Korespondenční Seminář z Programování

27. ročník **KSP** **Říjen 2014**

Milí řešitelé a řešitelky!

Léto už je za námi, přichází sychravý podzim a s ním delší večery. Abyste se za deštivých dnů a dlouhých večerů doma nenudili, přinášíme vám druhou sérii KSP. Můžete se těšit opět na sbírku teoretických i praktických úloh okořeněnou pokračováním seriálu o UNIXu.

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. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě.

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.

V závěru úvodu chceme všechny řešitele, stejně jako kohokoli dalšího se zájmem o studium na MFF UK, pozvat na **Den otevřených dveří MFF UK**, který proběhne ve **středu 26. listopadu**. Více informací naleznete na adrese *http://www.mff.cuni.cz/verejnost/dod/*.

Termín série: **Pondělí 8. prosince 2014 v 8:00 SEČ** (CodEx má termín o 24h později)
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: Každému, kdo vyřeší **tři libovolné úlohy na plný počet bodů**, pošleme **čokoládu**.

Druhá série dvacátého sedmého ročníku KSP

Stejně jako v první sérii, i teď zavítáme k nějaké zajímavé programátorské chybě. V minulém díle jsme se potkali s chybou v GPS způsobenou dělením nulou, typické to softwarové přehlédnutí. Dnešní chyba však bude ukryta ještě hlouběji, je to na první pohled těžko tušitelné selhání v návrhu celého systému.

Podobně jako minulý příběh nás i dnešní zavede do války v Perském zálivu, tentokrát ale do města Dhahran *na západním břehu Perského zálivu v Saudské Arábii. Bohužel však tato chyba bude mít mnohem temnější následky...*

⋆⋆⋆

Bylo 25. února ráno a spojenecká základna v Dhahranu *se probouzela do dalšího dne, hlídky přebíraly nové skupiny vojáků a z kantýny se začala linout vůně připálených vajíček. Svobodník George Matthews do sebe rychle naházel snídani, vajíčkům se raději vyhnul, a vydal se na hlídkovou věž vystřídat jiného strážného.*

Cestou ještě podrbal svého psa, kterého měl jako pyrotechnik už mnoho let přiděleného. Dopey byl už za svých třináct let zvyklý na vojenský život na základnách, a tak jen šťastně zavrtěl ocasem, na svém řetězu doběhl k jedné z podpůrných noh blízkého přívěsu a označkoval ji.

„Systém za desítky miliónů a ty si ho tady budeš značkovat?“ zasmál se George a vydal se okolo přívěsu s radarem protiraketového systému Patriot *dál. Systém to byl rozhodně impozantní a i díky němu si připadal v bezpečí. V bateriích v blízkosti radaru se nacházelo skoro třicet kusů protistřel, kterými systém sestřeloval blížící se irácké rakety* Scud. *Vždy si pro sestřel vybral tu nejvhodnější protistřelu a tu odpálil.*

27-2-1 Systém Patriot — 9 bodů

Protiraketový systém *Patriot* má k dispozici mnoho protistřel, které může proti blížící se hrozbě odpálit. Pro jednoduchost můžeme každou protistřelu charakterizovat pomocí jejího dostřelu (kladné reálné číslo). Za normální situace (pokud neurčí lidský operátor jinak) vyšle systém protistřelu, jejíž dostřel je mediánem mezi aktuálně dostupnými protistřelami.

Medián je prvek, který by se v setříděné posloupnosti nacházel přesně uprostřed. Pokud má posloupnost sudý počet prvků (tedy uprostřed leží dva prvky), budeme v tomto případě brát ten větší z nich (protistřelu s vyšším doletem).

Protistřely se do systému i doplňují (tak, jak jsou dopravovány na základnu), a proto by od vás spojenecká armáda potřebovala vybudovat rychlou datovou strukturu podporující dva typy operací:

1. Přidej protistřelu s doletem d_i do systému.
2. Odpal protistřelu s doletem, který je mediánem mezi aktuálními dolety (výsledkem by mělo být odebrání protistřely a vrácení jejího doletu).

⇧ **Lehčí varianta (za 3 body):** Vyřešte úlohu pro případ, kdy systém vysílá střelu s nejvyšším dostřelem.

Už se blížil čas oběda, když se základnou rozezněly poplašné sirény. George se rychle otočil na systém Patriot. Viděl, jak se jedna z baterií protistřel natočila směrem na severovýchod, a očekával odpal. Ale vteřiny ubíhaly a nic se nedělo. Po deseti vteřinách čekání skoro v ten samý moment většině přítomných vojáků došlo, že protistřela už nevyletí, že se něco porouchalo.

George se nedíval na ostatní, ale rychle sjel po žebříku z věže a sprintem se vrhl do blízkého úkrytu. Pak dopadl irácký Scud a obloha potemněla. Byl to den, kdy opěvovaný systém Patriot selhal, a nikdo zatím nevěděl proč.

Hned, jak lehce opadl mrak sutin a prachu, začali se z různých úkrytů vynořovat více či méně zranění vojáci. Ti zázrakem nezranění a ti s lehkými zraněními se hned vrhli do odhrabávání trosek zřícených budov.

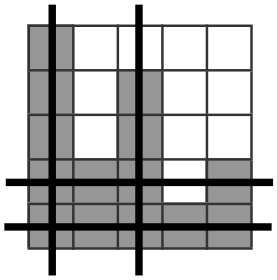
27-2-2 Prohledávání budov — 10 bodů

Po raketovém útoku stojí na základně několik poničených budov. Je nutné je všechny projít, odklidit trosky a hledat přeživší. Již se organizuje několik skupin záchranářů, ale je potřeba rozmyslet plán prohledávání.

Jedna skupina záchranářů může projít buď jednu budovu od přízemí do nejvyššího patra, nebo může naopak projít i-té patro v každé budově. Každé patro v každé budově je nutné prohledat alespoň jednou, vícenásobné prohledání nám nevadí.

Dostanete seznam budov a jejich počty pater. Rozmyslete, jak je všechny prohledat s co nejmenším počtem záchranných týmů.

Například pro výšky $(5, 2, 4, 1, 2)$ stačí čtyři záchranné týmy, jak je vidět na následujícím obrázku:



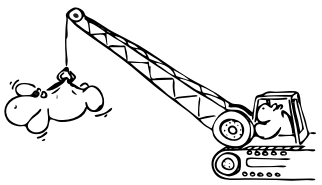
Už několik hodin pomáhal George vytahovat z trosek oběti. Většina z nich to přežije, ale už objevili i pár takových, kteří tolik štěstí neměli. Právě rozebírali pozůstatky po kantýně v centru základny, když pod troskami spatřil známou věc – lesklý obojek.

Rychle odházel stranou několik kovových nosníků a sevřel v náručí psí tělo. Dopey vypadal, že jen spí, ale nebylo tomu tak. Na George náhle dolehly události několika posledních hodin plnou silou, a tak se jen posadil na trosky a několik minut jenom hleděl do dáli.

„Třináct let a čtyřicet dva dní, pane," řekl důstojníkovi, který se u něj objevil, a pak ještě dodal: „Tolik mu bylo." „To je mi líto Matthewsi, ale teď sem potřebujeme rychle dostat nějaké jeřáby, aby nám pomohly s odklízením trosek." George odložil tělo Dopeyho opatrně ke stěně, osušil slzy a vrhl se na hordu map na plánovacím stole.

27-2-3 Průjezd jeřábu — 10 bodů

Potřebujeme dostat autojeřáb z jednoho konce města *Dhahran* na druhý, aby pomohl s odklízením trosek na vojenské základně. Stavební firma sídlící na druhém okraji města je ochotná půjčit jakýkoliv ze svých jeřábů. My bychom chtěli k troskám dostat co největší jeřáb, ale čím větší, tím také vyšší – a ulice *Dhahranu* jsou prošpikované nízko zavěšenými elektrickými dráty.



Na vstupu máme mapu města zadanou jako síť ulic a křižovatek. Ulice jsou obousměrné, ale pro každou ulici máme údaj, jaké nejvyšší vozidlo jí ještě může bezpečně projet. Mapa města tedy představuje ohodnocený graf. Navíc dostaneme ještě dvě označené křižovatky, sídlo firmy a vojenskou základnu. Najděte cestu mezi těmito dvěma místy, kterou může projet co nejvyšší jeřáb.

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.

Konečně se jim povedlo všechny vyprostit, a také znovu zabezpečit základnu. Konečný účet byl 28 mrtvých spojeneckých vojáků a jeden pes k tomu. Zraněných bylo několik desítek. Všem také vrtalo hlavou, proč systém Patriot ani nevystřelil. Na to přijel hledat odpověď i vyšetřovací tým, který dorazil ještě toho dne večer.

Přesuneme se teď v příběhu od svobodníka Matthewse, který sehrál důležitou roli při záchranných operacích, k poručíkovi Blairovi, vedoucímu vyšetřovacího týmu.

Poručík Blair začal ihned shánět všechny informace o selhání. Protiraketový systém fungoval tak, že radar kontinuálně snímal celou oblast. Ve chvíli, kdy zachytil blížící se raketu Scud, přepnul se do přesnějšího módu, omezil snímání jen na oblast, ve které se raketa nacházela, a tím zpřesnil zaměření před odpálením protistřely.

Podle očitých svědků radar něco zaregistroval a připravil jednu z baterií protistřel na odpal. K samotnému odpalu ale již nedošlo. První věcí, kterou vyšetřovací tým potřeboval, bylo stáhnout družicové snímky oblasti z okamžiku vypálení Scudu. Zhruba každých deset sekund snímkovala americká družice oblast Íráku a čas startu rakety a její trajektorie by mohly pomoci s objasněním, co se vlastně stalo.

Problémem, se kterým se ale vyšetřovací tým musel nějak poprat, bylo to, že rychlost místního připojení k družicové informační síti byla příliš pomalá – dostačovalo k předávání běžných zpráv, ale na stáhnutí mnoha kompletních snímků z družic již ne. Naštěstí si Američané již před časem pro podobné věci vypracovali postup.

27-2-4 Stahování map — 12 bodů

Přes pomalé připojení potřebujeme přenést několik snímků o rozměrech $R \times S$ políček. Každý snímek můžeme přenést buď samostatně nezávisle na ostatních, pak nás jeho přenesení stojí $R \cdot S$ času, nebo jako diferenci od jiného, již přeneseného snímku. V takovém případě se přenášejí jen rozdílná políčka, ale je nutné počítat s režií přenosu W navíc (například proto, že nestačí jen přenést hodnotu na políčku, ale musíme ještě udat jeho souřadnice, tedy posíláme tři čísla namísto jednoho).

Konkrétně, pokud bychom chtěli snímek A_i přenést jako diferenci oproti již přenesenému snímku A_j a D_{ij} by nám vyjadřovalo počet rozdílných políček obou snímků, pak by náklady na přenos A_i byly $D_{ij} \cdot W$.

Na vstupu dostanete počet snímků N, jejich rozměry R a S, režii přenosu W a pak všech N snímků. Vaším cílem bude uspořádat snímky v nějakém pořadí a u každého zvolit, jestli se má přenášet celý, či jako diference od nějakého zvoleného snímku, aby celkové náklady na přenos byly nejmenší možné.

Při řešení úlohy předpokládejte $N \leq 500$, $R, S \leq 20$.

Lehčí varianta (za 4 body): Řešte stejnou úlohu, ovšem s omezením $N = 3$.

[1] `http://ksp.mff.cuni.cz/viz/codex`

„Tohle je všechno správně…“ vzdychl jeden z techniků po prohlédnutí stažených družicových snímků. „Scud přilétl skoro přímo doprostřed zorného pole Patriotu, takže ho musel detekovat. Baterie protistřel se také aktivovala, ale pak už od řídícího systému nedostala finální zaměření a pokyn k odpalu.“

„Dobře. Zkuste se podívat na jakákoliv hlášení související s chybami a údržbou systému Patriot za poslední dva měsíce,“ rozkázal poručík jednomu z desátníků. „My zatím zkusíme prozkoumat instrukční sadu, jestli není chyba v ní,“ dodal ke zbytku týmu.

27-2-5 Nejdelší příkaz — 12 bodů

Nervovým centrem každého systému *Patriot* je řídící počítač s několikanásobnou zálohou. Při zkoumání důvodů problému již technici vyloučili fyzické selhání počítačů, a tak padl jejich zrak na software.

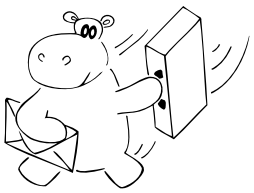
Při programování se používá specifický programovací jazyk, ve kterém se jednotlivé příkazy skládají z posloupnosti klíčových slov. Každý příkaz může začít libovolným klíčovým slovem, ale každé navazující klíčové slovo může vzniknout jen vložením jednoho písmene do předchozího (posloupnost `UA,DUA,DUHA,DUCHA` je korektní, ale posloupnosti `DUA,DUCHA` ani `DUA,DUHA,DUHY` již ne).

Techniky by zajímalo, jaký nejdelší příkaz (co do počtu klíčových slov) lze v jazyce sestavit a jestli náhodou touto délkou nepřekročí délku vestavěného příkazového zásobníku, a nemůže tak způsobit systémové selhání. Na vstupu dostanete seznam klíčových slov a na výstup byste měli odpovědět délkou nejdelšího možného příkazu.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

„Program se zdá v pořádku, budeme si tedy muset ušpinit ruce,“ řekl znaveně poručík a odvedl svůj tým techniků ven k lehce poškozenému radaru. „Zkusíme zjistit, jestli jsou všechny části systému na svých místech a komunikují spolu.“

„Ale pane, to bude hrozně moc práce, rozebrat to a prověřit každý kabel nám bude trvat dny!“ „Máte snad lepší nápad?“ zeptal se Blair. Mladý desátník se chvíli zamyslel, pak odběhl dovnitř, vytáhl ven jeden z počítačů a přinesl kupu propojovacích kabelů.



