Korespondenční Seminář

z Programování

Dokud existují počítače, bude existovat i **KSP**.
Že jsi o něm ještě neslyšel(a)? V tom případě
si zkus odpovědět na následující otázky:

- Zajímáš se o počítače?
- Rád(a) soutěžíš?
- Chceš se dozvědět něco nového?
- Chceš poznat nové lidi?
- Chceš užitečně vyplnit volný čas?
- Hledáš výzvu pro svoji hlavu?

Odpověděl(a) sis alespoň jednou „ano“? Pak hledáme
právě Tebe. Do KSP se může zapojit každý.
Máš-li chuť, otoč list …

Na této stránce najdeš odpovědi na základní otázky o KSP, vesmíru a vůbec.

Co všechno znamená KSP?

Korejská strana práce, Kulturní sdružení Pára, Klub severských psů, nebo třeba Korespondenční seminář z programování! Korejští kynologové a milovníci lokomotiv prominou, zůstaneme u posledního.

Korespondenční seminář z programování?

Celostátní a celoroční soutěž v programování pro studenty středních škol a vyšších ročníků základních škol. Letos pokračuje již svým 27. ročníkem.

Jak tato soutěž probíhá?

Jeden ročník je rozdělen na 5 sérií, přičemž v každé obdržíš zadání 7–8 úloh (poštou nebo po Internetu). Na vyřešení série pak máš několik týdnů, takže můžeš řešit v klidu v teple domácího krbu, v MHD nebo o nudné hodině ve škole.

Opravená řešení Ti později pošleme poštou spolu se vzorovými řešeními, případně si je můžeš stáhnout z našich stránek.

Jaké jsou úlohy?

Úlohy jsou převážně čistě algoritmické. Rychlejší a lépe popsané algoritmy mají přednost před programy hýřícími barvami. Oceníme vymyšlení rychlého a především správného postupu řešení, ne však krásná okénka a barvičky.

Jak se počítají výsledky?

Úlohy jsou za určitý počet bodů dle obtížnosti, do výsledků se každému započítá 5 nejlépe vyřešených úloh ze série. Začátečníky bodujeme mírněji, za drobné chyby ztrácejí méně bodů než zkušení řešitelé. Celkové hodnocení je tvořeno součtem bodů ze všech sérií.

Vůbec nevím, jak začít. Co mám dělat?

Součástí zadání bývá takzvaná *Programátorská kuchařka* – úvodní text o algoritmech a technikách programování. V první sérii najdeš ty úplně nejzákladnější principy a postupy.

A pokud s programováním teprve začínáš, mohla by Tě zajímat začátečnická kategorie KSP s jednoduššími úlohami. Její zadání najdeš na samém konci tohoto letáku.

Může také pomoci přečíst si vzorová řešení starších úloh na našem webu, odkaz najdeš na konci stránky. Na webu najdeš i Programátorskou encyklopedii a další učební texty.

Jak rozeznám lehké a těžké úlohy?

Jednak se můžeš kouknout na body, jež by měly přibližně odpovídat obtížnosti (samozřejmě záleží na znalostech a jak komu úloha sedne), jednak najdeš u některých úloh následující značky:

Takto označenou úlohu (či její část) považujeme za řešitelnou i pro začátečníky, zkušení řešitelé ji jistě zvládnou levou zadní. Pro její vyřešení by neměly být potřeba žádné speciální znalosti.


Aby si i pokročilí přišli na své, zařazujeme někdy do zadání těžkou úlohu, která se může stát leckomu noční můrou. Na její pokoření jsou často potřeba hlubší znalosti algoritmů a datových struktur, odměnou je však vyšší bodový zisk.


Takto označujeme *praktické* úlohy, které máme dvou různých typů. Jedním typem jsou *open-data* úlohy, kde si stáhneš sérii vstupů a odevzdáš jen správné výstupy. Druhým typem jsou úlohy vyhodnocované systémem *CodEx*, kde odevzdáváš přímo zdrojový kód programu. Bližší informace nalezneš přímo u zadání takto označených úloh.


V každém ročníku KSP rozebíráme na pokračování nějaké zajímavé informatické téma do hloubky. Úlohám na toto téma říkáme *seriál* – obsahují kromě samotného zadání ještě text, ve kterém se můžeš dozvědět o tématu něco nového. Jelikož díly seriálu na sebe navazují, vyplatí se mít nastudované i předchozí série.


Často je také v zadání úloha, již lze řešit algoritmem z kuchařky. Pokud se vyskytne, bývá u ní uvedena tato ikonka.

Dostanu za řešení nějakou odměnu?

Nejlepší řešitele zveme na začátku dalšího školního roku (obvykle v září) na týdenní **soustředění**, na kterém se v rychlém tempu střídají hry a odborný program. Vyspíte se až doma!

Dále se každý, kdo překoná 50% hranici bodů, stane úspěšným řešitelem a jako takovému mu budou **odpuštěny přijímačky** na Matfyz!

Jsi-li začínající řešitel, můžeš také jet na **jarní soustředění** (v dubnu či květnu), kde učíme základy programování a algoritmů.

A co když se stanu nejlepším z nejlepších?

Tři nejlepší řešitelé 27. ročníku obdrží libovolnou knihu dle svého ctěného výběru (v případě 2. a 3. nejlepšího jen českou).

Kde se dozvím více a jak se přihlásím?

Další informace a přihlášku nalezneš na

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

Dotazy (ale ne řešení úloh) můžeš posílat na

ksp@mff.cuni.cz

Hodně štěstí!

Korespondenční Seminář z Programování

27. ročník **KSP** **Červenec 2014**

Milí řešitelé a řešitelky!

Vítejte ve 27. ročníku KSP, jehož první leták držíte v ruce. I letos bude každá série obsahovat 7–8 úloh, často s jednoduššími variantami pro začátečníky. Do celkového bodového hodnocení se z každé série započítá 5 nejlépe vyřešených úloh.

Za úspěšné řešení KSP je také možno být přijat na MFF UK bez přijímacích zkoušek. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150.

Upozorňujeme letošní maturanty, že termín odevzdání páté série bude pravděpodobně příliš pozdě na to, aby pátou sérii dohánělí chybějící body. Diplom úspěšného řešitele ale můžeme v případě potřeby zaslat i dříve, budete-li mít dost bodů.

Připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok a tužku. Navíc každému, kdo vyřeší alespoň jednu ze dvou nejvíc bodovaných úloh první série na plný počet bodů, pošleme čokoládu.

Termín série: **20. října 2014 v 8:00 SELČ**
Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*.
Informace: Další podrobnosti o fungování KSP naleznete na *http://ksp.mff.cuni.cz/*. Pokud budete mít jakoukoliv otázku, neváhejte se nás zeptat na *ksp@mff.cuni.cz* nebo na našem fóru. Přejeme hodně štěstí! ;-)

První série dvacátého sedmého ročníku KSP

Tento rok jsme se rozhodli vám v každé sérii přinášet nějaký zajímavý příběh točící se okolo určité zajímavé programátorské chyby. Takové chyby dělá občas každý z nás, leč přesto někdy přerostou v příběh hodný zapamatování. V jednotlivých dílech příběhu se budeme pokoušet držet skutečných událostí, ale dovolíme si i jistou uměleckou licenci.

V prvním dílu se můžete začíst do příběhu, u kterého se nám nepodařilo zjistit, kde se přesně odehrál. Některé zdroje vsazují události tohoto příběhu do vojenského cvičení u Mrtvého moře a některé do Války v Zálivu (konflikt mezi Irákem a koalicí západních zemí v čele s USA v roce 1991), my jsme se rozhodli držet se verze z Perského zálivu.

