Korespondenční Seminář z Programování

30. ročník **KSP** **Červen 2018**

Milí řešitelé a řešitelky!

Ačkoliv by to tak někdy mohlo vypadat, my orgové na vás účastníky nezapomínáme. Až si budete venku užívat teplých dnů, které připomínají spíše léto než jaro, nezapomeňte si s sebou přibalit i toto zadání, ve kterém naleznete poslední sadu úloh tohoto ročníku. Získáte dostatečný počet bodů a diplom úspěšného řešitele? Pojedete na Podzimní soustředění? Ať už to vyjde nebo ne, doufáme, že jste si z řešení tohoto ročníku odnesli nové znalosti a zkušenosti a že nepovažujete čas strávený nad úlohami za promarněný.

Díky řešení KSP se také můžete vyhnout přijímacím zkouškám na MFF UK! Stačí, když získáte alespoň polovinu bodů z ročníku (tedy 150 bodů) a my vám vystavíme osvědčení úspěšného řešitele, díky kterému vás (nejdříve příští školní rok) přijmou na MFF bez zkoušek.

Termín série: **25. června v 8:00 (seriál 30. června)**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*

Odměna série: **Sladkou odměnu** si vyslouží každý, kdo získá alespoň 15 bodů z obou seriálových úloh dohromady.

Pátá série třicátého ročníku KSP

Ve třetí sérii jsme orgy opustili, protože to vypadalo, že pana Nápovědu už konečně dostali. . .

„Podle časových záznamů k tomu výbuchu došlo okamžitě potom, co jsme se teleportovali. Nápověda neměl sebemenší šanci to přežít. Navíc se ten japonský gang rozpadl.“

⋆⋆⋆

„Pomóóóóc! Nápověda je tady!“
Ale vraťme se na začátek.

⋆⋆⋆

Orgové si po poslední akci protentokrát řekli, že stíhání zločinců už bylo dost, a že by místo toho mohli začít stíhat deadliny Jarního soustředění. Oproti všem očekáváním týden s novými řešiteli proběhl poměrně organizovaně. Unavení, ale spokojení účastníci se na konci týdne vydali do svých domovů a organizátorům jen zbývalo rozvézt věci, které během týdne používali. Obzvlášť velké množství se uchovává v budově na Malé Straně, v jednom ze zdejších trezorů.

Trezorů? Ale opravdu! Kdysi budova Matfyzu sloužila jako sídlo Československé národní banky a v několika velkých trezorech pod budovou se uschovávaly státní rezervy zlata. Po přestěhování banky se ale trezory proměnily ve skladiště všemožného harampádí. Tlusté kovové dveře teď zůstávají pořád otevřené, koneckonců se od nich ztratily klíče.

Aspoň že část jednoho z trezorů patří KSPčku. Jirka s Filipem před chvílí přijeli autem do dvora malostranské budovy a naložili všechny věci ze soustředění na pojízdný vozík. „Proč mám dojem, že na každý sous toho vozíme víc a víc?“ podíval se Jirka kriticky na obsah vozíku. „Projedeme tím vůbec?“

30-5-1 Úklid po soustředku — 11 bodů

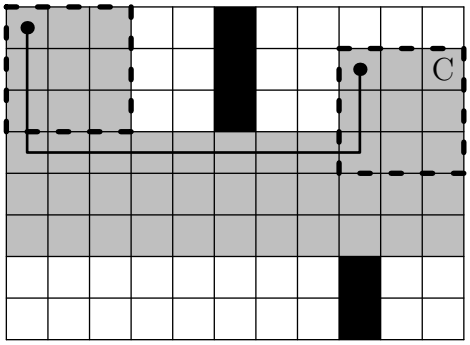
Jirka s Filipem se chystají krabice s potřebami na soustředění naložit na vozík a ten odvézt do jednoho z podzemních trezorů. Chodby k trezoru jsou však úzké a nemuseli by tam projet. Naštěstí mají k dispozici vozíky různých velikostí. Protože věcí je hodně, rádi by si vzali největší vozík, se kterým ještě projedou. Vozíky jsou čtvercové a dostupné ve všech možných velikostech: od 1×1 po velké jako celá budova (neptejte se, kde je skladují).

Prostor, v němž se orgové mohou pohybovat, si zakreslíme jako obdélníkové bludiště o M řádcích a N sloupcích, přičemž každé pole buď je, nebo není průchozí. Známe počáteční pozici vozíku (přesněji jeho levého horního rohu) a máme zakreslenou pozici trezoru (jedno políčko), do níž ho chceme dovézt.

V každém kroku mohou organizátoři posunout celý vozík o jedno políčko doleva, doprava, nahoru nebo dolů. Vždy však všechna pole vozíku musí být na průchozích polích.

Rádi bychom určili největší možnou velikost vozíku, se kterou se dá dojet do trezoru. Také bychom chtěli najít nejkratší cestu, po které s takovým vozíkem do cíle jet. Za dosažení cíle považujeme, když se libovolná část vozíku ocitne na cílovém políčku. Délkou cesty rozumíme celkový počet kroků (posunů vozíku).

Uvažume například následující bludiště:



Do cíle se dostaneme s vozíkem velkým nejvýše 3×3 a jedna z možných nejkratších cest je naznačena čarou (sleduje levý horní roh vozíku). Čárkovaně je naznačen obrys vozíku na začátku a konci trasy. Světle šedá jsou všechna políčka, na kterých se někdy vyskytla libovolná část vozíku. Nejkratší cesta má 13 kroků.

Stěhování vozíku proběhlo úspěšně a naše dvojice se ocitla v trezoru, jehož stěny byly obložené hromadou skříněk. Jirka si oddychl, utřel pot z čela a chtěl začít vykládat vozík do té, co patří KSPčku. Všimne si ale, že Filip zaujatě zírá na strop trezoru. „Nechceš mi pomoct? Takhle tu budeme

do rána...“ řekne naštvaně utahaný Jirka a pak si teprve všimne, co tam Filipa zaujalo.

Celý strop je pokreslený otazníky. Spousta otazníků, velké, malé, v různých barvách. „To je ale dementní vtip,“ řekne Jirka. „Koho tohle napadlo?“

Pak si oba uvědomí, že slyší nějaké kroky z chodby. „Je tu někdo?“ zavolá Filip. Kdyby se někdo ve sklepě nacházel, jistě by to zjistili už při příchodu. Ale u dveří do sklepení se tradičně potkali jenom s lidskou kostrou, která má za úkol vystrašit nezvané hosty.

„Je tu někdo?“ zavolá Jirka hlasitěji. A u dveří do trezoru se skutečně někdo objeví. Je to ta kostra ze vchodu! Skrze její lebku ale prosvítá něčí zašedlý obličej. A bedny uložené na vozíku se začnou otevírat a věci z nich začnou vystřelovat směrem na Jirku a Filipa!

Když Filipovi naplno dojde, že vidí oživlou kostru, ztuhnou mu nohy. Když připustí, že kostra má obličej, ztuhnou mu ruce. A v okamžiku, kdy se na něj sám od sebe začne sypat obsah vozíku, mu dojde, že teď může akorát buď omdlít, nebo začít ječet a utíkat.

Začne ječet a utíkat a následovat Jirku, který udělal to samé. Běží směrem k východu ze sklepení, jenže ten je zavřený a nejde otevřít. „Nemůžeme ty dveře odpálit?“ „Na vozíku byly výbušniny! Brali jsme je na soustředko!“

30-5-2 Útěk z trezorů — 11 bodů

I v této úloze se orgové ocitli v bludišti malostranských sklepů, ale jejich situace je o mnoho vážnější. Zjistili, že ze své pozice se nedostanou k východu. Mají ale možnost v nějaké chodbě odpálit výbušninu, čímž by se jim možná otevřela cesta k úniku.

Opět máme obdélníkové bludiště o R řádcích a S sloupcích, každé pole bludiště může, ale nemusí být průchozí. Je zadané startovní a cílové pole, mezi nimiž ovšem neexistuje přímá cesta po průchozích polích. Máme k dispozici výbušninu se silou popsanou čísly A a B. Pokud ji necháme explodovat na poli, které je přístupné ze startu, zničíme obdélník o $(2A + 1)$ řádcích a $(2B + 1)$ sloupcích, jehož střed se nachází v poli. Všechna pole v obdélníku se stanou průchozími.

Rozhodněte, které místo v bludišti odpálíte, abyste mohli projít mezi startem a cílem, a zda takové místo vůbec existuje.

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

Formát vstupu: Na prvním řádku jsou čtyři čísla R, S, A, B oddělená mezerami. Následuje R řádků o S sloupcích, na každém popis jednoho políčka bludiště: mezera značí volné políčko, `#` zeď, `^` startovní políčko a `$` cílové.

Formát výstupu: Dvě mezerou oddělená čísla: souřadnice políčka, na kterém chceme nechat explodovat výbušninu. Nejprve uvedeme číslo řádku, pak číslo sloupce, obojí počítané od 0.

Ukázkový vstup:

```
4 5 1 1
#^ ##
##
#$# #
  ###
```

Ukázkový výstup:

```
1 2
```

Po odpálení bomby na pozici $[1, 2]$ bude bludiště vypadat takto:

```
#^  #
#
#$  #
  ###
```

Správné jsou například i odpovědi `0 1` nebo `2 3`. Naproti tomu `3 1` není správná odpověď, protože toto políčko není dosažitelné ze startu.

„Stejně by to nestačilo, ta stěna je hrozně tlustá. A já už zpátky do toho trezoru nejdu... Musíme to zkusit po druhých schodech.“ S bušícím srdcem se oba vydali na druhou stranu sklepení. Aby toho nebylo málo, kvůli úsporným opatřením tady nesvítila světla.

30-5-3 Energetické úspory — 11 bodů

Vedení Matfyzu zjistilo, že fakultní budova na Malé Straně spotřebovává velké množství elektrické energie, proto se rozhodli co nejvíc osekat její spotřebu. Mimo jiné se zaměřili na sklepení budovy. To je z velké části nepoužité, stejně tu však všude svítí světla. Správci budovy vytipovali tři důležité body, mezi nimiž je nutné osvětlení zachovat (například vstup do sklepení nebo důležitý trezor).

Máte zadaný nákres sklepení budovy, skládající se ze křižovatek, které jsou propojeny chodbami. Pro každou chodbu znáte její délku v metrech. Tři křižovatky jsou označené jako důležité. Řekněte nám, které chodby nechat osvětlené, aby se mezi jakýmikoliv dvěma důležitými křižovatkami dalo dostat přes osvětlené chodby a aby součet délek osvětlených chodeb byl co nejnižší.

O struktuře chodeb nemůžete nic předpokládat – klidně se může stát, že mezi dvěma křižovatkami vede mnoho různých cest. Rovněž se chodby mohou křížit mimoúrovňově.

Jinými slovy: máte zadaný neorientovaný ohodnocený graf a v něm tři význačné vrcholy. Chcete najít nejmenší podmnožinu hran (měřeno součtem jejich délek) takovou, že mezi každými dvěma význačnými vrcholy vede cesta po hranách z této množiny.

Jirkovi s Filipem se podařilo dostat do počítačové laboratoře. Zastavili se uprostřed místnosti. Vypadala klidně, ale orgové si rychle všimli, že se nad jejich hlavami chvěje lustr. O vteřinu později se ozvalo zasvištění a nějaká neznámá síla začala srážet ze stolů počítačové displeje. „Pryč!“ Vyběhli k východu a za svými zády uslyšeli třeskot padajícího lustru. „To snad není pravda!“ Bylo už pozdě v noci a východ z budovy byl zamčený. Vyběhli po schodech, Filip se na chvíli rychle ohlédl a u vchodu do laboratoře zahlédl tu samou tvář, kterou spatřili dole ve sklepě. Tentokrát však nebyla na kostlivci, místo toho plula ve vzduchu a doširoka jí plály oči. Tvář byla zešedivělá a hrozivá, ale přesto Filip poznal, že patří Nápovědovi.

Běželi po schodech a snažili se vyhýbat těžkým věcem, které na ně duch Nápovědy (protože co jiného by to mohlo být?) házel. Zezadu slyšeli chechot a nadávky. Aniž by vůbec přemýšleli nad zastavením, doběhli až na půdu. Zavřeli se do jedné z místností a natahali ke dveřím stůl a židle, aby je nešlo otevřít.

⋆⋆⋆

Chvíli jim trvalo, než popadli dech.

„Takže první věc, duchové existují,“ pronesl Filip. „Druhá věc, je to duch chlápka, který má důvod nás nesnášet, protože jsme ho zabili. A ta třetí věc, nemáme tušení, jak si s ním poradit.“

Jirka se rozhlédl. „Tady jsem ještě nebyl a nevím, kdo tu pracuje, ale ten člověk asi moc informačním technologiím nefandí.“ V místnosti nebyl počítač, místo toho se v policích nacházela hromada kartoték. Jirku zaujala ta s nápisem Lidé a kontakty. „Papírová sociální síť? Jestli hledáš kontakt na vymítače duchů, možná by byl rychlejší internet,“ ušklíbl se Filip.

Zvenku někdo praštil do dveří a málem odsunul barikádu z nábytku.

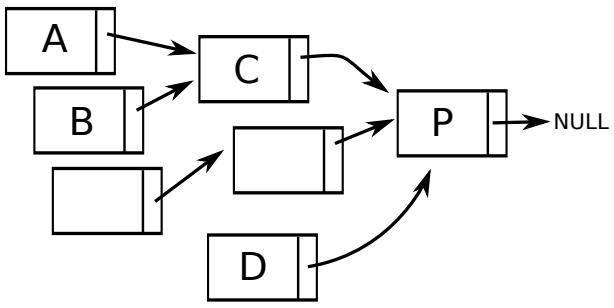
„Takže ty myslíš, že na internetu jen tak najdeš kontakt na typického vymítače duchů? Nový a přehledný web s rozesmátými obličeji, referencemi a tlačítkem Kontaktujte nás? Nebo dokonce s live chatem?“ zakroutil hlavou Jirka. „To bych spíš věřil téhle kartotéce. Kromě telefonních čísel tu je i napsáno, čím se ten člověk zabývá.“

30-5-4 Kartotéka — 10 bodů

Jeden z velmi konzervativních profesorů na Matfyzu si uchovává informace o různých osobách v papírové kartotéce. Každá karta odpovídá jedné osobě. Kromě kontaktních údajů si také profesor do jedné z kolonek na kartě zapisuje, přes jakého prostředníka se s osobou zná. Tento prostředník má v kartotéce také kartu.

