Korespondenční Seminář z Programování

30. ročník **KSP** **Prosinec 2017**

Milí řešitelé a řešitelky!

Vánoce jsou již tady a od Ježíška se můžete těšit na spoustu dárků. KSP také přispěje svým dílem do vínku, připravili jsme totiž pro vás další šťavnatou sérii úloh.

Protože přes Vánoce každý rád mlsá, u spousty úloh najdete předkrm v podobě lehčích variant. Nejen že za ně můžete dostat spoustu bodů, ale jako správný předkrm vás připraví na hlavní chod, tedy samotné úlohy. Navíc, pokud je vyřešíte všechny, vám KSP věnuje i sladký zákusek!

Ještě připomeneme, že každému řešiteli, který získá v tomto ročníku z každé série alespoň 5 bodů, pošleme KSP propisku, blok, placku a třeba i něco navíc.

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, díky kterému vás přijmou na MFF bez zkoušek. Pozor ale: pokud studujete poslední ročník střední školy a chcete letošní osvědčení využít, musíte mít potřebné body již po čtvrté sérii.

Termín série: **22. ledna v 8:00**

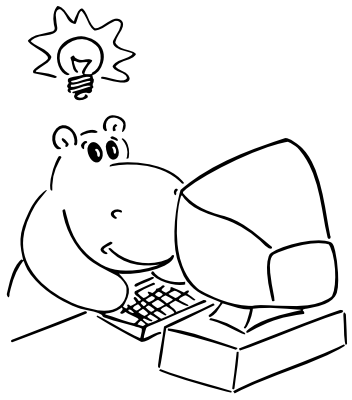
Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*

Odměna série: **Sladkou odměnu** si vysloužíš každý, kdo vyřeší **každou úlohu s lehkou variantou alespoň na polovinu bodů**.

Druhá série třicátého ročníku KSP

V budově Matfyzu na Malostranském náměstí je Rotunda jednoznačně největší a nejkrásnější místností. Je to vysoká kruhová dvorana sahající až do dalšího patra, kde skleněné stěny nabízí pohled do prostor školní knihovny. Kdysi se v Rotundě nacházely prostory národní banky, ale ty časy jsou už dávno pryč. Místo toho tu jsou do kruhu seřazené počítače, nalevo unixové, vpravo windowsové, jako kdyby se co nevidět měly pustit do boje. Venku se už dávno setmělo a otevírací doba počítačové laboratoře (neboli labu, jak říká každý správný matfyzák) se chýlí ke konci. Dělal jsem tu dnes celý den službu a teď zbývá lab, beztak už prázdný, zamknout.

Beru si věci, jdu ke dveřím a chystám se zhasnout světla, když vtom uvidím zvláštní věc. U jednoho z unixových počítačů úplně nalevo se rozblikala červená ledka. Co to má znamenat? Ani jsem nevěděl, že nějaký počítač v labu by uměl takhle blikat. Chci přijít blíž, ale pak si všimnu, že se úplně stejně rozblikal jeden z počítačů napravo. A po chvilce další. A další. A zanedlouho blikají červeně všechny počítače, a k tomu všemu navíc úplně synchronně.



„To je ale blbej vtip," mumlám si pro sebe, ale spíš chci zakrýt fakt, že jsem se začal trochu bát. Přijdu k jednomu ze strojů a pohnu myší. Displej se rozzáří a já vidím...

⋆⋆⋆

„... a Turingův stroj se vždycky zastaví."

Pane jo, já jsem zase usnul na přednášce? To by nebylo poprvé, ale nepamatuji si, že by se mi zdálo o sloužení v labu. Semestr už začal, musím ještě řešit resty z toho minulého, a do toho mi ostatní orgové KSP dají za úkol řídit vyšetřování toho, kam vlastně zmizel pan Nápověda. Jsem si jistý, že podobně perné týdny jsem měl minulý rok, ale musel bych se podívat do své databáze, jestli tomu tak opravdu bylo.

30-2-1 Zaneprázdněný org — 11 bodů

Náš vypravěč, organizátor KSPčka, si již dlouhou dobu zaznamenává, jak byl pro něj který týden hektický. Databázi těchto záznamů si můžete představit jako posloupnost celých čísel A_i, kde i představuje pořadové číslo týdne od samotného začátku měření. Hodnota každého prvku posloupnosti popisuje orgovu zaneprázdněnost během týdne.

Máte databázi k dispozici a chtěli bychom po vás, abyste uměli rychle odpovídat na dotazy „Kolikrát se vyskytlo v týdnech od x do y hodnocení H?", neboli „Kolikrát se v podposloupnosti A_x až A_y vyskytuje číslo H?". Dotazů může být mnoho, a proto se může hodit si na začátku předpočítat nějaká data a ta poté použít při odpovídání na dotazy. V takovém případě nás zajímají časové a paměťové složitosti jak předpočítání, tak jednoho dotazu.

Příklad vstupu: Posloupnost A_i je 1, 2, 2, 3, 2, 2, 3, 3. Předpokládáme, že indexujeme od jedničky. Na dotaz „kolikrát se od druhého do pátého týdne vyskytla zaneprázdněnost 2" je odpověď 3, na dotaz „kolikrát se od pátého týdne do osmého týdne vyskytla zaneprázdněnost 3" je odpověď 2.

⇧ **Lehčí varianta (za 6 bodů):** Řešte pro jednodušší dotazy „Vyskytuje se v týdnech od x do y hodnocení H?".

Skupince orgo-agentů s Jirkou v čele se podařilo zatopit Nápovědův bunkr, ale samotný Nápověda se někam vypařil a došla nám jen záhadná esemeska, že se přesouvá do Tokia. Moc jsme ale nevěřili tomu, že by nám ten padouch jen tak

vyzradil nějaké pravdivé informace. Navíc, se začátkem semestru se popravdě nikomu do Japonska odjíždět nechtělo. Po vysušení podzemních prostor se nám ale do ruky dostala některá zařízení, která on a jeho otroci vyvíjeli.

Na začátku jsme se ale nedostali k ničemu zajímavému. Nápověda pro jeden z experimentů vyžadoval mnoho náhodných čísel, ale asi byl hodně paranoidní a nevěřil tradičním generátorům. Proto si vyráběl vlastní, hardwarové generátory. Několik jsme jich zkoumali, ale všechny dělaly téměř to samé.

30-2-2 Hardwarový generátor — 13 bodů

Máme seznam M prvků a chceme z něj náhodně vybírat prvky. U každého prvku máme uvedené, s jakou pravděpodobností má být vybrán (pravděpodobnosti všech prvků se správně posčítají na 1).

Protože nevěříme tradičním generátorům čísel, používáme náš vlastní, hardwarový generátor. Ten však umí jedinou věc: rovnoměrně generovat nějaké číslo z uzavřeného intervalu $[0, 1]$.

Rovnoměrností myslíme, že všechna čísla mají stejnou šanci být vygenerována. (Kdybychom to chtěli definovat pořádně, nestačilo by říci, že mají stejnou pravděpodobnost, protože ta je pro každé z nekonečně mnoha čísel nulová. Mohli bychom třeba říci, že pro každý podinterval délky p platí, že se do něj strefíme s pravděpodobností přesně p.)

Určete, jak budete opakovaně náhodně vybírat prvky množiny se zadanými pravděpodobnostmi jen s pomocí našeho generátoru. Předpokládejte, že umíte přesně počítat s reálnými čísly, nevznikají zaokrouhlovací chyby a generování jednoho náhodného čísla probíhá v konstantním čase. Optimalizujte nejprve na čas strávený při generování jednoho prvku, následně pak na čas předvýpočtu před prvním generováním.

Známe řešení, které tráví na předvýpočtu $\mathcal{O}(M)$ a následná generování zvládne v konstantním čase. Vymyslíte nějaké stejně rychlé? Pokud ne, určitě pošlete i to pomalejší, také za něj dostanete body.

Příklad vstupu: Máte prvky A, B a C. A chcete generovat s pravděpodobností $1/6$, B s pravděpodobností $1/3$ a C s pravděpodobností $1/2$.

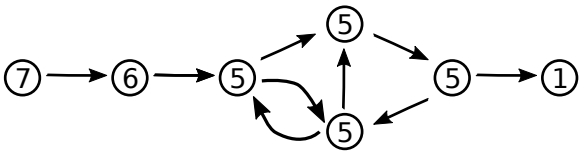
Naštěstí jsme měli k dispozici i disky se softwarem. To bylo o něco zajímavější. Hned po přednášce jsem na chodbě potkal Janku, jak zkoumá zdrojové kódy něčeho, co vypadalo jako počítačový virus. Po zátahu v Hostivaři Janka zjistila, že hackování počítačů ji vlastně baví, a začala téma studovat více do hloubky. Aby zdůraznila svoji novou zálibu, začala všude nosit černé brýle a chodit spát ještě později než většina organizátorů.

„To je dosť zaujímavý kúsok, tento vírus," vysvětlila mi. „nikto na internete ho nepozná, ale veľmi dobre sa šíri po sieti." „Jak to můžeš vědět?" ptám se. Ukázalo se, že Janka si stáhla simulátor počítačové sítě, v podstatě pár propojených virtuálních strojů, a zkoušela virus pustit v něm. Pokud se virus spustil na správném počítači, byl skutečně schopný celou síť zamořit.

30-2-3 Šíření viru podruhé — 10 bodů

Máte síť N navzájem propojených počítačů. V této síti zkoumáte chování viru, jehož úkolem je nakazit co nejvíce počítačů. Virus se ale nešíří úplně přímočaře. Máme seznam dvojic počítačů a u každé dvojice (A, B) platí, že se virus může přímo přenést z počítače A do počítače B (obráceně to být nemusí – pokud ano, tak se v seznamu vyskytuje další dvojice (B, A)).

Předpokládáme, že na začátku útoku je virus jen v jednom počítači v síti, odtud se rozšíří na všechny počítače, které dokáže přímo nakazit, následně se z těchto počítačů opět přenese dál, jak může... a takhle pokračuje, dokud je šíření možné. Pro každý počítač P chceme najít počet strojů, které se nakazí, pokud bude P nakažen na začátku útoku. Můžete se podívat na obrázek s příkladem. Kroužky odpovídají počítačům, dvojice počítačů, kde první může nakazit druhý, jsou propojeny a číslo u počítače P odpovídá počtu napadených počítačů, pokud je virus na začátku v P.

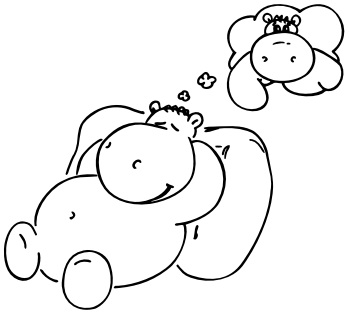


Lehčí varianta (za 6 bodů): Řešte stejnou úlohu za předpokladu, že v seznamu přenosů není cyklus.

Displej ukazuje šíření viru a já si mimoděk vzpomenu na ten divný sen na přednášce. To s těmi blikajícími počítači se nestalo, ale když to vidím, tak se nemůžu nezeptat: „Janka, určitě se ten program z těch virtuálních mašin nemůže dostat ven?"

„Nie," odpověděla. Ale nic dalšího neřekla, zavřela notebook a odešla. Jaký výraz měla ve tváři, to jsem kvůli těm černým brýlím nemohl odhadnout.

Zbytek dne jsem strávil podivně zmatený. Hlavou se mi honily podivné myšlenky. Vybavovaly se mi osoby, se kterými jsem se neznal, a místa, na kterých jsem určitě nikdy nebyl. Je vážně na čase se pořádně vyspat, řekl jsem si, navíc zítra máme kvůli vyšetřování nějakou zajímavou návštěvu.



⋆⋆⋆

Místnost S322 je základnou každého KSPáka, takže v devět hodin ráno (ano, ráno) tam jen tak na někoho nenarazíte. Když už ano, tak dotyčný vypadá, jako by právě vylezl z postele, a k dokonalému dojmu chybí jen pyžamo. Dnes jsme tu ale hostili partičku mediků, a to kvůli jednomu ze souborů v Nápovědově počítači, který podezřele připomínal sekvenci DNA. Než jsme se k souboru prokousali, museli jsme pochopit některé netradiční komprimační metody, které Nápověda používal.

30-2-4 Komprimace — 10 bodů

Vaším úkolem je rozbalit zkomprimovaná data. K jejich komprimaci došlo následujícím způsobem:

Data zapsaná jako posloupnost bitů se rozdělila na posloupnost různě dlouhých bloků. Každý blok je pak ve zkomprimovaném souboru reprezentován jedním ze dvou způsobů.

První způsob je jednoduchý, data bloku jsou zapsána přímo tak, jak byla v původním souboru. Druhý způsob umožňuje zkrátit zápis opakujících se kusů DNA. Místo samotných dat je zde pouze reference (odkaz) na kus původních dat, která jsou s daným blokem shodná. Reference na pozici i tedy znamená, že nultý bit bloku je shodný s i-tým bitem původních dat, první bit bloku s $(i+1)$-ním bitem atd., až do vyplnění celého bloku.

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 dostanete dvě mezerou oddělená čísla N a M. Číslo N značí počet bitů původního souboru a M počet bloků. Následuje M řádků, každý reprezentuje jeden blok. Každý z těchto řádků obsahuje 3 údaje oddělené mezerou. První je typ zápisu dat (`D` = data, nebo `R` = reference), druhý je velikost bloku. Třetí údaj je buď posloupnost bitů (tedy posloupnost nul a jedniček), pokud blok obsahuje přímo data, nebo odkaz na začátek úseku, jehož obsah je shodný s daty bloku (adresy bitů číslujeme od nuly). Bloky na vstupu jsou přesně v pořadí, jak jdou za sebou v původním souboru.

Formát výstupu: Na výstup vypište původní dekomprimovaný soubor, tedy posloupnost nul a jedniček, pokud je určena jednoznačně. Mohla se nicméně někde stát chyba a původní data nemusí jít ze zkomprimovaných informací určit jednoznačně, posloupnost původních bitů tedy nejde zrekonstruovat. V takovém případě vypište `NEJDE`.

Ukázkový vstup:

```
7 5
R 1 4
D 2 11
R 2 5
R 1 2
D 1 0
```

Ukázkový výstup:

```
0111010
```

Ukázkový vstup:

```
4 3
R 1 3
D 2 10
R 1 0
```

Ukázkový výstup:

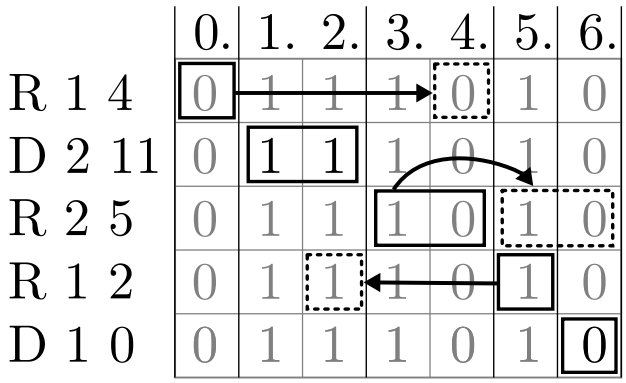
```
NEJDE
```

Ukázkový vstup:

```
10 2
D 2 01
R 8 0
```

Ukázkový výstup:

```
0101010101
```



Na obrázku vidíte zobrazený první vstup. Původní soubor měl 7 bitů. Máme pět bloků: pozici 0, pozice 1–2, pozice 3–4, pozici 5, pozici 6. Každý řádek reprezentuje zápis jednoho bloku.

