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 mají smůlu, 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 26. 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, co napsat do řešení. Co s tím?

Nalistuj si konec letáku, kde jsme pro Tebe přichystali stručný návod na řešení úloh. A pokud tápeš hlavně v programátorských technikách, tak těsně před tímto návodem je naše kuchařka – učební textík – vysvětlující základní programátorské principy a postupy.

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), druhak 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.

Této úloze říkáme *praktická*, jelikož není potřeba popsat algoritmus, jen ho naprogramovat a odevzdat přes Internet. Bližší informace nalezneš přímo v jejím zadání.

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.

Protože chápeme, že k „uvaření“ řešení jsou často potřeba znalosti základních algoritmů a datových struktur, obvykle též přikládáme do každé série tzv. *kuchařku*, ze které se můžeš takové věci naučit. Často je také v zadání úloha, již lze řešit algoritmem z kuchařky. A pozor – další kuchařky najdeš na našich webových stránkách.

Kdo seminář vůbec připravuje?

Studenti Matematicko-fyzikální fakulty Univerzity Karlovy (MFF UK), většinou bývalí řešitelé.

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 26. 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!

Milí řešitelé a řešitelky!

Držíte v ruce první leták 26. ročníku KSP. I letos bude každá série obsahovat 7–8 úloh, z nich alespoň dvě lehčí, vhodné i 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 sérií doháněli 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 tří nejvíc bodovaných úloh první série na plný počet bodů, pošleme čokoládu.

Termín odevzdání první série je stanoven na pondělí 21. října v 8:00 SELČ.

Řešení přijímáme elektronicky na stránce *https://ksp.mff.cuni.cz/submit/*. Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – zde je jeho SHA1 fingerprint: `0E:D9:B6:E5:6F:B0:51:D9:66:EB:E9:29:E4:58:AB:5F:99:D6:FD:A3`.

Před tím ale vyplňte přihlášku na *http://ksp.mff.cuni.cz/* (a to i tehdy, když jste se KSPčka účastnili loni). Na tomtéž místě najdete i další informace o tom, jak KSP funguje. Na webu máme rovněž fórum, kde se můžete na cokoli zeptat. Také nám můžete napsat na e-mail *ksp@mff.cuni.cz*.

První série dvacátého šestého ročníku KSP

Prázdnotu vesmíru prořízl oslepující záblesk. Kde před chvílí byl jenom volný prostor, nalézaly se teď megatuny hmoty hvězdné lodě. Z trupu se vysunuly anténní systémy a senzory začaly prozkoumávat okolní prostor.

Mohlo by se to jevit jako standardní skok z hyperprostoru do normálního vesmíru, nebýt dlouhého roztřepeného šrámu táhnoucího se skoro přes celý levobok. Těsně nad šrámem se nalézaly ještě stěží rozeznatelné insignie hlásající, že se jedná o UFC Freya*, těžkou nákladní loď Spojené federace.*

V tom se objevil další záblesk. Některé z hlavních motorů lodě se spustily a začaly do prostoru za lodí chrlit gejzíry světla z nukleární fúze tak jasné, že by se do nich nechráněné lidské oko nemohlo ani podívat. Poškození lodě bylo příliš rozsáhlé, Freya *umírala...*

Bylo skoro zázrakem, že se lodi podařilo přežít cestu hyperprostorem v takovémto stavu. Zdaleka však neměla vyhráno. Hlavní počítač stále zápasil se selhávajícími systémy a soupeřil o kontrolu nad lodí s poškozenými obvody vysílajícími do lodních rozvodů nesmyslné příkazy.

26-1-1 Blokující signály — 8 bodů

Hlavní počítač poškozené hvězdné lodě se pokouší obnovit kontrolu nad většinou důležitých systémů. Bohužel ale poškozené obvody, dříve než je hlavní počítač zvládl izolovat, vyslaly do lodní počítačové sítě několik vadných signálů, které je potřeba zastavit, než se dostanou do svého cíle.

Lodní počítačová síť je představována N propojenými počítačovými uzly, mezi kterými je M přímých spojení (kabel spojující nějaké dva uzly). Síť tedy vlastně představuje neorientovaný graf.

