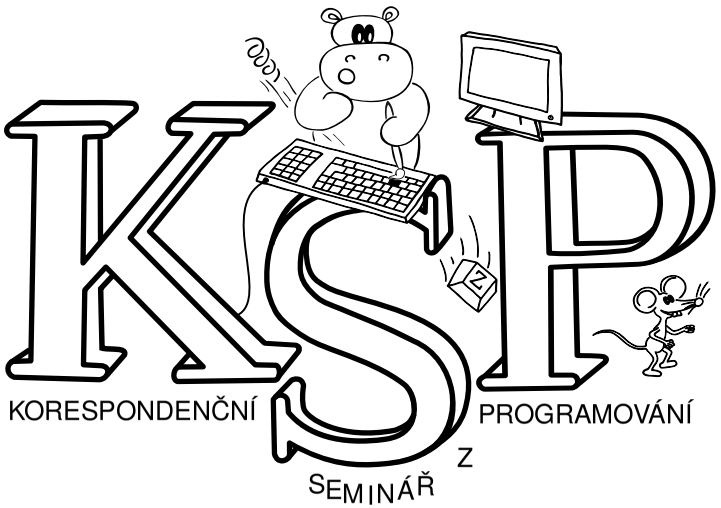
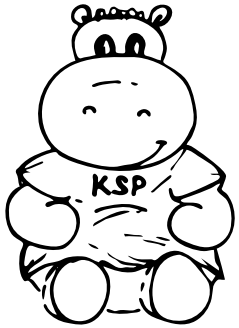
Dokud existují počítače, bude existovat i KSP!





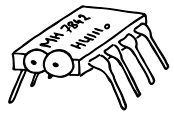
Toužíš po nových vědomostech?

Chceš poznávat nové lidi?

Zajímáš se o počítače?

Láká Tě trocha soutěžení?

Hledáš výzvu pro svou hlavu?



Byla Tvá odpověď alespoň jednou „ano“?
Pak hledáme právě Tebe. Do KSP
se může zapojit každý, tedy i Ty. Otoč list!

Co je KSP?

KSP je Korespondenční seminář z programování. Jak takový seminář funguje? Několikrát za rok vydáváme série obsahující různé úlohy a posíláme je řešitelům.

Ti mají několik týdnů na vymyšlení a odevzdání řešení. My je pak opravíme, okomentované a obodované pošleme zpět a zveřejníme autorská řešení. KSP má dvě kategorie: Z pro začínající řešitele a hlavní kategorii H pro ty zkušenější, kde číhají záludnější úlohy.

Kdo seminář organizuje?

Organizátoři jsou studenti Matematicko-fyzikální fakulty Univerzity Karlovy v Praze (MFF UK), většinou bývalí řešitelé.

Co najdu v zadání?

Můžeš řešit teoretické a praktické úložky. Vždy je důležité vymyslet postup (návod) jak nalézt řešení, například jak může počítač rychle najít nejkratší cestu z Kocourkova do Prčic.

Součástí zadání jsou i studijní texty, jejichž přečtení Ti dá nástroje k řešení úloh. Kuchařky jsou krátké texty o různých tématech. Seriál pro změnu probere v průběhu roku jedno téma do hloubky.

Jak úlohy vypadají?

V teoretických úlohách je třeba postup slovně popsat a odevzdat nám ho, my jej pak opravíme a okomentujeme.

V praktických úlohách nejde o popis, ale o výsledek. U úloh (jsou open-data) si stáhneš vstupní data, která zpracuješ Tebou zvoleným způsobem, nejlépe programem v libovolném programovacím jazyce. Výstup odevzdáš a ihned vidíš, zda je výsledek správný.

Vymýšlení mi nejde, co s tím?

V KSP-Z je také možné odevzdat praktické úlohy po termínu – ještě týden po zveřejnění slovních popisů řešení lze odevzdávat úlohy za třetinu bodů. Teprve poté se objeví i zdrojové kódy.

Proč mám KSP řešit?

Během řešení KSP se naučíš programovat. To se Ti může v životě hodit, obzvlášť pokud se chceš stát programátorem. ;)

Díky KSP můžeš poznat informatiku v celé její kráse – mocné programy, magické datové struktury... prostě to, co se ve škole nedozvíš.

To může být užitečné nejen při řešení matematické olympiády kategorie P. Navíc nejlepší řešitele zveme na soustředění, kde můžeš poznat nové kamarády.

Také pokud se staneš úspěšným řešitelem hlavní kategorie, vezmou Tě na Matfyz bez přijímaček.

Hmmm... soustředění?

Jsou dvě, jarní především pro řešitele KSP-Z a podzimní pro řešitele hlavního KSP. Obě jsou týdenní akcí plnou přednášek a zážitků, kterou určitě stojí za to zažít.

Dostanu i něco hmotnějšího?

Ano, pokud se dostaneš mezi ty nejlepší. Tři nejúspěšnější řešitelé si budou moci vybrat jako odměnu knížku nebo například tričko, hrneček, hrocha.

Vůbec nevím, jak začít...

Inu, žádný učený z nebe nespadl, chce to studovat a zkoušet. ;) Dobrým odrazovým můstkem může být naše Encyklopedie, jejíž součástí jsou i kuchařky včetně úplných základů.

S výběrem Ti může pomoci i bodování – lehčí úlohy bývají většinou za méně bodů.

Napadá mě jen špatné řešení

Tak prostě odevzdej i to. :) Špatné, pomalé nebo neúplné řešení je lepší než žádné. Za nedokonalá řešení u teoretických úloh hlavy rozhodně netrháme. Naopak, pokusíme se Ti poradit, co zlepšit. U praktických úloh zase bývá několik vstupů malých, takže za ně získáš část bodů i s jednodušším řešením.

Co když mi něco není jasné?

Klidně se nás ptej. Na dotazy k úlohám se nejvíc hodí naše diskusní fórum. Podrobnosti o fungování semináře nalezneš na webu. A budeš-li mít stále nějakou otázku, čteme mail a jsme na Facebooku.

Zadání

KSP-Z: *http://ksp.mff.cuni.cz/z/*

KSP-H: *http://ksp.mff.cuni.cz/*

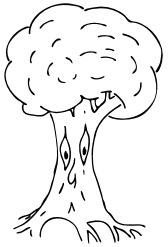
Studijní texty

http://ksp.mff.cuni.cz/encyklopedie/

Korespondenční Seminář z Programování

30. ročník **KSP** **Říjen 2017**

Milí řešitelé, řešitelky a řešitelčata!



Kulatý třicátý ročník hlavní kategorie KSP právě začíná a do ruky se vám dostal první leták. Letos bude každá série obsahovat 7 úloh, z nichž jedna bude praktická opendatová úloha a poslední vždy bude seriál, který se bude v navazujících úlohách táhnout skrz celý ročník. Letos bude seriál o assembleru.

Do celkového bodového hodnocení se z každé série **započítá 5 nejlépe vyřešených úloh.**

Za úspěšné řešení KSP je možno být přijat na MFF UK bez přijímacích zkoušek. Úspěšným řešitelem se stává ten, kdo získá za celý ročník (této kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě.

Každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, placku a možná i další překvapení.

Pokud budete mít jakoukoliv otázku, neváhejte se zeptat. Kontaktní adresy najdete v patičce na konci letáku. Přejeme hodně štěstí!

Termín série: **30. října 2017 v 8:00** **(pro seriál: 13. listopadu 2017 v 8:00)**

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

Značky úloh: Lehčí úloha (či její část) vhodná pro začátečníky; Praktická open-data úloha; Těžká úloha pro zkušené; Seriálová úloha; Úloha, u které doporučujeme začíst se do kuchařky

Odměna série: Každému, kdo vyřeší **tři libovolné úlohy na plný počet bodů**, pošleme **sladkou odměnu**.

První série třicátého ročníku KSP

To snad není pravda! Já jsem věděl, že není dobrý nápad začínat další ročník KSP takhle těžkou úlohou. Zadání jsem si přečetl před čtyřmi dny, neustále jsem nad ním přemýšlel, ale řešení ne a ne přijít. Já snad upíšu svoji duši ďáblu, aby mi aspoň trochu poradil.

Kupodivu se zdá, že se mi skutečně taková příležitost naskytne. Ráno jsem v e-mailu mezi zásobami spamu objevil podivnou zprávu:

Nápovědy od pana Nápovědy! Vyřešíme i váš algoritmický problém. Ulice Na Větru, jsme tu 24 hodin denně.

Chtěl jsem zjistit, kdo to ten pan Nápověda vlastně je. Zkusil jsem ho hledat podle e-mailu v nějaké veřejné online databázi osob. Nic jsem ale nenašel, akorát jsem odhalil, že celá databáze je pořádně rozbitá.

30-1-1 Oprava databáze — 7 bodů

Máme databázi osob, kde každý záznam obsahuje jen dva údaje: jméno osoby a její e-mail. V databázi jsou ale chyby a existuje mnoho záznamů buď se stejným e-mailem, nebo se stejným jménem.

Správci databáze se konečně rozhodli nepořádek vyřešit a pro každou osobu, která je v databázi, chtějí zjistit, kolik záznamů tam vlastně má. Platí, že kdykoli se dva záznamy shodují ve jméně *nebo* v e-mailu, patří stejné osobě.

Příklad: Záznamy `Pepa <pepa@example.com>`, `Josef <pepa@example.com>` a `Josef <josef@example.com>` všechny patří jedné a té samé osobě.

Nakonec jsem to nevydržel a do ulice Na Větru jsem se opravdu vypravil. Dorazil jsem tam až pozdě večer a na první pohled se nezdálo, že by liduprázdné místo nabízelo cokoliv zajímavého. Udělali si ze mě legraci, pomyslel jsem si zklamaně. Chtěl jsem odejít, ale pak jsem si všiml zvláštní telefonní budky na konci ulice.

Na první pohled vypadala docela obyčejně, ale uvnitř měla zvláštní panel: místo číselníku obsahoval padesát očíslovaných tlačítek a nad nimi byl displej, na kterém teď svítil nápis: „Pro Napovedu vlozte minci“. Tak vida! Vytáhl jsem z peněženky korunu a vhodil ji do automatu. „Nejdriv 3, potom 15“, objevilo se na displeji. Schválně zkusím další korunu. „Nejdriv 22, potom 38“, změní se text. Asi chtějí, abych ta tlačítka stisknul v nějakém pořadí, ale kolik mincí budu muset spotřebovat, abych to pořadí zjistil?

30-1-2 Telefonní hlavolam — 10 bodů

Aby se náš hrdina dostal k panu Nápovědovi, musí ve správném pořadí stisknout N tlačítek, očíslovaných od jedničky do N, přičemž každé tlačítko se stiskne právě jednou a existuje jen jedno správné pořadí. Naštěstí nám protistrana posílá řadu dvojic čísel, kde dvojice (I, J) znamená, že číslo I ve správném pořadí přijde před číslem J (nemusí však být těsně vedle sebe).

Dohromady dostaneme až M dvojic a je zaručeno, že dohromady nám dvojice jednoznačně určují pořadí (protistrana není nikterak zlomyslná). Za každou další napovězenou dvojici si ale musíme zaplatit, takže bychom chtěli najít správné seřazení na co nejmenší počet dvojic (dvojice dostáváme v nějakém náhodném pořadí).

Příklad: Pro $N = 4$ se protistrana rozhodla, že nám postupně napoví dvojice $(2, 1)$, $(3, 4)$, $(3, 2)$, $(4, 1)$, $(2, 4)$ a $(3, 1)$. Správné pořadí je $3, 2, 4, 1$ a je jednoznačně určené po pěti nápovědách – o poslední nápovědu tedy už nemusíme žádat.