„Můžeme zkusit do jednotlivých uzlů systému vysílat signály a sledovat, za jak dlouho se dostanou do jiných. To by nám mělo dát odpověď na to, jestli jsou spoje v pořádku.“

27-2-6 Testování odezvy — 14 bodů

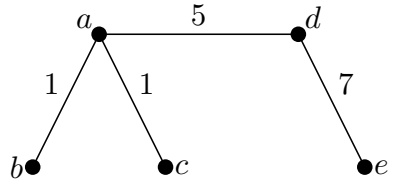
Technici naměřili na propojeném počítačovém systému, sestávajícím se z N uzlů, jak dlouho trvá signálu dostat se z každého uzlu do každého jiného. Systém je tvořený propojenými dráty, a vyslané signály tak mohou volně procházet skrz celou síť, jen jim překonání každého drátu trvá určitý čas.

Z provedeného měření jsme dostali čtvercovou matici A o rozměrech $N \times N$. Vaším úkolem je sestrojit ohodnocený strom o N vrcholech, v němž vzdálenost mezi vrcholem i a vrcholem j odpovídá hodnotě A_{ij}, nebo říct, že žádný takový strom neexistuje.

Například pro matici

	a	b	c	d	e
a	0	1	1	5	12
b	1	0	2	6	13
c	1	2	0	6	13
d	5	6	6	0	7
e	12	13	13	7	0

je výsledkem následující strom:



Naopak si snadno rozmyslíte, že pro matici

	a	b	c
a	0	1	2
b	1	0	10
c	2	10	0

řešení neexistuje.

Lehčí varianta (za 6 bodů): Řešte úlohu s předpokladem, že strom, kterému matice odpovídá, je neohodnocený (tj. každá jeho hrana má jednotkovou délku).

Když vyšetřovací tým vyloučil softwarové selhání i nefunkčnost radaru, začali být skutečně bezradní. Vtom ale do místnosti vešel desátník, kterého Blair poslal zkoumat stará hlášení, a nesl v ruce desky s jednou zprávou. Bez vysvětlení je podal poručíkovi a Blairovy oči se rozšířily, když mu došlo, co právě objevili.

Byla to zpráva z jedné izraelské základny stará asi dva týdny. Naměřili tam, že po osmi hodinách provozu se střed zaměřovací oblasti Patriotu při přesnějším módu odchýlil o zhruba dvacet procent od místa, kde se reálně nacházela sledovaná střela. Taková odchylka ještě systému nevadila, ale zpráva uváděla, že se tím začal zabývat výrobce protiraketového systému.

Rychlé prolistování skrz hlášení z místní základny odhalilo to, že v tomto případě byl systém Patriot v nepřetržitém provozu skoro 100 hodin – a to už vadilo.

Interně si totiž Patriot počítal čas od svého spuštění jako celé číslo, ale při výpočtu dráhy střely a odhadu místa, kam by měl zaměřit přesnější režim sledování, si rychlost cíle a čas přepočítával do 24-bitového čísla s plovoucí desetinnou čárkou. A jelikož bylo po 100 hodinách provozu číslo udávající čas již příliš veliké, převod do floatu a následný výpočet vyústil v odchylku zaměření skoro 600 metrů.

Systém Patriot tak správně zaregistroval blížící se Scud, připravil protistřelu k odpalu a přepnul se do přesnějšího módu. Kvůli špatnému výpočtu však útočící střelu v přesnějším módu již neviděl, zkrátka proto, že se díval na špatné místo. Co systém nevidí, to nemůže sestřelit, a tak ani nebyla odpálena žádná protistřela.

Nejvíce ironické na celém incidentu je to, že softwarová oprava Patriotu, na které firma pracovala již od izraelského hlášení, dorazila do Dhahranu o den později. Dorazit o den dříve, tato katastrofa by se vůbec stát nemusela…

Příběh pro vás převyprávěl

Jirka Setnička

27-2-7 Shellové skripty — 12 bodů

V minulém dílu jste se naučili pár základních příkazů shellu. Víte, jak vypadá adresářová struktura systému, a umíte si trochu hrát se soubory. Po přečtení tohoto dílu seriálu byste měli být schopni použít shell jako plnohodnotný programovací jazyk. Naučíte se vytvořit *skript,* používat proměnné a podobné věci.

Skripty

Možná víte, že programovací jazyky se dají rozdělit na kompilované a interpretované. Programem v interpretovaném jazyce je samotný zdrojový kód, který si interpret přečte a provede. A protože je shell mocný nástroj, umí sloužit jako interpretovaný jazyk.

Skript je tedy obyčejný textový soubor, který obsahuje příkazy pro shell. Otevřete si svůj oblíbený editor a v něm třeba soubor `skript.sh`. Koncovka souboru není rozhodně nutná, název souboru je informací jen pro uživatele. Do něj můžete napsat třeba toto:

```
echo "Jsi v adresáři:"
pwd
```

Věřím, že poznáte, co skript dělá. Pokud ne, můžete ho zkusit spustit:

```
hroch@ksp:~$ bash skript.sh
Jsi v adresáři:
/home/hroch
```

Takto spustíme nový shell. Pokud dáme shellu jako první poziční argument existující soubor, spustí jej. To je pěkné, ale není to „program“. Aby šlo skript spouštět samostatně, musíte ještě pár věcí zařídit.

Minule jsme si říkali, že soubory mají nějaká práva. Podrobnější rozepsání očekávejte v příštích dílech, dnes jen krátce. Každý soubor má různá práva pro svého vlastníka, skupinu a ostatní. Tři základní práva jsou Read, Write a eXecute, neboli čtení, zápis a spuštění. Aby byl shell ochoten soubor spustit, musíte mít právo spuštění a samozřejmě čtení.

Na změnu práv souboru použijte příkaz `chmod`. V prvním argumentu lze říct o jaká práva jde, dalšími jsou pak změněné soubory. Následující příklad učiní náš skript spustitelným:

```
hroch@ksp:~$ ls -l skript.sh
-rw-r--r-- hroch ksp 29 1. říj 12:00 skript.sh
hroch@ksp:~$ chmod +x skript.sh
hroch@ksp:~$ ls -l skript.sh
-rwxr-xr-x hroch ksp 29 1. říj 12:00 skript.sh
```

Ještě musíme říct systému, čím že to má skript spustit. Doplňte tuto konstrukci na první řádek souboru:

```
#!/bin/bash
```

Za normálních okolností značí mříž (`#`) na začátku řádku komentář, spolu s vykřičníkem na prvním řádku ale říká, jaký program se má spustit k interpretaci daného souboru. Takto už systém ví, že má použít program `/bin/bash`, tedy bash.

Při psaní příkazu v shellu jsme zatím první slovo na řádku označovali jako *příkaz.* Není to ale úplně pravda – prvním slovem je název souboru nebo interní příkaz shellu. Shell má několik adresářů, ve kterých programy hledá, pokud nejsou zadány s cestou. Jedním z nich je určitě `/bin`, takže když napíšete jen „bash“, shell si domyslí `/bin/` a spustí `/bin/bash`.

Pokud ale dáte na začátek řádky soubor i s adresou, shell si nic domýšlet nebude a zadaný soubor prostě spustí (pokud je spustitelný).

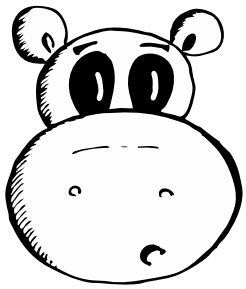
```
hroch@ksp:~$ /home/hroch/skript.sh
```

Možná si vzpomenete na všudypřítomný adresář `.` (tečka). Teď si ukážeme, k čemu se dá využít. Pokud byste se pokusili spustit jen příkaz

```
hroch@ksp:~$ skript.sh
```

tak se asi nic nestane. Většina shellů z bezpečnostních důvodů nehledá spustitelné soubory v aktuálním adresáři. Je tedy nutné shellu zadat plnou cestu k souboru – no a abychom nemuseli psát celou cestu, použijeme adresář tečka. Ten totiž odkazuje na adresář, ve kterém je umístěn. Můžeme toho využít i zde:

```
hroch@ksp:~$ ./skript.sh
```



Procesy

Na chvilku odbočíme od skriptování k samotnému UNIXu. Jistě jste si všimli, že v systému může běžet více programů současně. Dokonce můžete spustit jeden program dvakrát, třeba s jinými parametry. Jak tohle systém dělá?

Každý běžící program je zabalen do struktury, které říkáme *proces.* V ní najdeme například PID (Process ID, identifikátor procesu). To je nějaké číslo, unikátní pro každý běžící proces. Jakmile dojdou volná PID, systém vám již nedovolí další proces spustit.

Dále si proces pamatuje uživatele, pod kterým běží, stav, ve kterém se nachází, PID svého otce nebo třeba terminál, ke kterému je připojen.

Jak se ale takový proces vytváří? Předně, vyrobit „čistý“ proces není možné. To udělá při startu systému jádro, které spustí program `init`. Nadále se procesy můžou jen kopírovat.

Takové operaci se říká *fork* a můžete si ji představit jako rozdvojení. Před zavoláním systémového volání *fork* jste měli jeden proces, po něm máte dva. Úplně stejné, až na PID. Jednomu zůstane a stává se rodičovským procesem, ten druhý má PID nové a stává se synem.

Pro vypsání procesů můžete použít příkaz `ps`. Jeho parametry nejsou dané normou, ale něco jako `ps ax` by mělo vypsat všechny procesy běžící v systému.

Dalším systémovým voláním je *exec*, kterým se ukončí váš program a nahradí se jiným. Například, pokud v terminálu spustíte program, váš shell provede *fork.* Tím vznikne kopie shellu, která vzápětí zavolá *exec* na váš program.

Každý uživatelský proces má tedy svého otce. Může od něj dostat nějaké informace, třeba v paměti před *fork*em. Zpět je to ale složitější. Dokud oba běží, existují prostředky meziprocesové komunikace. Jak ale oznámit otci „Je mi líto, chyba, končím?“

Když proces ukončí svůj běh, není okamžitě vymazán. Místo toho mu jádro vystaví úmrtní list a čeká, než si ho jeho otec vyzvedne. Pokud mezitím skončil také otec, vyzvedne si ho `init`. Pokud otec běží, ale na svého syna zapomněl, bude syn čekat v paměti navěky. Operaci vyzvednutí můžete najít jako systémové volání *wait*.

Součastí ukončeného procesu je návratová hodnota. To je jednobajtové číslo (0–255) a jeho interpretace je čistě na otci. Můžete si předávat klidně výsledek nějaké operace, ale k tomu je jeden bajt docela málo. Místo toho se návratová hodnota používá pro signalizaci úspěchu nebo chyby. V UNIXovém prostředí je zvykem, že nula značí úspěch, nenula nějakou chybu.

Proměnné

Když už máme za sebou spuštění prvního skriptu, můžeme se podívat na jeho obsah. Na každém řádku leží příkaz se svými přepínači a argumenty. Jednotlivé příkazy může oddělovat také středník. Jako příkaz může sloužit i shellový skript.

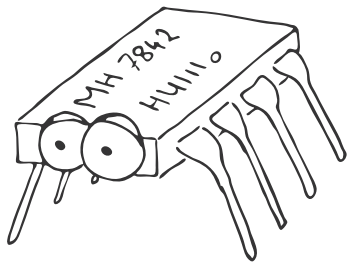
Opravdová legrace začne až s proměnnými. Shell má proměnné z různých důvodů pouze typu řetězec. Nelze s nimi provádět nic zázračného, lze do nich jen zapsat hodnotu nebo naopak proměnnou *expandovat* (nahradit, použít, ...). Tyto dvě operace ale stačí ke všem myslitelným potřebám.

```
prom=ksp
echo $prom
```

Na prvním řádku jsme do proměnné `prom` přiřadili „ksp“. Důležité je, že před rovnítkem není mezera – podle toho shell pozná, že chceme přiřadit do proměnné a ne spustit příkaz `prom`. Všimněte si také, že součástí názvu proměnné není znak `$`.

Na druhém potom spouštíme příkaz `echo`. Znak `$` říká „vlož sem proměnnou“. Shell nejprve vyhledá odpovídající proměnnou a vloží ji na dané místo ještě před spuštěním příkazu.

Tomuto procesu se říká expanze a popíšeme si jej přesněji. Na začátku má shell řádku, která mu byla zadána. Tu rozdělí poprvé na *slova,* posloupnosti nebílých znaků, případně části ohraničené uvozovkami. Poté provede nahrazení složených závorek `{}` a tildy `~` (viz minulý díl). Pak přijdou na řadu proměnné. Bash nabízí spoustu rozšíření expanze proměnných, odvážným doporučuji podívat se do manuálu. Neexistující proměnné jsou prázdné řetězce.



Další krok expanze si popíšeme podrobněji. Často se hodí si někam schovat výstup nějakého příkazu. Už to umíte se soubory. Bez souborů to lze s použitím jedné z těchto dvou konstrukcí pro substituci výstupu:

```
cas=‘date‘
cas=$(date)
```

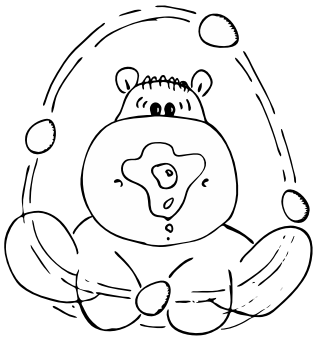
Zpětný apostrof (backtick) je kratší a používanější zápis, má ale několik nevýhod. Ty se projeví zejména pokud plánujete volání zanořit, jako v následujícím, trochu umělém, příkladě:

```
cas=‘date +\‘cat format-\\\‘date +%Y\\\‘\‘‘
cas=$(date +$(cat format-$(date +%Y)))
```

Ten nejprve zjistí, jaký je rok, poté přečte soubor, např. `format-2014`, a jeho obsah použije jako formátovací řetězec pro příkaz date. Zpětné apostrofy se musí escapovat, a escapovací lomítka na druhé úrovni také. Kolik zpětných lomítek musíte použít v třetí úrovni?

V dalším postupu expanze se jednotlivé shelly liší. Společné je jen další rozdělení výsledků předchozích expanzí na slova. Na to pozor, kdekoli použijete proměnnou, její obsah se znovu rozdělí. Pokud chcete použít obsah proměnné jako jeden argument, je potřeba zapsat ji jako `"$prom"`, nestačí jen `$prom`.