Pokud zná profesor někoho osobně, tak je v kolonce napsané přímo profesorovo jméno. Samotný profesor má v kartotéce speciální kartu se svými údaji (protože je často sám zapomene). Platí, že pokud vezmeme kartu odpovídající jakékoliv osobě a následujeme odkazy na prostředníky, dojdeme do profesorovy karty.

Jeden z doktorandů profesora přemluvil, aby si kartotéku uložil do počítače. Datová struktura, reprezentující kartu, obsahuje mimo jiné i ukazatel na kartu prostředníka. Ukazatel je vždy platný s výjimkou profesorovy struktury, kde je nulový. I tady se vždy dokážeme dostat do profesorovy struktury, prostě budeme následovat ukazatele.



Máte adresy na dvě karty v operační paměti. Uvažujte řetězec z každé z nich až do profesorovy karty a najděte místo, kde tyto řetězce srůstají. Má to komplikaci – protože počítač je starý téměř stejně jako pan profesor, můžete si během výpočtu používat jen konstantně mnoho pomocné paměti. Do karet samotných nesmíte nic zapisovat.

Na obrázku představuje P profesorovu kartu. Řetězce karet A a B srůstají v C, ale A a D až v P.

I když Jirka listoval kartotékou rychle, Filip byl přece jen o něco rychlejší. „Haló? Je tam Chryzantéma Bělohlavová? Já vím, že je pozdě večer, ale právě jsme zjistili, že duchové existují! Cože?“ Filip vzal ze stolu tužku a papír a začal na něj rychle něco čmárat. „Nestihnete se sem včas dostat? A jste si jistá, že nám tohle pomůže?“, nadskočil, když se ozvalo další praštění do dveří, „Tak vám zkusíme věřit,“ zavěsil telefon.

Filip v rychlosti Jirkovi vysvětlil, co mu Chryzantéma poradila. Musí z papíru sestrojit těleso o určitých specifických rozměrech. Pro výpočet rozměrů Ghostbusteru, jak orgové těleso nazvali, je ale třeba znát největší společné dělitele některých čísel.

30-5-5 Výroba likvidátoru — 11 bodů

Ghostbuster je výsledek nejnovější vědecké práce předních ezoteriků. Jde o papírové zařízení, které dokáže porazit duchy a jakékoliv jiné nepozemské kreatury. Při výrobě je ale třeba dbát na to, aby rozměry byly co nejpřesnější. K jejich výpočtu je třeba znát největší společné dělitele řady čísel.

Jistě víte, jak je definovaný největší společný dělitel dvou přirozených čísel A a B – jde o takové největší možné číslo, které dělí A i B. Tuto definici jde ale snadno rozšířit o větší množství čísel než dva, stačí říct, že hledané největší možné číslo musí dělit všechny z nich.

Máme seznam N různých kladných celých čísel. Chceme jedno z těchto čísel vyškrtnout, aby platilo, že největší společný dělitel zbylých čísel je největší možný.

Další náraz na dveře! Jirka k nim zpátky přitlačil stůl a s poděšením sledoval Filipa, jak zápasí s papírem, nůžkami a lepidlem. „Už ho dlouho neudržím!“ „V klidu, už jsem hotový,“ ukázal Filip hotový Ghostbuster.

„A tohle nás jako zachrání?“ vytřeštil oči Jirka, když si prohlédl papírový výtvor. „Vždyť to vypadá jako rozbitej čtyřrozměrnej dopravní kužel!“

Filipa začala přemáhat nervozita. „Tak si zkus sám!“ hodil po něm Ghostbusterem. „Ani náhodou!“ mrštil ho Jirka zpátky. „Ty ses bavil s Chryzantémou.“

Obří náraz přerušil jejich konflikt. Dveře se otevřely a dovnitř vletěla pobledlá Nápovědova hlava a z očí jí šlehaly blesky. Filip, který málem vyletěl z okna, z posledních sil natáhl ruku, jako kdyby chtěl tasit meč. Jenže místo meče držel papírovou skládanku pro děti...

⋆⋆⋆

Chryzantéma Bělohlavová se dostala na místo činu a když vyběhla na půdu budovy, spadl jí kámen ze srdce. Ti dva matfyzáci byli sice notně pošramocení a vyděšení, ale nebylo to nic, co by její alternativní medicína nezvládla. Ohořelý Ghostbuster se válel uprostřed místnosti – duch Nápovědy byl zřejmě navždy pryč. Přesto ho svým šestým ezosmyslem cítila a vnímala zbytky jeho myšlení. Musela uznat, že neměl úplně jednoduché dětství. I během té záplavy vzteku ho neopustily některé nepříjemné vzpomínky ze školy.

30-5-6 Nápověda na lyžích — 13 bodů

Jako žák základní školy jel Nápověda na lyžařský kurz. Byl v té době docela obtloustlý a lyžování mu vůbec nešlo. V závěrečném testu na konci kurzu měl docela jednoduchý úkol – projet mezi několika lyžařskými brankami. Ani to se mu ale nepovedlo a spolužáci si z něj dlouho dělali legraci.

Na vstupu dostanete popis lyžařského svahu. Jde o (v matematickém smyslu) rovinu, na níž se nachází tyčky dvou druhů, východní a západní. Tyčky ale nejsou nijak spárované, netvoří žádné branky nebo něco podobného. Svah je skloněný podél osy Y, při jízdě po svahu se hodnota souřadnice snižuje.

Tlustý lyžař má za úkol jet na lyžích zeshora dolů tak, aby projel napravo od všech západních tyček a nalevo od všech

východních tyček. Musí pak jet naprosto rovně (jeho trajektorie je přímka), kvůli své tloušťce ale zabírá kruhovou plochu o průměru D.

Při jakém maximálním D ještě může lyžař projet? Neboli, jaká je maximální šířka pásu, kterým od sebe dokážeme oddělit východní a západní tyčky?

Všimla si, že Jirka s Filipem se už pomalu dostali z šoku. „Všechno v pořádku?“ zeptala se. „Až zase natrefíte na nějakého jiného ducha, tak už budete v klidu.“

„S tím máme trochu problém, nějak se pořád nemůžeme smířit s tím, že duchové skutečně existují. Až to povíme řešitelům, tak nás za to sežerou.“

„Tak od toho odveďte pozornost.“

„Ale jak? Tohle je poslední série. Nápověda je mrtvý, příběh už pokračování mít nebude, tak jak máme odvést pozornost řešitelů?“

„Říkali jste, že na soustředění berete výbušniny?“

Kuba Maroušek

30-3-7 Funkce očima assembleru — 15 bodů

Ve druhém dílu seriálu jsme se naučili základní operace s pamětí, takže již umíme napsat program, který opravdu dělá něco užitečného – například řadí čísla podle velikosti. Tentokrát se naučíme vytvářet a volat funkce. To je asi poslední důležitá konstrukce z běžných (procedurálních) programovacích jazyků, na kterou jsme se zatím nepodívali.

Pravděpodobně jste se s funkcemi ve svém programátorském životě již setkali, takže je nemusíme složitě představovat. Funkce je zkrátka blok kódu, kterému na začátku předáme vstupy (argumenty), on je zpracuje a něco z nich spočítá. Navíc může provést nějaký vedlejší efekt, například vypsat číslo na obrazovku.

Funkce nám umožňují členit si práci na menší části, takže nemusíme při programování myslet na všechny detaily najednou. A také díky nim můžeme snadno zamezit opakování kódu – pokud stejné operace potřebujeme provádět na více místech, jednoduše místo opakování celé posloupnosti instrukcí vždy zavoláme tutéž funkci.

První pokus o funkci

Jak si pořídit něco jako funkci v assembleru? Pojďme to vyzkoušet na triviálním příkladu funkce, která dostane dvě čísla a vrátí jejich součet.

Především se dohodneme, že vstupy budeme předávat v registrech `r0` a `r1` a výsledek vrátíme v `r0`. Na nějaké místo v programu napíšeme kód naší funkce a označíme jeho začátek návěštím odpovídajícím názvu funkce.

Samotné zavolání bychom pak provedli uložením parametrů do správných registrů a skokem na ono návěští. Tím ale ještě zdaleka nemáme vyhráno. Hlavní výhoda funkcí spočívá v tom, že jednu funkci můžeme zavolat z více míst v kódu. Funkce se tedy musí umět vrátit na to místo, odkud jsme ji zavolali.

Jak to pozná? Mohlo by nás napadnout funkci předat jako jeden z parametrů adresu, na kterou se má vrátit (tzv. *návratovou adresu*). To je adresa instrukce následující hned po skoku, který funkci zavolal. Funkce by tedy provedla svou práci a na závěr by jen skočila na adresu, kterou takto dostala v některém registru (musíme mít samozřejmě předem domluveno ve kterém, tak použijme třeba `r2`).

Pro tento závěrečný skok existuje instrukce `BX` *(Branch and eXchange)*. Proč má v názvu zrovna exchange, je na delší povídání a teď to není důležité. Zatím se spokojíme s tím, že jako argument bere registr a provede skok na adresu, která je v něm uložena.

Když si vzpomeneme, že při čtení registru `pc` dostaneme automaticky adresu, která je o dvě instrukce dál, dostaneme velmi jednoduchý kód, kterým funkci předáme správnou návratovou adresu. Naše první funkce bude vypadat následovně.

```
  MOV   r0, #123 @ první argument
  MOV   r1, #456 @ druhý argument
  MOV   r2, pc   @ adresa instrukce MOV r3, r0
  B     secti
  MOV   r3, r0   @ výsledek funkce uložíme do r3
  B     konec

secti:
  ADD   r0, r1   @ funkce vrací výsledek v r0
  BX    r2

konec:
```

Všimněte si nepodmíněného skoku na návěští `konec`. V našem simulátoru program skončí tím, že dojde na konec souboru. Kdybychom skok vynechali, bude se po provedení `MOV r3, r0` automaticky provádět následující instrukce, což by byla naše funkce na sčítání. Tentokrát by jí ale nikdo nepředal novou informaci o návratové adrese, takže by skočila opět na stejné místo a celý program by se tak zacyklil.

Protože volání a návrat z funkcí je velmi častá operace a nebyli jsme první, koho napadlo předávat funkci návratovou adresu v nějakém registru, je na ARMu zvykem používat k tomuto účelu registr `r14` přezdívaný také `lr` *(Link Register)*. Pak můžeme pro samotné volání použít šikovnou instrukci `BL` *(Branch with Link)*, která sama uloží správnou návratovou adresu do `lr` a pak skočí na zadané místo.

Náš příklad se sčítáním bychom pomocí `BL` mohli zjednodušit takto:

```
  MOV   r0, #123
  MOV   r1, #456
  BL    secti
  MOV   r3, r0
  B     konec

secti:
  ADD   r0, r1
  BX    lr

konec:
```

Zásobník

Všimněte si, že dosud je celý náš kód vzájemně velmi provázaný. Musíme pořád myslet na to, které registry kde na co používáme, abychom jejich obsah nedopatřením nepřepsali něčím jiným. To je s rostoucí velikostí programu čím dál těžší.

Brzy také v popsaném postupu narazíme na jednu slabinu. Jak z jedné funkce zavolat funkci jinou tak, abychom si nepřepsali návratovou adresu v link registru? Mohli bychom si ji před zavoláním funkce někam uložit. Ale kam, abychom si ji nepřepsali tam?

S řešením nám pomůže *zásobník*. Možná jste o něm již někdy něco zaslechli. V plné (informatické) obecnosti je to datová struktura, do které můžeme v nějakém pořadí ukládat

data a v opačném je zase vybírat – tedy poslední vložený prvek bude první, který vyndáme.

Zásobník si zvládneme vyrobit sami. Vyhradíme pro něj souvislou oblast paměti a dohodneme se, že ji budeme zaplňovat shora dolů (od nejvyšších adres k nejnižším). Jedním registrem si budeme ukazovat na *vrchol zásobníku,* tedy poslední vloženou hodnotu. Nenechte se mýlit tím, že vrchol zásobníku leží na nejnižší využité adrese.

Vložení prvku na zásobník pak bude spočívat ve snížení tohoto ukazatele a uložení nové hodnoty. A naopak vyndání prvku nejprve načte hodnotu a poté zvýší ukazatel na vrchol zásobníku.

Důležité je uvědomit si, že zásobník nám stačí jeden, společný pro celý program. Libovolné funkci dovolíme si na něj uložit, cokoliv potřebuje, pod jednou podmínkou: než funkce skončí, musí vždy ze zásobníku odebrat vše, co do něj sama přidala. Tím zaručíme, že když si na zásobník něco odložíme na začátku funkce, například hodnotu link registru, a následně zavoláme nějaké další funkce, tak na závěr můžeme ze zásobníku naše data zase bez problému přečíst, aniž bychom museli hledat, kde jsou.

Zbývá se domluvit, který registr budeme používat jako ukazatel na zásobník. Protože se zásobníkem samozřejmě návrháři ARMu také počítali, vyhradili pro něj registr `r13` a zavedli pro něj přezdívku `sp` *(Stack Pointer).*

K ukládání na zásobník se hodí instrukce `PUSH {`*registr*`}`. Ta odečte od `sp` 4 a na takto spočítanou adresu zapíše obsah zadaného registru. Instrukce `POP {`*registr*`}` provede inverzní operaci: z adresy uložené v `sp` přečte číslo, uloží ho do registru a nakonec zvýší `sp` o 4.