Uf, mince mi vystačily jen tak tak. Naťukám správné po-

řadí, displej vítězoslavně zabliká a telefon začne zvonit. Zvědavě zvednu sluchátko.

„Je tam pan Nápověda?“ ptám se. „Výborně, jste docela schopný,“ dostanu odpověď. „Bude jistě lepší, když se uvidíme osobně. Zítra dopoledne, pražská ZOO, pavilon hrochů. Těšte se, poradím vám,“ řekne úsečně a zavěsí. Jsem nadšený, člověk, který si dělal práci s hádankou v telefonní budce, mi určitě dokáže poradit! Ale asi byl bych o něco méně nadšený, kdybych si na konci hovoru všiml jemného zachechtání.

⋆⋆⋆

Do ZOO jsem dorazil brzy ráno, ale v pavilonu hrochů se dlouho nic nedělo. Sledoval jsem přicházející návštěvníky, a hádal, kdo z nich bude muž, se kterým jsem včera mluvil. Pak mně najednou někdo vezme za rameno. „Kdo jste?“ leknu se a podívám se na něj. Je to nějaký vysoký člověk, tváří se udýchaně a nervózně. „Já jsem řešitel KSP“, zakoktám. „Vy jste pan Nápověda?“ „Ještě aby tohle,“ ušklíbne se dlouhán, „Hned pojď se mnou!“ řekne mi. Překvapivě nejdeme k východu, ale k nějakým špinavým dveřím, které určitě nejsou určené pro návštěvníky.

„Rychle ti vysvětlím situaci. Já jsem Jirka, organizátor KSPčka. Můžeš být rád, že ses panu Nápovědovi nedostal do spárů,“ vysvětluje, zatímco probíháme místnostmi obloženými krmením pro hrochy. „Chtěl jsem od něj řešení jedné úlohy,“ přiznám se. „Stejně jako spousta řešitelů před tebou! Nalákal je na pomoc při řešení nějaké úlohy, a teď pro něj píšou programy v jeho bunkru.“ Vyběhneme ven na malý dvorek, uprostřed kterého stojí zaparkovaný červený Volkswagen.

Jirka stiskne ovladač centrálního zamykání. Auto se nejenom odemkne, ale taky se mu bleskurychle automaticky otevřou dveře. „Nejnovější vychytávka,“ ušklíbne se. „Rychle, sedni si dopředu! Snad ještě Nápověda nezjistil, kde jsme.“ Nastoupíme, nastartujeme, dveře se samy zabouchnou a auto vyrazí dopředu, až se mu protočí kola.

„Otevřete nám, okamžitě“ volá Jirka do vysílačky. Odloží si ji na přední panel, pokrytý mnoha dalšími zařízeními, o jejichž funkci nemám tušení. Neskutečně rychle se proplétáme mezi ohradami s exotickou zvěří a najednou se před námi vynoří kovová vrata. Naštěstí je právě otevírá jakýsi zaměstnanec ZOO – jenže není dost rychlý. Proletíme branou a ozve se křupnutí, protože jsme právě přišli o boční zrcátka.

„Měli jsme mu zaplatit víc,“ zamumlá Jirka. „Víš, ve skutečnosti jsi nám dost pomohl. Nápověda totiž udělal chybu a při domlouvání tvého únosu použil mobilní síť. Takže teď víme, kde se jeho bunkr nachází.“ „Vy odposloucháváte mobilní síť?“

„Ne že by to byl takový problém,“ ozve se za mnou. Teprve teď si všimnu kluka v oranžovém tričku, který je usazený na zadním sedadle, na klíně má položený notebook a v uších sluchátka. „Já jsem Filip, taky org. Vypadáš trochu vyděšeně. Teď se musíš sebrat. Nechceš něco ostřejšího?“ ptá se. „Jako myslíš alkohol?“ Zatřese hlavou, podá mi kalíšek a nalije do něj z termosky tmavě hnědou tekutinu. „Dlouho louhovaný Puerh. To má povzbuzující účinky i na mrtvolu.“

30-1-3 Placení v čajovně — 12 bodů

Filip si před zátahem na Nápovědu kupuje Puerh ve svém oblíbeném čajovém obchodě. Protože má v peněžence mnoho drobných, chtěl by se jich zbavit, zvláště těch nejtěžších mincí.

Celková cena čaje je H korun. Existuje N různých mincí o hodnotách $H_1, \ldots, H_N$ a hmotnostech $M_1, \ldots, M_N$. Zároveň víme, kolik mincí každého druhu má Filip v peněžence.

Prodavači musíme dát mince celkové hodnoty alespoň H. Pokud nemáme přesnou částku, tak nám vrátí, ale vždy nejtěžší možnou sadou mincí. Zároveň platí, že můžeme zaplatit K mincemi a prodavač vrátí maximálně L mincí.

Na základě znalosti všech parametrů vyberte mince, kterými má Filip zaplatit, aby měl na konci transakce co nejlehčí peněženku.

„Nabereme Janku a jedeme do bunkru,“ vysvětlí mi Filip, zatímco Jirka kličkuje po pěší zóně a snaží se nesrazit ani chodce, ani tramvaj. „Nápověda je dost schopný, ale teď zůstal v ZOO. Musíme se do bunkru dostat dřív než on a osvobodit všechny uvězněné řešitele.“

Pořád mám trochu vyražený dech. „Tak co vlastně jste? Orgové KSPčka, nebo tajní agenti?“

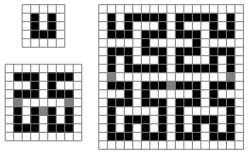
„Obojí,“ řekne a znuděně zazívá. „Potom, co nám začal pan Nápověda dělat problémy, jsme zjistili, že si s ním policie neumí poradit. A řekli jsme si, že být tajným agentem není v zásadě o tolik hektičtější, než třeba organizovat soustředění.“

Odněkud vytáhne další notebook a podá mi ho. „Najdeš tam soubor s mapou bunkru. Napiš mi program, který v něm dokáže najít cestu, ať nám jsi k něčemu užitečný.“

30-1-4 Cesta v bunkru — 15 bodů

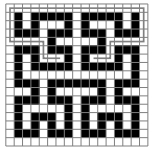
Pan Nápověda se rozhodl vystavět svůj bunkr jako nejdokonalejší bludiště na světě. Jeho zdi mají tvar Hilbertovy křivky, což je jistá křivka procházející všemi body roviny. Aby bludiště dokázal v konečném čase postavit, musel místo opravdové Hilbertovy křivky použít její aproximaci určitého konečného řádu $r \geq 1$.

Mapu bunkru si můžeme nakreslit jako čtvercovou síť velikosti $(2^{r+1}+1) \times (2^{r+1}+1)$ políček, přičemž políčka ležící na Hilbertově křivce tvoří zdi a ostatními políčky je možné procházet. Bludiště řádů 1, 2 a 3 vypadají následovně:



Z obrázku je také vidět, jak se křivka obecně konstruuje: Křivka řádu $r+1$ vznikne ze 4 kopií křivky řádu r, které vhodně natočíme a pospojujeme šedými políčky.

Vaším úkolem bude nalézt v Hilbertově bludišti nejkratší cestu mezi zadanými dvěma políčky. Například mezi políčky $(6, 6)$ a $(6, 10)$ obsahuje nejkratší cesta 41 políček a vypadá následovně:



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

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.

Webové stránky: *https://ksp.mff.cuni.cz/* **E-mail:** *ksp@mff.cuni.cz* **Diskusní fórum:** *https://ksp.mff.cuni.cz/forum/*

Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: `E9:DB:EE:C6:62:BC:14:DE:09:E4:E8:97:DC:36:0E:87:B3:50:B0:01`.

Tento princip přičítání a odečítání se dá zobecnit i pro libovolné vyšší rozměry, ale chce to již trošku představivosti, co se má přičíst a kolikrát. Říká se tomu také *princip inkluze a exkluze* a najde použití nejen u vícerozměrných prefixových součtů.

Vyvážení délky předvýpočtu a hlavního výpočtu

Správně vyvážit, kolik času můžeme věnovat na předvýpočet a kolik času na hlavní výpočet, je velice důležitá věc a spousta i zkušenějších řešitelů v tom občas chybuje. Přitom to při troše počítání není vůbec nic složitého.

Jako první je potřeba vědět, kolikrát nám předvýpočet během běhu programu pomůže. Předvýpočtem si totiž vybudujeme za nějaký čas určitou datovou strukturu, pomocí které pak dokážeme rychle odpovídat na zadané dotazy.

Označme si počet takovýchto dotazů, které program za běhu dostane, jako Q. Buď to může být hodnota přímo ze zadání typu „Zkonstruujte datovou strukturu pro n hodnot, která zvládne rychle odpovídat na dotazy daného typu, a očekávejte řádově m dotazů“, nebo se může jednat o nějaký interní dotaz v rámci běhu programu (příklad interního dotazu je třeba výše uvedený algoritmus na hledání souvislé podposloupnosti s co největším součtem, který se za běhu ptal na součty nějakých úseků).

Dále si označme jako $\mathcal{O}_p$ čas, který nám zabere předvýpočet a jako $\mathcal{O}_q$ čas, který nám ušetří každý předvypočítaný dotaz. Celkový čas, který ušetříme, je pak vlastně $Q \cdot \mathcal{O}_q$. Pokud je tento čas řádově větší než $\mathcal{O}_p$, předvýpočet má smysl.

Naopak nemá smysl trávit předvýpočtem řádově více času, než by trval samotný výpočet bez použití předpočítaných hodnot.

Jako příklad uvažujme problém o velikosti n, u kterého máme tři možnosti, které můžeme zvolit. Můžeme buď předvýpočet úplně vynechat a na každý dotaz odpovídat v čase $\mathcal{O}(n)$, nebo provést předvýpočet v čase $\mathcal{O}(n \log n)$ a poté odpovídat na každý dotaz v čase $\mathcal{O}(\log n)$, nebo provést předvýpočet v čase $\mathcal{O}(n^2)$ a pak odpovídat v čase $\mathcal{O}(1)$ na dotaz.

Kdy se nám co hodí?

- Pokud bychom dostali jen jeden dotaz, nemá smysl si cokoliv předpočítávat a odpovíme jednou v čase $\mathcal{O}(n)$.
- Pokud bude dotazů řádově n, má smysl použít první předvýpočet. Pak budeme mít čas na předvýpočet i na samotný výpočet $\mathcal{O}(n \log n)$, což je optimum.
- Naopak pokud by dotazů bylo řádově n^2 nebo víc, tak se nám již první předvýpočet nevyplatí, dostali bychom se totiž na čas $\mathcal{O}(n^2 \log n)$. Zde se hodí použít druhý, delší předvýpočet a pak se dostat na časovou složitost $\mathcal{O}(n^2 + n^2 \cdot 1) = \mathcal{O}(n^2)$.

Hladové algoritmy

Věřte nebo ne, ale i počítač se někdy cítí hladový. Po namáhavé práci mu můžeme dopřát to potěšení, aby si ukousl co největší kus dat. A ukážeme, že někdy je to i ku prospěchu. Řeč bude o *hladových algoritmech*.

Takovými algoritmy rozumíme ty, které hledají řešení celé úlohy po jednotlivých krocích a splňují následující dvě podmínky:

- V každém kroku zvolí lokálně nejlepší řešení.
- Provedené rozhodnutí již nikdy neodvolává (tedy nebacktrackuje).

Lokálně nejlepší řešení je takové, které v aktuálním kroku vybere tu možnost, která nám na tomto místě nejvíce pomůže (bez jakéhokoliv ohledu na globální stav). Může to být třeba nejvyšší hodnota, nebo nejkratší cesta v grafu.

