroch
```

Teď už jen vytvořit „neprůhledný" adresář a přesunout do něj hrochův stávající domovský adresář:

```
usermod -md /home/hroch_priv/hroch hroch
chown root:hroch /home/hroch_priv
chmod 750 /home/hroch_priv
```

Úkol 5 – Preprocessing

Od pohledu chceme použít jeden cyklus přes řádky a jeden cyklus přes tokeny na řádku. Když na řádku najdeme token `COMMENT`, zahodíme všechno až do konce řádku – tomu odpovídá příkaz `break` ve vnitřním cyklu. Token `BYE` odpovídá volání `return` – funkce v ten okamžik má skončit a víc ze vstupu nečíst.

Budeme číst zdrojáky, takže se nám bude hodit přepínač `-r` příkazu `read`. Jinak by zpětná lomítka dělala neplechu.

Při volání funkce se hodí možnost mít lokální proměnné. Třeba když měníme proměnnou `IFS`, není pěkné ji nechat změněnou po návratu z funkce, jiné kusy kódu se pak mohou chovat neočekávaně. POSIX v tomto ohledu nabízí dvě řešení, obě dost špatná.

- Můžeme uložit starou hodnotu `IFS` do jiné proměnné, změnit `IFS` a nakonec vrátit `IFS` původní hodnotu. Na to se ale snadno zapomíná, funkci bývá možné opustit více způsoby a po návratu z funkce zůstane nastavená zbytečná proměnná.
- Také můžeme celé volání funkce provádět v subshellu – stačí tělo funkce obalit do kulatých závorek a místo `return` volat rovnou `exit`. Nevýhody jsou nasnadě: pouští se další proces, nejde snadno nastavovat nelokální proměnné nebo ukončit shell.

Bash nabízí třetí řešení, které jiné shelly nemají. Tím je vestavěný příkaz `local`, o kterém se více dozvíte v `help local`. Používá se podobně jako `export` a příklad použití najdete ve vzorovém řešení.

Už ve druhém dílu seriálu jste potkali konstrukce `for`, `case` a podmínky. Podmínky jste se učili i pomocí `||`. Přidejme k tomu obsah třetího dílu a řešení je na světě.

Program (shell):
`http://ksp.mff.cuni.cz/viz/27-3-7-preprocess.sh`

Zvolil jsem nejjednodušší realizaci smyčky přes tokeny. Nechal jsem shell pomocí `IFS` rozdělit řádku vstupu podle mezer a přes výslednou množinu slov jsem pustil for-cyklus.

Stačí tedy z počátečního stavu procházet do šířky. Tím máme zaručeno, že projdeme všechny dosažitelné stavy a navíc je projdeme v tom pořadí, v jakém následují při průletu sondy. Můžeme tedy průběžně ve všech stavech počítat největší získatelné množství energie.

Při přechodu z (a, k) do $(a+1, k)$ odebereme energii rovnou vzdálenosti mezi místy a a $a+1$. Pokud však přecházíme přes prošlou oblast, tedy například z (a, k) do $(k-1, a)$, musíme odečíst i součet vzdáleností mezi všemi již prošlými místy. Nakonec nesmíme zapomenout přidat energii získanou na nové pozici.

Celý průchod je nakonec velmi jednoduchý, jenom se nesmíme ztratit v indexování a přičítání ± 1. Na detaily se podívejte do zdrojového kódu. Časová složitost celého algoritmu je $\mathcal{O}(N^2)$, protože každý z kvadraticky mnoha stavů zkoušíme nejvýše jednou. Díky tomu, že nepotřebujeme optimální cestu sondy zrekonstruovat, vystačíme si s lineární pamětí.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-3-4.c`

Jenda Hadrava

27-3-5 Komprese obrazu

Protože na vstupu dostaneme vždy obrázek o rozměrech $2^K \times 2^K$, v každé úrovni rekurzivního komprimování pracujeme vždy se čtvercem. Pokud má na aktuální úrovni čtverec rozměry 1×1, nic již nekomprimujeme a jen vrátíme jeho barvu.

Co přesně máme v každém kroku kvadrantistické komprese dělat? Potřebujeme vybrat dvě čtvrtiny uvažovaného čtverce a každou „odbýt“ jednou barvou, respektive jednu prohlásit za celobílou a druhou za celočernou. Řekněme, že cenou výsledné komprese je počet pixelů, které musely být přebarveny.

To nás může svádět k řešení, ve kterém pro každou čtvrtinu spočteme počet černých a bílých pixelů. Následně „nejčernější“ z nich obarvíme černě, nejbělejší bíle a zbylé čtvrtiny pak zpracujeme rekurzivně, přičemž rekurzivní funkce bude vracet cenu zakomprimování dané části. Toto řešení má dva problémy – jednak není úplně jasné, jak vybírat nejbělejší a nejčernější čtvrtiny (může jich být více se stejným počtem bílých/černých pixelů), a navíc nemusí vést k optimálnímu řešení.

Mohlo by se totiž stát, že sice v jedné čtvrtině je hodně černých pixelů, více než v ostatních, ale díky rozložení černých pixelů se dobře kvadrantisticky komprimuje. Pak by jí tento algoritmus obarvil celočerně a namísto toho se kvadrantisticky snažil komprimovat čtvrtinu, která má pixely rozložené dost nepravidelně.

Zkusme to trochu jinak – na každé úrovni se pro každou čtvrtinu zeptáme naší rekurzivní funkce, kolik by stálo její zkomprimování. Budeme si pamatovat i počet černých a bílých pixelů v každé čtvrtině. Pokud známe tyto informace, můžeme prostě vyzkoušet všech 12 možností, jak vybrat celočernou a celobílou čtvrtinu, a seskládat pro každou z možností její cenu. Ze všech dvanácti cen si pak vybereme tu nejmenší.

Kolik na takový výpočet potřebujeme času? Povšimněme si, že rekurzivní funkce se nikdy nevolá znovu pro stejný čtverec, protože uvažované čtverce stejné velikosti se nikde nepřekrývají. Víme, že na každém čtverci velikosti $2^m \times 2^m$ strávíme $\mathcal{O}(4^m)$ času, a takových čtverců uvažujeme na každé úrovni rekurze $\frac{4^K}{4^m}$. Celkem tedy potřebujeme $\mathcal{O}(4^K \cdot K)$ času i paměti.

Toto řešení by šlo ještě zrychlit: určit počet bílých pixelů pro každý čtverec lze v čase $\mathcal{O}(1)$, protože stačí sečíst výsledky ze čtyř menších čtvrtin. A pokud si vybrané „odbyté“ čtvrtiny budeme uchovávat trochu lépe, dostaneme řešení, které si vystačí s lineárním časem i pamětí v počtu pixelů.

Program (C++):
`http://ksp.mff.cuni.cz/viz/27-3-5.cpp`

Ondřej Hübsch

27-3-6 Ukládání přepravek

Někteří z vás poznali, že jde jen o jiné zadání daleko známější úlohy, totiž barvení intervalových grafů. Tato úloha se dá ve zkratce zadat jako hledání nejmenšího počtu poslucháren, které potřebujeme pro přednášky zadané svými časy začátků a konců.

Tato úloha se dá řešit hladově, a to ve velkém množství variant, proto byla většina řešení funkční. Hlavní část řešení je ovšem obhájit, že hladově řešit opravdu jde.

Začneme tím, že si přepravky seřadíme podle vnějšího průměru sestupně. Na vnitřním průměru tady nezáleží, jak později uvidíme. Budeme si udržovat seznam potřebných komínků, jakési průběžné řešení. Jediná hodnota, kterou z každého komínku potřebujeme, je vnitřní průměr nejmenší přepravky.

Přepravky budeme postupně probírat a snažit se je vložit do nějakého komínku. Protože víme, že žádná větší přepravka už nepřijde, stačí ze seznamu vybrat maximum. Bude se nám tedy hodit reprezentovat komínky maximovou haldou.

Pokud je maximum menší než vnější průměr aktuální přepravky, pak žádný vhodný komínek neexistuje a proto přidáme nový. Na konci běhu algoritmu prostě jen spočítáme počet komínků ve stromu.

Konečnost algoritmu je zřejmá, podívejme se na správnost. Dokážeme ji indukcí podle velikosti průběžného řešení. Dokud máme jen jeden komínek (případně žádný), nemůže existovat lepší řešení.

Mějme tedy k komínků. Jediné, co může k zvýšit, je pak situace, kdy neexistuje pro nějakou přepravku vhodný komínek. V tu chvíli ale existuje alespoň k přepravek, co mají vnější průměr alespoň takový jako aktuální přepravka (díky setřízení), a vnitřní průměr ostře menší (neexistuje vhodný komínek). S aktuální přepravkou je to tedy $k+1$ přepravek, ze kterých žádné dvě nejdou vložit do sebe. Optimální řešení tedy nemůže používat méně než $k+1$ komínků.

Jaké vlastnosti algoritmus má? Už kvůli prvotnímu setřízení vidíme, že nepoběží rychleji než v $\mathcal{O}(N \log N)$. Další kroky algoritmu závisí na počtu komínků, tedy na řešení. Ten nemůžeme nijak rozumně odhadnout – počet komínků může být až N. Každý krok tedy zabere až $\mathcal{O}(\log N)$ času. Tedy nakonec se opravdu do $\mathcal{O}(N \log N)$ vejdeme. Paměti spotřebujeme $\mathcal{O}(N)$, více nepotřebujeme.

Vzorový kód je napsaný v C++ a využívá standardní implementaci haldy.

Program (C++):
`http://ksp.mff.cuni.cz/viz/27-3-6.cpp`

Ondra Hlavatý

To můžeme zařídit prográmkem `curl`, který na spoustě unixových systémů najdeme ve výchozí instalaci, případně si jej lze snadno doinstalovat (i v Cygwinu). Jeho použití je přímočaré: jako parametr dostane URL webové stránky (či stáhnutelného souboru) a na standardní výstup vypíše její obsah (HTML kód). Odtud jej můžeme přesměrovat do souboru či poslat kamkoli rourou, jak jsme zvyklí.

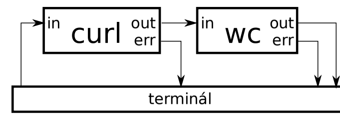
Zkuste si spustit například

```
curl http://ksp.mff.cuni.cz/ | wc -c
```

Asi vás překvapí, že kromě očekávaného výstupu se objeví několik podivných řádek, které vypadají jako informace o průběhu stahování. Jak je to možné, když je výstup příkazu `curl` přesměrován?

Ve skutečnosti má každý proces kromě svého standardního vstupu (též *stdin*) a výstupu (*stdout*) ještě třetí „datový kanál“, takzvaný *standardní chybový výstup* (*stderr*). Ten směřuje na terminál, i když je výstup příkazu přesměrován do souboru či roury. Jak již název napovídá, slouží k tomu, že když za běhu příkazu nastane chyba, uvidí ji uživatel namísto toho, aby se zapsala doprostřed výstupního souboru. Ale nepoužívá se jen pro chyby a varování, nýbrž i pro informace o stavu a průběhu programu. A právě `curl` na něj vypisuje informace o průběhu stahování, díky čemuž když stahujete velký soubor příkazem `curl http://adresa >soubor`, vidíte, kolik již je staženo a kolik času ještě zbývá.

Naše *pipeline* (tímto pojmem se označuje řada příkazů pospojovaných rourami) ve skutečnosti z pohledu operačního systému vypadá takto:



Teď už je jasné, kudy se ona hlášení na terminál dostanou. I stderr se dá přesměrovat, a to operátorem `2>`. Nejčastěji se to používá k umlčení všech hlášení, například takto: `curl http://... 2>/dev/null | wc -c`. O speciálním souboru `/dev/null` byla řeč v minulém díle. Je třeba dávat pozor, ke kterému příkazu přesměrování patří. Pokud byste ho napsali na konec, kýženého efektu nedosáhnete, neb přesměrujete stderr procesu `wc`, nikoli `curl`. V případě `curl`-u ale přesměrování používat nemusíte, neboť nabízí přepínač `-s`, který stavové zprávy utiší.

Na závěr dodáme, že ke stejnému účelu jako `curl` lze použít i příkaz `wget`, který ovšem ve výchozím nastavení ukládá staženou stránku do souboru. K vypisování na stdout ho přimějete parametrem `-O -`, hlášení o průběhu umlčíte `-q`. Více v manuálové stránce.

Filtrování řádek: grep

Občas by se nám hodilo z textového souboru vybrat řádky splňující nějaké kritérium. Nejjednodušším nástrojem, který takovou věc dělá, je `grep`. Ten jako parametr přijímá slovo (kus textu), čte postupně řádky ze svého vstupu a na výstup vypisuje jen ty, které zadané slovo obsahují (myšleno obsahují jako podřetězec, nemusí být oddělené mezerami). S parametrem `-v` (inVert) naopak vypisuje jen řádky **ne**obsahující dané slovo. Pokud hledané slovo obsahuje nějaké „divné“ znaky, je třeba `grep`-u dát ještě parametr `-F`. Které přesně znaky jsou divné a proč, si vysvětlíme později, prozatím můžete za bezpečná považovat písmena a číslice.

S parametrem `-i` ignoruje `grep` při hledání slova velikost písmen (v závislosti na nastavení systému nemusí fungovat pro české znaky).

Pokud dáte `grep`-u víc parametrů, další jsou brány jako názvy souborů, ve kterých se má hledat. Tedy `grep slovo soubor1 soubor2` se chová podobně jako `cat soubor1 soubor2 | grep slovo`, s tím rozdílem, že pokud je soubor víc než jeden, `grep` před každý vyhovující řádek napíše název souboru, ze kterého pochází (to se dá vypnout přepínačem `-h`). Například kdybych chtěl zjistit, zda jsme v některém díle seriálu zmiňovali proměnnou `$RANDOM`, použiji:

```
$ grep -F '$RANDOM' serial*
serial3.tex:X=$(($RANDOM % 100))
[...]
```

a vidím, že to bylo ve třetí sérii. Pokud nás zajímají jen názvy souborů, ve kterých se slovo vyskytuje, můžeme použít přepínač `-l` (dobře se pamatuje podle `ls`, které taky vypisuje názvy souborů), případně `-L` pro opačnou operaci (výpis souborů neobsahujících ani jednou dané slovo). S přepínačem `-r` můžeme `grep`-u předávat za parametry i názvy adresářů; v těch pak prohledá všechny soubory rekurzivně. `grep -lr bagr ~` najde v domovském adresáři všechny soubory obsahující slovo `bagr`. Takovéto hledání může chvíli trvat.

Přepínačem `-C` K (Context) řeknete `grep`-u, že má kromě vyhovujících řádek vypsat i K řádek okolo každé z nich. Pokud místo `-C` použijete `-B` (Before) či `-A` (After), bude vypsán jen kontext jednostranný (K řádek před, resp. za každým vyhovujícím). Pokud by se kontexty překrývaly, jsou slity do jednoho bloku, žádný vstupní řádek se na výstupu neobjeví dvakrát. Tedy např. `grep -C 999 : na /etc/passwd` vypíše to samé, co `cat`. Pokud na sebe sousední kontexty nenavazují, jsou odděleny řádkem s dvěma pomlčkami pro snazší vizuální orientaci.

Pokud používáte `grep` ručně, mohl by vás zajímat přepínač `--color`, který výskyty hledaného slova barevně zvýrazní. Ve skriptech by se vám naopak mohl hodit přepínač `-q`, který způsobí, že se na výstup nevypíše nic (jako `>/dev/null`). K čemu je taková věc dobrá? Zatím jsme vám zatajili, že `grep` vrací nulovou návratovou hodnotu, pokud našel alespoň jeden vyhovující řádek, jinak nenulovou. Takže můžete použít `if grep -q slovo soubor` pro test, zda je v souboru obsaženo slovo.

Teď už máme asi vše potřebné k výběru správného řádku:

```
$ curl -s ... | grep -a -A 3 "teplota vzduchu" \
            | head -n 4 | tail -n 1
<FONT COLOR="#FFFF00" FACE="Arial">0.4?C</FONT>
```

Od začátku psaní se trochu oteplilo. ;-)

První `head` je potřeba, protože se text na stránce vyskytuje vícekrát (a kvůli podivně kódovaným českým znakům nemůžeme hledat slovo „aktuální“). Přepínač `-a` slouží k tomu, aby `grep` vypsal obvyklý výstup, i pokud považuje soubor za binární (obsahuje nějaké neobvyklé znaky).

Ještě snad dodejme, že zpětné lomítko na konci prvního řádku znamená, že příkaz pokračuje na řádku následujícím. Do skriptu to můžete napsat přesně takto, jen je třeba dát si pozor, aby za lomítkem nebyla žádná mezera. Pokud byste chtěli příkaz spustit v terminálu, je lepší napsat vše na jeden řádek (a lomítko vynechat).

Sekání a slepování řádek: cut, paste a spol.

Už umíme vybírat ze souboru celé řádky. Občas by se nám mohlo hodit získat i jejich části. K tomu nám poslouží příkaz `cut`. Ten se nejčastěji používá se soubory „tabulkového" charakteru (např. `/etc/passwd`), kde každý řádek je rozdělen na několik „sloupečků" nějakým oddělovačem (v Unixu typicky dvojtečka či libovolná posloupnost bílých znaků – s tou si ale `cut` neporadí). Důležité přepínače jsou `-d`, který nastavuje oddělovač (musí být jednoznakový) a `-f`, jenž říká, které sloupečky chceme na výstupu. Jeho parametrem může být jedno číslo, rozsah čísel (`3-5`), případně seznam čísel a rozsahů oddělených čárkami (`1,3,5-10`). Sloupečky jsou číslovány od jedničky.

Kombinací příkazů `cut` a `grep` můžeme nyní třeba zjistit ID uživatele `hroch`:

```
$ grep hroch /etc/passwd | cut -d: -f3
4242
```

To ale není úplně spolehlivé, například se to rozbije, pokud se někdo bude jmenovat `druhyhroch`; později si ukážeme lepší způsob. Stejně jako spousta jiných příkazů, pokud `cut` dostane jako parametr jeden nebo více názvů souborů, čte z nich, jinak čte ze standardního vstupu. Zapisuje vždy na standardní výstup. Sloupečky nejde prohazovat: `-f 3,1` je to samé, jako `1,3`.

`cut` má ještě alternativní režim, kdy místo sloupečků vysekává z řádků jednotlivé znaky, např. `cut -c 1-3` vybere z každého řádku první tři znaky.

Inverzní operací ke `cut` je `paste`, který dostane za parametry několik souborů, které považuje za jednotlivé sloupečky. Pak vezme první řádek z každého souboru a všechny spojí zadaným oddělovačem (`-d`), čímž vznikne první řádek výstupu. A tak dále pro další řádky. S ním bychom mohli, byť trochu neobratně, zařídit například zmiňované prohození sloupečků:

```
$ cut -d: -f1 /etc/passwd >jmena
$ cut -d: -f3 /etc/passwd >uid
$ paste -d: uid jmena
0:root
4242:hroch
[...]
```

Formát „sloupečků" oddělených dvojtečkami je sice příjemný pro strojové zpracování, ale člověku se čte o dost hůře než opravdové sloupečky, kde jsou odpovídající si hodnoty zarovnané pod sebou. Takovýto oku přívětivý formát lze vyrobit programem `column`.

Ten pracuje ve dvou režimech. Prvním z nich je tabulkový (`column -t -s` *oddělovač*), ten načte ze vstupu soubor v „oddělovačovém" formátu a na výstup ho vypíše jako hezkou tabulku:

```
cut -d: -f 1-4 /etc/passwd | column -s: -t
root   x  0     0
hroch  x  4242  4242
[...]
```

Sloupcový režim očekává na vstupu jednoduchý seznam položek, které vypíše na výstup ve vícesloupcové sazbě, podobně, jako to dělá `ls`. Tím se dá šetřit místo na obrazovce, pokud jednotlivé vstupní řádky jsou krátké. Například kdybychom chtěli vypsat seznam všech uživatelských jmen v systému:

```
$ cut -d: -f1 /etc/passwd | column
root     hrosik   hacker
hroch    guest    nobody
```

Počet sloupců je zvolen automaticky, dá se ovlivnit přepínači, stejně jako se dá zařídit, aby se hodnoty vyplňovaly po řádkách namísto po sloupcích (používejte jen pokud víte, co děláte, čte se to hrozně).

Nahrazování znaků: tr

V nejjednodušším použití `tr` nahradí všechny výskyty jednoho znaku (určeného prvním parametrem) na svém vstupu za jiný (druhý parametr) a výsledek vypíše na výstup. Každý z parametrů může být i celým seznamem znaků, zadaným buď vyjmenováním těsně za sebou, rozsahem (např. `a-z`) nebo kombinací obojího. Pak platí, že každý výskyt nějakého znaku z prvního seznamu se nahradí za odpovídající znak z druhého seznamu.

Například `tr a-z A-Z` převede svůj vstup na velká písmena (funguje jen pro znaky anglické abecedy, ostatní nechá beze změny), `tr a-zA-Z A-Za-z` prohodí velikost písmen (z velkých udělá malá a z malých velká). Pokud některý ze seznamů je kratší, je doplněn zopakováním posledního znaku. Např. `tr aeiouy e` nahradí všechny (malé) samohlásky v textu za e.

S přepínačem `-d` očekává `tr` jen jeden seznam znaků a všechny znaky na tomto seznamu ze vstupu smaže. Přepínač `-c` vezme místo prvního seznamu znaků jeho doplněk. Nejčastěji se používá spolu s `-d` pro smazání všech znaků kromě zvolených nebo s jednoznakovým pravým seznamem. `tr -c a-zA-Z _` nahradí všechny znaky kromě písmen za podtržítka.

Řešení: Počasí

Nyní už máme všechny střípky k vyřešení našeho úvodního příkladu:

```
$ curl -s \
  http://www.planetarium.cz/meteo/PL_meteo.htm \
       | grep -a -A 3 "teplota vzduchu" \
       | head -n 4 | tail -n 1 \
       | cut -d'>' -f2 |cut -d'<' -f1 \
       | tr -cd '0-9.-'
-0.7
```

Takovéto číslo si pak můžeme nejen nechat někde zobrazit, ale také s ním libovolně dál pracovat. Například by nebylo těžké napsat skript, který se před vypnutím počítače podívá na aktuální teplotu, a pokud je méně než 15, zobrazí upozornění „Vezmi si bundu!"

Rest: seq

Při povídání o for-cyklech jsme vám zatajili velmi užitečný příkaz `seq`. `seq` *A B* na svůj výstup vypíše všechna čísla od *A* po *B* (obojí včetně, *A* lze vynechat, pak se vypisuje od jedničky), každé na samostatný řádek. Spolu s `for`-em se dá použít pro zopakování nějakých příkazů *n*-krát.

(Kdybychom se rozhodli slova nevkládat do trií, ale řadit, zvládneme to také lineárně – díky pevné velikosti abecedy to jde pomocí RadixSortu.)

Za zmínku možná stojí, že v téhle úloze občas realita spráská asymptotiku a pošle ji stydět se do kouta. I nepříliš optimalizovaná řešení se složitostí $\mathcal{O}(\ell \log \ell)$ můžou pro rozumně velké vstupy doběhnout rychleji než řešení lineární. Souvisí to mimo jiné s tím, že trie je budovaná pomocí ukazatelů, a s efekty keší… ale to už by byl úplně jiný příběh.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-3-2.c`

Karolína „Karryanna" Burešová

27-3-3 Výběr vysílačů

Úlohou je vlastně obarvit strom barvami 1, 2, 4, 8, … tak, aby sousední vrcholy měly různé barvy a součet hodnot barev přes všechny vrcholy byl co nejmenší. Připomeneme, že bez požadavku na minimální součet lze každý strom korektně obarvit dvěma barvami. To můžeme udělat například průchodem stromu do hloubky. Ten může vypadat například takto:

```
1. obarvi(v, b):
2.     barva[v] = b
3.     for (u in soused(v)):
4.         if (barva[u]==0): obarvi(u, 3 - b)
```

Nejdříve obarvíme libovolný vrchol barvou 1, pak víme, že jeho sousedi musí mít barvu 2, jejich sousedi zase barvu 1, a tak dále. Tento jednoduchý algoritmus má časovou složitost $\mathcal{O}(N)$ a budeme na něm dále stavět.

Tím, že strom umíme obarvit barvami 1 a 2, dostáváme obarvení se součtem maximálně $2N$. Tedy hledané obarvení s minimálním součtem určitě nepoužije barvy větší než $2N$. To zároveň znamená, že barev použijeme maximálně $\log N + 1$.

Nyní pomalu přejdeme k popisu algoritmu. Strom si zakořeníme v libovolném vrcholu a z něj pustíme prohledávání do hloubky. To vždy nejdříve spočítá řešení pro podstromy tvořené syny vrcholu a z těchto řešení složí řešení pro daný vrchol. Tento postup je takovým standardním dynamickým programováním na stromě. A co přesně v podstromech budeme počítat?

Pro každou barvu $c = 2^i$ pro $i = 1, \ldots, \log N + 1$ spočítáme nejlepší možné obarvení podstromu, ve kterém je kořen obarven barvou c. Abychom zjistili nejlepší obarvení pro barvu c, tak stačí pro každého syna vybrat nejlepší barvu jinou než c. To pro konkrétního syna vždy bude buď jeho nejlepší barva, anebo jeho druhá nejlepší barva. Tudíž nám stačí si pro každý podstrom pamatovat pouze dvě nejlepší možnosti.

Po zavolání výpočtu v kořeni stromu dostaneme výsledek. Během výpočtu pro každý vrchol zkoušíme $\log N + 1$ barev plus $(\log N + 1)$-krát u něj vybíráme jednu ze dvou barev pro otce. Tedy časová složitost algoritmu je $\mathcal{O}(N \log N)$. My ale algoritmus ještě vylepšíme.

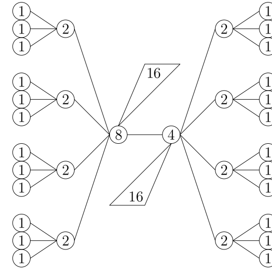
Všimneme si, že pokud se má vyplatit vrcholu přiřadit barvu 2^i, tak všechny barvy z $2^0, \ldots, 2^{i-1}$ musí být zastoupeny v jeho synech. Tedy u vrcholu s k syny nám pro získání dvou nejlepších možností stačí vyzkoušet $k+2$ barev (druhá nejlepší možnost pořád může mít barvu $k+2$).

Další věc, kterou na předchozím algoritmu můžeme vylepšit, je opětovné vybírání z nejlepší a druhé nejlepší barvy synů. Víme, že vždy vybereme tu nejlepší kromě případu, kdy pro kořen zvolíme stejnou barvu, pak vybereme druhou nejlepší. Stačí nám spočítat součet nejlepších možností synů a pro každou barvu $c = 2^i$ pro $i = 1, \ldots, k+2$ si předpočítat, o kolik se dohromady liší nejlepší a druhé nejlepší možnosti synů, kteří jako svou nejlepší barvu mají c. To pro vrchol s k syny můžeme jednoduše udělat v čase $\mathcal{O}(k)$.

Jelikož každý vrchol je synem právě (maximálně) jednoho jiného vrcholu, dostáváme se na časovou složitost $\mathcal{O}(N)$.

Pro zajímavost ještě řekneme, že stejný algoritmus funguje i pro verzi úlohy, kde strom barvíme barvami $1, 2, 3, \ldots, N$. Sami si můžete rozmyslet proč.

A kdo je zvědavý, tak strom, pro který jsou potřeba alespoň čtyři barvy, může vypadat například takto:



„Šestnáctkové" trojúhelníky zastupují šestnáct přímých potomků uzlu, kde je každý obarvený jedničkou.

Karel Tesař

27-3-4 Doplňování energie

První, co nás může napadnout, je vyzkoušet všechny možnosti průchodů. Těch je však velmi mnoho, protože nám nic nebrání chodit přes některá místa vícekrát.

Ujasněme si nejprve, jak bude vypadat přelet, který může získat maximální množství energie. Protože nemáme žádné omezení na uchovávanou energii, vyplatí se nám vždy při prvním navštívení nějakého místa z něj veškerou energii vyčerpat. Nemá tedy smysl si ji zde šetřit na později. Stejně tak je zbytečné nějaké místo při přeletu vynechat a nezastavovat se na něm.

Díky tomu můžeme předpokládat, že při každém čerpání energie máme projitou souvislou oblast, a navíc se nacházíme na jejím kraji. Tím jsme výrazně snížili počet možností, které chceme vyzkoušet. Nyní máme pouze N levých a N pravých konců, celkem N^2 možných koncových stavů. Stav je tedy dvojice čísel (a, k), kde a je aktuální pozice a k druhý konec prošlé oblasti.

Jak ve stavech spočítat optimální množství energie? Pro stav s oběma konci intervalu na startovní pozici M energii určíme snadno. Bude rovna právě energii, kterou z této pozice můžeme čerpat.

Do ostatních stavů se můžeme dostat nejvýše ze dvou předchozích. Konkrétně pro $a > k$ se do stavu (a, k) dostaneme ze stavu $(a-1, k)$ nebo $(k, a-1)$. A pro $a < k$ to jsou stavy $(a+1, k)$ a $(k, a+1)$. Do nich se umíme dostat zase ze dvou předchozích, a tak dál. V každém případě můžeme začínat pouze ve stavu (M, M).

Trampoty s velkými čísly

Naše algoritmy počítají s délkami intervalů a plochami čtverců jako s normálními čísly. Tato čísla ale rostou exponenciálně s hloubkou stromu, takže by se mohlo stát, že se nevejdou do běžné celočíselné proměnné. Zadání o této možnosti nevinně mlčelo, ale zkusme ji aspoň na chvilku připustit. Držte si klobouky…

První z algoritmů na konstrukci grafu s délkami ani plochami nepočítá, takže ho měnit nemusíme.

Druhý algoritmus používá délky při propojování seznamů intervalů. Můžeme si pomoci takto: podíváme se na intervaly x a y na začátku seznamů. Pokud jsou stejného řádu, nic se nemění. Pokud je (BÚNO) x vyššího řádu, rozdělíme ho na dva intervaly o 1 nižšího řádu a postup opakujeme. Části vzniklé rozdělením zdědí informace o vrcholu grafu a o barvě z původního intervalu. Jako cvičení ponecháváme dokázat, že vytvoříme nejvýše lineárně mnoho nových intervalů (nápověda: představte si strom popisující postupné dělení intervalů).

Hledání komponent souvislosti zůstane beze změny, ale pak potřebujeme spočítat plochy komponent. Nejprve každému čtverci přiřadíme číslo komponenty. Poté projdeme celý strom po hladinách shora dolů, a kdykoli narazíme na jednobarevný čtverec, přidáme ho k příslušné komponentě. Pro každou komponentu tudíž vznikne seznam jejích čtverců uspořádaný sestupně podle řádu.

Nyní v tomto pořadí budeme sčítat plochy čtverců. To jsou potenciálně obrovská čísla. Budeme je zapisovat ve dvojkové soustavě a do paměti ukládat jako seznam pozic jedniček ve dvojkovém zápisu, uložený od nejvyššího bitu k nejnižšímu. Každý další čtverec přitom přispěje plochou, která je menší nebo rovna zatím nasčítané ploše. Můžeme tedy připsat jedničkový bit na konec čísla, jen se nám mohlo stát, že už tam jeden bit tohoto řádu mohl být, takže dojde k přenosu. Vyřizováním všech přenosů ovšem strávíme lineární čas, protože s každým přenosem klesne celkový počet jedničkových bitů o 1. (To je podobná úvaha jako onehdy s intervaly.)

Plochy máme spočítány, zbývá z nich najít maximum. Kdykoliv při tom porovnáváme dvě čísla, procházíme je od nejvyššího řádu a jakmile se přestanou shodovat, porovnávání ukončíme. Čas strávený porovnáváním přitom účtujeme jedničkám v menším z obou čísel, které se už žádného dalšího porovnávání nezúčastní. Takto celkem naúčtujeme každé jedničce čas $\mathcal{O}(1)$ a všech jedniček za celou dobu běhu algoritmu vznikne $\mathcal{O}(n)$.

I s obrovskými čísly jsme tedy dokázali udržet lineární složitost algoritmu. Kouzlo se podařilo `;-)`

Martin „Medvěd“ Mareš & Filip Štědronský

27-3-2 Návrhy pro komisi

V úloze se pracuje s řetězci, zkušený řešitel si tedy přečte zadání, zamyslí se a řekne si „Ha, trie!“. To je speciální strom, ve kterém se hranám přiřazují písmenka a cesta od kořene nějak odpovídá vstupním řetězcům; více o ní píšeme v kuchařce o hledání v textu.[15]

Jenže jak přesně trii na naši úlohu použít? Můžeme jednoduše počítat, kolik členů komise schvaluje určité slovo – stačí si do vrcholů přidat počítadlo. Když se pak vydáme od kořene dolů a budeme tyto členy postupně sčítat, zajímá nás, jestli v daném vrcholu součet dosáhne alespoň hodnoty K. Pokud ano, jakýkoliv návrh začínající slovem odpovídajícím aktuálnímu vrcholu bude schválen.

Kolik takových návrhů existuje? Když aktuální hloubku označíme jako d, dá se jejich počet vyčíslit jako 26^{D-d}. Za každý ze znaků, který chybí do přijatelné délky návrhu D, totiž smíme zvolit libovolný znak anglické abecedy.

Také z uzlu, v kterém se nasbíralo alespoň K hlasů, nechceme postupovat níž – všechna možná pokračování budou začínat schvalovaným řetězcem, takže jsme je už zahrnuli. Při dosažení hodnoty K se tedy zastavíme, resp. vrátíme o úroveň výš a necháme procházení postupovat dál.

To zní dobře. Ale bude to opravdu fungovat? Co kdyby nějaký člen schvaloval jak slovo „psa“, tak slovo „psal“? Ouha! To bychom ho započítali vícekrát, což nechceme – a zadání nám rozhodně neslibuje, že taková situace nenastane.

Připomeňme si naznačenou myšlenku: pokud člen (či komise) schvaluje nějaké slovo, schválí i jakékoliv další, které tímto slovem začíná. Kdybychom tedy věděli, že člen už schvaluje nějaký prefix aktuálního slova, můžeme právě zpracovávané slovo s klidem zahodit.

Tady se nabízejí dva přístupy. Můžeme slova každého člena lexikograficky seřadit a přidávat je do trie už seřazená. Jen si musíme rozmyslet, jak v trii komise poznat, zda jsme do aktuálního vrcholu už přičetli hlas právě zpracovávaného člena.

Alternativu, se kterou pracuje i vzorový program, představuje vybudování druhé trie, tentokrát specifické pro člena. Do ní jednoduše naházíme všechna jeho oblíbená slova (značíme si ve vrcholech, jestli tam končí slovo). Následně ji projdeme a vždy, když narazíme na konec slova, odpovídající slovo přidáme do trie pro komisi a hned se vrátíme.

Po zpracování slov všech členů trii projdeme tak, jak jsme popsali na začátku. Teď už hlas každého člena započítáme pro libovolný řetězec maximálně jednou, takže program vydá správný výsledek. Zbývá se zamyslet nad složitostí.

Co je vlastně vstup? Kromě parametrů C, K a D to jsou zejména všechna oblíbená slova. Označme si počet všech znaků v nich jako ℓ.

Operace s trií, když máme konstantně velkou abecedu, můžeme bez obav prohlásit za konstantní. Do trie pro komisi přidáme maximálně ℓ znaků, čímž vytvoříme maximálně ℓ vrcholů. Při závěrečném průchodu navštívíme každý vrchol nanejvýš jednou, zpracování trie pro celou komisi tedy trvá $\mathcal{O}(\ell)$ času a zabírá $\mathcal{O}(\ell)$ paměti.

To samé platí pro trie jednotlivých členů. Zdánlivě tedy máme celkovou složitost $\mathcal{O}(C \cdot \ell)$. Jenže trie všech členů nemůžou mít dohromady víc než ℓ znaků, takže zpracování všech členských trií dohromady nemůže zabrat víc než $\mathcal{O}(\ell)$.

Započítali jsme všechny operace? Skoro. Neměli bychom zapomenout na mocnění, ač ve vašich řešeních jsem jeho zohlednění nevyžadovala. Zatímco násobení za konstantní prohlásit můžeme, u mocnění by to bylo poněkud odvážné, zvlášť když by teoreticky mohlo nastat $D \gg \ell$. Trochu si ovšem život usnadníme a jen prohlásíme, že mocnění trvá $\mathcal{O}(m)$. Pak má celý náš program časovou složitost $\mathcal{O}(\ell \cdot m)$, paměťovou $\mathcal{O}(\ell)$.

[15] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`

Přeuspořádání řádků: sort, shuf, tac

Představíme si skupinu programů, která nějakým způsobem mění pořadí řádků na svém vstupu. Nejdůležitějším z nich je `sort`, který setřídí vstup či soubor dle zadaných kritérií. Ve výchozím nastavení třídí řádky abecedně vzestupně. Můžeme použít přepínače `-r` pro sestupné třídění, `-f` pro ignorování velikosti písmen, `-n` pro číselné porovnávání (abecedně by se zatřídila „100“ před „11“) a `-k` pro třídění podle některého sloupečku (ve stejném významu jako u `cut`-u, oddělovač se nastavuje `-t`, výchozím oddělovačem je whitespace, tedy libovolná posloupnost bílých znaků). Například `sort -t: -k3 -n /etc/passwd` setřídí záznamy v `/etc/passwd` podle ID uživatele.

S parametrem `-R` `sort` místo třídění vstupní řádky náhodně zamíchá. To samé dělá `shuf`, jen nabízí nějaké parametry navíc, například vybrat ze vstupu náhodně jen K řádek (`-n` K) nebo povolit vybrat jeden řádek vícekrát (`-r`).

Například pokud byste byli učitel a měli v nějakém souboru uloženou hromadu otázek, ze kterých chcete vygenerovat 30 náhodných písemek po 10 otázkách, můžete napsat:

```
for i in `seq 30`; do
    shuf -n 10 otazky.txt >pisemka-$i.txt
done
```

`tac` vypíše řádky vstupního souboru v opačném pořadí (od posledního po první) a uvádíme jej zde hlavně, abyste se taky mohli těšit z kreativity, kterou do názvů příkazů raní unixoví programátoři vložili ;-) Tak trochu duálním příkazem k `tac` je `rev`, který pořadí řádků nemění, ale zato každý z nich napíše pozpátku. Takže pokud byste chtěli obrátit celý soubor (od posledního znaku po první) `tac | rev` zařídí přesně to.

Třídit soubory nemusíme jen z vlastního zájmu, ale také proto, že některé příkazy svůj vstup setříděný vyžadují. Jedním z nich je `uniq`, který ze vstupu vyhodí duplicitní řádky, ale pouze, pokud leží všechny duplikáty vedle sebe (což zařídíte právě setříděním). `uniq` má spoustu zajímavých přepínačů, jako např. `-c`, který před každý výstupní řádek napíše, kolikrát se vyskytl na vstupu. Třeba pokud máme dlouhý seznam jmen uživatelů a chtěli bychom vědět, která křestní jména se u nich vyskytují nejčastěji, můžeme použít:

```
$ sort jmena.txt | cut -d' ' -f 1 \
  | uniq -c | sort -nr | head -n 3
     64 Jan
     51 Martin
     42 Kateřina
```

S parametrem `-d` se naopak vypisují pouze duplicitní řádky (každý jen jednou), `-i` při porovnávání ignoruje velikost písmen a mnoho dalších zajímavých parametrů najdete v manuálové stránce. Protože `sort | uniq` je tak častá kombinace, existuje za ni zkratka `sort -u`.

Porovnávání souborů: comm, cmp, diff

Dalším příkazem, kterému se hodí setříděný vstup, je `comm`. Slouží pro porovnání dvou množin reprezentovaných řádky setříděných souborů. Ve výchozím nastavení vypíše výstup do tří sloupečků: v prvním jsou řádky obsažené pouze v prvním souboru, v druhém řádky unikátní pro druhý soubor a ve třetím řádky oběma souborům společné. To je hezké pro vizuální porovnání, ale ve skriptech víceméně nepoužitelné. Vhodným nastavením parametrů můžeme zajistit vypsání jen jednoho z těchto sloupečků, ale syntaxe není příliš intuitivní: parametrem $-n$ říkáme, že n-tý sloupeček **ne**chceme zobrazit. Takže typické použití je např. `comm -12 soubor1 soubor2` pro zobrazení řádků vyskytujících se v obou souborech.

Pokud chceme dva soubory jen porovnat na shodnost (pro to už pochopitelně nemusí být setříděné), můžeme použít příkaz `cmp`. Ten skončí s nulovou návratovou hodnotou, pokud je obsah souborů shodný, jinak s nenulovou (dá se tedy použít v rámci příkazu `if`). Při použití ve skriptech doporučujeme přidat parametr `-s`, aby se při neshodě nevypisovalo informativní hlášení.

Dalším nástrojem pro porovnávání souborů je `diff`, který umí zobrazit „v čem“ se dva soubory liší. Nejčastěji se používá pro porovnání dvou verzí téhož souboru, například když přemýšlíme, proč stará verze našeho programu fungovala a nová už ne. Doporučujeme používat přepínač `-u` (zapne trochu smysluplnější výstupní formát). S `-N -r` lze porovnávat rekurzivně celé adresáře. Zkuste si s ním pohrát.

Výstupu programu diff se obvykle říká buď „diff“ (mezi nějakou verzí a nějakou jinou), nebo „patch“. Druhé označení je odvozeno od příkazu `patch`, který s `diff`-em úzce souvisí. Vstupem příkazu `patch` je stará verze souboru a diff mezi starou a novou verzí, výstupem pak zrekonstruovaná nová verze. Například po spuštění

```
diff -u A B >AB.diff
patch -o C A AB.diff
```

bude mít soubor `C` stejný obsah jako `B`. To se dá používat k snadnému šíření úprav: pokud třeba uděláte malou změnu ve velkém programu a chcete se o ni s někým podělit, nemusíte mu posílat celý kód, stačí jen patch.

Ještě větší výhodou je to, že patch lze obvykle aplikovat, i když se mezitím původní soubor (`A` v našem příkladu) změnil. Tohle umožňuje několika lidem nezávisle na sobě změnit nějaký soubor a poté všechny změny automaticky sloučit. Stačí, když každý vyrobí patch mezi společnou původní verzí a svou upravenou verzí a nakonec se všechny tyto patche na soubor postupně aplikují. Problém nastane pouze v případě, že se dva lidé pokusí změnit stejnou část souboru; v takovém případě dojde k takzvanému *konfliktu*, který je třeba vyřešit ručně. Ale to se stává překvapivě málo často.

Na tomto principu je založen vývoj mnoha open source projektů. Přispěvatelé si lokálně program mění a testují, a hotové změny posílají autorům ve formě patchů. Ale i vytváření a aplikování patchů a udržování historie vývoje je spousta ruční práce, pročež se tyto procesy snaží automatizovat takzvané verzovací systémy, jako např. git.[4]

[4] `http://git-scm.org/`

Příklad: Spam

Někteří jste si již možná všimli, že v KSPčku většinu hromadných mailů posíláme každému s vlastním oslovením, včetně správně vyskloňovaných tvarů slov dle pohlaví adresáta. Jak to děláme? Obvykle k mailu vytvoříme šablonu, prostý textový soubor, který může vypadat třeba takto:

```
Mil[ý|á] $osloveni,

byl[a] jsi vybrán[a] jako $co na soustředění...
```

a pak seznam lidí, kterým se má poslat:

```
kc@example.net:Květoslave Čeňku:M:náhradník
po@example.net:Pokusná Osobo:F:účastník
hroch@example.net:Hrochu:M:maskot
```

A zbytek zařídí jednoduchý shellový skript. Pojďme si ho zkusit napsat. Stačilo by nám vymyslet, jak vyrobit z šablony mail pro jednoho konkrétního adresáta, hromadné zpracování už pak snadno zajistíme nějakým `while read`.

Potřebovali bychom umět v textu nahradit všechny řetězce v nějakém tvaru (např. `[`*něco*`|`*něco*`]`) za jiné řetězce, a ještě k tomu v náhradě použít nějaké části řetězce původního.

Regexy

Regex alias regulární výraz je řada písmenek a speciálních znaků, která právě dokáže popsat „řetězce nějakého tvaru", jako v příkladu výše. O libovolném řetězci lze rozhodnout, jestli danému výrazu *vyhovuje* (má správný tvar), nebo nikoli. Regex tak vlastně popisuje nějakou (potenciálně nekonečnou) množinu řetězců. S něčím podobným jsme se již setkali: byly to wildcardy. Například nahrazovaný řetězec z příkladu výše bychom se mohli pokusit popsat wildcardem `\[*\|*\]`. Ale možnosti wildcardů jsou poměrně omezené; s regexy se dají dělat mnohem zajímavější věci.

Níže najdete seznam konstrukcí použitelných v regexech. Literál je libovolná posloupnost znaků, které nemají speciální význam (nejsou použity v prvním sloupečku tabulky).

Kulaté závorky lze vynechat tam, kde by se v nich nacházel jediný znak nebo jiný nedělitelný element (např. množina), v případě alternativ, když by měly být kolem celého regexu.

Pár příkladů:

- `[a-zA-Z_][a-zA-Z0-9_]*` vyhovuje platný identifikátor, jak je definován většinou programovacích jazyků – tedy neprázdná posloupnost písmen, číslic a podtržítek nezačínající číslicí.
- `([01]?[0-9]|2[0-3]):[0-5][0-9]` vyhovuje čas zapsaný v 24-hodinovém formátu (např. `0:42`).

Podrobnější úvod do regexů najdete v seriálu 23. ročníku[5] či v desítkách internetových tutoriálů.

Nepleťte si regexy s wildcardy! Sice řeší podobný problém (popis množiny řetězců), ale jinak spolu nemají nic společného. Například pozor na to, že `*` v regexech je kvantifikátor, který značí opakování toho, co stojí před ní, samostatně stojící `*` nemá smysl. Obdobou wildcardové hvězdičky je regex `.*`. Liší se také použitím: wildcardy interpretuje shell a dají se použít pouze pro hledání názvů souborů, regexy se obvykle předávají nějakým pomocným utilitám a používají se pro hledání kusů textu.

Použití regexů: hledání a nahrazování

Kde lze regexy v Unixu použít? Například v nám dobře známém příkazu `grep`. Pokud mu místo přepínače `-F` (Fixed, hledej pevný řetězec) dáme `-E`, hledá řetězec vyhovující nějakému regexu. E znamená Extended a zapíná takzvanou rozšířenou syntaxi regexů (tu jsme si vysvětlili v předchozí kapitole). Existuje ještě „základní" syntaxe (starší), která se používá, pokud `grep`-u nedáte žádný přepínač. Ta je ale matoucí a nekonzistentní, tak ji raději nepoužívejte. Za `grep -E` existuje zkratka `egrep`.

Nezapomeňte, že `grep` hledá řádky *obsahující* řetězec vyhovující regexu, ne řádky celé *vyhovující* regexu. Například řádek `abcd` nevyhovuje regexu `[a-z]`, ale obsahuje `a`, které mu vyhovuje, a tedy `echo abcd | grep -E '[a-z]'` jej vypíše. Pokud bychom chtěli hledat řádky celé vyhovující regexu, stačí použít `^`*regex*`$`.

Slíbili jsme spolehlivější zjištění ID uživatele `hroch`, zde je:

```
grep -E '^hroch:' /etc/passwd | cut -d: -f3
```

Tady hledáme slovo `hroch` pouze na začátku řádku a následované dvojtečkou, nezmate nás tedy uživatel `druhyhroch` či `hrochodlak`, ani někdo, kdo si nastaví jako shell `/bin/hrochsh`.

`grep` má ještě jeden přepínač, který je užitečný až s regexy: `-o`. Ten zajišťuje, že se nevypisují celé vyhovující řádky, nýbrž jen nalezené výskyty regexu. Pokud je na jednom řádku více výskytů, každý se vypíše na samostatný výstupní řádek. Například takto bychom mohli najít seznam všech identifikátorů (názvů funkcí, proměnných apod.) použitých v nějakém programu:

```
grep -Eio '[a-z_][a-z0-9_]*' program.c | sort -u
```

Pro plnou funkčnost bychom ještě museli odfiltrovat komentáře, stringové literály a klíčová slova. K tomu by nám mohl pomoci nástroj, který si představíme za chvíli.

Syntaxe	Význam	Příklad	Vyhovují	Nevyhovují
literál	Řetězec shodný s literálem.	`bagr`	'bagr'	'bagrovat', 'lopata'
`\`*speciální-znak*	Escapuje speciální znak (udělá z něj literál).	`*`	'*'	'*', '\'
`.`	Libovolný znak.	`.`	'a', '%'	'abc', ''
`[`*množina*`]`	Libovolný znak patřící do množiny (jako v shellových wildcardech)	`[a-z_.]`	'q', '.'	'A', '-', 'aa'
`(`*regex1*`|`*regex2*`|`...`)`	Řetězec vyhovující alespoň jedné z možností.	`hro(ch|sik)`	'hroch', 'hrosik'	'hrosi'
`(`*regex*`)*`	Nula nebo více opakování.	`.*`	'', 'abc 123'	
`(`*regex*`)+`	Jedno nebo více opakování.	`(ab)+`	'ab', 'ababab'	'', 'baba'
`(`*regex*`)?`	Nepovinný prvek (0-1 opakování).	`[1-9]?[0-9]`	'5', '42'	'01', '333'
`(`*regex*`){`*m*`}`	*m* opakování.	`[A-Z]{2}`	'AA', 'ZQ'	'X', 'ERF'
`(`*regex*`){`*m*`,`*n*`}`	*m* až *n* opakování (obojí včetně).	`[0-9]{1,3}`	'4', '007'	'1337', '', 'xyz'
`^`	Začátek řádku.			
`$`	Konec řádku.			

[5] `http://ksp.mff.cuni.cz/viz/23-1-7`

Zbývá si rozmyslet, jak $S_s(v)$ určit. Pro jednoduchost uvažujme $s = P$, ostatní směry vyřešíme obdobně. Označme si p rodiče vrcholu v. Pokud čtverec odpovídající v leží v levé polovině čtverce p, stačí vzít správného sourozence v. Např. je-li v levým dolním synem p, $S_P(v)$ je pravý dolní syn p. V opačném případě se podíváme na vrchol $u := S_P(p)$ (p je ve vyšším patře, tedy jeho S už máme spočítané). Pokud u je list, pak $S_P(v) = u$. Pokud není list, víme, že leží na stejné úrovni jako p (kdyby ležel výš, nebyl by *nejhlubším* sousedem p, neb některý z jeho synů by také sousedil s p a byl hlouběji). V takovém případě $S_P(v)$ musí být jeden ze synů u, ten, který správně přiléhá k v. Například je-li v pravým horním synem p, pak $S_P(v)$ je levý horní syn u. Zvlášť musíme ošetřit případ, kdy v je v daném směru na kraji obrázku.

Určení $S_P(v)$ nás stojí konstantní čas, tedy celý průchod stromu zvládneme v lineárním čase a v jeho průběhu sestrojíme graf sousednosti jednobarevných čtverců. A už víme, že sestrojení kvadrantového stromu, rozklad grafu sousednosti na komponenty a výběr největší zvládneme rovněž v lineárním čase. Tedy i celý algoritmus si vystačí s $\mathcal{O}(n)$ času i paměti, kde n je délka vstupního kódu.

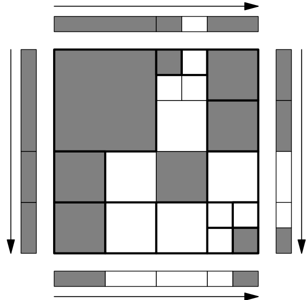
Program (Python):
`http://ksp.mff.cuni.cz/viz/27-3-1-bfs.py`

Řešení průchodem do hloubky

Předchozí řešení vytvářelo sousednosti po hladinách shora dolů. Ukážeme jiný způsob, který postupuje naopak zdola nahoru a dosahuje stejné časové složitosti.

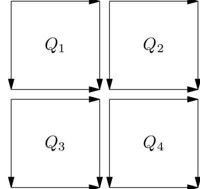
Strom budeme procházet do hloubky a průběžně budovat graf sousednosti. Přesněji řečeno, kdykoliv se při procházení stromu budeme vracet z nějakého čtverce, budou už zaznamenány všechny sousednosti mezi jeho potomky. Sousednosti vedoucí přes hranici aktuálního čtverce doplníme později.

Proto se nám bude hodit předávat do vyšších pater stromu informace o tom, jaké černé a bílé podčtverce leží na hranici aktuálního čtverce. To uložíme do čtyř seznamů. *Levý* seznam bude popisovat podčtverce přiléhající k levé hraně aktuálního čtverce, uspořádané shora dolů. Z každého podčtverce nás zajímá jen to, jak se dotýká hranice, což je nějaký *interval* y-ových souřadnic. Za každý interval přidáme do seznamu dvojici (v, r), kde v je příslušný vrchol grafu sousednosti a r řád podčtverce. Podobně vytvoříme *pravý* seznam (též uspořádaný shora dolů), *horní* a *dolní* (oba zleva doprava).



Vracíme-li se z jednobarevného čtverce (listu stromu), nemusíme vytvářet žádné sousednosti. Všechny čtyři seznamy budou obsahovat odkazy na tento čtverec.

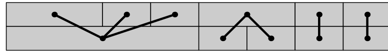
Zajímavější věci se dějí, pokud se vracíme z vícebarevného čtverce (vnitřního vrcholu stromu). Nechť Q_1 až Q_4 jsou kvadranty aktuálního čtverce. Z rekurze už známe sousednosti uvnitř kvadrantů a také podčtverce na hranicích kvadrantů. Nyní potřebujeme rozpoznat sousednosti vedoucí přes hranice kvadrantů a sestrojit hranici celého čtverce.



Nejprve vyřešíme hranici: levý seznam celého čtverce získáme spojením levého seznamu kvadrantu Q_1 s levým seznamem kvadrantu Q_3. Podobně získáme ostatní seznamy slepením seznamů z vnějších hranic kvadrantů.

Nyní sousednosti: uvažujme čtverce sousedící přes společnou hranici kvadrantů Q_1 a Q_2 (ostatní hranice zpracujeme obdobně), tedy intervaly z pravého seznamu Q_1 a levého seznamu Q_2. Kdykoliv se interval z Q_1 překrývá s intervalem z Q_2, chceme propojit hranou příslušné vrcholy v grafu sousednosti.

Propojování probíhá takto: podíváme se na počáteční interval každého seznamu – těmto intervalům říkejme třeba x a y. Pokud jsou stejného řádu, vytvoříme hranu. Pokud se řády liší (bez újmy na obecnosti x má vyšší řád), vezmeme x a z protilehlého seznamu budeme odebírat intervaly, dokud se jejich délky nenasčítají na délku x. (Všimněte si, že k tomu musí dojít přesně na hranici intervalu, neboť všechny intervaly vznikly postupným půlením délky hrany celého obrázku.) Kdykoliv odebíráme nějaký interval, vytvoříme hranu. Poté pokračujeme zbytkem obou seznamů, než se oba vyprázdní. Dopadne to například takto:



(Pokud vám na obrázku chybí hrana např. mezi třetím a čtvrtým intervalem v prvním řádku, uvědomte si, že odpovídá sousednosti v kolmém směru, takže vznikne při propojování jiných seznamů než těchto dvou.)

Umíme tedy zpracovat jak listy stromu, tak jejich vnitřní vrcholy. Nakonec se prohledávání vrátí z kořene stromu a graf sousednosti je hotov.

Zbývá určit časovou složitost. V každém vrcholu stromu strávíme konstantní čas spojováním seznamů – to je celkem $\mathcal{O}(n)$ za celý strom – a nějaký další čas zpracováním vnitřních hranic. Vnitřní hranice přitom může být poměrně komplikovaná, ale stačí si uvědomit, že kdykoliv sáhneme na nějaký interval, vypadne tento interval ze svého seznamu a už se nikdy do žádného seznamu nevrátí.

Čas strávený zpracováním všech hranic dohromady je tedy shora omezen počtem všech intervalů, které vložíme do seznamů. Vkládáme ale pouze v listech stromu: 4 intervaly za každý list. Tak vznikne celkem $\mathcal{O}(n)$ intervalů, takže jejich odebíráním strávíme čas $\mathcal{O}(n)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-3-1-dfs.c`

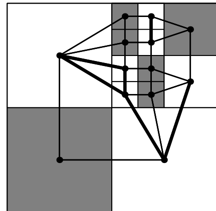
Vzorová řešení třetí série dvacátého sedmého ročníku KSP

27-3-1 Plocha k přistání

Většina z vás asi ví, jak se největší bílá oblast hledá, pokud jde o obyčejný bitmapový obrázek. Na obrázek se lze dívat jako na graf, kde pixely představují vrcholy a každá dvojice sousedních bodů je spojena hranou (tedy graf má tvar mřížky). V takovémto grafu bychom chtěli najít „komponenty bílé souvislosti“ – tedy maximální souvislé části grafu, které jsou celé bílé. K tomu můžeme použít klasický algoritmus na hledání komponent souvislosti prohledáváním do šířky či hloubky (pokud jej neznáte, nahlédněte do naší grafové kuchařky),[14] jen s tím rozdílem, že při prohledávání zcela ignorujeme černé vrcholy – vůbec je nenavštěvujeme. Snadno si rozmyslíte, že takto získáme očekávaný výsledek. Průběžně počítáme velikost nalezených komponent a nakonec jen vybereme maximum. Na tento postup jde pohlížet také jako na opakované použití klasického algoritmu *flood fill*.

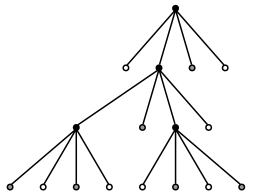
Zkusme se tím inspirovat. I kvadrantový obrázek má něco jako „pixely“, tedy elementární jednobarevné plochy – jsou to všechny nepodrozdělené čtverce. Jen je každý jinak velký a může mít víc než čtyři sousedy. Navíc není vůbec jasné, jak tyto sousedy najít. Ale pokud by nám někdo dal graf, jehož vrcholy jsou elementární čtverce ohodnocené svou plochou a hrany představují jejich sousednosti, dokázali bychom už jednoduše úlohu vyřešit.

Graf sousednosti může vypadat například takto (tučně jsou vyznačeny bílé komponenty):



Zbytek řešení bude pojednávat o tom, jak si takový graf pořídit, a to hned dvěma různými způsoby. Na úvod ovšem povězme pár věcí oběma řešením společných. Řešení víceméně každé úlohy nad kvadrantovým kódem začíná tím, že z kódu vytvoříme takzvaný *kvadrantový strom*. Kořen tohoto stromu reprezentuje celý obrázek. Pokud je jednobarevný, pamatuje si svou barvu a nemá žádné potomky; v opačném případě má právě čtyři syny představující kvadrantové stromy pro jednotlivé čtvrtiny.

Strom pro kvadrantový kód `(0((1010)1(0101)0)10)` (odpovídá obrázku výše) vypadá takto:



[14] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

Každý vrchol kvadrantového stromu představuje čtverec $2^r \times 2^r$, kde r je takzvaný *řád* vrcholu (čtverce). Kořen má řád h (výška stromu), jeho synové $h-1$, atd.

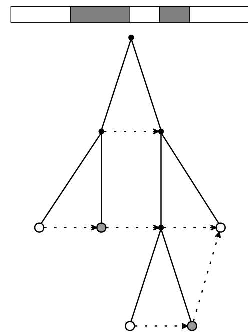
Strom vytvoříme přímočarou rekurzivní funkcí. Pokud se vyhneme zbytečnému kopírování řetězců (například si kód uložíme v globální proměnné a rekurzivním voláním budeme předávat jen index znaku, od kterého začít), zvládneme to v lineárním čase. Podrobněji ve zdrojáku.

Řešení průchodem do šířky

Graf sousednosti vytvoříme tak, že pro každý jednobarevný čtverec (tomu odpovídá list kvadrantového stromu) najdeme seznam jednobarevných čtverců s ním v obrázku sousedících. To je ale těžké, neb jeden čtverec může mít sousedů hodně, a to i v docela vzdálených částech stromu. Nám ovšem stačí, když každou sousednost „objeví“ jen jeden ze zúčastněných čtverců, např. ten menší. Pak nám stačí pro každý čtverec hledat jen sousedy stejného nebo vyššího řádu (příslušné listy jsou ve stromu stejně vysoko nebo výš). A takový je v každém směru nejvýše jeden. Tím získáme všechny hrany grafu sousednosti orientované směrem od menšího čtverce k většímu, opačný směr doplníme jejich otočením, což můžeme dělat i průběžně. Z toho je také hned vidět, že hran je lineárně mnoho.

Samotné hledání sousedů provedeme průchodem stromu po patrech (neboli do šířky od kořene). Když navštívíme nějaký vrchol v, určíme jeho nejhlubšího (obrázkového) souseda na každé straně, který ale není hlouběji než on sám. Toho si označíme $S_s(v)$ (kde $s \in \{L, P, H, D\}$ je příslušná strana). Všimněme si, že pokud je v tím hlubším z nějaké dvojice sousedních listů, je $S_s(v)$ právě druhým vrcholem této dvojice, a tedy jsme objevili jednu hranu grafu sousednosti (a po průchodu všech vrcholů objevíme všechny). V opačném případě je $S_s(v)$ nějaký vnitřní vrchol stromu, který se ale bude dále při hledání hodit.

To by si určitě zasloužilo obrázek. Abyste se neztratili v obrovském množství čar, ukážeme si to na příkladu jednorozměrného „kvadrantového“ obrázku. To je dlouhá „nudle“ rozměrů $2^k \times 1$, která je buď celá bílá, celá černá, nebo rozdělená na dvě poloviny, rekurzivně splňující stejnou definici. Kvadrantový strom takovéhoto obrázku je binární a má tu výhodu, že směr vlevo a vpravo ve stromu odpovídá stejnému směru v obrázku.



Tečkované šipky značí ukazatele S_P (vedou od v k $S_P(v)$).

Zamysleme se ještě nad jednou věcí. Pokud je v programu například identifikátor `bagr`, nachází se uvnitř něj i další platné identifikátory, jako např. `ag`. `grep -o` v takovém případě udělá to, co téměř vždy chceme: z každé množiny překrývajících se výskytů vypíše pouze ten nejlevější a nejdelší (což bude přesně odpovídat identifikátorům v programu doopravdy použitým).

Při předávání regexů jako parametrů příkazům je třeba dát si **velký pozor** na to, že mnoho regexových speciálních znaků má zvláštní význam i pro shell, a tedy pokud chceme, aby se např. ke `grep`-u dostaly nezměněné, musíme je před shellem zaescapovat. Nejjednodušším řešením je psát všechny regexy do apostrofů, kde se nic escapovat nemusí.

Pokud přece jen z nějakého důvodu escapovat budeme, je třeba si uvědomit, že máme co do činění se dvěma úrovněmi escapování. Například pokud napíšeme `grep ^* soubor`, nejprve dostane řetězec do rukou shell, který ví, že `\\` ve skutečnosti znamená `\` atp., všechny escapy odstraní a `grep`-u předá jako první parametr řetězec `^*`. `grep` zase ví, že `*` znamená literál „`*`“, tedy tento příkaz tedy vybere ze souboru řádky začínající hvězdičkou.

Když už umíme výskyty regexů v textu hledat, hodilo by se také umět je nahrazovat něčím jiným. Řešení si představíme zatím jen jako zaklínadlo `sed -re 's/`*regex*`/`*náhrada*`/g'`. Nebojte, za chvíli si ho vysvětlíme.

Jak jsme slibovali na začátku, v textu náhrady se lze odkazovat na části původního textu: přesněji na obsah libovolné kulaté závorky. Obalit závorkou můžeme cokoliv, aniž by se tím změnil význam regexu. Počítají se všechny závorky, včetně těch vynucených např. kvůli ohraničení skupiny alternativ. Obsah závorky do náhrady vložíme speciální sekvencí `\`*číslo*, kde číslo je pořadové číslo závorky (přesněji řečeno, páry závorek se číslují od jedničky v pořadí jejich otevíracích závorek). To mimo jiné znamená, že i v náhradě musíme escapovat zpětná lomítka, pokud je tam chceme vložit doslovně.

Například pokud chceme v textu nahradit slovo `bagr` ve všech tvarech za slovo `kombajn`, můžeme použít příkaz

