Korespondenční Seminář z Programování

27. ročník **KSP** **Prosinec 2014**

Milí řešitelé a řešitelky!

Vánoce jsou tu a s nimi přišel i čas dárků. I my pro vás máme jeden dárek, a to zadání třetí série KSP. Až vás omrzí neustálé pojídání cukroví, či si jen budete chtít zpestřit dlouhé večery, nahlédněte do úloh a zkuste nějaké vyřešit. Opět jsme připravili mix praktických i teoretických úloh korunovaných dalším dílem seriálu o UNIXu. Příjemné čtení!

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

Dodáváme, že 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 9. února 2014 v 8:00 SEČ** (CodEx má termín o 24 h 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: Řešitelům, kteří **z každé úlohy získají alespoň dva body**, pošleme **čokoládu**.

Třetí série dvacátého sedmého ročníku KSP

Letos se v jednotlivých sériích ohlížíme za zajímavými programátorskými chybami, a nejinak tomu bude i dnes. V předchozích sériích jsme viděli dělení nulou, ale také zrádnou chybu vzniklou převodem mezi celým číslem a floatem. Dnes nás oproti tomu čeká chyba, která vznikla zejména lidským přehlédnutím a strojovou kontrolou by byla těžko odhalitelná.

Také již opustíme válku v Perském zálivu a přesuneme se o několik let v čase, do doby, kdy většina z vás už byla na světě. Dnešní chyba ani nebude mít tak tragické následky, za oběť jí padlo „pouze“ několik set milionů dolarů. Teď se ale pojďme podívat do září 1999 na Patrickovu leteckou základnu.

⋆⋆⋆

James zaujatě pozoroval jednu z fotografií na zdi, zatímco hučení za ním sílilo. Když se ozvalo charakteristické cvaknutí oznamující, že voda je uvařená, vzal rychlovarku a zalil si kávu. Kuchyňkou se rozlila typická vůně.

Vyzbrojený milovaným nápojem se James vrátil do řídicí místnosti, kde se přidal ke svým kolegům navigátorům. Teď neměli mnoho práce, ale už za pár dní budou jejich znalosti velmi potřeba. Blížil se totiž čas, kdy Mars Climate Orbiter *vstoupí na oběžnou dráhu Marsu.*

Malé pozdvižení se ovšem dostavilo mnohem dříve. Na Zemi dorazila první fotografie Marsu. Pravda, obraz byl zkomprimovaný a možná patřičně nepřesný, ale navigátoři hned začali zkoumat, jestli na něm neobjeví vhodné místo k přistání. Po Mars Climate Orbiter, který má zkoumat atmosféru Marsu z jeho oběžné dráhy, totiž přijdou další sondy, a ty již budou na Rudé planetě přistávat.

27-3-1 Plocha k přistání — 14 bodů

Na Zemi dorazila fotografie zkomprimovaná do *kvadrantového kódu*. Nás zajímá, jaké místo na ní by bylo nejvhodnější k přistání, to znamená, kde je největší souvislá plocha.

Kvadrantový kód se používá pro dvoubarevné obrázky. Funguje tak, že se obraz nejprve rozdělí na čtvrtiny, které se postupně zakódují (pořadí kódování čtvrtin je „po řádcích“). Má-li celá plocha stejnou barvu (či je již tvořená jen jediným pixelem), zakóduje se jako jedno číslo (1 pro černou nebo 0 pro bílou barvu), v opačném případě se zpracovává rekurzivně.

Příklad takového kvadrantového kódu, který vznikl zakódováním z dvoubarevného obrázku, připojujeme níže. Tento zápis kvadrantového kódu je konzistentní s pátou úlohou, která ho také využívá.

```
1 1 0 0
1 1 0 0   ==>  (10(1100)(1010))
1 1 1 0
0 0 1 0
```

Vaším úkolem je v kvadrantovém kódu najít největší souvislou bílou oblast. Za sousední pixely považujeme jen ty, které spolu sousedí hranou (roh nestačí). Počítejte s tím, že se rozkódovaný obraz nevejde do paměti (tedy převést kvadrantový kód na obrázek a hledat oblast až v něm správné řešení není).

Poznámka: Kvadrantový kód funguje pěkně pro čtvercové obrázky o hraně délky nějaké mocniny dvou, ale dá se obdobně definovat i třeba pro obdélníkové obrázky. Protože to ale nepřináší nic nového, omezíme se v řešení úlohy jen na čtvercové obrázky o hraně délky mocniny dvou.

James po chvíli nechal své kolegy dál zkoumat a sám se ponořil do vzpomínek . . .

⋆⋆⋆

Když bylo v srpnu 1993 jen těsně před vstupem na oběžnou dráhu ztraceno spojení se sondou Mars Observer*, a tím podstatně oddáleny šance na bližší poznání Rudé planety, byl to šok, zvlášť pro Jamese a jeho tým.*

Netrvalo ale dlouho a začaly se připravovat nové mise. Problém vesmírných misí ovšem je, že stojí spoustu peněz, které na ně musí někdo přidělit. Za Jamesem brzy přišel šéf, že bude potřeba napsat žádost o grant. Naštěstí tehdy dobře věděli, na co jednotliví členové komise, která bude

o schválení rozhodovat, slyší; mohli jim tedy napsat návrh na míru. Zajímalo je ale, jakou mají vlastně konkurenci.

27-3-2 Návrhy pro komisi — 12 bodů

Je potřeba podat návrh komisi a nás by zajímalo, kolik různých návrhů komise schválí. Komise má C členů, kteří všichni sami za sebe rozhodují o schválení návrhu. Jako celek pak komise návrh schválí, pokud ho schválí alespoň K jejích členů.

Návrhy jsou ovšem dlouhé a členům se nechce číst je celé. Každý člen má proto nějaký seznam slov, která se mu líbí, a schvaluje právě ty návrhy, které začínají některým z jeho oblíbených slov. Návrhy i oblíbená slova jsou řetězce složené z malých písmen anglické abecedy a navíc panuje dohoda, že každý správný návrh má délku právě D písmen.

Na vstupu tedy dostanete počet členů komise a pro každého z nich jeho oblíbená slova. Dále dostanete počet členů nutných ke schválení a přijatelnou délku návrhů D. Vaším úkolem je zjistit, kolik různých návrhů (tvořených jen z malých písmen anglické abecedy) může komisí projít jako schválené.

Zajímá nás jen počet těchto návrhů, nemusíte je generovat. Navíc se nemusíte zabývat tím, že se vám toto číslo nevejde do běžné číselné proměnné (toto není úloha na velká čísla).

Příklad: Uvažme trojčlennou komisi, ve které jsou potřeba alespoň dva její členové ke schválení návrhu, a návrhy délky čtyř písmen. Oblíbená slova jednotlivých členů vyjadřuje tabulka níže.

```
1. člen: pes psa
2. člen: psal kun
3. člen: pest ps
```

Je jasné, že návrh musí začínat na `p`, jinak by ho neschválili alespoň dva členové (na slovo `kun` tedy můžeme zapomenout). Možnosti, které nám zbývají, jsou tedy buď `pest` nebo `psaX`, kde `X` může být libovolné písmeno (všimněte si, že třeba `psbX` už je přijímané jen jedním členem komise, `psal` všemi a `psat` alespoň dvěma).

Možností je tedy dohromady $26 + 1 = 27$.

Snad právě proto, že znali preference jednotlivých členů komise, nebylo pro Jamesův tým těžké peníze získat. Po vyřešení finanční otázky ovšem přišly na řadu otázky další, techničtější a v mnohém složitější.

Většinu konstrukčních záležitostí řešila společnost Lockheed Martin*, se kterou NASA uzavřela smlouvu na výrobu sondy, přesto občas některé řešené problémy probublaly i k Jamesovi. K těm zajímavějším patřila konstrukce antény.*

Jednou z klíčových vlastností každé vesmírné sondy je totiž schopnost komunikovat s lidmi na Zemi. Od začátku bylo jasné, že na straně Země se k tomuto účelu využije síť Deep Space Network*, která byla na Zemi vybudovaná již koncem šedesátých let a využívá se pro komunikaci s jinými sondami.*

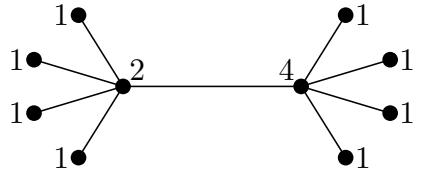
Aby mohla sonda do této sítě posílat informace, musí být ovšem vybavená dostatečně silnou anténou. Taková anténa se skládá z mnoha vysílačů, jejichž volba byla trochu oříšek. Tím spíš, že ač peníze byly, plýtvat se jimi nemohlo.

27-3-3 Výběr vysílačů — 13 bodů

Anténa vesmírné sondy má stromovou strukturu, přičemž v každém uzlu se nachází nějaký vysílač. Kvůli rušení ale v žádných dvou sousedních uzlech nesmí být vysílače stejných typů.

Různé vysílače mají různou cenu, i-tý typ vysílače stojí 2^i dolarů (číslujeme od 0). Na vstupu dostanete popis antény, tedy který uzel sousedí s kterým. Určete, kolik nejméně dolarů bude stát umístění vysílačů do všech anténních uzlů.

Příklad: Na anténě níže vidíte, že v tomto případě je nejvýhodnější použít tři typy vysílačů (s cenami $2^0, 2^1, 2^2$ neboli $1, 2, 4$). Použít jen dva typy vysílačů by v tomto případě vyšlo dráž.



Lehčí varianta (za 3 body): Jako součást řešení vymyslete nějaký rozumně malý příklad antény, na které je potřeba použít čtyři různé druhy vysílačů, aby výsledná cena byla co nejmenší.

Rozumně malým příkladem nemyslíme nutně, aby měl co nejméně vrcholů to jde, ale spíše aby byl rozumně jednoduše zkonstruovatelný (jednoduchý popis konstrukce je lepší než obrovský obrázek o tisíci vrcholech).

⋆⋆⋆

Uběhlo několik dní od chvíle, kdy Jamese hlas jednoho z jeho kolegů vytrhl ze vzpomínání a vrátil do reality. To navigátoři museli vyměnit hledání souvislé oblasti na fotografii za počítání, kontrolování, konzultování, nové počítání a tak stále dokola. Sonda se totiž rychle blížila k Marsu a bylo třeba navést ji na takovou dráhu, z které se dostane do správné výšky nad povrchem planety.

Ještě ten den spočítali vše potřebné. O týden později, ve středu 15. září 1999, byl provedený čtvrtý manévr upravující trasu letu. Očekávalo se, že až se sonda 23. září dostane do blízkosti Marsu, bude se nad jeho povrchem nacházet ve výšce 226 kilometrů. Teď, tři dny před očekávaným vstupem na oběžnou dráhu, ovšem navigátorům vycházelo, že při zachování trajektorie bude výška mnohem menší.

James se zamračil na obrazovku počítače. Pak rychle něco naťukal do kalkulačky, kterou měl položenou před sebou, ale stále mu vycházelo málo. 158. 158 kilometrů nad povrchem Marsu místo očekávaných 226. To bylo o dobrou třetinu méně. Zatím to nebylo kritické, Mars Climate Orbitter by měl s patřičnou úpravou oběžné rychlosti přežít ještě ve výšce 80 kilometrů, ale komu by se líbilo, když se realita takovým způsobem liší od očekávání? James se navíc děsil, že další den vyjde ještě méně.

Přitom od počátku mise probíhala dobře . . .

⋆⋆⋆

Psal se 11. prosinec 1998 a spousta lidí v čele s konstruktéry a navigátory sledovala start nosné rakety Delta II*, která měla Mars Climate Orbiter dopravit na Hohmannovu elipsu.*

Mezi sledujícími James pochopitelně nemohl chybět, ačkoliv on kromě rakety důsledně sledoval i lecjaké naměřené údaje. Po odpočtu, během kterého ještě víc vystoupalo očekávání všech zapojených, byly zažehnuty motory. Objevil se

jasný záblesk, který přešel v ohnivou čáru, a Delta II vystřelila vstříc modrému nebi, a ještě dál.

Tak začala 669 milionů kilometrů dlouhá cesta sondy, která měla odpovědět na mnoho otázek pozemšťanů.

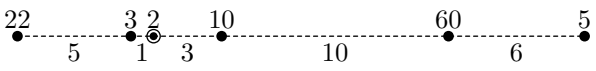
Let probíhal dobře, jen v jedné chvíli navigátoři zvažovali, zda by se nevyplatilo nechat sondu chvíli poletovat tam a zpět, aby její solární panely nasbíraly co nejvíc energie.

27-3-4 Doplňování energie — 12 bodů

Sonda prolétá vesmírem, kde některými místy prochází výjimečně silné sluneční paprsky. Solární panely dokáží z těchto paprsků získat energii, ovšem na přelet mezi místy vzdálenými i spotřebuje sonda i jednotek energie. Navíc odpadní látky zastíní paprsek, takže z jednoho místa lze energii čerpat pouze jednou.

Na vstupu dostanete popsáno, jaké množství energie se nachází v jednotlivých místech, a vzdálenosti mezi těmito místy. Dále dostanete určený výchozí bod, na kterém se sonda nachází. Určete, s jakou největší energií může sonda skončit.

Například pro situaci níže (horní čísla představují množství energie, dolní vzdálenosti mezi místy) se začátkem ve třetím bodě může sonda skončit maximálně se 64 jednotkami energie. Nejlepší řešení se z výchozího místa vydá těmito přelety: LRRLLLRRRR (L – doleva, R – doprava).



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.

⋆⋆⋆

Jamesovy obavy nebyly plané, během dalších dvou dní klesla očekávaná výška, v které by sonda měla k planetě přiletět, o dalších 50 kilometrů. To by ale ještě stále mělo stačit. A dál už očekávaná výška klesat nemohla, chvíle, kdy Mars Climate Orbiter vstoupí na oběžnou dráhu Marsu, již byla na dosah.

Právě proto bylo v řídicí místnosti rušno jako málokdy, přestože ještě nebyly ani čtyři hodiny ráno. Blížil se jeden z převratných okamžiků kosmonautiky.

Stále nebylo jasné, proč se očekávání a realita tak rozchází. Výška, ač aktuálně odhadovaná na málo přes 100 kilometrů nad povrchem Marsu, ovšem dostačovala a vstup na oběžnou dráhu byl zahájen. Sonda složila své solární panely, vhodně se vůči planetě natočila a zažehla hlavní motor.

Ve čtyři hodiny a čtyři minuty bylo spojení se sondou zničehonic přerušeno. Navigátoři si vyměnili několik vyděšených pohledů. Snažili se obnovit kontakt, ale nedařilo se.

Ani ne o dvě minuty později měla sonda navíc vstoupit do zákrytu Marsu, kdy by tak jako tak nebylo možné s ní komunikovat. Nedaří se navázat spojení, jen protože je sonda v zákrytu, nebo protože se stalo něco mnohem ošklivějšího?

Jamesovi padl pohled na fotografii, která před dvěma týdny ze sondy dorazila. Tehdy to ještě šlo všechno skvěle!