Následně proběhne vyhodnocení wildcardů a odstranění escapovacích zpětných lomítek. Až na konci tohoto procesu shell vezme první slovo, spustí jej a předá zbylá slova jako argumenty.



Teď už si můžeme říct pořádně, jaký je rozdíl mezi použitím uvozovek `"`, apostrofů `’` a ničeho. Pokud napíšete text jen tak, použijí se všechny expanze. V uvozovkách se ztrácí význam všech speciálních znaků vyjma `$`, `‘` a `\`, takže se použijí jen expanze proměnných a substituce výstupu. V apostrofech nemá speciální význam vůbec nic, dokonce ani zpětné lomítko. Napsat apostrof do literálu v apostrofech tedy nelze.

```
echo ’Napsat apostrof (’\’’) není snadné.’
```

Důležitý je rozsah platnosti proměnných. Každá proměnná, kterou ve skriptu máte, je viditelná jen a pouze pro váš skript. Nepřežije konec skriptu a nepředá se do vnořených spuštěných příkazů. To druhé lze změnit tzv. *exportem* proměnné.

Pomocí interního příkazu `export prom` zařídíte, že obsah proměnné `prom` bude dostupný i v programech, které spustíte zevnitř vašeho skriptu.

Naopak, proměnné z „vnějšku“ budou dostupné jako tzv. proměnné prostředí. V shellových skriptech se jejich obsah dostane do vnitřních proměnných, např. v C se k nim dá přistoupit pomocí funkce `getenv`.

V souvislosti s rozsahem platosti proměnných se hodí příkaz `source`, který provede vložení. Pokud nějaký skript zavoláte, dostanete z něj pouze výstup a návratovou hodnotu – není způsob, jak by mohl předat obsah proměnných. Pokud jej ale vložíte, spustí se jeho obsah ve stejném procesu. Má tedy přístup ke všem proměnným vašeho programu a obráceně.

K nahlédnutí do proměnných prostředí se dá použít příkaz `env`. Pokud chcete spatřit i neexportované proměnné, pomůže interní příkaz `set`.

Dále má shell spoustu magických proměnných:

- `$0` obsahuje název skriptu tak, jak byl volán
- `$1`...`$9` poziční argumenty
- `$#` počet argumentů
- `$*` a `$@` všechny argumenty (rozdíl níže)
- `$$` identifikátor procesu shellu
- `$IFS` znaky používané pro oddělení slov v konečné fázi expanze

`$*` a `$@` jsou obě zkratkou za vypsání všech pozičních argumentů, jen se každá chová jinak podle uvozovek:

`$*`, `$@` → `$1 $2 $3`
`"$@"` → `"$1" "$2" "$3"`
`"$*"` → `"$1c$2c$3"`, kde `c` je první znak `IFS`

Výchozí hodnota proměnné `IFS` (z angl. Internal Field Separator) je mezera, tabulátor a nový řádek. Díky tomu se chová dělení na slova před expanzí stejně jako po ní. `IFS` ale můžete změnit a tím si upravit chování některých příkazů k obrazu svému – nebo taky způsobit divné chování shellu.

Řídící struktury

Shell samozřejmě poskytuje běžné řídící struktury jako podmínky a cykly. Způsob, jakým to dělá, je ale poněkud... nezvyklý.

Začněme podmínkou. Její syntax je:

```
if cmd; then ...; else ...; fi
```

Část else je nepovinná. Důležitý poznatek – shell nemá nic jako číselnou, natožpak logickou proměnnou. Nemá tedy ani nic jako výraz v podmínce. Místo toho se větev `then` provede tehdy, když byl příkaz `cmd` úspěšný, neboli vrátil nulu. Ještě jednou pro jistotu – za `if` se píše příkaz.

Velmi důležitý je příkaz `test`, který umožňuje porovnávat svoje parametry, a to dokonce i v číselném kontextu. Užitečný je jeho manuál, určitě se do něj podívejte (`man test`). Pár používaných testů:

neprázdnost:	`test -n "$prom"`
řetězce:	`test "$USER" = "hroch"`
čísla $a = b$:	`test "$$" -eq 1`
$a \geq b$:	`test "$a" -ge "$b"`
existence souboru:	`test -f "$file"`
adresáře:	`test -d "$file"`

Příkaz `test` ale v mnoha zdrojových kódech nenajdete. Existuje na něj totiž alias, který vypadá přirozeněji – příkaz `[`. Pokud ale `test` voláte tímto jménem, musí být jeho posledním argumentem `]`.

```
if [ "$i" -ge "$j" ]
```

Z principu fungování podmínky a příkazu `test` je nutné všechny části oddělit mezerou. Nezapomeňte na uvozovky okolo proměnných – bez nich to sice zpravidla bude fungovat, ale jinak, než byste chtěli. Příkaz `test` je většinou jak samostatný program, tak interní příkaz shellu, to aby se kvůli každé podmínce nemusel spouštět další proces.

Syntaxe cyklů je v jistém smyslu podobná.

```
while cmd; do ...; done
```

Opět, cyklus se opakuje, dokud příkaz `cmd` vrací nulovou návratovou hodnotu. Místo negace existuje cyklus `until`. Pokud byste chtěli něco jako `do-while`, ten se v shellu dělá těžko.

Zajímavější je cyklus for. Neprve ukázka:

```
for i in 1 2 3 4 5; do ...; done
```

Cyklus iteruje přes seznam argumentů, které do proměnné postupně dosazuje. Může se to zdát nepraktické, ale vzpomeňte, co všechno umí shell udělat při expanzi.

Pokud vynecháte `in` a seznam slov za ním, bude for iterovat přes poziční argumenty.

Úkol 1 [2b]: Vypište názvy těch souborů v pracovním adresáři, které jsou prázdné.

Další užitečnou konstrukcí je něco, co připomíná `switch` z klasických programovacích jazyků. Protože ale shell pracuje jen s texty, umožňuje vybírat mezi variantami podle tvaru řetězce:

```
case "vstup" in
        vzor) ... ;;
        vzor1|vzor2) ... ;;
        [a-z]) ... ;;
        *) ... ;;
esac
```

Shell postupně zkouší, jestli vstup neodpovídá některému ze vzorů, a vybere první splňující větev. Vzory jsou klasické shellové wildcardy. Je možno jich uvést více, jako na druhém řádku, a vzájemně je oddělit svislou čarou.

Ve vzorech můžete použít většinu expanzí (proměnné, ``‘‘``, wildcardy, ...). Každou větev musíte ukončit jedním z operátorů `;;` a `;&`. První jmenovaný ukončí zpracovávání `case`, tedy již nespustí žádnou větev, kdežto po druhém bude shell pokračovat v porovnávání. Expanze vzorů probíhá až těsně před porovnáním.

Protože pravá zavírací závorka rozbíjí jednoduché zvýrazňování syntaxe, je možno v dnešních shellech před vzor napsat nepovinnou závorku do páru.

Na co nesmíme zapomenout, jsou operátory `&&` a `||`, můžeme číst jako *and* a *or*. Používají se místo oddělovačů příkazů, tedy tam, kde byste použili rouru nebo středník.

Podstatné je, že se vyhodnocují zkráceně. To znamená, máme-li seznam příkazů spojených `&&`, spouštějí se po sobě a první, který vrátí nenulu, sekvenci ukončí. Naopak, posloupnost příkazů spojených `||` ukončí první úspěšný příkaz.

Dají se využít k příjemnému podmínění vykonání příkazů:

```
[ -f soubor ] && ...
mkdir .lock || exit
```

Zejména druhý příklad si zapamatujte, `mkdir` vrátí nulu tehdy a jen tehdy, pokud adresář neexistoval a podařilo se ho vytvořit (znovu připomínáme, že ve světě UNIXu značí nulová návratová hodnota úspěch). Dá se tedy dobře použít jako virtuální zámek.

Pokud byste potřebovali návratovou hodnotu příkazu negovat, můžete tak učinit připsáním `!` před příkaz. Vykřičník a příkaz ale musí být dvě oddělená slova, musí mezi nimi být mezera.

Úkol 2 [3b]: Napište skript, který vypíše všechny své argumenty v opačném pořadí, než v jakém je dostal.

Složený příkaz

Kdekoli, kde je možno spustit jeden příkaz, je možno místo toho použít příkaz složený. Příklad použití:

```
{
        echo "Soubor 1:"
        cat s1
} >s2
```

Zde se výstup celého složeného příkazu zapíše do souboru `s2`. Možná jste viděli i použití s kulatými závorkami místo složených. S těmi se vnitřní příkazy spustí v tzv. *subshellu.* V subshellu se spustí příkaz i pokud do něj nebo z něj vede roura, pokud je spuštěn ve zpětných apostrofech nebo pokud jej spouštíte na pozadí.

To znamená, že shell provede *fork*, a obsah závorek provede v synovském procesu. Přesměrování vstupů a výstupů tedy provádíte na nové instanci shellu. V subshellu nelze nijak ovlivnit prostředí shellu otcovského, všechny proměnné jsou lokální pro subshell. V kontextu subshellu obsahuje proměnná `$$` id procesu otcovského shellu, přestože jde o jiný proces.

Vstup

Do proměnné lze snadno vložit obsah souboru, ale jak do ní přečíst vstup? Na to je k dispozici příkaz `read`.

```
echo prvni druha | {
        read prom1 prom2
        echo "prom1: $prom1"
        echo "prom2: $prom2"
}
```

`read` přečte řádek ze standardního vstupu, rozdělí jej na slova podle `IFS`, první slovo vloží do první proměnné, druhé do druhé a tak dále.

Pokud je méně slov na vstupu než proměnných, zbylé proměnné se vyprázdní. Pokud je tomu naopak, do poslední se uloží celý zbytek řádku bez oddělovačů na konci.

Úkol 3 [1b]: Poslední příklad by bez složených závorek jen tak nefungoval. Proč je musíme použít?

Úkol 4 [3b]: Pro každého uživatele z `/etc/passwd` vyrobte v aktuálním adresáři soubor, jehož název bude odpovídat uživatelskému jménu daného uživatele. Obsahem tohoto souboru budiž uživatelův výchozí shell. O struktuře `/etc/passwd` se dočtete v manuálu, `man 5 passwd`.

Úkol 5 [3b]: Napište skript, který bude zpracovávat textový log příchodů na velké setkání KSP. Řádky obsahují akce (`prichod`/`odchod`), pohlaví (`M`/`F`) a pak jméno (jednoslovné nebo i víceslovné), oddělené mezerou. Úkolem skriptu bude načíst tento soubor a každý řádek vypsat hezky ve tvaru „Prisla Jana Novakova“, „Odesel Petr“ a podobně. Dávejte pozor na správný tvar prvního slova podle pohlaví.

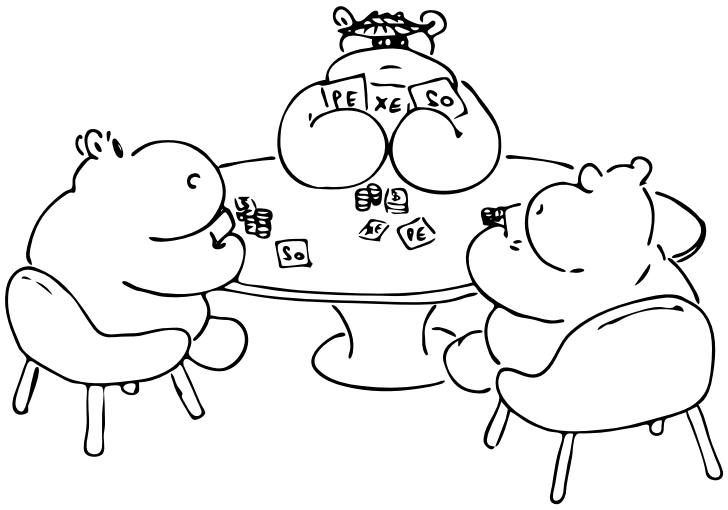
Závěr

Abychom si připomněli a ujasnili, jaké věci jsme se dnes naučili a jaké můžete použít v řešení jednotlivých úkolů, zde je jejich přehled:

- Psaní a spouštění skriptů (adresář tečka)
- Vytváření a život procesů v UNIXu (*fork*, *exec*, PID)
- Proměnné (proměnné pro argumenty, `$IFS`)
- Řídící struktury (`if`, `else`, `while`, `for`, `case`)
- Příkaz `test` a `&&` s `||`
- Složené příkazy a `read`.

V dalším pokračování se můžete těšit na některé pokročilejší UNIXové příkazy, jež vám umožní třeba třídit a filtrovat velká data jen pár stisky kláves. Doufáme, že nám zachováte přízeň i nadále.

Ondra Hlavatý



Recepty z programátorské kuchařky: Minimální kostra

Představme si následující problém: chceme určit silnice, které se budou v zimě udržovat sjízdné, a to tak, abychom celkově udržovali co nejméně kilometrů silnic, a přesto žádné město od ostatních neodřízli.

Města a silnice si můžeme představit jako graf, o kterém nyní budeme předpokládat, že je souvislý. Kdyby nebyl, náš problém nijak vyřešit nelze. Výsledný podgraf/seznam silnic, který řeší náš problém se sněhem, nazývají matematici *minimální kostra grafu*.

Pokud vůbec netušíte, co je to graf, přečtěte si úvodní grafovou kuchařku na našem webu.[2]

Co se v souvislém grafu přesně myslí pod pojmem *kostra*? Nazveme jí libovolný podgraf, který obsahuje všechny vrcholy a zároveň je stromem. Definici stromu a jejich vlastnostem se blíže věnuje dříve zmíněná grafová kuchařka; zde řekněme, že jsou to přesně ty grafy, které jsou souvislé (z každého vrcholu „dojedeme“ do každého jiného) a bez kružnice (takže nemáme v silniční síti žádné přebytečné cesty).

Pokud každou hranu grafu ohodnotíme nějakou *vahou,* což v našem případě bude vždy kladné číslo, dostaneme *ohodnocený graf.* V takových grafech pak obvykle hledáme mezi všemi kostrami *kostru minimální,* což je taková, pro kterou je součet vah jejích hran nejmenší možný.