To se dá zapsat i jinými instrukcemi: Místo `PUSH {`*registr*`}` bychom mohli provést `STR` *registr*`,[sp,#-4]!` a instrukci `POP {`*registr*`}` nahradit za `LDR` *registr*`,[sp],#4`.

Skutečný `PUSH` a `POP` jsou ovšem mnohem mocnější: do složených závorek můžete napsat více registrů, například `PUSH {r0-r3,r7}`. Dodejme, že registry jsou do paměti uloženy v rostoucím pořadí čísel, ten s nejmenším číslem skončí na nejnižší adrese, tedy na vrcholu zásobníku.

Ještě dodejme, že na některých jiných architekturách procesorů se zásobník používá i k ukládání návratové adresy z funkce (místo `lr`). Obvykle existuje instrukce `CALL`, která `PUSH`ne návratovou adresu a skočí na začátek funkce, a instrukce `RET`, která `POP`ne adresu a skočí na ni.

Rekurzivní funkce: faktoriál

Dejme to všechno dohromady a pojďme si napsat funkci počítající faktoriál čísla.[1] Obdobně jako v předchozím případě dostane argument v registru `r0` a také v něm vrátí výsledek.

Funkci napíšeme s využitím rekurze – bude volat sama sebe. Faktoriál čísel 0 a 1 je jednoduchý, je to jednička. Pro všechna ostatní čísla využijeme toho, že $n! = n \cdot (n-1)!$.

Celý kód včetně volání funkce bude vypadat následovně:

```
  MOV  r0, #7    @ Spočítáme faktoriál sedmi
  BL   faktorial
  @ ... zde pokračuje hlavní program

faktorial:
  CMP   r0, #1    @ porovnáme vstup (n) s 1
  MOVLS r0, #1    @ pro 0 a 1 vrátíme 1
  BXLS  lr        @ a hned skončíme
  PUSH  {r4, lr}  @ uložíme registry na zásobník
  MOV   r4, r0    @ r4 = n
  SUB   r0, #1    @ r0 = n-1
  BL    faktorial @ r0 = (n-1)!
  MUL   r0, r4    @ r0 = n*(n-1)!
  POP   {r4, lr}
  BX    lr
```

Nejprve porovnáme argument funkce s číslem 1. Pokud je menší nebo roven, chceme vrátit jedničku. Protože návratovou hodnotu předáváme ve stejném registru, v jakém dostaneme argument funkce, stačí nám provést dvojici instrukcí `MOV r0, #1` a `BX lr` a máme hotovo. V příkladu výše se vyhýbáme dalšímu skákání a obě instrukce zapisujeme rovnou v podmíněné variantě, takže se provedou právě tehdy, kdy je $\texttt{r0} \leq 1$.

Následně uložíme na zásobník ty registry, které budeme v průběhu měnit. Protože budeme volat funkci s argumentem $n-1$, potřebujeme si někde (v našem případě v `r4`) uložit původní hodnotu z `r0`. Pak zavoláme funkci `faktorial` na o jedna menším čísle a až se vrátí, výsledek vynásobíme zapamatovanou hodnotou.

Na závěr zbývá vrátit zásobník a registry `r4` a `lr` do původního stavu a vrátit se skokem na adresu v `lr`.

Volací konvence

Při programování funkcí jsme vždy museli určit, ve kterých registrech se předává který argument a kde výsledek. Bylo by ale nešikovné pokaždé to vymýšlet znovu a u každé funkce, kterou voláme, vzpomínat, jaké má rozhraní. Proto se lidé shodli na takzvané *volací konvenci* – souboru pravidel určujících, jak se volají funkce.

Na ARMu je situace jednoduchá, protože konvence vznikla spolu s procesorem a používají ji prakticky všichni: nejen programátoři v assembleru, ale i autoři překladačů dalších programovacích jazyků. (Jinde to může být jinak: pro architekturu x86 existuje mnoho různých konvencí pro různé operační systémy a programovací jazyky.)

Význam registrů `r14` (`lr`) a `r13` (`sp`) jsme si již prozradili. První – link register – obsahuje adresu, na kterou se má funkce vrátit. Pokud tedy funkce chce zavolat jinou funkci, musí si tuto hodnotu někam uložit.

Druhý – stack pointer – ukazuje stále na zásobník a musíme jej vrátit ve stejném stavu, v jakém jsme jej dostali. To zaručíme nejlépe tím, že funkce vždy zavolá instrukci `POP` na tolik hodnot, kolik jich tam předtím sama uložila pomocí `PUSH`.

Registry `r0-r3` se používají (postupně) na předání prvních čtyř argumentů funkce a slouží i pro návratovou hodnotu. Právě kvůli využití jako argumenty, existují pro tyto registry ještě alternativní jména `a1-a4` (pozor na posun čísel o jedna). Se všemi těmito čtyřmi registry si funkce může dělat, co chce. Můžete si do nich klidně ukládat mezivýsledky a nikdo se nebude zlobit, když je nevrátíte v původním stavu.

Registry `r4-r11` jsou určené pro ukládání lokálních proměnných uvnitř funkce. Funkce je musí vrátit ve stejném stavu, v jakém je dostala. Právě díky využití registrů `r4-r11` pro lokální proměnné (*variable*) dostaly tyto registry ještě alternativní názvy `v1-v8`. Silně doporučujeme používat pouze jeden typ jmen, jinak se v číslování snadno ztra-

[1] Faktoriál z n se značí $n!$ a je roven $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.

táte. My zůstaneme u názvů s písmenem `r`. (Pro úplnost poznamenejme, že registr `r9` může mít někdy speciální význam a není pak pro funkce použitelný ani jako proměnná. V našem prostředí se to ale nestane.)

Těchto volacích konvencí jsme se drželi již v příkladu s výpočtem faktoriálu. Před použitím registru `r4` pro lokální proměnnou jsme ho uložili na zásobník. Díky tomu jsme ale měli zaručeno, že nám její hodnotu naše funkce sama nezmění.

Zbývá nám registr `r12`, čili takzvaný *Intra-Procedure call scratch register* alias `ip`. To nám říká, že se s ním během volání funkce může stát cokoli. Používá se třeba při volání funkcí z dynamicky linkovaných knihoven (k přesnému mechanismu se v našem úvodním seriálu nedostaneme). Pokud ale píšete krátkou funkci, která již nic dalšího nevolá, můžete tedy použít i tento registr zcela dle libosti.

Vynechali jsme ještě registr `r15` (`pc`), který se přirozeně posouvá s každou instrukcí, takže nemá smysl jej příliš rozebírat.

Úkol 1 [2b]: Napište funkci, která dostane dva parametry: číslo $N \geq 3$ a adresu A. Funkce přečte N-prvkové pole čísel uložené od adresy A a vrátí jako výsledek třetí největší číslo. Řiďte se volací konvencí.

Intermezzo #1: Instrukce LDM a STM

Mimochodem, instrukce `PUSH` a `POP` jsou jenom speciální případy mnohem obecnějšího mechanismu pro přenos více registrů najednou. Jelikož se nám bude za chvíli hodit, pojďme se na něj podívat pořádně.

Jedná se o osm instrukcí tvaru

`(LD|ST)M(I|D)(B|A)` *registr*`!, {`*seznam registrů*`}`

První dvě písmena říkají, zda se data budou načítat z paměti (LoaD), nebo ukládat do paměti (STore). Zadaný *registr* určuje adresu v paměti, ale ta se ještě zvýší (Increment) nebo sníží (Decrement), a to před použitím (Before) nebo po něm (After). Pokud je uveden vykřičník, adresa se následně zapíše zpět do *registr*u. Registry v seznamu jsou zpracovávány postupně, v režimu I od nejnižšího čísla k nejvyššímu, v režimu D opačně.

Například `LDMIA sp!, {r0,r1}` vezme adresu z `sp`, přečte z ní číslo do `r0`, pak ji zvýší o 4, přečte z ní číslo do `r1`, opět ji zvýší o 4 a nakonec ji uloží zpět do `sp`. Je to tedy `POP`. Podobně `STMDB sp!, {r0,r1}` je `PUSH`.

Úkol 2 [3b]: Napište co nejrychlejší funkci, která vyplní velký blok paměti nulami. Jako argumenty dostane počáteční adresu bloku a jeho velikost v bajtech. Dodržujte volací konvenci. Minimalizujte celkový počet provedených instrukcí, ale nepoužívejte nezarovnané přístupy k paměti (třeba čtení 32-bitového čísla z adresy, která není dělitelná čtyřmi).

Intermezzo #2: Daleká cesta za konstantami

Ještě si dopřejme jednu odbočku. Zatím jsme v programech používali všelijaké konstanty, ale až na konci minulého dílu jsme si přiznali, že přímo do instrukce jdou zakódovat jen ve speciálních případech. Podívejme se, co si počít, když se konstanta do instrukce nevejde.

Jedna z možností je konstantu do registru dostat nadvakrát. K tomu slouží instrukce `MOVW` *registr*`, #`*hodnota* a `MOVT` *registr*`, #`*hodnota*. Obě jsou zakódované speciálním způsobem, takže *hodnota* může být libovolné 16-bitové číslo. Přitom `MOVW` toto číslo uloží do nižších 16 bitů registru a vyšších 16 vyplní nulami, zatímco `MOVT` uloží číslo do vyšších 16 bitů a nižších 16 nechá na pokoji. Můžeme tedy udělat třeba

```
MOVW R0, #0x5678  @ R0 = 0x00005678
MOVT R0, #0x1234  @ R0 = 0x12345678
```

Kdybychom chtěli do registru dostat adresu nějakého návěští, můžeme si ji od překladače nechat rozpůlit:

```
MOVW R0, #:lower16:funkce
MOVT R0, #:upper16:funkce
```

Další způsob je říci překladači, ať konstantu uloží do paměti jako posloupnost bajtů, a pak ji odtamtud přečíst. Zkusme do programu napsat

```
.BYTE  0x12, 0x34, 0x56, 0x78
.WORD  0xABADCAFE
.ASCII "kolemdokola"
.ALIGN 4
```

Tím jsme řekli překladači, aby místo instrukcí vygeneroval nejprve čtyři bajty s určenými hodnotami, pak určené 32-bitové slovo, posloupnost bajtů kódujících znaky řetězce (pozor, není ukončena nulovým bajtem) a nakonec zarovnání na adresu dělitelnou čtyřmi – to je potřeba, pokud za tímto blokem následují ještě nějaké instrukce, ty musí být zarovnané. Všimněte si tečky na začátku: ta překladači říká, že nemá očekávat instrukci assembleru, nýbrž *direktivu* překladače.

Pro přečtení konstanty se hodí jedna speciální forma instrukce `LDR`. Ta se v assembleru píše jako

`LDR` *cílový-registr*`,` *zdrojová-adresa*

ale ve skutečnosti se přeloží na adresu vypočítanou sečtením `pc` s malým offsetem uloženým v instrukci. Můžeme se pomocí ní tedy snadno odkazovat na data uložená v paměti blízko instrukce, která je používá. Samozřejmě si musíme dávat pozor na to, aby se procesor data nepokoušel provádět jako instrukce. Šikovné místo pro data tudíž leží za instrukcí `BX` uzavírající funkci.

Pojďme se podívat na příklad: následující funkce vynásobí svůj argument rychlostí světla.

```
nasob_c:
  LDR  R1, rychlost_svetla
  MUL  R0, R1    @ výsledek = argument * R1
  BX   lr
rychlost_svetla:
  .WORD 299792458  @ m/s
```

Jako konstantu můžete použít i adresu nějakého návěští, např. `.WORD nasob_c`.

Existuje milá zkratka:

`LDR` *cílový-registr*`, =`*konstanta*

Ta načte do registru danou konstantu, která může být libovolné 32-bitové číslo (případně návěští). Proč jsme vymýšleli kolem konstant takové komplikace, když se to dá zařídit jednou instrukcí? On je to tak trochu podvod a tato zkratka není instrukce. Ve skutečnosti se `LDR r0, =0x12345678` přeloží na:

```
  LDR r0, const_1
  ...
const_1:
  .WORD 0x12345678
```

Jen vám kompilátor sám najde pro uložení konstanty v programu vhodné místo (typicky taky za koncem funkce).

Kam s ním? aneb zase zásobník

Když jsme popisovali volací konvenci, tiše jsme předpokládali, že argumenty funkce jsou nejvýše čtyři a každý z nich se vejde do 32-bitového registru. Co si počneme, když na tato omezení narazíme?

S rozměrnějšími datovými typy si poradíme snadno: předáme je rozsekané ve více registrech. To by se mohlo hodit, kdybychom chtěli počítat třeba s 64-bitovými čísly.

Horší je, když nám dojdou registry vyhrazené pro předávání argumentů (přeci jenom, jsou jenom čtyři). Tehdy konvence káže předat zbývající argumenty na zásobníku, a to tak, aby první z nich byl uložený na nejnižší adrese. Pokud je tedy budeme ukládat instrukcí `PUSH`, musíme začít posledním argumentem.

Zavolaná funkce si argumenty ze zásobníku přečte, ale ponechá je tam (k tomu se hodí instrukce `LDMIA` bez vykřičníku). Odstranění ze zásobníku má na starost volající, až mu funkce zase předá řízení. Tím pádem se funkce nerozbije, pokud jí někdo předá více parametrů, než čekala.

Podobně si poradíme, pokud naše funkce potřebuje víc lokálních proměnných, než se jí vejde do registrů. Prostě si vyhradí místo na zásobníku odečtením od `sp` a do tohoto místa ukládá své proměnné, které adresuje relativně k `sp`. Také místo může využívat na argumenty podřízených funkcí. Než se vrátí, uvede `sp` do původního stavu.

Někdy je nepohodlné, že `sp` se neustále mění při dočasném odkládání hodnot na zásobník různými `PUSH`i. Tehdy může být příjemnější zkopírovat si na začátku funkce `sp` do nějakého jiného registru a adresovat pomocí něj (na kladných offsetech jsou pak argumenty, na záporných lokální proměnné). Často se k tomu používá registr `r11`, říká se mu *frame pointer* a v této roli se značí `fp`.