Náš příběh začíná na palubě letadlové lodě USS Dwight D. Eisenhower *přezdívané posádkou familiárně* Ike*...*

⋆⋆⋆

Poručík Stromboli rázoval chodbou k briefingové místnosti perutě. Ike *byla dlouhá loď, na délku přes 300 metrů, a on dostal kajutu zrovna na opačném konci. Vyhnul se probíhající četě námořních pěšáků a během hlášení palubního rozhlasu o příletu zásobovacího letu se protáhl průchodem do briefingové místnosti.*

Už od rána byla loď napjatá tím, kdy se konečně zapojí do probíhajících operací. Včera v noci vedly některé lodě operačního svazu masivní raketový útok na irácká radarová postavení a proslýchalo se, že dnes v noci se dostane řada i na ně.

A piloti už se nemohli dočkat, aspoň tak působil šum v místnosti, když Stromboli vešel a pokusil se sednout si na nějaké volné místo. Každý chtěl vidět co nejlépe na hlavní bojový plán, ale zároveň mezi piloty existoval určitý kodex, kterého se drželi i při uvolňování místa k sezení.

27-1-1 Zasedací pořádek — 8 bodů

Piloti v briefingové místnosti si postupně sedají na volné židle. Zasedací pořádek si můžeme představit tak, že máme nekonečně dlouhou řadu židlí (židlí je mnohem více, než dorazí pilotů) a každý nový pilot se nejdříve pokusí posadit na prostřední židli.

Pokud je židle, na kterou si chce pilot sednout, volná, je vše v pořádku, pilot se posadí a čeká na briefing. Pokud je však židle již obsazená jiným pilotem, tak, než aby se hádali, sedne si jeden z pilotů na židli o jedno místo vlevo a druhý na židli o jedno místo vpravo (původní židli tak nechají volnou).

Pokud by se při tomto rozsazování náhodou situace opakovala (pilot by se opět chtěl posadit na židli, kde už někdo sedí), bude se postup opakovat tak dlouho, dokud na každé židli nebude sedět maximálně jeden pilot.

Chceme po vás dvě věci:

a) Jak bude řada židlí vypadat po příchodu N pilotů?
b) Dostanete zapsaný nějaký zasedací pořádek pilotů (obsazené a neobsazené židle). Rozhodněte, zdali mohl vzniknout tímto postupem.

Konečně se všichni posadili, do místnosti vešel velitel a začal briefing. Vypadá to dobře, dneska se konečně odlepí od letové paluby, zasnil se Stromboli, a tak si skoro nevšiml na něj mířené otázky.

„Tak Stromboli, přestaňte lelkovat a poslouchejte!“ napomenul ho velitel, „Říkal jsem, že dneska odpoledne provedete s Thompsonem průzkumný let v nízké výšce. Přesné pokyny a letový plán obdržíte během několika hodin, zatím se připravte. Technici vám zrovna na vašeho ptáčka montují průzkumnou výbavu.“

„Ano pane!“ odpověděl spěšně Stromboli a s úsměvem mrknul na svého navigátora Thompsona.

Schůze ještě chvíli pokračovala, než se rozdaly úkoly pro všechny piloty, a pak Stromboli v závěsu se svým navigátorem vyrazil směrem k hangárové palubě. Chtěl si ještě před akcí promluvit s vrchním zbrojmistrem a vybrat si vybavení.

Ve chvíli, kdy dorazil do zbrojnice, trochu se zděsil. Z ve-

dlejšího zbrojního skladu se totiž ozýval děsivý lomoz, a tak tam opatrně nakoukl – vrchní zbrojmistr stál uprostřed místnosti a dirigoval sundavání palet se zbraněmi z vysokých polic.

27-1-2 Zbrojní sklad — 9 bodů

Vrchní zbrojmistr na letadlové lodi potřebuje vyndat ze skladu několik palet s výzbrojí. Sklad je ale zaskládaný do veliké výšky a manévrování s neohrabanými vysokozdvižnými vozíky je v něm celkem nebezpečné, aspoň do doby, než se část věcí vyndá.

Ve skladu operují dva různé vysokozdvižné vozíky (určené pro palety dvou velikostí) a bezpečností předpisy dovolují na začátku sundavat pouze palety uložené na policích v maximální výšce h_0. Na vstupu dostanete popis všech N palet ve skladu, paleta i je velikosti v_i (velká nebo malá), je uložena ve výšce h_i a má nebezpečnost x_i.

Vozíky se musí v sundavání palet střídat (velká paleta, malá paleta, velká paleta, ...) a po vyndání palety s nebezpečností x_i mohou oba vozíky začít sundavat palety z výšky o x_i větší než dosud.

Hlavního zbrojmistra by zajímalo, kolik palet může ze skladu vyvézt ven, aniž by porušil bezpečností předpisy.

Konečně byla Stromboliho stíhačka připravená a vyzbrojená k průzkumné misi. Zašel si tedy na velmi pozdní oběd a pak se opět vydal do hangáru ke svému stroji. Usedl do kokpitu a pustil se do předletové přípravy. Po jejím dokončení pak ukázal palubnímu mechanikovi zdvižený palec a nechal se vyvézt výtahem na letovou palubu, kde počkal, než na něj dojde řada se startem.

Konečně, rameno katapultu se zakleslo za přední podvozkovou nohu „ef čtrnáctky“, Stromboli ukázal technikovi zdvižený palec, přidal tah motorů a pak už se jejich F14 Tomcat vyřítil po krátké vzletové dráze vstříc slunci.

Rychle vystoupali do výšky několika kilometrů a tam začali kroužit. Museli počkat, než dostanou od velitelství povolení k provedení akce. Thompson na zadním sedadle mezitím zapnul nový navigační systém, počkal, než se přijímač GPS ustálí, a začal prověřovat jeho funkčnost a propojení s průzkumným kontejnerem s kamerami, který měli zavěšený pod pravým křídlem.

Stromboli nechal Thompsona hrát si, navedl letadlo na kruhovou vyčkávací dráhu a čekal na finální pokyn k zahájení akce. Vzdušný prostor aktuálně brázdilo mnoho spojeneckých letounů – hlídky, zásobovací stroje i další průzkumné mise – a tak bylo potřeba udržovat přesně vymezené letecké koridory, aby se nikdo s nikým nesrazil. Naštěstí měla jejich mise nejvyšší prioritu.

27-1-3 Letecké koridory — 10 bodů

Vzdušný prostor nad spojeneckým námořním svazem není vůbec prázdný, a tak všechny letouny, které se v něm pohybují, musí dodržovat předepsané letové trasy neboli koridory.

Koridory mají předepsaný směr, kterým se jimi dá proletět, a vedou mezi určenými místy vzdušného prostoru (koridory a místa tak tvoří hrany a vrcholy orientovaného grafu).

Letoun se potřebuje dostat od letadlové lodi k místu plnění své mise, tedy je potřeba nalézt orientovanou cestu mezi dvěma zadanými místy.

Pilot letounu chce letět nejkratší trasou a zajímá ho, kolik má možností volby, tedy kolik různých nejkratších orientovaných cest vedoucích mezi těmito dvěma místy existuje (různé cesty jsou takové, které se liší alespoň v jedné hraně).

Už začal přicházet soumrak, když konečně dostali očekávané rozkazy. Stromboli vysílačkou potvrdil příjem, uchopil knipl a začal s Tomcatem klesat. Když se přiblížili k pobřeží a sestoupili do několika desítek metrů nad vodu, snížil rychlost a změnil šípovitost křídel na pomalý let. Nastavitelná křídla, to byl důvod, proč tyhle starší stroje pořád miloval.

Letoun sestoupil ještě o kus níž, už letěli jen pár metrů nad vlnami. Pod nimi se mihla pláž, Stromboli navedl Tomcat do jedné prolákliny a vtom se to stalo!