⋆⋆⋆

Kdyby měla sonda lidské pocity, asi by se na své cestě dost nudila. Po zajímavém startu a troše poletování tam a zpět za světelnými paprsky již nic zajímavého nepřišlo. Zůstal jen dlouhý let černou tmou zpestřený pouze světly hvězd.

Po dlouhých devíti měsících sonda konečně doletěla na dohled Marsu. Ještě z velké dálky pořídila jeho fotografii, a protože na fotografii planety z vesmíru je mnoho tmavého místa, stejně jako mnoho světlého místa, rozhodl se počítač odeslat ji na zemi kvadrantisticky zkomprimovanou.

27-3-5 Komprese obrazu — 10 bodů

Sonda posílá snímek Marsu. Nejprve ho ovšem za pomoci ztrátové kvadrantistické komprese převede do kvadrantového kódu (popsaného v první úloze).

Při *kvadrantistické kompresi* se jedna čtvrtina obrazu prohlásí za celočernou, jedna za celobílou a zbylé dvě se zpracují rekurzivně. Pokud se rekurze dostane až na úroveň jednotlivých pixelů, může být už barva rekurzivních částí jakákoliv. Pro čtverec 2×2 ale ještě platí, že jedna jeho čtvrtina musí být celočerná, jedna celobílá a zbylé dvě libovolné. Pořadí kvadrantů je „po řádcích“.

Na vstupu dostanete původní obraz. Vaším úkolem je vypsat kvadrantový kód takové jeho kvandrantistické komprese, která se od původního obrazu liší v co nejméně pixelech.

Konkrétněji bude mít vstup podobu popisu obrázku ve formátu PBM.[2] To je jednoduchý formát na ukládání černobílých obrázků.

Obrázek je v něm kódovaný po řádcích, vždy jedno číslo (1 nebo 0) na jeden pixel. Na řádku jsou mezi jednotlivými čísly mezery a na konci každého řádku se nachází znak nového řádku, nic jiného se zde nevyskytuje. Platný PBM soubor je také uvozen na prvním řádku znaky `P1` a na druhém řádku mezerou oddělenými čísly udávajícími jeho šířku a výšku (v tomto pořadí).

Obrázky v této úloze budou pro jednoduchost vždy čtvercové o hraně 2^K pixelů a jejich velikost nepřesáhne 1024×1024 pixelů.

Na výstup vypište nejprve na první řádek počet změněných pixelů a následně na druhý řádek kvadrantový kód kvadrantistické komprese.

Ukázkový vstup:

```
P1
8 8
1 1 1 1 1 1 1 0
1 1 1 0 1 1 1 0
1 0 0 0 1 1 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0
```

Ukázkový výstup:

```
8
((1(1110)(1000)0)
 (1(1010)(1110)0)
 1
 0
)
```

Poznámka k příkladu: Pro přehlednost příkladu jsme druhý řádek výstupu rozlomili po jednotlivých kvadrantech, v reálném výstupu by vše od první do poslední závorky bylo na jediném řádku.

Ve webovém zadání naleznete ještě jeden další ukázkový vstup.

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

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

[2] `http://en.wikipedia.org/wiki/Netpbm_format`

K prohlédnutí obrázku ve formátu PBM můžete využít na Linuxu např. program Eye of Gnome (eog), na Windowsech programy Irfan View nebo XnView. Na obou systémech si s PBM poradí i Gimp či OpenOffice Draw.

⋆⋆⋆

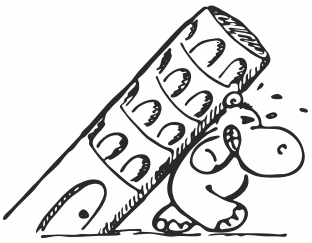
Jamesova myšlenka, že tehdy *to ještě šlo skvěle, se časem ukázala jako nepříjemně přesná. Mars Climate Orbitter se totiž navigátorům neozval nejen po dvaceti minutách, kdy se měl opět dostat mimo zákryt Marsu, ale ani po hodině, ani po dvou dnech.*

Po těchto dvou dnech byla sonda oficiálně prohlášena za ztracenou a mise za neúspěšnou. Navigátoři zpětně spočítali, že sonda se ve skutečnosti dostala do výšky pouhých 57 kilometrů nad povrchem Marsu, kde ji zřejmě spálila atmosféra.

Ještě před oficiálním ukončením mise bylo zahájeno vyšetřování s cílem zjistit, co se vlastně stalo a proč se sonda pohybovala mnohem níž, než všichni očekávali.

James si rychle zvykl, že se teď kolem něj pohybuje mnohem víc lidí, že se zkoumá hned tu, hned ono. I jeho samotného zajímala příčina tohoto selhání a snažil se přijít věci na kloub.

Do místnosti právě vešel i jeden z techniků. „Hej, lidi, pomůžete mi někdo uklidit přepravky ve skladu?“ ptal se hned místo pozdravu. James usoudil, že trocha fyzické aktivity mu jen prospěje a přidal se k několika ochotným pomocníkům.



27-3-6 Ukládání přepravek — 9 bodů

Ve skladu je třeba uspořádat přepravky, a to tak, aby zabíraly co nejméně místa. Přepravky jsou kulaté, každá má svůj vnější a vnitřní průměr. Pokud je vnější průměr jedné přepravky menší než vnitřní průměr druhé přepravky, dají se vložit do sebe (a do nich případně ještě menší přepravka, vznikají tak jakési „komínky“).

Na vstupu dostanete vnitřní a vnější průměry všech N přepravek. Vaším úkolem je zjistit, do kolika nejméně komínků se dají uspořádat.

„Tedy, tohle bude mít pěkných pár liber,“ prohlásil jeden z pomocníků zvedaje pořádný komínek mnoha přepravek.

„Cos to řekl?“ vytřeštil oči Jamesův kolega Thomas.

„Jen že je to těžké. . . “ bránil se pomocník.

Ostatní, včetně Jamese, Thomase jen nechápavě pozorovali.

„O to nejde. Jde o ty libry! A o to, že libry nejsou kilogramy,“ pokračoval vzdor nechápavým pohledům Thomas. „A taky o to, že kilogramy jsou to, co ta sonda očekávala.“

Ozvala se hlasitá rána. To Jamesovi z rukou vypadlo několik přepravek. A podle výrazů ostatních byla spíš náhoda, že se to samé nestalo více lidem.

⋆⋆⋆

Vyšetřování potvrdilo, že příčinou selhání byla neshoda v používaných jednotkách. Řídicí středisko ze Země odesílalo instrukce s imperiálními mírami, kdy sílu udávalo v silových librách. Sonda je ovšem očekávala v metrické podobě, tedy sílu čekala v Newtonech.

Jelikož silová libra je více než čtyřnásobek Newtonu, došlo při výpočtech k chybám, které byly pro úspěšnost vstupu na oběžnou dráhu fatální. Rozkol mezi očekávanou a naměřenou pozicí byl zaznamenaný a v týmu zodpovídajícím za let družice se uvažovalo o provedení ještě dalšího, pátého, manévru korigujícího dráhu, ten ale nebyl nikdy provedený.

Neúspěšnou misi s vámi sledovala

Karolína „Karryanna“ Burešová

27-3-7 UNIXové déjà vu — 15 bodů

Dnešní díl seriálu ve vás možná vyvolá pocit, že jste ho už někdy četli, ale ne tak docela. Vrátíme se totiž k mnohému z toho, co už umíte, a pronikneme ještě o kousek hlouběji. Připravte se na vydatnou porci povídání o nápovědě, o souborovém systému, uživatelích a právech, o řídicích strukturách a funkcích v shellu a o formátovaném výstupu.

Z předchozích dílů seriálu máte k dispozici shell, nejspíš Bash, umíte se v něm pohybovat po souborovém systému a zvládáte psát jednoduché skripty pro manipulaci s obsahem textových souborů. Také když zapomenete přepínače konkrétního příkazu, umíte si je v manuálových stránkách najít.

Umíte si ale pomoct, když zapomenete, jak se nějaký příkaz jmenuje?

Nápověda

Je nemožné si pamatovat všechny vlastnosti každého nainstalovaného programu, natož stíhat sledovat změny. Nápověda, manuál nebo dokumentace jsou základními prameny informací pro uživatele libovolného software a UNIX v tomto ohledu není výjimkou. Sebelepší informace je ale k ničemu, když ji neumíte najít.

Příkaz `man` už znáte. Napadlo vás podívat se na `man man`?

Zjistíte tam, že k hledání řetězců v popisech příkazů slouží přepínač `-k`. Pokročilejší možnosti nabízí utilita `apropos`.

Také narazíte na přepínač `-s`, jehož hodnotou je sekce manuálu a někdy jde dokonce přepínač vynechat a psát jen sekci (`man 1 man`). Počkat, co jsou sekce? Jsou očíslované, každá stránka je v nějaké zařazena a obvykle ji má uvedenou v závorce za svým jménem (např. `cat(1)`, `shells(5)` nebo `standards(7)`). Konkrétní význam a číslování se liší, POSIXový standard (který si zmíníme dále) o nich nemluví vůbec. Pro představu se podívejme na Debian Linux:

1 Spustitelné programy nebo příkazy shellu
2 Systémová volání (funkce poskytované jádrem)
3 Knihovní volání (funkce v programových knihovnách)
4 Speciální soubory (obvykle nalézané v `/dev`)
5 Souborové formáty a konvence (např. `/etc/passwd`)
6 Hry
7 Směs (včetně balíků maker a konvencí)
8 Příkazy administrace systému (obvykle jen pro roota)
9 Funkce jádra

V různých sekcích můžete najít stejnojmenné stránky. Například `passwd(1)` je utilita pro změnu hesla a `passwd(5)`[3] dokumentuje databázi uživatelů `/etc/passwd`. Musíte buď sekci znát, nebo použít přepínač `-a` a postupně si prohlédnout všechny.

[3] sekce 5 na Linuxu, jinde nejspíš jiná

Když už konkrétní stránku máte, pořád není vyhráno. Může být dlouhá a nepřehledná. Pak vám pomůže váš stránkovač. Příkaz `man` by mohl vysypat horu textu přímo do terminálu se škodolibým úsměvem a slovy „poraďte si“, jako příjemnější se ale ukázalo, když pustí `less` nebo starší a standardní `more` a manuál vám ukáže v něm. V prvním dílu jsme vám prozradili, že se oba zavírají klávesou `q` (quit), přidáváme `h` (help) pro nápovědu, `/` (lomítko) pro vyhledávání (potvrdíte enterem) a `n` (next) pro vyhledání dalšího výskytu.

Aby to nebylo příliš jednoduché, k manuálu existuje alternativa: infostránky. Některé jsou mnohem obsáhlejší než odpovídající manuálová stránka a jsou členěné, nevypadají jako jeden dlouhý dokument. Jejich prohlížeč se jmenuje `info`, nápovědu v něm získáte napsáním otazníku, zbytek už zjistíte sami.

Bash k nápovědě přistupuje po svém. Na `man bash` najdete i popis jeho vestavěných příkazů, jako je `cd` nebo `pwd`, kdo by ovšem chtěl hledat jehlu v kupce sena? Vysvobodí vás jeho příkaz `help`. Mrkněte na `help help`, je vcelku intuitivní.

Pokročilejší z vás by mohlo zajímat, které utility a jejich přepínače mají být dostupné na všech UNIXech, ať už je to Gnu/Linux, Solaris, OS X nebo nějaká odnož BSD. Taková znalost slouží k psaní přenositelných skriptů, tedy skriptů, které budou fungovat i na jiném systému, než na kterém jste je napsali. Vaši zvědavost ukojí norma POSIX, které se certifikované UNIXy držet musejí a ty ostatní aspoň plus minus chtějí. Kdykoliv se budeme odvolávat na normu nebo POSIX, myslíme POSIX 2013.[4] Na jeho stránce je vpravo dole odkaz ke stažení té kupky HTML stránek v jednom archivu, z neoficiálních zdrojů je možné sehnat POSIX i v podobě manuálových stránek. V Debianu je takovým zdrojem balík `manpages-posix` v repozitáři non-free.

Souborový systém

První díl seriálu se vás snažil nezahltit a nerozptylovat, o souborovém systému řekl jen to nejnutnější, minule jste nakoukli do práv souborů, když jste vytvářeli spustitelný skript. Je načase povědět o souborovém systému víc.

Logicky je souborový systém jediný, s kořenem `/` („root“), fyzicky jich ale bývá víc, z nichž některé mohou sídlit třeba jen v operační paměti nebo dokonce na úplně jiném stroji. Všechny dostupné na aktuálním stroji si můžete prohlédnout příkazem `df`. Vypíše pro každý souborový systém do tabulky název, velikost, využití a kam v logickém souborovém systému je připojený. Často je k dispozici přepínač `-T`, se kterým `df` ukáže i typ souborového systému, a přepínač `-h`, se kterým vypíše obsazené a volné místo v lidsky čitelných jednotkách.

Prostor zabraný konkrétním souborem umí spočítat `du`. Pokud dostane adresář, rozpitvá statistiku na jeho položky, podobně jako je tomu u `ls`. Příkazu `ls` to můžete zakázat přepínačem `-d`, obdobně u `du -s`.

Nenechte se zmást tím, že `df` i `du` přemýšlejí v blocích. Je to dáno běžnou strukturou disků, soubor zabírající blok jen z části nemůže jeho zbytek přenechat jinému souboru, přebytečné místo zůstane nevyužité. Velikost bloku se obvykle liší mezi normou, utilitami a diskem, buďte tedy obezřetní a uvědomujte si, jaká velikost se u vás kde používá.

Příkaz `ls` s přepínačem `-l` zobrazuje velikost souboru v bajtech a na zabrané bloky se nijak neohlíží. Počet zabraných bloků nechá zobrazit přepínač `-s`. Celkovou velikost zabraných bloků v lidsky čitelných jednotkách ukazuje `du -h`.

Úkol 1 [1b]: Zjistěte, jak velké bloky používá váš disk a vaše utilita `du`, která by podle normy měla používat 512B bloky. Svá zjištění doložte použitými příkazy, jejich výstupy a popisem své úvahy.

Konkrétní použitý souborový systém s sebou nese svá omezení. Na Windows se kdysi používal formát FAT, později NTFS, v Linuxu jsou doma ext2 až ext4, v BSD a Solarisu ufs. Lišit se mohou maximální délkou jména souboru, maximální velikostí souboru, maximální využitelnou velikostí disku, povolenými znaky v názvech souborů, (ne)podporou ukládání různých metadat, ...

Norma vyžaduje, aby souborový systém rozlišoval malá a velká písmena a názvy souborů neobsahovaly lomítko (oddělovač komponent cesty) a NUL (`\0` v jazyce C, bajt s hodnotou 0). Jazyk C vznikl pod UNIXem a UNIX do něj byl po čase přepsán, jsou spolu dodnes hodně prolnuté. Když zakážeme znak NUL, máme zaručeno, že je možné název souboru považovat za řetězec jazyka C a používat na něm řetězcové funkce, např. `strlen()`.