Každý vyslaný vadný signál je zadaný posloupností uzlů sítě (cestou) a svým cílovým uzlem. Každou časovou jednotku se posune o jeden uzel na své cestě dál.

Zastavení signálu probíhá tak, že ve vhodný okamžik vyšleme z hlavního počítače (jeden určený uzel sítě) vhodný blokující signál. Ten cestuje stejnou rychlostí jako vadný signál, ale může jinou cestou. Když se signály potkají (dorazí ve stejný čas do stejného uzlu), tak blokující signál zabrání dalšímu šíření vadného signálu.

Ptáme se, kolik signálů dokážeme zastavit ještě před tím, než dosáhnou svých cílových vrcholů.

Motory postupně ustálily svůj chod a Freya *se stabilizovala. Stav však zůstával kritický, poškozený hlavní reaktor, který byl nyní pod šrámem zčásti odhalený, stále hrozil výbuchem.*

Freya *byla transportní loď postavená ještě za války, měla tedy sice zastaralou, ale snad stále platnou mapu hvězdných soustav. Soustava, do které s vypětím všech sil dolétla, byla sice daleko od běžných letových tras, ale obsahovala planetu schopnou udržet lidský život.*

Pátrací senzory malou planetu konečně našly, loď mírně změnila svůj kurz a začala se připravovat na nouzové přistání. Původně měla celý svůj život zůstat v mezihvězdné prázdnotě a k planetě se přiblížit nejblíže na dosah raketoplánu, ale poctiví programátoři a konstruktéři ji před lety připravili i na tuto eventualitu. Devatenáct lidí v kryogenickém spánku stále nic netušilo...

26-1-2 Přeskládání nákladu — 9 bodů

Hvězdná loď potřebuje připravit svůj náklad na nouzové přistání. Přípravy spočívají mimo jiné v tom, že se musí rozdělit, který náklad bude ve skladišti na pravoboku a který na levoboku.

Náklad je tvořen celkem N kontejnery. Z důvodu bezpečnosti a rovnoměrného rozložení zásob (aby jich bylo dost i v případě ztráty jednoho skladiště) existuje také M doporučení. Každé doporučení říká, že nějaká dvojice kontejnerů by neměla být v tom stejném skladišti.

Všechny předpisy najednou pravděpodobně splnit nepůjde, ale hlavní počítač chce nalézt rozdělení kontejnerů do skladišť (množství kontejnerů v jednotlivých skladištích nemusí být stejné) takové, aby byla splněna alespoň polovina doporučení.

Dopravníky uvnitř lodi dokončily přesuny kontejnerů, loď přečerpala zásoby paliva tak, aby kompenzovala vyvážení, a byly uzavřeny všechny vzduchotěsné dveře.

Hlavní motory už nějakou chvíli nepracovaly, jejich další chvíle měla přijít v konečné fázi sestupu. Freya *pomalu klesala do atmosféry planety. Jak se začala třít o horní vrstvy atmosféry, tak její příď postupně změnila barvu přes temně rudou až do jasné oranžové. Okolo trupu začaly šlehat plameny a trupové nástavby začaly odletovat jedna za druhou. Během chvíle zmizely pátrací radary, jeřábové manipulátory i zbytky poškozených komunikačních antén.*

V přesně vypočtenou chvíli naběhly přední manévrovací motory. Nebyly stavěné jako brzdící, ale jejich předimenzovaná velikost a spuštění na kritický výkon by mohly stačit, pokud ten nápor vydrží.

Loď zpomalovala. V tom ale její pravobok pohltil oslňující výbuch. Zdejší vrchní vrstva atmosféry obsahovala nějaké kapsy výbušných plynů. Tentokrát to odneslo jen skladiště na pravoboku, ale další takový výbuch by mohl loď rozpůlit.

Hlavní počítač vysunul jeden z posledních fungujích radarů. Ve zlomku sekundy, než ho proud vzduchu vyrval z trupu, stihl zaznamenat rozložení kapes plynů v atmosféře.

26-1-3 Plynové kapsy — 8 bodů

Snímkovací radar dodal průřez atmosférou obsahující dva druhy plynů. Průřez je znázorněn v podobě posloupnosti znaků a a b (dva druhy plynů) délky N.