Podívejme se na příklad funkce s argumenty $x_1, \ldots, x_6$, která vrátí největší z x_4, x_5, x_6. Hodí se nám použít podmíněnou variantu instrukce `MOV`.

```
  MOV   r0, #5        @ x_5
  MOV   r1, #6        @ x_6
  PUSH  {r0,r1}       @ na zásobníku x_5 a x_6
  MOV   r0, #1        @ x_1
  @ podobně r1-r3 pro x_2 až x_4
  BL    max456
  ADD   sp, #8        @ smažeme x_5 a x_6
  @ ... zbytek programu

max456:
  PUSH  {r4,r5,fp}    @ uložíme, co budeme měnit
  ADD   fp, sp, #12   @ fp ukazuje na původní sp
  LDMIA fp, {r4,r5}   @ přečteme x_5 a x_6
  MOV   r0, r3        @ x_4
  CMP   r0, r4        @ x_5
  MOVLO r0, r4
  CMP   r0, r5        @ x_6
  MOVLO r0, r5
  POP   {r4,r5,fp}
  BX    lr
```

Úkol 3 [3b]: Napište funkci s proměnlivým počtem argumentů, která vrátí součet všech svých argumentů. Jelikož nemůže poznat, kolik argumentů dostala, zastaví se na prvním nulovém argumentu.

Krutá pravda o našem simulátoru

Nemůžeme to déle skrývat, pravda musí vyjít najevo: náš simulátor assembleru (`http://ksp.mff.cuni.cz/viz/asm`) je ve skutečnosti překladačem Céčka, do kterého jsou jenom vložené assemblerské instrukce (pomocí GCCčkové direktivy `asm`). Okolo assemblerských instrukcí se nachází jednoduchá obálka, která jenom posbírá hodnoty z registrů a předá je k vypsání obyčejné Céčkové funkci `printf`.

Takové `printf` je typickým příkladem funkce s proměnlivým počtem argumentů. Jako první argument dostane formátovací řetězec (přesně řečeno ukazatel na místo v paměti, kde začíná posloupnost znaků ukončená nulovým bajtem). Tento řetězec vypíše a kdykoliv v něm narazí na nějakou kombinaci znaků začínajících procentem, vypíše místo ní další argument. Speciálně `%d` nechá vypsat číslo v desítkové soustavě, `%x` šestnáctkové číslo a `%s` řetězec.

Nejspíš už vás to napadlo, ale pro jistotu dodáme, že název libovolné Céčkové funkce lze použít jako návěští a dá se s ním dělat cokoli, co s assemblerovými návěštími – např. na něj skočit.

Úkol 4 [3b]: Když je simulátor takový švindl, který si jen tak volá Céčkové `printf`, pojďme si ho ze simulovaného assembleru také zavolat. Napište program, který v simulátoru vypíše 100 číslovaných řádků s textem „Nebudu si číst pod lavicí popis instrukční sady ARMu.“.

V králíka se proměň!

Nakonec se podíváme na jednu specialitu: programy, které modifikují samy sebe. Teoreticky to není nic podivného – stejně jako jakákoliv data, program je v paměti také uložen jako posloupnost bajtů, takže do něj jde zapisovat (tedy pokud nám to operační systém nezakáže). V praxi se toho často využívá: nahráváme-li nový program z disku do paměti, nebo třeba když debugger potřebuje do programu umístit breakpoint.

Pojďme si to také vyzkoušet, ale abychom se vyhnuli záhadným chybám, prozradíme, že mezi vytvořením instrukcí v paměti a jejich spuštěním musí proběhnout dvojice instrukcí `DSB` a `ISB`. Ty zabrání situacím typu „procesor si instrukce přečetl v předstihu a začal je provádět, aniž si všiml, že se pak ještě změnily“. Též připomínáme, že cílová adresa v instrukci skoku je kódovaná relativně vůči její vlastní adrese, takže pokud takovou instrukci překopírujete jinam, bude skákat jinam.

Úkol 5 [4b]: Napište program pro simulátor, který v paměti upraví funkci `printf` tak, aby vypisované řádky číslovala. Prozradíme vám, že tato funkce začíná instrukcí `PUSH {r0-r3}`.

Jenda Hadrava & Martin Mareš

30-4-7 Překřikující se procesory — 15 bodů

Vývoj počítačů se žene vpřed. Toužíme po čím dál rychlejších procesorech, ale stále víc při tom narážíme na technické limity, či spíš rovnou na fyzikální zákony. Řešením může být pořídit si místo čtyřikrát rychlejšího procesoru čtyři stejně rychlé. Ostatně, většina dnešních počítačů tak vypadá – mluví se sice obvykle o více jádrech, ale minimálně z pohledu programátora to jsou samostatné procesory.

Programovat několik procesorů, které se všechny najednou snaží pracovat se společnými daty, je velké dobrodružství. Pojďme si ho v posledním dílu našeho seriálu vyzkoušet.

Tu a tam si sice realitu trochu zjednodušíme, ale všechny zásadní radosti a strasti paralelního světa při tom okusíme.

Pořídíme si počítač s několika ARMovými procesory, které mají společnou paměť. Jelikož program je uložen v paměti, znamená to, že také maji společný program. Každý procesor má ale svou sadu registrů včetně `sp` a `pc`, takže může vykonávat svou část programu a mít k tomu svůj zásobník. Takovému uspořádání se říká *symetrický multi-processing* neboli SMP.

Poněkud naivní pokus

Začneme triviálním problémem: chceme si pořídit *počítadlo* a zvyšovat ho, kdykoliv v programu kteréhokoliv z procesorů nastane určitá událost. Pojďme si tedy pořídit nějaké místo v paměti, uložit do něj 4-bajtové počítadlo a zvyšovat ho následující posloupností instrukcí:

```
@ v r0 očekáváme adresu počítadla
LDR   r1, [r0]
ADD   r1, #1
STR   r1, [r0]
```

Hotovo. Co by se mohlo stát špatně? Inu, ledacos...

Představme si třeba následující scénář. V paměti je hodnota 100. Procesor A si řekne, že počítadlo zvýší, tak do `r1` přečte 100 a přičte 1. Než stihne zapsat `r1` zpět do paměti, procesor B se také pokusí zvýšit počítadlo. (To se může stát, protože rychlost provádění programu závisí na spoustě okolností, takže někdy je jeden procesor rychlejší než druhý.) Procesor B také přečte 100 a vzápětí zapíše 101. Teď se dostane ke slovu opět procesor A, ale vůbec neví, co se mezitím stalo, takže zapíše tu 101, kterou má stále v `r1`. Místo dvou zvýšení počítadla tedy nastalo jen jedno.

Mohlo by to být i horší: co kdyby se hodnoty do paměti zapisovaly po částech, například po jednotlivých bajtech? Při přechodu počítadla z `0x0000ffff` na `0x00010000` by mohla být chvíli v paměti hodnota `0x0001ffff` a jiný procesor by ji mohl stihnout přečíst. To se naštěstí na ARMu nestane: máme zaručeno, že instrukce `LDR` a `STR` pracující se správně zarovnanými daty jsou takzvaně *atomické.* To znamená, že vždy provedou celou operaci najednou a ostatní procesory ji vidí buďto ještě neprovedenou, nebo už zcela provedenou.

Exkluzivní přístupy do paměti

Náš předchozí pokus tedy nefungoval hlavně proto, že mezi přečtením počítadla a jeho zápisem zpět mohl jiný procesor počítadlo změnit. ARM nám naštěstí umožňuje si této nečekané změny počítadla všimnout.

Existuje instrukce `LDREX` (LoaD Register EXclusive), která se chová stejně jako `LDR`, ale navíc si procesor zapamatuje, že má adresu, z níž se četlo, hlídat. Sleduje pak ostatní procesory, jestli některý z nich také neudělal `LDREX` ze stejné adresy.

Když pak místo instrukce `STR` použijeme `STREX` na hlídanou adresu, zápis do paměti se provede jen tehdy, pokud k adrese mezitím nikdo jiný nepřistoupil. V opačném případě se dozvíme, že se zápis nepovedl. Instrukce `STREX` má proto tři parametry: chybový registr (do něj se uloží 0 v případě úspěchu a 1 při neúspěchu), zdrojový registr a cílovou adresu v paměti.

V našem příkladu s počítadlem tedy můžeme zjistit, že se počítadlo během našeho zvyšovaní změnilo, a v takovém případě přečtenou hodnotu zahodit a pokus o zvýšení zopakovat od začátku. Vypadat to bude takto:

```
  @ v r0 očekáváme adresu počítadla
pokus:
  LDREX r1, [r0]
  ADD   r1, #1
  STREX r2, r1, [r0]
  CMP   r2, #0
  BNE   pokus
```

Pokud bychom chtěli přistupovat k jednotlivým bajtům nebo 16-bitovým číslům, existuje také `(LD|ST)REX(B|H)`. Také můžeme použít 64-bitovou variantu `(LD|ST)REXD`, která čte/ukládá dva registry po sobě (počáteční adresa musí být dělitelná 8).

Dodejme ještě, že procesor umí v jeden okamžik hlídat jenom jednu adresu. Pokud provedeme `LDREX` v okamžiku, kdy už nějaká adresa byla hlídaná, předchozí hlídání se zruší. Pokud se pokusíme o `STREX` na adresu, která zrovna není hlídaná, skončí neúspěšně. Pakliže nějaký jiný procesor přistupuje k hlídané adrese jiným způsobem než pomocí těchto instrukcí, není zaručeno, že si jich hlídací mechanismus všimne.

Zámky

Někdy se hodí trochu obecnější přístup: říci o nějakých datech (v našem případě počítadle, ale může to být třeba celá složitá datova struktura), že s ní smí v jeden okamžik zacházet nejvýše jeden procesor. K tomu se hodí pořídit si *zámek* neboli *mutex* (z anglického mutual exclusion – vzájemné vyloučení).

Zámek je objekt, který má dva stavy: *odemčeno* a *zamčeno.* Na počátku je v odemčeném stavu. Chce-li nějaký procesor pracovat s chráněnými daty, pokusí se zámek uzamknout. Pakliže byl v tomto okamžiku zámek odemčený, stane se zamčeným a program hned pokračuje. Až bude operace s daty hotova, program zámek odemkne a tím data zpřístupní ostatním procesorům. Pokud byl při pokusu o zamykání zámek uzamčen jiným procesorem, program čeká, dokud jiný procesor zámek neodemkne.

Nejjednodušší forma zámku je takzvaný *spinlock.* V paměti bude uložen jako 4-bajtová proměnná. V odemčeném stavu v ní bude 0, v zamčeném 1.

Zamčení spinlocku můžeme provést atomicky pomocí `LDREX` a `STREX`:

```
zamkni:   @ r0 = adresa zámku
  LDREX r1, [r0]
  CMP   r1, #0
  BNE   zamkni       @ už bylo zamčeno
  MOV   r1, #1
  STREX r2, r1, [r0]
  CMP   r2, #0
  BNE   zamkni       @ někdo nás předběhl
```

Odemčení je triviální: stačí do proměnné zapsat nulu obyčejným `STR` – náš procesor je jediný, který v daný okamžik může spinlock odemykat, takže se s nikým nepředháníme.

```
odemkni:  @ r0 = adresa zámku
  MOV   r1, #0
  STR   r1, [r0]
```

Náš příklad s počítadlem by se tedy dal naprogramovat stylem „zamkni spinlock, zvyš počítadlo, odemkni spinlock“. Program sice vyjde delší, ale snáz se o něm přemýšlí. Proto se spousta problémů se sdílením dat mezi procesory řeší právě pomocí zámků. Je ale potřeba dát si pozor na několik věcí:

- Předně nechceme zámek udržovat zamčený příliš dlouho – obvykle jenom na dobu nějaké elementární operace s daty. Jinak by totiž jednotlivé procesory většinu času trávily jenom čekaním na zámky.
- Podobně nechceme všechna data chránit jedním společným zámkem. Raději si pro každou datovou strukturu pořídíme samostatný zámek. Na druhou stranu nechceme zámek přidávat ke každé trivialitě, protože pak by nám polovinu paměti zabraly zámky.
- Nikdo nám nezaručuje, za jak dlouho se zamčení zámku povede. Uvažujme třeba tuto situaci: máme dva procesory, které se současně pokoušejí o zamčení téhož spinlocku. První z nich provede `LDREX`, pak druhý také `LDREX`. Obě `STREX`, ať už v libovolném pořadí, pak nutně selžou. Oba procesory tedy svůj pokus zopakují od začátku, načež opět selžou, a tak dále. Tento problém nás naštěstí v praxi neohrožuje, protože provádění programu závisí na spoustě vnějších okolností, takže se oba procesory po několika pokusech rozejdou natolik, že už si nebudou překážet. Exaktně dokázat to nicméně nedovedeme, aspoň s naším primitivním modelem procesoru.

Úkol 1 [3b]: Často se nám hodí dovolit několika procesorům číst společná data, ale nechceme do nich ani paralelně zapisovat, ani současně zapisovat a číst. Naprogramujte *read-write spinlock.* Ten má 3 stavy: odemčeno, zamčeno pro čtení a zamčeno pro zápis. V každém okamžiku musí být buďto odemčený, nebo zamčený pro čtení na libovolně mnoha procesorech, případně zamčený pro zápis na právě jednom procesoru.

Bariéry

Přístup „zamknu, modifikuji, odemknu“ se nám sice osvědčil, ale tak, jak jsme ho popsali, by nefungoval spolehlivě. Byl totiž založený na představě procesoru, který vykonává instrukce pěkně po pořadě. Skutečný procesor ale trochu „švindluje“ – sice tak, aby běžící program nemohl nic poznat, leč z jiných procesorů to už poznat může být.