Pokud ale od hladového algoritmu chceme, aby nám našel globálně nejlepší řešení, musí naše úloha splnit předpoklad, že si výběrem lokálně nejlepšího řešení nezhoršíme to globální. Tento předpoklad se nedá formulovat obecně a je nutné se nad ním zamyslet zvlášť u každé úlohy.

Příklady hladových algoritmů

První hladovou úlohou bude (jak jinak) automat na jídlo vracející mince. Automat by měl vracet peníze nazpět tak, aby vrátil daný obnos v co možná nejmenším počtu mincí. Pro náš měnový systém (máme mince hodnot 1, 2, 5, 10, 20 a 50 Kč) lze tuto úlohu řešit hladovým algoritmem – v každém kroku algoritmu vrátíme tu největší minci, kterou můžeme (tedy pro vrácení 42 Kč to bude $42 = 20 + 20 + 2$ Kč).

Měnové systémy většiny států jsou postavené tak, aby fungovaly takto pěkně, neplatí to ale obecně. Zkusme si vrátit 42 Kč, pokud bychom měli jen mince hodnoty 20, 10 a 4 Kč. Správným řešením je $42 = 20 + 10 + 4 + 4 + 4$ Kč, hladový algoritmus by ale zkusil vrátit $42 = 20 + 20 + \ldots$ a tady by selhal.

Dále se velmi často dají hladovým algoritmem řešit nějaké úlohy přidávání nebo odebírání skupin prvků. Typickým příkladem je třeba rozvržení naplánovaných přednášek do učeben. Seřadíme si začátky přednášek podle času a postupně bereme jednu za druhou a umísťujeme je do volných učeben s nejnižším číslem.

Tím jsme si určitě nic nerozbili, protože v nějaké učebně přednáška být musí. Určitě budeme potřebovat tolik učeben, kolik je maximálně přednášek v jeden čas, a díky tomu si umístěním přednášky do nějaké učebny nezablokujeme místo pro jinou přednášku, jelikož nám vždy zbude dostatek volných učeben.

Kdybychom ale naopak měli pevně zadaný počet učeben a chtěli jsme do nich umístit co možná nejvíce přednášek, nejedná se již o úlohu řešitelnou hladovým algoritmem, v takovém případě je potřeba zvolit nějaký chytřejší postup.

Závěr

Doufám, že jste si z tohoto rozsáhlého textu odnesli nějaké nové znalosti a poznatky, které vám pomohou nejen v řešení KSP.

Pokud jste začínajícími řešiteli, zkuste s pomocí kuchařky vyřešit několik lehčích úloh a jejich řešení poslat – nově nabyté znalosti je totiž nejlepší co nejdříve protrénovat. Nic si nedělejte z toho, pokud napoprvé nevyřešíte všechno, s postupným zkoušením se budou vaše znalosti jen zlepšovat. Zkušenější řešitelé možná v kuchařce nalezli nějaké ujasnění pojmů, či si některé techniky osvěžili.

A pokud tento text považujete za dobrý, budeme jen rádi, pokud ho doporučíte svým kamarádům a spolužákům, kteří chtějí s programováním začít.

Úvodním kurzem vaření podle kuchařky vás provedl

Jirka Setnička

Formát vstupu: Vstup má jediný řádek. Na něm se nachází pět mezerou oddělených čísel: řád Hilbertovy křivky ($1 \leq r \leq 30$), dvojice souřadnic počátečního políčka a dvojice souřadnic cílového políčka. Souřadnice se skládají z čísla řádku (číslováno shora od 0) a čísla sloupce (číslováno zleva od 0).

Formát výstupu: Výstupem je jediné přirozené číslo: počet políček na nejkratší cestě mezi zadanou dvojicí políček.

Příklad: Předchozímu obrázku odpovídá vstup `3 6 6 6 10` a výstup `41`.

Naštěstí se mi napsat program podařilo docela rychle, právě ve chvíli, kdy jsme se blížili ke dvěma vzrostlým věžákům. Zajeli jsme do průjezdu, kde na nás čekal další organizátor. Tedy, vlastně organizátorka.

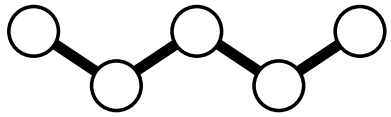
„Vyrušili ste ma pri opravovaní úloh, zasa nestihneme termín,“ řekla nespokojeně, ale stejně se usmála. „Podarilo sa mi pripojiť sa na sieť bunkru.“ „Takhle rychle?“ řekl Jirka nadšeně. „Nie je to komplikovanejšie ako web KSPčka,“ odpověděla, sedla si dozadu k Filipovi a požádala ho o kalíšek Puerhu. „Teraz tam len vypustiť môj obľúbený vírus.“

30-1-5 Zavirování sítě — 10 bodů

Janky oblíbený virus infiltruje počítačovou síť. Předpokládáme, že síť je stromem, kde vrcholy jsou počítače a hrany jsou propojením mezi počítači. Na začátku si vybereme několik počítačů, které nakazíme virem. Následně se virus může šířit a platí, že počítač se nakazí, pokud alespoň polovina počítačů, se kterými je propojený, je nakažena.

Najděte, kolik nejméně počítačů musíte nakazit (popřípadě které), aby se v dalších krocích postupně nakazily všechny počítače.

Příklad: Následující graf se dá vyřešit nakažením jednoho (jakéhokoliv) vrcholu, z něj se nákaza rozšíří na všechny stroje.



Jirka už zase vyrazil na cestu. Opouštěli jsme centrum Prahy a před naší přední kapotou se vynořovaly obrysy nějakého sídliště. „Sem matfyzáci obvykle jezdí na tělocvik, a ne zachraňovat středoškoláky,“ řekl Jirka, když jsme se skřípěním brzd zastavili u budovy s nápisem Sportovní centrum. „Nejraději bych prorazil tu skleněnou stěnu a vjel přímo k bazénu.“ „Nie, teraz musíme byť nenápadní,“ poprosila ho Janka. Vypadala, že ji ta myšlenka vylekala. „Musím zostať tu, ten vírus zatiaľ akosi nefunguje.“

Filip, Jirka a já jsme vešli do budovy a zamířili si to ke vchodu do bazénu. „Té paní u recepce se bojím víc než Nápovědy,“ zašeptal Filip. Ale zdálo se, že Jirka je na celou situaci připravený. Přišel k recepční a aniž by cokoliv řekl, nakreslil na pult nějaký tajemný symbol. Znuděně vypadající žena najednou ožila. Ze skříně vytáhla balíček karet, zamíchala ho a rozložila deset karet na pult. Na každé z nich bylo celé číslo.

„To je jeho další hádanka,“ řekl Filip. Přišel blíže k pultu a začal si karty pozorně prohlížet. „Vy jste se asi necvičili v xorování, že ano?“

30-1-6 Karetní hlavolam — 9 bodů

Nápověda si rád ověřuje bystrost svých studentů podle toho, jak rychle dokáží provádět binární operace s celými čísly. Jednou z těchto operací je XOR, známý též jako „exkluzivní nebo“ a značený $\oplus$.

Pokud chceme spočítat XOR dvou (kladných celých) čísel A a B, nejprve obě převedeme do dvojkové soustavy. Jednotlivým cifrám dvojkového zápisu říkáme *bity* a číslujeme je od nuly zprava: poslední cifra je nultý bit, předposlední je první bit atd.

Výsledek dostaneme opět ve dvojkové soustavě, a to následovně: i-tý bit výsledku je 1 právě tehdy, když jsou i-té bity čísel A a B různé.

Příklad: spočítáme $26 \oplus 6$ (uprostřed jsou čísla přepsaná do binární soustavy):

```
26       = 11010
6        = 00110
26 xor 6 = 11100 = 28
```

Takže $26 \oplus 6 = 28$

Recepční na stůl vyloží N karet, na každé kartě je kladné celé číslo. Úkolem je najít dvojici karet s takovými čísly A a B, že hodnota $A \oplus B$ je největší možná.

Zřejmě se Filipovi podařilo hádanku vyřešit správně, protože žena se usmála a položila na stůl klíč s visačkou s číslem. „Běžte do šatny,“ řekla. Tam odemkneme příslušnou skříňku a já vydechnu překvapením. Netušil jsem, že do obyčejné šatní skříňky se dá schovat výtah pro přepravu osob! Tedy, ve skutečnosti jen jedné osoby, víc se tam nevejde. Nápovědův bunkr je schovaný někde pod zemí a tohle je vchod.

⋆⋆⋆

„Ani se nehýbejte,“ ozvalo se ostře za námi. To snad není pravda. Mohli jsme být v bunkru snad pět minut a jsme chyceni! Pan Nápověda, prošedivělý stařec, byl za našimi zády. Ohlédli jsme se a strnuli, protože držel v ruce revolver a mířil na nás.

„Nic se nestane, nebudu vás zabíjet,“ ušklíbne se, „potřebuji další mozky pro svůj tým programátorů.“ „Jak jste se sem dostal tak rychle?“ ptá se Jirka a zdá se být skoro naštvaný z toho, že se sem ze ZOO dokázal dopravit rychleji někdo jiný než on. „Jednoduše! Zachytil jsem vaše auto, už když jste tam přijížděli.“ Jsme v háji, už tu zůstanem navždycky, proběhne mi hlavou.

Než stihneme cokoliv udělat, vyjede mezi námi a Nápovědou skleněná stěna. Nápověda se ale kupodivu zatváří překvapeně. „To jsem nebyl já,“ zamumlá. A najednou se ozve dunivý výbuch a chodbu za Nápovědou začne rychle zaplavovat voda. „Rýchlo zmiznite!“ ozve se z interkomu, nebo něčeho podobného. „Dvere vľavo, vpravo a hore po schodoch,“ upřesní hlas. Na nic nečekáme: chodba za skleněnou stěnou je téměř po strop zaplavená a samotná stěna to asi také dlouho nevydrží. Běžíme pryč podle instrukcí a najednou jsme zpátky na denním světle. Vyšli jsme u bazénu – nebo přesněji u toho, co z bazénu zbylo. Byl prázdný, všechna voda odtekla do bunkru přes díru v jeho prostředku.

„Chalani, vy si to tak komplikujete,“ přijde k nám Janka. Za ní jsou osvobození řešitelé, kteří teď překvapeně mžourají do denního světla. „Ak by ste boli opatrnejší, nemusela som robiť takú katastrofu.“ „Ale pan Nápověda je poražený. . . “ řekne Filip. Najednou mu ale pípne mobil a na displeji se objeví zpráva:

Porazili jste me imperium, ale ja se nevzdam. Uvidime se v Tokiu!

A takhle to v týmu organizátorů chodí. Občas musíte opravit úlohu, občas nestíháte a občas musíte jet zneškodnit schopného padoucha až do Japonska. Ale pokud jsi jenom účastník a nevyužiješ nějaké pochybné známosti, aby vyřešila úlohu za tebe, nemáš se čeho bát. Ačkoliv, i řešit KSP je taky někdy pořádná jízda!

Kuba Maroušek

30-1-7 Assembler — 15 bodů

V letošním seriálu se ponoříme hlouběji do nitra nejen našich počítačů a notebooků, ale i chytrých mobilních telefonů, a nahlédneme pod křemíkovou pokličku procesorů. Naučíme vás, jak vlastně vypadají příkazy, které procesor umí provádět, zkusíme si něco pomocí nich naprogramovat a povíme si i něco o paralelním počítání. Teď ovšem nepředbíhejme a začněme hezky od začátku…

0xE3A00001 nebo mov r0, 1

Procesor si můžeme představit jako takovou malou krabičku, která krok za krokem čte instrukce a každou z nich vykoná. Všechny tyto instrukce musí být nějakým způsobem „zadrátované“ v procesoru, a tak nás asi nepřekvapí, že tam nenajdeme složité instrukce typu *spočítej faktoriál* nebo *nakresli hrocha,* ale mnohem jednodušší instrukce. O to rychleji je však procesor dokáže vykonávat: typický dnešní procesor jich zvládne přes 2 miliardy za sekundu!