Pro loď širokou dva znaky je bezpečné proletět skrz souvislou oblast buď jednoho, nebo druhého plynu. Na rozhraní plynových kapes je to moc nebezpečné (tedy může proletět oblastí aa, bb, ale nemůže proletět místem, kde se setkávají – ab nebo ba).

Hlavní počítač potřebuje vytvořit datovou strukturu, které by se mohl rychle ptát, kolik míst k průletu je v zadaném intervalu. Tedy kolik v něm je stejných sousedních polí.

Počítejte s tím, že dotazů bude řádově N a že počítáme i ta místa, která se vzájemně překrývají.

Příklad: Výstup z radaru a odpovědi na dotazy na intervaly (indexujeme od nuly):

```
Radar:  aababbaaa
[0,7]   -> 3 místa
[0,8]   -> 4 místa
[2,4]   -> 0 míst
```

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.

Po dalším výbuchu vzplál jeden z brzdících motorů, a proto ho nouzový systém i s okolní sekcí odhodil. Explodoval v obrovskou kouli ohně těsně za lodí. To se však už Freya *dostala do spodních vrstev atmosféry.*

Kdyby se hlavní počítač mohl divit, asi by se teď divil. Ale nic takového neměl ve svém programu, a tak jen provedl rychlý výpočet, aby se nějak vypořádal s údaji o mnohem vyšší rychlosti, než byla původně plánovaná.

Hlavní motory podruhé naběhly, tentokrát s tryskami obrácenými na reverzní tah. Počítač vyřadil všechny pojistky a spustil je na výkon, na který ještě nikdy neběžely. Za lodí zůstával pruh doslova spálené atmosféry.

Loď brzdila s takovým přetížením, že začaly povolovat vnitřní přepážky. Jedno z kryogenních oddělení, společně s celým levobočním hangárem, vylétlo z lodi a zaniklo v záři atomového ohně motorů. Odletovaly kusy trupu a konstrukce se bortila.

Byla to dobře postavená loď, ještě podle válečných specifikací. Dnešní lodě by se už dávno rozpadly, Freya *však držela dál. Pak ale přišla i její chvíle. Nejdříve se v půlce trupu zkroutila a zanedlouho se celá zadní polovina třísetmetrové lodě utrhla společně se všemi hlavními motory.*

Svou práci však hlavní motory vykonaly, zbrzdily sestup. Zatím stále funkční přední část pokračovala v řízeném pádu korigovaném manévrovacími motory. Pak dopadla. Odrazila se a dopadla podruhé. Vyryla v místním ekvivalentu deštného pralesa brázdu dlouhou skoro tři kilometry a pak se konečně zastavila. . .

⋆⋆⋆

Jacob otevřel oči. Nad ním se nacházel otevřený poklop jeho kryogenní kóje. Otřepal z rukou poslední zbytky kryogenního gelu a protřel si oči.

„Tak počkat, tady něco nehraje!“ pronesl pomalu při pohledu na rozbitou místnost, do níž odněkud prosvítalo podivné žluté světlo. Vyhoupl se z kóje a přistál bosýma nohama na nakloněné podlaze.

Druhá strana místnosti, na které se nacházely zbylé kóje, byla celá zavalená. Pomocí kusu nosníku se mu povedlo roztáhnout dveře a skrz trosky se prodral na chodbu a k nejbližšímu počítačovému uzlu, aby zjistil, co se stalo.

26-1-4 Oprava databáze — 10 bodů

Databáze hlavního počítače je silně poškozena a samoopravný mechanismus je vyřazen. Musíme ji proto opravit ručně. Jak ale poznat, že jsme ji sestavili správně? Jistá naděje tu je: víme, že databáze měla podobu posloupnosti celých čísel, a počítač si pamatuje, kolik v ní bylo *trojných prvků.*

Trojný říkáme takovému prvku, který se dá poskládat jako součet nějakých tří předchozích prvků v posloupnosti. (Dodejme ještě, že jeden prvek nemůžeme v součtu použít vícekrát, ale dva prvky téže hodnoty už ano.)