Podívejme se třeba na tuto posloupnost instrukcí:

```
MOV   r1, #0        @ 1
STR   r1, [r0]      @ 2
STR   r1, [r0, #4]  @ 3
LDR   r1, [r0, #8]  @ 4
STR   r1, [r0]      @ 5
```

Instrukce 2 a 3 ukládají na dvě různá místa v paměti. Kdyby procesor zápisy do paměti provedl v opačném pořadí, program by nemohl nic poznat, protože mezitím žádná data z paměti nečte. Procesor toho může využít, pokud mu z nějakého důvodu přijde prohozená varianta lepší než původní.

Podobně může procesor zápis do paměti o pár instrukcí posunout: program by například nepoznal, kdyby se zápis z instrukce 1 provedl až mezi instrukcemi 4 a 5. Za instrukci 5 ho ale posunout nemůžeme, protože ta zapisuje na totéž místo v paměti.

Procesor by si nicméně mohl všimnout, že data zapsaná instrukcí 2 jsou přepsaná instrukcí 5 a mezi tím je nikdo nepřečte. Proto by mohl zápis z instrukce 2 úplně zrušit.

Konečně v instrukci 4 čteme data, do kterých žádná z předchozích instrukcí nemůže zapisovat, takže procesor může čtení provést už dříve. To je také běžná optimalizace: paměť je pomalá, takže může pomoci zadat jí požadavek na čtení dat v předstihu, aby v okamžiku, kdy data budeme potřebovat, už byla připravena.

Obecně platí, že procesory mohou přístupy do paměti libovolně přeházet, dokud si jsou jisté, že tím neovlivní chod programu. O chodu programu na jiných procesorech ovšem nic nevědí, takže jejich „ekvivalentní úpravy“ programu nakonec mohou uškodit.

Uvažujme následující program:

1. Zamkneme spinlock S
2. *počítadlo* $\leftarrow$ *počítadlo* $+ 1$
3. Odemkneme spinlock S

Uvnitř operací se spinlocky se na proměnnou *počítadlo* nesahá, takže procesor může přesunout její čtení i zápis mimo oblast chráněnou spinlockem a tím celý náš pečlivě budovaný mechanismus ochrany vyřadit.

To je samozřejmě nanejvýš nepraktické, proto existují takzvané *bariéry* – to jsou instrukce, přes které není dovoleno přehazovat některé typy přístupů do paměti. My si vystačíme s univerzální bariérou, což je instrukce `DMB` (Data Memory Barrier). Přes ni není dovoleno přehodit žádné čtení ani zápis.

V našem příkladu bychom tedy chtěli jednu bariéru umístit mezi kroky 1 a 2 a druhou mezi 2 a 3. To je tak častý obrat, že obvykle bariéry zabudováváme přímo do implementace zámků: na konec zamykání a na začátek odemykání.

Bariéry se mohou hodit i v jiných případech. Představte si, že budujete spojový seznam nějakých položek. Při vkládání nejprve vyplníte položku (obyčejnými neatomickými instrukcemi, protože je to mnohem jednodušší) a pak ji atomicky vložíte do seznamu. Tehdy ovšem hrozí, že ostatní procesory narazí při čtení seznamu na ukazatel na novou položku, ale když se do této položky podívají, ještě v něm neuvidí správný obsah. Proto je potřeba mezi vyplnění položky a její zařazení do seznamu přidat bariéru.

Paralelní seznamy

Ukažme si několik příkladů paralelní práce se seznamy. Začněme jednoduše: pořídíme si více jednosměrných spojových seznamů. Seznam bude mít hlavičku, ve které si bude pamatovat ukazatel na první prvek a zámek, který chraní všechna data seznamu. Kdykoliv chceme se seznamem provést nějakou operaci, nejprve zamkneme zámek, pak provedeme operaci a nakonec zámek odemkneme. To zaručí, že každá operace bude korektní (procesory si nebudou navzájem přepisovat ukazatele v seznamech), a přitom půjde současně pracovat s různými seznamy.

Problém nastane, pokud se pokusíme atomicky přehodit prvek ze seznamu A do seznamu B (atomicitou myslíme, že pro správně zamykajícího vnějšího pozorovatele bude v každém okamžiku tento prvek v právě jednom seznamu). Mohli bychom to udělat tak, že nejprve zamkneme zámek seznamu A, pak zámek seznamu B, načež prvek přepojíme a nakonec oba zámky odemkneme.

To je triviální … a špatně. Rozbije se to, pokud se jeden procesor bude snažit přehodit prvek z A do B a současně s tím jiný procesor nějaký jiný prvek z B do A. První procesor si stihne zamknout A a než stihne zamknout B, zamkne si ho druhý procesor. Nyní první procesor čeká na B a druhý na A, ale ani jeden z nich se nedočká odemčení. Tomuto stavu se říká *deadlock* (česky poněkud méně elegantně *uváznutí*).

Podobně by deadlock mohl nastat, pokud by se první procesor pokoušel přehazovat prvek z A do B, druhý z B do C a třetí z C do A. Možnosti, jak se mohou věci pokazit, jsou zkrátka nepřeberné :)

Úkol 2 [4b]: Naprogramujte atomické přehazování prvků mezi seznamy tak, aby nemohlo dojít k deadlockům. Správnost se pokuste dokázat.

Podívejme se ještě na dva úkoly na atomickou práci se seznamy. Nemusíte v něm rozepisovat všechny detaily do instrukcí, stačí dostatečně podrobný pseudokód. Operace se zámky, přístupy do paměti a bariéry z něj nicméně musí být jasné.

Úkol 3 [4b]: Naprogramujte *frontu:* jednosměrný spojový seznam, který si pamatuje ukazatel jak na začátek, tak na konec. Má umět operace „přidej na konec“ a „odeber ze začátku“. Přitom chceme, aby práce se začátkem a s koncem mohla probíhat současně (není-li zrovna fronta příliš krátká).

Úkol 4 [4b]: Naprogramujte *seznam počítadel:* jednosměrný spojový seznam dvojic (*klíč*, *počet*), kde všechny klíče jsou různé. Má umět dvě operace: „zvyš počet pro daný klíč a pokud ještě neexistovala dvojice s tímto klíčem, založ ji“ a „sniž počet pro daný klíč a pokud se snížil na nulu, dvojici odstraň“. Obě operace vrací novou hodnotu počtu. Snažte se, aby operace mohly probíhat co nejvíce paralelně.

Virtuální realita

V celém seriálu jsme se věnovali tomu, jak se programuje v assembleru, pokud nám do toho nikdo další nemluví. Skutečnost ale bývá komplikovanější: na počítači obvykle běží více programů, někdy i patřících více uživatelům, a místo aby se bavily přímo s hardwarem, bdí nad nimi *operační systém* a hlídá, aby se programy o hardware nepřetahovaly a navzájem si neškodily.

Fungování operačních systémů by vydalo na samostatný seriál (třeba někdy…), ale pokusme se na závěr alespoň pár věcí naznačit.

Procesory především mají dva různé režimy: *uživatelský* a *systémový.* Po zapnutí se procesor ocitne v systémovém režimu a začne vykonávat instrukce operačního systému. V tomto módu procesor není chod programu nijak omezen. Když chce časem spustit nějaký obyčejný program, nahraje ho na správné místo do paměti, vyhradí mu místo na zásobník, nastaví počáteční hodnoty registrů a přepne se do uživatelského režimu. V tom jsou některé věci zakázány a je přístupná pouze část paměti (například obvykle nelze přímo ovládat různá vstupně-výstupní zařízení počítače).

Musí tu ale být možnost, jak se dostat zpátky do systémového režimu – program může skončit, nebo může třeba chtít po operačním systému, aby mu přečetl nějaká data z disku. Nelze ovšem uživatelskému programu dovolit, aby jen tak přepnul režim a začal vykonávat své instrukce v systémovém režimu. Proto existuje mechanismus *systémových volání.* Na ARMu existuje speciální instrukce `SVC` (SuperVisor Call), která přepne do systémového režimu a začne vykonávat kód od adresy, kterou si předtím operační systém nastavil (v paměti, do které uživatelský kód neměl právo zapisovat).

Často také chceme oddělit nejen systémovou paměť od uživatelské, ale také oddělit data jednotlivých uživatelských programů. K tomu se používá *virtualizace paměti.* Uživatelské programy pak neadresují fyzickou paměť, nýbrž se procesor nastaví tak, aby všechny adresy překládal podle speciální tabulky. Program tedy přistupuje k nějakým virtuálním adresám a tabulka říká, jak se tyto adresy přeloží na fyzické (případně může být řečeno, že nějakou část paměti je třeba dovoleno pouze číst). Tabulku přitom systém nastaví každému uživatelskému programu jinak, takže programy žijí ve svém virtuálním světě a o ostatních programech nevědí (případně s nimi mohou komunikovat skrz operační systém).

Uživatelských programů, které chceme spouštět, je mnohdy víc, než má náš počítač fyzických procesorů. I tady nahradíme skutečnost vhodnou iluzí. Systém si udržuje libovolně mnoho *procesů,* které by chtěly běžet. Každý proces má přidělenou nějakou paměť (a vytvořenou překladovou tabulku ze svých virtuálnich adres na fyzické), v ní má své instrukce, svá data a svůj zásobník, a navíc si u něj systém pamatuje aktuální stav registrů.

Součástí systému je pak *plánovač* neboli *scheduler,* který rozhoduje, kdy má který proces běžet a na kterém procesoru se spustí. Pokud je procesů víc než procesorů, každý proces nechá běžet jenom chvíli, než se buďto sám rozhodne zastavit, nebo uplyne vhodná doba. Pak místo něj plánovač spustí další proces, přičemž se snaží přidělovat procesům procesory nějak spravedlivě.

Pokud tedy chceme programovat paralelně, obvykle systému neříkáme, co má spustit na kterém procesoru, ale prostě si pořídíme více procesů se společnou pamětí (těm se obvykle říká *vlákna* čili *threads*) a necháme plánovač, ať je rozhazuje mezi procesory, jak se mu hodí.

Při tomto přístupu je samozřejmě dost nešikovná naše implementace zámků pomocí spinlocků – pokud zamkneme spinlock, načež nás plánovač přeruší a spustí jiný proces, který by také chtěl tento spinlock, onen jiný proces pořád čeká ve smyčce na něco, čeho se před dalším přeplánováním nemůže dočkat. Místo toho by bylo lepší, kdyby se při neúspěšném pokusu o zamčení zámku vzdal procesoru a nechal se probudit až v okamžiku, kdy je zámek odemčen. Tomu se říká *pasivní čekání* (na rozdíl od *aktivního* u spinlocků). Operační systém vám obvykle dodá implementaci zámků, která čeká pasivně.

Ale to už je opravdu jiný příběh. Pojďme odemknout všechny zámky, přeplánovat a spustit proces prázdniny…

PS: Vlákna v simulátoru

Abyste si své výtvory mohli vyzkoušet, přidali jsme do simulátoru podporu více vláken. Ta je aktivována, pokud máte v programu návěští ve tvaru `thread`*číslo*, např. `thread5`. Pro každé takovéto návěští simulátor vytvoří jedno vlákno, které na začátku skočí na dané návěští a dál vykonává kód od tohoto místa. Protože jde o vlákna spravovaná operačním systémem, nemůžeme zaručit, že kód poběží současně a na různých procesorech. Pokud vlákna poběží krátce, může se například stát, že se nejdřív provede celé první vlákno a potom celé druhé.

O ukončení vlákna se musíte postarat sami, např. skokem na konec. Program skončí po doběhnutí všech vláken nebo uplynutí časového limitu 15 s. Vícevláknový mód nevypisuje na konci automaticky obsah registrů (protože každé vlákno má svoje registry a nebylo by to moc přehledné). Pokud potřebujete nějaké ladící výpisy, můžete si třeba zavolat `printf`.

Martin Mareš

Recepty z programátorské kuchařky: Teorie čísel

Dnes si budeme povídat o různých užitečných vlastnostech celých čísel, především o dělitelnosti a kongruencích. Mohlo by se zdát, že to nemá s informatikou nic společného, ale překvapivě v informatice zakopáváme o teorii čísel takřka na každém kroku. Někdy se jedná o hledání velkých prvočísel, jindy o rychlé násobení čísel s miliony cifer nebo o všudypřítomnou asymetrickou šifru RSA.

Začneme vyjasněním základních pojmů, postupně se prokoušeme kongruencemi k hledání největšího společného dělitele a Bézoutových koeficientů, chvilku se zamyslíme nad prvočísly a nakonec si také ukážeme, jak to všechno souvisí s čínskou armádou.

Definice na úvod

Množinu celých čísel si označíme $\mathbb{Z}$ a každé její podmnožině $\{0, 1, \ldots, n-1\}$ budeme říkat $\mathbb{Z}_n$.

Často nás bude zajímat *dělitelnost:* $a \backslash b$ (nebo $a \mid b$) budeme značit, že číslo a je dělitelem čísla b (nebude-li hrozit mýlka, čteme prostě „a dělí b“).

Pro *největšího společného dělitele* dvou čísel zavedeme symbol $\mathrm{nsd}(a, b)$. Pokud $\mathrm{nsd}(a, b) = 1$, říkáme, že čísla a a b jsou *nesoudělná,* zkráceně $a \perp b$. Když budou naopak a a b soudělná, napíšeme $a \parallel b$. Podobně nejmenší společný násobek dvou čísel označíme $\mathrm{nsn}(a, b)$ a všimneme si, že je roven $a \cdot b \,/\, \mathrm{nsd}(a, b)$.

Často nás může zajímat největší společný dělitel a nejmenší společný násobek více než dvou čísel. Není těžké si rozmyslet, že $\mathrm{nsd}(a, b, c) = \mathrm{nsd}(a, \mathrm{nsd}(b, c))$, $\mathrm{nsd}(a, b, c, d) = \mathrm{nsd}(a, \mathrm{nsd}(b, \mathrm{nsd}(c, d)))$ a obecně $\mathrm{nsd}(a_1, \ldots, a_k) = \mathrm{nsd}(a_1, \mathrm{nsd}(a_2, \ldots, a_k))$. Obdobný vztah platí pro nejmenší společný násobek.

