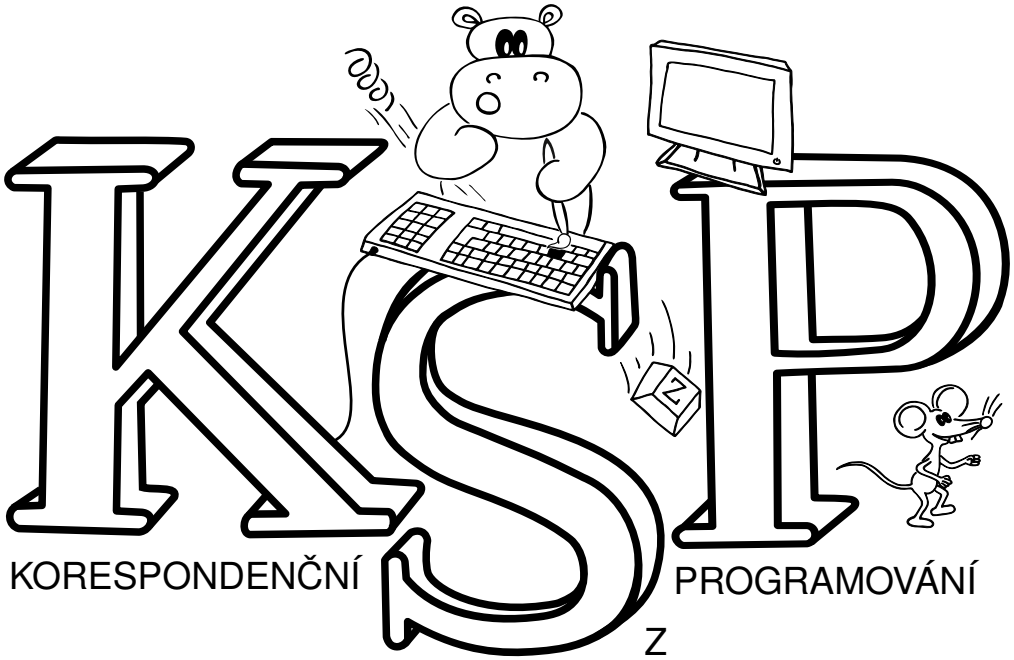
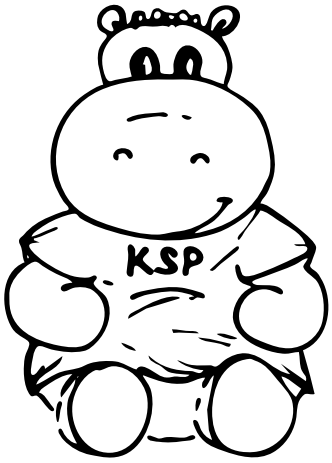
Dokud existují počítače, bude existovat i KSP!





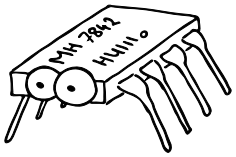
Toužíš po nových vědomostech?

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/*

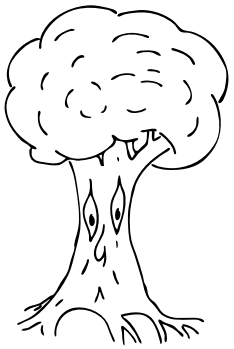
Studijní texty

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

Korespondenční Seminář z Programování

29. ročník **KSP** **Červenec 2016**

Milí řešitelé, řešitelky a řešitelčata!



Dvacátý devátý 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 (dřívější praktické CodExové úlohy jsme se rozhodli v tomto ročníku nepoužívat) a poslední vždy bude seriál.

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: **10. října 2016 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 dvacátého devátého ročníku KSP

„Nepovídej mi, že máme zase místo placek chlebové uhlí!“ poznamenal naoko výhružně Warin a poklepal si na meč zavěšený za pasem. Samozřejmě to nemyslel vážně, ale s Rheou se rádi navzájem škádlili. Znal se s ní déle než s kýmkoliv jiným z jejich malé družinky a prožili toho spolu již hodně – on jako rytíř řádu, ona jako nadaná kouzelnice.

Zbytek jejich družiny nacházející se v severské pustině tvořili ještě nadaný zloděj a lukostřelec Gorf a druhý rytíř, mladý Lian. Do této prapodivné skupinky je svedl důležitý úkol a už více než týden putovali daleko za známými a bezpečnými cestami království.

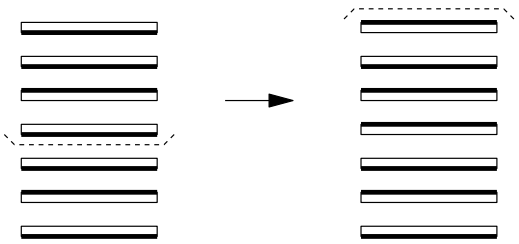
Teď ale byl jejich hlavní starostí ukrutný hlad. Připálené placky sice nejsou zrovna lahůdka, ale když se z nepřipálené strany namažou máslem, tak se jíst dají. Jenom při rychlém sundavání z ohně je Rhea poskládala náhodně na jednu hromádku.

29-1-1 Připálené placky — 8 bodů

Máme na sobě položenou spoustu placek, každá z nich je z jedné strany připálená a z druhé strany krásně dozlatova. Rádi bychom všechny placky otočili nepřipálenou stranou nahoru, abychom je pak mohli všechny rychle namazat máslem.

Bohužel placky jsou ještě příliš horké na to, abychom je brali do rukou. Jediné, co můžeme dělat, je podebrat si několik vrchních placek pánvičkou a celou tuto část otočit. Jak to vypadá, si můžete prohlédnout na následujícím obrázku.

Dostanete zadáno, jak vypadá hromada placek (posloupnost říkající, které z placek leží nahoru připálenou stranou a které nepřipálenou). Vaším úkolem je najít co nejkratší posloupnost podebrání a otočení takovou, aby se po jejím provedení všechny placky nacházely nepřipálenou stranou nahoru.



Další den ráno uklidila družina chvatně své ležení a vydala se dál k úbočí kopce, který se před nimi rýsoval. Nacházelo se zde jedno z těch samostatných měst vzdorujících místní divočině a občasným nájezdům goblinů. Tohle vypadalo, že i docela prosperuje.

Protože jim docházely zásoby a navíc potřebovali zjistit nějaké informace, vydal se Warin s ostatními k městské bráně. Warin s Lianem schovali svá brnění do nenápadných ranců na nákladním mezku a štíty se symboly řádu zakryli plátnem. Bezpečnější bylo tvářit se jako nějací náhodní dobrodruzi.