Příklad: Pro posloupnost $1, 4, 7, 2, 7, 16, 10$ jsou trojnými prvky druhá sedmička ($7 = 1 + 4 + 2$), šestnáctka ($16 = 7 + 2 + 7$) a desítka ($10 = 1 + 2 + 7$), jiné trojné nejsou. První sedmička není na rozdíl od druhé trojná, protože u ní ještě nemůžeme použít číslo 2.

Lehčí varianta (za 6 bodů): Vyřešte úlohu pro případ, kdy hledáme dvojné prvky namísto trojných (tedy skládáme jen ze dvou předchozích prvků v posloupnosti).

Když Jacob zjistil, co se stalo, chvíli stál naprosto bez pohnutí. Potom udeřil pěstí do přepážky.

„Zatraceně, tak jsem jediný přeživší na téhle planetě, která dokonce ani nemá jméno!“

Když se trochu vzpamatoval, začal postupovat směrem k bývalému můstku. Cestou se zastavil ve výstrojním skladu a z hromad popadaných beden vylovil nějaké základní věci jako baterku, nůž, lékárničku a nějaký batoh. Taky se převlékl do odolnější kombinézy a vzal si boty.

Po dalších několika minutách zjistil, že můstek lodě již neexistuje. Místo toho se mu však naskytl pohled na okolní prales. Stromy nevypadaly zase tolik jinak než ty pozem-

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

ské. Měly trochu jinou barvu a byly hlavně mnohem vyšší. Převyšovaly o pár metrů dokonce i zabořené torzo lodě.

Věnoval ještě asi hodinu průzkumu, během něhož obešel většinu lodě. Nakonec vylezl po již dávno vychladlém trupu na nejvyšší místo a posadil se pozoruje dlouhou brázdu od pádu lodě.

„Prales... To znamená, že tu asi bude nějaký mimozemský život. A nemusel by být úplně přátelský," zamyslel se Jacob nahlas. Už se také stmívalo a z lesa se začínaly ozývat podivné zvuky. Chtělo by to asi okolí nějak zajistit. Shodou okolností Freya *zrovna převážela senzorové soupravy, které by mohl použít.*

Vypravil se tedy do trosek skladu a rychle vytáhl několik funkčních senzorových věží a pár desítek metrů kabelů.

26-1-5 Senzory — 10 bodů

Chceme rozestavit K senzorových věží na čtvercovou síť o rozměrech $M \times N$. Pro správnou funkci senzorů je potřeba, aby žádné dvě senzorové věže nestály ve stejném sloupci nebo na stejném řádku.

Každá senzorová věž má navíc určený obdélník, ve kterém musí být postavena. Najděte pro dané zadání nějaké fungující rozestavení věží nebo určete, že takové rozestavení neexistuje.

Lehčí varianta (za 6 bodů): Vyřešte úlohu pro případ, že síť má tvar jednořádkové „nudle" (tedy $M = 1$) a dvě věže nesmí stát v témže sloupci.

Po zapojení poslední věže do systému se Jacob uložil ke spánku v bývalé jídelně.

Druhý den na planetě již probíhal trošku poklidněji. Po dalším průzkumu lodě zjistil, že má celkem dostatečné zásoby pitné vody a že nouzové palivové články mu budou schopny dodávat energii ještě přinejmenším rok, pokud se mu do té doby nepovede znovu nahodit záložní reaktor.

Mnohem větší problém byl ale s jídlem. Hlavní sklad jídla se totiž nacházel v zadní části lodě a vepředu byla jen hydroponická zahrada, která navíc při přistání dost utrpěla. Jelikož nechtěl zatím zkoušet ochutnávat místní jídlo, rozhodl se Jacob hydroponiku opravit.

26-1-6 Hydroponie — 11 bodů

Hydroponická zahrada je tvořena soustavou N očíslovaných přihrádek na rostliny, v každé může být maximálně jedna rostlina (ale také tam nemusí být žádná). Mimo to je zde M napájecích okruhů, každý napojený na určitý interval přihrádek.

V každém napájecím okruhu chceme mít umístěnou dohromady minimálně jednu rostlinu, jinak nám na rozmístění a počtu rostlin nezáleží.