Není-li jedno číslo dělitelné druhým, znamená to, že při celočíselném dělení vznikne zbytek: například pokud vydělíme $23/8$, dostaneme zbytek 7, protože $23 = 8 \cdot 2 + 7$. Obecně *zbytkem po dělení* a/b nazveme hodnotu z v rovnici $a = b \cdot x + z$, kde x je celé číslo a z je nezáporné celé číslo menší než b. Obvyklé programovací jazyky mívají takovouto operaci zabudovanou a říkají jí *modulo.* Programátoři počítají zbytky po dělení často nějakou konstrukcí podobnou `z = a % b`, zatímco matematici spíše píší $z = a \bmod b$.

Dodejme, že pro záporná čísla už není definice zbytku po dělení tak jednoznačná: v některých programovacích jazycích je `(-7)%3 = -1`, jiné se shodnou s naší definicí na tom, že vyjde `2`. Přitom oba výsledky vycházejí z jedné rovnice $a = b \cdot x + z$. Aby měla jednoznačné řešení, požadovali jsme $0 \leq z < b$. Lze na to ovšem jít i jinak: řekneme, že x má být celočíselný podíl a/b. Ten jde ale definovat dvěma způsoby: buď se zaokrouhlením dolů (což se shodne s naší definicí), nebo se zaokrouhlením k nule (to dělá většina procesorů), což dá pro záporné a záporný zbytek. Zkuste zjistit (a vysvětlit), jak je to ve vašem oblíbeném jazyce a co se stane, když je záporné i číslo, kterým modulíme.

Kongruence

Když čísla p a q dávají stejný zbytek po dělení číslem m, píšeme

$$p \equiv q \pmod{m}$$

a čteme „p je *kongruentní* s q modulo m“. To platí právě tehdy, je-li rozdíl $p - q$ dělitelný m.

Zápis kongruence tak trochu připomíná rovnici. To není náhoda – kongruence totiž můžeme upravovat podobně jako rovnice.

Součet dvou kongruencí: Pokud $a \equiv A$ a $b \equiv B$, pak také platí $a + b \equiv A + B$ (to vše modulo totéž m). Že je to pravda, nahlédneme snadno. Napišme si a jako $n_a \cdot m + z_a$ a čísla A, b, B obdobně. Pak dostaneme:

$$\begin{aligned} a + b &= (n_a \cdot m + z_a) + (n_b \cdot m + z_b) = \\ &= (n_a + n_b) \cdot m + (z_a + z_b), \\ A + B &= (n_A \cdot m + z_A) + (n_B \cdot m + z_B) = \\ &= (n_A + n_B) \cdot m + (z_A + z_B). \end{aligned}$$

Protože $a \equiv A$ a $b \equiv B$, musí být $z_a = z_A$ a $z_b = z_B$. Můžeme si tedy všimnout, že rozdíl mezi $a + b$ a $A + B$ je $((n_a + n_b) - (n_A + n_B)) \cdot m$. To je násobek m, takže $a + b \equiv A + B$.

Rozdíl dvou kongruencí: Nahlédneme obdobně.

Přičtení téhož čísla k oběma stranám: Pokud $a \equiv A$, pak platí $a + k \equiv A + k$ pro libovolné k. Stačí totiž přičíst evidentně platnou kongruenci $k \equiv k$.

Přičtení násobku m k jedné straně: Z $a \equiv A$ plyne $a + k \equiv A$ pro libovolné k, které je násobkem modulu m. Přičítáme totiž kongruenci $k \equiv 0$.

Vynásobení dvou kongruencí: Z $a \equiv A$ a $b \equiv B$ plyne $ab \equiv AB$. Stejně jako u součtů a rozdílů, i zde stačí čísla rozepsat na součty násobků m a zbytků po dělení m:

$$\begin{aligned} a \cdot b &= (n_a \cdot m + z_a) \cdot (n_b \cdot m + z_b) = \\ &= (n_a \cdot n_b \cdot m + n_a \cdot z_b + n_b \cdot z_a) \cdot m + (z_a \cdot z_b) \equiv \\ &\equiv z_a \cdot z_b, \\ A \cdot B &= (n_A \cdot m + z_A) \cdot (n_B \cdot m + z_B) = \\ &= (n_A \cdot n_B \cdot m + n_A \cdot z_B + n_B \cdot z_A) \cdot m + (z_A \cdot z_B) \equiv \\ &\equiv z_A \cdot z_B. \end{aligned}$$

Přitom opět víme, že $z_a = z_A$ a $z_b = z_B$.

Vynásobení obou stran kongruence týmtéž číslem: Pokud $a \equiv A$, platí také $ax \equiv Ax$ pro libovolné x. To plyne z násobení kongruencí $x \equiv x$.

Ekvivalentnost úprav: Běžné úpravy rovnic jsou takzvaně ekvivalentní – to znamená, že fungují oběma směry, takže řešení rovnic ani neubírají, ani nepřidávají. Jak je to s kongruencemi? Sčítání kongruencí ekvivalentní musí být, protože opačný směr odpovídá odečtení kongruencí, což víme, že je také korektní úprava.

U násobení kongruencí to už tak jasné není. Zkusme zjistit, jestli je pravda, že z kongruence $ax \equiv Ax$ plyne $a \equiv A$. Pro $x = 0$ to jistě neplatí, ale co když zvolíme jiné x?

Vyzkoušíme to třeba na následujícím příkladě:

$$a \equiv 5 \pmod{14}$$

Takováto kongruence má jednoduché řešení: a je každé celé číslo, které dostaneme sečtením 5 a nějakého násobku 14. To se dá zapsat třeba takhle:

$$a \in \{5 + k \cdot 14 \mid k \in \mathbb{Z}\},$$

tudíž a může být například 5, 19 nebo 33.

Vyzkoušíme nyní obě strany vynásobit… třeba trojkou:

$$3 \cdot a \equiv 15 \equiv 1 \pmod{14}.$$

Tato kongruence platí pro všechna a, která po vynásobení 3 dávají modulo 14 zbytek 1. Když máme nějaké a z první kongruence, je ve tvaru $5+k\cdot 14$. Vynásobíme-li ho 3, dostaneme $15+3\cdot k\cdot 14$. To je určitě kongruentní s 1 modulo 14, takže žádné řešení jsme neztratili a po chvíli uvažování zjistíme, že jsme ani žádné nepřidali.

Další pokus: původní kongruenci vynásobíme místo trojky dvojkou. Dostaneme:

$$2\cdot a \equiv 10 \pmod{14}.$$

Řešení původní kongruence pořád sedí, ale nová kongruence platí například i pro $a = 12$. Posuďte sami:

$$2\cdot 12 = 24 = 10 + 14 \equiv 10 \pmod{14}.$$

Ouha, najednou vynásobení obou stran konstantou není ekvivalentní úprava!

Proč nám násobení trojkou fungovalo, ale násobení dvojkou si vymýšlí kořeny navíc? Postupně se ukáže, že násobení k je ekvivalentní úprava právě tehdy, když $k \perp 14$ (či obecněji $k \perp m$, počítáme-li modulo m).

Než k tomu dojdeme, nejdřív na chvíli odbočíme k největším společným dělitelům.

Euklidův algoritmus

Největšího společného dělitele dvou čísel můžeme vypočítat pomocí prvočíselného rozkladu, ale to je pro velká čísla velmi pomalé. Daleko lepší je použít prastarý Euklidův algoritmus. (Jmenuje se podle starověkého matematika Euklida, v jehož díle Základy se nachází první dochovaná verze. Jedná se zřejmě o nejstarší netriviální algoritmus, jaký se s drobnými úpravami používá dodnes. Dokonce je pravděpodobné, že Euklidés pouze sepsal dávno známý trik.)

Pojďme si odvodit, jak Euklidův algoritmus funguje. Nahlédneme, že pro libovolná čísla a, b $(a > b)$ platí:

$$\mathrm{nsd}(a, b) = \mathrm{nsd}(a - b, b).$$

Proč je to pravda? Dokážeme, že dvojice (a, b) a $(a - b, b)$ sdílejí dokonce všechny společné dělitele, takže i toho největšího:

- Nechť d je společným dělitelem a a b. Platí tedy $a = a'\cdot d$, $b = b'\cdot d$ pro nějaká celá čísla a' a b'. Pak ovšem můžeme zapsat $a-b$ jako $(a'-b')\cdot d$, což je zase dělitelné číslem d.
- Nechť naopak d je společným dělitelem $a - b$ a b. Opět zapíšeme $a-b = c'\cdot d$, $b = b'\cdot d$ a získáme $a = (a-b)+b = (c' + b')\cdot d$.

Euklidův algoritmus dostane na vstupu nějaká dvě čísla a a b a opakovaně odčítá menší z nich od většího. Jak už víme, tato operace zachovává největšího společného dělitele. Pokaždé se přitom součet $a + b$ zmenší, takže po konečně mnoha krocích musíme jedno z čísel vynulovat. Pak víme, že největším společným dělitelem je druhé z nich (platí přeci $\mathrm{nsd}(0, x) = x$).

Pojďme si takhle nějakého největšího společného dělitele spočítat. A abychom netroškařili, zkusme rovnou čísla 1518 a 945.

a	b
1518	945
573	945
573	372
201	372
201	171
30	171

Zastavme se na chvíli. Teď bychom mohli pracně odečítat 30 od b, dokud bychom nenašli v b něco menšího než 30. Buďme trochu líní: když to budeme dělat dost dlouho, zbude nám v b zkrátka zbytek po dělení b číslem 30. Můžeme tedy místo odečítání modulit. Pokračujeme:

a	b
30	171
30	$21 = 171 \bmod 30$
$30 \bmod 21 = 9$	21
9	$3 = 21 \bmod 9$
$9 \bmod 3 = 0$	3

V a nám zbyla 0, takže $\mathrm{nsd}(1518, 945) = 3$.

Pojďme si tento postup převést do návodu pro počítač. Při implementaci Euklidova algoritmu se hodí držet si v jedné proměnné pořád to větší z čísel a a b. Navíc můžeme využít toho, že každým krokem algoritmu se z většího čísla stane menší, takže je stačí prohodit a není potřeba znovu porovnávat. (Dokonce i porovnání před cyklem bychom si mohli ušetřit, kdyby nám nevadilo, že první průchod cyklem může projít „naprázdno“.)

```
def Euclid(a, b):
    # Prohodíme, je-li třeba
    if b > a:
        a, b = b, a

    # Zde je vždy a >= b
    while b > 0:
        # nsd(a % b, b) = nsd(a, b).
        a = a % b
        a, b = b, a

    # nsd(0, a) = a.
    return a
```

Jak rychle náš algoritmus běží? Podívejme se, co se stane, když pustíme dva kroky algoritmu na čísla a_1 a b_1 $(a_1 \geq b_1)$:

$$\begin{aligned}
&a_2 = a_1 \bmod b_1 \\
&b_2 = b_1 && \text{(nyní } a_2 < b_2\text{)} \\
&a_3 = a_2 = a_1 \bmod b_1 \\
&b_3 = b_2 \bmod a_2 = b_1 \bmod (a_1 \bmod b_1) && \text{(nyní } a_3 > b_3\text{)}
\end{aligned}$$

Dokážeme, že $a_3 < a_1/2$. Rozebereme přitom dva případy podle toho, jestli bylo b_1 menší, nebo větší než $a_1/2$:

- Pokud $b_1 \leq a_1/2$, pak určitě platí $a_3 < b_1$, a tedy i $a_3 < a_1/2$. (Zde využíváme toho, že zbytek po dělení čehokoliv číslem b_1 musí být menší než b_1.)
- V opačném případě leží b_1 mezi $a_1/2$ a a_1, takže $a_3 = a_1 \bmod b_1 = a_1 - b_1 < a_1/2$. (Poslední rovnost platí, protože $\lfloor a_1/b_1 \rfloor = 1$.)

Dokázali jsme tedy, že po dvou krocích algoritmu se větší z obou proměnných zmenší přinejmenším na polovinu a opět bude větší. Po $\mathcal{O}(\log n)$ krocích tedy musí větší proměnná klesnout pod 1, čímž se algoritmus zastaví. Euklidův algoritmus proto provede $\mathcal{O}(\log n)$ elementárních operací.

Jak dlouho ale trvá jedna elementární operace? Pokud počítáme s malými čísly, která se našemu počítači vejdou do celočíselné proměnné, zvládneme ji v konstantním čase. Jsou-li ovšem čísla větší, musíme ještě zohlednit složitost aritmetických operací: porovnání čísel a operace modulo. Když použijeme modulení pomocí školního dělení, které je kvadratické v počtu cifer, strávíme v každém z $\mathcal{O}(\log n)$ kroků

Euklidova algoritmu čas $\mathcal{O}(\log^2 n)$. Celková složitost algoritmu tedy vzroste na $\mathcal{O}(\log^3 n)$.

Rozšířený Euklidův algoritmus

Právě jsme našli největšího společného dělitele d nějakých dvou obrovských čísel a a b. Jak ale přesvědčíme svého pochybovačného kolegu, že je náš výsledek správně? Snadno ověříme, že d dělí obě čísla. Ale jak ukážeme, že žádné větší číslo už a ani b nedělí? Překvapivě to jde jednoduše dosvědčit: pokud pro nějaká u a v platí

$$a \cdot u + b \cdot v = d,$$

musí d být dělitelné každým společným dělitelem a a b, takže i číslem $\mathrm{nsd}(a, b)$. Nemůže tedy být menší než $\mathrm{nsd}(a, b)$.

Dobrá – kde taková u a v vzít? Kupodivu snadno: trochu upravíme Euklidův algoritmus. Nejprve ale prozradíme, že rovnici

$$a \cdot u + b \cdot v = \mathrm{nsd}(a, b)$$

se říká *Bézoutova identita* a číslům u a v *Bézoutovy koeficienty*.