„Víte, měli jsme takový projekt. Nedotáhli jsme ho do konce, ale nevylučujeme, že by se to Nápovědovi mohlo povést," řekl jeden z mediků. „Týkalo se to přenosu lidského vědomí. Zjistili jsme, že určitou hypnotizační technikou je možné přehrát vědomí do mozku jiného člověka."

„Jedná se v podstatě o silnou reakci organismu na jeden vizuální vjem," vysvětluje druhý. „Pak je jinými vjemy docela snadné přeprogramovat velkou část buněk v mozku."

Zamumlám: „To zní, jako kdyby ten člověk byl posedlý." Spánek mi nějak nepomohl, divně se mi točí hlava, asi bych potřeboval nějakou odbornou pomoc...

⋆⋆⋆

... nějaké dobré doktory prý mají v Tokiu.

⋆⋆⋆

Jeden z organizátorů beze slova vstal a odběhl z místnosti.

„Nevíte, co se s tím Kubou děje?" ptá se Filip. „Chová se od včerejška dost divně." Ostatní jenom pokrčili rameny. Ještě chvíli se s mediky bavili o tom, jak by bylo možné transplantaci vědomí využít, a jak by ji asi využil Nápověda. Pak už ale přišel čas se rozloučit. Orgové zamkli S322 a doprovázeli návštěvu k východu, když se zvenku ozvalo hlasité PRÁSK!

„Co to sakra je?" Filip otevřel vchodové dveře a zděšeně uskočil. Prohnalo se kolem něj rameno nějakého velkého stroje. „To je ten nový vysavač odpadků! Jak se tady tahle věc vzala?" Dostal rychle odpověď. Kolem dveří projela kabina stroje, ve které seděl Kuba. Ale podle jeho výrazu ve tváři nebylo jasné, jestli je to skutečně on.

30-2-5 Autovysavač — 12 bodů

Kuba, respektive pomocník pana Nápovědy v Kubově těle, si „vypůjčil" nový stroj Pražských služeb: obří vysavač na odpadky. Právě ho přivezl ho na parkoviště na Malostranském náměstí, a protože si chce vytvořit volný prostor, chce s ním nasát nějaká z aut, která tu parkují. Protože je pomocník škodolibý, chtěl by nasátím zničit co nejdražší auta, konkrétně takovou skupinu aut, aby medián jejich cen byl co nejvyšší.

Jak definujeme *medián* pro množinu čísel? Provedeme to jen pro případ, že je v množině lichý počet prvků. Pokud seřadíme čísla v množině podle velikosti, je medián tím číslem, které se nachází uprostřed seznamu, Platí tedy, že 50 % ostatních čísel je menší nebo rovno mediánu a 50 % je vyšší nebo rovno mediánu.

Parkoviště reprezentujeme jako čtverečkovou síť o rozměrech $M \times N$. V každém poli je uvedena hodnota zde stojícího auta. Vysavač dokáže vysát nějakou obdélníkovou oblast o rozměrech $P \times Q$ polí. Zjistěte, která oblast této velikosti má nejvyšší medián cen – máte jistotu, že $P \cdot Q$ je vždy liché.

Mějme například parkoviště velikosti 4×4 s následujícími hodnotami cen aut, přičemž dokážeme vysát oblast 3×3:

```
200 30 10 40
20  10 50 40
60  60 10 40
10  10 10 20
```

Ačkoliv je nejdražší auto v levém horním rohu, medián této oblasti velikosti 3×3 je jen 30. Lepší je pravá horní oblast 3×3, která má medián 40.

Lehčí varianta (za 6 bodů): Řešte za předpokladu, že $N = 1$.

Filip se s ostatními opatrně podíval ven. Kuba neustále popojížděl po parkovišti, hýbal ramenem vysavače na všechny strany a zdálo se, že pořád není spokojený s výběrem vozidel. Najednou se ale ozvalo zaburácení motoru a do prostoru parkoviště vjelo auto. Byl to Jirka a jeho Volkswagen! Než se kdokoliv stačil vzpamatovat, udělal Jirka pár obratů na ruční brzdě a naštrádoval si to přímo pod rameno vysavače.

„Ne! Tam nejezdi!“ vykřikl Filip. Všiml si, že se Kuba v kabině stroje zachechtal a sáhl po velké páce, která jistě zapínala vysávání. Než jí ale stačil pohnout, světla v kabině zhasla a motor stroje se zastavil.

„Zatracená kraksna!“ ozvalo se nadávání. Zdálo se, že Kubovi se také zablokovaly dveře, protože bylo slyšet, jak s nimi lomcuje. Jirka vylezl ven ze svého auta, jakoby nic. „Vy jste si mysleli, že nevím, co dělám?“ ušklíbl se na ostatní. „Věděl jsem, že tahle mašina je naprogramovaná hrozně mizerně. Ta řídící jednotka nedokáže odhadovat ceny podobně vytuněných aut, jako toho mého. Když naskenuje všechna moje vylepšení, tak se prostě uvaří a pošle do kopru celý vysavač.“

Z budovy vyběhla Janka. „Už som na to prišla! Virus je v labu a vykonával tie transplantácie, o ktorých ste sa bavili. Jaj,“ řekla zkroušeně, když viděla pohromu na parkovišti.

„Vsadím se, že Kuba, ehm, ten šílenec, který teď ovládá Kubovo tělo, tu nezůstával jenom kvůli tomu, že by chtěl jen tak ničit auta,“ řekl Filip. „Dávalo by smysl, aby utekl do Tokia za Nápovědou. No jasně,“ došlo mu, „nechtěl on jenom odkrýt vstup do parlamentního metra? Už jsme jeho síť zmapovali, dostal by se jím minimálně na letiště.“

30-2-6 Parlamentní metro — 12 bodů

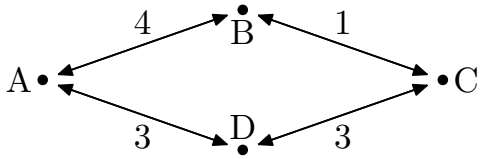
Za časů studené války vznikla v Praze tajná podzemní dráha s jediným účelem – evakuovat politiky a státníky v případě hrozící jaderné apokalypsy. Metro má mnoho stanic, které jsou propojeny tunely. Do jedné stanice může ústit libovolný počet tunelů, a ať už vlak přijede z jakéhokoliv směru, může se vydat dál jakýmkoliv tunelem.

Z bezpečnostních důvodů není metro napojeno na elektrickou síť, místo toho je každý vlak poháněn svou vlastní baterií. Aby měly vlaky co nejmenší tření, je z tunelů vypuštěn vzduch a vlaky se pohybují jen po magnetické kolejnici. Znamená to, že energii musí vlak vyvinout jen v momentě, když vyjíždí ze stanice, a to tím větší energii, čím větší rychlost chce vyvinout. Na délce úseku mezi stanicemi spotřeba vůbec nezávisí. Vlak ale musí (z bezpečnostních důvodů) zastavit v každé stanici na cestě.

Formálněji, vede-li mezi dvěma stanicemi tunel s délkou s a vlak jím projede rychlostí v, urazí celou trasu za s/v jednotek času a bude ho to stát v jednotek energie (čas na rozjíždění a brždění zanedbáváme).

Pro danou počáteční a cílovou stanici a zadané nabití baterie rozhodněte, jak se co nejrychleji dostat ze startu do cíle, abychom si přitom vystačili pouze s energií z baterie. Kromě seznamu stanic v pořadi, v jakém je navštívíme, nás zajímají i rychlosti, kterými budeme projíždět tunely mezi nimi.

Příklad: Pro následující rozložení stanic, počáteční stanici A, koncovou stanici C a baterii s kapacitou 6 jednotek je optimálním řešením jet přes stanici B, a to následovně: Na úseku A–B pojedeme rychlostí 4, na úseku B–C rychlostí 2, dohromady tedy využijeme celou baterii. Celkový čas cesty bude $4/4 + 1/2 = 3/2$ jednotek. Kdybychom místo toho jeli přes D, nestihli bychom se do cíle dostat dříve než za dvě časové jednotky.



Lehčí varianta (za 7 bodů): Všechny stanice tvoří jednu souvislou trasu – z koncových stanic vede jediný tunel, ze všech ostatních právě dva. Najděte nejrychlejší způsob jak se přepravit mezi koncovými stanicemi.

Po tom bláznivém večeru byl Kuba zatčen, ale naštěstí nebylo těžké ukázat, že je „posedlý“ a že do něj bylo transplantováno vědomí jednoho z pomocníků pana Nápovědy. Naštěstí se schopným medikům podařilo najít způsob, kterým posednutí zvrátit. O měsíc později se před místností S322 konala oslava na počest navrátivšího se Kuby. Sice se s obavami tvářil na jakýkoliv displej okolo sebe, ale byl pevně rozhodnutý pokračovat ve studiu.

„Kdy jen toho padoucha dostaneme,“ povzdechl si. „Neboj,“ uklidnil ho Jirka. „Už jsme prošli všechen jeho software. Žádné další viry tu být nemůžou, na všechny počítače jsme nainstalovali ochranný software.“ „Tak teda jo, budu ti věřit,“ usmál se Kuba.

⋆⋆⋆

Byla to pořádná párty. Poslední skupinka orgů odešla až nad ránem. Ten úplně poslední org zhasl světlo, takže v chodbě zůstalo šero, přerušované jen nesmělými paprsky ranního slunce.

Náhle šero přeruší bliknutí. A další. A zase další. Co to? V rohu chodby stojí velký kopírovací stroj. A ta červená ledka, co na něm bliká, by určitě blikat neměla...

Pak zhasne. Nevadí. Ono na ni ještě dojde.

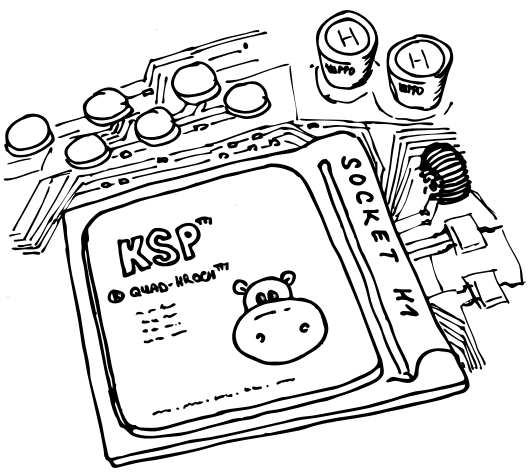
Kuba Maroušek (snad)

30-2-7 Paměť očima assembleru — 15 bodů

Asi vás při čtení minulého dílu napadlo, že pro spoustu úloh si s třinácti 32-bitovými registry nevystačíme. Pojďme se naučit pracovat s pamětí – té máme obvykle k dispozici řádově gigabajty.

Co je paměť vlastně zač?

V běžných programovacích jazycích většinou přistupujeme k paměti prostřednictvím proměnných, polí, objektů atp. Ale to vše jsou jen abstrakce poskytované naším překladačem či interpretem.



Z pohledu procesoru je paměť prostě dlouhá řada okének, každé z kterýchž si pamatuje jeden bajt, tedy číslo od 0 do 255. Těmto okénkům se občas říká paměťové buňky. Každé okénko je jednoznačně určené svým pořadovým číslem, kterému říkáme *adresa*.

Pro začátek řekněme, že adresy mají rozsah od 0 do $N-1$, kde N je velikost paměti v bajtech. Časem se ukáže, že situace je o malinko složitější.

Přístup k paměti

ARM patří mezi takzvané *load/store architektury*. To znamená, že většina instrukcí neumí přímo pracovat s pamětí, pouze s registry. Namísto toho existují speciální instrukce sloužící k přenosu dat z paměti do registrů (kde s nimi pak můžeme provádět nějaké výpočty) a z registrů do paměti.

Začneme tím nejjednodušším: čtením a zápisem jednoho bajtu. K tomu slouží instrukce:

- `LDRB` *cílový-registr*`,` *zdrojová-adresa* (LoaD Register Byte) pro čtení z paměti do registru,
- `STRB` *zdrojový-registr*`,` *cílová-adresa* (STore Register Byte) pro zápis z registru do paměti.

Registr se zapisuje, jak jste zvyklí, např. `r3`. Adresu lze zapsat vícero způsoby, ale překvapivě ne jako číselnou konstantu. Asi nejjednodušší zápis je [*registr*], který použije jako adresu obsah nějakého registru.

Takže například instrukce `LDRB r1, [r5]` načte do registru `r1` bajt z adresy uložené v registru `r5`. Obdobně následující posloupnost instrukcí zapíše bajt s hodnotou 42 na adresu 0x10000:

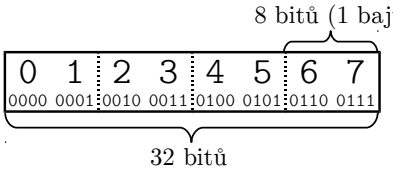
```
MOV  r0, #42
MOV  r1, #0x10000
STRB r0, [r1]
```

Pozor je třeba dát na to, že přístup k paměti je výrazně pomalejší než práce s registry – zhruba 3× až 100×. Proč tak velké rozpětí by bylo na delší povídání – souvisí to s takzvanou *cache* procesoru, o které možná bude řeč v některém z dalších dílů. Zjednodušeně lze říct, že opakovaný přístup k částem paměti, ke kterým jste přistupovali nedávno, bude rychlejší.

Každopádně se vyplatí hodnoty, se kterými provádíte spoustu výpočtů za sebou, držet v registrech a do paměti uložit třeba až na samém konci nějaké série výpočtů, kdy potřebujete registr uvolnit pro jiné účely.

Paměťová reprezentace čísel

Když registry i aritmetické operace pracují s 32-bitovými čísly, hodilo by se nám tato čísla ukládat do paměti. Do jednoho bajtu se vejde 8 bitů, takže k uložení jednoho čísla potřebujeme 4 bajty. Uvažujme například číslo 0x1234567. To můžeme rozdělit na bajty následovně:



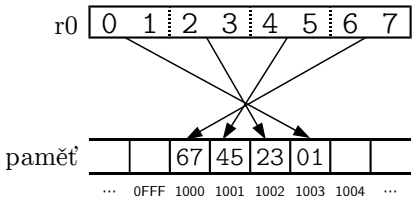
Existují dva běžné způsoby, jak takové číslo do paměti uložit, které se liší pořadím těchto bajtů v paměti. *Big endian* znamená uložení bajtů v pořadí od nejvýznamnějšího po nejméně významný, jak by je asi přirozeně zapsal člověk. *little endian* znamená pořadí přesně opačné, tedy naše číslo by bylo zapsané sekvencí bajtů 0x67 0x45 0x23 0x01. Na ARMu se obvykle používá právě little endian (stejně jako na intelských procesorech).

A přestože by se dalo číslo uložit a načíst vhodnou kombinací `STRB`/`LDRB` a aritmetických instrukcí, je to natolik běžná operace, že pro ni ARM nabízí speciální instrukce: `STR` *registr*`,` *adresa* a `LDR` *registr*`,` *adresa*. Význam parametrů je stejný jako u bajtových verzí, k uložení čísla se použijí paměťové buňky *adresa* až *adresa* + 3.