Graf může mít více minimálních koster – například jestliže jsou všechny váhy hran jedničky, všechny kostry mají stejnou váhu $n-1$ (kde n je počet vrcholů grafu), a tedy jsou všechny minimální.

Pro vyřešení problému hledání minimální kostry se nám bude hodit datová struktura *Disjoint-Find-Union* (DFU). Ta umí pro dané disjunktní množiny (disjunktní znamená, že každé 2 množiny mají prázdný průnik neboli žádné společné prvky) rychle rozhodnout, jestli dva prvky patří do stejné množiny, a provádět operaci sjednocení dvou množin.

Algoritmus

Algoritmus na hledání minimální kostry, který si předvedeme, je typickou ukázkou tzv. hladového algoritmu. Nejprve setřídíme hrany vzestupně podle jejich váhy. Kostru budeme postupně vytvářet přidáváním hran od té s nejmenší vahou tak, že hranu do kostry přidáme právě tehdy, pokud spojuje dvě (prozatím) různé komponenty souvislosti vytvořeného podgrafu. Jinak řečeno, hranu do vytvářené kostry přidáme, pokud v ní zatím neexistuje cesta mezi vrcholy, které zkoumaná hrana spojuje.

Je zřejmé, že tímto postupem získáme kostru, tj. acyklický souvislý podgraf grafu (pokud vstupní graf je souvislý, což mlčky předpokládáme). Než si ukážeme, že nalezená kostra je opravdu minimální, podívejme se na časovou složitost našeho algoritmu: pokud vstupní graf má N vrcholů a M hran, tak úvodní setřídění hran vyžaduje čas $\mathcal{O}(M \log M)$ (použijeme některý z rychlých třídicích algoritmů popsaných v kuchařce o třídění) a poté se pokusíme přidat každou z M hran.

V druhé části kuchařky si ukážeme datovou strukturu, s jejíž pomocí bude M testů toho, zda mezi dvěma vrcholy vede hrana, trvat nejvýše $\mathcal{O}(M \log N)$. Celková časová složitost našeho algoritmu je tedy $\mathcal{O}(M \log N)$ (všimněte si, že $\log M \leq \log N^2 = 2 \log N$). Paměťová složitost je lineární vzhledem k počtu hran, tj. $\mathcal{O}(M)$.

Důkaz správnosti

Zbývá dokázat, že nalezená kostra vstupního grafu je minimální. Bez újmy na obecnosti můžeme předpokládat, že váhy všech hran grafu jsou navzájem různé: pokud tomu tak není již na začátku, přičteme k některým z hran, jejichž váhy jsou duplicitní, velmi malá kladná celá čísla tak, aby pořadí hran nalezené naším třídicím algoritmem zůstalo zachováno. Tím se kostra nalezená hladovým algoritmem nezmění, a pokud bude tato kostra minimální s modifikovanými váhami, bude minimální i pro původní zadání.

Označme si nyní T_{alg} kostru nalezenou hladovým algoritmem a T_{min} nějakou minimální kostru. Co by se stalo, kdyby byly různé? Víme, že všechny kostry mají stejný počet hran, takže musí existovat alespoň jedna hrana e, která je v T_{alg}, ale není v T_{min}. Ze všech takových hran si vyberme tu, která má nejmenší váhu, tedy kterou algoritmus přidal jako první. Když se podíváme na stav algoritmu těsně před přidáním e, vidíme, že sestrojil nějakou částečnou kostru F, která je ještě součástí jak T_{min}, tak T_{alg}.

Přidejme nyní hranu e ke kostře T_{min}. Tím vznikl podgraf vstupního grafu, který zjevně obsahuje nějakou kružnici C – už před přidáním hrany e totiž T_{min} byla souvislá. Protože kostra T_{alg} neobsahuje žádnou kružnici, na kružnici C musí být alespoň jedna hrana e', která není v T_{alg}.

Všimněme si, že hranu e' nemohl algoritmus zpracovat před hranou e: hrana e' neleží v T_{min} na žádném cyklu, takže tím spíš netvoří cyklus v F, a kdyby ji algoritmus zpracoval, musel by ji přidat do F, což, jak víme, neučinil. Z toho plyne, že váha hrany e' je větší než váha hrany e. Když nyní z kostry T_{min} odebereme hranu e' a přidáme místo ní hranu e, musíme opět dostat souvislý podgraf (e a e' přeci ležely na společné kružnici), tudíž kostru vstupního grafu. Jenže tato kostra má celkově menší váhu než minimální kostra T_{min}, což není možné. Tím jsme došli ke sporu, a proto T_{min} a T_{alg} nemohou být různé.

Cvičení

- V důkazu jsme předpokládali, že váhy hran jsou různé (respektive jsme je různými udělali). Není potřeba i v samotném algoritmu přičítat velmi malá čísla k hranám se stejnou vahou?
- Dokažte, že pokud jsou váhy hran různé, minimální kostra je určena jednoznačně.

Další algoritmy

Kromě tohoto algoritmu (říká se mu Kruskalův) můžeme minimální kostru hledat i jinými způsoby. Jejich bližší popis naleznete ve skriptíčkách z algoritmů a datových struktur,[3] my zde jen nastíníme myšlenku:

- *Jarníkův algoritmus* nechává strom rozrůstat se z jednoho vrcholu. Počáteční vrchol zvolí libovolně a pak vždy vybírá nejlehčí hranu vedoucí z už sestrojené části kostry do zbytku grafu.
- *Borůvkův algoritmus* postupně spojuje stromy. Začne triviálními jednovrcholovými stromy bez hran. Pak si každý

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

[3] `http://mj.ucw.cz/vyuka/ads/`

strom vybere nejlehčí hranu, která z něj vede ven, a všechny tyto hrany přidá do kostry. Tím se stromy propojí do větších stromů a to se opakuje, dokud nezbude jediný strom.

Disjoint-Find-Union

Datová struktura *DFU* slouží k udržování rozkladu množiny na několik disjunktních podmnožin (čili takových, že žádné dvě nemají společný prvek). To znamená, že pomocí této struktury můžeme pro každé dva z uložených prvků říci, zda patří či nepatří do stejné podmnožiny rozkladu.

V algoritmu hledání minimální kostry budou prvky v DFU vrcholy zadaného grafu a budou náležet do stejné podmnožiny rozkladu, pokud mezi nimi v již vytvořené části kostry existuje cesta. Jinými slovy podmnožiny v DFU budou odpovídat komponentám souvislosti vytvářené kostry.

S reprezentovaným rozkladem umožňuje datová struktura DFU provádět následující dvě operace:

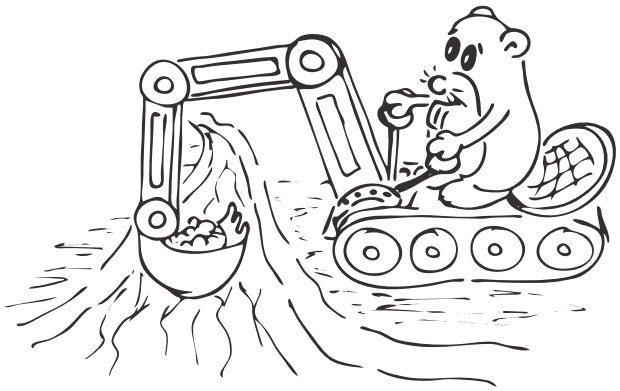
- **find:** Test, zda dva prvky leží ve stejné podmnožině rozkladu. Tato operace bude v případě našeho algoritmu odpovídat testu, zda dva vrcholy leží ve stejné komponentě souvislosti.
- **union:** Sloučení dvou podmnožin do jedné. Tuto operaci v našem algoritmu na hledání kostry provedeme vždy, když do vytvářené kostry přidáme hranu (tehdy spojíme dvě různé komponenty souvislosti dohromady).

Povězme si nejprve, jak budeme jednotlivé podmnožiny reprezentovat. Prvky obsažené v jedné podmnožině budou tvořit zakořeněný strom. V tomto stromě však povedou ukazatele (trochu nezvykle) od listů ke kořeni. Operaci *find* lze pak jednoduše implementovat tak, že pro oba zadané prvky nejprve nalezneme kořeny jejich stromů. Jsou-li tyto kořeny stejné, jsou prvky ve stejném stromě, a tedy i ve stejné podmnožině rozkladu. Naopak, jsou-li různé, jsou zadané prvky v různých stromech, a tedy jsou i v různých podmnožinách reprezentovaného rozkladu. Operaci *union* provedeme tak, že mezi kořeny stromů reprezentujících slučované podmnožiny přidáme ukazatel a tím tyto dva stromy spojíme dohromady.

Implementace dvou výše popsaných operací, jak jsme ji právě popsali, následuje. Pro jednoduchost množina, jejíž rozklad reprezentujeme, bude množina čísel od 1 do N. Rodiče jednotlivých vrcholů stromu si pak pamatujeme v poli *parent*, kde 0 znamená, že prvek rodiče nemá, tj. že je kořenem svého stromu. Funkce *root*(v) vrátí kořen stromu, který obsahuje prvek v.

```
class DFU:
    def __init__(self, n):
        self.parent = [0] * n

    def root(self, v):
        if self.parent[v] == 0:
            return v
        else:
            return self.root(self.parent[v])

    def find(self, v, w):
        return (self.root(v) == self.root(w))

    def union(self, v, w):
        v = self.root(v)
        w = self.root(w)
        if v != w:
            self.parent[v] = w
```

S právě předvedenou implementací operací *find* a *union* by se ale mohlo stát, že stromy odpovídající podmnožinám budou vypadat jako „hadi“, a pokud budou obsahovat N prvků, na nalezení kořene bude potřeba čas $\mathcal{O}(N)$.



Ke zrychlení práce DFU se používají dvě jednoduchá vylepšení:

- **union by rank:** Každý prvek má přiřazen *rank*. Na začátku jsou ranky všech prvků rovny nule. Při provádění operace *union* připojíme strom s kořenem menšího ranku ke kořeni stromu s větším rankem. Ranky kořenů stromů se v tomto případě nemění. Pokud kořeny obou stromů mají stejný rank, připojíme je libovolně, ale rank kořenu výsledného stromu zvětšíme o jedna.
- **path compression:** Ve funkci *root*(v) přepojíme všechny prvky na cestě od prvku v ke kořeni rovnou na kořen, tj. změníme jejich rodiče na kořen daného stromu.

Než si obě metody blíže rozebereme, podívejme se, jak se změní implementace funkcí *root* a *union*:

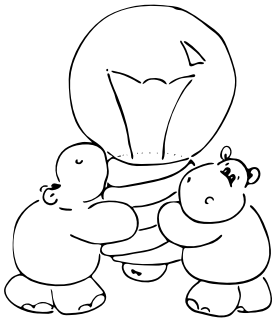
```
class DFU:
    def __init__(self, n):
        self.parent = [0] * n
        self.rank = [0] * n

    # Změna: path compression
    def root(self, v):
        if self.parent[v] == 0:
            return v
        else:
            self.parent[v] = \
              self.root(self.parent[v])
            return self.parent[v]

    # Stejná implementace
    def find(self, v, w):
        return (self.root(v) == self.root(w))

    # Změna: union by rank
    def union(self, v, w):
        v = self.root(v)
        w = self.root(w)
        if v == w:
            return

        if self.rank[v] == self.rank[w]:
            self.parent[v] = w
            self.rank[w] = self.rank[w]+1

        elif self.rank[v] < self.rank[w]:
            self.parent[v] = w

        else:
            self.parent[w] = v
```

Zaměřme se nyní blíže na metodu *union by rank.* Nejprve učiníme následující pozorování: pokud je prvek v s rankem r kořenem stromu v datové struktuře DFU, pak tento strom obsahuje alespoň 2^r prvků.

Naše pozorování dokážeme indukcí podle r. Pro $r = 0$ tvrzení zřejmě platí. Nechť tedy $r > 0$. V okamžiku, kdy se rank prvku v mění z $r-1$ na r, slučujeme dva stromy, jejichž kořeny mají rank $r-1$. Každý z těchto dvou stromů má dle indukčního předpokladu alespoň 2^{r-1} prvků, a tedy výsledný strom má alespoň 2^r prvků, jak jsme požadovali.

Z našeho pozorování ihned plyne, že rank každého prvku je nejvýše $\log_2 N$ a prvků s rankem r je nejvýše $N/2^r$ (všimněme si, že rank prvku v DFU se nemění po okamžiku, kdy daný prvek přestane být kořen nějakého stromu).

Když tedy provádíme jen *union by rank,* je hloubka každého stromu v DFU rovna ranku jeho kořene, protože rank kořene se mění právě tehdy, když zvětšujeme hloubku stromu o jedna. A protože rank každého prvku je nanejvýš $\log_2 N$, hloubka každého stromu v DFU je také nanejvýš $\log_2 N$. Potom ale procedura *root* spotřebuje čas nejvýše $\mathcal{O}(\log N)$, a tedy operace *find* a *union* stihneme v čase $\mathcal{O}(\log N)$.



Amortizovaná časová složitost

Abychom mohli pokračovat dále, musíme si vysvětlit, co je *amortizovaná* časová složitost. Řekneme, že nějaká operace pracuje v amortizovaném čase $\mathcal{O}(t)$, pakliže provedení libovolných k takových operací trvá nejvýše $\mathcal{O}(kt)$. Přitom provedení kterékoliv konkrétní z nich může vyžadovat čas větší. Tento větší čas je pak v součtu kompenzován kratším časem, který spotřebovaly některé předchozí operace.

Nejdříve si předveďme tento pojem na jednoduchém příkladě. Řekněme, že máme číslo zapsané ve dvojkové soustavě. Přičíst k tomuto číslu jedničku jistě netrvá konstantní čas, neboť záleží na tom, kolik jedniček se vyskytuje na konci zadaného čísla. Pokud se nám ale povede ukázat, že N přičtení jedničky k číslu, které je na počátku nula, zabere čas $\mathcal{O}(N)$, pak můžeme říci, že každé takové přičtení trvalo amortizovaně $\mathcal{O}(1)$.