Instrukce musíme procesoru předávat v nějakém formátu, kterému rozumí. Protože počítače si již od počátku lépe rozumí s čísly než se slovy, i instrukce jsou pro procesor zakódované pomocí čísel. To, jaké číslo znamená kterou instrukci, se může u jednotlivých procesorů značně lišit. Proto existují *instrukční sady,* což jsou pevně dané předpisy, jaké číslo odpovídá jaké instrukci. Pokud jste někdy slyšeli pojmy jako *x86, ARM, PowerPC nebo MIPS,* to všechno jsou instrukční sady. Díky tomu dva x86 procesory chápou instrukci číslo 123 identicky nehledě na to, jestli jsou od Intelu či AMD.

I když počítače lépe rozumí číslům, u lidí tomu tak zdaleka není. Schválně, dokážete z následujících čísel v šestnáctkové soustavě aspoň trochu vytušit, co by ARMový procesor provedl?

```
0c 00 a0 e3
2a 10 a0 e3
91 00 03 e0
```

Pokud nemáte tušení, nebojte. Protože to nešlo většině lidí, velmi záhy stvořili *assemblery.* No posuďte sami, nečte se tohle lépe?

```
MOV  r0, #12
MOV  r1, #42
MUL  r3, r1, r0
```

Pokud si spojíte `MUL` se slovem multiply, možná i odhadnete, že tento program vynásobí 12 a 42, i když jste nikdy žádný assembler neviděli. Assemblery nám tedy ve své nejjednodušší podobě umožňují zapsat instrukce pro člověka čitelnějším způsobem. Assembler se však stále před použitím musí tzv. *assemblovat,* neboli přeložit do číselné podoby instrukcí, kterou jsme viděli dříve.

V celém našem seriálu se budeme věnovat procesorům z rodiny ARM. Rodina ARM zahrnuje několik architektur, neboť si ARM postupně prošel několika verzemi, jejich koncepty jsou však na úrovni assembleru naštěstí velmi podobné, takže by nás neměla jiná verze ARMu zaskočit. Tyto procesory najdete ve většině chytrých telefonů a tabletů a například i v Raspberry Pi.

Protože většina z vás asi nemá v šuplíku ARMový procesor, na kterém byste mohli testovat svá řešení úložek, připravili jsme si pro vás jednoduchý simulátor. Ten najdete na adrese *http://ksp.mff.cuni.cz/viz/asm.*

Čísla, čísla, čísla…

Zatímco počítače umí pracovat s nejrůznějšími typy dat (textem, obrázky, zvukem, …), jejich procesory zvládnou zpracovávat jenom čísla. A navíc ta čísla nesmí být moc velká. Typický procesor pracuje najednou buďto s 32-bitovými nebo 64-bitovými celými čísly. My si budeme povídat o jedné ze starších verzí architektury ARM, která je 32-bitová. Práci s čísly si nicméně pro větší názornost předvedeme na 8-bitových číslech.

Nejprve se podíváme, jak se ukládají přirozená (celá nezáporná) čísla; říká se jim také *čísla bez znaménka.* Procesor je ukládá ve dvojkové soustavě, v každém bitu je uložena jedna číslice dvojkového zápisu. Například číslo 42 bychom zapsali jako `0010 1010` (protože $42 = 2^5 + 2^3 + 2^1$). Řády bývá zvykem oddělovat po čtveřicích, nebo ještě lépe číslo místo ve dvojkové soustavě zapisovat v šestnáctkové (hexadecimální; s číslicemi `0` až `9` a `a` až `f`). Čtyři dvojkové číslice totiž odpovídají právě jedné šestnáctkové. Šestnáctkové zápisy se obvykle uvozují znaky `0x`, takže číslo 42 bychom zapsali jako `0x2a`.

Bity čísla číslujeme od 0 do 7, nejnižší řád má číslo 0, nejvyšší 7; i-tý bit má tedy váhu 2^i. Nejmenší 8-bitové číslo je tudíž `0000 0000` $= 0$, nejvyšší `1111 1111` $= 2^0 + 2^1 + \ldots + 2^7 = 2^8 - 1 = 255$.

Pokud při sčítání čísel dojde k přetečení, čili výsledek se nevejde do 8 bitů, pak přebytečnou část výsledku prostě odřízneme. To je totéž, jako kdybychom řekli, že počítáme modulo 2^8. Například:

$$175 + 85 = \texttt{1010 1111} + \texttt{0101 0101} = (\texttt{1})\,\texttt{0000 0100} = 4.$$

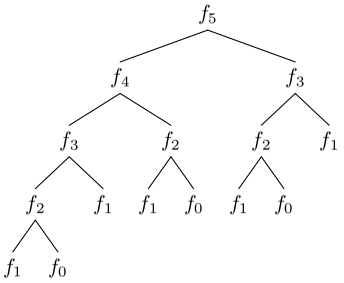
Odčítání také probíhá modulo 2^8, takže kdyby mělo vyjít záporné číslo, přičteme k prvními číslu 2^8, aby výsledek vyšel kladný. Třeba takto:

$$2 - 4 = (\texttt{1})\,\texttt{0000 0010} - \texttt{0000 0100} = \texttt{1111 1110} = 254.$$

V posloupnosti sčítání a odčítání se tedy nemusíme o přetékání starat: dokud víme, že výsledek se vejde do 8-bitového čísla, vyjde správně.

Také si všimněme, že číslo 254 se chová stejně jako -2. To ostatně dává smysl, neboť tato dvě čísla se liší o násobek 2^8, takže jsou modulo 2^8 stejná. Toho můžeme využít k reprezentaci záporných čísel. Jen by se nám občas hodilo, abychom uměli poznat záporné číslo od kladného.

Proto zavedeme reprezentaci *čísel se znaménkem* pomocí *dvojkového doplňku.* Nejvyšší bit nám poslouží jako *znaménkový bit:* bude-li nulový, ukládáme nezáporné číslo obvyklým způsobem (bude tedy v rozsahu 0 až 127). Čísla s jedničkou ve znaménkovém bitu budeme považovat za záporná: `1111 1111` bude znamenat -1, `1111 1110` $= -2$, … až `1000 0000` $= -128$. Ukládáme tedy o 256 více, než je hodnota záporného čísla. Dohromady proto dovedeme ukládat všechna čísla od -128 do 127.



Kdybychom si je namísto opakovaného počítání někde pamatovali, mohli bychom pak odpověď na dotaz na již vypočtené číslo vytáhnout jako králíka z klobouku v konstantním čase. Zavedením jednoho globálního pole, ve kterém si tyto hodnoty pro jednotlivá n budeme pamatovat, nám sníží časovou složitost z $\mathcal{O}(2^n)$ na pěkných $\mathcal{O}(n)$. Takovému postupu se obecně říká *dynamické programování.*

Dynamické programování

Nejprve uveďme na pravou váhu výraz „dynamické“ v názvu. Nevystihuje tak úplně podstatu této techniky a jeho historické pozadí je celkem složité, avšak dnes je tento název již tak zažitý, že se už pravděpodobně nezmění.

Slovo „dynamické“ částečně odkazuje na to, že se dynamicky (za běhu programu) postupně staví řešení jednodušších problémů, která jsou následně použita pro řešení složitějších. Jeho hlavní podstatou je tedy ukládání a opětovné použití již jednou vypočtených údajů.

Hodí se na úlohy, které se dají dělit na podúlohy, které jsou si podobné a mohou se opakovat. Výsledky takovýchto podúloh si poté ukládáme a při dotazu na stejnou podúlohu vrátíme jen uložený výsledek a výpočet již neprovádíme.

Pro další prohloubení znalostí můžete na našem webu nahlédnout do další kuchařky, tentokrát nesoucí (překvapivě) název Dynamické programování.[8]

Prefixové součty

Velmi často se nám hodí si ještě před samotným výpočtem předpočítat a uložit nějaké hodnoty, které poté použijeme.

Představme si například problém nalezení souvislého úseku s největším součtem v nějaké posloupnosti kladných i záporných čísel. Že to není úplně jednoduchý příklad, si ukažme na následující posloupnosti:

$$1, -2, 4, 5, -1, -5, 2, 7$$

Máme zde dvě ryze kladné souvislé posloupnosti, každou se součtem 9 (4, 5 a 2, 7). Ale přesto je výhodnější vzít i nějaké záporné hodnoty a vytvořit tak souvislou posloupnost o součtu 12 (zkuste ji nalézt).

Mohlo by nás napadnout, že prostě zkusíme vzít všechny možné začátky a všechny možné konce. To nám dává $\mathcal{O}(n^2)$ možných posloupností (máme n možných začátků a ke každému z nich řádově n možných konců), pro každou posloupnost si spočteme součet (to zvládneme v $\mathcal{O}(n)$) a budeme si pamatovat ten největší nalezený. Celý náš postup tak trvá $\mathcal{O}(n^3)$.

To není pro takhle jednoduchou úlohu zrovna ten nejpěknější čas, zkusme ho zlepšit. Ukážeme si, jak vypočítat součet libovolné posloupnosti v konstantním čase. Celý princip je vlastně až kouzelně jednoduchý, ale zároveň velmi mocný. Na začátku výpočtu si do pomocného pole P stejné délky jako posloupnost na vstupu (té říkejme S) uložíme takzvané *prefixové součty:* i-tý prefixový součet je součet prvních $i+1$ prvků S, neboli $P[i] = S[0] + S[1] + \ldots + S[i]$.

Pro náš ukázkový případ a pro vstupní pole označené S by to dopadlo takto:

i	-1	0	1	2	3	4	5	6	7
$S[i]$		1	-2	4	5	-1	-5	2	7
$P[i]$	0	1	-1	3	8	7	2	4	11

Pole prefixových součtů umíme získat v linerárním čase – prostě jen od začátku procházíme vstupní pole, počítáme si průběžný součet a ten zapisujeme.

Součet libovolného úseku $a \ldots b$ pak získáme v konstantním čase jako prefixový součet od začátku do indexu b minus prefixový součet od začátku do indexu a. Zapsáno programově to pak je:

```
soucet = P[b] - P[a-1];
```

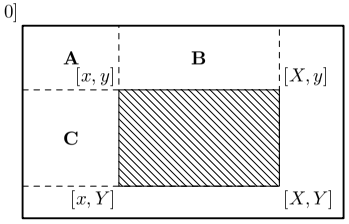
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas; prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

Dvourozměrné prefixové součty

Prefixové součty se dají zobecnit i do více rozměrů, ale princip je vždy stejný. Například dvourozměrné prefixové součty u matice fungují tak, že si předpočítáme součty podmatic začínajících levým vrchním políčkem a končící na indexu $[x, y]$.

Z toho je vidět, že prefixový součet zpravidla obsadí stejně velký prostor jako původní data, v tomto případě tedy budeme mít matici hodnot prefixových součtů končících na zadaných souřadnicích. Jak ale získat součet nějaké podmatice, která se nachází někde „uprostřed“ naší matice?

Použijeme stejný princip jako u jednorozměrného případu: Přičteme větší část, kterou chceme započítat, a odečteme od ní části, které započítat nechceme. Pro případ podmatice začínající vlevo nahoře na pozici $[x, y]$ a končící napravo dole na $[X, Y]$ to ilustruje následující obrázek:



Nejdříve přičteme celý prefixový součet končící na pozici $[X, Y]$. Tím jsme ale započítali i části A, B a C z obrázku, které započítat nechceme. Tak odečteme prefixové součty končící na indexech $[X, y]$ a $[x, Y]$. Ale pozor, teď jsme odečetli jednou $A + B$ a jednou $A + C$, tedy část A (prefixový součet končící na pozici $[x, y]$) jsme odečetli dvakrát, musíme ji proto ještě jednou přičíst.