Pokud se v `r0` nachází číslo 0x1234567 a provedeme instrukce:

```
MOV r1, #0x10000
STR r0, [r1]
```

bude výsledek vypadat následovně:



Je dobrým zvykem ukládat čísla na adresy, které jsou násobkem velikosti daného typu – v tomto případě násobkem čtyř.

Existují i další varianty load/store instrukcí. Kompletní přehled ukazuje následující tabulka:

	bitů	**znaménkovost**	**min.**	**max.**
`LDRB`	8	bezznaménkově	0	255
`LDRSB`	8	znaménkově	−128	127
`STRB`	8	nezáleží	dle znaménkovosti	
`LDRH`	16	bezznaménkově	0	65 535
`LDRSH`	16	znaménkově	−32 768	32 767
`STRH`	16	nezáleží	dle znaménkovosti	
`LDR`	32	nezáleží	dle znaménkovosti	
`STR`	32	nezáleží	dle znaménkovosti	

Úkol 1 [1b]: Vysvětlete, proč zatímco `LDRB` a `LDRH` mají znaménkovou a bezznaménkovou variantu, `LDR` a všechny store instrukce jsou společné pro znaménková i bezznaménková čísla.

Proměnné a paměťové reprezentace

Ve vyšších programovacích jazycích jsme zvyklí pracovat i s jinými typy, než jen čísly. Ukážeme si, jak různé typy reprezentovat v paměti. *Paměťovou reprezentací* nějakého typu rozumíme schéma určující, jak převést libovolnou hodnotu daného typu na posloupnost bajtů v paměti (a zpět). Reprezentace obvykle mají pevnou velikost, abychom si pro ně mohli vyhradit místo v paměti.

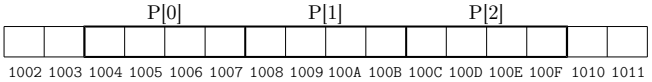
Proměnná pak je prostě vyhrazený úsek paměti obsahující hodnotu proměnné uloženou dle paměťové reprezentace dané typem proměnné.

- **Celá čísla** jsou reprezentována 1 až 4 bajty, jak bylo popsáno výše, dle potřebného rozsahu. Velikost reprezentace je neměnná: udává maximální číslo, které daná proměnná může uchovat. Ale pokud do 32-bitové proměnné uložíte třeba jedničku, stále bude zabírat v paměti 4 bajty.
- **Desetinná čísla** se ukládají v takzvaném formátu s plovoucí čárkou (floating-point, IEEE 754). Čísla se ukládají ve tvaru $m \cdot 2^e$, kde čísla m (tzv. *mantisa*) a e (*exponent*) jsou uložena zvlášť a každému je vyhrazen nějaký počet

bitů. Díky tomuto zápisu mají floatové typy obrovský rozsah, ale omezenou přesnost.

Existuje několik variant, které se liší velikostí, nejčastěji potkáte takzvanou double precision (typ `double` v Céčku), která zabírá 64 bitů, z toho 53 bitů tvoří mantisa a 11 exponent. Díky tomu umožňuje reprezentovat čísla v rozsahu řádově od -2^{1000} do 2^{1000}, ale pamatuje si jen 53 nejvýznamnějších dvojkových číslic. Práce s desetinnými čísly je na ARMu trochu komplikovanější a v seriálu se jí věnovat nebudeme.

- **Pole** vytvoříme tak, že prostě uložíme spoustu reprezentací daného typu těsně za sebe. Například pole 32-bitových celých čísel o ℓ prvcích bude souvislý úsek paměti délky 4ℓ. Tady je důležité, že reprezentace jednotlivých prvků mají pevnou velikost. Díky tomu dokážeme spočítat v konstantním čase vzdálenost libovolného prvku od začátku pole (tzv. *offset*, a tedy i jeho adresu. V našem příkladu má i-tý prvek offset $4i$ a nachází se na adrese $A + 4i$, kde A je adresa začátku pole. ARM navíc nabízí užitečné zkratky pro přístup k prvkům pole, které si ukážeme níže.



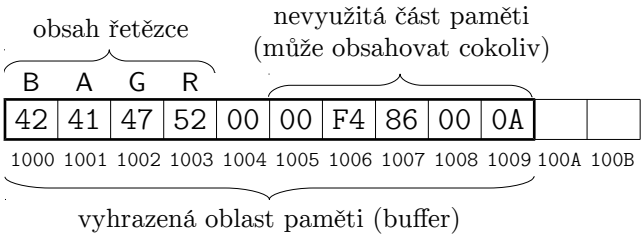
Abychom si pro ně mohli vyhradit místo v paměti, musí mít pole pevnou nejen velikost jednoho prvku, nýbrž i počet prvků.

Občas by se nám ale hodilo mít pole s proměnlivým počtem prvků. Pokud známe nějakou rozumnou horní hranici na to, kolik prvků v poli nejvýše bude, můžeme si v paměti vyhradit prostor odpovídající tomuto limitu (obvykle nazývanému *kapacita* pole) a pak z něj využít jen aktuálně potřebnou část. To typicky znamená, že si někam vedle uložíme počet prvků, které jsou v poli opravdu uložené (k), potom prvních k prvků pole obsahuje smysluplné hodnoty a zbytek ignorujeme.

Alternativně můžeme konec obsazené části pole poznat tak, že si za něj přidáme nějakou speciální značku, která se v normálních datech nevyskytuje, například 0 nebo -1.

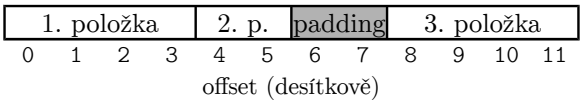
- **Řetězce** můžeme chápat prostě jako pole znaků, ať už „znak“ znamená cokoli. Pokud se obejdeme bez diakritiky, můžeme znaky reprezentovat ASCII kódy[1] a každý znak zabere jeden bajt. Povídání o tom, jak zacházet s unicodovými řetězci, by vydalo na samostatný seriál.

  U řetězců opět narážíme na to, že mohou mít proměnlivou délku. Zde je nejčastějším řešením (vycházejícím z Céčkové tradice) ukončit platnou část řetězce nulovým bajtem. Například řetězec obsahující slovo `BAGR` by mohl vypadat následovně:



- **Struktury a objekty** reprezentujeme podobně jako pole, tedy prostě nasázíme do paměti jednou položku za druhou, jen tentokrát může mít každá jiný typ a velikost. Ale protože seznam položek, jejich typy i pořadí jsou pevné a dopředu známé, můžeme opět snadno spočítat offset každé položky od začátku struktury. Ten je neměnný, můžeme si ho klidně spočítat s tužkou a papírem a napsat do programu jako konstantu (v kompilovaných jazycích tohle udělá překladač).

  Ještě je třeba dát pozor na jednu věc. V minulém dílu jsme zmiňovali, že je vhodné, aby číselné typy byly v paměti zarovnané na násobek své velikosti (některé verze ARMu to dokonce vyžadují). Proto se občas mezi prvky struktury nechává vhodně velká mezera (*padding*), aby následující prvek byl správně zarovnaný. Příklad struktury, která obsahuje 32bitové, 16bitové a znovu 32bitové číslo (v Céčku bychom zapsali jako `struct s { int a; short b; int c; }`):



  V takovéto struktuře víme, že 3. položka bude mít vždy offset 8, takže pro přístup k ní stačí přičíst 8 k adrese začátku struktury. Aby nám zarovnání správně vyšlo, je třeba i začátek struktury mít zarovnaný na násobek čtyř.

- **Ukazatele či reference** (např. vzájemné odkazy mezi prvky spojového seznamu či vrcholy binárního stromu) ukládáme prostě jako 32-bitové číslo obsahující adresu cíle (kterým je často nějaká struktura). Níže si ukážeme příklad reprezentace spojového seznamu. Null pointer („ukazatel nikam“) reprezentujeme číslem 0. To znamená, že na adresu 0 bychom neměli nic ukládat, protože ukazatel na takovou věc by nešlo rozpoznat od null pointeru. Ale to vám operační systém ani nedovolí.

Pole a režimy adresování

Pojďme se nyní pozorněji podívat na to, jakými způsoby se dá zapsat adresa v load/store instrukcích. Tři základní varianty jsou následující:

- `[`*registr*`]`, např. `[r4]` – adresou je obsah registru. Tento zápis už jsme potkali výše.
- `[`*registr*`,#`*konstanta*`]`, např. `[r2,#10]` – jako adresa se použije součet obsahu registru a číselné konstanty (může být i záporná).
- `[`*registr*`,`*registr*`]`, např. `[r2,r7]` – jako adresa se použije součet hodnot obou registrů.
- `[`*registr*`,`*registr*`,LSL #`*posun*`]` – k hodnotě prvního registru se přičte hodnota druhého registru posunutá o daný počet bitů doleva, tedy vynásobená 2^{posun}. Například `[r3,r2,LSL #3]` odpovídá adrese $\mathrm{r3} + 2^3 \cdot \mathrm{r2} = \mathrm{r3} + 8 \cdot \mathrm{r2}$.

Varianta s konstantním posunem se hodí, když máme spoustu číselných proměnných. Pokud je máme v paměti blízko sebe, nemusíme si před každým čtením nějaké proměnné připravit do registru její přesnou adresu. Místo toho si v jednom registru budeme udžovat začátek celé oblasti a k jednotlivým proměnným přistupovat pomocí tohoto registru a různých offsetů. Například pokud máme číselné proměnné na adresách 0x10000, 0x10004, 0x10008, ..., uložíme si např. 0x10000 do r10 a pak k jednotlivým proměnným přistupujeme instrukcemi typu `LDR r1, [r10, #8]`.

Taktéž se hodí pro přístup k položkám struktur: pokud máme v r0 adresu struktury a chceme přistoupit k její položce s offsetem 10, můžeme použít `LDR r5, [r0, #10]`.

[1] `https://cs.wikipedia.org/wiki/ASCII`

Dvouregistrové adresování, zvlášť ve verzi s bitovým posunem, se naopak hodí pro práci s poli. Například máme-li v r0 adresu pole a v r1 index, můžeme příslušný prvek přečíst pomocí `LDR r5, [r0, r1, LSL #2]`.

Následující kód projde pole 32-bitových čísel bez znaménka začínající na adrese 0x10000 o 1024 prvcích a vypíše index maximálního prvku:

```
MOV r0, #0x10000
MOV r1, #0  // index při procházení
MOV r2, #0  // prozatímní maximum
MOV r3, #-1 // index prozatímního maxima
smycka:
  LDR r4, [r0, r1, LSL #2] // načte prvek pole
                           // z adresy r0 + 4*r1
  CMP r4, r2
  BLO neni_vetsi
    // pokud je větší nebo rovno...
    MOV r2, r4 // ...nahradíme maximum...
    MOV r3, r1 // ...a uložíme jeho index
  neni_vetsi:
  ADD r1, #1
CMP r1, #1024
BLO smycka
// na konci je v r3 index maxima
```

Z toho, jak fungují pole, vidíme, proč musí mít pevnou velikost. Poté, co si pro pole najdeme nějaké místo v paměti, můžeme přidávat prvky maximálně tak dlouho, dokud konec pole nenarazí na začátek něčeho jiného, co je v paměti uložené o kus dál.

ARM má ještě nabízí ještě další nezvyklé adresovací módy, které usnadňují procházení polí:

- `[`*registr*`,`*offset*`]!` – použije *registr*+*offset* jako adresu pro load/store instrukci a na konci ji zapíše zpátky do *registru*.
- `[`*registr*`],`*offset* – použije hodnotu registru jako adresu pro load/store instrukci a po jejím provedení do něj zapíše hodnotu *registr*+*offset*.

Offset může opět být konstanta, další registr nebo registr s posunem. Tyto instrukce se chovají trochu podobně jako Céčkový prefixový a postfixový operátor `++`. Použitím těchto adresovacích módů můžeme jednou instrukcí přečíst prvek z pole a skočit na další, ušetříme si tak jednu instrukci `ADD`.

Ukážeme si to na příkladu kódu, který sečte všechny prvky pole (opět od 0x10000 délky 1024):

```
MOV r0, #0x10000
MOV r2, #0
smycka:
  LDR r1, [r0], #4
  ADD r2, r1
CMP r0, 0x10400 // pokud je r0 před koncem pole
BLO smycka
```

Úkol 2 [3b]: V paměti na adrese 0x10004 máte pole 32-bitových celých čísel a na adrese 0x10000 32-bitové celé číslo udávající jeho délku. Vaším úkolem je toto pole obrátit pozpátku na místě (aby se na místě prvního prvku ocitl poslední, ..., až na místě posledního první). Ideálně byste se měli obejít bez další pomocné paměti.

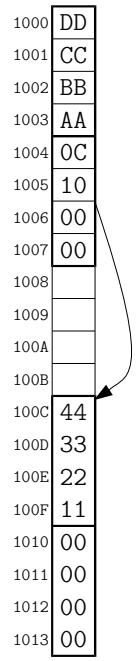
Úkol 3 [3b]: V registru `r0` dostanete číslo N. Napište program, který vynuluje souvislý blok N bajtů v paměti začínající od adresy 0x10000. Program smí celkem provést nejvýše $0.3 \cdot N$ instrukcí (plus konstanta nezávislá na N).

Úkol 4 [5b]: V paměti na adrese 0x10004 máte pole 32-bitových celých čísel se znaménkem a na adrese 0x10000 32-bitové celé číslo udávající jeho délku (můžete předpokládat, že je to mocnina dvojky). Vaším úkolem je toto pole setřídit. Pro plný počet bodů implementujte nějaký efektivní třídící algoritmus (s lepší než kvadratickou složitostí).

Můžeme doporučit např. nerekurzivní (bottom-up) variantu MergeSortu popsanou v naší kuchařce,[2] případně vhodně implementovaný RadixSort. Máte k dispozici pomocnou paměť velkou jako původní pole (plus nějaká konstanta) od adresy 0x8000000. Výsledné setříděné pole můžete uložit buď místo původního, nebo do této pomocné oblasti.

Příklad: spojové seznamy

Podíváme se na příklad trochu složtější datové struktury, totiž (jednosměrného) spojového seznamu. Ten ve vyšších programovacích jazycích obvykle reprezentujeme jako spoustu struktur (objektů) provázaných ukazateli.



S tím, co jsme si ukázali výše, už víme, jak tyto struktury reprezentovat. Např. bude-li náš seznam obsahovat 32-bitová čísla, bude každý jeho prvek reprezentován souvislým osmibajtovým úsekem paměti. První čtyři bajty budou obsahovat hodnotu prvku, druhé čtyři bajty adresu následujícího prvku.

Poslední prvek má místo adresy následovníka uložen null pointer, tedy nulu.

Na obrázku vpravo je příklad spojového seznamu obsahujícího dvě čísla, 0xAABBCCDD a 0x11223344. Připomínáme, že prvky seznamu mohou být v paměti rozmístěné naprosto libovolně: s mezerami, pozpátku, napřeskáčku, etc.

Následující kód projde seznam a spočítá jeho délku. První prvek seznamu je uložen na adrese 0x10000.