Najednou za táhlého pískání zablikaly a zhasly všechny displeje v kokpitu a stroj se hrozivě otřásl, jak se začal naklánět na bok. Stromboli hned popadl knipl a přitáhl ho, šlo to mnohem hůř než obvykle. „Co se stalo, zasáhlo nás něco?“ křikl dozadu na Thompsona. „Nevím. Já... najednou všechno zhaslo, asi porucha.“

Stromboli zaklel, zatracená elektronika, pomyslel si. Jeho pohled zabloudil k panelu vysílačky u levého kolena, ta ještě jako jedna z mála svítila, nebyla připojená k modernizovanému palubnímu počítači. „Mayday, mayday. Tady Krysa jedna, volám základnu. Mayday, mayday. Těžká porucha palubní elektroniky, stroj stěží ovladatelný, vracíme se na základnu.“

Ještě že v zapadajícím slunci byla jasně vidět černá tečka letadlové lodi. Stromboli k ní zamířil a doufal, že to stroj zpět na loď zvládne, bez asistence palubní elektroniky totiž vůbec nevěděl, jaký je jeho stav. Thompson se mezitím vzadu pokoušel zprovoznit alespoň průhledový Head-up display, aby Stromboli při přiblížení viděl před sebou jejich rychlost.

27-1-4 Head-up display — 10 bodů

Head-up display (HUD) zobrazuje informace v zorném poli pilota, a ten tak nemusí sklánět oči dolů a pak se jimi zase vracet. Bohužel je však docela citlivý na vyladění barev a kontrastu a pokud se nastaví nesprávně, spíš pilota ruší.

HUD má N různých prvků. Každému z nich můžeme nastavit kontrast na nějakou hodnotu mezi 0 a K včetně. Prvky jsou na HUDu uspořádány vedle sebe, takže si je můžeme představit jako řadu N čísel.

Když poprvé spustíme HUD, dostaneme nějak nastavený kontrast. Chceme přenastavit kontrast všech prvků tak, aby se žádné dva prvky vedle sebe nelišily o více než D jednotek, a zároveň chceme provést co nejmenší celkovou změnu kontrastu (součet změn bude nejmenší možný).

Formát vstupu: Na prvním řádku budou čísla N, K a D, na druhém řádku pak N čísel udávajících výchozí kontrast všech prvků. Čísla jsou na řádku oddělena mezerou.

Formát výstupu: Na první řádek vypište součet provedených změn, na druhý pak uveďte nové hodnoty kontrastu pro všechny prvky (tedy N čísel oddělených mezerou). Pokud existuje více optimálních řešení, vyberte si libovolné z nich.

Ukázkový vstup:	*Ukázkový výstup:*
`6 30 3`	`3`
`2 7 9 13 16 14`	`4 7 10 13 16 14`

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

Stromboli navedl Tomcat na přistání a vysunul brzdící hák. Tohle přiblížení bez přístrojů nechtěl opakovat a byl rád, že slunce ještě nestihlo zapadnout. Už mu však moc nescházelo, a tak Stromboli s pozorností vybičovanou na maximum zahájil závěrečnou fázi přistávacího manévru.

Zadní kola dosedla na přistávací dráhu. Hák sice minul první brzdící lano, ale o druhé se již pevně zasekl a Tomcat, brzděný pružným lanem, zpomalil na několika metrech dráhy na nulu. Stromboli si vydechl, vypnul oba motory, sundal si helmu a prohrábl si zpocené vlasy. Tohle zvládli, teď bylo potřeba přijít na to, co se stalo.

Ještě ten večer si technici vzali jejich Tomcat do parády. Začali k němu připojovat všemožné diagnostické přístroje a zkoumali jejich údaje. Připojení diagnostických přístrojů k nefunkční elektronice ale není tak jednoduché, každý přístroj má totiž mírně odlišné požadavky na napájení.

27-1-5 Napájení přístrojů — 10 bodů

Technici připojují diagnostické přístroje k rozbité elektronice. Každý kus elektroniky má nějaký svůj povolený rozsah napájení (minimální a maximální hodnotu napětí, při které bezpečně funguje).

Technici mají k dispozici laboratorní zdroje, které je možné nastavit na přesné napětí. Každý laboratorní zdroj může napájet neomezeně mnoho kusů elektroniky.

Protože jsou ale laboratorní zdroje hodně používaná věc a na technické palubě letadlové lodě je o ně velký zájem, chtějí jich technici použít co možná nejméně (aby jich co nejvíce zbylo na ostatní práce). Pomozte jim zjistit nejmenší možný počet zdrojů, se kterými ještě dokážou uspokojit požadavky napájení všech kusů elektroniky dohromady.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[1] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

Technici pracovali celou noc, ale na žádnou závadu na hardware nepřišli. Pro jistotu vyměnili jednotku palubního počítače a navigační systém. Další den večer se průzkumný let měl opakovat a rozkaz dostali opět Stromboli s Thompsonem.

Tentokrát se vznesli už za tmy a rovnou zamířili k oblasti, kterou měli v nízké výšce prolétnout. Stromboli opět klesl s letounem při nízké rychlosti do kaňonu a začali snímkovat kaňonem vedoucí silnici.

Pak Stromboli přitáhl Tomcat do těsného stoupání na konci kaňonu. Teď půjde do tuhého, pomyslel si, a navedl letoun nad hlavní cíl jejich průzkumu, těsný průlet nad irácký letištěm. Díky letu v nízké výšce o nich do poslední chvíle nevěděli, a tak se protiletecká palba začala objevovat až s dlouhým zpožděním.

První zareagovala nějaká hlídka. Spustila palbu z ručních zbraní, se kterými ale neměli skoro žádnou šanci Tomcat zasáhnout. První protiletadlový kanón vystřelil až ve chvíli, kdy dokončili oblet letiště a začali se stáčet směrem zpět k základně.

Stromboli spustil přídavné spalování a za bojového pokřikování navedl letoun do táhlé zatáčky, která je dostala mimo dosah protiletecké palby. Rád by udržoval přídavné spalování déle a vychutnával si ten příval adrenalinu, když ho silné přetížení tlačilo do sedačky, ale pohled na ukazatel paliva mu to rozmluvil.

Stáhl výkon zpět do normálních hodnot a začal svým letem kopírovat zemi, aby se držel mimo dosah posledních fungujících iráckých radarů. Když v tom se to stalo znovu...

Tomcat právě klesal do nějaké prohlubně, když zhasly všechny displeje v kabině a přestal fungovat radar, kterým se Stromboli řídil při nízkém letu nad zemí. Instinktivně přitáhl letoun o kus výš, aby se vyhnul překážkám, které teď neviděl, a opět zahlásil do vysílačky celkové selhání palubní elektroniky.

Teď byla ale situace vážnější, byla noc a návrat na základnu byl o to těžší. Z letecké patroly v blízkosti byla odvelena jedna F15, která je rychle dohnala a s rozsvícenými pozičními světly se usadila půl kilometru před nimi. Takhle je jako pasáček ztracenou ovci dovedla až nazpět k Ike, kde mezitím palubní personál řešil další problém s navigačními světly.

27-1-6 Přistávací světla — 12 bodů

Navigační světla na palubě letadlové lodě jsou tvořena třemi barvami: červenou, zelenou a modrou. Jsou rozmístěna podél dráhy v řadě N světel.

Palubní důstojník chce rozsvítit nějaký úsek světel tak, aby při pohledu skrz všechna svítící světla působil bíle. A zároveň chce, aby světlo bylo co nejsilnější, tedy aby svítilo co možná nejvíce světel.

Potřebuje tedy najít nejdelší souvislý úsek, ve kterém jsou všechny tři barvy zastoupeny ve stejném počtu.

Lehčí varianta (za 8 bodů): Vyřešte stejnou úlohu, ale jen pro dvě různé barvy.