Jak tedy ukážeme, že N přičtení jedničky k číslu zabere čas $\mathcal{O}(N)$? Použijeme k tomu „penízkovou metodu“. Každá operace nás bude stát jeden penízek, a pokud jich na N operací použijeme jen $\mathcal{O}(N)$, bude tvrzení dokázáno.

Každé jedničce, kterou chceme přičíst, dáme dva penízky. V průběhu celého přičítání bude platit, že každá jednička ve dvojkovém zápisu čísla má jeden penízek (když začneme jedničky přičítat k nule, tuto podmínku splníme). Přičítání bude probíhat tak, že přičítaná jednička se „podívá“ na nejnižší bit (tj. ve dvojkovém zápise na poslední cifru) zadaného čísla (to ji stojí jeden penízek). Pokud je to nula, změní ji na jedničku a dá jí svůj zbylý penízek. Pokud to je jednička, vezme si přičítaná jednička její penízek (čili už má zase dva), změní zkoumanou jedničku na nulu a pokračuje u dalšího bitu, atd.

Takto splníme podmínku, že každá jednička v dvojkovém zápisu čísla má jeden penízek. Tedy N přičítání nás stojí $2N$ penízků. Protože počet penízků utracených během jedné operace je úměrný spotřebovanému času, vidíme, že všech N přičtení proběhne v čase $\mathcal{O}(N)$. Není těžké si uvědomit, že přičtení některých jedniček může trvat až $\mathcal{O}(\log N)$, ale amortizovaná časová složitost přičtení jedné jedničky je konstantní.

Dokončení analýzy DFU

Pokud bychom prováděli pouze *path compression* a nikoliv *union by rank*, dalo by se dokázat, že každá z operací *find* a *union* vyžaduje amortizovaně čas $\mathcal{O}(\log N)$, kde N je počet prvků. Toto tvrzení nebudeme dokazovat, protože tím bychom si nijak oproti samotnému *union by rank* nepomohli. Proč tedy vlastně hovoříme o obou vylepšeních? Inu proto, že při použití obou metod současně dosáhneme mnohem lepšího amortizovaného času $\mathcal{O}(\alpha(N))$ na jednu operaci *find* nebo *union*, kde $\alpha(N)$ je inverzní Ackermannova funkce. Její definici můžete nalézt na konci kuchařky, zde jen poznamenejme, že hodnota inverzní Ackermannovy funkce $\alpha(N)$ je pro všechny praktické hodnoty N nejvýše čtyři. Čili dosáhneme v podstatě amortizovaně konstantní časovou složitost na jednu (libovolnou) operaci DFU.

Dokázat výše zmíněný odhad časové složitosti funkcí $\alpha(N)$ je docela těžké, my si zde předvedeme poněkud horší, ale technicky výrazně jednodušší časový odhad $\mathcal{O}((N+L)\log^* N)$, kde L je počet provedených operací *find* nebo *union* a $\log^* N$ je tzv. iterovaný logaritmus, jehož definice následuje. Nejprve si definujeme funkci $2 \uparrow k$ rekurzivním předpisem:

$$2 \uparrow 0 = 1, \quad 2 \uparrow k = 2^{2 \uparrow (k-1)}.$$

Máme tedy $2 \uparrow 1 = 2$, $2 \uparrow 2 = 2^2 = 4$, $2 \uparrow 3 = 2^4 = 16$, $2 \uparrow 4 = 2^{16} = 65536$, $2 \uparrow 5 = 2^{65536}$, atd. A konečně, iterovaný logaritmus $\log^* N$ čísla N je nejmenší přirozené číslo k takové, že $N \leq 2 \uparrow k$. Jiná (ale ekvivalentní) definice iterovaného logaritmu je ta, že $\log^* N$ je nejmenší počet, kolikrát musíme číslo N opakovaně zlogaritmovat, než dostaneme hodnotu menší nebo rovnu jedné.

Zbývá provést slíbenou analýzu struktury DFU při současném použití obou metod *union by rank* a *path compression*. Prvky si rozdělíme do skupin podle jejich ranku: k-tá skupina prvků bude tvořena těmi prvky, jejichž rank je mezi $(2 \uparrow (k-1)) + 1$ a $2 \uparrow k$. Např. třetí skupina obsahuje ty prvky, jejichž rank je mezi 5 a 16. Prvky jsou tedy rozděleny do $1 + \log^* \log N = \mathcal{O}(\log^* N)$ skupin. Odhadněme shora počet prvků v k-té skupině:

$$\frac{N}{2^{(2\uparrow(k-1))+1}} + \cdots + \frac{N}{2^{2\uparrow k}} = \frac{N}{2^{2\uparrow(k-1)}} \cdot \left(\sum_{i=1}^{2\uparrow k - 2\uparrow(k-1)} \frac{1}{2^i} \right) \leq$$

$$\leq \frac{N}{2^{2\uparrow(k-1)}} \cdot 1 = \frac{N}{2 \uparrow k}.$$

Teď můžeme provést časovou analýzu funkce *root*(v). Čas, který spotřebuje funkce *root*(v), je přímo úměrný délce cesty od prvku v ke kořeni stromu. Tato cesta je pak následně rozpojena a všechny prvky na ní jsou přepojeny přímo na kořen stromu. Rozdělíme rozpojené hrany této cesty na ty, které „naúčtujeme“ tomuto volání funkce *root*(v), a ty, které zahrneme do faktoru $\mathcal{O}(N \log^* N)$ v dokazovaném časovém odhadu. Do volání funkce *root*(v) započítáme ty hrany

cesty, které spojují dva prvky, které jsou v různých skupinách. Takových hran je zřejmě nejvýše $\mathcal{O}(\log^* N)$ (všimněte si, že ranky prvků na cestě z listu do kořene tvoří rostoucí posloupnost).

Uvažme prvek v v k-té skupině, který již není kořenem stromu. Při každém přepojení rank rodiče prvku v vzroste. Tedy po $2 \uparrow k$ přepojeních je rodič prvku v v $(k+1)$-ní nebo vyšší skupině. Pokud v je prvek v k-té skupině, pak hrana z něj na cestě do kořene nebude účtována volání funkce *root*(v) nejvýše $(2 \uparrow k)$-krát. Protože k-tá skupina obsahuje nejvýše $N/(2 \uparrow k)$ prvků, je počet takových hran pro všechny prvky této skupiny nejvýše N. A protože počet skupin je nejvýše $\mathcal{O}(\log^* N)$, je celkový počet hran, které nejsou započítány voláním funkce *root*(v), nejvýše $\mathcal{O}(N \log^* N)$. Protože funkce *root*(v) je volána $2L$-krát, plyne časový odhad $\mathcal{O}((N+L) \log^* N)$ z právě dokázaných tvrzení.

Inverzní Ackermannova funkce $\alpha(N)$

Ackermannovu funkci lze definovat následující konstrukcí:

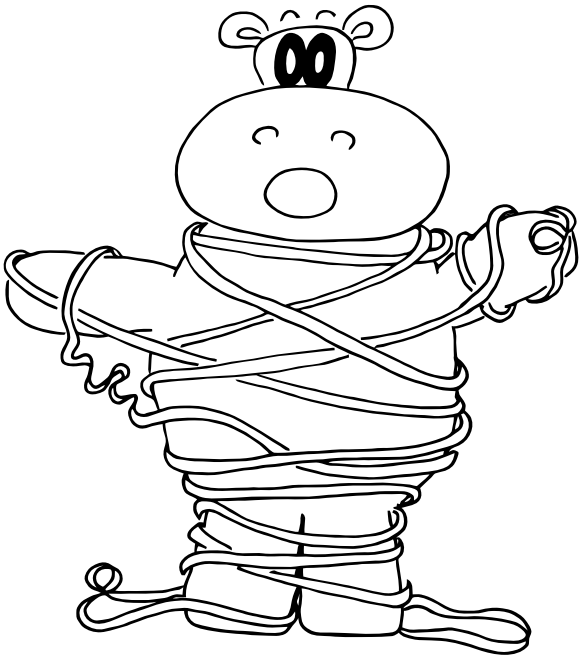
$$A_0(i) = i + 1, \quad A_{k+1}(i) = A_k^i(i) \text{ pro } k \geq 0,$$

kde výraz A_k^i zastupuje složení i funkcí A_k, např. $A_1(3) = A_0(A_0(A_0(3)))$. Platí tedy následující rovnosti:

$$A_0(i) = i + 1, \quad A_1(i) = 2i, \quad A_2(i) = 2^i \cdot i.$$

Ackermannova funkce s jedním parametrem $A(k)$ je pak rovna hodnotě $A_k(2)$, takže $A(2) = A_2(2) = 8$, $A(3) = A_3(2) = 2^{11}$, $A(4) = A_4(2) \approx 2 \uparrow 2048$ atd... Hodnota inverzní Ackermannovy funkce $\alpha(N)$ je tedy nejmenší přirozené číslo k takové, že $N \leq A(k) = A_k(2)$. Jak je vidět, ve všech reálných aplikacích platí, že $\alpha(N) \leq 4$.

Dan Kráľ, Martin Mareš a Milan Straka



Podmínkou na zisk čokolády bylo získání plného počtu bodů z některé ze dvou nejvíce bodovaných úloh první série (což byly úlohy *27-1-6 Přistávací světla* a *27-1-7 Učíme se s UNIXem*). To se povedlo pěti řešitelům, gratulujeme.

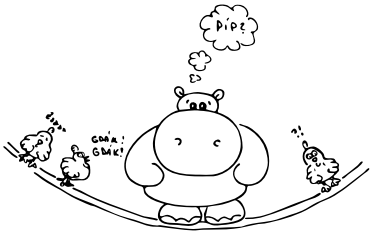
27-1-1 Zasedací pořádek

Piloti na letadlové lodi s vámi mohou být spokojeni – jak pokročilí řešitelé, tak i začátečníci přišli na správné rozsazení osob u stolu. Body jsme však museli strhávat za neúplné (nebo úplně chybějící) popisy druhé podúlohy, případně za drobnosti, jako například chybějící složitost.

Jak budou piloti u stolu rozsazeni, záleží na tom, zda je jejich počet sudý, nebo lichý. Při lichém počtu budou všichni piloti sedět vedle sebe bez mezer, přičemž na centrální židli bude sedět prostřední z nich; jinak bude centrální židle prázdná a piloti budou sedět na obě strany od ní, tvoří tedy dvě souvislé řady délky $N/2$.

Na řešení nebylo těžké přijít simulací rozsazovacího popisu na papíře pro malá N, jen někteří z vás se ale pokusili dokázat, že výše zmíněná pravidla platí pro jakýkoliv počet pilotů. U takto jednoduché úložky jsme za to body nestrhávali, ale pamatujte si, že komplikovanější problémy důkaz vyžadují – všechno „pouhým okem“ neuvidíte!

Správnost zde dokážeme matematickou indukcí. Pro malá N experimentálně ověříme, že náš popis funguje. Teď musíme ukázat, že se „sudá“ řada při příchodu pilota změní na „lichou“ řadu a naopak: první případ je jednoduchý, příchozí usedne na centrální židli.



Komplikovanější je přechod z liché řady na sudou: v momentě, kdy přichází nový pilot, je centrální židle obsazená – oba se přesunou na židle vedle té centrální, a pokud jsou také obsazené, proces se nutně opakuje, dokud nedojdeme k volným židlím na kraji řady, kam si piloti sednou, a řadu tak rozšíří. Aniž bychom zatím přemýšleli nad piloty, kteří se musí přesunout směrem do středu (představme si, že zatím stojí), vypadá řada takto: `1000...0001` (`1` značí obsazenou, `0` volnou židli).

S piloty, kteří se musí ze svých původních míst posunout doprostřed, se řada změní na `1011...2...1101`. Ouha, vše je v pořádku, kromě prostřední židle, o kterou mají zájem dva piloti – opět tedy musí nastat přesouvání, které ale dopadne podobně, jako to předchozí (jen se díky mezerám na koncích zkrátí délka řady, na které dochází k přesunům, o dva piloty, kteří sedí nyní na konci a jsou odděleni od ostatních prázdnou židlí).

Každou takovou iterací se krajní prázdná místa posunou směrem doprostřed. Přesouvání bude trvat tak dlouho, dokud se prázdná místa nesetkají na centrální židli, a řada tím pádem bude vypadat tak, jak jsme popisovali.

Nyní k řešení prvního úkolu: Kdybychom přesouvání jen simulovali podle zadaných pravidel, trvalo by nám to dlouho. Ale to není potřeba, díky výše dokázanému pozorování zvládneme vygenerovat řadu pilotů v lineárním čase a konstantní paměti, stačí sestavit jedničky a nuly podle pravidel.

Program, který ověří, zda je zadaný zasedací pořádek správně, si postupně načítá informace o obsazenosti židlí a kontroluje, zda se řada skládá buď z jedné liché řady jedniček (obsazených míst), nebo dvou stejně dlouhých a sudých řad jedniček, oddělených právě jednou nulou. To zvládneme v čase $\mathcal{O}(N)$, kde N je délka vstupu. Paměťová složitost je konstantní, protože nás nic nenutí řadu do paměti načítat.

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

Kuba Maroušek & Jenda Hadrava

27-1-2 Zbrojní sklad

Většina z řešení, která nám přišla, využívala principu hladového algoritmu a vždy vybírala palety s co největší nebezpečností, což byl správný postup. Takový výběr palety nám totiž umožní v příštím kroku dosáhnout nejvýš. Ukážeme si tedy, jak se to dá udělat efektivně.

Abychom mohli rychle hledat palety, na které dosáhneme, rozdělíme si je nejprve na malé a velké, a potom je vzestupně setřídíme podle výšky do dvou polí.

Poté si založíme dvě maximové haldy, odděleně pro malé a velké palety. V nich si budeme udržovat, jaké palety můžeme v každém kroku odebrat a která z nich má tu největší nebezpečnost. V každém kroku tedy vybereme příslušnou haldu pro velké nebo malé palety a odstraníme z ní paletu s nejvyšší nebezpečností. Nyní musíme do obou hald přidat všechny prvky, na které díky odebrání dosáhneme teď.