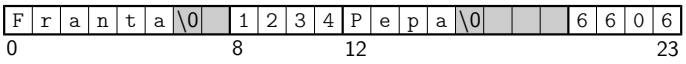
```
MOV r0, #0x10000
MOV r1, #0
smycka:
ADD r1, #1
LDR r0, [r0, 4] // na pozici r0+4 je ukazatel
                // na následující prvek
CMP r0, 0
BNE smycka
// v r1 je délka seznamu
```

Úkol 5 [3b]: V paměti na adrese 0x10000 máte uložený ukazatel na první prvek spojového seznamu (pozor, nikoli přímo první prvek). Seznam obsahuje 32-bitová celá čísla setříděná ve vzestupném pořadí. V registru r0 dostanete adresu nového prvku – kompletní struktury včetně zatím nevyplněného odkazu na následníka. Vaším úkolem je připojit nový prvek na správné místo do původního seznamu, aby zůstal setříděný a aby na adrese 0x10000 stále byl ukazatel na jeho začátek.

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

Příklad: vyhledávání v telefonním seznamu

Máme v paměti pole struktur popisující něco jako telefonní seznam. Každá položka obsahuje dva řetězce: jméno (osmibajtový buffer pro řetězec proměnlivé délky ukončený nulovým bajtem) a číslo (4-znakový řetězec pevné délky bez ukončení). Jednoduchý seznam o dvou položkách by mohl v paměti vypadat takto:



Na obrázku pro přehlednost ukazujeme znaky místo jejich ASCII kódů. „\0“ značí nulový bajt (konvenční zápis z Céčka). Odpovídající Céčková deklarace by vypadala takto:

```
struct polozka {
  char jmeno[8];
  char cislo[4];
};
struct polozka seznam[2];
```

V registru r0 dostaneme ukazatel na řetězec se jménem, ke kterému bychom chtěli v seznamu najít odpovídající číslo. O to se postará následující kus kódu:

```
// r0 - vyhledáváný řetězec
// r1 - adresa začátku pole
// r2 - počet záznamů
// r3 - adresa aktuálně zkoumaného záznamu

MOV r3, r1
MOV r10, #12
MUL r10, r10, r2
ADD r10, r1
// r10 = r1 + 12*r2 (adresa konce pole)

porovnej:
// Porovná řetězce na adresách r0 (hledaný)
// a r3 (aktuální jméno v seznamu). Skočí
// na label "stejne" nebo "ruzne" podle výsledku

MOV r4, #0
znak: // porovnání r4-tého znaku obou řetězců
LDRB r5, [r0, r4] // znak hledaného řetězce
LDRB r6, [r3, r4] // znak jména ze seznamu
CMP r5, r6
BNE ruzne
CMP r5, 0
// narazili jsme na konec řetězce, aniž bychom
// předtím našli neshodu
BEQ stejne
ADD r4, #1
CMP r4, #8
BHS ruzne // řetězec neukončený nulou - chyba
B znak

ruzne:
ADD r3, #12 // přejdeme na další záznam
CMP r3, r10
BLO porovnej
B nenalezeno

stejne: // našli jsme shodující se jméno
ADD r3, #8 // o 8 bajtů dál je číslo
// tady bychom ho mohli třeba vypsat, kdybychom
// to uměli

nenalezeno:
```

Při porovnávání je třeba dát si pozor, abychom se zastavili, když libovolný z řetězců skončí (narazíme na nulový bajt). Protože zbytek bufferu za koncem řetězce může být vyplněný nějakým náhodným smetím, pokud bychom neskončili, mohli bychom dva shodné krátké řetězce vyhodnotit jako různé, protože se liší v této ignorované části.

V kódu výše je to trochu schované: pokud jsou řetězce různě dlouhé a narazíme na konec jednoho z nich, porovnávání skončí, protože se na daném místě znaky liší (jeden řetězec obsahuje nulový bajt a druhý smysluplný znak). Pokud jsou oba řetězce stejně dlouhé, narazíme na oba nulové bajty současně a pomůže nám podmínka `CMP r5, 0`.

Ale zrovna tak je dobré si pohlídat i maximální délku, pokud by se někde nedopatřením objevil řetězec neukončený nulou, aby porovnávání nepokračovalo donekonečna (případně do doby, než narazí na nulu v úplně nesouvisející části paměti).

Paměťová reprezentace instrukcí

Paměť kromě dat, které si tam uložíme, osahuje také kód našeho programu. Tomuto principu se říká von Neumannnova architektura: kód je uložený ve stejné paměti jako data, není pro něj vyhrazené žádné speciální místo. To má spoustu výhod: pokud operační systém načítá kód programu z disku do paměti, může použít stejné instrukce pro práci s pamětí jako pro data (v tuto chvíli ten kód jsou pro něj data). Občas se taky programu může hodit generovat části svého kódu až za běhu.

Jak už jsme naznačili v minulém dílu, program není v paměti uložen jako textový zápis v assembleru, tomu by procesor nerozuměl. Místo toho je každá instrukce (včetně svých parametrů) zakódovaná jedním 32-bitovým celým číslem. Tohle je jedna z vlastností, která činí ARM příjemně jednoduchým; na jiných procesorech mají často různé druhy instrukcí různě dlouhé kódy. Instrukce musí být v paměti zarovnané (uložené na adresách, které jsou násobkem čtyř).

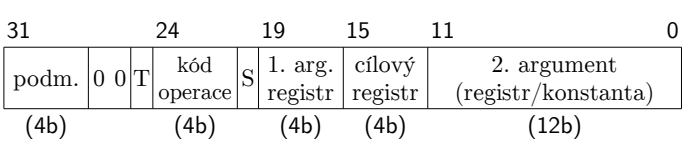
V procesoru existuje speciální registr označovaný `pc` (Program Counter, dostupný též jako `r15`), který obsahuje paměťovou adresu aktuálně vykonávané instrukce. Velmi zjednodušeně bychom si činnost procesoru mohli představit jako neustálé opakování následujících kroků:

1. Načti instrukci z adresy `pc` v paměti
2. Dekóduj a proveď instrukci
3. Pokud instrukce nezměnila `pc` (neprovedla skok), zvyš automaticky `pc` o 4 (přejdi na následující instrukci).

Skutečnost je o dost složitější, protože procesor například zpracovává několik instrukcí částečně paralelně (zatímco se jedna provádí, další už se načítá atp.). To má občas trochu podivné důsledky. Například pokusíte-li se přečíst hodnotu registru `pc` instrukcí typu `MOV r0, pc`, neuloží se do `r0` adresa této instrukce `MOV`, jak by možná člověk čekal, nýbrž hodnota o 8 vyšší (o 2 instrukce dál).

Podívejme se nyní, jak takový kód nějaké instrukce vypadá. Existuje několik různých kódování pro různé druhy instrukcí: jedno pro všechny aritmetické, jedno pro load/store, jedno pro skoky, atd.

Ukážeme si například kódování aritmetických instrukcí (používané i pro další podobné instrukce, jako např. `MOV`):



Podíváme-li se na kód instrukce, najdeme v něm:

- Podmínku. Už v minulém díle jsme stručně zmiňovali, že podmínku jde připojit k většině instrukcí, nejen ke skoku. Daná instrukce se pak provede, pouze pokud je podmínka splněná. Každá podmínka z minulého dílu má svůj 4-bitový kód, např. 0000=EQ, 0010=HS, ... Existuje speciální podmínka AL (ALways, 1110), která zařídí nepodmíněné spuštění dané instrukce. Ale v assemblerovém zápisu ji lze vynechat (píšeme `MOV` místo `MOVAL`).
- Typ druhého argumentu („T“ v diagramu výše). Pokud T=1, druhý argument je konstanta, jinak je to registr.
- Kód operace (opkód), který říká, o jakou instrukci vůbec jde. Např. `ADD`=0100, `MOV`=1101.
- Bit S udávající, zda se dle výsledků má nastavit stavový registr. Tímto jsou odlišeny například instrukce `ADDS` a `ADD`.
- Číslo registru, který tvoří první argument. Prvním argumentem musí být vždy registr. Číslo registru je přesně to, které je obsažené v jeho názvu – např. `r5` je reprezentován číslem 5 (0101).
- Číslo cílového registru, kam se uloží výsledek operace.
- Druhý argument (registr nebo konstanta).

Pravděpodobně jste při hraní s naším simulátorem narazili na to, že některé konstanty nejde v assembleru zapsat (překladač si stěžuje, že jsou příliš velké). Teď už víte proč: na konstantní argument v instrukci zbývá jen 12 bitů, takže tam určitě libovolné 32-bitové číslo nevtěstnáme.

Autoři ARMu ale těchto 12 bitů využili velmi chytře. Namísto jednoho 12-bitového čísla (s rozsahem 0 až 4096) je rozdělili na dvě části: 8-bitovou hodnotu (x) a 4-bitovou rotaci (r). Hodnota druhého operandu vznikne jako x doplněné nulami zleva na 32 bitů a následně bitově zrotované doprava o $2r$ bitů. Pro $r \geq 4$ se tato rotace chová jako bitový posun doleva (rozmyslete si). Takže takto dokážeme snadno vytvářet konstanty tvaru $x \cdot 2^k$ pro malá x.

Filip Štědronský

Recepty z programátorské kuchařky: Binární vyhledávání

Binární vyhledávání je mocná technika, která se objevuje ve všemožných algoritmech a úlohách. Obecně spočívá ve využívání monotónnosti nějakého pole nebo funkce k rychlejšímu prohledávání. To zní pravděpodobně velmi nejasně; začneme proto konkrétnějšími příklady a budeme postupně zobecňovat.

Ve své nejjednodušší podobě je binární vyhledávání technika, která nám umožní rychle prohledávat seřazené pole. Řekněme, že máme vzestupně setříděné pole A, o kterém víme, že obsahuje číslo k. Chceme zjistit, jaký má k index v poli, čili pro jaké i platí $A[i] = k$. Budeme v podstatě hádat, který index to je, a zpřesňovat náš odhad. Napřed odhadneme index v polovině pole (ten označíme *mid* jako middle, čili střed).

- Pokud $A[mid] = k$, je hotovo.
- Pokud $A[mid] < k$, všechny prvky A nalevo od *mid* musí být také menší než k. Proto vyhledávání spustíme znovu, ale pouze na prvcích napravo od *mid* (samozřejmě už bez *mid*).
- Pokud $A[mid] > k$, analogicky spustíme vyhledávání na levé polovině.

Pokud pole nemá přesný střed (má sudou délku), zvolíme za *mid* a libovolný ze dvou prostředních indexů.

Řekněme například, že chceme v poli $[1, 4, 5, 7, 11, 16, 20]$ zjistit index čísla 11. Vyhledávání bude postupovat takto:

0	1	2	3	4	5	6	
1	4	5	**7**	11	16	20	málo
1	4	5	7	11	**16**	20	moc
1	4	5	7	**11**	16	20	trefa!

Výsledek je tedy 4. Stačilo nám podívat se na tři prvky, takže výrazně méně než 7 v naivním prohledávání.

Protože délka intervalu, na kterém hledáme, se v každé iteraci zmenší alespoň na polovinu, po i-té iteraci bude mít interval délku nejvýše $N/2^i$ (kde N je délka pole). Celkem proto provedeme maximálně $\log_2 N$ iterací, než se dostaneme na délku 1. Časová složitost je proto $\mathcal{O}(\log N)$.

Nejintuitivnější je asi rekurzivní představa, která v kódu vypadá takto:

```
# pole A je dané
# Vyhledávání čísla k, pokud víme, že se
# nachází někde v intervalu <l, r>.
def index_prvku(l, r, k):
    if l > r:  # voláme se na prázdný úsek
        return None  # zde hledané k určitě není

    mid = (l + r) // 2  # průměr hranic
    if A[mid] == k:  # hotovo,
        return mid   # mid je hledaný index
    elif A[mid] < k:  # chceme víc
        # spusť na pravé půlce (ale už bez mid)
        return index_prvku(mid + 1, r, k)
    else:  # poslední možnost: chceme méně
        # spusť na levé polovině
        return index_prvku(l, mid - 1, k)

# Zavoláme na <0, len(A) - 1>, tedy na celé pole
index = index_prvku(0, len(A) - 1, k)
if index is None:
    print("{} se v poli nevyskytuje.".format(k))
else:
    print("{} se v poli vyskytuje na pozici {}"
        .format(k, index))
```

Může se nám stát, že hledané k v poli neleží. To při vyhledávání poznáme tak, že se nám interval, kde k ještě může být, zmenší na prázdný. Komentáře kód značně prodlužují, ale v praxi je to opravdu jen pár řádků. Častěji se však používá implementace, kde místo rekurze použijeme cyklus. Převod je jednoduchý, uvedeme si tedy i tuto verzi:

```
# Pole A je dané
def index_prvku(k):
    # po celou dobu platí, že k se musí
    # nacházet někde v intervalu <l, r>
    l, r = 0, len(A) - 1
    while l <= r:
        # dokud je interval <l, r> neprázdný
        mid = (l + r) // 2
        if A[mid] == k:
            return mid
        elif A[mid] < k:
            l = mid + 1
        else:
            r = mid - 1
    return None
    # Pole neobsahuje k,
    # jinak bychom ho už našli
```

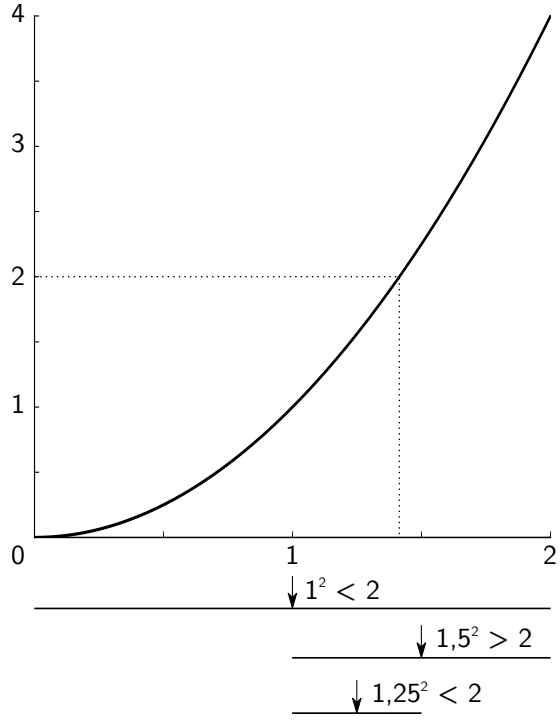
Binární vyhledávání přes funkce

Uvažme, jak by se kód změnil, kdybychom pole A neměli uložené v paměti, ale načítali bychom ho například z disku nebo po síti. Pak bychom nejspíš měli nějakou funkci f, které bychom se mohli ptát na jednotlivé indexy a ona by nám vracela příslušné hodnoty. Jediná změna by pak byla v tom, že bychom se místo přistupování k prvkům pole ptali této funkce na odpovídající indexy.

Binární vyhledávání je totiž mnohem mocnější než prosté vyhledávání v poli. Naše funkce pro binární vyhledávání nyní vlastně nepracuje s žádným polem, jen s nějakou funkcí f. Za tu ale můžeme dosadit něco úplně jiného než jen prvky pole. Musíme však dodržet vlastnost, že f je neklesající, aby mohlo vyhledávání fungovat (stejně tak by mohla být nerostoucí, ale tento připad je analogický, takže se budeme zabývat pouze neklesajícími funkcemi).