Díky dobře osvětlené přistávací dráze a skvělému pilotnímu umu se Strombolimu povedlo i podruhé usadit letoun do brzdících lan bez jakékoliv navigační pomoci. Teď už byl však rozhněvaný, dvakrát stejná závada se mu vůbec nelíbila. Během převozu letadla výtahem na hangárovou palubu tedy prohodil několik nevybíravých vět s vrchním mechanikem a po ohlášení u velitele a krátkém hlášení padl vyčerpaný do postele. Alespoň že snímky tentokrát dovezli, a mise tak byla hotová.

⋆⋆⋆

Technici mezitím znovu prolezli celou F14 a hledali závadu v hardware. Nikde však žádnou nenašli, a tak obrátili svůj pohled k softwaru. A zde je čekalo velké překvapení, jedno drobné přehlédnutí, které způsobilo pád celého palubního počítače.

Letoun byl totiž vybaven novou verzí systému GPS, která mimo jiné počítá podle signálu z družic i nadmořskou výšku. Ukázalo se však, že se v jednom místě nadmořskou výškou dělí, aniž by byla zkontrolována nenulovost této hodnoty. A jelikož poručík Stromboli navedl letoun při obou letech do nízkého průletu kaňonem, jehož nejnižší bod se nacházel pod úrovní referenční mořské hladiny používané v GPS, došlo v obou případech k dělení nulou.

To pak vlivem propojení přístrojů v F14 zapříčinilo pád zbytku elektroniky (řízení ale ovlivněno nebylo, to je v F14 přenášeno ještě hydraulicky a mechanicky). Technici se z tohoto problému snad poučili a aktualizovali software zbytku amerických letadel – alespoň od té doby žádné podobné příběhy nejsou. Nebo vlastně... ale o tom zase třeba příště.

Jirka Setnička

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

27-1-7 Učíme se s UNIXem — 14 bodů

V letošním seriálu jsme se rozhodli trochu vás seznámit s UNIXovými systémy, přesněji hlavně s tím, jak efektivně (a efektně) používat jejich příkazovou řádku. Naučíme vás, že UNIXová příkazová řádka je kamarád, kterého se nemusíte bát – neoplývá sice (většinou) klikacím barevným rozhraním, kterým vás provede virtuální pomocník v podobě *pana Sponky*,[2] ale o to je mocnější.

Seriál bude směřován hlavně praktickým způsobem a jednotlivé úkoly by vás měly naučit UNIX skutečně používat. Nejprve se ale nevyhneme malému historickému úvodu o vývoji UNIXu a různých shellů, hlavně bashe.

UNIX, POSIX, shell, bash, ... Co to všechno je?

Historie UNIXu se začala psát v roce 1969, kdy v Bellových laboratořích vznikl tak trochu potají nový operační systém (vedení firmy v nově vyvíjeném operačním systému tehdy nevidělo velkou budoucnost, a tak ho jeho vývojáři před vedením vydávali za textový editor, aby na vývoj získali čas a zdroje). Teprve začátkem 70. let dostal systém oficiální podporu a začal se překotně vyvíjet.

UNIX je ale v současnosti registrovaná ochranná známka a mohou ji využívat pouze systémy, které splňují určené podmínky a k tomu platí licenční poplatky (navíc licenci získává vždy jen určitá verze systému, a licencování je tak pro rychle se vyvíjející systémy finančně i časově neúnosné).

Z tohoto důvodu vzniklo několik systémů, které jsou od UNIXu pouze odvozené. Všechny ale splňují společný standard zvaný POSIX,[3] který zaručuje vzájemnou kompatibilitu, a obecně se o nich mluví jako o UNIXových systémech. Nejrozšířenějším z nich je Linux, který sám existuje v záplavě různých variant (tzv. *distribucí*). Dalším známým systémem je například BSD vyvíjený na Kalifornské univerzitě v Berkeley.

Všechny tyto systémy ale spojuje příkazová řádka, ve světě UNIXu se jí říká *shell* a běží v *terminálu*. Terminál je přímo věc, která se stará o čtení vstupu z klávesnice a zobrazení výstupu na monitor, ale nemá žádnou vnitřní logiku, o tu se stará shell (který se zase nemusí zajímat o klávesnici a monitor, ale má už jen svůj *standardní vstup a výstup*).

Shell je jednoduše řečeno textové rozhraní, které umožňuje spouštět příkazy a pomocí nich ovládat celý systém. Je to takový předchůdce grafických rozhraní a existují stroje (například některé servery), kde grafické rozhraní vůbec není nainstalováno a celé ovládání se děje právě jen přes shell.

Ale i shell sám existuje v několika různých variantách, i on se postupně vyvíjel a byl obohacován o nové vlastnosti a příkazy. Shell, se kterým se dneska setkáme skoro na všech linuxových strojích, se nazývá *bash* (zkratka za *Bourne again shell*, což je odkaz na starší *Bourne shell*). V něm se budeme pohybovat většinu času, ale pokusíme se zdůrazňovat, které příkazy jsou univerzální a budou fungovat ve všech POSIXových shellech, a které jsou jen specialitou bashe.

Jak si bash pořídit?

Pokud máte Linux, skoro určitě máte bash nainstalovaný. Stačí ve vašem systému pouze spustit *Terminál* (či nějak podobně nazvanou aplikaci) a objeví se vám (většinou černé) okno, kam je možné psát příkazy a prohlížet si jejich výsledky. Je ale možné, že na svém stroji nebudete mít nastavený bash jako výchozí a spustí se vám nějaký jiný, jednodušší, shell. V takovém případě v něm jen spusťte příkaz `exec bash` a jste v bashi.

Ve Windows je situace o trochu složitější, ale i zde si můžete UNIXový bash pořídit (organizátoři KSP ho při práci ve Windows doporučují jako věc, která vám usnadní život). Nejlepším řešením bude instalace programu *Cygwin*, který vám nainstaluje bash a spoustu šikovných utilit. Jeho instalaci máme popsanou v naší Encyklopedii.[4]

První kroky po systému

Než vůbec uděláme první krok, měli bychom si v rychlosti představit UNIXový souborový systém. Jeho hlavní myšlenkou je, že vše je uspořádáno ve stromu, do kterého se na různá místa zapojují správné věci, třeba různé disky. *Kořen* souborového systému se označuje jako adresář `/` (lomítko), domovským adresářem uživatele `hroch` pak adresář `/home/hroch`. Je klidně možné, že celý adresář `/home` sídlí fyzicky na jiném disku (třeba i síťovém), který se do stromu souborů připojil na správné místo.

Názvy souborů i adresářů mohou tvořit libovolné znaky (mezery, písmena s diakritikou, ...) s jedinou výjimkou, a tou je lomítko, to se totiž používá pro oddělování názvů adresářů v cestě. Nedoporučujeme ale vytváření názvů obsahujících různé speciální znaky jako `[]"?*` a podobně (může se dokonce stát, že některé programy a systémy budou – pro vaši ochranu – vytváření takových souborů blokovat).

V názvech také záleží (na rozdíl od operačních systémů Windows) na velikosti písmen, `fotka.jpg` a `fotka.JPG` jsou rozdílné soubory.

Když poprvé spustíme bash, uvidíme podobný řádek:
`hroch@ksp:~$`

Ten nám říká náš login (jméno, pod kterým jsme se přihlásili), stroj, na kterém náš bash běží, a také nám prozrazuje jméno adresáře, ve kterém se aktuálně nacházíme. Za znakem `$` pak můžeme psát příkazy. Ale počkat, co je vlnka za adresář a kde je připojený ve stromu souborů? Není to nic tajuplného, je to jen zkratka za náš domovský adresář, tedy za `/home/hroch`, aby ve výpisu nezabíral tolik místa.

Začneme se shellovým Hello World: Napište příkaz `echo Hello World` a spusťte ho klávesou Enter. Příkaz `echo` (anglicky „ozvěna“) udělá to, že opíše všechny své parametry na svůj výstup.