Ptáme se na počet možných způsobů, jak můžeme obsadit přihrádky rostlinami tak, aby byly splněny výše zmíněné podmínky. Jelikož počet může být obrovský, spočítejte ho modulo 1 000 000 007.

Jacob zrovna přemístil poslední rostlinku, když v tom se ozval z chodby poplach. To počítač napojený na senzory nahlásil, že něco velkého porušilo perimetr. Jacob popadl velký nůž a vyběhl ven.

Doběhl na místo, odkud senzory hlásily narušitele. Na zemi tam byly v bahně obtisknuté nějaké stopy a... V Jacobovi hrklo, vedle stop ležel umně vyrobený a nazdobený malý oštěp. To znamená, že je tady inteligentní život! To musí prozkoumat!

Natěšený doběhl nazpět do torza lodi, během pěti minut si pobral zásoby jídla a vody na několik dní, sbalil si lékárničku, kameru a další potřebné věci a vydal se opět na místo, kde nalezl ten oštěp.

Vydal se opatrně po stopách směrem do lesa. Po několika metrech přišel na malý palouk, kde ho zaujal místní podivný hmyz. Podobal se trochu pozemským mravencům, jen byl mnohem větší. Zvedl kus klacku s několika těmito tvory a chvíli je pozoroval.

26-1-7 Mravenci — 8 bodů

Tvorové podobní mravencům lezou po rovném kusu klacku dlouhém D centimetrů. Na začátku se jich zde nachází N a každý tvor se pohybuje buď doprava, nebo doleva, a to rychlostí jednoho centimetru za sekundu.

Když tvor přijde na konec klacku, tak spadne. Když se dva tvorové potkají, tak se oba otočí čelem vzad. Nás zajímají dvě věci:

a) (za 3 body) Za kolik sekund z klacku spadne poslední tvor?

b) (za 5 bodů) Na jaké pozici tento tvor na začátku stál?

Tvory v této úloze považujte za zanedbatelně malé vzhledem k velikosti klacku.

Od pozorování tvorů na klacku náhle Jacoba vyrušilo zapraskání za jeho zády. Rychle se otočil a...

Pokračování příště...

Úvodem dobrodružství na neznámé planetě vás provedl

Jirka Setnička

26-1-8 Turingova strojovna — 12 bodů

Při řešení KSP po vás obvykle chceme, abyste na daný problém sestrojili *algoritmus*. Přemýšleli jste někdy nad tím, co to takový algoritmus přesně je? Intuitivně je to jasné: nějaký zápis postupu, jak něco spočítat, poskládaný z dostatečně jednoduchých kroků. To se ale sotva dá pokládat za pořádnou definici.

Tak jinak: algoritmus je totéž co program v našem oblíbeném programovacím jazyce, jen zbavený přízemních detailů (třeba omezené velikosti datových typů). Je to lepší? O moc ne – sice už je jasné, co to znamená, ale zase se nám definice rozrostla o specifikaci celého programovacího jazyka (umíte popsat Céčko nebo Python jedním odstavcem? stránkou? knížkou?). A navíc ani není jasné, jestli pro nás algoritmus znamená totéž jako pro dědečka Pascalistu nebo pro pradědečka, který ještě programy děroval ve strojovém kódu.

Dobrá, jaká je tedy správná definice? Teoretičtí informatikové obvykle postupují tak, že si zavedou nějaký *výpočetní model*. Tím se myslí jednoduchý matematický stroj s přesně určenými operacemi, řízený programem. Za algoritmus pak prohlásíme program pro tento stroj. Na první pohled to vypadá, že jsme se z deště přesunuli pod okap, ale přeci jen tu prší méně: Výpočetní modely jsou daleko jednodušší zvířátka než běžné programovací jazyky (však za chvíli uvidíme). Navíc není těžké o různých výpočetních modelech dokázat, že dovedou spočítat totéž, takže nezáleží na tom, který z nich jsme pro zavedení algoritmu použili.

V letošním seriálu se spolu vydáme na procházku po zoo výpočetních modelů. Výběhů tu je habaděj, my se zastavíme u pěti z nich a pokaždé si v daném modelu zkusíme

něco naprogramovat a třeba i dokázat pár obecných větiček o jeho vlastnostech. Mezitím můžete sami rozmýšlet, jak do každého modelu překládat programy z vašeho oblíbeného jazyka. Pěknou cestu!