Zkusíme tedy za f dosadit něco jiného. Řekněme, že chceme pomocí binárního vyhledávání umět počítat třeba druhou odmocninu čísla: máme dané číslo x a chceme najít číslo $mid \geq 0$ takové, že *mid* je odmocnina z x. Jinak řečeno, $mid^2 = x$. Binárním vyhledáváním úlohu vyřešíme tak, že budeme hádat čísla *mid* a vždy srovnáme $f(mid) = mid^2$ s x. Pokud je mid^2 větší než x, *mid* je větší, než chceme. Horní hranici proto nastavíme na toto *mid*. A naopak, pokud je mid^2 menší, nastavíme spodní hranici na toto *mid*. Důležité je, že pro kladná *mid* je funkce f neklesající, jinak bychom na ní binárně vyhledávat nemohli.

Na obrázku je případ, kdy hledáme odmocninu ze dvou. V první iteraci jsme *mid* odhadli na 1 a $f(mid)$ je proto taky 1, takže méně než x (které je 2). Zahodíme proto levou polovinu.



Protože odmocnina může být iracionální, nemůžeme předpokládat, že ji dokážeme spočítat přesně, tzn. že by nastal případ $mid^2 = x$. Odmocninu tedy jen aproximujeme. Pro takovéto aproximační úlohy stačí cyklus zopakovat dostatečněkrát, abychom získali rozumnou přesnost. Přesnost se velmi rychle zvyšuje (s každou iterací se velikost intervalu, kde se může nacházet výsledek, zmenší na polovinu), a proto většinou stačí třeba 100 iterací.

Drobný, ale podstatný detail je volba intervalu $\langle \ell, r \rangle$, ve kterém vyhledávání začínáme – pokud hledané mid leží mimo tento interval (tj. neplatí, že $f(\ell) \leq f(mid) \leq f(r)$), algoritmus samozřejmě nemůže fungovat. Pro náš příklad s $f(x) = x^2$ jsme zvolili $\ell = 0$, $r = \max(1, x)$, rozmyslete si, proč tato volba funguje.

V kódu:

```
def f(a):
    return a * a

def odmocnina(x):
    l, r = 0., max(float(x), 1.)
    for iterace in range(100):
        mid = (l + r) / 2
        if f(mid) < x: l = mid  # chceme víc
        else: r = mid  # chceme míň
    return l
    # l a r jsou dostatečně blízko,
    # je jedno, které vrátíme
```

Kód je velmi podobný předchozímu, přestože dělá něco docela jiného. Oproti minule nepracujeme s celými čísly, ale s čísly reálnými. Z toho plynou změny, které jsme vysvětlili výše. Případ `f(mid) == x` přeskakujeme proto, že chceme jen dostatečně dobrý odhad a v naprosto přesný výsledek nedoufáme.

Funkce f je nyní úplně jiná než předtím, ale vyhledávání funguje analogicky. Důležité je, že hodnotu f zjišťujeme tolikrát, kolik proběhne iterací cyklu (zde stokrát, v celočíselném případě $\mathcal{O}(\log N)$-krát), takže jen málokrát. Její výpočet tak může být i poměrně pomalý a celý program poběží pořád rychle [3] – řádově rychleji, než kdybychom si počítali všechny funkční hodnoty.

Jak se vypořádat se stejnými hodnotami

Přesuňme se zpět do celých čísel. Hned v příkladu s vyhledáváním v poli jsme opomenuli případ, kdy se nějaké číslo v poli vyskytuje víckrát. V takových případech chceme zpravidla najít první nebo poslední pozici prvku. Takto můžeme například najít v seřazeném poli poslední prvek, který má hodnotu maximálně k. Pokusme se k tomuto účelu binární vyhledávání upravit.

Existuje několik způsobů, jak se s tím vypořádat, ale často bývají náchylné na chyby, zejména na plus-minus-jedničkové. Nejjednodušší je pamatovat si nejvyšší index pole, který podmínku splňuje, a ve vyhledávání pokračovat jako obvykle, dokud se nám interval nezmenší na nulový nebo záporný. Může to vypadat takto:

```
# pole A je dané
def neni_vetsi_nez_k(x, k):
    # není větší než k, takže je možným řešením
    return A[x] <= k

def nejvetsi_prvek_ne_vetsi_nez_k(k):
    l, r = 0, len(A) - 1
    nejlepsi = None
    while l <= r:
        # dokud není interval prázdný
        mid = (l + r) // 2
        if neni_vetsi_nez_k(mid, k):
            nejlepsi = mid
            l = mid + 1
        else:
            r = mid - 1
    return nejlepsi
    # Pokud žádný prvek nesplňuje podmínku,
    # vrátí se None
```

Všimněme si, že když najdeme prvek, který podmínku splňuje, zmenšíme interval na pravou polovinu. To znamená, že když pak najdeme další prvek, který také podmínku splňuje, bude nutně lepší. Proto můžeme `nejlepsi` nastavit rovnou na `mid` bez jakýchkoli srovnání.

Vyhledávání podle predikátu

Schválně používáme frázi „prvek, který splňuje podmínku“ místo něčeho jako „prvek, který není větší než k“. Můžeme totiž udělat další abstrakci – při vyhledávání nám nezáleží na porovnávání nějakých čísel, jen na tom, jestli prostřední hodnota splňuje nějakou podmínku, čili predikát. Důležité je, že tento predikát je „nerostoucí“, čili na začátku je na nějakém úseku vždy splněný (vrátí `true`) a dále už nikdy. My hledáme právě hranici mezi těmito částmi: nejvyšší hodnotu, pro kterou funkce vrátí `true`.

Formálně, predikát $p(x)$ musí splňovat:

$$p(x) = \texttt{false} \Rightarrow (y > x \Rightarrow p(y) = \texttt{false}).$$

To znamená, že když někde predikát není splněn, dále už nebude splněn nikde.

Predikát je v tomto případě `f(mid) >= hledany`, ale klidně by to mohlo být něco jako „je možné vyskládat x koní na šachovnici tak, aby se neohrožovali?“ Tímto způsobem můžeme řešit úlohy typu „najděte největší k, pro které ještě platí

[3] Samozřejmě v rozumných mezích – bude-li nás výpočet jedné hodnoty stát $\mathcal{O}(2^N)$ času, binárním vyhledáváním to moc nezachráníme.

podmínka“. Někdy se totiž stává, že původní problém není možné jednoduše zodpovědět, ale dokážeme rychle (řekněme v $\mathcal{O}(N)$ nebo $\mathcal{O}(N \log N)$) odpovědět na dotazy typu: „Platí ještě pro dané k *podmínka*?“ Pak stačí implementovat tyto dotazy a binární vyhledávání nám dá odpověď. Například úlohu „Kolik nejvíce koní je možné umístit na šachovnici tak, aby se neohrožovali?“ můžeme redukovat na „Je možné vyskládat x koní na šachovnici tak, aby se neohrožovali?“ za použití binárního vyhledávání.

Uveďme příklad (zjednodušení P-I-1 z MO-P 2015): Máme hotel, který má N (max. 10^6) pater. V každém patře je jeden pokoj. Postupně se do hotelů ubytovává H hostů ($H \leq N$), z nichž i-tý může být ubytován maximálně v patře a_i (protože se bojí výšek). Kolik hostů dokážeme ubytovat, než budeme muset nějakého odmítnout, protože se nikam nevejde?

Řešení: Dokážeme jednoduše zjistit, zda dokážeme ubytovat prvních K hostů. Stačí vzít prvních K, tuto posloupnost seřadit a pak zabírat patra odspoda – host s největším strachem z výšek do prvního patra a tak dále. To zabere $\mathcal{O}(N \log N)$ času. S touto podmínkou už spustíme binární vyhledávání a dostaneme odpověď za $\mathcal{O}(N \log^2 N)$ času.

Ne vždy je predikátová forma úlohy jednodušší než původní úloha, třeba při hledání nejkratší cesty nedokážeme rychle zjistit, zda existuje cesta dlouhá maximálně x. Vždy je ale dobré se zamyslet, zda by binární vyhledávání mohlo pomoci.

Příklady

Papír

(Těžší verze této úlohy byla zadána jako úloha D na ACM ICPC World Finals 2015.)

Máme obdélníkový papír, ve kterém jsou vystřihnuté díry ve tvaru kruhu. Všechny díry jsou celé uvnitř papíru a nepřekrývají se. Kde máme vést řez papírem rovnoběžný s osou x tak, aby byl papír rozdělen na dvě části o stejném obsahu? Děr může být až 100 000.

Vzducholoď

(Převzata z Codeforces, úloha 590B.)

Letíme vzducholodí a chceme se dostat z bodu A do bodu B (body jsou zadané souřadnicemi), ale situace je komplikovaná větrem. Otáčení vzducholodi netrvá žádný čas, ale vzducholoď má maximální rychlost v (oproti vzduchu). Prvních t sekund vane vítr s vektorem (w_x, w_y), po t sekundách se změní na vektor (u_x, u_y) a v tomto směru už zůstane.

Formálně, pokud má vzducholoď oproti větru nulovou rychlost a je na $[x, y]$ a vane vítr (u_x, u_y), po τ sekundách bude vzducholoď na $[x + \tau \cdot u_x, y + \tau \cdot u_y]$. Maximální rychlost vzducholodi je vždy větší než velikost vektoru větru.

Za jak dlouho se nejrychleji dostaneme z A do B? (Maximální povolená odchylka je 10^{-6}.)

Kuchařku pro vás sepsal

Václav Volhejn

30-1-1 Oprava databáze

Cílem bylo spojit všechny záznamy, které spolu mají nějaké „propojení“. Emailové adresy a jména si můžeme představit jako vrcholy neorientovaného grafu a záznamy v databázi jako jeho hrany, které emaily se jménem spojují (pokud si s grafy nerozumíte, můžete se mrknout do grafové kuchařky).[4] Jedné osobě pak libovolná dvojice vrcholů patří, pokud mezi nimi existuje libovolně dlouhá cesta – když se nachází v jedné komponentě souvislosti. Projít všechny komponenty souvislosti už jde jednoduše prohledáváním grafu do hloubky, jen je potřeba dát pozor na to, že není souvislý. Stačí si ale pro každý vrchol pamatovat, jestli už jsme jej navštívili (třeba v nějakém poli boolů). Pak všechny projdeme, a pokud jsme v něm ještě nebyli, spustíme prohledávání do hloubky. Nakonec vypíšeme počet nalezených prvků.

Samotné prohledání grafu nám zabere $\mathcal{O}(N + M)$, kde N je počet vrcholů a M počet hran – procházíme všechny vrcholy a pokaždé spustíme prohledávání do hloubky. Všechny prohledané vrcholy ale označíme jako „už prohledané“, takže se nikdy nespustí dvakrát prohledání stejné komponenty a každá hrana se navštíví jen jednou.

Teď zbývá vymyslet, jak vyrobit graf z databáze. Je potřeba nějakým způsobem mapovat jména a emailové adresy na vrcholy v grafu. Nejjednodušší asi bude každému textovému řetězci přidělit index v poli, do kterého si vrcholy budeme ukládat. To půjde udělat pomocí nějaké vyhledávací datové struktury – projdeme všechny záznamy, najdeme vrchol jména a vrchol emailové adresy, případně vyrobíme nový, přidáme ho do pole, a vyrobíme mezi nimi hranu.

Jako vyhledávací strukturu jste často používali hashovací tabulku, nebo vyhledávací strom, ale asymptoticky rychlejší je použít trii (písmenkový strom), která umí přidávat i vyhledávat prvky v lineárním čase s délkou textového řetězce. Hashovací tabulka to umí skoro stejně rychle, ale není deterministická, takže je konstantní jen v průměrném případě – mohla by tedy být výrazně pomalejší, kdybychom měli velkou smůlu. Vyhledávací strom by provedl logaritmický počet porovnání, které každé potřebuje až $\mathcal{O}(\ell)$, kde ℓ je délka řetězce. Zpracování vstupu by pak zabralo čas $\mathcal{O}(L \cdot \log(n))$, kde L je součet délek všech řetězců a n počet řetězců na vstupu.

Trie je strom, kde každý vrchol má $\mathcal{O}(\Sigma)$ synů (kde Σ je velikost abecedy), v každém vrcholu je uložen jeden znak řetězce a celá cesta do nějakého listu je jeho klíčem. Ve vrcholech, kde nějaký řetězec končí, je pak uložen i odkaz na vrchol našeho grafu. Pro šťouraly, kdybychom chtěli podporovat UNICODE, tak asi nechceme milión synů pro každý vrchol, ale stačí to zakódovat pomocí UTF-8 a pracovat s tím jako řetězci bytů. Případně řetězci bitů, to by fungovalo taky, kdyby bylo i 256 synů moc. Viz kuchařka o vyhledávání v textu,[5] kapitola Adresář pomocí trie.

Případně je možné si pole setřídit, nejdříve podle jmen, pak přidělit indexy a pak to samé podle emailů. Třídění standardním algoritmům trvá $\mathcal{O}(n \cdot \log(n \cdot c))$, kde n je počet prvků a c je složitost porovnání, která je úměrná délce textových řetězců. Nicméně třídit texty jde i lineárně vzhledem k součtu délek, například pomocí trie.

Algoritmus provede při stavbě grafu maximálně dvě operace s vyhledávací strukturou – vyhledání vrcholu a případně vložení nového, kde každá operace trvá trii lineárně s délkou textového řetězce. Což může být docela podstatné, v zadání nikdo neslíbil, že všichni mají hezkou a krátkou emailovou adresu, a často jste na to zapomínali. Pak je třeba graf projít – což trvá $\mathcal{O}(N + M)$. Dohromady to tedy trvá lineárně dlouho s velikostí vstupu v bajtech, protože počet hran i vrcholů je určitě menší než počet bajtů na vstupu.

Někteří jste také úlohu řešili pomocí datové struktury DFU (Disjoint-Find-Union), která nám umožní si pamatovat, co je s čím v jedné komponentě, a rychle tyto komponenty spojovat, když se nám vyskytne nový záznam, který „spojí“ dva lidi dohromady. Pro tuto úlohu to sice bylo maličko asymptoticky nevýhodné, ale rozhodně by se to hodilo na online verzi úlohy – kdybychom chtěli postupně zadávat záznamy a už v průběhu načítání dat vědět, kolik je to doopravdy osob. Více informací o této datové struktuře je v kuchařce o kostrách v grafu.[6]

Program (C):
`http://ksp.mff.cuni.cz/viz/30-1-1.c`

Pali Rohár & Standa Lukeš

30-1-2 Telefonní hlavolam

Představme si, že stojíme před telefonem, už jsme si vyžádali nějaké nápovědy a nyní přemýšlíme, zda můžeme s jistotou zmáčknout nějaké tlačítko a případně které. Určitě nemá smysl uvažovat ta tlačítka, o kterých z nějaké nápovědy víme, že je máme zmáčknout až po nějakém jiném. Zaměřme se tedy na ta ostatní a říkejme jim třeba *předvoj*.

Klíčem k optimálnímu řešení je si uvědomit, že tlačítko můžeme zmáčknout, jen pokud je v předvoji jediné. Proč tomu tak je? Představme si, že máme v předvoji nějaká dvě tlačítka X a Y. Správné pořadí je, dejme tomu, $XAB \ldots YPQ \ldots$ Všimněte si, že když Y z tohoto pořadí „vytrhneme“ a dáme na začátek, tak nám to žádnou nápovědu „neporuší“. A tedy v současnosti ještě nemůžeme vědět, které z těchto pořadí je správné, takže nemůžeme s jistotou zmáčknout X ani Y. Naopak pokud už je v předvoji jediné tlačítko, tak jej jistě zmáčknout můžeme. Před ostatními tlačítky totiž musíme zmáčknout nějaké jiné, a jistě tedy nejsou první.