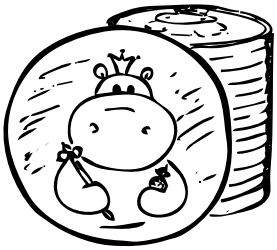
Když u městské brány uplatili strážného dvěma zlaťáky vtisknutými do dlaně, nemuseli ani odpovídat na žádné otázky a byli vpuštěni dovnitř. Po přiblížení se k tržišti se ale družinka stala svědky nějaké hádky. Skupina místních kupců se dohadovala, kdo má komu co zaplatit. Rhea se rozhodla, že se mezi ně vetře, zkusí jim poradit a přitom získat nějaké informace.

29-1-2 Kupecké počty — 11 bodů

Skupina kupců se společně podílela na jedné velké investici. Každý platil nějakou část, některé z nich to stálo více a některé naopak méně. Nyní by si chtěli všechny náklady rovnoměrně rozdělit tak, aby ve výsledku investovali všichni stejně.

Kupci se mohou vyrovnat tím, že ti, kteří platili málo, dají nějaký obnos těm, kteří platili hodně. O každém předání peněz se ovšem musejí dělat záznamy v účetních knihách, a tak by kupci chtěli provést těchto převodů peněz co možná nejméně.

Navíc se ale žádný z kupců nechce zadlužit více, než už je, nebo se naopak nechat přeplatit víc, než už přeplacený je. Není tak třeba možné kupci, který má dostat 100 zlatých, dát 120 zlatých, protože by se tím stal o 20 zlatých přeplacený. Stejně tak není možné poslat jakékoliv peníze kupci, který ostatním dluží (zadlužil by se ještě víc).



Vymyslete postup, který pro zadané částky zaplacené jednotlivými kupci spočítá, kdo komu má kolik dát tak, aby byla respektována uvedená pravidla a převodů bylo málo.

Spočítat řešení s úplně nejmenších počtem převodů je těžký problém a tak to po vás ani nechceme (jako dobrovolné cvičení si však můžete zkusit rozmyslet, proč je to těžký problém). K vyřešení úlohy stačí, když vymyslíte postup vedoucí k maximálně dvojnásobnému počtu převodů peněz, než je optimum. Důležitou částí je i důkaz, že uděláte nejvýše dvojnásobek převodů, než je nezbytně nutné udělat.

Rhea se rychle prodrala davem a využila svého přirozeného půvabu k tomu, aby si od kupců získala pozornost. Tu prohodila nějaké slovo, jinde poradila a kupci se rychle dostali k vzájemné dohodě.

Za necelých dvacet minut se Rhea opět vynořila u zbytku skupinky a vítězoslavně zahlásila: „To byla hračka, to mě bavilo. Od tamtoho obchodníka s kožešinami jsem se dozvěděla, kdo by mohl vědět víc o té potvoře. Je to stopař, kterého najdeme prý v kasárnách městské gardy."

Vydali se tedy do kasáren stojících u východní městské brány. Jak už to tak v těchto malých městech bývá, výzbroj místních gardistů byla dost různorodá – od různě dlouhých mečů přes všelijaká kopí až k náhodně vypadající sbírce halaparten. Co ale oba rytíře příjemně překvapilo, byla důslednost, se kterou se místní velitelé věnovali výcviku. Teď zrovna cvičili gardisté s kopími.

29-1-3 Střídání zbraní — 12 bodů

Gardisté městské gardy mají k dispozici přesně tolik kopí, kolik jich v gardě slouží. Každé kopí je ale jinak dlouhé a jednotliví gardisté jsou také různě vysocí.

Když si gardisté vybírají, se kterým kopím půjdou bojovat, jsou ochotni si vzít jenom kopí nanejvýš tak dlouhé, jak jsou oni vysocí (neboli gardista nemůže mít delší kopí, než je jeho výška).

Při výcviku dbají velitelé na to, aby si co nejvíce gardistů zkusilo bojovat s co nejvíce různými kopími. Zajímalo by je tedy, kolik existuje různých možností, jak si gardisté mohou rozdělit kopí tak, aby každý dostal právě jedno a to nebylo vyšší, než je on sám. Vaším úkolem to je pro zadané délky kopí a výšky gardistů spočítat.

Příklad: Pro kopí o délkách $7, 3, 6, 1$ a pro gardisty vysoké $3, 7, 4, 8$ existují celkem 4 možnosti, jak si kopí mohou rozdělit. Gardisté ve výše uvedeném pořadí dostanou kopí délek $(1, 6, 3, 7)$, $(3, 6, 1, 7)$, $(1, 7, 3, 6)$ nebo $(3, 7, 1, 6)$.

Na dvoře kasáren se rozdělili a začali se co nejnenápadněji vyptávat osamocených gardistů. Nechtěli moc rozhlašovat svoji příslušnost ke královskému rytířskému řádu, to by tady daleko v severních krajích mohlo způsobovat problémy. Lepší bylo zůstat neznámými pocestnými.

Gorfovo písknutí je po chvíli všechny svolalo dohromady. Na jedné straně nádvoří Gorf objevil malého chlapíka, který odpovídal popisu od kupce.

„Prý jsi zahlédl nějakou velkou potvoru," spustil Warin a vtiskl muži do ruky pár zlaťáků. Muž si Warina nervózně prohlédl a pak je všechny posunkem vyzval, aby ho následovali za roh. Tam jim začal líčit své setkání s drakem.

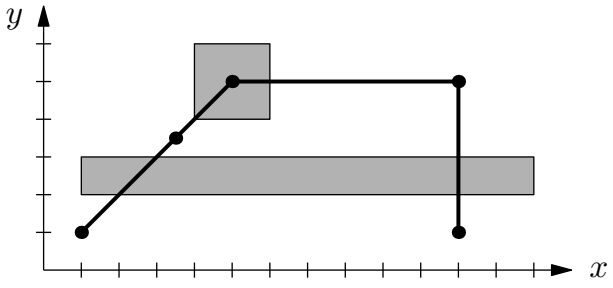
Popsal jim, že procházel místem, kudy už předtím šel mnohokrát, když tu se přes skalní převis nad ním přehnala obrovská potvora – drak. Chvíli se před ním schovával a pozoroval ho ukrytý ve křoví, čekající na příležitost. A když se naskytla, tak vyběhl, jak nejrychleji dovedl.

29-1-4 Zběsilý útěk — 10 bodů

Stopaře překvapil v divočině drak, a tak se před ním dal na útěk. Stopař má představu o tom, jak vypadá okolní terén, takže si naplánoval dostatečně klikatou trasu na zmatení draka. Okolí se skládá buď z normálně prostupného terénu, nebo ze *strašidelných hustolesů*.