K tomu využijeme setřízená pole pro malé a velké palety. U každého si budeme pamatovat index prvku, který jsme přidali naposledy. Pokaždé, když budeme chtít přidat nově dosažitelné palety, jen zkontrolujeme všechny palety od tohoto indexu dále, dokud nenarazíme na paletu, která vyžaduje větší výšku, než je aktuálně dovolená. Všechny nalezené palety s výškou menší nebo rovnou dovolené budeme průběžně přidávat do příslušných hald.

Toto budeme opakovat, dokud jedna z hald nebude prázdná (tedy jsme právě narazili na omezení bezpečnostních předpisů), nebo nám v našich polích nedojdou palety (pak jsme všechny přidali do haldy, tedy na všechny dosáhneme, neboli můžeme všechny palety vyndat bez porušení předpisů).

Velká většina vás narazila na problém volby počáteční palety a mnoho z vás si závorku v zadání (velká, malá, velká) vyložilo tak, že budeme začínat velkou paletou. Po dlouhé diskusi jsme za toto nestrhávali body, ačkoli původně byla úloha myšlená tak, že nebude blíže určeno, kterou paletou začínáme. Řešením, u kterých nebylo očividné, zda si to autoři rozmysleli nebo ne, jsme to připsali do poznámky, která je s tímto vysvětlením snad již trochu jasnější.

Časová složitost algoritmu vychází jednak ze třídění, které nezvládneme obecně rychleji než v čase $\mathcal{O}(N \log N)$, a také z hald (musíme v nich probublat až N prvků, každý v čase $\mathcal{O}(\log N)$. Celková časová složitost je tedy $\mathcal{O}(N \log N)$. Paměti zabereme lineárně s velikostí vstupu (každou paletu uložíme do haldy a pole jednou).

Program (C++):
`http://ksp.mff.cuni.cz/viz/27-1-2.cpp`

Štěpán Hojdar & Jan „Oggy“ Škoda

27-1-3 Letecké koridory

Nejprve bychom se chtěli omluvit za určité nepřesnosti v zadání samotné úlohy. Konkrétně nebylo jasné, zda se úloha má řešit pro ohodnocený či neohodnocený graf. Plný počet bodů jsme proto udělovali za optimální řešení libovolné z těchto variant.

Stejně dlouhé koridory

Většina z vás, kteří jste brali v úvahu neohodnocený graf, přišla na to, že prohledávání do šířky[4] je správný směr k optimálnímu řešení úlohy. Ve vašich řešeních se pak vyskytovaly nanejvýš drobné implementační chyby.

Graf tedy budeme procházet po hladinách od počátečního vrcholu. Budeme si udržovat frontu vrcholů, jež máme postupně zpracovat, do které na začátku umístíme pouze počáteční vrchol. Pro každý vrchol si navíc zapamatujeme délku nejkratší cesty do tohoto vrcholu (pokud jsme nějakou našli) a počet doposud nalezených nejkratších cest. Na začátku víme, že do počátečního vrcholu vede pouze jedna nejkratší cesta délky nula. O ostatních vrcholech nevíme nic.

Při zpracovávání každého vrcholu u vyjmutého z fronty postupně projdeme všechny jeho hrany. Cesta, která vede do u a poté po zkoumané hraně do vrcholu v, má délku rovnou délce nejkratší cesty do u zvětšené o jedničku. Tuto délku potenciálně nejkratší cesty porovnáme s délkou známé nejkratší cesty do v. Mohou nám nastat tři situace. Buď cestu do vrcholu v neznáme – v tom případě je nejkratší cesta do v právě ta námi uvažovaná (pokud by existovala kratší, přišli bychom na ni při zpracovávání předchůdce koncového vrcholu, který se na této nejkratší cestě nachází). Počet takovýchto cest je roven počtu nejkratších cest do u. Protože jsme tento vrchol ještě neviděli, přidáme jej do fronty vrcholů ke zpracování.

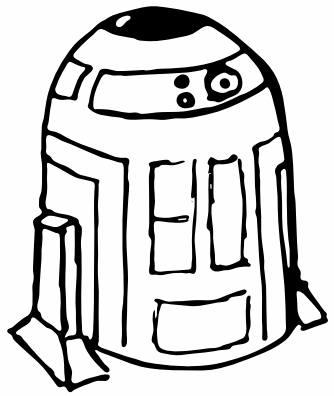
Pokud je délka naší navržené cesty stejně dlouhá jako známá délka nejkratší cesty do v, přišli jsme na další nejkratší cesty, které končí v tomto vrcholu. Zvětšíme tedy počet možných nejkratších cest, jež do vrcholu v vedou, o počet nejkratších cest vedoucích do vrcholu u. Do fronty vrchol v přidávat nebudeme, neboť jsme jej tam přidali, když jsme poprvé určili délku nejkratší cesty. A pokud je délka navržené cesty delší než známá délka nejkratší cesty do koncového vrcholu, nebudeme dělat nic, protože námi navržená cesta určitě není nejkratší.

V okamžiku, kdy frontu vyprázdníme, vypíšeme na výstup počet nejkratších cest vedoucích do cílového vrcholu a algoritmus ukončíme. Můžeme jej ukončit i v případě, když zpracovávané cesty jsou delší než nejkratší cesta do cílového vrcholu, v takovém případě totiž už nenalezneme žádnou další nejkratší cestu, nicméně toto nijak nezlepší asymptotickou časovou složitost. Výsledek je pak počet nalezených nejkratších cest do cílového vrcholu.

Správnost algoritmu můžeme ukázat indukcí. Na začátku vede pouze jedna cesta do počátečního vrcholu. V okamžiku, kdy vyjmeme nějaký vrchol z fronty, zpracovali jsme již všechny jeho předchůdce, po kterých vede nejkratší cesta (všechny vrcholy, které jsme dosud nezpracovali, leží na stejné hladině jako zkoumaný vrchol, nebo na hladině ještě vyšší, proto přes ně nemůže vést nejkratší cesta do zkoumaného vrcholu). Z čehož vyplývá, že pro každý vrchol vyjmutý z fronty nalezneme všechny nejkratší cesty. Z popisu algoritmu přímo plyne, že jsme žádnou cestu nezapočítali dvakrát, čímž je dokončen důkaz správnosti.

Každý vrchol přidáváme do fronty nanejvýš jednou a stejně tak každou hranu zkoumáme maximálně jednou. Časová složitost je tedy $\mathcal{O}(N + M)$, kde N je počet vrcholů a M je počet hran. U každého vrcholu si musíme pamatovat délku a počet nejkratších cest. Dále si musíme pamatovat všechny hrany. Dostáváme tedy i paměťovou složitost $\mathcal{O}(N + M)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/27-1-3a.cpp`



Ohodnocené hrany

V případě, že pracujeme s ohodnocenými grafy, řešení bude podobné tomu předchozímu. S obyčejným průchodem do šířky si však nevystačíme. Místo něj budeme vycházet z Dijkstrova algoritmu,[5] který se prohledávání do šířky podobá, jen místo toho, aby bral nejbližší vrcholy podle počtu hran, je bere podle vzdálenosti. Hrany tedy nemůžeme uchovávat v obyčejné frontě, ale musíme je mít ve frontě prioritní, tedy haldě. V tomto řešení budeme pro jednoduchost uvažovat haldu binární. Další rozdíl mezi řešením pro neohodnocené grafy a tímto bude, že při přidání hrany do cesty nezvětšíme délku cesty o jedna, ale o celou délku dané hrany.

Při zpracování vrcholu u, který označíme jako *definitivní*, budeme v daných případech postupovat stejně jako při zpracovávání vrcholu v předchozím algoritmu. Tedy když neznáme cestu do sousedního vrcholu v, prohlásíme nově nalezenou cestu za nejkratší. Počet nejkratších cest vedoucích do vrcholu v tak bude stejný jako počet nejkratších cest do vrcholu u, navíc vrchol v přidáme do haldy. Když nalezneme stejně dlouhou cestu, upravíme počet nejkratších cest, a když nalezneme delší cestu, nic neděláme.

Nyní však už nemáme garantováno, že když jsme vrchol navštívili, nenalezneme ještě lepší cestu než tu, kterou jsme nalezli poprvé. Když takováto situace nastane, postupujeme stejně jako v případě, kdy bychom žádnou cestu do v neznali – zahodíme informace o počtu a délce nejkratších cest do vrcholu v, jako délku nejkratší cesty určíme nejkratší cestu do u zvětšenou o délku hrany a počet nejkratších cest do v bude stejný jako počet nejkratších cest do u. Vrchol do haldy ale přidávat nebudeme, neboť se v ní již nachází.

Nesmíme ale zapomenout změnu nejkratší cesty do haldy k vrcholu zaznamenat. Abychom daný vrchol v haldě rychle našli, budeme si navíc ještě bokem uchovávat pole, které nám říká, kde v haldě se daný vrchol nachází. Toto pole bu-

[4] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`
[5] `http://ksp.mff.cuni.cz/viz/kucharky/haldy-a-cesty`

deme aktualizovat při každé změně pozice nějakého vrcholu v haldě.

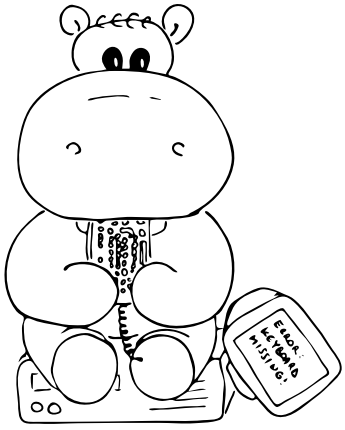
Změnou délky nejkratší cesty se nám však mění i pozice, kde by se vrchol v haldě měl nacházet. V případě, kdy tuto délku měníme, necháme vrchol probublat v haldě výš (nikdy nepotřebujeme posunovat vrchol níže, protože když měníme délku nejkratší cesty, vždy ji nahrazujeme cestou ještě kratší).

V haldě si v každém okamžiku uchováváme nanejvýš všechny vrcholy. Hloubka haldy je tedy maximálně $\log N$. Jedno vyjmutí vrcholu z haldy nám zabere s následným přerovnáním $\mathcal{O}(\log N)$ času. Každý vrchol odebereme nanejvýš jednou, celkově tedy $\mathcal{O}(N \log N)$. Každý vrchol do haldy nanejvýš jednou přidáme, jedno přidání zabere logaritmický čas, dohromady tedy opět $\mathcal{O}(N \log N)$. A konečně při změně nejkratší cesty do nějakého vrcholu musíme upravit pozici daného vrcholu v haldě. Pro každý vrchol to zabere $\mathcal{O}(\log N)$ času a těchto operací provádíme nanejvýš M. Tyto operace tedy zaberou $\mathcal{O}(M \log N)$ času. Celý algoritmus bude mít časovou složitost $\mathcal{O}(M \log N + N \log N)$ neboli $\mathcal{O}((M + N) \log N)$, což je stejná složitost, jakou má běžný Dijkstrův algoritmus.

Pamatovat si potřebujeme u každého vrcholu délku nejkratší dosud nalezené cesty, počet takových cest a umístění v haldě. Dále si potřebujeme pamatovat všechny hrany. Paměťová složitost bude tedy $\mathcal{O}(N + M)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/27-1-3b.cpp`

Lukáš Folwarczný & Dominik „Drecker“ Smrž



27-1-4 Head-up display

Displej má N pixelů, každé jeho nastavení má hodnotu, do které přispívá každý pixel rozdílem kontrastu oproti původnímu nastavení. Ze všech přípustných variant změny kontrastu vybíráme tu nejmenší. Na to se můžeme dívat jako na nejkratší cestu v grafu.

Jak takový graf vypadá? Má $N \cdot K$ vrcholů, pro každý pixel a každou možnou hodnotu kontrastu jeden. (Ono vlastně bude stačit N-krát nejvyšší hodnota kontrastu na vstupu, je snadno vidět, že v nejkratší cestě nikdy výš nepůjdeme, ale v nejhorším případě jich stejně bude $N \cdot K$.) Můžeme si představit, že jsou uspořádány v tabulce, vodorovně jich je N, svisle K.

Potřebujeme zajistit, aby se hodnoty kontrastu v sousedních sloupcích (neboli sousedních pixelech displeje) nelišily o více než o D. Přidáme proto do grafu hrany. Budou orientované, povedou zleva doprava vždy mezi vrcholy v sousedních sloupcích, a to nanejvýš o D řádků výš nebo níž. Tím pádem bude každá cesta z prvního sloupce do posledního korektní nastavení displeje. To nastavení přečteme snadno, napíšeme si za sebe navštívené řádky a ty přesně odpovídají hodnotám kontrastu jednotlivých pixelů.

Jaká bude hodnota cesty? Součet změn pixelů. Hranám dáme ohodnocení. Je-li pixel nastaven na hodnotu 10, pak všechny hrany vedoucí do 10. řádku tohoto sloupce budou mít hodnotu 0. Všechny hrany vedoucí do 11. a 9. řádku budou mít hodnotu 1, protože hodnotu pixelu měníme o jedničku. Hrany do 12. a 8. řádku budou mít dvojku a tak dále. Hodnota cesty je samozřejmě součet vah hran.

Teď už jen najít nejkratší cestu z prvního sloupce do posledního. Pokud znáte pouze algoritmus, který hledá nejkratší cesty z jednoho vrcholu do dalšího, a ne z K vrcholů do některého z jiných K vrcholů, nezoufejte, upravíme si graf, abychom ho mohli použít. Přidáme si do grafu dva speciální vrcholy. Jeden bude startovní, z něho budou vycházet hrany do všech řádků prvního sloupce, ohodnoceny budou změnou kontrastu prvního pixelu oproti jeho původní hodnotě. Druhý speciální vrchol bude ten cílový, ze všech vrcholů posledního sloupce povedou hrany do něj. Ohodnoceny budou nulou.