Výše uvedená omezení jsou dnes běžně opravdu jediná vynucená, všechno ostatní funguje.[5] Čímž neříkáme, že celý zbytek Unicode v názvech souborů najdete, nebo dokonce že můžete obskurními znaky soubory beztrestně pojmenovávat.

Díky absenci NUL v názvu souboru máme jisté, že když za sebe naskládáme jména souborů oddělená znakem NUL, budeme je umět opět jednoznačně rozdělit. Toho využívají některé běžné utility, bohužel pomocí nestandardních přepínačů. Tuto jistotu nám norma nedává, pokud použijeme jako obvykle znak LF (`\n` v C, „konec řádku“).

Proto se z bílých znaků používá jen mezera, LF v názvu naštěstí není běžné, tedy můžeme být v klidu. Kdo takovou zvyklost poruší, následky nechť si nese sám. Zkuste si nějaký soubor s LF v názvu vyrobit a pohrát si s ním!

Soubory s diakritikou, interpunkcí a mezerami v názvech se opravdu dají potkat, takže by s nimi vaše skripty měly umět pracovat. Pomohou vám k tomu znalosti z prvního dílu: escapování, uvozovkování a ve vzácném případě minusu na začátku názvu souboru parametr `--`.

[4] `http://pubs.opengroup.org/onlinepubs/9699919799/`

[5] Nepočítáme-li obskurní starožitný FAT, přežívající na některých flashdiscích.

Soubory systému a programů obvykle dodržují ještě mnohem striktnější omezení, než je to popsané výš. Volí jména souborů, která

1. obsahují jen písmena velké a malé anglické abecedy, číslice, podtržítko, tečku a minus (`[A-Za-z_.-]`), a navíc
2. nezačínají minusem (aby se nepletla s přepínači).

Soubory s tečkou na začátku jsou skryté před wildcardy a `ls`. Druhé běžné použití tečky je oddělení přípony od zbytku jména souboru, jinak se tímto znakem šetří.

Shell se hodí na jednoduché skripty spořící čas, ne na psaní neprůstřelných programů (to v něm ani dobře nejde). Pokud v něm budete pracovat víc, dojdete ke kompromisu mezi omezováním se a nutností uvozovkovat při běžné práci moc často. Vynecháte nejspíš běžné oddělovače (LF, mezeru, tabulátor, dvojtečku, středník a čárku), speciální znaky shellu (`$‘"’#!?*[]{}();|\<>&`), a dáte si pozor na minus na začátku názvu. Možná si navíc budete šetřit čas při psaní vynecháním diakritiky a velkých písmen, ačkoliv to zas tolik nepomůže. Důležité je hlavně trefit začátek slova a zbytek už zařídí doplňování tabulátorem.

Části cesty, přípony

Přípony. Ve Windows se podle nich točí svět, binárku bez přípony `.exe` spustíte těžko. UNIX se s nimi vypořádal jinak – přípony považuje za informaci pro uživatele, sám se řídí prvními několika bajty souboru, kde obvykle je „magic number“. Podle něj umí formát určit třeba i utilita `file`:

```
hroch@ksp:~$ file /etc/passwd
/etc/passwd: UTF-8 Unicode text
hroch@ksp:~$ file /usr/bin/vimtutor
/usr/bin/vimtutor: POSIX shell script text exe-
cutable
```

U binárních programů toho `file` umí zjistit hodně. Kdybychom ho neměli, museli bychom binárku prohlížet nějak ručně a třeba si všimnout toho, že na začátku jsou znaky DEL, E, L a F, přičemž ELF je jméno formátu spustitelných souborů pro UNIX.

```
hroch@ksp:~$ od -c -Ax -tx1 -N10 /bin/sh
000000 177   E   L   F 002 001 001  \0  \0  \0
        7f  45  4c  46  02  01  01  00  00  00
00000a
```

Zkuste si sami pomocí `od` nebo rozšířeného, ale nestandardního `hd` prohlédnout nějaký obrázek PNG, dokument PDF, ... Pokud chcete být drsní, vynechte u `od` přepínač `-c` a ve vedlejším terminálu si otevřete `man ascii`. ;-)

Příponu tedy běžně není potřeba od zbytku jména souboru oddělovat, obzvlášť u textových a spustitelných souborů často ani žádná přípona použita není. Zato bychom někde ve skriptu mohli chtít získat zvlášť jméno souboru a zbytek cesty. Poslouží nám příkazy `basename` a `dirname`:

```
hroch@ksp:~$ dirname /usr/bin/less
/usr/bin
hroch@ksp:~$ basename /usr/bin/less
less
```

Jejich opakovaným použitím můžete rozebrat cestu na jednotlivé komponenty.

Typy souborů

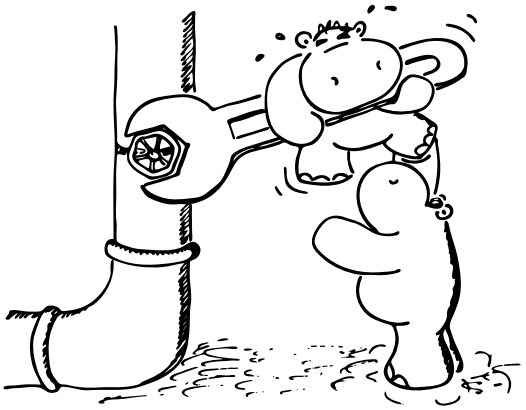
Když už jsme nakousli soubory v UNIXu, podívejme se na ně blíž. UNIXová filosofie se totiž drží zásady, že skoro všechno je soubor. Běžné textové soubory nebo soubory s binárními daty (fotky, videa, ...) nás asi nepřekvapí. Ale UNIX jako soubory reprezentuje i takové věci jako vstup z klávesnice (systém odtud čte po znacích) nebo výstup do zvukové karty. Podstatné je, jak se který soubor chová při používání.

Soubor je na disku typicky reprezentován jedním *inodem*, každý z nich má v rámci souborového systému svoje unikátní číslo. Uvnitř mezi dalšími metadaty systém ukládá informace o právech a vlastnících souboru, jeho typ a velikost, počítadlo odkazů (viz dále) a hlavně odkazy na jednotlivé datové bloky se samotným obsahem.

Je důležité, že jméno souboru si nepamatuje sám soubor, ale pamatuje si ho nadřazený adresář (což je jen speciální typ souboru). Inode reprezentující adresář obsahuje ve své datové části jména a příslušná čísla inodů pro všechny v něm obsažené soubory.

V předchozích dílech jsme se věnovali jen dvěma typům souborů: běžným souborům a adresářům. Dalšími jsou již zmiňovaná vstupní a výstupní zařízení, která sídlí hlavně v adresáři `/dev` a dělí se na bloková (disk) a znaková (terminál). V neposlední řadě se hodí vědět o rourách.

Zatím jsme potkali jen roury anonymní, které shell natahuje mezi dvěma příbuznými (společně spouštěnými) procesy: `ls -l /bin | head`. Mezi nepříbuznými procesy (spouštěnými třeba i dvěma různými uživateli) anonymní rouru natáhnout nejde, ale oba mohou znát cestu k pojmenované rouře. Jeden z ní čte, druhý do ní zapisuje a jméno potřebují jenom k tomu, aby ji mohli otevřít. A kde pojmenovanou rouru sebereme? Vytvoří ji příkaz `mkfifo`.



V shellu je bezesporu nejpoužívanějším speciálním souborem `/dev/null` neboli „černá díra“. Je to ideální místo, kam zahazovat věci, které na nic nepotřebujeme. Můžeme ho využít, pokud nás nezajímá standardní výstup příkazu, a jde nám jenom o jím vyrobený soubor nebo jeho návratovou hodnotu. Pokud bude příkaz chtít vstup a my mu budeme chtít dát prázdný soubor, `/dev/null` je také vhodné použít.

```
echo Windows > /dev/null
cat /dev/null
```

Podobně se chová `/dev/zero`, až na to, že při čtení dodává nekonečně dlouhou posloupnost znaků NUL. Soubor délky 1024 bajtů vytvoří `head -c 1024 /dev/zero > soubor`. Pěkné hraní je i se soubory `/dev/random` a `/dev/urandom`.[6]

[6] `http://cs.wikipedia.org/wiki//dev/random`

Někdy přesměrujeme našemu skriptu výstup do souboru, a přesto bychom chtěli vybrané informace o jeho provádění vidět na terminálu. Může nám je ukazovat tak, že je bude zapisovat do `/dev/tty`, který reprezentuje aktuální terminál. Podobně když máme přesměrovaný vstup a chceme z terminálu (z klávesnice) číst.

```
(       echo 'Potvrdte "ano":' > /dev/tty
        read odpoved < /dev/tty
        [ "$odpoved" = ano ] || exit 1
        cat
) < soubor1 > soubor2
```

Málem bychom zapomněli... Ve výstupu `ls -l` poznáte jednotlivé typy souborů podle prvního znaku na řádku, ještě před právy. Kdyby nějaké písmeno nebylo jasné, v manuálu `ls` jsou zkratky vysvětleny. Minus je běžný soubor.

Odkazy v souborovém systému

Občas se nám hodí pořídit si zkratku, rychlý odkaz na nějaký soubor či adresář. Na takové věci se v UNIXu využívají *hardlinky* (linky) a *symlinky*.

Hardlink je odkaz, který „natvrdo“ ukazuje na stejný inode jako odkazovaný soubor. V adresáři je pod jménem linku uloženo přímo číslo inode, na který odkazuje. Při vytvoření linku se v daném inode zvedne počítadlo odkazů, při jeho smazání se zase sníží, a když dojde na nulu, odstraní se i samotná data. Výjimkou jsou otevřené soubory – dokud nějaký proces má soubor otevřený, souborový systém přepsání jeho dat neumožní. Tak může mít proces bezpečně otevřený i soubor, který už nemá žádné jméno.

Po vytvoření hardlinku jsou původní a nový soubor od sebe nerozeznatelné, ale nese to s sebou několik omezení. Předně musí všechny linky sídlit na stejném diskovém oddílu, kde jsou uložena samotná data, a také nejde vytvořit hardlink na adresář, jen na soubor. (Kdo by se vyznal v souborovém systému, kde může být adresář umístěný sám v sobě?) Při přesunutí (přejmenování) nebo vymazání původního souboru bude hardlink ukazovat stále na původní verzi. Protože mohou být hardlinky někdy zrádné, doporučujeme pro běžnou práci používat spíše symlinky.

Symlink, neboli *symbolic link*, je jen jednoduchý zástupce, který ukazuje na nějakou cestu v souborovém systému (na libovolném připojeném oddílu a na libovolný soubor či adresář). Ve skutečnosti vypadá skoro jako malý textový soubor, který má v sobě zapsanou absolutní (začínající lomítkem) či relativní cestu ke svému cíli.

Při vymazání nebo přesunutí původního souboru přestane symlink fungovat, pokud neobsahuje relativní cestu a není přesunut spolu s cílem. Ukazuje totiž na cestu, ne na konkrétní soubor. Pokud cílový soubor smažeme a nahradíme novým, bude symlink ukazovat na tento nový soubor.

K symlinkům je třeba dodat dvě varování, abyste se nedivili a nedomýšleli si něco, co není pravda.

1. Předně, symlink a zástupce z Windows se chovají zásadně jinak. Zástupci jsou běžné soubory, podobné `.desktop` souborům na Linuxu, jen jsou binární.
2. Druhé varování se týká výstupu `ls -s`. Na mnoha systémech bude u symlinků ukazovat nulový počet zabraných bloků. Pokud je totiž symlink dostatečně malý, není problém ho celý uložit přímo uvnitř jeho inode. Jak úsporné! :-) Říká se tomu inlining a nové souborové systémy (třeba ext4) umí toto chování zapnout i u jiných typů souborů.

Hardlinky a symlinky se vytvářejí příkazem `ln`. Bez přepínače vyrobí hardlink, s přepínačem `-s` symlink:

```
ln cesta/k/souboru novy_hardlink
ln -s cesta/k/souboru relativni_symlink
ln -s /cesta/k/souboru absolutni_symlink
```

Cíl symlinku běžně vyčtete z `ls -l`:

```
lrwxrwxrwx 1 hroch users 3 datum zdroj -> cil
```

Ve skriptech se nám bude hodit spíš příkaz `readlink`, který vypíše jenom cíl odkazu. Také má užitečný přepínač `-f`, se kterým vypisuje absolutní cestu získanou z argumentu nahrazováním všech symlinků na cestě skutečnými cestami, kam ukazují. Příkazu `readlink -f` tedy má smysl dávat nejenom symlinky, ale i běžné soubory.

Pokud budete uvnitř adresáře, do kterého jste se dostali přes symlink, můžete si plnou cestu k němu nechat vypsat pomocí `pwd -P`, přepínač `-P` zajistí rozepsání všech symlinků v cestě. Chová se stejně jako `readlink -f .` (všimněte si použití tečky).

Úkol 2 [2b]: Pusťte si `ls -ld` na nějaký adresář a všimněte si, kolik má hardlinků. Odkud vedou?

Úkol 3 [2b]: Vysvětlete, jak se v souvislosti s hardlinky liší

```
cat </dev/null >soubor
```

a

```
rm soubor
cat </dev/null >soubor
```

Uživatelé, skupiny, vlastníci a práva

Teď už máme představu o tom, co všechno v souborovém systému můžeme najít. Zatím jsme v něm ale beznadějně sami. Sotva přijdou další uživatelé, potřebujeme před nimi občas něco schovat, nenechat je měnit naše soubory, ... Tedy budeme potřebovat systém oprávnění, který jsme v minulém dílu načali právem ke spuštění. Vůbec jsme ale nemluvili o tom, komu `chmod +x` právo ke spuštění přidělí.

UNIX je odpradávna víceuživatelský systém. Oddělené účty uživatelů a dodržování principu minimálních oprávnění mu zajišťují slušnou míru bezpečnosti. Soubor `/etc/passwd` obsahující databázi uživatelů už jste potkali, jeho dokumentaci najdete na Linuxu v `man 5 passwd`, jinde musíte hledat v jiné sekci. Možná vás zajímalo, kde se ukládají hesla – pak vězte, že v dřevních dobách to bylo opravdu tam, ale moderní UNIXy mají na citlivé údaje oddělenou databázi jinde v adresáři `/etc`. Hesla se navíc ukládají zakódovaná. Na Linuxu vám víc prozradí `man shadow`.

Vžijte se na chvíli do role administrátora školního UNIXového serveru. Máte na něm účty desítek, možná i stovek uživatelů a chcete jim přidělit nějaká oprávnění (jaká, k tomu se dostaneme později). Chtělo by se vám u každého zvlášť přemýšlet, co smí a nesmí? Asi byste si všimli, že studentské účty mají mít obecně jiná oprávnění než účty učitelů, že dokumenty k maturitnímu plesu jedné třídy nemá co upravovat nikdo z jiných tříd, tedy pokud nemají dvě třídy ples společný, atd. Přáli byste si, abyste mohli systému o třídách a učitelích říct a on vám dovolil oprávnění přidělovat hromadně.