Stopařova trasa je tvořena lomenou čárou (neboli na sebe navazujícími úsečkami), která může procházet i skrz hustolesy, ale v takovém případě v nich stopař zvládne běžet jen poloviční rychlostí. Hustolesy jsou obdélníkové oblasti, které budou, stejně jako stopařova trasa, zadány na vstupu.

Navíc máme slíbeno, že každou úsečku na trase protíná nejvýše jeden hustoles a že hustolesů je zhruba stejně mnoho, jako úseček trasy. Představu můžete získat třeba z obrázku níže. Vaším úkolem bude spočítat, jak dlouho bude stopařovi proběhnutí celé trasy trvat.



Při řešení by se vám mohlo hodit nahlédnout do geometrické kuchařky.[1]

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

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

Formát vstupu: Na prvním řádku vstupu dostanete tři celá čísla oddělená mezerou: Nejprve rychlost stopaře v normálním terénu V vyjádřenou v metrech za sekundu. Dále pak číslo N udávající počet bodů na stopařově trase (mezi těmito body se stopař pohybuje po úsečce) a nakonec číslo H udávající počet hustolesů (pozor, jeden hustoles může zasahovat i do více úseček).

Na dalších N řádcích naleznete vždy dvě čísla, každá dvojice udává souřadnice jednoho z bodů na stopařově trase. A nakonec na dalších H řádcích najdete vždy 4 čísla a_x, a_y, b_x, b_y udávající souřadnice levého dolního a pravého horního rohu daného hustolesa. Všechny souřadnice jsou zadány v metrech.

Formát výstupu: Na výstup vypište počet sekund, za které stopař překoná celou trasu, zaokrouhlený na celé sekundy.

Ukázkový vstup:	*Ukázkový výstup:*

`10 5 2`
`100 100`
`350 350`
`500 500`
`1100 500`
`1100 100`
`100 200 1300 300`
`400 400 600 600`
	`205`

Stopař celkově uběhne cca 1566 m, z toho cca 483 m hustolesem.

Po setkání se stopařem se Gorf vydal zařizovat něco do města, možná se zkontaktovat s místním cechem zlodějů, což byl také velmi dobrý zdroj informací. Zbytek družiny se usadil v krčmě a nad džbánkem probíral, co dál.

Byli sem vysláni, aby zjistili více o záhadném nebezpečí ze severu. V království už nějakou dobu kolovaly historky, že na severu se sbíhají nějaké temné síly, ale zatím ani král, ani řád neměli žádné důkazy. Jen spoustu mlhavých příběhů od obchodníků.

Živého draka nikdo neviděl po několik set let. Většinu z nich pobili za dávných dob drakobijci, a pokud nějací draci přežili, tak se myslelo, že spí nekonečným spánkem ve svých hlubokých slujích. Tenhle jeden ale rozhodně nevypadal na to, že by byl vyhuben, ani na to, že by spal nekonečným spánkem. Musela ho probudit nějaká veliká temná síla.

Warin se zamyslel nad jejich nynější situací a přitom se přes svůj džbánek zahleděl na druhou stranu stolu. S úsměvem si všiml, jak Lian pokukuje po Rhee. Rheu znal už spoustu let a vždycky ji vnímal jen jako dobrou přítelkyni. Ale musel uznat, že je to okouzlující kouzelnice a vůbec se nedivil, že mladý rytíř z ní může mít hlavu v oblacích. Rhea samotná si toho pravděpodobně byla vědoma, ale vypadalo to, že jí to vůbec nevadí. Jen kdyby ji Lian pořád neoslovoval „Mylady“ ...

Tok Warinových myšlenek přerušil Gorf, který rozrazil dveře do krčmy, doběhl k jejich stolu a polohlasem pronesl „Warine, Rheo, Liane, na severní hradby útočí drak!“

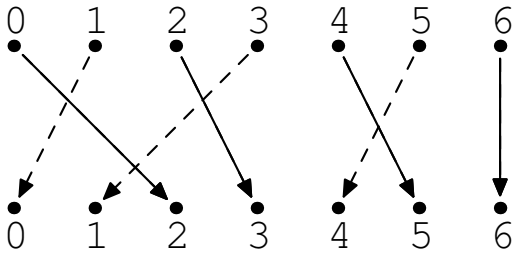
Warin se nerozmýšlel moc dlouho a poslal Gorfa s Rheou přímo na hradby, aby pomohli obráncům. On společně s Lianem se zatím rozeběhli ke stájím, kde měli svého nákladního mezka, aby se převlékli do brnění. Byli sice jen čtyři, ale všichni byli zatraceně dobří bojovníci.

Gorf s Rheou doběhli na hradby právě ve chvíli, kdy se nad nimi přehnal drak, spálil jeden dům těsně za hradbami a začal se otáčet k dalšímu kolečku. Proti němu vylétlo sporadicky několik šípů, ale bylo jich málo a navíc se nezkušení lukostřelci ohrožovali spíše navzájem, než aby trefovali draka. Gorf stáhl ze zad svůj luk a začal je organizovat.

29-1-5 Lučištníci — 10 bodů

Lučištníci stojící v řadě na hradbě mají vyhlédnuté své cíle. Každý z nich míří lukem na nějaké místo, ale je neúčinné a nebezpečné, aby všichni stříleli, jak se jim zachce. Rádi by svou palbu koncentrovali a navíc by měli střílet jen ti, jejichž dráhy palby se nekříží.

Lučištníky můžeme popsat pomocí bodů, na které míří. Například na obrázku níže nultý lučištník míří na bod 2, první na bod 0, druhý na bod 3, třetí na bod 1, čtvrtý na bod 5, pátý na bod 4 a šestý na bod 6.



Ze všech lučištníků chceme vybrat co největší skupinu, jejíž dráhy střelby se nekříží. V uvedeném příkladě jsou to třeba lučištníci 0, 2, 4 a 6 (zvýraznění plnými čarami). Stejně tak by fungovali třeba lučištníci 1, 2, 5 a 6.

Navrhněte algoritmus, který takovou skupinu najde. Pokud existuje více řešení, stačí oznámit libovolné jedno z nich.

Ve chvíli, kdy na hradby doběhli Warin s Lianem v těžké kroužkové zbroji s bílomodrými štíty, už létaly šípy v mohutných salvách. Drak se raději držel dál, ale z lesa naproti městské bráně začali vybíhat goblini.

„Kdo k sakru jste?“ vykřikl překvapeně kapitán městské stráže, když se k němu Warin a Lian rozeběhli. „Rytíři Alvarezova řádu, pokud si chcete zachránit město, poslouchejte nás!“

Protože oba ozbrojenci vypadali, že jsou mnohem bojeschopnější, než kdokoliv z jeho mužů, nemluvě o těžké výstroji, kterou nesli, tak kapitán bez větších námitek souhlasil. Goblini začali dorážet na hradby pomocí žebříků a část z nich se pokoušela i o proražení brány.