Celý vzorec tedy vypadá takto:

```
soucet = P[X,Y] - P[X,y] - P[x,Y] + P[x,y];
```

[8] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

nevyjde, zkusíme v tomto kroku použít ještě tříkorunu. Takto se rozhodujeme v každém kroku rekurze a případně se vracíme z neúspěšných větví výpočtu a zkoušíme další možnosti.

Takovým postupem ale vyzkoušíme až exponenciálně mnoho možností ($\mathcal{O}(2^n)$), což není moc rychlé. Proto je doporučováno se backtrackování raději vyhnout, nebo ho nějak chytře vylepšit. Je však dobré o backtrackování vědět, protože existují problémy, které efektivněji řešit neumíme.

Rozděl a panuj

Jednou ze základních technik je rozdělení složitějšího problému na menší části, které opět můžeme rozdělit na menší a tak dále, dokud se nedostaneme k problémům tak malým, že je už umíme triviálně vyřešit.

Binární vyhledávání v poli

Představme si, že máme seřazené pole n prvků a chceme zjistit, jestli se v něm nachází prvek s hodnotou k. Určitě můžeme projít celé pole v lineárním čase (tím, že budeme brát jeden prvek za druhým a kontrolovat, zda je roven hodnotě k), ale to je zbytečně pomalé a nevyužívá toho, že máme pole seřazené.

Můžeme totiž začít s velkým problémem a ten postupně zmenšovat na stále menší a menší. Nejdříve hledáme k v celém poli. Podíváme se na jeho prostřední prvek:

- Pokud je roven k, jsme hotovi.
- Je-li větší než k, víme, že se k musí nacházet nalevo od něj. Můžeme tedy hledat znovu, ale tentokrát se omezit jen na levou polovinu pole.
- Analogicky, je-li menší než k, můžeme hledat jen v pravé polovině.

Když tímto postupným dělením problémů na menší dojdeme až k poli o velikosti jednoho prvku, stačí tento prvek jenom porovnat, dál už se pole nepokoušíme rozdělovat.

Jelikož se nám každým krokem problém zmenší na polovinu, maximálně po $\log n$ krocích se dostaneme na pole velikosti jedna. Říkáme, že algoritmus má *logaritmickou časovou složitost*, píšeme $\mathcal{O}(\log n)$.[6]

Prakticky postup provádíme tak, že si udržujeme levý a pravý okraj aktuálně zpracovávaného úseku a postupně je k sobě přibližujeme.

Ukázka hlavní smyčky v C:

```
int pole[] = {1,2,5,7,12,16,42};
int hledane = 8;
int L = 0, R = 6;
int x;

do {
    int prostredni = (L+R)/2;
    x = pole[prostredni];
    if (x == hledane)
        printf("Pole obsahuje hledane\n");
    else if (x < hledane)
        L = prostredni + 1;
    else
        R = prostredni;
} while (L < R && x != hledane);
if (x != hledane)
    printf("Hledane neni v poli\n");
```

Ukázka v Pythonu jako funkce vracející index prvku nebo -1, pokud hledané číslo nenalezne:

```
def bin_vyhled(pole, hledane,
               levy_index=0, pravy_index=None):
    if pravy_index is None:
        pravy_index = len(pole)
    while levy_index < pravy_index:
        prostredni = (levy_index +
                      pravy_index) // 2
        x = pole[prostredni]
        if x < hledane:
            levy_index = prostredni + 1
        elif x > hledane:
            pravy_index = prostredni
        else:
            return prostredni
    return -1

# Zavolání:
print(bin_vyhled([1,2,5,7, 8,12,16,42], 1))
```

Další aplikace

Další typickou aplikací postupu rozděl a panuj je například třídění posloupnosti pomocí *Mergesortu*. Ten v základu funguje tak, že každou posloupnost, kterou dostane k setřídění, rozdělí na poloviny a každou z nich setřídí rekurzivním zavoláním sebe sama. Zanořování se zastaví ve chvíli, kdy třídíme posloupnost délky jedna (ta už je z podstaty setříděná). Pak jen v každém kroku ze dvou setříděných menších posloupností vyrobí jejich slévání setříděnou posloupnost dvojnásobné délky.

Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[7]

Předpočítané mezivýsledky

Motivací k této kapitole je následující motto: „Proč počítat něco vícekrát, když nám to stačí spočítat jednou a zapamatovat si to?“.

Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme připomenout naši rekurzivní implementaci počítání Fibonacciho čísel zmíněnou výše.

Když se podíváme na výpočet čísla `fib(5)`, vidíme, že pro něj voláme `fib(4)` a `fib(3)`, `fib(4)` volá `fib(3)` a `fib(2)`, `fib(3)` volá `fib(2)` a `fib(1)` a tak dále. Všimli jste si, kolikrát se nám tyhle výpočty opakují? Některá Fibonacciho čísla spočteme totiž zbytečně mnohokrát.

[6] Pokud není řečeno jinak, znamená pro nás v informatice značka log *dvojkový logaritmus*, což je funkce opačná k funkci 2^n a roste o hodně pomaleji než funkce lineární. Pro velká n platí: $1 < \log n < n$ a například $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

[7] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`

Dodejme ještě, že sčítání a odčítání funguje úplně stejně pro čísla se znaménkem jako bez znaménka: obojí pracuje modulo 2^8, jen si pokaždé výsledek vykládáme různě. Násobení a dělení už ale musí znaménka brát v úvahu.

Škatulata, hejbejte se

Procesor si čísla, se kterými zrovna pracuje, potřebuje někde pamatovat. K tomu slouží *registry*. Registry našeho ARMu jsou 32-bitové a je jich celkem 16. Prvních 13 z nich se jmenuje `r0, r1, ..., r12` a jsou *general purpose* (tedy k obecnému použití), což znamená, že si do nich můžeme ukládat naprosto cokoliv. Poslední tři registry mají speciální význam, o kterém si však něco povíme až v příštím dílu seriálu.

První instrukcí, kterou si společně představíme, je `MOV`. Ta slouží k přesunu dat (z anglického *MOVe*). Pozor na to, že argumenty dostává v pořadí *kam, co*. První argument je vždy název registru, druhý argument může být buď název jiného registru nebo číselná konstanta. Pokud chceme v ARMovém assembleru zapsat číselnou konstantu, je třeba před samotné číslo napsat znak `#`. Pro pořádek ještě zmíníme, že zavináčem začíná komentář do konce řádku. Příklad použití:

```
MOV  r7,  #42   @ Do r7 se uloží číslo 42
MOV  r10, r7    @ Do r10 se uloží číslo 42 z r7
MOV  r2,  #0xFF @ Do r2 se uloží číslo 255
```

Když už jsme si pořídili počítač, bylo by hezké, kdyby i něco počítal. K tomu si musíme ukázat ještě jeden registr, který se chová jinak než všechny ostatní. Nazývá se `CPSR` a je to takzvaný registr příznaků (anglicky *flag register*). Aritmetické operace mohou do tohoto registru nastavit jednotlivé příznaky podle toho, jak výpočet dopadl. Na ARMu máme čtyři aritmetické příznaky, a to `N` (*Negative*), pokud je výsledek záporný, `Z` (*Zero*), pokud je výsledek nula, a `V` (*oVerflow*) upozorňující, že došlo k přetečení. Čtvrtým příznakem je `C` (*Carry*), do kterého se uloží ten bit výsledku, který se do registru při ukládání již nevejde (tedy na našem 32-bitovém ARMu 32. bit, číslujeme-li od nuly). K čemu je to dobré, uvidíme záhy.

Sčítání a odčítání zajišťuje šestice instrukcí `ADD` *(ADD)*, `ADC` *(ADd with Carry)*, `SUB` *(SUBtract)*, `RSB` *(Reverse SuBtract)*, `SBC` *(SuBtract with Carry)*, `RSC` *(Reverse Subtract with Carry)*. Pozor na to, že odčítání nastavuje carry opačně, než odpovídá definici!

- `ADD a, b: a = a + b`
- `ADC a, b: a = a + b + carry`
- `SUB a, b: a = a - b`
- `RSB a, b: a = b - a`
- `SBC a, b: a = a - b - (1 - carry)`
- `RSC a, b: a = b - a - (1 - carry)`

Instrukce, které pracují s carry, nám umožňují sčítat velmi velká čísla – to, co se nám přeneslo z předchozího bloku, prostě přičteme k bloku dalšímu. Jelikož je u odčítání carry definováno obráceně, SBC a RSC odčítají jeho negaci. Také pozor, že na ARMu příznaky mění pouze varianty instrukcí zakončené písmenem `S`, např. `ADDS` nebo `RSCS`!

ARM nám zároveň umožňuje uložit výsledek do libovolného registru, k čemuž slouží instrukce se třemi parametry, např. `ADD c, a, b` uloží `a+b` do `c`. Stejně jako u `MOV` je `a` registr a `b` může být registr nebo číslo.

Nejlépe si to ukážeme na příkladu:

```
MOV   r1, #42
MOV   r2, #30
ADD   r3, r1, r2   @ r3 = 72
SUB   r4, r1, r2   @ r4 = 12
RSCS  r1, r2       @ r1 = -12, příznak N
```

Pro násobení existuje na ARMu mnoho instrukcí, my si pro jednoduchost ukážeme pouze jednu z nich, která nám pro násobení malých čísel bude stačit. Je to `MUL` *(MULtiply)* pro násobení znaménkových čísel a používá stejný formát argumentů jako aritmetické instrukce. U ní existuje i varianta `MULS`, která nastaví příslušné příznaky. S dělením je situace přehlednější, existují pouze instrukce `SDIV` *(Signed DIVide)* pro znaménkové dělení a `UDIV` *(Unsigned DIVide)* pro bezznaménkové, ovšem tyto instrukce nemají variantu nastavující příznaky. Násobení i dělení používají opět 2 až 3 argumenty se stejným významem jako ostatní aritmetické operace.

Úkol 1 [4b]: Napište v assembleru posloupnost instrukcí, která do registru `r0` spočítá povrch kvádru, jehož rozměry jsou zadané v registrech `r1`, `r2` a `r3`.

Podobnou kategorii operací představují bitové operace `AND` (bitové and), `ORR` (bitové or), `EOR` (bitové xor) a `BIC` (bitové and not). Tyto instrukce vždy berou tři argumenty, tedy cílový registr, zdrojový registr a třetím parametrem může být další zdrojový registr nebo číslo.

Tyto operace fungují nezávisle pro jednotlivé bity čísla: i-tý bit výsledku z_i spočítáme z i-tých bitů vstupních čísel x_i a y_i. Pro `AND` je přitom $z_i = 1$ právě tedy, když je $x_i = y_i = 1$. Pro `OR` je $z_i = 1$, kdykoliv $x_i = 1$ nebo $y_i = 1$. A pro `EOR` potřebujeme, aby právě jeden z bitů x_i a y_i byl jednička. `NOT` prostě přepne jedničku na nulu nebo nulu na jedničku. Zde je příklad:

```
MOV   r1, #0xF0FF
MOV   r2, #0x0FF0
AND   r3, r1, r2   @ r3 = 0x00F0
ORR   r4, r1, r2   @ r4 = 0xFFFF
EOR   r5, r1, r2   @ r5 = 0xFF0F
```

Hop sem, hop tam

Občas se nám může hodit skočit na jiné místo programu. K tomu na ARMu slouží instrukce `B` *(Branch)* (v jiných instrukčních sadách se často jmenuje `JMP` *(JuMP)*). Abychom mohli assembleru říct, kam má daný skok vést, tak si příslušné místo v programu pojmenuje návěštím (anglicky *label*). Návěští v sobě může mít číslice, písmena anglické abecedy a podtržítka. Hezký nekonečný cyklus by mohl vypadat třeba takto:

```
  MOV   r1, #42
cyklus:
  SUB   r1, #1
  B     cyklus
```

Pokud bychom nyní chtěli v assembleru začít psát nějaké užitečnější programy, asi by nám velmi rychle začaly chybět podmínky. To většina procesorů řeší implementací podmíněných skoků – v případě, že se podmínka vyhodnotí jako pravdivá, procesor skočí na cílové návěští, v opačném případě pokračuje následující instrukcí. Jaké podmínky procesor umí? Zde se konečně dostáváme k využití registru příznaků, neboť podmínky využívají právě uložených příznaků.