Jakmile zmáčkneme nějaké tlačítko, můžeme všechny nápovědy, které se ho týkají, zapomenout (už nám k ničemu nejsou) a celý postup opakovat se zbylými tlačítky (s využitím nápověd, které nám po „zapomenutí“ zbyly). Takto budeme postupně mačkat tlačítka v jediném správném pořadí, dokud nezmáčkneme všechna. Už víme, že na to spotřebujeme co nejméně nápověd, protože na nápovědu se ptáme jen tehdy, když je výsledné pořadí nejasné.

Z toho plyne jednoduchý algoritmus: Po každé nápovědě projdeme všechna ještě nezmáčknutá tlačítka a pro každé si spočítáme, kolik tlačítek máme podle nápověd zmáčknout před ním. Pokud nemáme před daným tlačítkem zmáčknout

[4] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`
[5] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`
[6] `http://ksp.mff.cuni.cz/viz/kucharky/minimalni-kostry`

žádné jiné, přidáme ho do předvoje. Pokud je po zkontrolování všech tlačítek v předvoji jen jediné, zmáčkneme ho, zapomeneme o něm všechny nápovědy a postup opakujeme. Tento postup funguje, bohužel se nám ale může stát, že pro každou novou nápovědu musíme při počítání tlačítek projít skoro všechny předchozí, takže časová složitost takového řešení bude $\mathcal{O}(M^2)$.

Neděláme však něco zbytečně? Všimneme si, že znovu a znovu u každého tlačítka přepočítáváme, kolik tlačítek musíme zmáčknout před ním. Tato čísla se ale příliš nemění: pro nějaké tlačítko B se jeho čítač změní, když dostaneme nějakou nápovědu (A, B) (to se pak čítač zvýší o jedna), nebo když naopak po zmáčknutí nějakého tlačítka A nápovědu (A, B) zapomeneme (čítač o jedna snížíme).

Budeme si tedy u každého tlačítka toto číslo pamatovat a při získání/zapomenutí nápovědy ho jednoduše přepočítáme. K tomu si pro každé tlačítko A musíme pamatovat všechny jeho nápovědy (A, B) (tj. jen všechny nápovědy „A zmáčkneme dříve než B“), abychom po jeho zmáčknutí věděli, jaké čítače máme snížit o jedničku. Kromě toho si taky budeme pamatovat aktuální předvoj, abychom uměli rychle kontrolovat, kdy už je v něm jen jedno číslo, a jaké.

Algoritmus je pak přímočarý:

Dokud máme nějaké nezmáčknuté tlačítko:

- Dokud je v předvoji víc jak jedno tlačítko, ptáme se na nápovědy: dostaneme nějakou nápovědu (A, B), a pokud jsou A i B ještě nezmáčknuté (tj. nápověda je relevantní), zvýšíme čítač v B o jedničku a u A si nápovědu poznamenáme. Pokud bylo B v předvoji (mělo čítač na nule), tak ho z něj odstraníme.
- Dokud je v předvoji jen jediné tlačítko: zmáčkneme ho, podíváme se na všechny nápovědy (A, B), které jsme si k němu poznamenali, a všem B snížíme čítač o jedničku. Ta čísla, jejichž čítač jsme snížili na nulu, přidáme do předvoje.

Zbývá si rozmyslet, jak toto všechno udržovat. K počítání předcházejících tlačítek nám stačí obyčejné pole čísel, pro seznam nápověd, které po zmáčknutí tlačítka projdeme, můžeme použít pole polí (nebo spojových seznamů). Pro předvoj můžeme použít například hešovací tabulku; pokud by vám vadilo, že tak získáme konstantní vkládání a mazání pouze v průměrném případě, zkuste si rozmyslet, jak k tomuto účelu upravit obyčejný spojový seznam.

Spotřebujeme nejvýše M nápověd, pro každou nápovědu provedeme jen konstantně mnoho práce. Druhý krok sice může trvat dlouho, ale dohromady každé tlačítko zmáčkneme právě jednou a každou nápovědu také zapomeneme právě jednou. Celková časová složitost je tedy $\mathcal{O}(N+M) = \mathcal{O}(M)$, stejně jako paměťová.

Na závěr poznamenáme, že úloha se dala poměrně přímočaře převést na hledání topologického uspořádání v postupně budovaném grafu (nevíte-li, co je to graf nebo topologické uspořádání, zkuste se podívat do naší grafové kuchařky).[7] Myšlenka algoritmu je stejná, jen bychom místo o tlačítkách a nápovědách hovořili o vrcholech a grafech.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-1-2.py`

Dominik Smrž & Říša Hladík

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

30-1-3 Placení v čajovně

Mnoho z vás se pokusilo jít na problém placení co možná nejtěžšími mincemi hladově. Většinou tak, že jste si spočetli poměr váhy ku hodnotě mince a postupně jste platili od mince s největším poměrem (abyste se zbavili co možná největší váhy).

Tento postup ale nedá nejlepší kombinaci mincí, vyvrátíme to jednoduchým protipříkladem. Představte si, že třeba existují mince o hodnotách 1, 3 a 5, kde mince s hodnotou 1 váží jednu tunu, mince o hodnotě 5 váží jeden kilogram a mince o hodnotě 3 váží jeden gram (a obvyklá peněženka obyvatel této země má podobu slušně velkého nákladního auta).

Protože nechceme, aby naše peněženka vážila více, jak my samotní, tak v ní máme jenom lehčí mince o hodnotách 3 a 5 a chceme s nimi zaplatit částku 21. Hladový postup by používal minci o hodnotě 5, dokud by se vešla (tedy čtyřikrát), a pak by se pokusil přeplatit o co možná nejméně. No a ouha, přeplatili bychom tak požadovanou částku o 2 a čajovník by nám vrátil dvě těžké mince o hodnotě 1. Správné řešení je očividně zaplatit pomocí tří mincí o hodnotě 5 a dvou o hodnotě 3.

Hladový postup dokonce ani nemusí vrátit validní kombinaci mincí. Zrecyklujme příklad výše, jenom zrušme minci o hodnotě 1. Stejně jako předtím bychom přeplatili o částku 2, ale čajovník by neměl žádný způsob, jak nám částku 2 vrátit.

Když jsme si tedy primitivní hladové postupy vyvrátili, zkusme se na to podívat trošku jinak.

Do čajovny lépe a s batohem

Nejdříve si vyřešíme lehčí úlohu: jakých hodnot a hmotností umíme dosáhnout pomocí kombinace nanejvýše K mincí N různých typů? Jedna z možností, jak to spočítat, je spustit rekurzi hloubky K a na každé úrovni rekurze se rozhodnout, jakou z $N + 1$ mincí (přidáme si jednu virtuální minci znamenající „nepoužít nic“) zvolit. To nám ale dá čas $\mathcal{O}(N^K)$ a přitom spousta větví výpočtu povede k tomu samému – třeba protože nám vůbec nezáleží na pořadí, v jakém mince vybereme.

Vyřešme tento problém lépe. Pojďme se dívat, jaké částky umíme dosáhnout s použitím pouze jedné mince. Pak zkusme z každé této částky vyjít a podívejme se, jaké všechny částky umíme dosáhnout s použitím dvou mincí a tak dále.

Připravme si dvourozměrnou tabulku, kde řádky budou značit počet použitých mincí a sloupce budou odpovídat poskládané hodnotě. Řádků budeme potřebovat K, počet sloupců si omezme nějakou maximální částkou M. Tabulka tedy bude mít rozměr $K \times M$. Vyplněné políčko $[i, j]$ v tabulce bude znamenat, že umíme s použitím i mincí dosáhnout částky j.

Dobře, ale kam se nám ztratila hmotnost? Budeme ji psát přímo do políček tabulky. Třeba pro příklad výše s mincemi 1, 3 a 5 budeme mít políčko $[2, 2]$ vyplněné váhou 2 tuny (protože pomocí dvou mincí umíme poskládat hodnotu 2 jako dvě mince o hodnotě 1). Co ale s políčkem $[2, 6]$? To umíme poskládat jako dvě mince hodnoty 3, ale také jako jednu minci hodnoty 1 a jednu minci hodnoty 5. Druhá možnost je těžší, a tak zde zapíšeme tu.

Tabulku budeme postupně vyplňovat po řádcích. Na začát-

ku vyplníme v prvním řádku políčka odpovídající jednotlivým mincím (kdyby nějaké dvě mince měly stejnou hodnotu, zapíšeme sem pochopitelně tu těžší) a pak budeme postupně vyplňovat další řádky. Na každé vyplněné políčko z předchozího řádku zkusíme aplikovat všech $N + 1$ mincí (včetně té naší virtuální s hodnotou a váhou nula), a pokud nám tato kombinace dá větší hmotnost, než je zapsaná na odpovídajícím políčku ve vyplňovaném řádku, tak políčko přepíšeme novou kombinací.

Výsledky na konci výpočtu najdeme na posledním řádku. Pro získání seznamu mincí, který vedl na nějakou hodnotu, nám stačí si u každé hodnoty poznamenávat, odkud z předchozího řádku jsme na ni přišli. Pro rekonstrukci pak stačí projít tyto zpětné odkazy a vypisovat přitom hodnoty mincí.

Tím jsme právě popsali postup, kterému se velmi často říká *batoh* a úloze *problém batohu* (anglicky *knapsack problem*).

Nyní si naši úlohu můžeme rozdělit na dvě menší: jakou nejtěžší sadu mincí můžeme použít k zaplacení nějaké částky a obdobně to samé pro čajovníka. Úlohy se mírně liší v tom, že my máme omezené počty jednotlivých mincí, kdežto čajovník má od každé mince libovolně mnoho kusů, ale na problému to prakticky nic nezmění.

Spočítejme si nejprve, jak může vracet čajovník. Ten má sice neomezený počet kusů každé mince, ale je důležité, že obě strany mají omezený maximální počet mincí, které mohou k placení použít – v zadání jsme tyto limity označily jako K a L. Čajovník může zaplatit maximálně L mincemi, ale u každé z nich se může rozhodnout, která mince to bude.

Budeme tedy potřebovat tabulku vysokou L řádků. Šířku tabulky můžeme omezit maximální částkou, kterou můžeme zaplatit. Bohužel nelze udělat žádný odhad typu „maximálně dvojnásobek částky k zaplacení“ – zkuste si třeba zaplatit částku 45, pokud máte k dispozici jenom mince o ceně 1 000 000 007 a čajovník vám může vracet ještě navíc mince hodnoty 59 (ano, obě to jsou prvočísla). Správné řešení je zaplatit 42 velkými mincemi, aby vám čajovník mohl vrátit spoustou menších mincí. Nejlepší odhad na šířku tabulky je tak LH_{max}, kde H_{max} je hodnota nejcennější mince.

Teď už jenom tabulku potřebujeme vyplnit. Budeme to dělat postupem popsaným výše – na každé políčko minulého řádku zkusíme aplikovat všech $N + 1$ mincí (včetně mince znamenající „nepoužívám žádnou minci“) a tím získáme políčka v dalším řádku. V posledním řádku tak získáme všechny možné hodnoty, které může čajovník poskládat, a ke každé z nich i nejtěžší kombinaci mincí. Vyplnění této tabulky nám zabere čas $\mathcal{O}(LH_{max}N)$, protože na každém políčku tabulky můžeme potřebovat ozkoušet všechny možné mince.

Nyní to samé budeme potřebovat udělat pro nás. Na naší straně je ale problém ztížen tím, že máme omezené počty jednotlivých mincí. Mohli bychom si u každého políčka ještě ukládat odkaz na další pole velikosti N, kde bychom si mince počítali, ale to by nás zbytečně zdržovalo.

Místo toho udělejme jednoduché pozorování, že na pořadí mincí nezáleží. Budeme tedy skládat částky tak, že si u každého záznamu v tabulce budeme pamatovat jen typ nejvyšší použité mince a počet již použitých mincí tohoto typu a dovolíme si na toto políčko navázat jenom mincí stejné nebo vyšší hodnoty. Při použití stejného typu mince musíme samozřejmě ověřit, že počítadlo nepřekročí dostupný počet mincí, a toto počítadlo inkrementujeme. Při použití vyšší mince počítadlo nastavíme na jedničku. A pokud na stejné políčko umíme se stejnou váhou přijít více způsoby, budeme preferovat ten s menším typem mince (a v případě stejných typů ten s menším počtem).

Tuto tabulku zvládneme vyplnit v čase $\mathcal{O}(KH_{max}N)$.

Finální krok

Nyní máme vyplněné tabulky toho, jaké částky (a díky zpětným odkazům i jakou kombinací mincí) umíme na naší straně i na straně čajovníka poskládat. Z obou tabulek nás budou zajímat jenom poslední řádky (ve kterých jsou kumulované všechny výsledky pro jakýkoliv počet mincí), pojďme v nich najít nejlepší řešení.

Chceme zaplatit za čaj v hodnotě H, a to tak, že můžeme i přeplatit a čajovník nám obnos vrátí. Přitom chceme skončit s co nejlehčí peněženkou.

Zkusíme tedy projít všechny možné částky v posledním řádku naší tabulky a ke každé z nich budeme hledat v posledním řádku čajovníkovy tabulky částku k vrácení (k nějakým částkám nemusí existovat, takovým způsobem tedy zaplatit nemůžeme). Naši tabulku budeme procházet od nejmenší k největší částce, čajovníkovu naopak, a oba dva řádky nám stačí projít jednou. Přitom si budeme počítat výslednou váhu naší peněženky a na konci vydáme to nejlepší řešení.

Čaj nakoupen, teď hurá na pana Nápovědu!

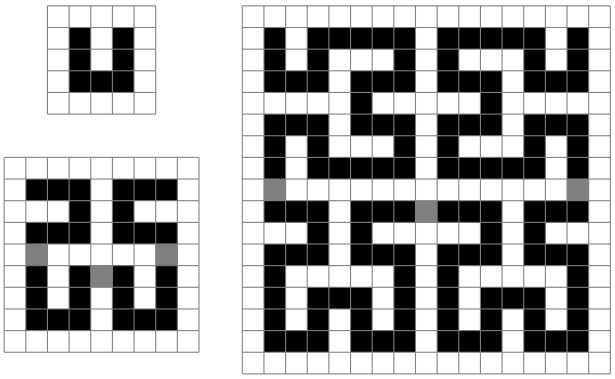
Jirka Setnička

30-1-4 Cesta v bunkru

Nejprve detailně prozkoumáme, jak Hilbertovo bludiště vypadá a jaké má vlastnosti.

Anatomie Hilbertova bludiště

Připomeňme si obrázek ze zadání s bludišti řádů 1, 2 a 3:



Bludiště řádu r se skládá ze čtyř *kvadrantů* (čtvrtin), což jsou různě otočená bludiště řádu $r-1$. Levý horní kvadrant je otočený o 90° proti směru hodinových ručiček, pravý horní o 90° po směru, oba dolní jsou v původní poloze.