Lian si vzal na starost obranu vršku hradeb a Warin začal šikovat gardisty dole pod hradbami, aby se připravili na prolomení brány.

Drak se ale nevzdával. Jakoby ho modrobílé postavičky obou rytířů vyburcovaly k ještě větší bojechtivosti, začal se střemhlavě vrhat na hradby a svým mocným dechem je začal spalovat tak, až pukal kámen.

Rhea se ale také činila. Připravovala si štítové kouzlo a nyní ho začala rozprostírat.

29-1-6 Štítové kouzlo — 10 bodů

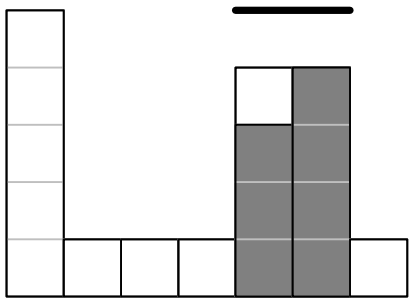
Na městské hradby útočí drak a postupně je ničí. Městské hradby si můžeme představit jako řadu různě vysokých kusů. Drak při každém svém náletu ubere všem nechráněným kusům hradeb jednu jednotku výšky. Níže, než úplně do základů, je ale spálit nemůže (výška hradeb nemůže jít do záporných čísel).

Hradby můžeme chránit pomocí štítového kouzla, ale kouzelnice Rhea umí roztahovat štít vždy jen o jeden kus hradeb mezi každými dvěma drakovy útoky. Navíc chráněná oblast hradeb musí být souvislá.

Začneme tedy tvořit štít nad libovolným kusem hradeb, který je od této chvíle chráněn a již se neničí. V každém dalším kroku ho můžeme rozšířit o jedna doleva, nebo doprava.

Vaším cílem je pro zadané výšky hradeb poradit kouzelnici Rhee, kde má začít a jak má štít postupně rozšiřovat, aby se v součtu uchránilo co nejvíce hradeb (tedy aby součet zbylých výšek hradeb byl co největší).

Příklad: Pro hradby vysoké $5, 1, 1, 1, 4, 4, 1$ je nejlepší začít se štítem nad předposledním úsekem hradeb a pak ho rozšiřovat směrem doleva. Tím se nám povede uchránit celkově 7 dílů hradeb. Kdybychom začali na páté pozici a rozšiřovali doprava, uchráníme stejně dílů hradeb, kdežto kdybychom začali se štítem nad nejvyšší hradbou, tak se nám povede uchránit nejvýše 5 dílů hradeb – vysoké hradby na opačné straně padnou dříve, než nad ně stihneme rozšířit štít.



Na obrázku jsou znázorněny výšky hradeb a části, které z nich zůstanou, pokud začneme stavět štít nad předposlední hradbou a roztáhneme ho doleva (další stavění štítu pak už nic jiného nezachrání).

Jeden z posledních drakových útoků vedl na část hradeb, kde Lian odrážel útočící gobliny. Těsně předtím, než se i zde zhmotnil štít, pronikl drakův útok skrz a smetl všechny obránce společně s hromadou sutin dolů. Rhea se vyděšeně ohlédla, Liana však nikde neviděla. Nesměla ale polevit v posilování štítu, na zachraňování bude čas později!

Městskou bránu mezitím prorazili goblini, ale ani ve snu nebyli připraveni na modrobílý uragán, který se mezi ně vrhl. Warin rozdával rány na všechny strany, městští gardisté se sice také snažili, ale úrovně cvičeného alvarezského rytíře dosáhnout nemohli.

Drak už mezitím utrpěl příliš mnoho zranění od šípů a viděl, že jeho útoky jsou odráženy štítem, a tak s mocným zařváním svůj pokus vypálit město vzdal a dal se na ústup. Ve spojení s krvavou lázní u městské brány to na gobliny bylo asi moc. Většina z nich zpanikařila a začala utíkat nazpět do lesa. Po necelých deseti minutách už na dohled od brány nezůstal jediný živý goblin.

Když se Warin vrátil s gardisty zpět za městskou bránu, uviděl, jak Rhea a Gorf usilovně odhazují trámy a kusy zdiva u jedné zřícené části hradeb, Rhea měla slzy v očích. Hned mu bylo jasné, co se stalo. Odložil zbraně a dal se také do odhrabování. Vyprostili několik živých a bohužel i několik mrtvých vojáků, když tu pod sutinami zahlédli modrou a bílou. Po odhození pár trámů Liana konečně mohli vytáhnout. Nehýbal se.

Pak se náhle ostře nadechl, otevřel oči a rozkašlal. Rhea ho beze slova sevřela do náruče tak silně, až mu málem vymáčkla dech. Zase byli všichni čtyři pohromadě a živí a čekala je tady na severu určitě ještě spousta dobrodružství. Třeba o nich ještě uslyšíme ...

Příběh pro vás vyprávěl

Jirka Setnička

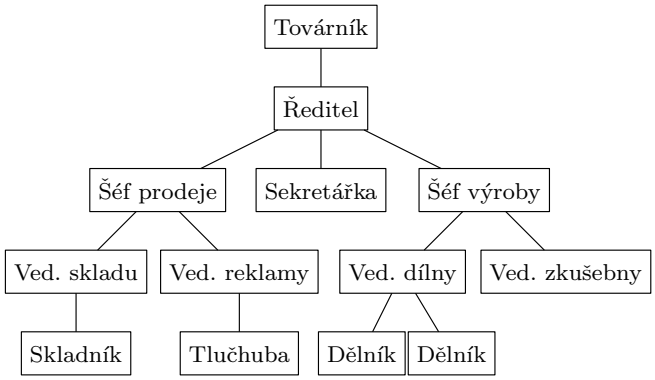
29-1-7 Stromy kolem nás — 15 bodů

Pojďte, v letošním seriálu spolu budeme zkoumat stromy. Ne snad že bychom vás chtěli přeučit na botaniky – půjde nám o stromy ryze abstraktní, matematické. A ještě více o algoritmy pro práci s nimi.

Klíč k určování stromů

Definici stromu najdete v kuchařce této série: je to souvislý graf bez kružnic. To zní učeně, ale kupodivu je to i velmi praktické: příklady takových stromů potkáváme všude kolem sebe.