Turingovy stroje

Nejklasičtějším a nejspíš i nejstarším modelem počítače je bezpochyby Turingův stroj. Alan Turing ho popsal v roce 1936 ve své práci, kterou položil základy matematického zkoumání počítačů.

TS je vybaven *oboustranně nekonečnou páskou* rozdělenou na *políčka.* Na každém políčku je zapsán jeden *znak* ze zvolené *abecedy*. Tou se myslí nějaká množina znaků, o níž víme jen to, že je konečná a že obsahuje mezeru (tu značíme $\sqcup$).

Nad páskou se pohybuje *hlava.* Vždy se nachází nad jedním políčkem a umí z něj přečíst znak a případně ho přepsat na jiný.

Činnost stroje ovládá *řídicí jednotka* podle *programu.* Řídicí jednotka se v každém okamžiku nachází v jednom z konečně mnoha *stavů.* V každém kroku výpočtu se podívá, v jakém stavu S se nachází a jaký znak z vidí hlava, a podle toho vybere jednu *instrukci* programu. Ta stroji říká, že má znak z přepsat na z', posunout hlavu o políčko v daném směru a nakonec se přepnout do stavu S'.

Program tedy můžeme popsat tabulkou, jejíž řádky odpovídají možným stavům S, sloupce znakům z a v každé buňce tabulky je uložena jedna instrukce v podobě trojice (z', p, S'). Instrukce říká, co má stroj ve stavu S, který přečetl znak z, udělat. Tedy zapsat znak z', posunout se ve směru $p \in \{\leftarrow, \rightarrow, \bullet\}$ (o políčko doleva či doprava, případně zůstat na místě) a přejít do stavu S'.

Nyní definujeme *výpočet* stroje. Na počátku výpočtu je na pásce zapsán *vstup,* hlava stroje stojí na začátku vstupu a zbytek pásky je vyplněn mezerami. Řídicí jednotka se nachází ve zvoleném *počátečním stavu* S_0. Výpočet probíhá v *krocích* (taktech) – v jednom kroku stroj provede jednu instrukci programu (přečte znak, rozhodne se podle tabulky, zapíše znak, posune hlavu, změní stav). Tak pokračuje až do doby, kdy se dostane do některého z *koncových stavů*.

Koncové stavy budeme mít dva a budeme jim říkat ANO a NE, čímž umožníme stroji, aby výpočet ukončil úspěšně nebo neúspěšně. Pokud chceme, aby výsledkem výpočtu bylo něco víc než jediný bit, dohodneme se, že výstup bude opět napsán na pásce, obklopen mezerami.

Dodejme ještě, že vždy hledáme jeden TS, který danou úlohu vyřeší pro všechny vstupy – počet stavů, velikost abecedy nebo obsah programu nesmí záviset na vstupu. Pak můžeme snadno definovat časovou a prostorovou složitost.

Čas budeme měřit počtem provedených instrukcí, *prostor* počtem políček pásky, jež během výpočtu navštívila hlava stroje. Nezapomeňme, že mohou existovat i výpočty, které se nikdy nezastaví; ty pak mají nekonečnou časovou složitost a možná i nekonečnou prostorovou.

Příklad

Suchá formální definice bude stravitelnější, když ji doplníme příkladem. Sestrojíme TS, který dostane řetězec složený ze znaků `+` a `-`, vždy se zastaví a odpoví ANO právě tehdy, když je *vyvážený*. Tím myslíme, že obsahuje stejně plusek jako minusek.

Stroj bude na pásce opakovaně hledat dvojice znaků `+` a `-` (ne nutně vedle sebe) a oba znaky přepisovat na `*`. Vyvážený vstup tedy nutně předělá na samé hvězdičky. Jestliže vstup vyvážený není, zbude nakonec jedno `+`, ke kterému už neexistuje `-`, či opačně.