Většina příkazů v shellu totiž může být ovlivněna zadanými parametry, některé příkazy bez zadaných parametrů dokonce ani nefungují. Parametry jsou od sebe a od příkazu v shellu odděleny mezerami a rozlišujeme dva základní typy parametrů – přepínače a poziční argumenty. K jejich pořádnému vysvětlení se dostaneme za chvíli, zatím si pojďme ještě chvíli hrát.

Chcete nějaký zajímavější příkaz než `echo`? Zkuste si spustit příkaz `pwd`. Tento příkaz vypíše cestu do aktuálního adresáře (je to zkratka za „print working directory“). Jak se ale přepnout do nějakého jiného? K tomu slouží příkaz na změnu adresáře `cd` („change directory“), tomu do parametru můžeme napsat, kam nás má přepnout (bez parametru nás přepne do našeho domovského adresáře).

[2] personifikovaná nápověda v jednom nejmenovaném kancelářském balíku
[3] `http://cs.wikipedia.org/wiki/POSIX`
[4] `http://ksp.mff.cuni.cz/encyklopedie/cygwin.html`

Zkuste si třeba `cd /` nebo `cd /home`. Cestám zapsaným s lomítkem na začátku říkáme *absolutní* a udávají přesné místo v adresářovém stromě, kam se přesunout. Druhou možností je použít cestu *relativní* vzhledem k pracovnímu adresáři, ta se píše bez úvodního lomítka. Například spuštění `cd hroch` v adresáři `/home` nás přesune do `/home/hroch`. Můžeme si to představit tak, že před takovouto cestu se automaticky připojí výstup příkazu `pwd` a lomítko.

Speciálním „podadresářem“, který se vyskytuje všude, je adresář `..` (dvě tečky). To je odkaz ukazující o úroveň výš, když tedy ve svém domovském adresáři spustíme `cd ..`, přepne nás to do adresáře `/home`. Dalším speciálním adresářem je adresář `.` (tečka), která odkazuje na aktuální adresář – to vám teď může připadat matoucí a nadbytečné, ale v dalších dílech ukážeme, na co se tento odkaz používá.

Poslední, co k pohybu po systému potřebujeme, je příkaz, který by nám vypsal obsah aktuálního adresáře. To je příkaz `ls` (od anglického slovesa „list“). V základní verzi nám vypíše všechny adresáře a soubory v aktuálním adresáři kromě skrytých (začínajících tečkou), později se s ním naučíme některé další šikovné věci.

Pokud si budete chtít nějaký ze zmíněných příkazů více prostudovat, můžete použít **manuálové stránky**. Pro zobrazení manuálových stránek k příkazu `abc` zadejte v bashi příkaz `man abc`, a pokud k příkazu `abc` tato stránka existuje, zobrazí se vám (k příkazům `cd` a `pwd` ale v některých systémech manuálové stránky neexistují, jelikož se jedná o interní příkazy shellu a ne o samostatné programy).

V manuálové stránce je většinou uvedený základní popis příkazu, možné parametry a občas i ukázkové použití. Pro ukončení prohlížení a návrat do shellu stiskněte `q`. Doporučujeme si zběžně pročíst manuálové stránky dále zmiňovaných příkazů, mohou se vám hodit.

Vytváření a mazání adresářů a souborů

Již umíme procházet po adresářích, pojďme si také nějaké vytvořit a smazat:

- Vytvoření prázdného adresáře provedeme zavoláním příkazu `mkdir` *název adresáře*.
- Smazání adresáře (musí být prázdný) uděláme pomocí příkazu `rmdir` *název adresáře*, nesmíme se v tu chvíli ale nacházet uvnitř tohoto adresáře.
- Vytvoření prázdného souboru můžeme provést pomocí příkazu `touch` *název souboru*. Pokud takový soubor neexistuje, příkaz ho vytvoří; pokud existuje, nastaví mu datum modifikace na aktuální okamžik.
- Smazání souboru zařídíme zavoláním `rm` *název souboru*.

Všechny názvy můžeme uvádět i jako cesty, zavolání `touch adresar/soubor` vytvoří `soubor` v `adresar`, pokud `adresar` již existuje (jinak skončí zavolání chybou). Pokud chceme najednou spustit více příkazů, dá se to provést jejich zápisem na jednu řádku a oddělením pomocí středníku.

> **Úkol 1** [2b]: Vytvořte prázdný soubor `test` umístěný v nově vytvořeném podadresáři `~/a/b/c/d` (vlnka tu, jak je zvykem, zastupuje váš domovský adresář). Pak zase adresáře i soubor vymažte. Zkuste použít co možná nejméně příkazů.
>
> U všech úloh odevzdávejte posloupnost příkazů vedoucích ke splnění dané úlohy (pokud nebude uvedeno jinak).

Ještě doplníme další tři příkazy, které souvisejí s prací se soubory:

- Příkaz `cp` (zkratka za *copy*) slouží ke zkopírování (klidně více) souborů do zadaného umístění. Původní soubory zůstanou na místě a v cílovém umístění se vytvoří jejich kopie.
- Příkaz `mv` (zkratka za *move*) dělá podobnou věc jako `cp`, jen soubory nekopíruje, ale přesouvá (a umí přesouvat i adresáře). Stojí za poznámku, že pokud je původní i cílové umístění na stejném fyzickém disku, je `mv` řádově rychlejší, než kombinace `cp` a `rm` – jen se totiž upraví záznam v tabulce souborů a data se fyzicky nemusejí nikam přesouvat (dalo by se říci, že vlastně dojde jen k přejmenování a přepsání adresy).
- Příkaz `cat` (zkratka *concatenate*) vypisuje obsah zadaných souborů na terminál. Hodí se jednak pro prohlížení obsahu krátkých souborů, jednak pro svůj původní účel (konkatenaci – zřetězení). Když zadáme více názvů souborů, `cat` je všechny spojí a vypíše na terminál (později se dozvíme, jak výstup na terminál přesměrovat někam, kde se nám hodí víc).

Příkazy `cp` a `mv` přebírají libovolně mnoho parametrů: vezmou všechny parametry až na poslední jako zdrojové soubory a zkopírují/přesunou je do místa, kam odkazuje poslední parametr. Teď by stálo za to pořádně si rozebrat, co všechno může být obsaženo v parametrech a jaké triky s nimi umíme.

Parametry: Přepínače a poziční argumenty

Přepínače jsou, jak již název napovídá, parametry, které upravují nějakým způsobem běh příkazu. Jsou uvozeny jednou nebo dvěma pomlčkami (je zvykem, že jednou pomlčkou jsou uvozeny jednopísmenné a dvěma pomlčkami vícepísmenné, ale neplatí to vždy).

Například nám již známý příkaz `ls` má přepínač `-l` zapínající dlouhý výstup. Když si tedy spustíme příkaz `ls -l`, vypadne na vás pravděpodobně výpis podobný tomuto:

```
drwxr-xr-x  hroch ksp 4096 čen 16 12:00 adresar
-rw-r--r--  hroch ksp    0 čen 16 12:00 soubor
```

Zde se dozvíme (popořadě) přístupová práva k souboru, jeho vlastníka a skupinu (těmito věcmi se budeme zabývat v některém z příštích dílů), velikost, datum poslední změny a na úplném konci nalezneme název.

Při zápisu více přepínačů je můžeme psát buď všechny samostatně (`prikaz -a -b -c`), nebo můžeme jednopísmenné i sdružit dohromady (`prikaz -abc`), fungovat budou stejně. Pokud nějaký přepínač bude přijímat doprovodný parametr (většinou to je číslo), může zápis vypadat třeba takto: `prikaz -ab 3 -c` (tady přepínač `b` přijal parametr 3), nebo dokonce `prikaz -acb3`.

Pořadí přepínačů je u většiny základních příkazů libovolné (když nebude, upozorníme vás), ale u některých programů, které nejsou napsané tak pečlivě jako základní shellové příkazy, na jejich pořadí záležet už může. V takovém případě doporučujeme pročíst manuál od daného programu.