Hierarchie – kupříkladu továrna má továrníka, pak ředitele, ten své náměstky, jejich podřízenými jsou šéfové oddělení, jim podléhají mistři v dílnách, a tak dále až k řadovým dělníkům. Příslušný strom sestává z vrcholů (což jsou zaměstnanci) a hran (vztahy „být přímým podřízeným“).[2]

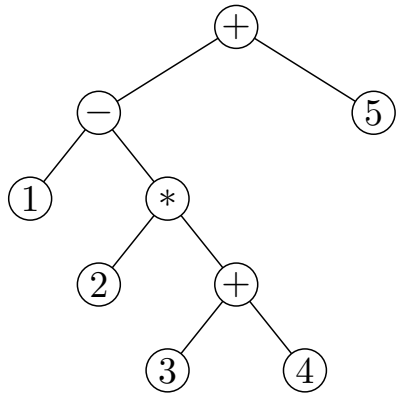


Stromy popisující nějakou hierarchii mají většinou jeden význačný vrchol – v našem příkladě je to pan továrník, obecně se mu říká *kořen* stromu. Jakmile nějaký kořen zvolíme, hned je jasné, kterým směrem je to ve stromu *nahoru* (ke kořeni) a kterým *dolů*. Nenechte se prosím zmást tím, že na rozdíl od biologů matematici kreslívají stromy kořenem nahoru. Podle směru také můžeme sousedy každého vrcholu rozdělit na *otce* (směrem ke kořeni, v našem příkladě nadřízený) a *syny* (směrem od kořene, tedy podřízení). Kořen nemá otce, ostatní vrcholy mají právě jednoho otce. *Listy* jsou ty vrcholy, jež nemají syny.

Libovolný vrchol v spolu se všemi svými potomky (to jsou jeho synové, pak synové synů, atd.) tvoří *podstrom,* jehož kořenem je v.

[2] Sám název *hierarchie* pochází z byzantské řečtiny: *hieros* znamená svatý, z toho *hiereus* je kněz; *arché* značí moc, vládu. Jak to souvisí: Původně se hierarchií nazývaly složité vztahy podřízenosti mezi církevními hodnostáři.

Aritmetický výraz – mějme nějaký výraz s čísly a aritmetickými operacemi, třeba `1-2*(3+4)+5`. Pořadí, v jakém se jednotlivé operace vyhodnocují, můžeme popsat stromem:



Listy odpovídají zadaným číslům, ostatní (vnitřní) vrcholy jednotlivým podvýrazům. V každém vnitřním vrcholu bydlí jedna operace, která dostane mezivýsledky ze svých synů a pošle svůj výsledek otci. V kořeni pak získáme výsledek celého výrazu. Jelikož běžné operace pracují s právě dvěma čísly, půjde o strom *binární* – tedy každý vrchol kromě listů bude mít právě dva syny.

Městečko lakomců – mapu nějakého městečka si můžeme představit jako obecný graf: hrany odpovídají ulicím, vrcholy křižovatkám; slepé konce budeme považovat za speciální případ křižovatek. Pokud ale je pan starosta držgrešle a chce udržovat co nejméně silnic, brzy z městečka zbude jen *kostra,* tedy strom propojující všechny křižovatky. To je příklad stromu, který nemá žádný přirozený kořen. Můžeme ho tedy zakořenit libovolně, třeba na náměstí s radnicí.

Úkol 1 [1b]: Najděte nějaký další pěkný příklad stromu. Čím méně bude podobný těm našim, tím lépe.

Reprezentace stromu

Jak stromy reprezentovat v paměti počítače? Zajisté můžeme použít totéž, co u obecných grafů: vrcholy očíslujeme a pro každý z nich si zapamatujeme seznam čísel sousedů. Jinými slovy, každý vrchol si pamatuje seznam všech hran, které z něj vedou.

Zakořeněné stromy obvykle prohledáváme od kořene dolů. Tehdy si stačí v každém vrcholu pamatovat seznam jeho synů. Někdy se hodí zapamatovat si navíc i otce vrcholu (případně informaci, že jde o kořen).

Snadno odvodíme, že strom o n vrcholech má přesně $n-1$ hran: Zakořeníme ho v libovolném vrcholu a všimneme si, že z každého vrcholu kromě kořene vede právě jedna hrana do otce. Tak jsme každou hranu započítali právě jednou.

Budeme se proto snažit, aby všechny algoritmy pro práci se stromy běžely v čase $\mathcal{O}(n)$, kde n je počet vrcholů. To postačí na projití všech vrcholů a hran, ale už ne na cokoliv náročnějšího.

Prohledávání do hloubky

Chceme-li navštívit všechny vrcholy stromu, můžeme to udělat třeba *prohledáváním do hloubky.* Začneme v kořeni, a kdykoliv přijdeme do nějakého vrcholu, rekurzivně se zavoláme na všechny jeho syny. Přesněji řečeno, nejprve se zavoláme na prvního syna, až se z rekurze vrátíme (prohledali jsme podstrom pod tímto synem), zavoláme se na druhého syna, a tak dále.

Abychom lépe viděli, jak prohledávání do hloubky probíhá, doplníme do něj výpis levé závorky při vstupu do vrcholu a pravé při jeho opuštění.

DFS(v):
1. Vypíšeme `(`.
2. Pro všechny syny s vrcholu v:
3. Zavoláme DFS(s)
4. Vypíšeme `)`.

Spustíme-li algoritmus DFS (tak se mu říká podle anglického názvu *depth-first search*) na náš příklad stromu výrazu, vypíše `(((()(()(()))))())`. Všimněte si, že jsme strom jakoby „obešli po obvodu“ – kdykoliv jdeme po hraně dolů, píšeme levou závorku, kdykoliv nahoru, pravou. Navíc přidáme jeden úplně vnější pár závorek.

Celý průchod stromem trvá $\mathcal{O}(n)$: v každém vrcholu stráníme čas $\mathcal{O}(1+\text{počet synů})$. Pokud to posčítáme přes všechny vrcholy, jedničky se sečtou na n a počty synů na $n-1$.

Během procházení stromu můžeme také zpracovávat hodnoty uložené ve vrcholech, třeba je zase vypisovat. Můžeme je vypsat hned při navštívení vrcholu (tomu se říká *preorder* neboli *prefixový* výpis), nebo až při opuštění (*postorder*, *postfixový* výpis), případně mezi návštěvami synů (*inorder*, *infixový*). Porovnejme, jak to dopadne:

prefixový	`(+(-(1)(*(2)(+(3)(4))))(5))`
postfixový	`(((1)((2)((3)(4)+)*)-)(5)+)`
infixový	`(((1)-((2)*((3)+(4))))+(5))`

Infixový zápis tedy (až na přebytečné závorky) odpovídá tomu, jak obvykle výrazy zapisujeme. Aby se hodnoty v listech neztratily, samozřejmě je vypisujeme přímo, i když mezi žádnými syny neleží.