Přání se vám splnilo, řešení má jméno *skupiny*. Jsou to pojmenované množiny uživatelů, je jim možné nastavovat společná práva, jejich databáze sídlí v `/etc/group` a více si

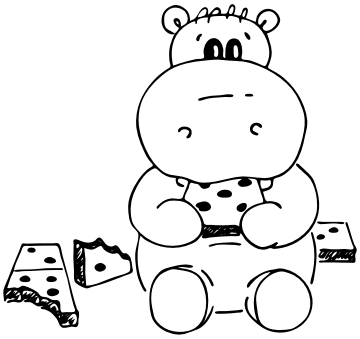
o ní můžete přečíst v `man group`. V `/etc/passwd` má každý uživatel navíc skupinu, pod kterou se přihlašuje, neboli login group. Do ní automaticky patří.

Co znamená, že se uživatel pod nějakou skupinou přihlašuje? Inu, tato skupina je skupinovým vlastníkem jím vytvářených souborů. Co to pro ně znamená?

Když se podíváte na výstup `ls -l`, uvidíte řádky jako:

```
-rwxr-x--x 1 hroch users 42 8. pro 08:00 soubor
```

Uživatelským vlastníkem souboru `soubor` je `hroch`, skupinovým je `users`. Pokud se řekne jenom vlastník, myslí se uživatelský vlastník. Nenechte se zmást, až někde uvidíte `hroch` v obou sloupcích – skupiny a uživatelé se mohou jmenovat stejně a na některých systémech se takto pro každého uživatele zakládá jeho vlastní login group.



Na uživatelského vlastníka (user, owner) se vztahuje jiné nastavení oprávnění než na toho skupinového (group, group owner), a na toho zase úplně jiné než na všechny zbývající uživatele (others, world) kromě uživatele `root`.

Root, nebo také superuživatel, je takovým místním bohem. Smí všechno. Jeho oprávnění je potřeba např. k zakládání nových účtů nebo ke změně vlastníka souboru. Změnit skupinu souboru na některou ze skupin, ve které je členem, může ale i vlastník souboru.

Tolik pravomocí s sebou nese i hodně zodpovědnosti a moc velký průšvih, když se k účtu roota dostane někdo nepovolaný. Přestože se hledají alternativní cesty, jak si poradit bez superuživatele, root s námi ještě nějaký čas bude.

Pomocí příkazu `su` můžeme pouštět příkazy rootovým jménem, nebo i jménem jiného uživatele, pokud známe jeho heslo. Pokud si pustíme rootovský shell, bude mít v promptu místo dolaru mřížku (`#`). Alternativou `su` je příkaz `sudo`, který nám dovoluje pouštět příkazy pod jiným vlastníkem po zadání k tomuto účelu speciálně nastaveného hesla.

Pokud root zrovna nejste a snažíte se přistupovat k souboru, jaká pro vás platí oprávnění? Ve výše uvedeném příkladu

- `rwx` pokud jste `hroch`, jinak
- `r-x` pokud jste ve skupině `users`,
- `--x` ve všech ostatních případech.

Tento *symbolický zápis práv* není až tak spletitý, jak na první pohled vypadá. První pozice v každé trojici je Read (čtení), druhá Write (zápis), třetí eXecute (spuštění). Minus znamená, že právo není přiděleno. Při čtení práv ve výstupu `ls -l` nezapomeňte, že těsně před nimi je typ souboru. Připomínáme, že minus znamená běžný soubor.

Často se používá ještě druhý způsob zápisu práv, *numerický*, nebo také *oktalový*, tedy pomocí osmičkové číselné soustavy. Práva vlastníka, skupinového vlastníka a ostatních v něm jsou reprezentovaná třemi osmičkovými ciframi. Jedna osmičková cifra má tři bity. 4 = `r`, 2 = `w`, 1 = `x`. Skládání se dělá bitovým součtem (or). 000 jsou tedy doslova nulová práva, 777 všechno všem, 700 všechno vlastníkovi, 755 všechno vlastníkovi a ostatním jen čtení a spouštění, 640 čtení a psaní vlastníkovi, čtení skupině a nic ostatním.

U běžných souborů jsou práva téměř intuitivní. Drobné zádrhele mohou být, že interpretované programy potřebují kromě hashbangu (např. `#!/bin/bash`, vizte předchozí díl seriálu) a práva ke spuštění i právo ke čtení. Interpret se k textu programu holt nějak musí dostat. U binárek problém není. Naopak binárky nespustíte bez práva ke spuštění, kdežto u skriptu v Bashi si snadno poradíte zavoláním `bash skript.sh`.

Větší potíže bývají s významem práva k zápisu. My jsme si ale už vysvětlili, jak fungují linky na soubor a že data souborů a záznamy v adresářích jsou na sobě do značné míry nezávislé, takže to máme snazší. Právo k zápisu mluví u souboru o zápisu do jeho dat. K přejmenování nebo smazání se vztahuje právo zápisu do adresáře. Přesto utilita `rm` váhá, když má mazat soubor, který nemá právo k zápisu. Je potřeba smazání ručně potvrdit nebo předem přidat přepínač `-f` (force – „Na nic se neptej a maž!“).

Práva souboru se ukládají s jeho inode – pokud je změníme přes jeden link, projeví se změna i ve všech ostatních. Zde se hodí třetí varování k symlinkům: práva u nich nemají smysl. Symlink se často chová transparentně a rozhodující jsou práva cílového souboru. Nenechte se zmást tím, že `ls` mu přisuzuje práva 777.

U adresářů jsme už právo k zápisu vyřídili o dva odstavce výš. Právo ke čtení adresáře potřebují ke své správné funkci wildcardy, `ls`, doplňování tabulátorem a vůbec cokoliv, co potřebuje vidět obsah adresáře. Právu `x` se u adresářů říká search (prohledávání). Dovoluje nám se znalostí jména souboru přistoupit k jeho obsahu, tedy při hodně nízkoúrovňovém pohledu vlastně přečíst z adresáře číslo inode souboru, pokud dodáme jeho jméno.

Bez práva ke čtení, ale s právem k prohledávání můžeme jaksi „poslepu“ pracovat s obsaženými soubory známých jmen, a v závislosti na právu k zápisu je můžeme i vytvářet a mazat. S právem ke čtení, ale bez práva k prohledávání můžeme obsah adresáře jenom vypsat, a to ještě mizerně. U každé položky uvidíme v `ls -l` jen jméno (a na některých souborových systémech i typ), ostatní metadata jsou uvnitř inode a místo nich se zobrazí jen otazníky. Ani nám pak nebude vadit, že bez práva k prohledávání nemůžeme nastavit adresář jako svůj pracovní (`cd adresar`).

Když už víme, k čemu práva jsou, jak je změnit? Příkaz `chmod` už jsme párkrát zmínili. Teď už navíc budete rozumět jeho manuálu, kde se můžete dočíst pikantní podrobnosti o právech včetně těch, které se sem nevešly.

Příkaz `chmod` může pracovat s právy zadanými oktalově i symbolicky. Se symbolickými právy umí nastavovat i jednotlivá práva při zachování ostatních. Kombinací je docela hodně, uveďme tedy jen příklad: `chmod u+x,go-rw soubor` přidá uživateli právo ke spuštění a skupině i ostatním sebere práva ke čtení a k zápisu, ostatní práva nechá netknutá.

Vlastníka umí měnit příkaz `chown`, skupinového `chgrp`. Příkaz `chown` navíc umí měnit skupinového i uživatelského vlastníka najednou, stačí je zadat oddělené dvojtečkou:

```
ksp:~# ls -l s; chown palec:ksp s; ls -l s
-rw-r--r-- 1 hroch users 1 8. pro 08:00 s
-rw-r--r-- 1 palec ksp   1 8. pro 08:00 s
```

Pokud u `chown` vynecháme uživatele před dvojtečkou, změní jen skupinového vlastníka. Příkazy `chmod`, `chown` i `chgrp` mají přepínač `-R`, který je nechá změnu aplikovat na daný adresář a rekurzivně na všechen jeho obsah.

Když už soubory existují, poradit si s nimi umíme. Teď si povíme, s jakými právy a vlastníky se zakládají.

Stejně jako soubor, i každý proces má vlastníka. Vlastník procesu se použije jako vlastník souborů tímto procesem vytvářených. Se skupinovým vlastníkem je to složitější, ten se občas může dědit od adresáře. Přesný algoritmus výběru závisí na systému a jeho nastavení. Náš shell má jako vlastníka nás a jako skupinového vlastníka naši login group. Procesy, které spouští, tyto vlastníky zdědí. Všechny „běží pod námi“.

Výše zmíněné příkazy `su` a `sudo` umožňují ovlivňovat vlastníka, o skupinového vlastníka se postará `newgrp`. Informace o aktuálním uživateli, skupině a ostatních skupinách, ve kterých uživatel je, poskytuje příkaz `id`.

Výchozí práva běžných souborů jsou 666, výchozí práva adresářů 777. Při vytváření se ale aplikuje maska, která zruší v přidělovaných právech ty bity, které jsou v ní nastavené na jedničku. Maska je stejně jako vlastník a skupina vlastností procesu a stejně jako oni se dědí. U shellu ji můžeme zjistit příkazem `umask` bez parametrů a nastavit stejným příkazem, kterému předáme novou masku jako argument. Typické nastavení je `umask 002` („nedávej `w` celému světu“) nebo `umask 022` („`w` nech jen vlastníkovi“).

Úkol 4 [3b]: Jak root zařídí, aby domácí adresář uživatele hroch nebyl přístupný ostatním a aby na tom hroch nemohl nic změnit? Dodejme, že je dobrým zvykem, aby domácí adresář patřil příslušnému uživateli.

Řídicí struktury a proměnné – podruhé na návštěvě

Ve druhém dílu seriálu jsme se potkali se základními řídicími strukturami. Pokud si nepamatujete použití podmínky `if` nebo dvou základních cyklů níže, připomeňte si je.

```
if cmd; then ...; else ...; fi
while cmd; do ...; done
for i in 1 2 3 4 5; do ...; done
```

Dnes k těmto základům přidáme mocnější zbraně. Nejdříve si pořídíme na hraní nějký nekonečný `while` cyklus, tedy cyklus, jehož podmínka bude vždy splněná. Abychom nemuseli psát podmínku stylu „nula je menší než jedna“, nabízí nám shell příkazy `true` a `false`.

Ano, nepřepsali jsme se, nejsou to konstanty nabývající hodnoty pravda a nepravda jako ve většině programovacích jazyků. V shellu se jedná o samostatné příkazy, které doslova dělají nic, a to buď úspěšně, nebo neúspěšně. Podívejte se na jejich manuálové stránky.

Náš nekonečný cyklus, ve kterém si třeba budeme k proměnné `X` na konec připisovat `D` a to celé vypisovat, může vypadat takto:

```
X=
while true; do
        X=${X}D
        echo $X
done
```

Všimněte si složených závorek. Dovolíme si zde malé odbočení k proměnným. Konstrukce `${X}` funguje stejně jako `$X`, jen ji shell i v tomto případě správně pozná. Jméno proměnné smí obsahovat písmena anglické abecedy, číslice (kromě prvního znaku) a podtržítko. Shell čte tyto znaky za dolarem, dokud se nezasekne o něco jiného, a teprve pak hledá, jestli proměnnou zná. Kdybychom jako přiřazení použili `X=$XD`, shell by neúspěšně hledal proměnnou `XD`, tedy by se do `X` přiřazoval prázdný řetězec.

Mohli byste namítat, že stejně dobře a elegantněji bychom mohli přidávat místo na konec na začátek pomocí `X=D$X` a složeným závorkám se vyhnout. Máte pravdu, tady to jde. Jenže ne vždy je možné se z průšvihu vylhat. Ještě můžeme psát `X=$X""D`, což je opravdu divné, a kdybychom už uvozovky používali, musíme víc přemýšlet. Za chvíli bychom ještě byli rádi za složené závorky.

Cyklus, který jsme vyrobili, je nám zatím docela nanic, protože běží donekonečna. Hodilo by se nám umět ho ve vhodnou chvíli zastavit, k tomu slouží příkaz `break`. Chová se podobně jako v jiných programovacích jazycích, tedy ukončí provádění celého cyklu. Pokud ho spojíme se šikovnou podmínkou, můžeme cyklus zastavit ve chvíli, kdy délka generovaného řetězce písmen `D` překročí deset.

```
X=
while true; do
        X=D$X
        delka=‘echo -n $X | wc -c‘
        if [ $delka -gt 10 ]; then break; fi
        echo $X
done
```

Všimli jste si, jak je zjišťování délky proměnné neohrabané? Ještě si musí člověk dávat pozor, jestli náhodou nepočítá i znak konce řádku… Taková běžná operace, to přeci musí jít lépe! A taky že jo: podobně jako je `${X}` expandováno na obsah proměnné `X`, tak je `${#X}` expandováno na její délku.

Společně s příkazem `break` jde ruku v ruce příkaz `continue`, který přeskočí všechny za ním následující příkazy, až před test podmínky další iterace cyklu. Vynechání řetězců kratších pěti znaků se sice dá udělat i lépe, přesto použijme příkaz `continue`:

```
X=
while true; do
        X=D$X
        delka=${#X}
        if [ $delka -lt 5 ]; then continue; fi
        if [ $delka -gt 10 ]; then break; fi
        echo $X
done
```

Až budete pracovat se zanořenými cykly a budete chtít příkazem `break` nebo `continue` ovlivnit jiný než nejvnitřnější z nich, vzpomeňte si, že oba příkazy mají nepovinný parametr. Zbytek už najdete. Pokud jste zvyklí na C, příkaz goto byste hledali marně.

Jako třetí mezi podobnými přichází na řadu příkaz `exit`, který najde použití hlavně ve skriptech. Jeho zavolání ukončí shell (je tedy silnější než `break`), a když se mu předá číslo (třeba `exit 42`), nastaví ho jako *návratovou hodnotu*. S ní jsme se potkali ve druhém dílu. Ve zkratce nula znamená úspěšné ukončení, cokoliv nenulového neúspěšné.

Návratová hodnota je na Linuxu 8-bitová (tedy může nabývat hodnoty 0–255), na většině jiných UNIXových systémů je však jen 7-bitová (0–127). Bash sám o sobě využívá hodnoty 126 a 127 pro své vlastní potřeby, tedy pokud chcete psát univerzální programy, omezte se jen na hodnoty 0–125.

Pokud chcete zjistit, s jakou návratovou hodnotou skončil poslední provedený příkaz, můžete k tomu využít speciální proměnnou `$?`. Zkuste si třeba zavolat příkazy `true`, `false` nebo `(exit 42)` a hned po nich `echo $?`. Pokud `exit` žádnou návratovou hodnotu nedostane jako argument, použije právě hodnotu této proměnné.