Náš graf je orientovaný acyklický neboli po anglicku zkráceně DAG. Pro něj existuje rychlejší algoritmus na nejkratší cesty, než je Dijkstrův. Pro každý vrchol si budeme pamatovat délku zatím nejkratší nalezené cesty (na začátku bude na startu nula, ve zbytku grafu nekonečno) a vrchol, ze kterého jsme do aktuálního vrcholu přišli (podle toho pak zrekontruujeme nejkratší cestu). Značit je budeme jako $delka(u)$ a $odkud(u)$.

Budeme odebírat vrcholy v topologickém pořadí, (tak se to dělá u obecných DAGů), pro nás to znamená po sloupcích zleva doprava. Vezmeme vrchol a *zrelaxujeme* všechny jeho hrany. Tak se říká postupu, kdy zkusíme najít kratší cesty procházející danou hranou. Pokud si budeme váhu hrany značit jako $w(u, v)$, znamená to, že pro hranu (u, v) uděláme toto:

1. Pokud $cesta(u) + w(u, v) < cesta(v)$:
2. $\quad cesta(v) \leftarrow cesta(u) + w(u, v)$
3. $\quad odkud(v) \leftarrow u$

Řečeno slovy, známe již nejkratší cestu do u, a vede-li hrana z u do v, pak se do v mohu dostat za cenu rovnou délce cesty do u a váze hrany (u, v). Pokud je to lepší než dosud nejkratší nalezená cesta, tak si novou délku uložím a zapamatuji si, odkud jsem přišel.

A to je všechno. Už jenom vypíšeme délku nejkratší cesty a díky zpětným odkazům zrekonstruujeme průběh cesty (jen ho ještě musíme obrátit, dostaneme ho totiž pozpátku).

Určíme časovou složitost. Bereme do ruky každý vrchol, těch je $N \cdot K$, a každou hranu. Z každého vrcholu vede nanejvýš $2D+1$ hran, takže jich je celkem $\mathcal{O}(NKD)$, a to je i celková časová složitost. Paměti nám ale stačí jen $\mathcal{O}(NK)$. Hrany totiž nemusíme mít nikde uloženy, snadno si spočítáme, která hrana existuje a která ne.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-1-4.c`

Program (Python):
`http://ksp.mff.cuni.cz/viz/27-1-4.py`

Dominik Macháček

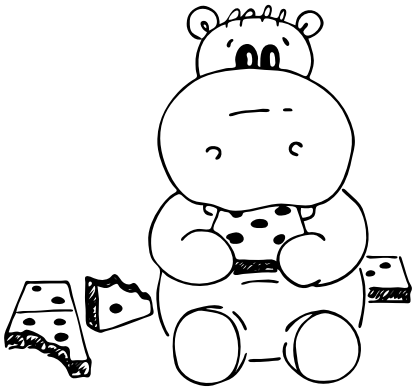
27-1-5 Napájení přístrojů

Tolik přístrojů a tolik různých požadavků na napětí... no naříkat na standardizaci můžeme jindy, teď pojďme vyřešit úlohu.

Vezměme si na chvíli požadavky na napájení jednotlivých přístrojů jako intervaly. Je jasné, že přístroje, jejichž intervaly se nepřekrývají, nemohou sdílet stejný zdroj napájení. A pokud budeme mít trojici intervalů, kde první se překrývá s druhým a druhý s třetím (ale první a třetí průnik nemají), můžeme sice vždy alespoň dva přístroje napájet z jednoho zdroje, ale třetí do toho nezařadíme. To je sice pěkné, ale co z toho?

Každý přístroj musíme z nějakého zdroje napájet. Když si vezmeme naše intervaly a podíváme se na interval (a, b), který končí nejdříve (jehož konec má nejnižší hodnotu), tak určitě musíme nějaký napájecí zdroj umístit mezi a a b, jinak bychom tento přístroj nepokryli. No ale má smysl nastavovat první napájení na menší hodnotu než b?

Protože všechny další konce intervalů jsou až za b, nemá. Nastavíme tedy první zdroj na b, připojíme na něj všechny přístroje, kterým vyhovuje, a odebereme jejich intervaly. Pak se podíváme na zbylé intervaly a budeme tento *hladový* postup opakovat, dokud nezapojíme všechny přístroje.



Teď je potřeba si rozmyslet dvě věci. První z nich je, jak rychle algoritmus běhá. Pokud už dostaneme intervaly setříděné podle konců, stačí nám je v čase $\mathcal{O}(N)$ od nejmenšího projít a postupně je označovat za zapojené, pokud je setříděné mít nebudeme, zvedne se nám čas tříděním na $\mathcal{O}(N \log N)$.

Druhou a zajímavější otázkou je, zdali to skutečně funguje. To by chtělo dokázat. Postupným zapojováním nějakých přístrojů na vyhovující zdroje vytvoříme nějakých K množin intervalů (tvořených z původních povolených intervalů napájení přístrojů obsažených v každé množině). Každou množinku si můžeme sjednocením převést na jeden nový interval, čímž nám vznikne K různých intervalů.

No ale protože do množinky pokaždé zapojíme všechny vyhovující přístroje, jsou jednotlivé vzniklé intervaly navzájem disjunktní. A pokrýt K disjunktních intervalů určitě nejde pomocí méně než K zdrojů, tím jsme dokázali optimálnost našeho řešení.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-1-5.c`

Program (Python):
`http://ksp.mff.cuni.cz/viz/27-1-5.py`

Jirka Setnička & Dominik Macháček

27-1-6 Přistávací světla

Dvojbarevná verze

Začneme jednodušší verzí úlohy, která používá dvě barvy (říkejme jim třeba **X** a **Y**) a chce po nás najít nejdelší „bílý“ úsek, tedy takový, v němž je obou barev stejně.

Úlohu převedeme na podobný problém s čísly: barvu **X** přepíšeme na $+1$ a **Y** na -1. Bílé úseky jsou nyní přesně ty se součtem 0. Poznáme je pomocí prefixových součtů z kuchařky: označíme p_i součet čísel na prvních i pozicích ($p_0 = 0$) a kdykoliv $p_i = p_j$, znamená to, že na pozicích $i + 1$ až j leží bílý úsek.

Chceme tedy v posloupnosti prefixových součtů najít dvě stejné hodnoty, které od sebe leží co nejdále. To provedeme snadno: budeme procházet vstup zleva doprava a průběžně si počítat prefixové součty. Pro každou možnou hodnotu prefixového součtu ($-n$ až n) si budeme v nějakém poli pamatovat, jestli jsme ji už viděli, a pokud ano, tak kde poprvé. Kdykoliv pak narazíme na hodnotu prefixového součtu, kterou jsme už viděli, spočítáme vzdálenost od prvního výskytu – to je délka nejdelšího bílého úseku končícího aktuálním prvkem – a započítáme ji do průběžného maxima.

Časová složitost bude lineární: $\mathcal{O}(n)$ na inicializaci pomocného pole a pak $\mathcal{O}(1)$ na zpracování každého prvku. Paměti nám také stačí $\mathcal{O}(n)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-1-6a.c`

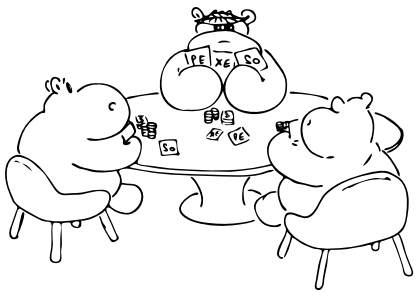
Od dvou barev ke třem

„Plnotučná“ verze úlohy pracuje se třemi barvami (třeba **R**, **G** a **B**) a chce po nás úseky, kde je všech třech stejně.

Pomůžeme si drobným úskokem: bílý úsek je ten, ve kterém je stejně **R** jako **G** a současně **G** jako **B**. To nám dává dvě instance dvojbarevné verze, které chceme vyřešit současně.

Vstup proto budeme překládat na posloupnost dvojic čísel (dvojsložkových vektorů): **R** bude $(1, 0)$, **G** bude $(-1, 1)$ a konečně **B** přeložíme na $(0, -1)$. Bílé úseky teď budou ty, které mají součet $(0, 0)$, přičemž sčítáme zvlášť první složku všech vektorů a zvlášť druhou.

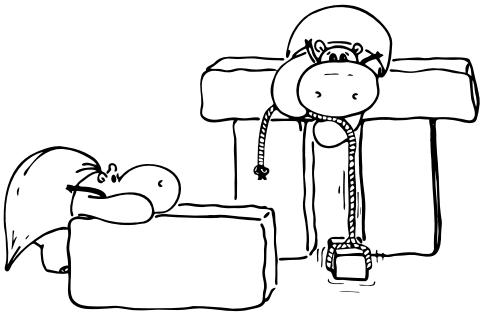
Opět do práce zapřáhneme prefixové součty – co na tom, že teď to nebudou čísla, ale dvojice čísel, fungovat budou stejně.



Jen už nemůžeme hodnotami prefixových součtů indexovat pole: muselo by být dvojrozměrné, takže by jeho inicializace trvala $\mathcal{O}(n^2)$ a zkazila nám jinak lineární složitost.

Místo pole si proto pořídíme chytřejší datovou strukturu, třeba vyvážený vyhledávací strom (dvojice porovnáváme lexikograficky). Dvojic je ve stromu nejvýše n, takže každá operace se stromem trvá $\mathcal{O}(\log n)$, a celý algoritmus tudíž potřebuje $\mathcal{O}(n \log n)$ času a $\mathcal{O}(n)$ prostoru.

Kdybychom si místo pole pořídili hešovací tabulku, bude jedna operace s dvojicemi trvat průměrně $\mathcal{O}(1)$, takže celý algoritmus poběží v čase průměrně $\mathcal{O}(n)$ a prostoru $\mathcal{O}(n)$.



Skutečně lineární řešení

Existuje ovšem řešení, které má lineární složitost i v nejhorším případě. Inspirujeme se přechozí úvahou o prefixových součtech, ale stejné dvojice nebudeme hledat on-line pomocí datové struktury, nýbrž dávkově tříděním.

Vytvoříme posloupnosti trojic: kdykoliv má i-tý prefixový součet hodnotu (a_i, b_i), přidáme trojici (a_i, b_i, i). Tyto trojice pak setřídíme lexikograficky, čímž se nám dostanou k sobě všechny výskyty stejných prefixových součtů. Snadno pak mezi nimi najdeme ty nejvzdálenější.

No dobrá, třídění trvá $\mathcal{O}(n \log n)$ a takhle rychlý algoritmus jsme už měli. Nenechte se mýlit, třídit jde i rychleji. V našem případě jsou složky trojic malá celá čísla, takže zafunguje přihrádkové třídění, které to zvládne v lineárním čase. Pokud ho ještě neznáte, je nejvyšší čas podívat se do kuchařky o třídění.[6]

Zde ho popíšeme aspoň stručně: pořídíme si přihrádky indexované od $-n$ do n (rozsah možných hodnot složek). Nejprve trojice rozházíme do přihrádek podle poslední složky a zase je vysbíráme (v pořadí od nejnižší přihrádky k nejvyšší). Pak provedeme totéž podle druhé složky a nakonec ještě jednou totéž podle první. Přitom u trojic, které padnou do téže přihrádky, stále zachováváme pořadí z předchozích průchodů. Proto nám nakonec vyjde přesně lexikografické pořadí.

Pokaždé strávíme čas $\mathcal{O}(n)$ rozhazováním trojic do přihrádek a $\mathcal{O}(n)$ jejich vysbíráváním. To provedeme celkem třikrát a pak ještě sekvenčně projdeme setříděné trojice, což nám všechno zabere $\mathcal{O}(n)$ času. Paměti nám stále postačí $\mathcal{O}(n)$ buněk.

Dodejme ještě, že kdybychom měli b barev namísto tří, použijeme b-tice a vše nám bude trvat $\mathcal{O}(bn)$ s pamětí $\mathcal{O}(bn)$. Též bychom k barvám mohli přidat i „anti-barvy“, které se chovají opačně – tím se bychom se přiblížili k prapůvodní fyzikální inspiraci úlohy teorií kvarků.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-1-6b.c`

Poznámka o rekurzi

Existuje i jiné lineární řešení, také založené na dvojím použití dvojbarevné verze úlohy.

Nejprve řešíme úlohu pro **R** a **G**. Tím nám vzniknou nějaké prefixové součty a pro každé dvě pozice i, j se stejným prefixovým součtem chceme vyřešit úlohu pro **G** a **B** mezi těmito pozicemi.

Rozházíme si tedy pozice ve vstupu do přihrádek podle hodnoty prefixového součtu pro **R** a **G**. Pro každou přihrádku pak spustíme novou instanci úlohy pro **G** a **B**; nebudeme-li pořád dokola inicializovat datové struktury a počítat prefixové součty, zvládneme to lineárně s množstvím pozic v této přihrádce. V součtu přes všechny přihrádky tedy $\mathcal{O}(n)$.

Implementační detaily nejsou triviální, ale pokud se jimi prokoušeme, uvědomíme si, že jsme právě popsali přihrádkové třídění „naruby“, tedy od nejvyššího řádu k nejnižšímu. Nic nového pod sluncem.

Poznámka o komplexních číslech

Ještě zmíníme jeden mírně bláznivý způsob, jak řešit tříbarevnou úlohu. Je zajímavý tím, že pro 4 barvy by už nefungoval a že v něm hrají hlavní roli komplexní čísla.

Pořídíme si komplexní třetí odmocniny z jedničky, totiž čísla $c_1 = 1$, $c_2 = -1/2 + i \cdot \sqrt{3}/2$ a $c_3 = -1/2 - i \cdot \sqrt{3}/2$. Tato tři čísla jsou rovnoměrně rozmístěna po jednotkové kružnici po násobcích $120\,^\circ$.

Nyní přeložíme **R**, **G** a **B** na čísla c_1, c_2 a c_3 a nahlédneme, že bílé úseky jsou přesně ty se součtem 0. (Rovnost imaginárních částí vynucuje, že všech **G** je stejně jako **B**. Z rovnosti reálných částí pak dostaneme, že **R** je stejně jako **G** i **B**.)