Postfixový zápis má zase tu hezkou vlastnost, že je jednoznačný i bez závorek. I z očesaného `1 2 3 4 + * - 5 +` můžeme rekonstruovat celý strom. Podobně pro prefixový zápis.

Úkol 2 [2b]: Vymyslete algoritmus, který vytvoří strom z jeho postfixového výpisu bez závorek. Vyzkoušejte si to třeba na výrazech.

Úkol 3 [1b]: Ukažte dva různé binární stromy se stejným infixovým zápisem bez závorek. Jak vypadá jejich prefixový a postfixový zápis?

Výpočty pomocí DFS

Spoustu vlastností stromů můžeme počítat rekurzivně: stačí je umět spočítat pro listy a pak říci, jak z výsledků pro podstromy stanovit výsledek pro celý strom. Takový výpočet jde jednoduše zabudovat do DFS. Začněme triviálním příkladem.

Počet listů – stačí vracet z listů jedničku a ve vnitřních vrcholech hodnoty sčítat:

PočetListů(v):
1. Je-li v list, vrátíme 1.
2. $\ell \leftarrow 0$
3. Pro všechny syny s vrcholu v:
4. $\quad \ell \leftarrow \ell +$ PočetListů(s)
5. Vrátíme ℓ.

Jelikož jsme každý krok prohledávání jenom konstanta-krát zpomalili, algoritmus má opět lineární složitost.

Hloubka stromu – tak se říká délce nejdelší cesty z kořene do listu (délka cesty se měří v hranách). Opět ji spočítáme úpravou DFS: nahlédneme, že hloubka stromu je o jedna více než maximum z hloubek podstromů (nejdelší cesta z kořene do listu musí vést z kořene do některého podstromu a pak pokračovat nejdelší cestou uvnitř podstromu). Hloubka listu je 0. Lineární algoritmus následuje:

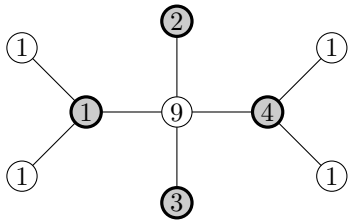
Hloubka(v):
1. Je-li v list, vrátíme 0.
2. $h \leftarrow 0$
3. Pro všechny syny s vrcholu v:
4. $\quad h \leftarrow \max(h,$ HloubkaStromu(s))
5. Vrátíme $h + 1$.

Úkol 4 [4b]: Vymyslete, jak spočítat *průměr* stromu. Tak se říká délce nejdelší cesty. Pozor, není to totéž jako hloubka stromu, protože nejdelší cesta nemusí vést „shora dolů“ (v příkladu s továrnou jedna taková vede mezi skladníkem a některým z dělníků).

Vyhodnocení výrazu – výraz reprezentovaný stromem můžeme snadno vyhodnotit: kdykoliv se vracíme z vrcholu, spočítáme výsledek příslušného podvýrazu. Pro list vrátíme číslo v něm uložené, ve vnitřních vrcholech vezmeme hodnoty ze synů a aplikujeme na ně operaci uloženou ve vrcholu.

Strážníci – ve „stromovém“ městečku bují zločin (lidé kreslí křídou na chodníky karikatury radních!). Starosta chce na vybrané křižovatky rozestavět strážníky tak, aby každá ulice byla z alespoň jedné strany hlídaná. Jak už ale víme, je to skrblík. Pro každou křižovatku proto zjistil, kolik musí zaplatit strážníkovi, aby tam byl ochoten stát, a hledá celkově nejlevnější rozestavění strážníků.

Formálně řečeno, hledáme podmnožinu vrcholů stromu, která z každé hrany obsahuje alespoň jeden vrchol a navíc je součet *cen* vrcholů v množině nejmenší možný.



Nabízí se opět zapřáhnout DFS a vracet z podstromů, jak nejlaciněji je jde ohlídat. Jenže pak nedovedeme z výsledků pro podstromy zkombinovat výsledek pro celý strom. Pomůže, když si pro každý podstrom místo jednoho čísla spočítáme rovnou dvě. Budeme jim říkat h a o. To první („hlídaná cena“) bude udávat, kolik nejméně stačí zaplatit na ohlídání hran podstromu za podmínky, že v kořeni podstromu bude stát strážník. Naopak o („opuštěná cena“) hlásá minimální cenu za podmínky, že kořen je opuštěný.

V listu je triviálně h rovno ceně listu a $o = 0$ (podstrom nemá žádné hrany, takže není potřeba nic hlídat).

Nyní se podívejme na obecný podstrom s kořenem v ceny c_v se syny $s_1, \ldots, s_k$, pro které už známe $h_1, \ldots, h_k$ a $o_1, \ldots, o_k$. Počítejme h: pokud je kořen hlídaný, jeho strážník ohlídá všechny hrany vedoucí do synů. V synech si proto můžeme vybrat, zda tam umístíme strážníka či nikoliv, takže pro i-tý podstrom postačí $\min(h_i, o_i)$ peněz. Celkově tedy $h = c_v + \sum_{i=1}^{k} \min(h_i, o_i)$.

A nyní o: Jestliže do kořene strážníka nedáme, musí být strážníci ve všech synech (jinak by některá z hran do synů nebyla hlídaná). Proto $o = \sum_{i=1}^{k} h_i$.

Všechna h a o tedy hravě spočítáme pomocí DFS v čase $\mathcal{O}(n)$. Odpověď na starostovu otázku pak získáme jako minimum z h a o kořene.

Strážníci(v):
1. $c_v \leftarrow$ cena vrcholu v
2. Je-li v list, vrátíme $(c_v, 0)$.
3. $h \leftarrow c_v$, $o \leftarrow 0$
4. Pro všechny syny s vrcholu v:
5. $\quad (h_s, o_s) \leftarrow$ Strážníci(s)
6. $\quad h \leftarrow h + \min(h_s, o_s)$
7. $\quad o \leftarrow o + h_s$
8. Vrátíme (h, o).

Úkol 5 [4b]: Strážníci si pořídili drony a dokáží hlídat i „za jeden roh“. Přesněji řečeno, ulice je hlídaná, pokud stojí strážník na alespoň jedné z krajních křižovatek, nebo na křižovatce, která s krajní sousedí. Pomozte najít nejlevnější rozestavění strážníků pro ohlídání všech ulic.

Vandalská indukce a centrum stromu

Ukážeme si ještě jeden způsob, jak zkoumat stromy. Tentokrát je budeme procházet od listů směrem „dovnitř“. Začneme dvěma jednoduchými pozorováními, přičemž *netriviální* budeme říkat stromům s alespoň dvěma vrcholy.