Bludiště je ohraničeho *okrajem* z prázdných políček. Okraje jednotlivých kvadrantů se překrývají. Část z nich tvoří okraj celého bludiště; zbytek, který leží uvnitř bludiště, tvoří *tunely*. Všechny tunely se potkávají ve středu bludiště; podle toho, kterým směrem ze středu vedou, je můžeme pojmenovat levý, pravý, horní a dolní tunel.

V tunelech ovšem leží 3 *závaly* – dodatečné zdi spojující jednotlivé kvadranty (šedá políčka na obrázku). Ty dělí tunely na dvě komponenty souvislosti: v jedné je levý, pravý a horní tunel, v druhé dolní tunel.

Nyní dokážeme, že bludiště řádu r měří $(2^{r+1}+1)\times(2^{r+1}+1)$ políček (této délce strany budeme říkat *velikost* bludiště). Důkaz povedeme indukcí podle řádu bludiště. Bludiště řádu 1 má velikost 5. Bludiště řádu $r>1$ se skládá z kvadrantů řádu $r-1$. Ty mají podle indukčního předpokladu velikost 2^r+1. Jelikož se jejich okraje překrývají o jedno políčko, velikost celého bludiště činí $2(2^r+1)-1 = 2\cdot 2^r+1 = 2^{r+1}+1$. Tím je indukční krok hotov.

Ještě si všimneme jedné užitečné vlastnosti. Pokud z bludiště odstraníme okraj, zbude *vnitřek*. O vnitřku platí, že jeho prázdná políčka tvoří *les* (graf, jehož komponenty souvislosti jsou stromy; jinými slovy graf bez cyklů) a že každý strom lesa sousedí s okrajem v právě jednom místě (tomuto místu napojení budeme říkat *portál*). Dokážeme to opět indukcí podle řádu: bludiště řádu 1 má uvnitř jediné prázdné políčko, což je les o jednom stromu a tento strom sousedí s okrajem.

Indukční krok opět využívá toho, že bludiště řádu $r>1$ se skládá z kvadrantů řádu r. Každý z nich je lesem stromů připojených k okraji kvadrantu. Některé z nich jsou tedy připojené i k okraji celého bludiště. Ty zbylé sousedí s některým z tunelů. Tunely tedy mohou spojit více stromů dohromady, ale jelikož tunely samy neobsahují cykly, vznikne propojením stromů opět strom. A jelikož tunely vedou k okraji bludiště, každý nový strom také sousedí s okrajem.

Víme tedy, že celé bludiště bez okraje tvoří les. Navíc všechny jeho stromy jsou připojeny na okraj, takže se dá dostat z libovolného políčka do libovolného. Uvnitř téhož stromu dokonce právě jednou cestou, mezi stromy je možné po okraji projít dvěma způsoby.

Konstrukce bludiště

Abychom si rozmysleli, jak se s rekurzivní strukturou bludiště zachází, pokusíme se nejprve sestavit funkci $f(r,i,j)$, která nám řekne, zda se v bludišti řádu r na souřadnicích (i,j) vyskytuje zeď. První souřadnice bude udávat řádek shora dolů od 0 do 2^{r+1}, druhá podobně sloupec zleva doprava.

Označíme $n=2^{r+1}$ (mez souřadnic v bludišti) a $q=2^r$ (totéž v kvadrantu).

Nejprve se postaráme o okraje: pokud buď i, nebo j je buď 0, nebo n, políčko (i,j) leží na okraji, a tedy je prázdné.

Pokud je libovolná souřadnice rovna q, políčko leží v tunelu. Není-li to některý ze závalů $(q,1)$, $(q,n-1)$, $(q+1,q)$, je políčko opět prázdné.

V ostatních případech se stačí zaměřit na jeden kvadrant, tedy otázku převést na bludiště řádu $r-1$. Jen je potřeba souřadnice správně otočit. V levém horním kvadrantu se ptáme na $f(r-1,j,q-i)$, v pravém horním na $f(r-1,n-j,i)$, v levém dolním na $f(r-1,i-q,j)$ a v pravém dolním na $f(r-1,i-q,j-q)$.

Zbývá dořešit, jak se rekurze zastaví. Rozmysleme si, co se stane pro $r=0$ (to má být bludiště 3×3 s jedním jediným políčkem zdi uprostřed). Na jeho okraje odpovídáme správně, zeď na pozici $(1,1)$ leží na předpokládané poloze tunelu $(q,1)$, takže také hned odpovíme a dále se rekurzivně nevoláme.

Jelikož rekurze má hloubku r a na jedné úrovni trávíme konstantní čas, celý výpočet funkce f trvá $\mathcal{O}(r)$. Celé bludiště bychom pak zkonstruovali v čase $\mathcal{O}(2^r\cdot 2^r\cdot r)=\mathcal{O}(4^r\cdot r)$. (Dodejme, že to jde i v čase $\mathcal{O}(4^r)$, tedy lineárně v počtu políček. Zkuste přijít na to, jak.)

Program (Python):
`http://ksp.mff.cuni.cz/viz/30-1-4-gen.py`

Jakmile umíme bludiště sestrojit, můžeme nejkratší cestu hledat prostým průchodem do šířky. To by fungovalo v lineárním čase s velikostí bludiště a dalo by se za to získat 10 bodů. U větších bludišť nám ovšem dojde paměť (samotné bludiště bychom si sice nemuseli pamatovat a místo toho políčka konstruovat, kdykoliv se na ně dostaneme, ale beztak musíme evidovat, kde už jsme byli, abychom se nezacyklili). U ještě větších bludišť nám dojde i čas.

Cesta na okraj

Pokusíme se najít rychlejší algoritmus, který nepotřebuje procházet všechna políčka. Budeme bludiště rekurzivně rozebírat na kvadranty a sledovat, jak se cesta proplétá mezi kvadranty. Situaci usnadní, že vnitřek bludiště je les, takže kromě průchodu okrajem je cesta určená jednoznačně.

Nejdříve vyřešíme jednodušší případ: výpočet cesty ze zadaného políčka kamkoliv na okraj. Jako výsledek budeme vracet nejen vzdálenost, ale také souřadnice políčka na okraji, kam jsme se dostali.

Mějme bludiště řádu r a políčko (i,j), z něhož se chceme dostat na okraj. Opět označíme $n=2^{r+1}$ a $q=2^r$.

Pokud i nebo j je 0 nebo n, na okraji již jsme, takže hned vrátíme 0 a (i,j). Pokud $i=q$ nebo $j=q$, jsme v tunelu, takže stačí dojít na okraj. Podle toho, který tunel to je, najdeme správný portál (buď $(0,q)$, nebo (n,q)). Do portálu jdeme pravoúhle, takže stačí spočítat pravoúhlou neboli *manhattanskou vzdálenost* mezi políčkem (i,j) a portálem. Manhattanská vzdálenost bodů (x,y) a (x',y') je definována jako $|x-x'|+|y-y'|$.

V ostatních případech leží políčko uvnitř nějakého kvadrantu. Přepočítáme tedy polohu políčka na souřadnice uvnitř kvadrantu (po patřičném posunutí a otočení), zavoláme se rekurzivně na kvadrant a pak přepočítáme souřadnice cílového políčka zpět do celého bludiště (opačné otočení a posunutí). Cílové políčko přitom leží buď na okraji celého bludiště, nebo v tunelu. Odtamtud už na okraj dokážeme dojít, takže stačí sečíst vzdálenost uvnitř kvadrantu se vzdáleností tunelem.

Rekurze má hloubku nejvýše r, na každé úrovni strávíme konstantní čas. Celkem tedy $\mathcal{O}(r)$.

Obecná cesta

Nyní algoritmus rozšíříme, aby uměl najít cestu mezi libovolnými dvěma políčky.

Máme-li propojit políčka (i_1,j_1) a (i_2,j_2), nejprve se podíváme, v jakých leží kvadrantech. Leží-li v tomtéž kvadrantu, chtěli bychom se na tento kvadrant zavolat rekurzivně. Ovšem pozor: může se stát, že každé políčko leží v jiném stromu, takže je potřeba propojit stromy cestou ležící mimo

kvadrant. Podobně kdyby políčka ležela v různých kvadrantech, museli bychom se nejprve dostat na okraj kvadrantů a pak body na okrajích propojit.

Budeme proto řešit obecnější úlohu: pro zadané dvě políčka chceme buďto najít cestu vnitřkem bludiště (to jde, pokud jsou políčka v tomtéž stromu), anebo najít cestu z každého políčka do nějakého portálu na okraji bludiště. V prvním případě je výsledkem vzdálenost, v druhém souřadnice obou portálů a vzdálenost mezi políčky a portály.

Nyní již rekurze funguje. Pokud jsou (i_1, j_1) a (i_2, j_2) v tomtéž kvadrantu, zavoláme se na tento kvadrant. Rozlišíme dva případy:

- Pokud rekurze nalezla propojení vnitřkem kvadrantu, nezbyla na nás žádná práce a výsledek pouze předáme dál.
- Pokud rekurze našla propojení do nějakých portálů P_1 a P_2, pokusíme se tyto portály propojit vnitřkem. Nechť Q_i je políčko na okraji nejbližší k P_i (je-li P_i v tunelu, pak je to portál na ústí tohoto tunelu; je-li P_i na okraji, pak $Q_i = P_i$).
  - Pokud $Q_1 = Q_2$, leží P_1 i P_2 v tunelech a nejsou odděleny závaly. Propojíme je tedy tunelem a ke vzdálenosti připočítáme manhattanskou vzdálenost políček P_1 a P_2.
  - V opačném případě propojení vnitřkem neexistuje (buď některé z P_i neleží v tunelu, nebo nám braní zával). Tehdy jako výsledek vrátíme dvojici portálů Q_1, Q_2 na okraji našeho bludiště a k celkové vzdálenosti připočítáme mahnattanskou vzdálenost Q_1 od P_1 a Q_2 od P_2.

Pokud naopak (i_1, j_1) a (i_2, j_2) leží v různých kvadrantech, spustíme na každý z nich předchozí algoritmus, čímž jsme se přesunuli do portálů na okrajích kvadrantů a ty pak propojíme výše popsaným způsobem.

Libovolný problém řádu r tedy buď převedeme v konstantním čase na problém řádu $r - 1$, nebo ho předeveme na dva problémy „bod–okraj“ řádu $r - 1$. Celá rekurze tedy doběhne v čase $\mathcal{O}(r)$.

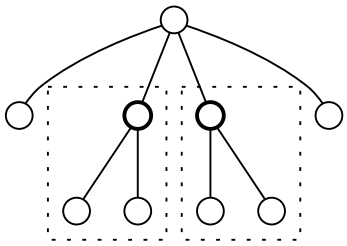
Ale pozor, ještě nejsme hotovi: z rekurze nám může místo cesty vnitřkem vypadnout dvojice portálů na okraji celého bludiště, které ještě musíme okrajem propojit. To je potřeba udělat speciálně, protože okraj není strom. Můžeme si ale všimnout, že leží-li portály na protilehlých stranách bludiště, máme dvě možnosti, jak je propojit, takže si stačí vybrat tu kratší z nich. A pokud portály leží na téže straně bludiště, případně na dvou sousedních, stačí vždy započítat jejich manhattanskou vzdálenost, protože druhá z možných cest je nutně delší.

Program (Python):
`http://ksp.mff.cuni.cz/viz/30-1-4.py`

Martin „Medvěd“ Mareš

30-1-5 Zavirování sítě

Nejdříve se podívejme na jeden pokus o řešení, který bohužel nefunguje. Algoritmy, které hladově infikují podle stupňů, ať už s následným odsimulováním, kam se virus rozšíří, nebo bez něj, bohužel nefungují. Podívejme se na protipříklad:



V obou tečkovaných obdélnících musí být alespoň jeden vrchol nakažen, jinak se nedají „tučné“ vrcholy nakazit. K nákaze celého grafu dokonce tyto dva vrcholy stačí. Hladové řešení by ale vybralo kořen, protože má největší stupeň, a pak by ještě muselo nakazit alespoň další dva vrcholy, tedy nejde o nejmenší řešení.

Většina odevzdaných řešení začínala jednoduchým pozorováním, totiž, že nemá smysl nakazit list. Místo něj můžeme nakazit jeho souseda a tím se ihned nakazí i on. Pojďme tuto úvahu trochu zobecnit.

Pro jednodušší přemýšlení si strom zakořeníme v libovolném vrcholu. Můžeme jít od listů a pro každý vrchol rozhodnout, zda jej na začátku nakazíme, nebo ne, jen podle již rozhodnutých vrcholů pod ním. Budeme potřebovat, abychom pro každý vrchol provedli rozhodnutí až poté, co jsou rozhodnuti všichni potomci. Někteří z vás to řešili pomocí rozdělení stromu na vrstvy, ale jednodušší je využít prohledávání do hloubky (DFS). Poté, co se vrátíme z rekurze posledního potomka, už máme všechno pod sebou rozhodnuté a můžeme se také rozhodnout.

Z rekurze budeme vracet, zda je vrchol nakažen, ať už od svých potomků, nebo ho bylo nutné nakazit na začátku. V druhém případě ho přidáme do výstupu, ale vracet budeme v obou případech stejnou hodnotu. Jak se tedy rozhodnout? Spočítáme si počet nakažených a nenakažených potomků a počet sousedů, do kterého nesmíme zapomenout započítat rodiče. Jen si musíme dát pozor na případ, kdy jsme v kořeni, který ho nemá. Podle nich rozhodneme, zda se vrchol nakazí od potomků, nebo zda se nakazí, když nakazíme i rodiče, nebo zda ho musíme nakazit na začátku.

- Pokud je počet nakažených potomků alespoň polovina počtu sousedů, tak se nakazil od potomků a vrátíme „nakažen“.
- Pokud je počet nakažených potomků o jedna méně než polovina počtu sousedů a vrchol není kořenem, tak se nakazí od otce, vrátíme „nenakažen“.
- Pokud je počet nakažených potomků ještě menší, tak tento vrchol je potřeba nakazit na začátku, přidáme do seznamu nakažených vrcholů a vrátíme „nakažen“.

Všimněme si, že nepotřebujeme žádnou speciální podmínku pro listy, ty přirozeně spadnou do druhé varianty. To odpovídá našemu pozorování, že se vyplatí je nechat nakazit od otce.

Tento postup bude fungovat, jelikož je po celou dobu běhu algoritmu jasně dané, kterou hodnotu musíme vrátit, a pokud můžeme ušetřit počáteční infikování, tak ho ušetříme.

Časová složitost je stejná jako pro DFS, vše stihneme provést v čase $\mathcal{O}(N)$, paměti budeme také potřebovat $\mathcal{O}(N)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-1-5.py`

Jirka Sejkora

30-1-6 Karetní hlavolam

V této úloze jste měli za úkol pracovat s operací XOR. Ukázalo se, že tato neobvyklá operace nejednomu z vás zamotala hlavu a poslala vás do slepých uliček.

Pojďme se tedy odrazit od takřka univerzálního řešení na všechny úlohy – vyzkoušíme všechny možnosti. Kolik těch možností je? Každé číslo můžeme spárovat postupně se všemi ostatními, to nám dává $\mathcal{O}(N^2)$ možností. Jelikož procesor stejně pracuje v binární soustavě, XOR pro něj není problém a zvládne jej stejně rychle jako sčítání (pokud jste se s ním ještě nesetkali, tak věřte, že ve většině standardních programovacích jazyků je zapisován pomocí střížky „^“). Takže $\mathcal{O}(N^2)$ je i časová složitost celého algoritmu.