Nabízí se opět použít prefixové součty. Jenže radost nám poněkud kazí, že musíme počítat s divokými iracionálními čísly. Proto ještě změníme měřítko reálné osy vynásobením 2 a měřítko imaginární osy vydělením $\sqrt{3}/2$. Tím jsme nezměnili, které úseky mají nulový součet (reálná a imaginární složka se při sčítání neovlivňují), a dostali jsme samá celá čísla: $c'_1 = 2$, $c'_2 = -1 + i$, $c'_3 = -1 - i$.

Tato úvaha dále vede na podobná lineární řešení.

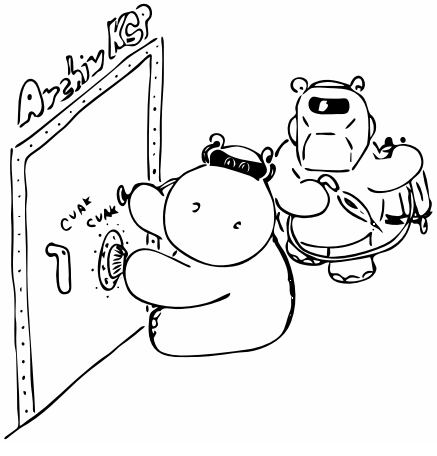
Martin „Medvěd“ Mareš

27-1-7 Učíme se s UNIXem

Jsme velmi rádi, že jste se UNIXu nezalekli a přišlo nám tolik vašich řešení. Většina úkolů byla jednoduchá, ale sem tam se objevily nějaké zrádnosti nebo zbytečně obtížné konstrukce.

Často bylo bodování obtížné, ale pokoušeli jsme se hodnotit praktičnost a jednoduchost vašich řešení a podle nich přidělovat nějaké zlomky bodů.

Pokusíme se tedy v autorském řešení ukázat, jaký je podle nás nejlepší přístup ke všem úkolům.



[6] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Úkol 1 – složky

Zde bylo správné podívat se do manuálových stránek příkazů `mkdir` a `rm`. Když se pokusíte pomocí prvního zmíněného vytvořit složku ve složce, která ještě neexistuje, vynadá vám. Pokud ale zavoláte `mkdir -p`, přepínač `-p` způsobí vytvoření všech složek po cestě.

Při mazání je možné mazat také pomocí `rmdir -p`, ale předtím je nutné vymazat vytvořený soubor. Pokud ale použijeme `rm -r` (`-r` jako zkratka za *rekurzivní*), můžeme rovnou vymazat všechno.

```
mkdir -p ~/a/b/c/d
touch ~/a/b/c/d/test
rm -r ~/a
```

Úkol 2 – head a tail

Zde bylo řešení velmi jednoduché. Rychlým pohledem do manuálových stránek jde zjistit, že příkazy `head` i `tail` mají oba přepínač `-n K`, který vypíše prvních, respektive posledních K řádků ze souboru. Jak ale vypsat všechno až na prvních K řádků?

Pečlivějším pročtením manuálových stránek se dá přijít na to, že zavoláním `tail -n+K` (všimněte si, že ani není potřeba mezera mezi přepínačem a jeho hodnotou) vypíše celý soubor začínající od K-tého řádku. My ale chceme K řádků vynechat, tedy chceme začít až od $(K+1)$-ního:

```
head -n15
tail -n15
tail -n+16
```

Úkol 3 – escapování

Problém zapisování podivných znaků se dá vyřešit dvěma způsoby: *escapováním* speciálních znaků nebo uzavřením do uvozovek. Při escapování musíme zrušit význam znaků jako závorka, mezera nebo otazník, při zavírání do uvozovek nám stačí dávat si pozor na jiné uvozovky. Správným řešením tedy mohlo být:

```
rm a\?c
rmdir "adresar[567]"
```

Úkol 4 – wildcardy

Toto byl první trochu více tvůrčí úkol a jsme rádi, že nám na něj přišla spousta různých elegantních řešení. Nakonec jsme se rozhodli hodnotit vaše wildcardové výrazy podle toho, jak efektivně jste použili dostupné prostředky.

Asi nejkratší rozumný výraz, který odpovídá zadaným pravidlům je tento:

```
*{[0-9]*[0-9],[a-zA-Z]*[a-zA-Z]}*/*.{jp{e,}g,gif}
```

Část `{[0-9]*[0-9],[a-zA-Z]*[a-zA-Z]}` říká to, že se někde v názvu musí nacházet buď dvě čísla, nebo dvě písmena oddělená libovolnými jinými znaky mezi sebou. K nim přidáme libovolné množství znaků před a za (obalení hvězdičkami) a tím máme část složky hotovou.

Název souboru může být libovolný a pak následuje přípona `{jp{e,}g,gif}`. Všimněte si hlavně elegantně použité vnořené složené závorky: ta nám říká, že se zde může nacházet buď `e`, nebo nic.

Přišlo nám i několik málo ještě šílenějších konstrukcí, ale výraz uvedený výše považujeme asi za optimální vzhledem k jeho složitosti a plně nám dostačoval.

Úkol 5 – datum v souboru

Hlavní věcí, na které byl tento úkol postaven, bylo přesměrování výstupu. Použití jedné šipky vždy soubor přepisuje celý, ale použití dvou šipek pouze přidává na konec. Správným řešením tedy bylo:

```
echo "Dnes je:" > datum
date >> datum
```

Poznámka: U příkazu `echo` by uvozovky ani nebyly potřeba, `echo` opíše na výstup všechny své parametry. Ale přijde nám, že s použitím uvozovek je příkaz přehlednější.

Úkol 6 – počítání složek a adresářů

V tomto případě stačilo přesměrovat výstup z příkazu `ls` do příkazu `wc` a tím spočítat řádky:

```
ls | wc -l
```

Část z vás možná zmátlo to, že `ls` vypisuje všechny soubory na jeden řádek a používali jste s ním přepínač `-1`. To ale není potřeba, jelikož `ls` je trochu speciální příkaz. Při spuštění si totiž „osahá“ svoje okolí a zjistí, jestli je spuštěný v rouře, nebo zobrazuje výstup uživateli. V tom druhém případě se pokusí svůj výstup zkrátit a vypisuje soubory na jednu řádku, v tom prvním je ale vypíše všechny normálně pod sebou.

Častou chybou v tomto úkolu bylo použití přepínače `-w` u `wc` namísto přepínače `-l`. Na první pohled se může zdát, že oba vracejí správný výsledek, ale to jen do chvíle, kdy se objeví soubor obsahující v názvu mezeru. V tom případě `-w` počítající slova již dá špatný výsledek.

Úkol 7 – kombinace head a tail

Hlavní kouzlo spočívalo v tom správně si spočítat řádky a pak nakombinovat příkaz. Jedenáctou až třicátou řádku získáme pomocí některého z následujících příkazů:

```
head -n30 < soubor.txt | tail -n+11 | wc -w
tail -n+11 < soubor.txt | head -n20 | wc -w
```

Zde se ale také objevila jedna z nejpodivnějších věcí na celém řešení, zhruba dvě pětiny z vás totiž místo třicáté řádky přečetli v zadání řádku třináctou a podle toho jste pak psali příkaz. Přivedlo nás to k zajímavému lingvistickému rozhovoru o čtení velmi podobných slov. :-)

Úkol 8 – velikosti souborů

V posledním úkolu jsme jasně nespecifikovali, že chceme jen soubory v aktuálním adresáři a nemusíte vypisovat rekurzivně všechny soubory v podsložkách (na což stejně v tomto dílu seriálu ještě ani nebyly ukázány vhodné příkazy a konstrukce). Omlouváme se za to.

Těm, kteří ale pro řešení použili nějaké složitější ještě nevysvětlené příkazy a povedlo se jim správně vyřešit obtížnější verzi, jsme také dávali plný počet bodů. Pokud jste ale zbytečně komplikované příkazy použili na řešení jen lehčí verze, o nějakou tu desetinku bodu jste přišli.

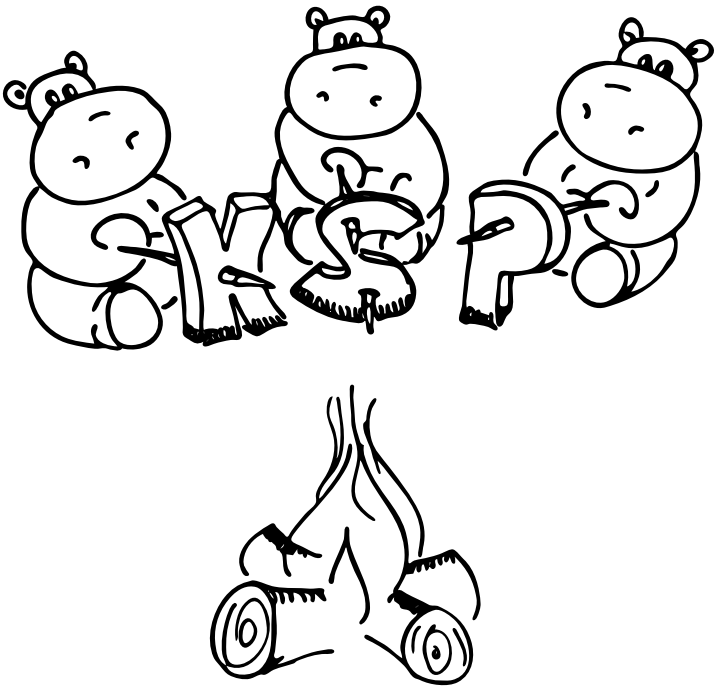
Nejtěžší na celém úkolu bylo zkonstruovat ten správný wildcard, kterému by vyhovovaly i skryté soubory (wildcardová hvězdička se totiž normálně na skryté soubory uvedené tečkou *nerozexpanduje*). Šlo to udělat například jako `{.,}*[0-9]*` (tuto konstrukci se složenou závorkou jsme mohli potkat už ve čtvrtém úkolu).

Na spočítání velikosti souborů pak šel využít příkaz `wc -c` a pokud jsme chtěli jen celkový součet (který `wc -c` vypisuje na poslední řádek), mohla celá konstrukce vypadat třeba takto:

```
wc -c {.,}*[0-9]* | tail -n1
```

Doufáme, že UNIXu zůstanete věrní i v další sérii.

Jirka Setnička & Marek Dobranský



Výsledková listina první série dvacátého sedmého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*1-1*	*1-2*	*1-3*	*1-4*	*1-5*	*1-6*	*1-7*	*série*	*celkem*
0.					8	9	10	10	10	12	14	56,0	56,0
1.	Marek Černý	G Chrudim	4	6	8	9	10	10	10	12	14	56,0	56,0
2.	Jan Špaček	G Wicht	4	6		9	10	3	10	10	13,6	53,8	53,8
3.	Martin Scheubrein	G MNám Třb	3	1	8		10	4	10	8	14	52,7	52,7
4.	Richard Hladík	GOAMarLaz	2	11	8	9	10	10	10	10	12,3	52,3	52,3
5.	Jan Tománek	GPelhřimov	4	2	8	2	10	10	10	8	11,4	52,2	52,2
6.	Jakub Tětek	CírkG Plzeň	1	1	6,5	9	10	0		11	13,5	52,1	52,1
7.	Přemysl Šťastný	GŽamberk	2	4	8	9	9	10	10	3	11,7	51,3	51,3
8.	Stanislav Lukeš	GPísnickáPH	2	2	7,5	9	9	1	0	11	10,8	51,1	51,1
9.	Michal Töpfer	G DrJPekMB	2	1	7	9	2	3	10	9	12,2	50,6	50,6
10.	Adrián Goga	SPŠNitra	4	1	8		9	2	10	4	13,6	49,3	49,3
11.	Anna Gajdová	GFPValMez	4	3	6	9	10	2	10	4	11	49,2	49,2
12.	Štěpán Hudeček	G Litovel	3	1	7,5	7	5	3	10	4	13,8	48,3	48,3
13.	Jan Knížek	G Strakon	4	14	7,5	8	7	3	10	10	13,9	48,1	48,1
14.	Jiří Sejkora	GVoděraPH	3	1	8		2	4	10	11	12,6	48,0	48,0
15.	Róbert Selvek	G KošiceS	3	1	8	4	10	1	1	4	14	46,3	46,3
16.	Jan Kočur	G Wicht	4	1	6	9	2		0	11	11,8	46,2	46,2
17.	Lukáš Ulrich	SSŠVTPraha	4	1	8	3	2	2		8	11,6	42,3	42,3
18.	Václav Šraier	GČeskoliPH	2	1	8	3	4	4		4	13,2	42,2	42,2
19.	Jakub Zárybnický	GTomkovaOL	4	6	6,5	6	7	4	0		12,5	39,9	39,9
20.	Václav Volhejn	GKepleraPH	2	11		8,5	10		10	11		39,5	39,5
21.	Jan Gocník	GJŠkodyPŘ	3	1	8			4	6		14	34,6	34,6
22.	Václav Končický	GSOŠ FrMís	4	2	6		3			6	13	32,9	32,9
23.	Michal Převrátil	GKlatovy	2	1	8		10	1			9,4	31,1	31,1
24.	Martin Zoula	GNadKavaPH	3	1	7		3			2	9	30,9	30,9
25.	Jiří Vozár	G UherBrod	3	1	5		3				14	26,6	26,6
26.	Pavel Turinský	G Brandýs	2	1	8						12,7	21,7	21,7
27.	David Cholewa	GMatOS	4	1	7	7				2		21,0	21,0
28.	Filip Bialas	GOpatovPHA	2	5				10	10			20,0	20,0
29.	Barbora Sedláková	GKonštanPV	4	1	3			1			9	18,9	18,9
30.	Vít Macura	GOAMarLaz	2	1	6,5						5	16,9	16,9
31.	Jan Bouček	GKepleraPH	2	1						4	4,9	16,8	16,8
32.	Jakub Lukeš	GNAlejíPH	2	1							13,9	14,0	14,0
33.	Dalimil Hájek	GKepleraPH	4	15							13,6	13,4	13,4
34.	Martin Kubeša	GJŠkodyPŘ	3	1							10	12,8	12,8
35.	David Juřica	GNadŠtolPH	2	2				0			7,1	10,9	10,9
36.	Jan Kaifer	GČesBrod	−1	1				0			6,3	10,6	10,6
37.	Michael Novák	SSŠVTPraha	4	1				2				2,0	2,0