Pozorování 1: Každý netriviální strom má nějaký list.

Důkaz: Strom si zakořeníme v libovolném vrcholu a podíváme se na nejhlubší vrchol (nejvzdálenější od kořene). Ten nemá žádné syny a vede z něj jediná hrana do otce. Proto je to list.

Pozorování 2: Odstraníme-li z netriviálního stromu list, vznikne opět strom.

Důkaz: Je třeba ověřit, že graf zůstal souvislý a bez kružnic. Odstraněním vrcholu jsme sotva mohli vytvořit novou kružnici. Pokud byly nějaké dva neodebrané vrcholy spojené cestou, budou spojené i nadále, protože odebraný list nemůže ležet uvnitř cesty.

Z toho plyne následující, poněkud vandalská, technika indukce: ve stromu najdeme list, odtrhneme ho, čímž získáme další strom. Postup opakujeme, dokud nám nezůstane triviální strom. Jestliže zaznamenanou posloupnost odebírání listů obrátíme, dostali jsme postup, jak z jednoho vrcholu postupným přilepováním listů vytvořit původní strom.

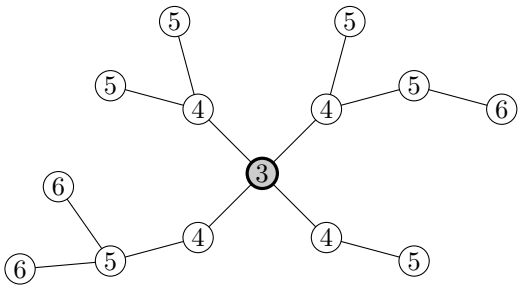
Pokud má ovšem celý algoritmus seběhnout v lineárním čase, nemůžeme listy hledat pokaždé znovu. Budeme si udržovat frontu objevených, ale dosud neutržených listů. Na začátku spočítáme stupně všech vrcholů a listy přidáme do fronty. V každém dalším kroku odebereme jeden list z fronty a odstraníme ho ze stromu.

Jeho sousedovi snížíme stupeň o jedničku, a pokud se tento soused stal listem, přidáme ho do fronty. Každý vrchol se dostane do fronty právě jednou a jeho obsluhou strávíme konstantní čas. Celé otrhávání tedy beží v čase $\mathcal{O}(n)$, jak jsme chtěli.

Můžete si zkusit vymyslet, jak tuto „vandalskou indukci“ použít na libovolný z příkladů, které jsme řešili pomocí DFS. Není divu – když se DFS vrací z rekurze, také postupuje od listů ke kořeni. My zde ukážeme, jak najít „střed“ stromu, což se přímo pomocí DFS dělá obtížně.

Definice: Pro libovolný vrchol v zavedeme jeho *excentricitu* (výstřednost) jako maximum ze vzdáleností z v do ostatních vrcholů. (Pokud bychom tedy strom ve v zakořenili, bude nám to říkat, jak hluboko je nejhlubší list.) *Centrum* stromu říkáme množině všech vrcholů s nejmenší možnou excentricitou.

Příklad vidíte na následujícím obrázku. Čísla udávají excentricity, zvýrazněný vrchol je centrem stromu.



Dokážeme, že centrum každého stromu je tvořeno buďto jedním vrcholem, anebo dvěma vrcholy spojenými hranou. Navíc ho lze najít v lineárním čase.

Nejprve si všimneme, že centrum stromu neobsahuje žádný list, neboť soused listu má o jedničku nižší excentricitu. Aby tato úvaha fungovala, nesmí být soused listu také list, takže strom musí mít aspoň 3 vrcholy.

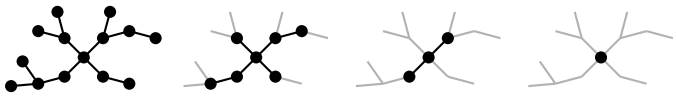
Nyní se nám bude hodit, že odstraněním všech listů se sníží excentricity všech ostatních vrcholů právě o 1. Každá nejdelší cesta z vnitřního vrcholu totiž končila v nějakém listu, tím pádem jsme ji zkrátili o jednu hranu.

Zkombinujeme-li tyto dvě vlastnosti, zjistíme, že „očesáním“ všech listů dostaneme menší strom, který má stejné centrum. Opakováním tohoto postupu graf „oloupeme“ až na jedno- nebo dvouvrcholový strom. U těch snadno nahlédneme, že jejich centrum obsahuje všechny vrcholy.

K nalézání listů nám opět poslouží fronta. Jen musíme umět rozlišit, kde končí listy z jedné „slupky“ a začínají ty z další. K tomu stačí do fronty přidávat zarážku za konec slupky, případně střídat dvě fronty. V obou případech to stihneme v lineárním čase. Může to vypadat třeba následovně:

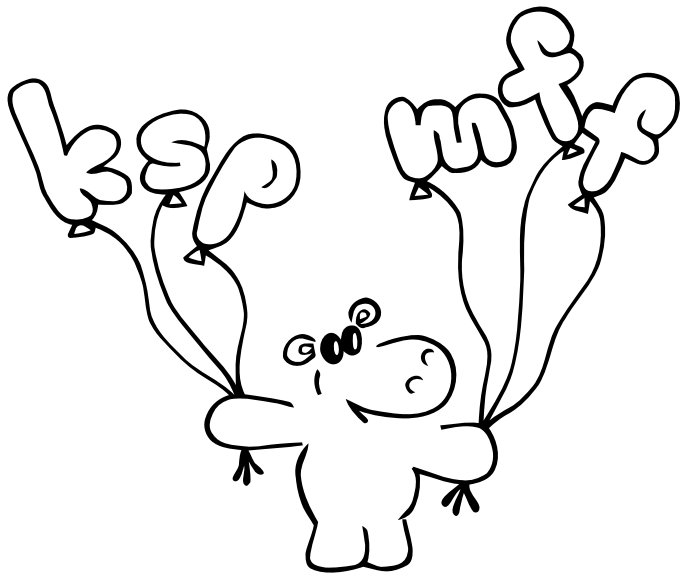
1. Založíme dvě prázdné fronty F a G.
2. Všem vrcholům spočítáme stupně.
3. Listy přidáme do F.
4. Dokud strom obsahuje alespoň 3 vrcholy:
5. Dokud je F neprázdná:
6. $v \leftarrow$ další vrchol z F
7. Odebereme v.
8. Pro všechny sousedy s vrcholu v:
9. Snížíme stupeň s o 1.
10. Pokud stupeň klesl na 1, přidáme s do G.
11. Prohodíme F a G.
12. Zbývající vrcholy prohlásíme za centrum.