Druhý typ parametrů nazýváme **poziční argumenty**. Ty se zadávají bez nějakých uvozujících pomlček a tradičně až za všemi přepínači. Často to jsou například názvy souborů a adresářů (viz příkazy `cp` a `mv`).

Zvídavější z vás možná ve spojitosti s výše jmenovanými příkazy napadla jedna otázka: „Jak zkopírovat/smazat soubor s mezerou v názvu?“ Na to se dá jít dvěma způsoby:

- Speciální znaky (mezi něž patří i mezera) můžeme *escapovat*, tedy zbavit je jejich speciálního účinku, předřazením

zpětného lomítka. Napsáním `rm deravy\ nazev` tedy předáme příkazu `rm` jediný parametr – název souboru obsahujícího mezeru.

- Druhou možností je použití uvozovek, příkaz výše bychom mohli přepsat na `rm "deravy nazev"` se stejným účinkem. Použít se dá i zápis s jednoduchými uvozovkami (apostrofy) a v tomto případě by byly ekvivalentní, rozdíl je však v tom, že ve dvojitých se expandují proměnné, kdežto v jednoduchých ne (více o proměnných v příštích dílech).

Poslední důležitou věcí ohledně parametrů je, jak oddělit přepínače od pozičních argumentů. Představme si například, že bychom chtěli smazat soubor, který by se jmenoval „`-f`“. Nemůžeme napsat jenom `rm -f`, protože to se vyhodnotí jako přepínač, a ani `rm "-f"` nám nepomůže, protože bash stejně předá příkazu `rm` jenom parametr `-f`.

Řešením je oznámit příkazu místo, kde končí přepínače. To uděláme pomocí osamocené dvojice pomlček. Za tímto místem se již nemohou nacházet žádné přepínače a příkaz všechno zbylé vyhodnotí jako poziční argumenty. Řešení tedy vypadá takto: `rm -- -f`.

Úkol 2 [2b]: Prostudujte si manuálové stránky příkazů `head` a `tail`, hlavně jejich parametry, a zjistěte, jak vypsat prvních a posledních patnáct řádků souboru a jak vypsat všechno až na prvních patnáct řádek. Vyzkoušejte si to třeba na souboru `/etc/passwd`.

Doplňování a wildcardy

Představme si, že se chceme přepnout do adresáře, jenž má hrozně dlouhý název. Bash nám to dokáže usnadnit: Pokud totiž při psaní názvu zmáčkneme tabulátor, pokusí se doplnit (podle již napsaného začátku) zbytek názvu souboru nebo adresáře.

Pokud existuje několik souborů nebo adresářů, které mají stejný prefix jména, doplní bash po stisku tabulátoru nejdelší společnou část a po dalším stisku zobrazí jména, kterými se dá pokračovat. Pak stačí jen napsat další část názvu, opět stisknout tabulátor a nechat si doplnit zbytek. Věřte, že to řádově urychlí pohyb po adresářovém stromě a je to jedna z nejpoužívanějších kláves. :-)

Dalším dobrým trikem jsou šipky nahoru a dolů, které nám dovolí listovat v historii příkazů a znovu je spouštět nebo upravovat.

Dobře, teď již umíme s bashem pracovat efektivněji, ale co když budeme chtít zkopírovat stovky souborů (třeba fotek z výletu), to je musíme vážně všechny vypisovat? Bash nám pomůže i v tomto případě, užitím zástupných značek neboli *wildcardů*.

Wildcardy fungují jako žolíky, umožní nám nahradit část názvu souboru zástupným znakem. Ve skutečnosti se stane to, že bash najde všechny soubory, které odpovídají použitým zástupným znakům, a nahradí jimi výraz s wildcardy v příkazu (říkáme, že se *expanduje* na tyto soubory). Samotný příkaz tedy nevidí wildcardy, ale dostane od bashe rovnou seznam odpovídajících souborů. Jedinou výjimkou je, když žádné takové soubory neexistují, v takovém případě nechá bash výraz s wildcardy beze změny.

Mezi wildcardy patří:

- Otazník `?` zastupuje libovolný znak: `mal?.txt` tedy odpovídají například soubory `mala.txt`, `maly.txt` a `male.txt` (naopak `mal.txt` neodpovídá).
- Hvězdička `*` zastupuje libovolný (i nulový) počet nějakých znaků: `fot*.jpg` odpovídají `foto001.jpg` i `fot.jpg`. Speciální výjimkou jsou skryté soubory, na ty se wildcardy neexpandují (pokud explicitně nenapíšeme tečku na začátku názvu).
- Hranaté závorky `[]` se používají pro výčet nebo rozsah: výrazu `[13579]` bude odpovídat libovolná lichá číslice, výrazu `[0-9]` libovolná číslice z rozsahu 0 až 9 a výrazu `[0-9A-F]` zase libovolná šestnáctková číslice. Pokud jako první znak v závorce uvedeme stříšku, funguje celá závorka jako negace (výrazu `[^0-5]` odpovídá všechno až na číslice 0 až 5). Stejného efektu docílíme ve většině moderních shellů také vykřičníkem.

  Další speciální konstrukcí shellu, která se často kombinuje s wildcardy, jsou složené závorky `{}`. Nejsou to wildcardy, takže se vůbec nedívají na to, jestli nějaké jimi popisované soubory existují nebo ne, ale daly by se přirovnat spíše k syntaktické zkratce.

  Jejich zápis je tvořen několika výrazy oddělenými čárkami (ve spojení s wildcardy například `*.{jpg,mp[34]}`) a bash udělá to, že ještě před zpracováním klasických wildcardů rozepíše výrazy obsahující složené závorky na všechny jejich možné varianty – vytvoří samostatný výraz pro každou z variant uvedených ve složené závorce (z příkladu výše tak vznikne dvojice `*.jpg *.mp[34]`, která se teprve zpracovává dál).

  Pokud budeme pracovat v bashi (jiné shelly podobnou funkci obecně mít nemusí, i když zase mohou obsahovat jiná vylepšení), můžeme ve složených závorkách použít i rozsah. Zápis `{1..20}` je ekvivalentní s vypsáním dvaceti čísel oddělených čárkou ve složených závorkách.

  Rozdíl oproti wildcardům se dá pozorovat třeba mezi příkazy `mkdir adresar{5,6,7}` a `mkdir adresar[567]`. První provede, co bychom od něj očekávali (vzniknou tři nové adresáře), ale druhý pro neexistenci daných adresářů vytvoří adresář s hranatými závorkami v názvu (wildcardy se neexpandují).

  Pro zkoušení wildcardů v nějakém adresáři můžete použít příkaz `echo`, který vám vypíše všechno, co dostane jako parametry – tedy všechny expandované názvy souborů v aktuálním adresáři, nebo původní wildcard, pokud se expanze nepovedla.

Úkol 3 [1b]: Jak byste smazali soubor, který obsahuje v názvu nějaký wildcard? Například jak byste smazali soubor `a?c` a přitom nesmazali `abc`, nebo smazali `adresar[567]`, ale již ne existující `adresar5`?

Úkol 4 [3b]: Vymyslete co nejkratší zápis pomocí wildcardů, kterému budou odpovídat právě všechny soubory s příponami `jpg`, `jpeg` nebo `gif` v podadresářích aktuálního adresáře; názvy podadresářů musí obsahovat buď alespoň dvě číslice 0 až 9 nebo alespoň dvě písmena anglické abecedy (pozor na velikost písmen). Důkladně popište, co která část výrazu dělá. Můžete předpokládat, že od každého typu bude alespoň jeden soubor a adresář existovat.

Roury a přesměrování vstupu a výstupu

Kromě parametrů pracují ještě shellové příkazy se vstupem a výstupem. Parametry většinou slouží k nastavení chování příkazu, kdežto data, která chceme příkazem zpracovat, patří na jeho vstup.