Dokážeme, že spustíme-li Euklidův algoritmus na čísla a a b, v každém okamžiku se v proměnných `a` a `b` budou nacházet čísla tvaru $\alpha \cdot a + \beta \cdot b$ (kde α a β jsou nějaká celá čísla). Na začátku to triviálně platí, neboť $\mathtt{a} = a$ a $\mathtt{b} = b$, a pokaždé, když se proměnné mění, buď se prohazují, nebo se jedna odčítá od druhé. Obě tyto operace z výrazů uvedeného tvaru dělají opět výrazy uvedeného tvaru. Takže i konečný výsledek algoritmu, tedy $\mathrm{nsd}(a, b)$, musí jít zapsat v takovém tvaru.

Algoritmus proto upravíme tak, aby si stále udržoval proměnné α_a, α_b, β_a a β_b a vždy platilo

$$\mathtt{a} = \alpha_{\mathtt{a}} \cdot a + \beta_{\mathtt{a}} \cdot b,$$
$$\mathtt{b} = \alpha_{\mathtt{b}} \cdot a + \beta_{\mathtt{b}} \cdot b.$$

Ke konci algoritmu je, jak víme, $\mathtt{b} = \mathrm{nsd}(a, b)$, takže $\alpha_{\mathtt{b}}$ a $\beta_{\mathtt{b}}$ jsou hledané Bézoutovy koeficienty. Opět si to vyzkoušejme na výpočtu $\mathrm{nsd}(1518, 945)$:

a	$\alpha_{\mathtt{a}}$	$\beta_{\mathtt{a}}$	b	$\alpha_{\mathtt{b}}$	$\beta_{\mathtt{b}}$
1518	1	0	945	0	1
573	1	−1	945	0	1
573	1	−1	372	−1	2
201	2	−3	372	−1	2
201	2	−3	171	−3	5
30	5	−8	171	−3	5
30	5	−8	21	−28	45
9	33	−53	21	−28	45
9	33	−53	3	−94	151
0	315	−506	3	−94	151

Algoritmus tedy tvrdí, že hledaný nsd splňuje rovnost

$$1518 \cdot (-94) + 945 \cdot 151 = \mathrm{nsd}(1518, 945) = 3.$$

Snadným výpočtem ověříme, že je to pravda. (Poznamenejme, že Bézoutova identita má nekonečně mnoho řešení. Jak byste našli ta další?)

Převeďme své myšlenky do zdrojového kódu. Do `Aa`, `Ba`, `Ab`, `Bb` budeme ukládat koeficienty $\alpha_{\mathtt{a}}$, $\beta_{\mathtt{a}}$, $\alpha_{\mathtt{b}}$ a $\beta_{\mathtt{b}}$.

```
def ExtEuclid(a, b):
    Aa, Ba = 1, 0  # a = 1 * a + 0 * b
    Ab, Bb = 0, 1  # b = 0 * a + 1 * b
    # Prohodíme, je-li třeba
    if b > a:
        a, b = b, a
        Aa, Ab = Ab, Aa
        Ba, Bb = Bb, Ba
    # Zde je vždy a >= b
    while b > 0:
        # Odečteme od proměnné a proměnnou b
        # tolikrát, kolikrát se tam vejde.
        # ("/" značí celočíselné dělení)
        Aa = Aa - (a / b) * Ab
        Ba = Ba - (a / b) * Bb
        # nsd(a % b, b) = nsd(a, b).
        a = a % b
        # Prohodíme
        a, b = b, a
        Aa, Ab = Ab, Aa
        Ba, Bb = Bb, Ba
    # nsd(0, a) = a.
    # Vrátíme také Bézoutovy koeficienty.
    return (a, Aa, Ba)
```

Žádná operace, kterou děláme s koeficienty pro proměnné `a` a `b`, netrvá asymptoticky déle než operace modulo. Přidáním počítání Bézoutových koeficientů si tedy časovou složitost Euklidova algoritmu nezhoršíme.

Řešení lineárních kongruencí

Bézoutovy koeficienty jsou užitečné také k řešení kongruencí. Pojďme si to na jedné kongruenci vyzkoušet.

Máme nakoupena 4 vajíčka. V obchodě se vajíčka prodávají pouze v balíčcích po 6 kusech, zatímco my je skladujeme v platech po 20 kusech. Kolik si musíme koupit balíčků, abychom neměli v žádném platu volno?

Přepišme si tento příklad do formy kongruence:

$$4 + 6 \cdot x \equiv 0 \pmod{20},$$

čili

$$6 \cdot x \equiv 16 \pmod{20}.$$

To je totéž, jako že pro x a nějaké další celé číslo y platí

$$6 \cdot x + 20 \cdot y = 16.$$

Ejhle, to je rovnice podobná Bézoutově identitě. Kdyby na její pravé straně byl $\mathrm{nsd}(6, 20) = 2$, byla by to přesně Bézoutova identita a rozšířený Euklidův algoritmus by nám prozradil, že platí

$$6 \cdot (-3) + 20 \cdot 1 = 2.$$

Tím bychom měli vyřešeno.

Jenže v našem případě je na pravé straně 8krát víc, než bychom potřebovali. Tak obě strany Bézoutovy identity vynásobíme 8:

$$6 \cdot (-3) \cdot 8 + 20 \cdot 1 \cdot 8 = 2 \cdot 8.$$

Řešením naší rovnice tedy je $x = -24$, $y = 8$.

Tak hurá do obchodu nakoupit -24 balíčků vajec. Cože? Že záporné nemají? Nevadí – stačí si vzpomenout, že jsme původně počítali modulo 20, takže k x můžeme přičíst libovolný násobek 20 a dostaneme další řešení. Můžeme tedy jít třeba pro 16 balíčků.[2]

[2] Mimochodem, není to nejmenší počet balíčků, který vyhovuje úloze: 6 balíčků by také fungovalo. Rozmyslete si, jak najít *nejmenší* řešení kongruence.

Teď už můžeme zformulovat obecný *návod na řešení kongruence*

$$ax \equiv b \pmod{n}$$

s neznámou x. Kongruenci přepíšeme do tvaru

$$ax - ny = b$$

a označíme $d = \text{nsd}(a, n)$. Rozlišíme 3 případy:

- $d = b$ ... tehdy jsou hledaná x a y rovná Bézoutovým koeficientům a najdeme je rozšířeným Euklidovým algoritmem.
- $d \setminus b$ ... pak najdeme řešení x' a y' rovnice s d na pravé straně a položíme $x = x' \cdot b/d$ a $y = y' \cdot b/d$.
- b není násobkem d ... v tomto případě kongruence nemůže mít žádné řešení, neboť levá strana rovnice je pro každé x a y dělitelná d, zatímco pravá strana dělitelná d nikdy není.

Inverzní prvky modulo m

Vraťme se teď zpátky z dlouhé odbočky a zkusme se znovu zamyslet nad tím, kdy je vynásobení obou stran kongruence ve tvaru

$$x \equiv z \pmod{m}$$

konstantou k ekvivalentní úprava. Tak říkáme úpravě, která neubírá ani nepřidává řešení. Už máme dokázáno, že když $x \equiv z$, tak pro každé k platí i $k \cdot x \equiv k \cdot z$. Takže zbývá zajistit, aby každé řešení kongruence $k \cdot x \equiv k \cdot z$ bylo i řešením $x \equiv z$.

Nejprve ukážeme, že pokud k je soudělné s m, je naše snaha předem ztracená. Označme $d = \text{nsd}(k, m) > 1$. Vezměme libovolnou dvojici x a z splňující kongruenci $x \equiv z$, což je totéž jako $x - z \equiv 0$. Nyní vytvořme novou dvojici $x' = x$ a $z' = z + m/d$. Pro tu dostaneme

$$x' - z' \equiv x - (z + m/d) \equiv x - z - m/d \equiv -m/d \not\equiv 0.$$

Ovšem kongruenci vynásobenou k tato nová dvojice stále splňuje:

$$\begin{aligned} kx' - kz' &\equiv kx - k(z + m/d) \equiv kx - kz - km/d \equiv \\ &\equiv -km/d \equiv m \cdot (-k/d) \equiv 0. \end{aligned}$$

Dokázali jsme tedy, že pokud číslo k, kterým násobíme obě strany kongruence, je soudělné s modulem m, nejedná se o ekvivalentní úpravu. Teď naopak ukážeme, že jsou-li k a m nesoudělná, ekvivalentní to je.

Nahlédneme, že kdykoliv $k \perp m$, existuje nějaké číslo $k^{-1} \in \mathbb{Z}_m$ takové, že $k \cdot k^{-1} \equiv 1$. Tomuto číslu se říká *inverzní prvek ke k* (nebo také *multiplikativní invers čísla k*), a pokud jím kongruenci $kx \equiv kz$ vynásobíme, získáme

$$k \cdot k^{-1} \cdot x \equiv k \cdot k^{-1} \cdot z \pmod{m},$$

což je kýžená kongruence $x \equiv z$.

Kongruenci $k \cdot k^{-1} \equiv 1$ přitom už umíme vyřešit – předchozí kapitola nám říká, že takové k^{-1} existuje právě tehdy, je-li $k \perp m$, a že se dá najít Euklidovým algoritmem. Dodejme ještě, že prvkům, které mají multiplikativní invers, se říká *invertibilní prvky modulo m*.

Konečná tělesa

Když speciálně zvolíme za m nějaké prvočíslo, budou všechny prvky $\mathbb{Z}_m$ kromě nuly invertibilní. Tím pádem se $\mathbb{Z}_m$ bude chovat dost podobně racionálním nebo reálným číslům. Má s nimi například tyto společné vlastnosti (sčítáním a násobením v případě $\mathbb{Z}_m$ myslíme operace modulo m):

- *Sčítání* je asociativní a komutativní.
- Pro každé a platí $a + 0 = a$.
- Pro každé a existuje $(-a)$ takové, že $a + (-a) = 0$.
- *Násobení* je asociativní a komutativní.
- Pro každé a je $a \cdot 1 = a$.
- Pro každé nenulové a existuje a^{-1} takové, že $a \cdot a^{-1} = 1$.
- Násobení a sčítání jsou distributivní: $a \cdot (b - c) = a \cdot b - a \cdot c$.

Obecněji, máme-li libovolnou množinu, můžeme v ní označit „jedničku“ a „nulu“ a „přibalit“ operace sčítání, násobení, „dej mi $(-a)$“ a „dej mi a^{-1}“. Operací přitom myslíme libovolnou funkci, která prvkům množiny nebo jejich dvojicím přiřazuje prvky. Pokud navíc pro naši množinu s operacemi platí všechny vyjmenované vlastnosti, říká se jí *komutativní těleso*.

Racionální, reálná i komplexní čísla jsou příklady takových těles a my jsme k nim přidali *konečná tělesa* velikosti prvočísla. (Na okraj poznamenejme, že je známo, že všechna konečná tělesa mají velikost mocniny prvočísla, což ovšem neznamená, že se vždy chovají jako celá čísla modulo nějakým m.)

Malá Fermatova věta

S prvočísly úzce souvisí takzvaná *Malá Fermatova věta.* Říká, že pokud je p prvočíslo a a libovolné číslo od 1 do $p-1$, tak

$$a^{p-1} \equiv 1 \pmod{p}.$$

Tato věta má mnoho různých použití (třeba ve známém šifrovacím algoritmu RSA nebo níže v algoritmu na testování prvočíselnosti), nám se bude především hodit jako další způsob invertování čísel modulo prvočíslo:

$$a^{p-2} \cdot a \equiv a^{p-1} \equiv 1 \pmod{p},$$

takže a^{p-2} je inverzní prvek k a.

Pojďme nyní Malou Fermatovu větu dokázat. Indukcí podle a budeme dokazovat ekvivalentní tvrzení

$$a^p \equiv a \pmod{p}.$$

(Jelikož a je určitě nesoudělné s modulem p, tak už víme, že násobení obou stran kongruence je ekvivalentní úprava.)

Pro $a = 1$ je snadné vidět, že věta platí:

$$a^p = 1^p = 1 \equiv 1 \pmod{p}.$$

Teď uděláme indukční krok. Řekněme, že máme dokázáno, že naše věta platí pro nějaké a, a chceme ji dokázat i pro $a + 1$. K tomu se bude hodit známá Binomická věta, která říká, že pro každé reálné x a y a přirozené n platí:

$$(x + y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i},$$

přičemž $\binom{n}{i}$ je takzvané *kombinační číslo* tvaru

$$\binom{n}{i} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-i+1)}{i \cdot (i-1) \cdot \ldots \cdot 1},$$

mající v čitateli i jmenovali zlomku právě i členů.

Indukce po nás chce, abychom dokázali, že $(a+1)^p \equiv a$. Rozepíšeme tedy levou stranu kongruence pomocí Binomické věty:

$$(a+1)^p = \binom{p}{0} a^0 + \binom{p}{1} a^1 + \ldots + \binom{p}{p} a^p.$$

Jelikož $\binom{p}{0}$ i $\binom{p}{p}$ jsou rovny 1, tvoří první a poslední člen součtu dohromady $a^p + 1$. To je podle indukčního předpokladu kongruentní s a.

Zbývá tedy dokázat, že všechny ostatní členy jsou dělitelné p, takže se v kongruenci modulo p neprojeví. Vskutku: pro $0 < i < p$ se ve zlomku definujícím $\binom{p}{i}$ objeví p v prvočíselném rozkladu čitatele, ale ne v rozkladu jmenovatele, takže se nemá s čím zkrátit.

Tím je indukce hotova.

Fermatův test prvočíselnosti

Jako malou odměnu za dlouhý důkaz předvedeme, jak Malou Fermatovu větu využívat ke zjištění, zda je nějaké obrovské číslo n prvočíslem. Jistě bychom mohli zkoušet všechny kandidáty na dělitele od 2 do $n - 1$ (nebo chytřeji do $\lfloor\sqrt{n}\rfloor$), ale to by trvalo příliš dlouho.

Raději zkusíme vybrat nějaké náhodné $a \in \{1, \ldots, n-1\}$ a spočítat, kolik je $a^{n-1} \bmod n$. Pro prvočíselné n musí vyjít jednička, takže pokud vyjde něco jiného, usvědčili jsme n z toho, že není prvočíslem (aniž jsme našli jediného dělitele – zvláštní, že?).