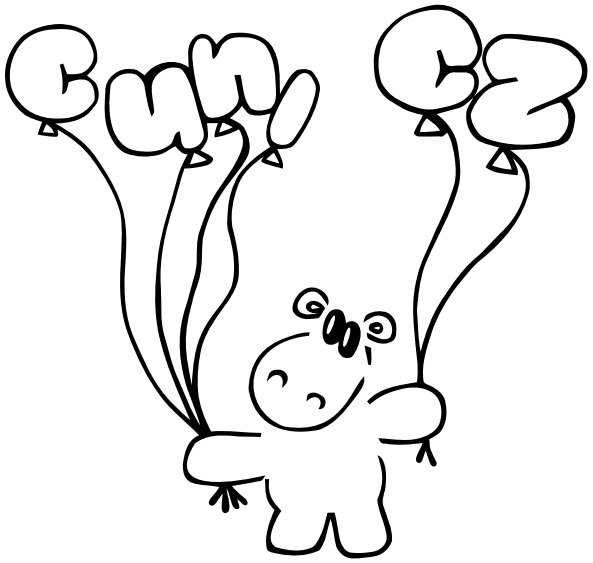
Pro strom z předchozího obrázku proběhne výpočet takto:



Úkol 6 [3b]: Navrhněte a naprogramujte algoritmus, který v daném stromu spočítá excentricity všech vrcholů.

Martin „Medvěd“ Mareš





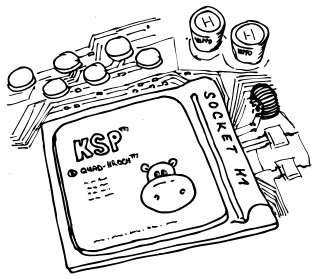
Recepty z programátorské kuchařky: Základní algoritmy

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.[3] 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“.[4]

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 určité 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.

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

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

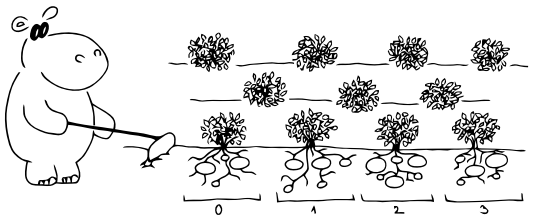
Pole

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], ...`).[5]

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,[6] nejdříve však doporučujeme dočíst tuto kuchařku.

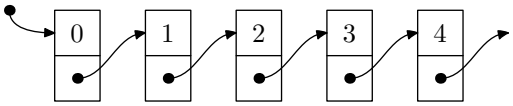
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu 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).

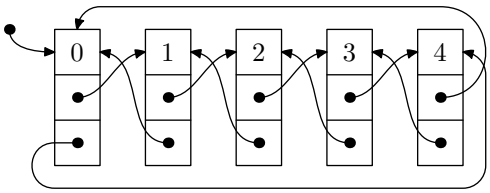


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti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

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

[6] `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.

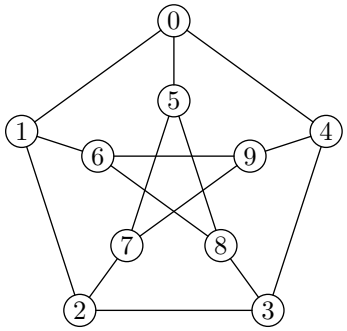
Stromy a grafy v informatice

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.

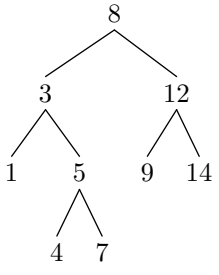
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[7]

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*.



[7] `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 $\mathcal{O}(\log n)$.[8]

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.

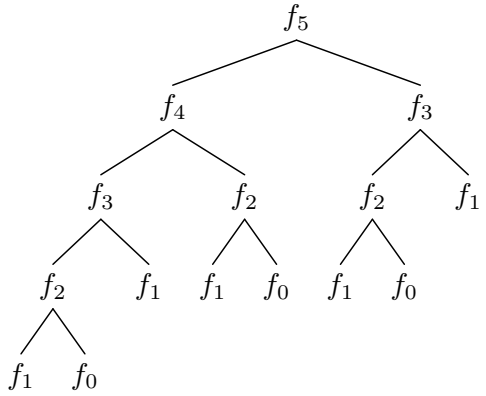
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[9]

Předpočítané mezivýsledky

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

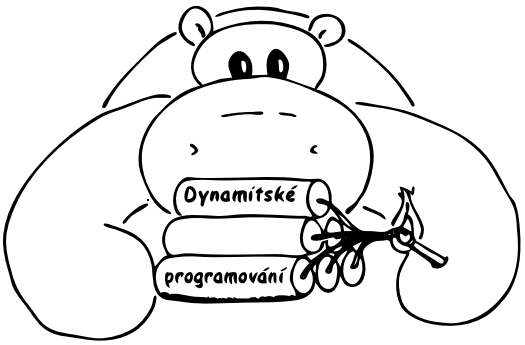
Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme vzít naši rekurzivní implementaci počítání Fibonacciho čísel z kapitolky Rekurze.

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.

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

[9] `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.[10]

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];
```

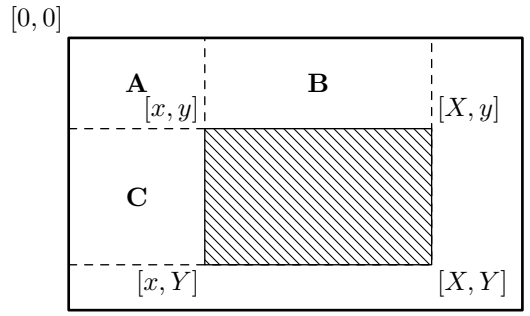
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas, prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

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.

[10] 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

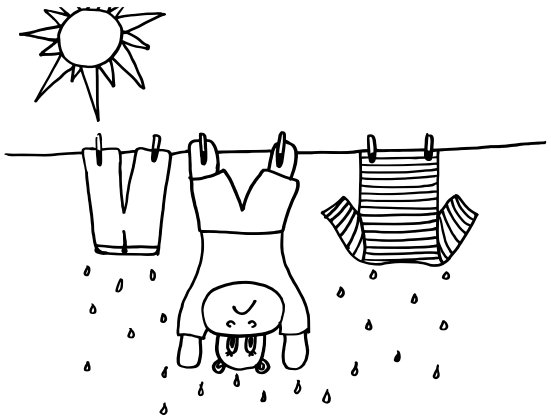
Doufám, že jste si z tohoto rozsáhlého textu odnesli nějaké nové znalosti a poznatky, které vám pomohou nejen v řešení KSP.

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



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`.