První metodou zadávání vstupu je přímo jeho psaní na terminál. Zkuste si spustit například příkaz `wc` (zkratka za

word count). Ten slouží k počítání slov, písmen a řádek souborů, které dostane jako parametry, což ale teď nebudeme používat – bez pozičních argumentů totiž `wc` provádí to samé se svým standardním vstupem a na svůj standardní výstup vypisuje výsledek. Ve výchozím nastavení směřují standardní vstup i výstup na terminál.

Když `wc` spustíme, můžeme psát libovolný text, Zadávání ukončíme speciálním znakem `EOF` (*End-of-file*), který napíšeme stiskem Ctrl+D na prázdném řádku. Tím ukončíme vstup a `wc` provede svoji práci – vypíše dané počty. Pokud bychom chtěli příkaz ukončit bez toho, aby vypsal výsledek, můžeme ho násilně zastavit pomocí Ctrl+C.

Toto ale není příliš praktické použití, mnohem lepší by bylo přesměrovat na vstup nějaký soubor. To uděláme pomocí operátoru šipky: Když zapíšeme `wc < soubor.txt`, tak přesměrujeme obsah zadaného souboru na standardní vstup příkazu `wc`. Zkuste si to. Stejně tak můžeme zápis převrátit a jako první napsat přesměrování. Dokonce nemusíme ani okolo operátoru přesměrování psát žádné mezery (lze tedy psát `<soubor.txt wc`). Syntaxe je dost volná, stačí si zvolit styl, který se vám bude nejvíce líbit.

Stejně jako vstup můžeme přesměrovat i výstup, to uděláme šipkou ukazující na druhou stranu, směrem k souboru. Příkaz `ls > seznam.txt` přepíše soubor `seznam.txt` a uloží do něj výstup z příkazu `ls`, na terminálu se v takovém případě žádný výstup neobjeví. Kdybychom namísto přepsání souboru chtěli jenom připojit nové řádky na jeho konec, můžeme použít dvojitou šipku: `ls >> seznam.txt`.

Co když budeme chtít použít výstup jednoho příkazu jako vstup pro druhý? Určitě bychom mohli použít pomocný soubor, třeba pomocí `a > tempfile ; b < tempfile`, ale shell nám nabízí mnohem elegantnější věc, a tou je takzvaná **roura.**

Roura se zapisuje jako svislá čára (na anglické klávesnici ji najdeme nad Enterem) a funguje tak, že vezme standardní výstup příkazu nalevo od sebe a použije ho jako vstup pro příkaz napravo. Zápis `a < soubor | b | c` (nebo ekvivalentní `<soubor a|b|c`) znamená to, že spustíme příkaz `a`, kterému předáme na vstupu soubor a jeho výstup použijeme jako vstup pro příkaz `b`. Výstup příkazu `b` pak použijeme jako vstup pro `c` a výstup `c` zobrazíme na terminálu. Jednotlivé programy přitom běží současně a mezivýsledky si rovnou předávají, ty tedy nezabírají žádné místo na disku.

Pokud budete používat přesměrování do souboru, dejte si ale pozor na jednu věc. Nelze najednou načítat a zapisovat stejný soubor, shell totiž jako první smaže původní soubor (pokud používáme přesměrování jednou šipkou), a pak teprve by se ho pokoušel načíst. Pozor na to hlavně při úpravách již hotového textu, dá se takto nenávratně smazat několikahodinová práce.

Úkol 5 [1b]: Do souboru `datum` vypište na první řádek „Dnes je:“ a na druhý aktuální datum a čas v jakémkoliv formátu. Asi vám k tomu pomůže příkaz `date` a pro zkontrolování obsahu souboru můžete použít buď již zmíněný `cat`, nebo prohlížeč souborů `less` – ten se ukončuje stiskem `q`.

Úkol 6 [1b]: Spočtěte nějakým příkazem počet adresářů a souborů ve svém domovském adresáři (kromě skrytých).

Úkol 7 [2b]: Napište příkaz využívající roury, který ze souboru `soubor.txt` vezme jedenáctou až třicátou řádku včetně a spočítá počet slov na nich. Můžete předpokládat, že `soubor.txt` je dostatečně dlouhý.

Úkol 8 [2b]: Vypište velikost v bajtech všech souborů v aktuálním adresáři, které obsahují v názvu alespoň jednu číslici (nezapomeňte na skryté soubory).

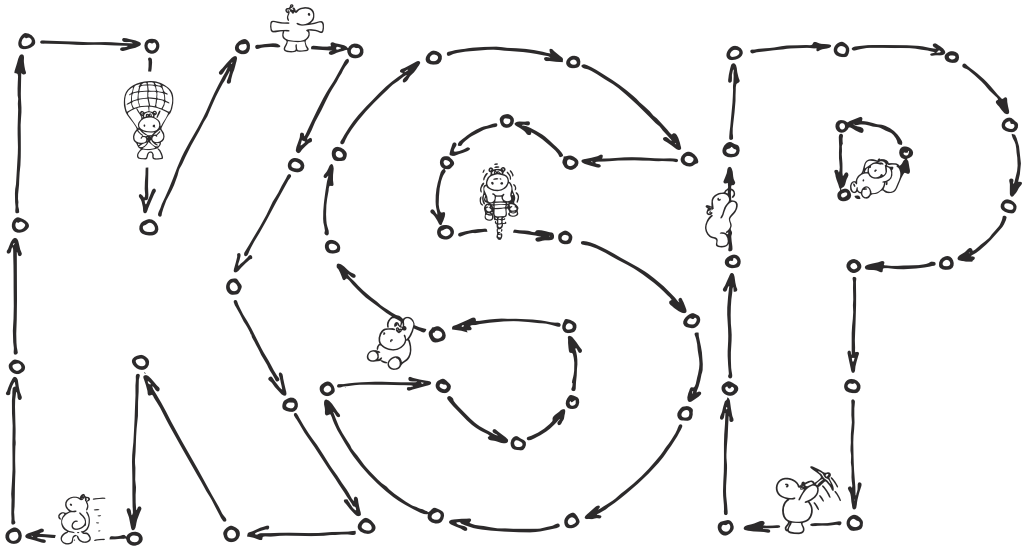
Závěr

Dnes jsme si prošli základní příkazy a principy použitelné v bashi. Zopakujme si všechny příkazy, které jsme se naučili:

- Navigace: `pwd`, `cd`, `ls`
- Manipulace se soubory: `touch`, `cp`, `mv`, `rm`, `mkdir`, `rmdir`
- Obsah souborů: `cat`, `head`, `tail`, `wc`
- Další: `echo`, `less`, `date`
- Manuál: `man`
- Wildcardy, roury a přesměrování vstupu a výstupu.

Příští díl se již můžete těšit na některé pokročilejší UNIXové techniky, my se budeme těšit na vaši účast. :-)

Jirka Setnička



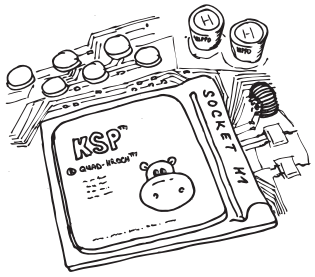
Recepty z programátorské kuchařky: Základní algoritmy

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.[5] Pokud žádný z těchto jazyků neumíte, nezoufejte. KSP můžete řešit i bez toho, stačí když svá řešení důkladně slovně popíšete (konkrétní jazyk se pak můžete naučit až během dalších sérií).



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“.[6]

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 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 setkáte s problémem, že budete mít nějakou posloupnost příkazů, která se bude na spoustě míst programu opakovat, což zbytečně prodlužuje a znepřehledňuje kód.

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.