Pro snazší pochopení názvů podmínek si ještě představme instrukci `CMP` *(CoMPare)*. Ta se chová naprosto identicky

jako `SUBS`, akorát výsledek se nikam neukládá, pouze se podle něho nastaví příslušné příznaky. V pravém sloupci následující tabulky naleznete význam podmínky po provedení `CMP` a,b. Sloupce příznaků obsahují 1 pro nastavený příznak a 0 pro nenastavený.

podmínka	Z	C	N	V	význam
`EQ` *(EQual)*	1				$a = b$
`NE` *(Not Equal)*	0				$a \neq b$
`CS` *(Carry Set)*		1			$a \geq b$ (neznaménk.)
`HS` *(Higher or Same)*					
`CC` *(Carry Clear)*		0			$a < b$ (neznaménk.)
`LO` *(LOwer)*					
`VS` *(oVerflow Set)*				1	Došlo k přetečení
`VC` *(oVerflow Clear)*				0	Nedošlo k přetečení
`HI` *(HIgher)*	$Z = 0 \wedge C = 1$				$a > b$ (neznaménk.)
`LS` *(Lower Same)*	$Z = 1 \vee C = 0$				$a \leq b$ (neznaménk.)
`GT` *(Greater Than)*	$Z = 0 \wedge N = V$				$a > b$ (znaménkově)
`LE` *(Lower or Equal)*	$Z = 1 \vee N \neq V$				$a \leq b$ (znaménkově)
`GE` *(Greater Equal)*	$N = V$				$a \geq b$ (znaménkově)
`LT` *(Lower Than)*	$N \neq V$				$a < b$ (znaménkově)
`MI` *(MInus)*			1		$a - b < 0$
`PL` *(PLus)*			0		$a - b \geq 0$

Příznaky `MI` a `PL` moc nemá smysl používat po `CMP`. Lepší je použít `LT/LO` nebo `GE/HS`. Pro výsledky aritmetických operací se však hodí.

Například si ukažme jednoduchý cyklus, který (pokud je `r1` ≥ 1) spočítá součin `r1` $\cdot$ (`r1` $- 1$) $\cdot \ldots \cdot 1$ do `r0`:

```
  MOV   r0, #1
cyklus:
  MUL   r0, r1
  SUBS  r1, #1
  BNE   cyklus  @ skoč, pokud r1 není 0.
```

ARM je mezi instrukčními sadami výjimečný tím, že dovoluje podmínku k mnoha dalším instrukcím než jen ke skokům. Nicméně zatím se bez toho obejdeme.

Úkol 2 [5b]: Vymyslete, jak pomocí podmíněných skoků zapsat následující pseudokód:

```
if r1 == 0:
  r0 = r0+1
else:
  r0 = r0-1
r2 = r0*2
```

Úkol 3 [6b]: Napište v assembleru posloupnost instrukcí, která do registru `r0` spočítá největšího společného dělitele čísel zadaných v registrech `r1` a `r2`.

To je pro tento rychlý úvod do assemblerů vše, příště se vrhneme na paměti a další pěkné věci.

Jan „Goci“ Gocník & Martin „Medvěd“ Mareš

Recepty z programátorské kuchařky: Základní algoritmy

Tato naše kuchařka je nejzákladnější ze základních a je určena hlavně pro začínající řešitele. To však neznamená, že zkušenější řešitelé do ní nahlédnout nemůžou – třeba na nějakou konkrétní programátorskou techniku, kterou by si potřebovali osvěžit.

V první části kuchařky se seznámíme hlavně se základními principy programování, uchovávání dat v počítači a základy rychlé manipulace s nimi. Po přečtení této části bychom měli být schopní převést své myšlenky z hlavy na papír či do počítače a měli bychom vědět, proč je námi zvolený postup rozumný.

Druhá část nás poté seznámí se základními postupy, jak řešit určité konkrétní problémy. Naučíme se například, jak rychle vyhledávat v uspořádané posloupnosti hodnot nebo jak si pomocí předpočítání usnadnit řešení těžké úlohy.

Většinu klíčových částí se pokusíme též ukazovat v podobě zdrojového kódu ve dvou různých jazycích (v nízkoúrovňovém C, kde je zápis blízký tomu, jak počítač doopravdy pracuje, a v Pythonu, ve kterém se píše o něco příjemněji). Nebudeme ale probírat základy syntaxe těchto jazyků, ty si případně můžete nastudovat jinde.[1]

Část první: Základní pojmy

Algoritmus a program

Pod tajemným slovem *algoritmus* se skrývá jen jiný výraz pro postup. Můžete si to představit jako příkaz od maminky „Běž do krámu, kup chleba, a když budou mít měkké rohlíky, tak jich vem tucet“.[2]

Takovýto příkaz klidně můžeme nazvat algoritmem, ačkoliv to bude asi znít nezvykle – pojem algoritmus se totiž používá hlavně ve světě počítačů. Je to tedy nějaká posloupnost základních příkazů, která řeší nějaký problém. Výběr konkrétního programovacího jazyka rozhoduje o tom, jaké základní příkazy budeme mít k dispozici. Většinou jsou ale skoro stejné.

Mezi základní příkazy patří:

- Manipulace s daty v paměti (uložení či načtení hodnoty, detailněji v další kapitole).
- Provedení nějakého numerického výpočtu ($+, -, *, /$).
- Vyhodnocení nějaké konkrétní podmínky a odpovídající větvení programu: *Pokud platí A, tak proveď B, jinak proveď C.* Přitom B i C mohou být klidně celé *bloky kódu*, tedy libovolně mnoho dalších základních příkazů.
- Opakování nějakého příkazu: *Dokud platí A, dělej B.* Takové konstrukci říkáme *cyklus* a podobně jako u podmínky může být B blok kódu, který se celý opakuje.
- Vstup a výstup programu (typicky vstup od uživatele z klávesnice či načtení vstupu ze souboru; výstup pak může znamenat vypsání výsledku na obrazovku nebo třeba zapsání dat do souboru).

Z těchto základních stavebních kamenů se skládá každý algoritmus. Programem potom rozumíme realizaci algoritmu v nějakém konkrétním programovacím jazyce.

U složitějších programů se pak často setkáme s problémem, že budete mít nějakou posloupnost příkazů, která se bude na spoustě míst programu opakovat, což zbytečně prodlužuje a znepřehledňuje kód.

[1] `http://ksp.mff.cuni.cz/study/odkazy.html`

[2] A jako slušně vychovaní se tedy vydáte do krámu a koupíte tucet chlebů, protože měli měkké rohlíky :-)

se po skončení má vrátit), a pokud má rekurzivní funkce ještě nějaké lokální proměnné, musíme si někde uložit všechny lokální proměnné funkcí, z kterých jsme se doposud nevrátili.

Rekurzivní funkce a převod na nerekurzivní cyklus

Typickým příkladem rekurzivní funkce je výpočet Fibonacciho čísel. Ta jsou definována tak, že $f_0 = 1$, $f_1 = 1$ a n-té Fibonacciho číslo je součtem dvou předchozích ($f_n = f_{n-1} + f_{n-2}$). To nám dává posloupnost čísel 0, 1, 1, 2, 3, 5, 8, 13, … pokračující donekonečna. Pokud toto přepíšeme do programového kódu, tak dostáváme následující zápisy:

V jazyce C:

```
int fib(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

V Pythonu:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Jak vidíme, je přepis celkem přímočarý. Pokud by se nám však rekurze v nějakém případě nelíbila, můžeme se každé rekurze zbavit. Rekurzivní volání totiž můžeme šikovně přepsat na nějaký cyklus se *zásobníkem*.

Pak jen v cyklu odebíráme prvky ze zásobníku, dokud není prázdný, a za každé rekurzivní zavolání do zásobníku přidáme parametry, se kterými bychom naši funkci volali. Tímto postupem převedeme každou rekurzivní funkci na nerekurzivní.

Ještě doplníme poznámku, že ve většině programovacích jazyků každé volání funkce stojí nějaký čas, sice malý, ale když se volání provádí opakovaně, tak se to už nasčítá. Pro reálnou implementaci je tedy nejlepší pokusit se rekurzi převést na nerekurzivní volání, pokud to nějak rozumně jde.

Občas to jde dokonce i jednodušeji a bez zásobníku. Podívejte se na alternativní variantu výpočtu Fibonacciho čísel níže a rozmyslete si, co dělá.

V jazyce C:

```
int fib2(int n) {
        if (n == 0) return 0;

        int a = 0; int b = 1;
        while (n > 1) {
                int pomocna = a + b;
                a = b;
                b = pomocna;
                n--;
        }
        return b;
}
```

V Pythonu:

```
def fib2(n):
    if n == 0:
        return 0
    a = 0; b = 1
    while n > 1:
        (a, b) = (b, a+b)
        n -= 1
    return b
```

Jak vidíte, je i tato funkce elegantní a navíc běhá mnohem rychleji než její rekurzivní varianta. Tato funkce běhá v $\mathcal{O}(n)$, kdežto rekurzivní varianta počítala stejné věci mnohokrát dokola (zkuste si nakreslit nějaký strom volání předchozí funkce, případně se podívat dopředu do podkapitolky Předpočítané mezivýsledky).

Rekurzivní varianta v tomto případě může běžet až v čase $\mathcal{O}(2^n)$, což je pro velká n mnohem pomaleji než $\mathcal{O}(n)$. Dá se ale celkem lehce rozmyslet úprava rekurzivní varianty, aby běžele také v $\mathcal{O}(n)$, zkuste si rozmyslet jak.

Backtracking

S rekurzí silně souvisí i pojem *backtracking*, česky by se snad dalo říci „metoda pokusu a omylu“. Tímto pojmem označujeme proces, kdy postupně zkoušíme všechny možnosti, jak vyřešit nějaký problém.

Metoda pokusu a omylu se tento proces nazývá proto, že pokud již nemůžeme pokračovat dál (třeba v případě, že v bludišti dojdeme do slepé uličky), vrátíme se kus zpět a zkusíme jinou (zatím nevyzkoušenou) možnost. Takto postupně zkusíme každou možnost, a buď nalezneme námi hledané řešení, nebo se vrátíme až na výchozí pozici a zjistíme, že řešení neexistuje.

Backtracking bývá často realizován pomocí rekurze, ukážeme si to na příkladu hledání rozkladu zadané částky na mince o hodnotách 5 Kč a 3 Kč (všimněte si, že v takto omezeném peněžním systému nejde složit třeba částka 7 Kč). Naše funkce dostane jako parametr zbývající částku a zkusí rekurzivně provést rozklad na jednotlivé mince:

V jazyce C:

```
bool rozloz(int castka) {
        // Koncová podmínka rekurze
        if (castka == 0) return true;
        else if (castka < 0) return false;
        else if (rozloz(castka-5)) {
                printf(" 5 Kc");
                return true;
        } else if (rozloz(castka-3)) {
                printf(" 3 Kc");
                return true;
        } else return false;
}
```

V Pythonu:

```
def rozloz(castka):
    if castka == 0:
        return True
    elif castka < 0:
        return False
    elif rozloz(castka-5):
        print(" 5 Kc")
        return True
    elif rozloz(castka-3):
        print(" 3 Kc")
        return True
    else:
        return False
```

V každém kroku zkusíme nejdříve použít pětikorunovou minci a zavoláme se na zbylou částku, a když náš rozklad

tam hrana není (u neorientovaných grafů je navíc matice symetrická – je jedno, jestli vezmeme $[i, j]$ nebo $[j, i]$). Hodí se pro husté grafy, kde $m \sim n^2$.

- **Matice incidence** – řádky reprezentují vrcholy, sloupce hrany. V každém sloupci jsou právě dvě jedničky – indexy vrcholů, mezi kterými hrana vede. Zabírá však $\mathcal{O}(mn)$ a její použití bývá dost neohrabané, takže je většinou lepší dát přednost jiné reprezentaci grafu. Je ale dobré o ní vědět.

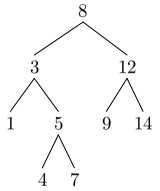
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[5]

Stromy

Možná si říkáte, co má informatika u všech elektronů společného s lesnictvím? Kupodivu celkem mnoho a bez stromů bychom se v leckterém případě jen těžko obešli. Informatické stromy sice nejsou většinou tak zelené, mají ale, na rozdíl od svých dřevnatých sourozenců, mnoho jiných pěkných vlastností.

Strom je vlastně speciálním případem souvislého grafu, který neobsahuje žádnou kružnici (cyklus). To znamená, že mezi každými dvěma vrcholy stromu existuje právě jedna cesta.

Díky této vlastnosti můžeme nějaký zvolený vrchol prohlásit za *kořen* a strom za něj pomyslně zavěsit (tak, že strom roste od kořene směrem dolů), této operaci se říká *zakořenění*. Pak můžeme mluvit o tom, že z kořene směrem dolů (informatické stromy mají tradičně kořen nahoře) vyrůstají nějaké *podstromy*.



Pokud je strom zakořeněný, můžeme v něm mluvit o *hloubce* každého vrcholu, neboli o jeho vzdálenosti od kořene. Hloubka celého stromu je pak nejdelší ze vzdáleností od kořene k nějakému z *listů* (tak říkáme vrcholům, které již nemají žádné *syny*, tedy vrcholy, které by z nich vyrůstaly). Podle hloubky poté můžeme vrcholy stromu uspořádat do jednotlivých *hladin*.

Velmi často používáme stromy, které jsou nějak pravidelné. Příkladem jsou třeba *binární stromy*, které mají v každém vrcholu maximálně dva syny (říkáme jim *levý a pravý podstrom*). Reprezentovat se dají buď obecně jako každý jiný strom (v každém vrcholu spojový seznam podstromů), nebo velmi pěkně i v poli.

Stačí si pomyslně doplnit binární strom na *úplný* (to je takový, který má všechny své hladiny plné) a pak ho od kořene směrem dolů po hladinách očíslovat (kořen dostane číslo nula, jeho synové čísla jedna a dva, další hladina čísla tři až šest, atd.).

Můžeme si všimnout, že pokud si v takovém očíslování vezmeme jakýkoliv vrchol s číslem (indexem) i, jeho synové jsou právě vrcholy s indexy $2i + 1$ a $2i + 2$. Do pole níže je zapsaný binární strom z obrázku výše.

index	0	1	2	3	4	5	6	7	8	9	10
hodnota	8	3	12	1	5	9	14	–	–	4	7

Jak plyne z očíslování, pro úplný binární strom je uložení v poli efektivní a neplýtváme místem. Pokud ale strom úplný nebude, zůstanou nám v poli volná místa. Uložení v poli se tedy vyplatí jen pro stromy, které se od úplných příliš neliší.

Speciálním případem binárních stromů jsou pak ještě *binární vyhledávací stromy*. Jsou to normální binární stromy, pro něž navíc platí, že ať si vezmeme libovolný vrchol, budou hodnoty vrcholů v jeho levém podstromu menší než hodnota tohoto vrcholu a hodnoty v jeho pravém podstromu naopak větší.

V takovém stromu pak zvládneme snadno vyhledávat. Budeme ho postupně procházet od kořene a v jednotlivých vrcholech budeme porovnávat hledanou a aktuální hodnotu a podle toho sestupovat do správného podstromu. Podobná technika je detailněji popsaná ve druhé části kuchařky, v sekci *Rozděl a panuj*.

Na složitější datové struktury stavějící na těchto základních (haldy, intervalové stromy, ...) se můžete podívat do některé z našich dalších kuchařek, na jejichž přehled jsme vás už odkázali o kapitolu výše.

Část druhá: Programátorské techniky

Tato část by měla sloužit jako rychlý přehled a ukázka různých technik, které se dají použít při řešení úloh z KSPčka, nebo při programování obecně.

Rekurze

Rekurze je velmi důležitá programátorská technika. V podstatě znamená definování nějaké věci (ať už je to nějaký objekt či postup výpočtu) pomocí sebe sama.

Rekurzivně může být například zadána nějaká datová struktura. Třeba stromy jsou pěkným příkladem rekurzivně definované datové struktury – každý vrchol stromu může mít syny, a každý z těchto synů je sám o sobě strom (tedy i osamocený vrchol bez synů je stromem).

Prakticky je to realizováno tak, že každý vrchol má svou hodnotu a pak ještě seznam ukazatelů vedoucích na další případné podstromy. S ukazateli jsme se již potkali a s jejich pomocí jsme si postavili spojový seznam. A přesně tak, spojový seznam je také ve své podstatě rekurzivní datová struktura.

Kromě rekurzivních datových struktur se ale často potkáváme i s rekurzivním postupem výpočtu nějakého programu, nejčastěji realizovaným ve formě funkce, která volá sama sebe (většinou s jinými parametry, jinak by to asi nemělo smysl), takové funkci se říká *rekurzivní funkce*.

U rekurzivních funkcí je nejdůležitější věc definovat nějakou *koncovou podmínku*, tedy podmínku, při níž už se rekurze zastaví. Jinak by se totiž mohlo stát, že by rekurze běžela donekonečna.

Přesněji rekurze by se i tak v nějakou chvíli zastavila, ale skončila by chybou, protože by jí došla paměť – každé volání funkce si totiž ukousne kus paměti (musí si pamatovat, kam

[5] http://ksp.mff.cuni.cz/kucharky/

Řešením tohoto problému je použití *funkcí*. Funkci si můžeme představit jako nějakou pojmenovanou část programu (s vlastní pamětí), kterou můžeme opakovaně použít tím, že ji v různých částech programu *zavoláme*. Funkci při zavolání předáme parametry (například seznam čísel), které se dostanou do její vnitřní paměti.

Funkce pak na základě obdržených parametrů může provádět nějaké operace, při kterých pracuje se svojí vnitřní pamětí (mluvíme o *lokální* paměti, změny v ní se neprojeví nikde mimo funkci). Na konci nám funkce může vrátit nějaký výsledek. Pokud funkce během svého běhu změní i nějaká data v *globální* paměti, či provede nějakou globální operaci (například výpis textu na monitor), mluvíme pak o funkci s *vedlejšími efekty* (neboli side-efekty).

Konkrétním příkladem může být funkce, která nám spočítá odmocninu ze zadaného čísla. Ta dostane jako svůj parametr číslo, uvnitř si provede nějaký výpočet, o který se jako uživatel funkce nemusíme starat, a jako výstup nám vrátí spočtenou odmocninu.

Reprezentace dat v počítači

Celkem často si v průběhu výpočtu našeho algoritmu potřebujeme pamatovat nějaké hodnoty. K tomu nám programovací jazyky dávají nástroj s názvem *proměnná*. Ta představuje jakési pojmenované místo v paměti (přihrádku), do kterého si můžeme data ukládat a pak je odtud zase načítat.

Typickým příkladem může být počítání součtu čísel, která nám uživatel zadá na vstupu. Na začátku nejdříve do nějakého místa v paměti uložíme hodnotu 0. Poté postupně, jak nám uživatel zadává čísla, tuto proměnnou pokaždé přečteme, k její hodnotě přičteme nově zadané číslo a výsledek opět uložíme na stejné místo.

Takovéto použití jedné proměnné je velmi jednoduché (tak jednoduché, že ho takto podrobně do řešení KSPčka nepište, není to potřeba), ale také celkem omezené. Co kdybychom si chtěli pamatovat třeba celou zadanou posloupnost čísel? Mohlo by nám stačit vyrobit si spoustu různě pojmenovaných proměnných, ale nejde to lépe? Jde.

Jednotlivé proměnné se mohou kombinovat do složitějších konstrukcí, které obecně nazýváme *datovými strukturami*. Zkusíme si ty nejzákladnější představit.

Pole

První datovou strukturou, kterou si představíme a která se na výše nastíněnou situaci náramně hodí, je *pole*. To představuje spoustu přihrádek (proměnných) naskládaných v paměti za sebou, ke kterým typicky přistupujeme přes jeden společný název pole a jejich pořadové číslo neboli index (jako `NazevPole[0], NazevPole[1], ...`).[3]

Ve většině základních jazyků je pole jen *statické*, tedy v okamžiku jeho vytváření musíme počítači říct, jak ho chceme velké. Některé vyšší jazyky ale nabízejí i pole, které se dynamicky zvětšuje, takovou konstrukci si ukážeme ve druhé části kuchařky.

Abychom nebyli omezeni jen jedním rozměrem, můžeme si klidně vyrobit pole dvourozměrné (případně obecně n-rozměrné). Dvourozměrné pole je vlastně tabulka hodnot, nazýváme ji také někdy *matice*, a může se nám hodit například při reprezentaci různých map (plán bludiště) nebo, jak uvidíme níže, pro reprezentaci dalších datových struktur.

U pole již má smysl přemýšlet, jak dlouho bude která operace trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, je snadné spočítat umístění konkrétní přihrádky. Proto když se počítače zeptáme na obsah přihrády `pole[42]`, vrátí nám hodnotu ihned.

Tomu budeme říkat *operace v konstantním čase* a budeme značit, že trvá čas $\mathcal{O}(1)$. Efektivitu programu totiž nepočítáme v sekundách (protože každý z nás má asi jinak rychlý počítač), ale v počtu základních operací, které musí program řádově vykonat. Více o časové složitosti si můžete přečíst v kuchařce o složitosti,[4] nejdříve však doporučujeme dočíst tuto kuchařku.

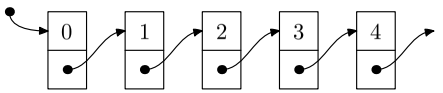
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu pozici dál, aby se vkládaný prvek vešel na své místo. Taková operace tedy může pro pole délky N (čili pole obsahující N prvků) trvat řádově až N kroků, což zapisujeme jako $\mathcal{O}(N)$ a říkáme, že je to vzhledem k N *lineární časová složitost*.

To je značná nevýhoda oproti struktuře, kterou si ukážeme za chvíli. Určitě ale pole nezavrhujme. Je to základní datová struktura, která nalezne použití ve spoustě programů, a jak si ve druhé části kuchařky ukážeme, můžeme ho použít třeba k rychlému hledání hodnoty metodou *binárního vyhledávání*. Nyní ale již slibovaná další datová struktura.

Spojový seznam a ukazatele

Pole jsme měli v paměti určené jenom tím, že počítač věděl, kde je jeho začátek a kolik místa v paměti zabírají jeho prvky. Při dotazování na konkrétní index pak podle indexu a podle velikosti prvků počítač přesně věděl, kam do paměti se má podívat, aby našel námi požadovaný prvek (to vše zvládl v konstantním čase). Jednotlivé prvky si tedy vůbec nemusely pamatovat, kde se nachází jejich sousedi, protože všechny prvky seděly v paměti za sebou.

Představme si ale teď situaci, kdy by si každý prvek ještě pamatoval pozice sousedů. Pak bychom mohli mít prvky libovolně rozházené v paměti a jen by se na sebe vzájemně odkazovaly (první prvek by tvrdil, že druhý je na pozici X, druhý by tvrdil, že třetí je na pozici Y, a tak dále).



K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti a na tomto místě v paměti je její hodnota.

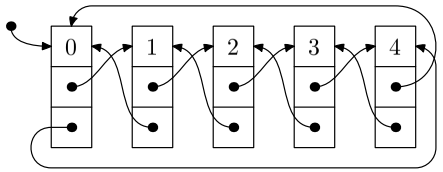
Co kdyby ale hodnota proměnné byla adresa nějakého jiného místa v paměti? Pak takové proměnné říkáme *pointer* a umožňuje nám vytvářet výše popsanou strukturu rozházených prvků v paměti.

[3] Pozor, ve světě počítačů se velmi často indexuje od nuly, tedy první prvek má v tomto případě index 0.

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

Spojový seznam je tedy určený svým prvním prvkem (máme v jedné proměnné pointer na tento prvek, který se často nazývá *kořen*, protože z něj „vyrůstá“ zbytek struktury) a poté u každého dalšího prvku máme za sebou uloženou hodnotu tohoto prvku a odkaz (pointer) na další prvek. Odkazy mezi prvky mohou být i obousměrné, mohou vést dokola (poslední ukazuje na první) či mohou dokonce tvořit nějakou složitější strukturu (pak to ale již nebude čistý spojový seznam).

Pokud pointer nemá nikam ukazovat, realizuje se to odkázáním tohoto pointeru na adresu `NULL`. To skoro doslovně říká „Neukazuji nikam“.



Co nám takto vystavěná struktura umožňuje v porovnání s polem? Přístup na konkrétní prvek v ní stojí lineárně času, protože ho musíme „odkrokovat“ od prvního prvku (na který máme pointer), tedy musíme udělat až $\mathcal{O}(N)$ kroků. Pokud bychom však pointer na daný prvek už nějak měli, samozřejmě na něj můžeme přistoupit v konstantním čase.

Naopak přidávání prvků na konkrétní místo (i jejich odebírání) máme v podstatě zadarmo a spojový seznam můžeme rozšiřovat, dokud na něj máme v počítači paměť. Ve chvíli, kdy chceme přidat nový prvek za prvek, na který máme pointer, jen šikovně přepojíme ukazatele. Pokud předtím ukazatele vedly $A \to B$, teď povedou $A \to C \to B$ (a při odebírání naopak).

Zde můžete vidět ukázku pointerů a spojových seznamů v jazyce C, kde jsou tyto věci mnohem více nízkoúrovňové (ale zato rychlejší):