Nacyklili jsme se dost, podívejme se ještě na složitější podmínky. Poprvé potkáváme `elif`. Funguje analogicky ke stejnojmenné konstrukci v Pythonu, `elsif` v Perlu, `elseif` v PHP a prosté kombinaci `else if` v C, Javě a Pascalu. Pokud na ni při vyhodnocování dojde řada, spustí podmínku[7] a kontroluje, jestli uspěla. Pokud ano, nechá spustit kód za svým `then`, jinak pokračuje další větví podmínky (další `elif` a jako poslední `else`).

V košatějších podmínkách se může hodit nějakou větev mít, ale nic v ní nedělat. Když hned za `then` napíšete středník, shell si postěžuje na syntaktickou chybu. Mohli byste použít příkaz, který nic nedělá, třeba `true`, ale víc se hodí funkčně shodná dvojtečka (`:`). Její primární účel je naznačení, že „tady nic není“, pro svou krátkost se ovšem zneužívá i na místech, kam by se víc hodilo `true`.

```
X=$(($RANDOM % 100))
if [ "$X" -lt 10 ]; then
        echo malé
elif [ "$X" -le 42 ]; then
        : nevím, něco mezi, mlčím
else
        echo velké
fi
```

Zde je `$RANDOM` speciální proměnná Bashe, která obsahuje při každé expanzi nové náhodné číslo mezi 0 a 32767. Tento úryvek kódu tedy vygeneruje náhodné číslo mezi 0 a 99 a rozhodne, jestli je malé, nebo velké. Pokud je vygenerované číslo mezi 10 a 42, neumí se rozhodnout a mlčí.

Konstrukce `$((...))` je *aritmetická expanze*. Shell výraz uvnitř vyhodnotí a nahradí ji jeho hodnotou, přitom se k výrazu chová, jako by byl ve dvojitých uvozovkách (expanduje proměnné, vyhodnotí zpětné apostrofy a výrazy `$(...)`). Pokud najde jméno proměnné (bez dolaru), přečte si její hodnotu. POSIX vyžaduje, aby se v proměnných hledaly aspoň konstanty s volitelným znaménkem, Bash jde ještě dál. Pokud obsah proměnné je možné interpretovat jako výraz (třeba i jen jako jméno jiné proměnné), zkusí ho vyhodnotit. Teprve když se mu to nepovede, vyhlásí chybu.

```
X=21+21
Y=X
Z=Y
echo $((Z))
echo $(($Z))  # funguje, jen zbytečně složitěji
```

Výrazy jsou jinak přejaté z jazyka C. Podporovány jsou desítkové, oktalové a hexadecimální konstanty, většina aritmetických, bitových a logických operátorů, závorky, podmínkový operátor (dvojice `?` a `:`) a operátory přiřazení. Bash podporuje i operátory `++` a `--`. Vyhodnocuje se ve znaménkovém celočíselném typu – norma vyžaduje `signed long` nebo větší. Proběhlá přiřazení do proměnných shell uvidí i po dokončení expanze.

Alternativou k shellové aritmetice jsou příkaz `expr`, který podle nás nemá žádné výhody, a příkaz `bc`, který má vlastní jazyk a neomezenou přesnost.

Funkce

V shellu už umíme kdeco z toho, co umí běžné programovací jazyky. Proměnné, příkazy, řídicí konstrukce, aritmetiku, vstup a výstup, ... O jednom důležitém konceptu z programovacích jazyků ale dosud nepadlo slovo. O funkcích.

V běžných jazycích jsou funkce posloupností příkazů, která má nějaké (formální) parametry. Při volání se funkci předají argumenty (skutečné parametry), které se dosadí do formálních parametrů, a příkazy se spustí. V shellu je situace úplně stejná.

Mnoho ze znalostí o skriptech, které jste si přinesli z předchozího dílu, platí i pro funkce. Funkce také dostává poziční parametry, také má proměnné `$#`, `$@`, `$1` (až `$9`), má návratovou hodnotu a volání funkce i skriptu vypadají jako každý jiný příkaz (žádné závorky kolem argumentů!).

Na rozdíl od skriptu se pro funkci nespouští zvláštní shell, její příkazy běží ve stejném procesu, který ji volá. Má tedy stejný obsah proměnné `$0`, může využívat (a trvale měnit!) i neexportované proměnné a při zavolání `exit` neskončí jen ona sama, ale i celý shell. Pro nastavení návratové hodnoty používá příkaz `return`.

Funkce se tedy chová stejně jako skript vložený příkazem `.` (bashovsky také `source`), jen se jinak definuje a nemusí být ve vlastním souboru.

Ukažme si pro ilustraci definici funkce a její volání.

```
# definice
echo_t() {
        test -t 1 || echo "$@" > /dev/tty
        echo "$@"
}
# volání
echo_t hroch > soubor
```

Tato funkce se jmenuje `echo_t`, všechny své argumenty vytiskne na svůj standardní výstup, a pokud standardní výstup nejde na terminál (je přesměrovaný a ne zrovna do terminálu), argumenty vytiskne i přímo na terminál.

Definice funkce začíná jejím jménem, následovaným kulatými závorkami. Pro jméno funkce platí stejná pravidla jako pro jméno proměnné. Jmenné prostory funkcí a proměnných jsou oddělené – můžeme mít stejně pojmenovanou funkci i proměnnou, obě budou fungovat správně. Před jménem funkce Bash dovoluje ještě (zbytečné) klíčové slovo `function`.

Za jménem a kulatými závorkami následuje téměř obyčejný složený příkaz, jak ho znáte z minula. Zvláštní je v tom, že místo aby se v něm hned expandovaly proměnné a vůbec dělo všechno, co shell se svým vstupem provádí, shell si ho zapamatuje beze změny a expanze provádí až při volání.

[7] Nezapomínejte, že podmínka je příkaz!

Za zmínku stojí, že za složeným příkazem mohou být přesměrování vstupů a výstupů, která se stanou součástí definice funkce; snadno tedy můžete napsat třeba hloupou funkci pro záznam zpráv opatřených časem pořízení záznamu:

```
log() {
        date
        echo "$@"
        echo
} >> zaznamy_chyb
```

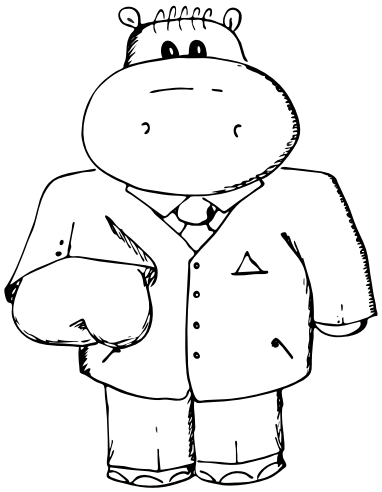
Co kdybychom ale chtěli každý argument zalogovat jako samostatný záznam a na každý záznam mít jeden řádek? Mohli bychom použít cyklus, který projde jednotlivé argumenty. Proč bychom se ale nudili už dobře známým, když si můžeme ukázat pěknou novou utilitu?

Úkol 5 [5b]: Napište funkci, která dostane jako svůj parametr jméno souboru se zdrojovým kódem, načte a předzpracuje jej pro kompilátor a výsledek vypíše na standardní výstup.

Zdrojový kód obsahuje jednotlivé textové tokeny oddělené právě jednou mezerou. Komentáře jsou uvozené tokenem `COMMENT` a platí do konce řádku. Program je ukončený buď koncem souboru, nebo tokenem `BYE`.

Úkolem funkce je opsat zdrojový kód až do konce programu s vynecháním komentářů. Na výstupu by se neměly objevit ani žádné prázdné řádky.

Poznámka: Na tuto úlohu nepoužívejte příkazy `grep`, `sed` ani žádný jiný jazyk (AWK, Perl, ...).



Formátovaný výstup

Už jsme vám až příliš dlouho zamlčovali `printf`. Mnoho jazyků má funkci toho jména a ve všech má velmi podobný význam parametrů a stejný účel – pěkné a snadné formátování výstupu. Na rozdíl od `echo` implicitně neodřádkovává, takže často budete na konec jejího prvního argumentu muset psát `\n`. To je další rozdíl proti běžnému `echo`, umí céčkové escape-sekvence jako `\n` pro konec řádku nebo `\t` pro tabulátor (`echo` s přepínačem `-e` nebo POSIXové `echo` escape-sekvence také vyhodnocují).

První argument `printf` je šablona, do které dosazuje zbylé argumenty. Pokud je argumentů málo, domyslí si pár prázdných navíc. Pokud jich je moc, zopakuje šablonu. Toho využívá i výše slíbená vylepšená funkce `log`:

```
log() {
        d="$(date)"   # znak % neobsahuje
        printf "[$d] %s\n" "$@"
} >> zaznamy_chyb
```

V šabloně jsou formátovací direktivy, které poznáte podle znaku `%` na začátku. (Pokud procento potřebujete vypsat doslova, pište `%%`.) Za každou direktivu dosadí `printf` pozicí odpovídající argument. Běžně budete potřebovat `%s` pro string a `%d` pro desítkový zápis čísla. Mimo jiné `printf` umí i převádět čísla do šestnáctkové (`%x`) a osmičkové soustavy (`%o`). Direktivy mají kromě povinného typu (`s` pro string, ...) ještě dost nepovinných částí. Na jejich samostudium se dlouhé zimní večery náramně hodí. ;-)

Úkol 6 [2b]: Napište skript, který formátuje `/etc/passwd` do podoby tabulky.

Závěr

Tento díl byl delší než oba předchozí dohromady. Dotáhli jsme v něm ale do konce spoustu rozdělaných věcí a poskytli vám tak mnohem ucelenější pohled na UNIXový svět. Pokud bychom měli zrekapitulovat, co si z tohoto dílu máte odnést, mohl by takový seznam vypadat následovně:

- Nápověda: vyhledávání pomocí `apropos`, sekce nápovědy, `info`, norma POSIX
- Souborové systémy: `df [-Th]`, `du -h`, konvence na názvy souborů, (ne)potřebnost přípon, `file`
- Typy souborů: inodes; běžné soubory, adresáře a zařízení reprezentovaná soubory (`/dev/null`, `/dev/tty`, ...)
- Symlinky a hardlinky: `ln -s` a `ln`, `readlink`
- Uživatelé a práva: význam skupin a práv pro soubory a adresáře, `su` a `sudo`
- Řídící struktury: `true`, `false` a dvojtečka, `continue`, `break` a `exit`, vnořené cykly, expanze délky proměnné (`${#promenna}`), `$RANDOM` a aritmetická expanze
- Funkce v shellu: definice, volání, `return`, souvislost se skripty
- Formátovaný výstup: `printf`

V příštím díle se už konečně podíváme na slíbené utility pro práci s textem. Přesným obsahem se nechte příjemně překvapit. :-)

Tomáš „Palec“ Maleček

Recepty z programátorské kuchařky: Rozděl a panuj

Dnešní díl programátorské kuchařky se bude zabývat algoritmy založenými na metodě *Rozděl a panuj.* Slušelo by se začít tím, jaká je myšlenka této metody: Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase snadno složit výsledek původní velké úlohy. Přitom menší úlohy můžeme řešit opět týmž algoritmem (zavoláme si ho rekurzivně), leda by již byly tak maličké, že dokážeme odpovědět triviálně bez jakéhokoliv počítání. Zkrátka jak říkali staří římští císařové: Divide et impera. Uveďme si pro začátek jeden staronový příklad:

Quicksort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už jste si o něm mohli přečíst v kuchařce o třídění. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QS v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní. Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QS je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali v třídící kuchařce (hlavně proto, že se nám z ní pak budou snadno odvozovat další algoritmy) a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
pole = [1,2,8,42, 9, 17, -5, 20, 2]

# Přerovnej část pole od L do P - 1
def prerovnej(pole, L, P):
  pivot = pole[P - 1]

  # i je nejlevější nepřerovnaný prvek
  i = L
  # j je aktuální probíraný prvek
  for j in range(L, P - 1):
    if (pole[j] <= pivot):
      # Prohodíme tyto dva prvky
      (pole[i], pole[j]) = (pole[j], pole[i])
      i += 1

  # Dáme pivota na správné místo
  (pole[P-1], pole[i]) = (pole[i], pole[P-1])
  return i

def quicksort(pole, L, P):
  if (L >= P):
    return

  # Přerovnáme úsek...
  pivot = prerovnej(pole, L, P)
  # ... a zavoláme se rekurzivně na oba podúseky
  quicksort(pole, L, pivot)
  quicksort(pole, pivot + 1, P)
```

Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které to nastane pokaždé), takže dostaneme posloupnost délky N, rozdělíme ji na úseky délek $N-1$ a 1, načež pokračujeme s úsekem délky $N-1$, ten rozdělíme na $N-2$ a 1, atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $\mathcal{O}(N + (N-1) + (N-2) + \ldots + 1) = \mathcal{O}(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $\mathcal{O}(N \log N)$. To dokážeme snadno:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě dlouhé $(N-1)/2 \pm 1$); přerovnávání v obou částech dohromady trvá opět $\mathcal{O}(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\lceil \log_2 N \rceil$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\lceil \log_2 N \rceil$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $\mathcal{O}(N \log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: Zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak?
- *Spokojit se se „lžimediánem“:* Kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti „v prostřední polovině“ (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $\mathcal{O}(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1-1/4) \cdot N$, takže na k-té hladině budou úseky délek nejvýše $(1-1/4)^k \cdot N$, čili hladin bude maximálně $\log_{1-1/4} N = \mathcal{O}(\log N)$. Místo 1/4 by fungovala i libovolná jiná konstanta mezi nulou a jedničkou, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.
- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. (Také se tak často QS implementuje.)
- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností 1/2 to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme. Proto časová složitost takovéhoto randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $\mathcal{O}(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. To tak nemůžeme nechat, a proto rovnou zkusíme vyřešit obecnější problém: najít k-tý nejmenší prvek (medián dostáváme pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Má to však jeden háček. Pokud prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $\mathcal{O}(N \log N)$ – rychleji prostě třídit nelze, důkaz můžete najít například v třídící kuchařce.

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká QuickSelect). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné).

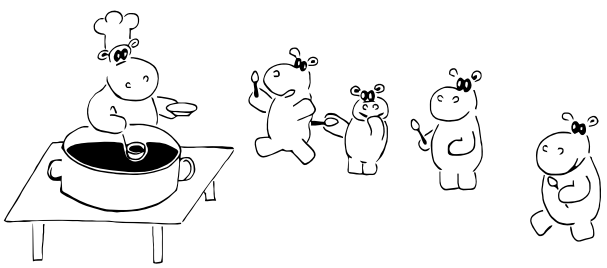
Pokud se pivot nalézá na k-té pozici, je to hledaný k-tý nejmenší prvek posloupnosti, protože právě $k-1$ prvků je menších. Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivota v posloupnosti větší než k, pak se hledaný prvek nalézá nalevo od pivota a postačí rekurzivně najít k-tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek v posloupnosti napravo od pivota. Mezi těmito prvky však nebudeme hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti.

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Pokud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost $\mathcal{O}(N + N/2 + N/4 + \ldots + 1) = \mathcal{O}(N)$. Pro lžimedián dostaneme rovněž lineární složitost a opět stejně jako u QS můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejný čas jako se lžimediánem.

Program bude velmi jednoduchý, využijeme-li přerovnávací proceduru od QS:

```
def kty(pole, k, L, P):
  pivot = prerovnej(pole, L, P)

  if (k == pivot):
    return pole[pivot]
  if (k < pivot):
    return kty(pole, k, L, pivot)
  else:
    return kty(pole, k, pivot + 1, P)
```



k-tý nejmenší podruhé, tentokrát lineárně a bez náhody

Existuje však algoritmus, který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na ďábelském triku: zvolit vhodného pivota (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Zařídíme to takto:

1. Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme a vrátíme k-tý prvek setříděné posloupnosti.
2. Rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
3. Spočítáme medián každé pětice. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku tříděním. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
4. Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a na ní začneme opět od prvního bodu).
5. Přerovnáme vstupní posloupnost po quicksortovsku a jako pivota použijeme prvek m. Po přerovnání je pivot, podobně jako v předchozím algoritmu, na $(z+1)$-ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.
6. Opět, podobně jako u předchozího algoritmu, pokud je $k = z+1$, pak je právě pivot m k-tým nejmenším prvkem posloupnosti. V případě, že tomu tak není a $k < z+1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy posloupnosti, v opačném případě, kdy $k > z+1$, budeme hledat $(k-z+1)$-ní nejmenší prvek mezi posledními $n-z-1$ prvky.