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

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

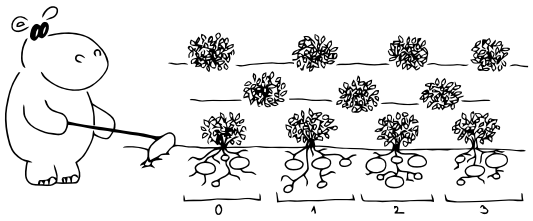
Pole

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], ...`).[7]

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, když se počítače zeptáme na obsah přihrádky `pole[42]`, přesně ví, na kterém místě v paměti se její obsah nachází, a proto nám hodnotu vrátí 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,[8] nejdříve však doporučujeme dočíst tuto kuchařku.

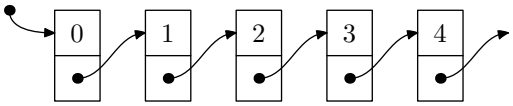
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu pozici dál, aby se vkládaný prvek vešel na své místo. Taková operace tedy může pro pole délky N 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 docela 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).

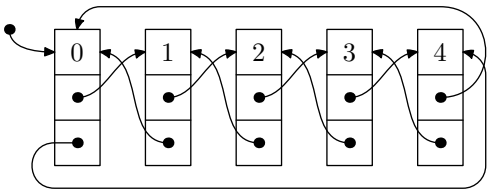


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti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

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):

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

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

```c
#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`):

```python
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 VlozPo(self, prvek, zaPrvek = None):
        if zaPrvek is not None:
            prvek.dalsi = zaPrvek.dalsi
            prvek.predchozi = zaPrvek
            zaPrvek.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.VlozPo(prvekB)
seznam.VlozPo(prvekD, prvekB)
seznam.VlozPo(prvekC, prvekD)
seznam.VlozPo(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, ten také první přijde na řadu. Můžeme si ji také 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 na 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.

Knihovny

Tyto základní struktury už jsou často předpřipravené jako součást nějakých *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.

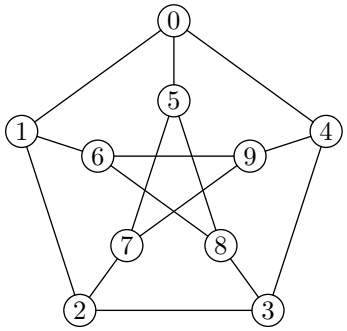
Stromy a grafy v informatice

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 (n bude 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 (případně jiná hodnota, v případě ohodnoceného grafu), pokud z i do j vede hrana, a nula, pokud 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 však dobré o ní vědět.

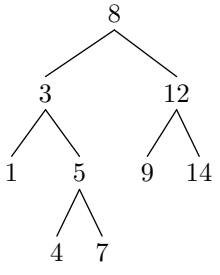
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[9]

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*.



[9] `http://ksp.mff.cuni.cz/study/cooks/`

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, tak 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 podstromě menší než hodnota tohoto vrcholu, a hodnoty v jeho pravém podstromě naopak větší.

V takovém stromě 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 kapitolce *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. Například 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.

Mimo 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 – každá volání funkce si totiž ukousne kus paměti (musí si pamatovat, kam 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;
        else if (n==1) return 1;

        int a = 0; int b = 1;
        for(int i = 2; i<=n; i++) {
                int c = a + b;
                a = b;
                b = c;
        }
        return b;
}
```

V Pythonu:

```
def fib2(n):
        if n==0:
                return 0
        elif n==1:
                return 1

        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 kapitoly Předpočítané mezivýsledky).

Rekurzivní varianta tedy běžela až v čase $\mathcal{O}(2^n)$, což je pro velká n mnohem pomaleji než $\mathcal{O}(n)$ (avšak šla by celkem snadno zachránit, aby běžela 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 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

Představte 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, tak se maximálně po $\log n$ krocích dostaneme na pole velikosti jedna. Říkáme, že algoritmus má *logaritmickou časovou složitost*, píšeme $\mathcal{O}(\log n)$.[10]

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, L=0, R=None):
    if R is None:
        R = len(pole)
    while L < R:
        prostredni = (L+R)//2
        x = pole[prostredni]
        if x < hledane:
            L = prostredni + 1
        elif x > hledane:
            R = prostredni
        else:
            return prostredni
    return -1

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

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 sléváním setříděnou posloupnost dvojnásobné délky.

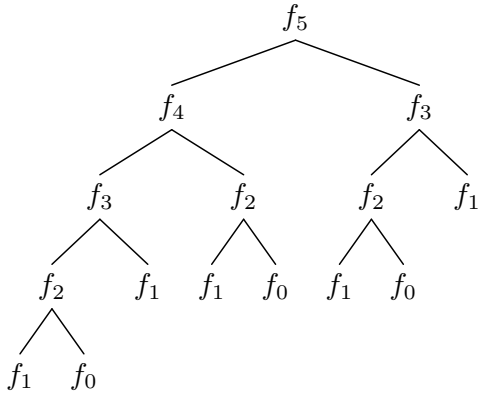
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[11]

Předpočítané mezivýsledky

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

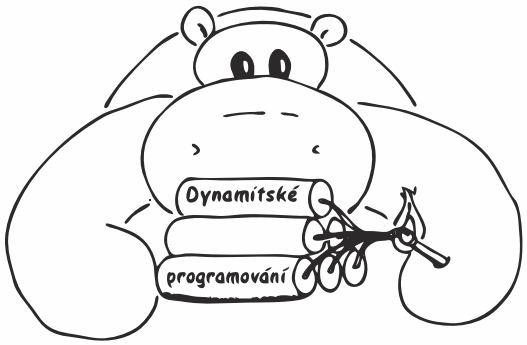
Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme vzít naši rekurzivní implementaci počítání Fibonacciho čísel z kapitolky Rekurze.

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.



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.

[10] 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 $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

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

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.[12]

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];
```

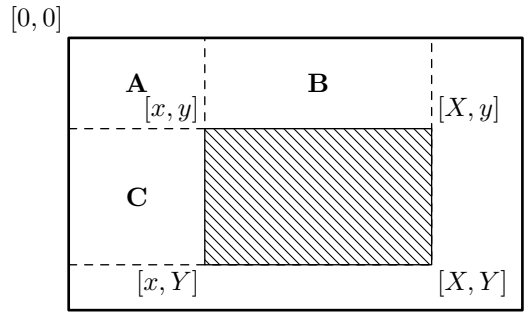
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas, prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

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];
```

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 $\mathcal{O}_p$, pak má předvýpočet smysl.

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

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, tak se již nejedná o úlohu řešitelnou hladovým algoritmem, v takovém případě je potřeba zvolit nějaký chytřejší postup.

Závěr

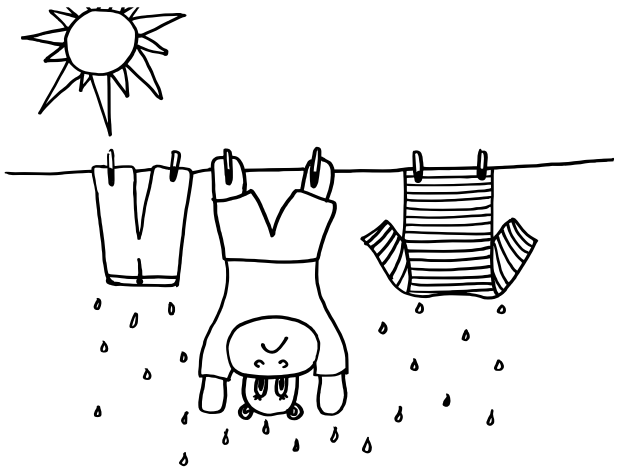
Doufám, že jste si z tohoto rozsáhlého textu odnesli nějaké nové znalosti a poznatky, které vám pomohou nejen v řešení KSP.

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



Chcete-li s námi komunikovat šifrovaně a bezpečně, můžete si ověřit náš HTTPS *certifikát – jeho SHA1 fingerprint je:* `0E:D9:B6:E5:6F:B0:51:D9:66:EB:E9:29:E4:58:AB:5F:99:D6:FD:A3`.