```
#include <stdio.h>
#include <stdlib.h>
// Příkazy výše načetly do programu
// standardní knihovny a funkce z nich.

// Struktura pro prvek obsahující dopředné
// i zpětné odkazy. Zkráceně tomuto typu
// budeme říkat "tprvek".
typedef struct prvek tprvek;
struct prvek {
    int hodnota;
    tprvek *dalsi;
    tprvek *predchozi;
};

// Vytvoří nový prvek:
tprvek *novy(int i) {
    tprvek *aktualni =
        malloc(sizeof(tprvek));
    aktualni->dalsi = NULL;
    aktualni->predchozi = NULL;
    aktualni->hodnota = i;
    return aktualni;
}

// Odstraní prvek a vrátí pointer na další
// prvek (vrácení pointeru se hodí při
// odstraňování kořene):
tprvek *odstran(tprvek *aktualni) {
    if (aktualni->predchozi != NULL)
        aktualni->predchozi->dalsi =
            aktualni->dalsi;
    if (aktualni->dalsi != NULL)
        aktualni->dalsi->predchozi =
            aktualni->predchozi;

    tprvek *pomocna = aktualni->dalsi;
    free(aktualni);
    return pomocna;
}

// Vloží a vrátí pointer na nový prvek:
tprvek *vloz_za(tprvek *aktualni, int i) {
    tprvek *pomocna = aktualni->dalsi;
    aktualni->dalsi = novy(i);

    if (pomocna != NULL)
       pomocna->predchozi = aktualni->dalsi;

    aktualni->dalsi->dalsi = pomocna;
    return aktualni->dalsi;
}

// Použití:
int main(void) {
    tprvek *koren = novy(1);
    tprvek *aktualni = vloz_za(koren, 2);

    aktualni = koren;
    while (aktualni != NULL) {
        printf("%d\n", aktualni->hodnota);
        aktualni = aktualni->dalsi;
    }

    return 0;
}
```

Zde je ukázka spojových seznamů v Pythonu, kdybychom si je podobně jako v C chtěli naprogramovat sami (Python totiž obsahuje spoustu základních struktur již hotových, podívejte se na modul jménem `collections`):

```
class Prvek:
    def __init__(self, hodnota):
        self.hodnota = hodnota
        self.dalsi = None
        self.predchozi = None

class Spojak:
    def __init__(self):
        self.koren = None

    def vypis(self, aktualni):
        if aktualni is not None:
            print(aktualni.hodnota)
            self.vypis(aktualni.dalsi)

    def vloz_po(self, prvek, za_prvek = None):
        if za_prvek is not None:
            prvek.dalsi = za_prvek.dalsi
            prvek.predchozi = za_prvek
            za_prvek.dalsi = prvek

        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = prvek

        if self.koren is None:
            self.koren = prvek

    def odstran(self, prvek):
        if prvek.predchozi is not None:
            prvek.predchozi.dalsi = \
                prvek.dalsi
        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = \
                prvek.predchozi

# Použití:
prvekA = Prvek("A")
prvekB = Prvek("B")
prvekC = Prvek("C")
prvekD = Prvek("D")

seznam = Spojak()

seznam.vloz_po(prvekB)
seznam.vloz_po(prvekD, prvekB)
seznam.vloz_po(prvekC, prvekD)
seznam.vloz_po(prvekA, prvekC)
seznam.odstran(prvekC)

seznam.vypis(seznam.koren)
```

Fronta a zásobník

S použitím spojových seznamů (nebo v jednodušším případě dokonce i polí) můžeme zkonstruovat dvě velmi užitečné datové struktury, frontu a zásobník.

Fronta funguje tak, jak si ji asi každý z nás představuje: ten, kdo se do fronty postaví první, také první přijde na řadu. Trochu jinak si ji můžeme představit jako trubku, do které na jedné straně sypeme nějaké věci a na druhé je odebíráme. Anglicky je též nazývaná *FIFO* (*„First In, First Out“*).

Praktickou realizaci uděláme jednoduše spojovým seznamem. Budeme si držet dva ukazatele, jeden na začátek seznamu, druhý na konec. Když se objeví nový prvek, který do fronty budeme chtít vložit, přidáme ho na konec, zatímco při odebírání z fronty využijeme druhého ukazatele a vezmeme prvek ze začátku.

Druhou velmi podobnou datovou strukturou je *zásobník*. Jak už ale plyne z anglického názvu *LIFO* (*„Last In, First Out“*), funguje spíše jako plný šuplík: Nahoru do něj přidáváme nové prvky, a když chceme nějaký odebrat, vezmeme také zvrchu. To znamená, že první se na řadu dostane naposledy vložený prvek.

Implementace je velmi obdobná jako u fronty, jen bude ukazatel pouze jeden a bude ukazovat jenom na jeden konec spojového seznamu, konkrétně na poslední prvek.

Knihovny

Tyto základní struktury už jsou často předpřipravené jako součást určitých *knihoven* v daném jazyce. Knihovna je většinou sbírka nějakých navzájem souvisejících funkcí, které již někdo sepsal a které si můžeme do našeho programu načíst a používat. Ukázku načtení knihoven můžete vidět například ve výše zmíněném kódu v jazyce C.

Je ale velmi důležité rozumět tomu, jak knihovní funkce vnitřně fungují. Protože jedině když budeme vědět, co je jak rychlé a efektivní, budeme schopni psát rychlé programy.

Teď již víme, jak reprezentovat nejzákladnější datové struktury v počítači, ale mohlo by se nám hodit zastavit se ještě chvíli u dalších struktur. Tentokrát je už budeme studovat trochu teoretičtěji.

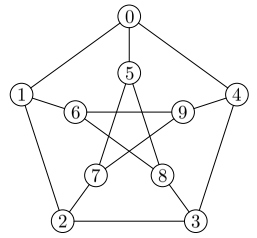
Stromy a grafy v informatice

Grafy

S nějakými grafy jste se již možná potkali, ale tento pojem je bohužel docela přetěžovaný. Jedním jeho významem jsou „koláčové grafy“ a jiné další diagramy znázorňující nějaký poměr (ať už to jsou výsledky voleb, nebo poměr lidí, kteří sledovali v televizi Večerníček).

Další význam můžeme nalézt v analytické matematice, kde se potkáme s grafy průběhu nějakých funkcí. My však nemáme na mysli ani jedno z výše zmíněných, teď se budeme bavit o *kombinatorických grafech*.

Grafem tedy máme na mysli nějakou množinu objektů, říkejme jim *vrcholy*, a nějaké vztahy mezi nimi. Tyto vztahy nazýváme *hranami* a jsou vyjádřené dvojicemi vrcholů, mezi kterými vedou. Ukázku takového grafu vidíme třeba na následujícím obrázku.



Jako praktickou ukázku grafu si můžeme například představit silniční síť nějakého státu: vrcholy budou města a hrany budou silnice, které mezi nimi vedou.

Občas se můžete setkat s pojmem *souvislý* graf. Ten znamená jen to, že mezi každými dvěma vrcholy existuje nějaká cesta. Pokud tomu tak není, je graf *nesouvislý* a dá se rozložit na několik menších grafů, které již souvislé jsou a říká se jim *komponenty souvislosti*.

Samotný graf poté můžeme doplnit tím, že si v každém vrcholu nebo na každé hraně budeme pamatovat nějakou hodnotu (například cenu nejlevnějšího benzínu ve městech a délku v kilometrech na silnicích). Pamatování si hodnot ve vrcholech je docela obvyklá technika a nemá speciální název, ale pokud budeme mít graf, který si pamatuje hodnoty na hranách, budeme o něm mluvit jako o *ohodnoceném grafu*.

Další možnou úpravou je, že každá hrana povede jen jedním směrem (jednosměrné silnice), takovým grafům říkáme *orientované* (pokud pak v orientovaném grafu chceme silnici oběma směry, prostě do něj přidáme dvě hrany, jednu v každém směru).

Poslední, co nám schází k praktickému použití grafů, je naučit se, jak je reprezentovat v počítači. Existuje několik možností (v popisech bude n značit počet vrcholů, m počet hran):

- **Seznam sousedů** – vrcholy grafu budeme mít uložené v poli a u každého vrcholu budeme mít (spojový) seznam čísel dalších vrcholů, do kterých z aktuálního vrcholu vede hrana. Zabírá místo $\mathcal{O}(n+m)$ a hodí se pro řídké grafy (tedy grafy, kde je m řádově stejné jako n).
- **Matice sousednosti** – tabulka $n \times n$, kde na souřadnicích $[i, j]$ je jednička (nebo jiná hodnota, v případě ohodnoceného grafu), pokud z i do j vede hrana, a nula, pokud