```
def prerovnej_podle(pole, L, P, podle):
  q = L
  while (pole[q] != podle):
    q += 1
  pole[q], pole[P - 1] = pole[P - 1], pole[q]
  return prerovnej(pole, L, P)

def kty(pole, k, L, P):
  pocet = P - L
  # Jednoduché případy
  if (pocet <= 1):
    return pole[L]
  if (pocet <= 5):
    quicksort(pole, L, P)
    return pole[k]

  # Rozdělení na pětice
  petic = (pocet + 4) // 5;
  mediany = [0] * petic
  for i in range(0, pocet, 5):
    if (i + 5 > pocet):
      break # Ignorujeme neúplnou pětici
    quicksort(pole, i, i + 5)
    mediany[i // 5] = pole[i + 2]

  # Nalezneme medián mediánů pětic
  median = kty(mediany, petic // 2, 0, petic)
  pivot = prerovnej_podle(pole, L, P, median)

  if (pivot == k):
    return median
  if (pivot < k):
    return kty(pole, k, L, pivot)
  else:
    return kty(pole, k, pivot + 1, P)
```

Zbývá dokázat, že tato dvojitá rekurze má slíbenou lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10 \cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

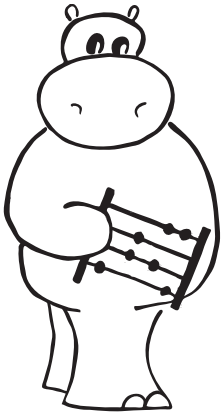
Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c > 0$. Pak už algoritmus pouze dvakrát rekurzivně volá sám sebe: nejprve pro $N/5$ mediánů pětic, pak pro $\leq 7/10 \cdot N$ prvků před/za pivotem. Pro celkovou časovou složitost $t(N)$ našeho algoritmu tedy platí:

$$t(N) \leq cN + t(N/5) + t(7/10 \cdot N).$$

Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhodneme, že výsledkem bude lineární funkce, tedy že $t(N) = dN$ pro nějaké $d > 0$. Dostaneme:

$$dN \leq (c + 1/5 \cdot d + 7/10 \cdot d) \cdot N.$$

To platí např. pro $d = 10c$, takže opravdu $t(N) = \mathcal{O}(N)$.



Násobení dlouhých čísel

Dalším pěkným příkladem na rozdělování a panování je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – teď zvolíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním“ algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob.

Libovolné $2N$-ciferné číslo mužeme zapsat jako $10^N A + B$, kde A a B jsou N-ciferná. Součin dvou takových čísel pak bude $(10^N A + B) \cdot (10^N C + D) = (10^{2N} AC + 10^N (AD + BC) + BD)$. Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N-ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N) = cN + 4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N) \approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Přijde trik. Místo čtyř násobení čísel poloviční délky nám budou stačit jen tři: spočteme AC, BD a $(A+B)\cdot(C+D) = AC+AD+BC+BD$, přičemž pokud od posledního součinu odečteme AC a BD, dostaneme přesně $AD + BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N) = c'N + 3t(N/2)$. (Konstanta c' je o něco větší než c, protože přibylo sčítání a odčítání, ale stále je to konstanta. My si ovšem zvolíme jednotku času tak, aby bylo $c' = 1$, a ušetříme si tak spoustu psaní.)

Jak naši rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned} t(N) &= N + 3(N/2 + 3t(N/4)) = \\ &= N + 3/2 \cdot N + 9t(N/4) = \\ &= N + 3/2 \cdot N + 9/4 \cdot N + 27t(N/8) = \ldots = \\ &= N + 3/2 \cdot N + \ldots + 3^{k-1}/2^{k-1} \cdot N + 3^k t(N/2^k). \end{aligned}$$

Pokud zvolíme $k = \log_2 N$, vyjde $N/2^k = 1$, čili $t(N/2^k) = t(1) = d$, kde d je nějaká konstanta. To znamená, že:

$$t(N) = N \cdot (1 + 3/2 + 9/4 + \ldots + (3/2)^{k-1}) + 3^k d.$$

Výraz v závorce je součet prvních k členů geometrické řady s kvocientem $3/2$, čili $((3/2)^k - 1)/(3/2 - 1) = \mathcal{O}((3/2)^k)$. Tato funkce však roste pomaleji než zbylý člen $3^k d$, takže ji klidně můžeme zanedbat a zabývat se pouze oním posledním členem:

$$3^k = 2^{k \log_2 3} = 2^{\log_2 n \cdot \log_2 3} = (2^{\log_2 n})^{\log_2 3} = n^{\log_2 3} \approx n^{1.58}$$

Konstanta d se nám „schová do $\mathcal{O}$-čka“, takže algoritmus má časovou složitost přibližně $\mathcal{O}(n^{1.58})$. Existují i rychlejší algoritmy se složitostí až $\mathcal{O}(n \log n)$, ale ty jsou mnohem ďábelštější a pro malá n se to sotva vyplatí.

Program si pro dnešek odpustíme, šetřímeť naše lesy.

Poznámky na ubrousku aneb Rozmyslete si

- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečny“ na délku, která je mocninou deseti?
- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?
- Ještě jednou QS: Představte si, že budujete binární vyhledávací strom vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm půjde vyhledávat v čase $\mathcal{O}(\log N)$. Žádný div: Stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu.

David Matoušek & Martin Mareš

27-2-1 Systém Patriot

Toto je úloha, která má více než jedno správné řešení. Předvedeme si to, které vymyslela většina řešitelů. Jiná řešení však nemusí být špatně.

Střely si nastrkáme do dvou hald, A a B. Halda A bude maximová (na vrcholu sedí maximum), B minimová. Dále, všechny prvky v A budou nejvýše tak velké, jako libovolný prvek B (tedy formálněji, $\forall a \in A, b \in B; a \leq b$). A počty prvků v haldách se budou lišit maximálně o 1.

K čemu nám takové komplikované podmínky budou? Celkem jednoduše nahlédneme, že medián se nachází na vrcholu jedné z hald a z velikostí hald lze vždy určit, na vrcholu které.

Nyní si rozmyslíme, jak s touto datovou strukturou pracovat. Když chceme střelu přidat, porovnáme ji s aktuálním mediánem a určíme, do které haldy ji přidáme (pokud je větší než medián, jde do B, pokud menší, tak do A, u stejných prvků je to jedno). Poté zjistíme, jestli nám tato halda příliš nevyrostla. Pokud by byla o 2 prvky větší, tak její vrchol přestěhujeme do druhé haldy, aby se to vyrovnalo.

Jak vystřelit střelu? Medián najít umíme, takže jej stačí odebrat a poté opět zkontrolovat, že haldy mají dostatečně podobnou velikost.

V každé operaci provádíme konstantně mnoho operací vložení a odebrání z hald, tedy jedna operace nám zabere $\mathcal{O}(\log n)$ – takovou složitost má operace s haldou. Paměť nám bude stačit lineární, opět proto, že halda potřebuje lineární množství paměti s množstvím prvků v ní uložených.

Chceme ještě nahlédnout, že to opravdu funguje. Protože nalevo i napravo od hranice mezi haldami je stejné množství prvků a na vrcholu je ten prvek „k hranici“, opravdu je medián jedním z vrcholů. Nakonec si všimneme, že pravidla z druhého odstavce neporušíme ani jednou operací, a tedy medián neztratíme.

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

Michal „vorner“ Vaner

27-2-2 Prohledávání budov

Než pošleme záchranné týmy prodírat se hromadami trosek v budovách, nejprve učiníme několik pozorování.

Tím prvním je, že nám nezáleží na pořadí budov. Můžeme si je přeuspořádat, jak chceme, a stejně to na počtu nutných týmů nic nezmění. Pro jednoduchost si tedy budovy seřadíme podle velikosti od nejmenší po největší.

Druhým důležitým pozorováním je to, že budovy začínají všechny u země. Na tom není asi nic objevného, ale díky tomu platí, že u země jsou patra budov zastoupena nejhustěji a směrem nahoru jich jen ubývá (jak jednotlivé budovy postupně končí).

Pokud tedy chceme nějaká patra prohledat horizontálně, tak to určitě bude nějaký úsek pater začínajících u země a končící v nějakém patře K. Budovy přesahující nad toto patro pak prohledáme vertikálně. Kdyby v nějakém optimálním řešení netvořila horizontálně prohledávaná patra souvislý úsek od země, tak takové řešení můžeme bez jakéhokoliv zhoršení převést na stav se souvislým úsekem.

Našemu řešení tedy stačí jen najít toto číslo K. Všech N budov určitě umíme prohledat pomocí N týmů (do každé budovy jeden vertikálně), takže začneme s $K = 0$ a budeme ho zkoušet zvedat. Jako V si označíme počet vertikálně prohledávaných budov (na počátku tedy $V = N$).

Budeme naší hranicí postupně skákat budovu po budově nahoru a budeme si pamatovat, kdy byl počet týmů $(V+K)$ nejmenší. Vždy, když odebereme budovu, snížíme V o jedna a zvedneme K na výšku této budovy.

Hlavní část řešení zvládneme v lineárním čase vůči velikosti vstupu, ale kvůli úvodnímu třídění máme časovou složitost bohužel $\mathcal{O}(N \log N)$. Neumíme to ještě zrychlit?

Stačí si přidat další pozorování: Budovy vyšší než N určitě prohledáme vertikálně, protože K nikdy nemá smysl zvedat nad N. Můžeme si tedy tyto budovy na začátku v lineárním čase vyřadit a uvažovat jen budovy nižší než N. A ty již umíme setřídit přihrádkovým tříděním v lineárním čase. Dostali jsme se tedy k paměťové i časové složitosti $\mathcal{O}(N)$.

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

Jirka Setnička

27-2-3 Průjezd jeřábu

Tato CodExovka se ukázala býti těžší než předchozí (soudě dle vámi získaných bodů). Nicméně podobně jako předchozí CodExovka, tak i tato má celkem jednoduché řešení. Ukážeme si dva možné přístupy.

Spojování komponent

Budeme si sestavovat mapu oblastí, mezi kterými křižovatkami se můžeme s jak vysokým jeřábem pohybovat. Přidání do mapy ještě nepoužité silnice, po které se dá jet s nejvyšším jeřábem, nemůže nic pokazit. Principem prvního řešení je toto pravidlo opakovaně použít na zbylé silnice, dokud nespojíme základnu se skladištěm.

Na začátku představuje každá křižovatka osamocenou komponentu, žádné silnice jsme zatím ještě nepoužili. Všechny silnice si na začátku setřídíme a poté je po jedné odebíráme a propojujeme dvě komponenty, které spojuje tato silnice. Pokračujeme, dokud nezískáme komponentu, ve které se nachází jak základna, tak sklad. Pak vypíšeme, že nejvyšší jeřáb může mít výšku odpovídající hodnotě na poslední přidávané silnici. Pak pro nalezení cesty stačí projít ze základny tu jednu komponentu do hloubky, dokud nenarazíme na sklad.

Řešení má tedy časovou složitost $\mathcal{O}(M \log M)$, protože tak dlouho nám trvá setřídit všechny silnice, kterých je M. A pokud použijeme chytře implementovanou datovou strukturu, která umí rychle najít, zda dva vrcholy jsou v jedné komponentě, tak si tím časovou složitost nezhoršíme, protože celkem všechny testy a spojení zaberou $\mathcal{O}(M \log N)$.

Pro podrobnosti se podívejte na datovou strukturu Disjoint-Find-Union (DFU),[8] popsanou v kuchařce druhé série. Projití do hloubky nám zabere času nejvýše $\mathcal{O}(N + M)$, tedy opravdu celková časová složitost je $\mathcal{O}(M \log M)$. Paměťová

[8] `http://ksp.mff.cuni.cz/viz/kucharky/minimalni-kostra`

složitost je $\mathcal{O}(N+M)$, protože máme uložený celý vstup a DFU přidá ke každé silnici jen konstantně mnoho informací.

Program (C++) – Disjoint-Find-Union:
`http://ksp.mff.cuni.cz/viz/27-2-3-dfu.cpp`

Inspirace Dijkstrovým algoritmem

Jistě si říkáte, že předchozí řešení asi nebude nejlepší, protože se nám může mnohokrát stát, že spojujeme křižovatky, které nakonec stejně nebudou v té komponentě, kde je základna a sklad. A máte pravdu, lze udělat o trošku lepší řešení.

Začneme silnice procházet do šířky ze základny. Přitom budeme rozšiřovat oblast, do které se umíme dostat. Vždy použijeme silnici vedoucí z prozkoumané oblasti někam dál, po které může jet nejvyšší jeřáb, což je v podstatě princip Dijkstrova algoritmu.[9] Předchozí krok opakujeme až do chvíle, kdy se dostaneme do skladu, nebo nám dojdou silnice. Pak stačí zrekonstruovat cestu, po kterých silnicích jsme šli, protože víme, že lepší cesta nemůže existovat. Důkaz platnosti tohoto pozorování necháváme na rozmyšlení si čtenářům.

Pro každou křižovatku si tak budeme pamatovat hodnotu, kterou si označíme `value`. Bude představovat to, jakým nejvyšším jeřábem jsme schopní dojet na tuto křižovatku. U každé silnice víme, jaký nejvyšší jeřáb jí může projet. Tuto hodnotu si označíme jako `h`. A nakonec v `lastValue` bude `value` křižovatky, ze které jsme přijeli.

Pak při každém použití některé silnice nastavíme na křižovatku, do které dojedeme, hodnoty takto:

```
value = max(value, min(h, lastValue))
```

Abychom zbytečně nemuseli zjišťovat, jestli jsme někdy na dané křižovatce již byli, tak na začátku všechny křižovatky mají `value = 0`, jen křižovatka základny (start) má nastaveno `value =` ∞ (v programu nějaké dostatečně vysoké číslo).

Toto řešení má časovou složitost $\mathcal{O}(M \log N)$, pokud aktualizujeme hodnoty v haldě. Pokud pouze vkládáme, tak je časová složitost $\mathcal{O}(M \log M)$, což je až $\mathcal{O}(M \log N^2) = \mathcal{O}(2 \cdot M \log N)$. Tedy na soutěžích je asi rozumné nezdržovat se programováním aktualizování hodnot v haldě, protože je to asi nejsložitější operace, navíc musíte vědět, kde se který vrchol v haldě nachází. V praxi je dobré aktualizování implementovat, protože budeme mít zhruba dvakrát rychlejší algoritmus a navíc zabereme méně paměti. Asymptoticky si ale nepomůžeme, protože musíme mít uložený celý vstup. To vede na paměťovou složitost $\mathcal{O}(N+M)$.

Program (C) – Dijkstra s aktualizovanou haldou:
`http://ksp.mff.cuni.cz/viz/27-2-3-dijkstra.c`

Vojta Sejkora

27-2-4 Stahování map

Lehčí varianta

Nejprve se zastavme u lehčí varianty úlohy, kde máme jenom $N = 3$ mapy o rozměrech $R \times S$, kde $R, S \leq 20$, a číslo W, které udává režii přenosu mapy diferencí.

Pojmenujme si mapy A, B a C. Vytvoříme si z nich vrcholy grafu. Natáhneme mezi nimi hrany, pokud $D_{AB}W < RS$, kde D_{AB} je počet rozdílných políček mezi mapami A a B. Všimněme si, že $D_{AB} = D_{BA}$, hrany jsou tedy neorientované.

Hrana mezi A a B znamená, že přenést mapu B diferencí od A se vyplatí, protože to bude stát méně, než kdybychom mapu B přenášeli samostatně. Hranám přidáme ohodnocení, to bude právě $D_{AB}W$.

Kolik hran v grafu teď můžeme mít?

- 0 – v tom případě všechny mapy stáhneme přímo ze serveru.
- 1 – jednu mapu, ze které vede hrana, stáhneme přímo, druhou diferencí od ní. Nezáleží na tom, která bude která, obojí nás bude stát stejně. Třetí mapu nám nezbývá než přenést přímo.
- 2 – graf má tedy tvar cesty na třech vrcholech. Vybereme jednu mapu, například tu prostřední, tu stáhneme přímo, zbylé diferencí od ní. Všimněme si, že kdybychom krajní mapu stáhli přímo, od ní diferencí prostřední, a od ní diferencí tu poslední, také nás to bude stát stejně.
- 3 – Nejradši bychom všechny mapy přenášeli diferencí, tak to ale nejde. Nejméně jednu mapu musíme přenést ze serveru, a zbylé dvě od ní. Aby nás přenášení stálo co nejméně, použijeme dvě nejlehčí hrany. Tu nejtěžší z grafu vyjmeme. Tímto jsme situaci převedli na známý případ se dvěma hranami.

Toto by k vyřešení lehčí varianty úplně stačilo. Podívejme se ale na graf ještě trochu jinak. Představme si, že máme čtyři vrcholy, mapy A, B, C a server. Ze serveru vede do každé mapy hrana o váze RS, všechny ostatní hrany mají váhu danou počtem rozdílných políček krát W.

Z tohoto grafu potřebujeme vybrat tři hrany, protože celkem budeme provádět tři přenosy. Vybírat ale nesmíme úplně libovolně. Každý vrchol musí být připojen nějakou hranou k ostatním, protože kdyby nebyl, zůstane nějaká mapa nestažená nebo stáhneme všechny diferencí, ale nic ze serveru jako první.

Vrcholy jsou čtyři, hrany jsou tři, výsledný podgraf musí být souvislý a neobsahovat cyklus, to znamená, že to bude strom. Navíc hledáme nejlehčí strom, takový, který má součet hran nejmenší možný. Takovýto strom se nazývá minimální kostra.

Generické řešení

A je to. Máme řešení i pro plnou variantu úlohy, kde je map více než tři, ale ne víc než 500. Vytvoříme si z map a serveru graf, ohodnotíme hrany diferencí map krát W. (Tu spočítáme snadno, projdeme mapy políčko po políčku a spočítáme si počet těch rozdílných.) V tomto grafu nalezneme minimální kostru algoritmem z kuchařky. Ne nadarmo v ní byly zmíněny tři. Vyberme si třeba *Kruskalův algoritmus*.

(Mimochodem, když neřekneme přesně, který algoritmus na minimální kostru se má použít, bude naše řešení generické. Ten, kdo ho bude programovat, si musí sám nějaký vybrat a dosadit.)

Pořadí stahování map pak snadno vypíšeme procházením stromu od serveru do šířky nebo do hloubky v čase $\mathcal{O}(N)$.

Z toho vyplývá časová složitost, ta je $\mathcal{O}(N^2(RS + \log N))$, protože $\mathcal{O}(N^2RS)$ trvá postavení grafu a $\mathcal{O}(M \log N)$ zabere Kruskalův algoritmus (M je počet hran, v našem případě N^2).

[9] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

Paměťová složitost je $\mathcal{O}(N^2 + NRS)$, potřebujeme si pamatovat celý vstup a všechny hodnoty diferencí.

Jelikož jste mohli předpokládat, že $N \leq 500$ a $R, S \leq 20$, je toto řešení dostatečně rychlé na to, aby mohlo být v praxi použitelné. Většina z vás se u něčeho takového tedy zastavila a neměli jste motivaci přemýšlet nad ještě rychlejším řešením. Proto jste i za toto mohli získat plný počet bodů.

Někteří z vás si všimli, že existuje vlastně jen RS možných ohodnocení hran, počet rozdílných políček ve dvou mapách může být jedině mezi 0 a RS. To znamená, že je můžeme setřídit přihrádkově, v čase lineárním s jejich počtem, tedy v $\mathcal{O}(N^2)$.

Kruskalův algoritmus s využitím DFU (s *union by rank* a s *path compression*) pak vytvoří minimální kostru v čase $\mathcal{O}(M\alpha(N))$, což je v našem případě $\mathcal{O}(N^2\alpha(N))$, kde α je inverzní *Ackermannova funkce* (její definici najdete v kuchařce, zde stačí, že roste extrémně pomalu).

Když sečteme vytváření grafu, třídění a vytváření minimální kostry, vyjde nám časová složitost $\mathcal{O}(N^2(RS + \alpha(N))$.

S naším omezením je $RS \leq 400$ a $\alpha(N) \leq \alpha(500) \leq 4$. Můžeme tedy v klidu říct, že celková složitost je zhruba $\mathcal{O}(N^2RS)$. Paměťová zůstává.

Program – Kruskalův algoritmus (Python 3):
`http://ksp.mff.cuni.cz/viz/27-2-4-kruskal.py`

Zlepšení o slepičí krok

Tím se ale nespokojíme, existuje pěkné řešení, které je asi tak o slepičí krok lepší, není „téměř“ lineární, ale „úplně“ lineární s velikostí grafu. Použijeme upravený Jarníkův algoritmus.

Ten staví kostru postupně od jednoho vrcholu a vždy k ní přidává tu nejlehčí hranu, která vede z kostry do zbytku grafu. K tomu používá nějakou datovou strukturu Q, která je vlastně množinou vrcholů schopnou vydat minimum. Ještě si připravíme dvě pole t a d indexované vrcholy; d udává nejkratší hranu z kostry do vrcholu, v t bude na konci algoritmu minimální kostra zadaná jako postup přilepování hran stromu od kořene.

Na začátku si do Q vložíme všechny vrcholy, d bude pro každý vrchol kromě startovního nekonečno, pro startovní vrchol 0, t nechť je pro každý vrchol NULL, neexistující vrchol. Tato hodnota nakonec zbude jen u startovního vrcholu.

Potom se provedou následující kroky:

1. dokud $Q \neq \emptyset$:
2. $\quad u \leftarrow$ vyjmi minimum z Q podle d
3. $\quad$ pro každého souseda v vrcholu u dělej:
4. $\quad\quad$ pokud $v \in Q$ a $d(v) > w(u, v)$:
5. $\quad\quad\quad d(v) \leftarrow w(u, v); t(v) \leftarrow u$

Výstup: t

Co bude datová struktura Q? Podobnou jistě znáte z Dijkstrova algoritmu, tam se používá halda. My ale využijeme toho, že hrany mají jen RS různých hodnot, a tak použijeme pole. Indexovat ho budeme od 0 do RS. Výběr minima bude trvat $\mathcal{O}(RS)$, pole budeme procházet od začátku do konce a v nejhorším případě najdeme první vrchol až na konci. Každý vrchol má $\mathcal{O}(N)$ sousedů, pro každého se bude provádět změna hodnoty, ta trvá pro každého ale jen $\mathcal{O}(1)$, vrchol se pouze přemístí na jiný index.

Časová složitost tedy bude $\mathcal{O}(N^2RS)$. Kolikrát budeme potřebovat znát váhu hrany? Jen jednou. Nemusíme si ji tedy nikam ukládat, ale spočítáme si ji ve chvíli, kdy ji poprvé využijeme. Díky tomu nám stačí pouze lineární paměť k velikosti vstupu, $\mathcal{O}(N \cdot RS)$.

Zbývá si už jen domyslet pár implementačních detailů. K tomu vám může pomoci zdroják:

Program – Jarníkův algoritmus (Python 3):
`http://ksp.mff.cuni.cz/viz/27-2-4-jarnik.py`

Dominik Macháček

27-2-5 Nejdelší příkaz

Úlohu si nejdříve zanalyzujeme. Potřebujeme v zadaných slovech najít co nejdelší žebřík, tj. posloupnost slov, kde každé další vznikne vložením právě jednoho písmene do slova předešlého. Pokud se na žebřík podíváme z druhé strany, hledáme posloupnost slov, kde každé další vznikne vyškrtnutím právě jednoho písmene.

Pro jedno konkrétní slovo můžeme zkusit všechny možnosti vyškrtnutí a kontrolovat, zda nově vzniklé slovo máme na seznamu. Pokud ano, vyzkoušíme pro něj to samé rekurzivně. Výše popsaný proces zkusíme začít v každém slově a tam, kde se dostaneme v rekurzi nejhlouběji, je naše řešení.

Pro dokončení popisu řešení ještě potřebujeme najít vhodnou datovou strukturu pro uchovávání slov a jejich hledání. Takovou je třeba trie. Pokud jste o ní ještě neslyšeli, můžete se o ní dočíst v naší kuchařce o hledání v textu.[10]

Tím dostáváme funkční řešení. Kvůli rekurzi ale bohužel ne dost dobré. Můžeme si všimnout, že při výpočtu můžeme zbytečně spouštět výpočet vícekrát pro stejné slovo. Tomu se budeme snažit vyvarovat a výpočet pro jedno slovo vždy spouštět pouze jednou (použijeme myšlenku dynamického programování).[11]

Trii budeme stavět od nejkratších slov po nejdelší a v koncovém vrcholu slova si budeme pamatovat délku nejdelšího žebříku pro dané slovo nebo nulu, pokud ve vrcholu trie žádné slovo nekončí.

Při přidávání slova do trie tedy nejdřív zkusíme najít všechny možné předchůdce v dosavadní trii, z nich vybereme toho s nejvyšší hodnotou žebříku a aktuální slovo přidáme s hodnotou o jedna větší. Na konci pak jen najdeme největší hodnotu v trii a je to! Z toho pak jednoduše získáme výsledný žebřík.

Časová složitost je $\mathcal{O}(NL \cdot$(složitost trie)), což je pro konstantní abecedu $\mathcal{O}(NL^2)$, kde N je počet slov na vstupu a L je délka nejdelšího z nich. Pro každé slovo provádíme $\mathcal{O}(L)$ dotazů na trii, kde každý má složitost $\mathcal{O}(L)$.

Pro více detailů můžete nahlédnout do vzorové implementace v jazyce C++.

Program (C++):
`http://ksp.mff.cuni.cz/viz/27-2-5.cpp`

Karel Tesař

[10] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`
[11] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

27-2-6 Testování odezvy

Na vstupu máme matici A o rozměrech $N \times N$ a chceme určit, jestli to může být matice vzdáleností v ohodnoceném stromě, a najít takový strom T. Z příkladu je jasné, že nám jde o neorientovaný graf (matice je symetrická, tedy $A_{ij} = A_{ji}$) bez smyček (na diagonále jsou nuly: $A_{ii} = 0$).

Pokud strom existuje, ve vstupní matici je spousta hodnot, které odpovídají cestám s více hranami. My ale potřebujeme vědět, které hodnoty odpovídají jednotlivým hranám. Kdybychom to věděli, strom už si ze seznamu hran snadno převedeme do příjemnější reprezentace a stejně snadno ověříme, že A je jeho matice vzdáleností.

Při ověřování můžeme využít toho, že cesta mezi každými dvěma vrcholy je ve stromu určena jednoznačně, takže vzdálenost všech vrcholů od jednoho pevně zvoleného jde snadno spočítat během průchodu (třeba DFS). Postupně tedy spustíme průchod z každého vrcholu stromu T a ověříme, že ve vstupní matici A jsou správné hodnoty. Strom by měl mít N vrcholů a $N - 1$ hran (udělejte si cvičení z grafové kuchařky[12] a dokažte to!), takže všech N průchodů by dohromady trvalo $\mathcal{O}(N^2)$. To je lineární s velikostí kontrolované matice, kterou je potřeba projít celou, takže jsme na optimu.

Dobře, dobře, kontrolu matice vzdáleností pro daný strom jsme vymysleli, kde ale ten strom sebrat?

V lehčí variantě úlohy měl být strom T neohodnocený, lépe řečeno ohodnocený jedničkami. Stačilo tedy vzít všechny jedničkové hrany a ověřit, že tvoří strom. Žádná jedničková hrana nemůže odpovídat cestě s více než jednou hranou, protože by pak musela mít ohodnocení alespoň 2, takže jsme do T nepřidali žádnou hranu, která tam být neměla. Ze zadání a výběru hran plyne, že tam ani žádná nechybí. Ještě by se nám ale mohlo stát, že jsme dostali něco, co vůbec není strom. To můžeme zkontrolovat více způsoby, podle použité charakterizace stromů.

Strom je souvislý graf bez kružnic. Pokud prohledáním grafu (třeba DFS) najdeme všechny vrcholy, je souvislý. Kružnici detekujeme, pokud najdeme zpětnou hranu ve stromě průchodu do hloubky. (Pokud nevíte, o čem je řeč, osvěžte si paměť pohledem do již zmiňované grafové kuchařky.) Alternativně můžeme testovat, že má strom T správný počet vrcholů i hran. Oba tyto algoritmy běží v čase i prostoru $\mathcal{O}(N)$. Ostatní charakterizace stromů nejsou pro testování vhodné. Mimochodem, tahle část našeho algoritmu je shodná s úlohou 27-Z2-5.[13]

V lehčí variantě tedy máme snadný tip na strom T a ověříme, že to opravdu strom je a že vzdálenosti v něm odpovídají vstupní matici A. Načíst vstup zvládneme v čase i prostoru $\mathcal{O}(N^2)$, najít všech $N - 1$ jedničkových hran a postavit z nich rozumně reprezentovaný strom taktéž, průchod stromem pro ověření souvislosti zabere dokonce jen čas $\mathcal{O}(N)$ a konečnou kontrolu matice vzdáleností jsme už spočítali na $\mathcal{O}(N^2)$. Celkově se tedy vejdeme do času $\mathcal{O}(N^2)$, což je lineární s velikostí vstupu, tedy zaručeně optimální. V paměti bude nejvíc místa zabírat matice A, zbytek se v tom ztratí, takže i paměťové nároky budeme mít lineární.

V těžší variantě se musíme zamyslet o trochu víc. Možná nás napadne se na matici A koukat jako na matici sousednosti úplného grafu K, kde strom T hledáme jako podgraf. Hm, a T musí mít stejný počet vrcholů jako K... takže je jeho kostrou! Nebude to minimální kostra, když už je popsaná v kuchařce k této sérii?

Úplný graf je souvislý, takže vždycky nějakou kostru má, speciálně tedy i minimální kostru T. Dokažme, že pokud má úloha řešení, je jím minimální kostra. Pro spor předpokládejme, že existuje řešení S a není to minimální kostra T. Vezmeme nejlehčí hranu uv z T, která není v S, a podíváme se na běh Kruskalova algoritmu, který v každém kroku spojí dvě komponenty souvislosti nejlehčí hranou mezi nimi.

V řešení S jsou vrcholy u a v spojené cestou $S[u, v]$.

- Pokud je $S[u, v]$ tvořena pouze hranou uv, dostáváme spor s volbou uv, protože jsme chtěli, aby nebyla v S.
- Jinak je $S[u, v]$ tvořena aspoň dvěma hranami. Jelikož při přidávání hrany uv do T musely být vrcholy u a v v různých komponentách a Kruskalův algoritmus zpracovává hrany v pořadí rostoucí váhy, musí v $S[u, v]$ být aspoň jedna hrana aspoň stejně těžká jako uv. To je spor s tím, že S je řešení, jelikož by uv nutně byla lehčí než $S[u, v]$, ale měla by být stejně těžká.

A máme dokázáno. Nebo ne? Celou dobu uvažujeme jen kladné váhy hran. Jestli jste předpokládali na vstupu matici sousednosti kladně ohodnoceného úplného grafu automaticky, nic se neděje, za to jsme body nestrhávali. Úloha byla zamýšlená s kladným ohodnocením a příklad tomu nasvědčoval. Přesto bychom rádi v rychlosti zmínili také rozšířenou variantu i se zápornými nebo nulovými vahami, která některé z vás napadla.

Minimální kostru pro **libovolné celočíselné váhy hran** zvládnou všechny standardní algoritmy, dokonce bez modifikace. Přinejhorším bychom mohli nezáporné váhy získat odečtením váhy nejzápornější hrany w^- od váhy každé hrany. Tím bychom každou kostru ztížili o $(N - 1) \cdot w^-$, takže bychom uspořádání koster podle váhy nezměnili.

Potíž je v tom, že se zápornými vahami minimální kostra neřeší naši úlohu. Představte si třeba trojúhelník s hranami $1, 2, -1$, obsahující cestu $2, -1$. Pro úlohu se zápornými hranami neznáme efektivní řešení. Ale ani nemáme důkaz její NP-těžkosti, takže kdybyste na jedno nebo druhé přišli, dejte nám vědět na fóru. :-)

Nulové hrany umíme bez újmy na obecnosti spravit kontrakcí na začátku a dekontrakcí na konci algoritmu. V podstatě tak uvedeme v realitu, co nulová hrana neformálně říká: že její krajní vrcholy jsou vlastně vrchol jeden. Kontrakce hrany uv se projeví vyhozením řádku v a sloupce v z matice A. Přitom kontrolujeme, že se shodují s řádkem u a sloupcem u, jinak strom neexistuje. Tato úprava se hodí jen pro účely analýzy, algoritmus poběží správně i bez ní (zdůvodnění si rozmyslete).

Celý algoritmus v pseudokódu:

1. Načti matici A a zkontroluj, že je symetrická, má nuly na diagonále a jinak obsahuje jen kladné hodnoty. Pokud kontrola neuspěje, vrať „strom neexistuje“.
2. Najdi minimální kostru T úplného grafu s maticí sousednosti A.
3. Ověř, že získaná kostra T má matici vzdáleností rovnou matici A. Pokud se matice liší, vrať „strom neexistuje“.
4. Vrať strom T.

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

[13] `http://ksp.mff.cuni.cz/viz/27-Z2-5`

Zbývá vybrat algoritmus pro **nalezení minimální kostry**. Vymýšlet vlastní nemá smysl, jen bychom znovu vynalézali kolo a mořili se s důkazem korektnosti a složitosti.

Kruskalův algoritmus máme rozebraný v kuchařce, nejdéle na něm trvá třídění hran v čase $\mathcal{O}(M \log M) = \mathcal{O}(M \log N)$, což by pro nás bylo $\mathcal{O}(N^2 \log N)$. Hodí se pro řídké grafy a kvůli snadné implementaci, když zrovna nemáte po ruce haldu.

Pro husté grafy, kterým úplný graf bezesporu je, ale je vhodnější Jarníkův algoritmus. S Fibonacciho haldou běží v čase $\mathcal{O}(M + N \log N)$, což by pro náš případ bylo $\mathcal{O}(N^2 + N \log N) = \mathcal{O}(N^2)$.

Kniha The Algorithm Design Manual od Stevena S. Skieny tvrdí, že Jarník s párovacími haldami je nejrychlejším praktickým řešením pro řídké i husté grafy, se stejnou asymptotickou složitostí jako s Fibonacciho haldou. Nás z knihy bude zajímat implementace Jarníka, která běží vždycky v $\mathcal{O}(N^2)$ a ani nepotřebuje žádnou složitou datovou strukturu. V češtině ji najdete popsanou v Medvědových poznámkách z ADS.[14]

Tato varianta Jarníka je k vidění ve vzorovém řešení. Nalezení kostry je nejtěžší částí algoritmu, tedy i těžší varianta úlohy má řešení v lineárním čase i prostoru.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-2-6.c`

Tomáš „Palec“ Maleček

27-2-7 Shellové skripty

Snad každý, kdo se do úloh pustil, vyřešil všechny – to nás těší. Řešení obsahovala většinou jen drobné nedostatky, hlavně u vypisování proměnných. Když vypisujete obsah proměnné, který může obsahovat mezery, je potřeba ji vypsat v uvozovkách, jinak se mezery ztratí během rozdělování na slova.