Pokud pro toto konkrétní a jednička vyjde, samozřejmě to neznamená, že n je určitě prvočíslo. Vyzkoušíme proto několik různých a, a pokud test pro žádné z nich neselže, drze prohlásíme, že n je pravděpodobně prvočíslo.

Jak moc velká drzost to je? Překvapivě ne moc velká. Pro skoro každé složené číslo n platí, že alespoň polovina a-ček dosvědčí, že se nejedná o prvočíslo. Takže jeden pokus selže s pravděpodobností nejvýše $1/2$ a pokud uděláme t pokusů, pravděpodobnost chybného výsledku je nanejvýš $1/2^t$.

Jedinou výjimku z našeho pravidla tvoří takzvaná *Carmichaelova čísla* (nejmenší z nich je číslo 561). To jsou čísla, jejichž složenost prokážeme jen tehdy, když se strefíme do a soudělného s n, a takových a je velmi málo. Naštěstí není Carmichaelových čísel moc (relativně k prvočíslům), takže Fermatův test funguje docela spolehlivě.

Existují i důmyslnější testy, které se Carmichaelovými čísly obalamutit nenechají. Jejich popis, jakož i důkaz našeho tvrzení o spolehlivosti Fermatova testu, najdete v literatuře zmíněné na konci kuchařky.

Rychlé mocnění

Ve Fermatově testu nebo při počítání inverzí pomocí Malé Fermatovy věty potřebujeme spočítat $a^k \bmod m$ pro velké k. Pokud budeme mocninu a^k počítat přímo podle definice, tedy jako $a \cdot a \cdot \ldots \cdot a$, budeme potřebovat $\mathcal{O}(k)$ násobení, což je příliš.

Jednoduchou fintou lze počet operací snížit na $\mathcal{O}(\log k)$. Například a^{16} můžeme spočítat jako:

$$a^{16} = (a^8)^2 = ((a^4)^2)^2 = (((a^2)^2)^2)^2.$$

Pro obecný exponent bude rychlejší umocňování vypadat takto:

```
def FastExp(a, k):
    # Nejdříve ošetříme triviální případy.
    if k == 0: return 1
    if k == 1: return a

    # Když je x sudé, vrátíme a^(k/2) * a^(k/2).
    # Když je x liché, vrátíme a * a^(k - 1).
    if k % 2 == 0:
        i = FastExp(a, k / 2)
        return i * i
    else:
        return a * FastExp(a, k - 1)
```

Každé volání `FastExp` pro sudé k jednou zavolá `FastExp` s poloviční k a jednou vynásobí dvě čísla. Když je k liché, převede se na sudé a provede se jedno vynásobení. `FastExp` tedy provede $\mathcal{O}(\log k)$ násobení.

Je ale důležité uvědomit si, že kdybychom si neuložili výsledek $a^{k/2}$ do pomocné proměnné `i`, ale rovnou vraceli `FastExp(a, k / 2) * FastExp(a, k / 2)`, byl by náš kód stejně pomalý, jako kdybychom počítali mocninu podle definice!

Pro použití ve Fermatově testu (nebo obecně na spočítání mocniny $a^k \bmod m$) stačí po každém násobení výsledek vymodulit m.

Síto na prvočísla

Už jsme zjistili, že se nám hodí umět najít prvočísla. Kde je ale vezmeme? Můžeme určitě zkoušet jedno číslo po druhém a pokaždé otestovat, jestli držíme prvočíslo (třeba Fermatovým testem nebo zkoušením všech dělitelů). Už staří Řekové ale znali algoritmus, který najde všechna prvočísla menší než n efektivněji. Říká se mu *Eratosthenovo síto.*

Síto funguje na docela jednoduchém principu. Budeme si uchovávat v paměti pro každé číslo od 2 do n příznak, jestli je prvočíslo, nebo složené. Začneme u dvojky a označíme všechny násobky 2 ležící mezi 4 a n jako složená čísla. Další prvočíslo je 3. Označíme všechny násobky 3 ležící od 6 do n jako složená čísla. Další číslo na řadě je 4. Když jsme ale vyškrtávali násobky 2, vyškrtli jsme i 4. Nebudeme tedy provádět nic a rovnou přejdeme na 5. Takto najdeme všechna prvočísla od 2 do n a stihneme to rychle.

Ukažme si ještě zdrojový kód v Pythonu:

```
def Eratosthenes(n):
    prvocislo = [ True ] * n

    for i in range(2, n):
        if prvocislo[i]:
            print("%d je prvocislo." % i)

            # Násobky prvočísla jsou složené
            j = i * 2
            while j < n:
                prvocislo[j] = False
            j += i
```

Jak dlouho síto poběží? Dá se dokázat, že jeho časová složitost činí $\mathcal{O}(n \log\log n)$, ale není to snadné. My si zde předvedeme jenom slabší odhad $\mathcal{O}(n \log n)$. Zájemce o těžší důkaz menší složitosti odkazujeme na vzorové řešení úlohy 24-3-5.[3]

Síto tráví čas $\mathcal{O}(n)$ hledáním prvočísel a mezitím škrtá jejich násobky. Když škrtáme násobky dvojky, vyškrtneme nejvýše $n/2$ čísel, když škrtáme násobky trojky, vyškrtneme jich nejvýše $n/3$, atd. Složitost Eratosthenova síta tedy bude shora omezena součtem

$$n + \frac{n}{2} + \frac{n}{3} + \ldots + \frac{n}{n} = n \cdot \sum_{i=1}^{n} \frac{1}{i}.$$

[3] `http://ksp.mff.cuni.cz/viz/24-3-5/reseni`

Sumě

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

se říká n-té harmonické číslo a dokážeme o něm, že leží v $\mathcal{O}(\log n)$.

Uvažujme, o co se zvětší H_{2n} oproti H_n:

$$H_{2n} - H_n = \frac{1}{n+1} + \frac{1}{n+2} + \ldots + \frac{1}{2n}.$$

To je součet n členů, z nichž každý je menší než $1/n$. Celý součet je tedy menší než 1.

Zjistili jsme tedy, že $H_{2n} < H_n + 1$. Funkce, které rostou takhle pomalu, jdou shora omezit nějakým logaritmem n, takže $H_n = \mathcal{O}(\log n)$.

Proto složitost celého Eratosthenova síta činí $\mathcal{O}(n \log n)$.

Čínská zbytková věta

Následující věta dostala své jméno po staročínském způsobu počítání vojáků. Čínská armáda je velká, a kdybychom chtěli počítat vojáky jednoho po druhém, trvalo by to dlouho. Pomáhalo prý armádu rozřadit do řad o velikostech m_1, m_2, …, m_n (součin všech m_i si označíme jako M bez indexu). Někdy zbyli nezařazení dva, někdy třicet, někdy se seřadili všichni. Tyto zbytky si označíme z_1, z_2, …, z_n.

A co Čínská zbytková věta říká? Tvrdí, že když jsou všechna m_i navzájem nesoudělná a počet vojáků je menší než M, lze ho ze zbytků z_i jednoznačně určit. Když například rozdělujeme vojáky do řad velikostí 2, 3, 5, 7, 11 a 13, můžeme zbytky z řad jednoznačně vyjádřit každý počet vojáků menší než $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 = 30\,030$.

Formálněji řečeno: Jsou-li dána navzájem nesoudělná přirozená čísla $m_1, \ldots, m_n$ (jejichž součin označíme M) a zbytky $z_1, \ldots, z_n$, pak existuje právě jedno číslo $x \in \mathbb{Z}_M$ takové, že pro všechna i je

$$x \equiv z_i \pmod{m_i}.$$

A jak dokážeme, že něco takového platí? Mějme 2 čísla $a, b \in \mathbb{Z}_M$ taková, že mají stejné zbytky po dělení všemi m_i. Ukážeme, že musí nutně být stejná.

Víme, že pro všechna i platí

$$a \equiv b \pmod{m_i}.$$

To podle definice kongruence znamená, že rozdíl $a - b$ je dělitelný všemi m_i. Proto je dělitelný i nejmenším společným násobkem všech m_i, což ovšem díky nesoudělnosti musí být jejich součin M.

Máme tedy dvě čísla ze $\mathbb{Z}_M$, jejichž rozdíl je dělitelný M. To nutně znamená, že jsou stejná.

Dokázali jsme tedy, že jedna sada zbytků $z_1, \ldots, z_n$ odpovídá jednoznačně určenému číslu $x \in \mathbb{Z}_M$, ale ještě nevíme, jak bez zkoušení všech možností toto x najít. Půjdeme na to od lesa.

Nejprve se hodí všimnout si toho, že když sečteme dvě čísla, sečtou se i jejich zbytky modulo všemi m_i.

Co kdybychom nyní dokázali sehnat čísla $Q_1, \ldots, Q_n$ taková, že Q_j je dělitelné všemi m_i kromě m_j a že $Q_j \equiv 1 \pmod{m_j}$?

To by potom stačilo položit

$$x = (z_1 \cdot Q_1 + z_2 \cdot Q_2 + \ldots + z_n \cdot Q_n) \bmod M.$$

Vskutku: počítáme-li $x \bmod m_i$, všechny členy $z_j \cdot Q_j$ pro $j \neq i$ vyjdou nulové a člen $z_i \cdot Q_i$ bude roven z_i. To, že celý výsledek nakonec vymodulíme M, na věci nic nemění, protože přičtení či odečtení libovolného násobku M zbytek po dělení žádným m_i neovlivní.

Jak se ale k číslům Q_i dostaneme? Číslo Q_i má být dělitelné všemi m_j kromě m_i. Uvažujme tedy součin

$$S_i = m_1 \cdot \ldots \cdot m_{i-1} \cdot m_{i+1} \cdot \ldots \cdot m_n.$$

Ten modulo každé m_j ($j \neq i$) dá nulu, zatímco modulo m_i nějaké číslo r_i nesoudělné s m_i (nesoudělné musí být, protože jinak by m_i bylo soudělné s některým m_j). Speciálně to znamená, že r_i není 0.

Potřebujeme tedy z tohoto nenulového zbytku udělat jedničku. To zařídíme snadno: pořídíme si r_i^{-1}, což bude inverzní prvek k r_i modulo m_i, a tímto prvkem celé S_i vynásobíme:

$$Q_i = S_i \cdot r_i^{-1}.$$

Toto Q_i už má požadované vlastnosti: $Q_i \bmod m_j$ pro $j \neq i$ vyjde nulové, protože Q_i je násobkem S_i, které bylo dělitelné m_j. A modulo m_i získáme

$$Q_i \equiv S_i \cdot r_i^{-1} \equiv r_i \cdot r_i^{-1} \equiv 1.$$

„Kouzelná“ čísla Q_i tedy dokážeme sestrojit a jejich zkombinováním i hledané x.

Pojďme si to teď zkusit v praxi. Chceme najít nejmenší x takové, že platí následující kongruence:

$$\begin{aligned} x &\equiv 3 \pmod{5} \\ x &\equiv 1 \pmod{9} \\ x &\equiv 14 \pmod{16} \end{aligned}$$

Spočítáme si nejdříve M a všechna S_i:

$$\begin{aligned} M &= 5 \cdot 9 \cdot 16 = 720, \\ S_1 &= 9 \cdot 16 = 144, \\ S_2 &= 5 \cdot 16 = 80, \\ S_3 &= 5 \cdot 9 = 45. \end{aligned}$$

Teď zjistíme, kolik vychází každé S_i modulo m_i a určíme příslušné multiplikativní inverze (například pomocí rozšířeného Euklidova algoritmu):

$$\begin{aligned} r_1 &= 144 \bmod 5 = 4, \\ r_2 &= 80 \bmod 9 = 8, \\ r_3 &= 45 \bmod 16 = 13, \\ r_1^{-1} &= 4 \quad (4 \cdot 4 \bmod 5 = 1), \\ r_2^{-1} &= 8 \quad (8 \cdot 8 \bmod 9 = 1), \\ r_3^{-1} &= 5 \quad (13 \cdot 5 \bmod 16 = 1). \end{aligned}$$

Z toho vypočteme Q_i jako $S_i \cdot r_i^{-1}$:

$$\begin{aligned} Q_1 &= S_1 \cdot r_1^{-1} = 144 \cdot 4 = 576, \\ Q_2 &= S_2 \cdot r_2^{-1} = 80 \cdot 8 = 640, \\ Q_3 &= S_3 \cdot r_3^{-1} = 45 \cdot 5 = 225. \end{aligned}$$

Nakonec sečteme příslušné násobky Q_i a zjistíme x:

$$x \equiv 3 \cdot 576 + 1 \cdot 640 + 14 \cdot 225 = 5518 \equiv 478 \pmod{720}.$$

Výsledek opravdu vypadá správně:

$$x = 478 = 3 + (5 \cdot 95) = 1 + (9 \cdot 53) = 14 + (16 \cdot 29).$$

Pár slov na závěr

Doufáme, že se vám naše povídání o teorii čísel líbilo a že jste poznali, že i tak základní objekty, jako jsou celá čísla, mají spousty zajímavých vlastností.

Přejete-li si dozvědět se více o prvočíselných testech nebo o RSA, můžeme navrhnout ke studiu textík *Algoritmy okolo teorie čísel*[4] od jednoho z autorů kuchařky. Důkladný rozbor Eratosthenova síta a jiné zajímavosti o prvočíslech najdete v článku *Tři věty o prvočíslech*[5] od téhož autora.

S teorií čísel také souvisí algebra, která zobecňuje různé poznatky na libovolné množiny opatřené nějakými operacemi (například tělesa). Máte-li o ni zájem, mohla by vám pomoci například skripta *Základy algebry* od Davida Stanovského.

Kuchařku pro vás namíchali

Michal Pokorný a Martin Mareš

[4] `http://mj.ucw.cz/papers/numth.pdf`
[5] `http://mj.ucw.cz/papers/bert.pdf`