```
sed -re 's/bagr(|u|em|y|ů|ům|ech)/kombajn\1/g'
```

Tenhle trik samozřejmě funguje pouze, pokud mají slova stejný (pod)vzor a žádné nepravidelnosti.

Řešení: Spam

Nyní už máme téměř vše, co potřebujeme k řešení spamovacího příkladu. Můžeme si jej zkusit načrtnout. Předpokládejme, že v shellových proměnných `$osloveni`, `$pohlavi` a `$co` máme příslušné údaje k vyplnění.

```
tvar=$(echo $pohlavi | tr MF 23)
<sablona sed -re "s/\\\$osloveni/$osloveni/g" \
   | sed -re "s/\\\$co/$co/g" \
   | sed -re 's/\[((.*)\|)?(.*)\]/\'$tvar/g
```

Jak to funguje? První dva `sed`-y jsou přímočaré nahrazení jednoho řetězce za jiný, jen pozor na escapování dolarů (od shellu i regexu). Poslední regex hledá řetězce tvaru `[`*něco*`|`*něco*`]`, kde každé „něco“ načte do jedné závorky. V proměnné `$tvar` pak máme číslo závorky, kterou chceme vybrat. Parametr `sed`-u bude po expandování proměnných vypadat např. jako:

```
s/\[((.*)\|)?(.*)\]/\3/g
```

První část („*něco*`|`“) je nepovinná, pokud mužský tvar není uveden, předpokládá se prázdný.

Toto řešení skoro funguje, ale ještě ne úplně. Zapomněli jsme totiž upozornit na jednu věc: regexy jsou *žravé*. To znamená, že při hledání vždy vyberou nejdelší kus textu, který jim vyhovuje. Tedy například pro text `Mil[ý|á] účatn[íku|ice]` nebudou nalezeny očekávané dva výskyty, nýbrž jeden velký, kde v první závorce skončí „`ý|á] účatn[íku`“ a v druhé „`ice`“. Snadno si rozmyslíte, že to vyhovuje našemu regexu. Nejjednodušeji to spravíme tak, že uvnitř jednotlivých tvarů zakážeme používat `|` a `]`. Tedy místo `.*` musíme psát `[^]|]*` (stříška na začátku množiny znamená doplněk a pokud napíšeme `]` jako první, neukončí se tím množina, nýbrž do ní vložíme znak `]`).

Nyní už to opravdu dělá, co má. Celý rozesílací skript by mohl vypadat takto:

```
cat adresati \
| while IFS=: read mail osloveni pohlavi co; do
t=$(echo $pohlavi |tr MF 23)
<sablona sed -re "s/\\\$osloveni/$osloveni/g" \
 | sed -re "s/\\\$co/$co/g" \
 | sed -re 's/\[(([^]|]*)\|)?([^]|]*)\]/\'$t/g
 | mail -s "Pozvánka na soustředění" "$mail"
done
```

Kde `mail` je jedním z mnoha příkazů, které umožňují odeslat e-mail. K tomu samozřejmě potřebuje správné nastavení, které je nad rámec tohoto seriálu. Parametrem `-s` určujeme předmět.

sed pro pokročilé

Je na čase naše `sed`ové zaklínadlo rozklíčovat. `sed` ve skutečnosti dostane text a aplikuje na něj posloupnost příkazů. Tu mu můžeme předat buď jako parametr s použitím přepínače `-e` *příkaz*, nebo načíst ze souboru parametrem `-f` *soubor* (hodí se pro delší a složitější příkazy; v souboru nemusíme řešit shellové escapování). Jednotlivé příkazy se oddělují středníkem nebo koncem řádku. Též můžeme více příkazů zadat několikanásobným použitím příkazu `-e`.

Přepínač `-r` zapíná rozšířenou syntaxi regexů, podobně jako u `grep`-u `-E`. Doporučujeme používat v podstatě vždy.

Všechny příkazy mají jednoznakový název. My jsme dosud používali příkaz `s` (substitute) sloužící k nahrazování. Ten má tvar `s/`*regex*`/`*náhrada*`/`*modifikátory*. Místo lomítek můžeme použít jakýkoli jiný znak (kromě písmen). Můžeme se tak vyhnout problémům s regexy obsahujícími lomítka. Oblíbenými volbami jsou např. `|` či `#` pro svou vizuální výraznost. Modifikátor `g` (global) znamená nahrazení všech výskytů na řádku (jinak by se na každém řádku nahradil jen první). Dalším zajímavým modifikátorem `i` (ignoruj velikost písmen).

`sed` zpracovává vstup po řádcích a na každém z nich zvlášť provede všechny příkazy v pořadí, ve kterém jsou zapsány. Ve skutečnosti se každý řádek načte do řetězcové proměnné, které se říká *pattern space*, na ní se provádějí jednotlivé příkazy a po jejich skončení je výsledná hodnota pattern space vypsána na výstup, není-li `sed` spuštěn s parametrem `-n` („nevypisovat“).

Před většinu příkazů lze napsat takzvanou *adresu* a tím určit, že se budou provádět jen na některém řádku či řádcích. Adresou může být například:

- *číslo* – Této adrese vyhovuje právě tolikátý řádek vstupu, počítáno od jedničky.
- `$` – Poslední řádek.

- `/`*regex*`/` – Řádek obsahující výskyt regexu. Chceme-li použít místo lomítka jiný oddělovač, musíme na začátek napsat backslash, např. `\#/dev/null#` adresuje řádky obsahující řetězec „`/dev/null`“.
- *adresa*`!` – Negace adresy, vyhovují řádky nevyhovující původní adrese.

Nyní už má smysl vysvětlit si některé další příkazy, které bez adresování nejsou příliš užitečné:

- `p` – vypíše aktuální obsah pattern space na výstup. Nejčastěji se používá spolu s parametrem `-n` pro selektivní vypisování řádků. Například `sed -nre '/`*regex*`/ p'` je to samé, jako `grep -E '`*regex*`'`. Kombinací `s/^.*$/`*text*`/` a `p` můžeme na výstup vypsat libovolný text.
- `d` – „smaže“ řádek (zabrání jeho vypsání a skočí na další). Tedy `sed -re '/`*regex*`/ d'` funguje jako `grep -vE`.
- `y/`*znaky*`/`*znaky*`/` – nahradí znaky z jednoho seznamu za odpovídající znaky z druhého, stejně jako `tr`, včetně stejného zápisu výčtů a rozsahů.

Příkazu můžeme dát místo jedné adresy také dvojici adres oddělených čárkami. Tím zadáváme rozsah – příkaz se provádí na všech řádcích od první adresy po druhou. Přesněji řečeno kdykoli se narazí na řádek vyhovující první adrese, příkaz se začne provádět a když se narazí na řádek vyhovující druhé adrese, zase se provádět přestane. Rozsah vždy zahrnuje oba krajní řádky (vyhovující příslušným adresám). Takto se může příkaz provést i pro několik bloků řádek, pokud každý z nich začíná řádkem vyhovujícím první adrese a končí řádkem vyhovujícím druhé.

Některé formáty (například e-mailové zprávy) mají takovou strukturu, že obsahují nejprve hlavičky (s informacemi jako odesilatel, předmět atp.), potom prázdný řádek, a teprve za ním tělo (text zprávy). Pokud bychom chtěli blok hlaviček odstranit, můžeme použít příkaz:

```
sed -re '1,/^$/ d'
```

Ukázky, jak pomocí `sed`-u nahradit `grep`, nebyly samoúčelné. `sed` totiž oproti `grep`-u má jednu velkou výhodu, totiž přepínač `-i` (inplace). Již v prvním díle jsme se bavili o tom, že nemůžeme z jednoho souboru zároveň načítat a zároveň do něj zapisovat (`grep slovo <soubor >soubor` neudělá to, co byste chtěli). `sed -i` tohle umí zařídit. Jen mu dáte jako parametr (tedy nikoli shellové přesměrování) název souboru, a on z něj načte vstup a do toho samého souboru uloží svůj výstup. Interně to dělá tak, že vytvoří dočasný soubor, do kterého výstup zapisuje, a po svém skončení s ním nahradí (pomocí ekvivalentu příkazu `mv`) atomicky původní soubor. Tedy `sed -i -re ... soubor` je ekvivalentní posloupnosti příkazů:

```
sed -re ... <soubor >soubor.tmp
mv soubor.tmp soubor
```

Toto je velmi častý unixový idiom, který je dobré si zapamatovat. Hodí se nejen když chceme zapisovat do stejného souboru, ze kterého čteme, ale víceméně kdykoli nahrazujeme či vytváříme nějaký soubor. Tím, že data zapisujeme do nějakého dočasného souboru, který kromě nás nikdo jiný nepoužívá, a teprve když je „hotový“, jej přesuneme na cílové místo, se nemůže stát, že se nějaká jiná aplikace pokusí číst soubor v průběhu vytváření, kdy ještě neobsahuje smysluplná data. Také pokud můžeme snadno zajistit (bashovým operátorem `&&`), že pokud náš příkaz modifikující nějaký soubor selže, `mv` se neprovede a zůstane zachovaná původní verze.

sed pro šílence

Se `sed`-em se dají dělat i větší šílenosti. Jeho příkazy tvoří vlastně jednoduchý programovací jazyk.[6] Kromě pattern space máte k dispozici ještě druhou(!) stringovou proměnnou, které se říká *hold space*. Pomocí příkazů `h` a `H` můžete aktuální obsah pattern space zapsat do hold space, resp. připojit na jeho konec. `g` a `G` dělají to samé opačným směrem. Při připojování je nový obsah od původního oddělen znakem konce řádku (`\n`). Příkazem `x` lze prohodit obsah pattern a hold space.

Pokud bychom si chtěli řídit načítání řádek nějak přesněji, než že pro každý řádek je náš program spuštěn znovu od začátku, můžeme. K tomu slouží příkazy `n`, který do pattern space další řádek (původní zahodí) a `N`, který připojí další řádek na konec pattern space (oddělený `\n`, podobně jako `G`). To vše se děje stále v rámci jedné iterace našeho skriptu. Pokud doběhne na konec, `sed` automaticky načte první ještě nenačtený řádek do pattern space a spustí náš program znovu od začátku. Tyto příkazy nám umožňují nezpracovávat jednotlivé řádky jen nezávisle, ale dělat i nějaké složitější úpravy napříč řádky.

Proměnné už máme, ale správný programovací jazyk ještě potřebuje nějaké řídící konstrukce. `sed` nabízí návěští (`:` *název*), skoky (`b` *název*) a podmíněné skoky (`t` *název*). Podmíněný skok se provede, pokud od posledního podmíněného na tomto řádku vstupu došlo k alespoň jednomu úspěšnému nahrazení příkazem `s` (existuje i verze `T`, která má podmínku invertovanou).

Pokud chceme testovat, zda uspěl jeden konkrétní nahrazovací příkaz, musíme si nejdřív případné dřívější úspěchy na daném vstupním řádku „vyresetovat“ prostřednictvím příkazu `t`:

`t reset; : reset; s/`*regex*`/`*náhrada*`/; t cil;`

Nechceme-li nic nahrazovat a rádi bychom jen skákali podle toho, zda pattern space vyhovuje nějakému regexu, můžeme použít příkaz `b` podmíněný adresou: `/`*regex*`/ b`.

Všechny verze skoku při vynechání argumentu skáčou za poslední příkaz. `q` a `Q` ukončí celý `sed` s vypsáním aktuálního pattern space, resp. bez něj. Umí nastavit návratovou hodnotu. Dále existují příkazy `r` a `w` pro čtení/zápis z/do pomocných souborů. Příkazy `[qQT]` jsou rozšířením GNU `sed`-u (nejběžněji se vyskytující verze) a nemusí být dostupné v jiných verzích.

Zkusme si například napsat skript, který spojuje příkazy rozdělené na několik řádek pomocí zpětných lomítek do jednoho řádku:

```
sed -re ': loop; s/\\$//; T; N; s/\n//; t loop'
```

Ukázkový vstup:	*Ukázkový výstup:*
`prvni`	`prvni`
`dr\`	`druhy`
`uh\`	
`y`	

Už byste měli zvládnout si rozmyslet, jak funguje.

[6] Je turingovsky úplný, ale asi v podobném smyslu, jako Brainfuck. Dá se najít implementace Turingova stroje v `sed`-u. Nebo Sokoban. Zagooglete si.

nici, 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 kapitolu připravovaných skript z předmětu ADS na Matfyzu.[12]

Více o třídě $\mathcal{NP}$ i o dalších aspektech složitosti můžete najít na stejné adrese, nebo zkuste vynikající anglicky psanou knížku *Algorithms* od profesorů exotických jmen Dasgupta, Papadimitriou a Vazirani.

Jak už jsme zmínili, existují i problémy, které jsou mimo $\mathcal{P}$ i $\mathcal{NP}$, a dokonce existuje spousta různých dalších tříd problémů. Je jich celá zoologická zahrada – pěkný souhrn můžete najít na stránkách Univerzity ve Waterloo.[13]

Seznam $\mathcal{NP}$-úplných problémů

Sedíte-li nad zatím nevyřešenou úlohou, kterou stále nemůžete rozlousknout, je možné, že bude $\mathcal{NP}$-úplná. Abyste mohli mezi $\mathcal{NP}$-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 $\mathcal{NP}$-úplné. Převody se nám sem už sice nevešly, ale většinu z nich (ne-li všechny) zvládnete vymyslet sami – zkuste 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 {muž, žena, 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 sepsali

Martin Böhm & Jirka Setnička

[12] http://mj.ucw.cz/vyuka/ads/49-prevody.pdf
[13] https://complexityzoo.uwaterloo.ca/

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

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

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

To postačí, protože pak libovolný jiný problém v $\mathcal{NP}$ nejprve převedeme na zvolený $\mathcal{NP}$-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 $\mathcal{NP}$-úplnosti jednoho problému na příkladu, pokud nám uvěříte, že již probíraný problém *Hamiltonovská kružnice* je $\mathcal{NP}$-úplný. Nejprve zadefinujme jiný problém:

Název problému: Hamiltonovská cesta.

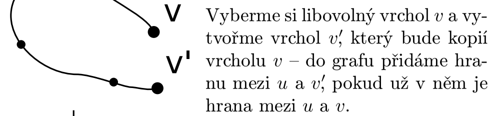
Vstup: Neorientovaný graf, dva speciální vrcholy x a y.

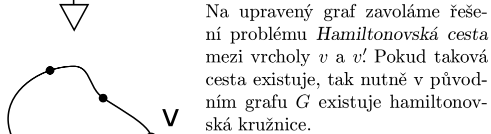
Problém: Existuje cesta z x do y (posloupnost vrcholů, ve které se žádné dva neopakují), která prochází každým vrcholem právě jednou?

Certifikát: Posloupnost vrcholů tvořící správnou cestu.

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

Důkaz $\mathcal{NP}$-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.

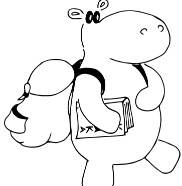
Na upravený graf zavoláme řešení problému *Hamiltonovská cesta* mezi vrcholy v a v'. Pokud taková cesta existuje, tak nutně v původním grafu G existuje hamiltonovská kružnice.

Cesta z vrcholu v' přesně odpovídá pokračování 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, zápis čísla B byl veliký $\mathcal{O}(\log_2 B)$, ale naše složitost závisela na $B = 2^{\log_2 B}$, tedy exponenciálně vzhledem k velikosti vstupu.

Problém batohu, respektive jeho rozhodovací verze, je dokonce $\mathcal{NP}$-úplný problém.

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

Mimochodem: pokud bychom na vstupu zapisovali čísla v unárním zápisu (tedy místo každého čísla by bylo třeba tolik hvězdiček, jakou hodnotu představuje), každý pseudopolynomiální problém by ležel v $\mathcal{P}$.

Poznámky na závěr

Otázku „Je třída $\mathcal{P}$ rovna $\mathcal{NP}$?" 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 $\mathcal{P} = \mathcal{NP}$, pak by mnoho lidí zajásalo – mnoho přirozených problémů, které nastávají i v reálném životě, by najednou bylo řešitelných 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 $\mathcal{NP}$-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 $\mathcal{NP}$-úplnými problémy je užitečné i v reálném světě – mnohdy jde vymyslet třeba dobrá aproximace řešení.

Například nenajdeme hamiltonovskou kružnici v polynomiálním čase, ale nalezneme nějakou relativně dlouhou kruž-

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

Další nástroje

Posledním významným nástrojem, který jsme nezmínili, je `awk`. To by nejspíš vydalo na samostatný díl. Slouží primárně k složitější práci s tabulkovými soubory. Na rozdíl od `cut`-u umí používat složitější než jednoznakové oddělovače, např. libovolnou posloupnost bílých znaků (ta je u `awk` dokonce výchozím oddělovačem). To se hodí při práci se soubory, jako je `/etc/fstab`. Program v `awk` je podobně jako v `sed`-u posloupnost příkazů, která se spouští pro každý řádek a příkazům lze předřadit podmínku omezující, na kterých řádcích běží. Jazyk `awk` má daleko blíže k plnohodnotnému programovacímu jazyku než sed: má pojmenované proměnné, asociativní pole a další vychytávky.

Uvedeme jen několik málo příkladů použití, měly by být zčásti samovysvětlující, zčásti interpretovatelné s pomocí manuálové stránky:

- Vyseknutí sloupečku odděleného obecnou posloupností bílých znaků z tabulky:
  ```
  awk '{print $2}' /etc/fstab
  ```
- Nalezení uživatele s daným ID:
  ```
  awk -F: '($3 == 0) { print $1 }' /etc/passwd
  ```
- Sečtení všech (číselných) řádků v souboru:
  ```
  awk '{ sum += $1 } END { print sum }'
  ```

Postavení `awk` na půli cesty mezi jednoduchou utilitkou a plnohodnotným programovacím jazykem z něj činí trochu zvláštní nástroj. Někdy je lepší použít místo něj `cut` či `sed`, jindy naopak opravdový programovací jazyk. Na zpracování textu nejlépe poslouží Perl, který má bohatou podporu pro regexy a spoustu syntaktických zkratek, jež v něm umožňují většinu jednoduchých věcí napsat podobně krátce jako v jednoúčelových nástrojích.

Jakou malou ochutnávku Perlu vám ukážeme skript, který z HTML dokumentu vypíše titulky všech hypertextových odkazů:

```
perl -0777 -ne 'for (m{<a.*?>.*?</a>}gcs) {
    s/<.*?>//g; s/(^\s+|\s+$)//g;
    print "$_\n" if $_;
}'
```

Úkoly

V řešení se nebojte používat pomocné soubory, kde je to na místě, klidně pro jednoduchost s pevnými jmény. Můžete používat vše, co jsme se naučili, a další utilitky podobného ražení, můžete používat `awk`. Perl ani jiné „velké" programovací jazyky nepoužívejte.

Úkol 1 [2b]: Ve vstupním souboru máte seznam jmen (sudý počet, jedno na řádek). Napište skript, který z nich vytvoří náhodné dvojice a vypíše je na výstup ve formátu *první osoba*`:`*druhá osoba*.

Ukázkový vstup:	*Ukázkový výstup:*
A	C:A
B	B:D
C	
D	

Úkol 2 [4b]: Napište skript řešící úlohu 27-Z3-2.[7] Ve vstupním souboru dostanete slovník (jedno slovo na řádku), na výstup vypište nejdelší slovo, které má ve slovníku i svou verzi napsanou pozpátku.

Ukázkový vstup:	*Ukázkový výstup:*
kecup	kecup
ves	
vrabec	
pucek	
sev	

Řešení s pomocí bashových cyklů je nudné, pro plný počet bodů to zkuste bez nich. Mohlo by se vám hodit `awk` a jeho funkce `length`.

Úkol 3 [4b]: V nějakém adresáři máte staženou spoustu dílů svého oblíbeného seriálu (z legálních zdrojů, pochopitelně). A jak už to tak u legálních zdrojů chodí, soubory jsou pojmenované naprosto neuspořádaně. Jediné, čím si můžete být jisti, je, že název obsahuje číslo série a číslo epizody v tomto pořadí, mezi nimi je alespoň jeden nečíselný znak a pro jednoduchost název žádné jiné číslice neobsahuje.

Napište skript, který stáhne z Wikipedie (či jiného příhodného zdroje) názvy dílů a všechny soubory přejmenuje tak, aby v jednotném formátu obsahovaly číslo série a číslo a název epizody. Chcete-li, můžete předpokládat, že všechny soubory jsou ze stejné série.

Možná se vám snáz než HTML bude parsovat zdrojový wikitext, který získáte připojením `?action=raw` na konec URL článku.[8]

Úkol 4 [4b]: Vylepšete příklad vypisující všechny identifikátory v Céčkovém programu tak, aby ignoroval obsah řetězců a komentářů, případně základní klíčová slova. Pro plný počet bodů by měl zvládnout jednořádkové (`// ...`) i víceřádkové (`/* ... */`) komentáře a neměl by se nechat zmást escapovanými uvozovkami a zpětnými lomítky v řetězcích. Ale určitě má smysl poslat i jednoduché či částečné řešení. Obskurnosti jako komentář uvnitř řetězce (nebo naopak) ošetřovat nemusíte, pokud vyloženě nechcete. Předpokládejte samozřejmě, že program je syntakticky správný.

Ukázkový vstup:

```
/* print a greeting
   with quoted name */
printf("Hello, \"%s\"",
       name);
```

Ukázkový výstup:

```
name
printf
```

Pokud si s Céčkem nerozumíte, můžete si vybrat nějaký jiný srovnatelně složitý (tedy třeba by měl ideálně mít víceřádkové komentáře či řetězce) programovací jazyk – třeba Python nebo Pascal.

Z cvičných důvodů zkuste nenačítat celý vstupní soubor do paměti najednou. Ano, bylo by to jednodušší a pro praktické účely možná nejlepší řešení, ale tolik se toho na něm nenaučíte.

Filip Štědronský

[7] `http://ksp.mff.cuni.cz/viz/27-Z3-2`
[8] `http://en.wikipedia.org/wiki/The_Big_Bang_Theory_(season_4)?action=raw`

Recepty z programátorské kuchařky: Těžké problémy

Občas se v informatice potkáme s problémem, který nám připadá skutečně těžký, s problémem, na který zatím nikdo nezná efektivní algoritmus. V tomto textu se pokusíme si lépe vysvětlit, co vlastně pro informatika znamená sousloví *těžký problém*.

Úvod a třída problémů $\mathcal{P}$

Když mluvíme o efektivních algoritmech řešících nějaký problém, tak většinou máme na mysli algoritmus běžící v nějakém polynomiálním čase ve vztahu k velikosti vstupu. Například pro problém se vstupem velikosti N to jsou algoritmy, jejichž časovou složitost v nejhorším případě lze omezit shora nějakým polynomem závisejícím na N (sem spadají časové složitosti jako třeba $\mathcal{O}(N)$, $\mathcal{O}(N \log N)$ nebo i $\mathcal{O}(N^5)$). Jestli v základech časové složitosti tápete, nahlédněte do naší kuchařky o složitosti.[9]

Pokud na problém existuje alespoň jedno známé polynomiální řešení, tak o problému můžeme prohlásit, že leží ve třídě $\mathcal{P}$, neboli skupině úloh řešitelných v polynomiálním čase (třída tu je jen pomocné označení pro nekonečnou množinu). Stále můžeme problém zkoumat a nacházet rychlejší polynomiální řešení, ale pro teorii složitosti stačí, že máme alespoň nějaké.

Jak je to ale s úlohami, u kterých žádný polynomiální algoritmus neznáme? To mohou být třeba problémy, kde nejlepší známé řešení vede přes vyzkoušení všech možností, a jejichž časová složitost je tak třeba $\mathcal{O}(2^N)$, neboli exponenciální. Je důležité si uvědomit, že funkce jako 2^N rostou mnohem rychleji, než jakékoliv polynomiální funkce (na názornou tabulku se můžete podívat do již zmíněné kuchařky o složitosti).

I takové problémy chceme nějakým způsobem zařadit do hierarchie složitostních tříd. Než tak ale učiníme, uděláme si malou odbočku – co kdyby nám někdo k problému poskytl i nápovědu, nějaký tahák?

S mapou v bludišti

Představme si, že jsme v bludišti a hledáme nejkratší cestu ven. Můžeme určitě použít prohledávání do šířky[10] a cestu najít v čase lineárním k 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.

Jak by se změnila naše situace, kdybychom si ale od kamaráda půjčili tahák – mapu bludiště s vyznačenou nejkratší cestou? Pak by stačilo držet se této cesty a vyběhli bychom nejkratší cestou ven, aniž bychom kdekoliv 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.

V tomto případě nám nápověda tedy zase tolik nepomohla, nalezení cesty z bludiště ven je totiž úloha, kterou umíme vyřešit v polynomiálním čase (patří do třídy $\mathcal{P}$). Pojďme si ale úlohu trochu zkomplikovat a podívejme se, jestli s nápovědou tentokrát umíme dosáhnout lepšího výsledku než bez ní.

Opět jsme v bludišti, ale tentokrát jsou na všech stanovištích umístěny koláčky. Labyrint je to zvláštní, cesty se v něm nekříží, ale je tam plno nadchodů a podchodů (lze tedy říci, že je to obecný nerovinný graf).

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 na žádném stejně 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, neboť nám může zablokovat další cestu. Zároveň ale posloupnost (začátek, druhé, první) už může být dobrá.

Nebude tedy už platit, že každý vrchol při prohledávání navštívíme maximálně jednou, ale každou *posloupnost* stanovišť navštívíme maximálně jednou. Takových posloupností je ale exponenciálně mnoho vzhledem k velikosti bludiště.

Pokud si pořídíme nový tahák, na kterém bude vyznačena optimální cesta přes všechna stanoviště, tak jsme na tom ale stále dobře. Tahák bude mít lineární velikost vzhledem k počtu stanovišť (cestou proběhneme každé z nich právě jednou) a umožní nám tak problém vyřešit v lineárním čase prostým následováním vyznačené cesty.

Našli jsme tedy problém, který nevíme jak vyřešit bez nápovědy v polynomiálním čase (a tedy ho nemůžeme s klidným svědomím zařadit do třídy $\mathcal{P}$), ale s pomocí nápovědy už to umíme. Nedá se takovým způsobem definovat také nějaká třída složitosti? Dá! A to dokonce velmi důležitá.

Certifikáty a nedeterminismus

Vraťme se znovu k naší úloze s koláčky v bludišti. Zde celý problém tkví v tom, že se v některých chvílích prohledávání musíme rozhodnout, jakou z mnoha možností zkusíme nejdříve. Kdybychom pokaždé zvolili správně, tak zvládneme bludiště projít v lineárním čase.

Typický algoritmus, které napíšeme, většinou v případě více možností pokračování zvolí tu první (jeho volba je pevně určená, říkáme jí *deterministická*). Také ale můžeme přemýšlet o algoritmu, který si na každém takovém místě hodí kostkou a podle toho se rozhodne. Takový algoritmus nám na stejném vstupu může dát při různých spuštěních různé výsledky – jeho výpočet není „předurčen" a proto mu říkáme *nedeterministický*.

Přidejme ale k nedeterministickému algoritmu naši nápovědu neboli *certifikát*. Je to nějaká (vzhledem k velikosti vstupu) polynomiálně velká informace. Můžeme si jej představit jako data, která náš program nalezne v pomocném vstupním souboru, ke kterému program z třídy $\mathcal{P}$ nemá přístup.

Certifikát nám pomůže v každém místě, kde nevíme kudy dál, zvolit tu správnou cestu. Bez něj bychom se museli zkusit vydat každou z nabízených možností, abychom objevili tu správnou, ale s jeho pomocí se vždy vydáme správně a existenci takové cesty ověříme rychle.

Pokud náš algoritmus s použitím takového věšteckého orákula (či křišťálové koule, chcete ji), jakým je certifikát, zvládne ověřit řešení problému v polynomiálním čase, říkáme o problému, že je *nedeterministicky polynomiální*, neboli že náleží do třídy $\mathcal{NP}$.

Rozhodovací problémy a třída $\mathcal{NP}$

Aby se nám problémy lépe formálně popisovaly a zařazovaly do tříd, omezíme se v dalším textu jen na *rozhodovací problémy*. To jsou vlastně otázky, na které existují jen dvě možné odpovědi: ANO, nebo NE. Například:

- Existuje cesta z bludiště délky k?
- Je součet čísel $8 + 3$ roven 5?

Jestli se obáváte, že to výrazně sníží množství problémů, které umíme řešit, tak se nemáte proč obávat – skoro vždy se rychlé řešení rozhodovacího problému dá převést na rychlé řešení příslušného vyhledávacího problému jen s nějakým malým zpomalením. Třeba nalezení délky nejkratší cesty z bludiště můžeme udělat pomocí binárního vyhledávání a opakovaného dotazu na existenci cesty délky k (detaily si jako cvičení domyslete).

V úvodu jsme si už řekli, že třída $\mathcal{P}$ představuje problémy řešitelné v polynomiálním čase (u rozhodovacího problému to bude znamenat, že existuje polynomiální algoritmus odpovídající na zadaný vstup korektně ANO, nebo NE). U třídy $\mathcal{NP}$ si ale už musíme dát trošku pozor.

Třída $\mathcal{NP}$ je také třídou problémů. Problém do ní náleží ve chvíli, kdy existuje algoritmus a ke každému zadání, na nějž má být odpověď ANO, navíc i certifikát, pomocí kterého zvládne algoritmus existenci řešení ověřit v polynomiálním čase. Ověřením se myslí to, že odpoví ANO tehdy a jen tehdy, když řešení skutečně existuje.

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 $\mathcal{NP}$!

Když si certifikát představíme jako ono orákulum, které nám vždy napoví správnou cestu, může být algoritmus nějaké nedeterministické řešení daného problému. Orákulum, ať bude napovídat jakkoliv špatně, ho nikdy nemůže přesvědčit o existenci nějakého řešení, pokud takové neexistuje.

V reálné situaci (při dokazování, viz níže) si pak často za orákulum (za certifikát) zvolíme optimální řešení úlohy, kterého se stačí držet a najdeme hledanou odpověď (třeba dokážeme, že existuje cesta kratší než k).

Příslušnost do třídy $\mathcal{NP}$ tedy znamená schopnost s pomocí certifikátu dokázat existenci kladného řešení. (To vůbec nemusí znamenat, že dovedeme dokázat jeho *ne*existenci – to by byla zase jiná třída, které se říká *co*$\mathcal{NP}$.)

Asi je vám jasné, že celá třída $\mathcal{P}$ (všechny problémy z ní) jsou součástí i třídy $\mathcal{NP}$ (stačí si za certifikát zvolit třeba prázdný soubor a problém vyřešit normálním polynomiálním algoritmem). A jak už jsme naznačili výše, existují i problémy, jež leží „ještě za třídou $\mathcal{NP}$", tedy takové, které neumíme vyřešit v polynomiálním čase ani s pomocí certifikátu. Ale dokazování toho, že takové problémy existují, už je nad rámec této kuchařky.

Je $\mathcal{P}$ rovno $\mathcal{NP}$?

Ukázali jsme, že celé $\mathcal{P}$ leží uvnitř $\mathcal{NP}$. Existuje však vůbec nějaký problém, který by byl v $\mathcal{NP}$, ale nebyl by v $\mathcal{P}$? To je otázka, jež trápí informatiky už mnoho let, jeden z nejslavnějších otevřených problémů informatiky.

Vezměme si za příklad 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 $\mathcal{NP}$, ale bez certifikátu už jej nelze řešit v polynomiálním čase. Kdyby takový neexistoval, třídy $\mathcal{P}$ a $\mathcal{NP}$ by se rovnaly. Díky převoditelnosti problémů v $\mathcal{NP}$, které si nyní ukážeme, by dokonce stačilo najít polynomiální řešení bez certifikátu na jediný $\mathcal{NP}$-úplný problém.

Převoditelnost a $\mathcal{NP}$-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 vhodným grafem.

Ukazuje se, že se ve třídě $\mathcal{NP}$ 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 $\mathcal{NP}$, ale není zároveň lehký (v $\mathcal{P}$). Nemůžeme jen tak ledabyle říci „je v $\mathcal{NP}$ a není v $\mathcal{P}$", protože to nevíme. To je právě ta slavná otázka.

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

Všechny problémy v $\mathcal{NP}$ 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 $\mathcal{P}$ (a tedy i v $\mathcal{NP}$) jdou převést na nějaký $\mathcal{NP}$-úplný superproblém.

[9] http://ksp.mff.cuni.cz/viz/kucharky/slozitost

[10] http://ksp.mff.cuni.cz/viz/kucharky/grafy