```
echo "$prom"
```

Teď už k řešení samotných úloh. První byla velmi jednoduchá, stačilo se podívat do manuálu k příkazu `test`. Častou chybou nicméně bylo, že jste se snažili testovat na neprázdnost i adresáře. Taky se často v řešení vyskyla konstrukce:

```
for f in $(ls)
```

To není špatně (ani jsme za to nestrhávali body), jen je to zbytečně pomalé. Stejnou službu umí vykonat shell sám expanzí:

```
for f in *
```

Občas se taky v řešení objevilo testování neprázdnosti pomocí `wc -c`. Tuto metodu jsme použili v řešení minulé série, protože jsme ještě neznali lepší prostředky. Musí vám být ale jasné, že takový test bude ukrutně pomalý.

Jak tedy mohlo vypadat ideální řešení?

```
for f in *; do
        [ -f "$f" -a ! -s "$f" ] && echo "$f"
done
```

Druhá úloha byla trochu triková. Někteří se k tomu snažili použít pole, je to ale zbytečně silný a nepřenositelný kanón. Dnes už je situace s podporou polí v různých shellech lepší než před lety, stále se ale budeme snažit používat jednodušší prostředky.

Naproti tomu, využít k řešení malou utilitku `tac`, která vypíše řádky vstupu v opačném pořadí, je moc pěkný nápad:

```
IFS='
'
echo "$*" | tac
```

Slepit malé utilitky, které za nás udělají špinavou práci, je přesně filozofie programování v shellu. Jak to udělat bez ní, je zamýšlené autorské řešení:

```
for arg; do
        p="$arg $p"
done
echo "$p"
```

Jestli budou argumenty na oddělených řádcích, nebo na jedné, nebylo důležité, stejně tak jestli se přidají apostrofy, uvozovky atp. okolo. Důležité ale bylo, aby se neztrácely mezery a argumenty tvořené pouze bílými znaky.

Třetí úloha byla jen krátký test pozornosti. Pokud použijeme rouru, odsuzujeme příkazy k tomu, aby se spustily v subshellu. Pokud v subshellu změníme nějaké proměnné, změní se jen pro subshell – a zaniknou spolu s ním. Proto musíme `read` a `echo` provést v jednom složeném příkazu.

Ve čtvrté úloze jste si měli vyzkoušet sílu proměnné `IFS` ve spolupráci s příkazem `read`. Někteří se snažili řádky zpracovávat příkazem `cut`, nebo ještě kostrbatěji pomocí `sedu`. To bohužel většinou vedlo k tomu, že jste soubor četli pro každý řádek celý znovu, a to opravdu není správný přístup.

```
while IFS=: read user x x x x x shell; do
        [ -n "$user" ] && echo "$shell" >"$user"
done </etc/passwd
```

Všimněte si pěkné zkratky na posledním řádku.

A konečně poslední pátá úloha byla na procvičení možností `case`. Jen málokdo to ale pochopil, proto jste do řešení často psali šílené podmínkové konstrukce. Také jste často zapomínali uvozovkovat alespoň jméno. Jinak ale nebyl na úloze žádný chyták, proto pojďme rovnou k řešení:

```
while read akce pohl jmeno; do
        case "$akce $pohl" in
                "prichod M")
                        echo "Prisel $jmeno" ;;
                "odchod M")
                        echo "Odesel $jmeno" ;;
                "prichod F")
                        echo "Prisla $jmeno" ;;
                "odchod F")
                        echo "Odesla $jmeno" ;;
        esac
done
```

Díky za pěkná řešení, těšíme se na další!

Ondra Hlavatý

[14] `http://mj.ucw.cz/vyuka/1011/ads1/7-kostry.pdf`

Výsledková listina druhé série dvacátého sedmého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*2-1*	*2-2*	*2-3*	*2-4*	*2-5*	*2-6*	*2-7*	*série*	*celkem*
0.					9	10	10	12	12	14	12	60,0	116,0
1.	Jan Špaček	G Wicht	4	7	9	8	10	12	12	12	10	57,6	111,4
2.	Marek Černý	G Chrudim	4	7		8	10		12	12	11	55,0	111,0
3.	Richard Hladík	GOAMarLaz	2	12	9	9	7	12	12	12	10,5	55,1	107,4
4.	Jakub Tětek	CírkG Plzeň	1	2	9			12	12	8	8,5	54,3	106,4
5.	Stanislav Lukeš	GPísnickáPH	2	3	9			11	12	8	11	55,1	106,2
6.	Adrián Goga	SPŠNitra	4	2	8	7	10	4	12	11	10	55,5	104,8
7.	Jan Tománek	GPelhřimov	4	3	5	10	4		12	12	7	49,5	101,7
8.	Martin Scheubrein	G MNám Třb	3	2	9		10		12	3	10	48,6	101,3
9.	Přemysl Šťastný	GŽamberk	2	5	8	7,5	10	1	12		8	48,1	99,4
10.	Anna Gajdová	GFPValMez	4	4	9	8	4	12	0	9		48,1	97,3
11.	Róbert Selvek	G KošiceS	3	2	9	3	10		12		10	48,1	94,4
12.	Jan Knížek	G Strakon	4	15	6	8	1	12	12	10	8	46,1	94,2
13.	Jan Kočur	G Wicht	4	2	9	7	0		12	1	10	43,9	90,1
14.	Václav Volhejn	GKepleraPH	2	12	9	8	0	11	12	11		50,4	89,9
15.	Štěpán Hudeček	G Litovel	3	2		4	2		12		10	34,4	82,7
16.	Lukáš Ulrich	SSŠVTPraha	4	2	4	3,5	3		12	3	6	40,3	82,6
17.	Václav Šraier	GČeskoliPH	2	2	8	4	0		12		9	38,5	80,7
18.	Jakub Zárybnický	GTomkovaOL	4	7	9	8			12		9,5	40,3	80,2
19.	Michal Töpfer	G DrJPekMB	2	2	8	4			1		9	26,1	76,7
20.	Michal Převrátil	GKlatovy	2	2	9	6	4	5	12			43,0	74,1
21.	Jan Gocník	GJŠkodyPŘ	3	2	4		0		12		9	29,5	64,1
22.	Jiří Sejkora	GVoděraPH	3	2					12			12,0	60,0
23.	David Cholewa	GMatOS	4	2	3	7,5		3	0	2		25,2	46,2
24.	Jiří Vozár	G UherBrod	3	2	4						10	17,6	44,2
25.	Pavel Turinský	G Brandýs	2	2	2		2				11	20,0	41,7
26.	Martin Zoula	GNadKavaPH	3	2	3				0			5,5	36,4
27.	Jan Bouček	GKepleraPH	2	2		8			7			16,4	33,2
28.	Václav Končický	GSOŠ FrMís	4	2								0,0	32,9
29.	Jan Pokorný	G_Bučovice	3	5	9	7	0		2		12	31,5	31,5
30.	Barbora Sedláková	GKonštanPV	4	2	2	1			2			8,5	27,4
31.	Filip Bialas	GOpatovPHA	2	5								0,0	20,0
32.	Vít Macura	GOAMarLaz	2	2					2			2,0	18,9
33.	Jakub Lukeš	GNAlejíPH	2	1								0,0	14,0
34.	Dalimil Hájek	GKepleraPH	4	15								0,0	13,4
35.	Martin Kubeša	GJŠkodyPŘ	3	1								0,0	12,8
36.	David Juřica	GNadŠtolPH	2	2								0,0	10,9
37.	Jan Kaifer	GČesBrod	−1	1								0,0	10,6
38.	Matěj Konečný	GJírovcČB	4	2			10					10,0	10,0
39.	Jan Burda	G_Holice	1	1					9			9,0	9,0
40.	Václav Steinhauser	GDačice	1	1		5						7,9	7,9
41.	Josef Vávra	SJec	4	1	3	0						5,7	5,7
42.	František Dostál	VSPŠEOc	4	1					4			4,0	4,0
43.	Michael Novák	SSŠVTPraha	4	1								0,0	2,0
44.	Václav Rozhoň	GJirsíkaČB	4	7			1					1,6	1,6