Výborně! Tak jsme si pořídili řešení, které nemá vůbec špatnou časovou složitost, a skoro nic nás to nestálo. Pojďme ho zkusit trošku vylepšit. Co kdybychom znali hned nejlepší číslo na spárování? Dobře, to bychom chtěli asi trošku moc, ale pojďme se podívat, jak bude vypadat pro nějaké číslo jeho *ideální partner*, který dá nejlepší výsledek.

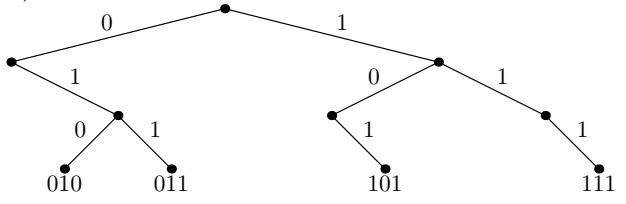
Vzhledem k operaci, se kterou pracujeme, bude výhodné se na čísla koukat ve dvojkovém zápisu. Stejně tak se bude hodit, když všechna čísla budou „stejně dlouhá“, doplníme si tedy všechna čísla zleva nulami tak, aby měla stejně dlouhý dvojkový zápis.

Když tedy hledáme ideálního partnera, chceme, aby na všech místech měl opačnou cifru, tedy tam, kde má původní číslo jedničku, chceme nulu a naopak. Toto nám dá výsledek složený ze samých jedniček, což je vzhledem na naše omezení na délku největší možné číslo.

Obvykle se nám ale takto dobrého partnera najít nepodaří. V tom případě si všimněte, že nejdůležitější jsou pro nás cifry více vlevo. Když hledáme partnera pro číslo začínající jedničkou, partner musí začínat nulou (pokud nějaký takový existuje). Jakékoliv číslo, které začíná jedničkou, by totiž dalo výsledek začínající nulou, zatímco při spárování s číslem začínajícím nulou bychom dostali výsledek začínající jedničkou.

Jak tedy najdeme nejlepšího partnera pro dané číslo z těch nabízených? Jistě chceme číslo, které se liší v první cifře. Pokud je takových více, zaměříme se na ty, které se liší ve druhé cifře, a tak dále. Může se nám ale také stát, že takto v nějakém kroku vyloučíme všechna čísla. V takovém případě se nedá nic dělat a budeme muset vzít takové číslo, že výsledek bude mít na daném místě nulu. Každopádně až nám v tomto eliminačním procesu zbude jediné číslo, je to jistě náš hledaný partner.

Pomohli jsme si ale vůbec? Takto se zdá, že budeme muset pro každé číslo v nejhorším případě opakovaně procházet všechna ostatní. Klíčem bude si čísla šikovně uložit. Chceme se rychle dozvědět, jestli máme nějaké číslo, které začíná jedničkou (nebo nulou). Poté opět jestli ve zbylé skupině čísel je nějaké, které má na druhé pozici jedničku (nebo nulu), atd.



To nás nabádá si čísla ukládat v (neúplném) binárním stromu, jak vidíte na obrázku. Kořen bude reprezentovat všechna binární čísla. Vrcholy těsně pod kořenem reprezentují samostatně čísla, která začínají nulou a ty, která začínají jedničkou. Takto pokračujeme dále až listy budou reprezentovat přímo celá čísla. Může se nám stát, že nějaké skupiny budou celé chybět, v tom případě bude chybět i příslušný vrchol v daném stromu.

V takovémto stromu se už jednoduše najde nejlepší partner. Čteme původní číslo po cifrách a vždy, pokud je to možné, se vydáme dolů opačnou hranou, než je příslušná cifra. Pokud to možné není, jdeme dolů zbylou hranou (v tom případě bude ve výsledku na dané pozici nula).

Jednoduché je i tento strom postavit. Začneme se samotným kořenem a postupně přidáváme všechna čísla. Každé číslo čteme po cifrách. Pokud ještě ve stromu hrana odpovídající dané cifře neexistuje, tak ji vytvoříme a přesuneme se po ní dolů. Jestli ve stromu už je, tak ji vytvářet nemusíme, jen se po ní vydáme.

Pro každé číslo nám tedy stačí strom dvakrát projít od kořene až k listu (jednou při vytváření a jednou při hledání partnera). Všimněte si, že strom je tak vysoký, jak je dlouhé největší číslo ve dvojkovém zápise. Tedy jinými slovy jako jeho dvojkový logaritmus.

Na začátku jsme předpokládali, že čísla na vstupu jsou dostatečně malá, takže i jejich ciferný zápis je krátký, a tedy hloubka stromu je jen konstantní. Mohli bychom tuto konstantu při počítání složitosti jednoduše zanedbat. Koneckonců na začátku jsme řekli, že XOR zvládneme v konstantním čase, což je pravda jen pro dostatečně malá čísla, typicky 2^{64}, nebo přesněji libovolná konstantou omezená čísla. Buďme ale v tomto poctivější a velikost největšího čísla zahrňme do počítané časové složitosti (vyjádříme tedy časovou složitost našeho algoritmu pro libovolně velká čísla na vstupu). Navíc si všimněte, že XOR zvládneme rovnou počítat už při hledání partnera, takže i tak nám stačí počítat XOR jednobitových čísel.

Jelikož čísel máme N, dostaneme se na celkovou časovou složitost $\mathcal{O}(N \log k)$, kde k je největší číslo. Paměťová bude stejná, jelikož si potřebujeme pamatovat celý strom.

Poznámka na závěr. Při programování se vám může hodit operace bitový posun, která se typicky zapisuje jako „<<“ resp. „>>“. Tedy například 5 << 2 znamená vezmi pětku a (v binárním zápisu) ji posuň o dvě místa doleva (a zprava doplň nulami). Pokud vám to trošku zamotalo hlavu a zaráží vás, proč zvládneme XOR v konstantním čase jen pro dostatečně malá čísla, tak trošku světla pomůže vnést náš letošní seriál!

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-1-6.py`

Dominik Smrž

30-1-7 Assembler

Předně nám dovolte omluvit se za počet chybek, které se nám do seriálu vloudily. Bohužel se zdá, že džungle assemblerů je občas opravdu divoká a i my jsme v ní nejednou zabloudili. Naštěstí jste se nenechali zmást například chybějící variantou `MUL` s číselnou konstantou a dorazilo nám spoustu pěkných řešení.

Úkol 1: Povrch kvádru

Chceme do assembleru přepsat formuli $S = 2 \cdot (ab + bc + ca)$. Nejsnazší je jednotlivé matematické operace přímo přepsat do instrukcí assembleru:

```
        MUL r0, r1, r2 @ r0 = ab
        MUL r1, r3     @ r1 = ac
        MUL r2, r3     @ r2 = bc
        ADD r0, r1     @ r0 = ab + ac
        ADD r0, r2     @ r0 = ab + ac + bc
        MOV r4, #2
        MUL r0, r4     @ r0 = 2 * (ab + ac + ca)
```

Existuje však i lepší řešení – násobení je obecně pro procesor docela drahá operace, a tak je lepší se jí vyhnout, pokud to umíme. Pro případ násobení dvěma by šlo použít bitový posun doleva, který jsme si neukazovali, ale stejně tak lze využít jednoduchého faktu, že $2 \cdot a = a + a$:

```
        MUL r0, r1, r2 @ r0 = ab
        MUL r1, r3     @ r1 = ac
        MUL r2, r3     @ r2 = bc
        ADD r0, r1     @ r0 = ab + ac
        ADD r0, r2     @ r0 = ab + ac + bc
        ADD r0, r0     @ r0 = 2 * (ab + ac + ca)
```

Úkol 2: Podmínky

Chceme, aby existoval společný začátek, potom nějaké rozvětvení na if/else a následně aby se tyto větve zase spojily v jednu. Pokud pouze triviálně zapíšeme tuto myšlenku do assembleru, získáme něco takového (násobení dvěma opět píšeme jako sečtení samo se sebou):

```
        CMP r1, #0
        BEQ if
        BNE else
if:
        ADD r0, #1
        B both
else:
        SUB r0, #1
        B both
both:
        ADD r2, r0, r0
```

Jenže opravdu na takovýto jednoduchý if potřebujeme až čtyři různé instrukce skoku? Snadno nahlédneme, že skok `B both` v bloku `else` je naprosto k ničemu – „skočí“ na následující instrukci, tedy tam, kam se stejně procesor chystá pokračovat. Pokud bychom navíc řádky `BEQ if` a `BNE else` prohodili, všimneme si, že můžeme ušetřit ještě jeden skok. Ideální řešení tedy vypadalo zhruba takto:

```
        CMP r1, #0
        BNE else

        ADD r0, #1
        B endif
else:
        SUB r0, #1
endif:
        ADD r2, r0, r0
```

Úkol 3: Největší společný dělitel

Správným algoritmem pro řešení tohoto problému je samozřejmě Euklidův algoritmus. Více o něm se můžete dočíst v naší kuchařce.[8] V tomto případě nám ani nešlo tolik o zvolenou variantu tohoto algoritmu, ale o to, popasovat se s absencí instrukce modulo, tedy instrukce, která spočítá zbytek po dělení.

Jedno řešení tohoto problému je odčítat dělitele, dokud nedojdeme k zápornému číslu, a pak ho jednou zpět přičíst. Například spočítání $r0 = r0 \bmod r1$ může vypadat takto:

```
modulo:
        SUBS r0, r1
        BPL modulo

        ADD r0, r1
```

Druhá možnost využívá celočíselného dělení. Platí, že $x = a \cdot y + b$. Tomu říkáme, že „x děleno y je a, zbytek b“. ARM nám však umí spočítat a pomocí celočíselného dělení! Když známe a, můžeme ho vynásobit y a po odečtení od x získáme b, náš hledaný zbytek. V řeči assembleru:

```
        SDIV r2, r0, r1  @ r2 = a
        MUL r2, r1       @ r2 = a * y
        SUB r0, r2
```

My se rozhodneme pro odčítací verzi modula. Ač má tento postup asymptoticky horší složitost, pro mnoho procesorů je instrukce pro dělení stále příliš velký luxus, a tohle řešení je tedy univerzálnější. Celý algoritmus by mohl v pseudokódu vypadat například takto:

```
while b <> 0:
        tmp := b
        b := a
        b := a mod tmp
        a := tmp
```

V assembleru potom například takto:

```
        MOV r1, #84
        MOV r2, #72
while:
        MOV r0, r2
        MOV r2, r1
modulo:
        SUBS r2, r0
        BPL modulo
        ADD r2, r0

        MOV r1, r0

        CMP r2, #0
        BGT while
```

Honza Gocník

[8] `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

Výsledková listina první série třicátého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*1-1*	*1-2*	*1-3*	*1-4*	*1-5*	*1-6*	*1-7*	*série*	*celkem*
0.					7	10	12	15	10	9	15	62,0	62,0
1.	Josef Minařík	GJarošeBO	3	1		5,5	11	15	10	6,5	14	59,9	59,9
2.	Martin Kurečka	GJarošeBO	4	3		8,5		15	10	9	10	56,5	56,5
3.	Veronika Nechaieva	KyivLyceum	4	1	6	5	11	10	1	6,5	15	56,4	56,4
4.	Jan Kaifer	GKepleraPH	2	7	5	7	1	15	7	9	15	55,0	55,0
5.–6.	Michal Kodad	SPŠSmíchov	2	9	5,5	7,5		15	10		13	52,3	52,3
	Jiří Škrobánek	G Wicht	4	1	2	6,5	6	4	9,5	9	15	52,3	52,3
7.	Matěj Kripner	GEBenešeKL	3	1	6	10		15			15	46,8	46,8
8.	Klára Tauchmanová	GOhradníPH	4	2	3	1,5	2	4	10	5,5	14	45,2	45,2
9.	Jan Černý	BiGy Žďár	2	1	1	1,5	2	15	1	6	8	43,7	43,7
10.	Ondřej Bleha	GBNěmcovHK	3	1		6		10		3,5	15	43,3	43,3
11.	Filip Geib	G MMH LM	4	6		1,5		15		8,5	15	41,3	41,3
12.	Daniel Skýpala	GTomkovaOL	0	4	3,5	1,5		10			15	35,7	35,7
13.	Petr Zahradník	GaSOŠ ÚL	3	1	5	2,5				9	12	34,9	34,9
14.	Jakub Komárek	GUHradiště	3	1	3,5			2		9	15	34,5	34,5
15.	Martin Zimen	GJMasarJI	3	1		2		2	8		15	33,9	33,9
16.	Michal Zaslavský	GKepleraPH	3	1				12			15	29,2	29,2
17.	Václav Pavlíček	SPSEPard	2	8	3	1,5	5		1	1,5	15	29,1	29,1
18.	Jáchym Mierva	BiGy Žďár	1	1	1	2		15		3,5		28,2	28,2
19.	Vladimír Chudý	G Chrudim	1	1		3,5	1		1	1	11	28,1	28,1
20.	Tomáš Strnad	GŽamberk	4	1				10			5	23,0	23,0
21.	Jiří Löffelmann	GLitoměřPH	4	8	0,5					8,5	11	21,5	21,5
22.	Lucia Krajčoviechová	GJHroncaBA	2	1	5			10				19,8	19,8
23.	Miroslav Hrabal	GTomkovaOL	4	6	6						10	18,5	18,5
24.	Adam Dejl	G JGJ PH	4	1	7						5	16,6	16,6
25.	Vít Skalický	GPísnickáPH	0	2	0,8			10				15,1	15,1
26.–30.	Jiří Kvapil	GTomkovaOL	0	1							15	15,0	15,0
	Vojtěch Michal	GNVPláníPH	3	1							15	15,0	15,0
	Jakub Pelc	G UherBrod	4	11							15	15,0	15,0
	Zuzana Urbanová	GFXŠaldyLI	4	2							15	15,0	15,0
	Ondřej Wrzecionko	GTěš	3	1	0,5	1	0	2	0	3,5	0	15,0	15,0
31.	Filip Masár	PiarGNitra	4	3							14	14,7	14,7
32.	Eliška Vlčinská	GHladnov	3	3	0,5						11	14,6	14,6
33.–34.	František Kmječ	G Brandýs	2	6	0,5				10	1,5		13,4	13,4
	Jakub Růžička	GNymburk	3	1				10				13,4	13,4
35.	Jakub Jirkal	GJungmanLT	3	2				10				13,2	13,2
36.	Michal Tomek	GHumpolec	4	1				8				12,3	12,3
37.	Tomáš Sládek	GJHroncaBA	2	1	2,5					3,5		10,9	10,9
38.	Karel Balej	GRokycany	3	3						9		9,0	9,0
39.	Václav Zvoníček	GJarošeBO	2	1		4,5						7,5	7,5
40.	Viktor Fukala	GKepleraPH	1	4	6,5							6,8	6,8
41.	Ondřej Gonzor	G Brandýs	1	5	0	2,5			1			6,1	6,1
42.	Lenka Vincenová	GTomkovaOL	4	1	2							4,0	4,0
43.	Lukáš Caha	GZborovPH	4	5	2,5							3,8	3,8
44.	Jakub Ucháč	ŠMaVVzt	2	1	0,5	0,5	0					2,7	2,7
45.–46.	Vojtěch Březina	GCoubTábor	1	1	0,5							1,3	1,3
	Vojtěch Káně	G Brandýs	2	1	0,5							1,3	1,3