Za abecedu stroje zvolíme množinu $\{\sqcup, +, -, *\}$, řídicí jednotka bude vybavena stavy $\{S_0, P, M, R, \text{ANO}, \text{NE}\}$ a následujícím programem:

stav/znak	$\sqcup$	`+`	`-`	`*`
S_0	$(\sqcup, \bullet, \text{ANO})$	$(*, \rightarrow, P)$	$(*, \rightarrow, M)$	$(*, \rightarrow, S_0)$
P	$(\sqcup, \bullet, \text{NE})$	$(+, \rightarrow, P)$	$(*, \leftarrow, R)$	$(*, \rightarrow, P)$
M	$(\sqcup, \bullet, \text{NE})$	$(*, \leftarrow, R)$	$(-, \rightarrow, M)$	$(*, \rightarrow, M)$
R	$(\sqcup, \rightarrow, S_0)$	$(+, \leftarrow, R)$	$(-, \leftarrow, R)$	$(*, \leftarrow, R)$

Ve stavu S_0 hledáme první znak různý od `*`. Pokud je to `+`, přejdeme do stavu P, v němž hledáme `-` do páru. Podobně stav M odpovídá tomu, že jsme našli `-` a hledáme párové `+`. Po nalezení páru pokračujeme stavem R, který hlavu stroje vrátí zpět na začátek vstupu a pak přejde na hledání dalšího páru, tedy do stavu S_0.

Časová složitost tohoto stroje pro vstup délky n činí $\mathcal{O}(n^2)$, neboť na vstupu se vyskytuje až n párů znamének a každý z nich můžeme hledat až $\mathcal{O}(n)$ kroků. Paměti spotřebuje $\mathcal{O}(n)$ políček.

Úkol 1 [3b]: Navrhněte Turingův stroj, který dostane posloupnost závorek `(` a `)` a odpoví ANO nebo NE podle toho, zda je vstup správně uzávorkovaný. Tím myslíme, že závorky jsou správně spárované a páry se nekříží. Například na vstupy `()()` a `(()())` odpoví ANO a na `)()(` odpoví NE.

Součástí řešení úkolu by měl být kompletní popis stroje: abeceda, množina stavů, program. Sluší se též spočítat, jakou má stroj časovou a paměťovou složitost.

Vícepáskové stroje

Možná vás překvapilo, že stroj z předchozího příkladu potřeboval na tak obyčejnou věc, jako rozpoznání vyváženosti, kvadratický čas. Zčásti to bylo způsobené naší nešikovností (zkuste sestrojit jiný stroj, který tutéž úlohu zvládne rychleji), zčásti tím, že musíme neustále přejíždět hlavou mezi různými místy, která nás zajímají současně.

Často se proto uvažuje vícepásková varianta Turingova stroje. Ta má libovolný počet pásek (budeme ho značit p) a na každé pásce samostatnou hlavu. Řídicí jednotka se pak rozhoduje podle kombinace symbolů viděných jednotlivými hlavami. Program proto není 2-rozměrná tabulka, nýbrž $(p+1)$-rozměrná (jeden rozměr odpovídá aktuálnímu stavu stroje, ostatní přečteným znakům). Instrukce programu pak říkají každé hlavě, jaký symbol má na svou pásku zapsat a kterým směrem se má posunout; různé hlavy se mohou pohybovat různě, nebo zůstat stát.

U vícepáskových TS bývá zvykem, že jedna z pásek je vyhrazena pro vstup a jiná pro výstup. Na vstupní pásce je na počátku vstup, stejně jako u jednopáskového TS. Ostatní pásky ze začátku obsahují samé mezery. V průběhu výpočtu smí program ze vstupní pásky pouze číst a na výstupní pouze zapisovat; ostatní pásky (těm se říká *pracovní*) může využívat libovolně. Do prostorové složitosti se pak počítají pouze políčka na pracovních páskách.

Příklad podruhé

Předveďme si, jak úlohu s plusky a minusky vyřešit rychleji za pomoci stroje se dvěma páskami: jednou vstupní a jednou pracovní (jelikož odpovídáme pouze ANO/NE, výstupní pásku nepotřebujeme).

Půjde to snadno: nejprve projdeme vstup a všechna – zkopírujeme na pracovní pásku. Poté vstup projdeme podruhé a za každé + smažeme jedno – z pracovní pásky; pokud už tam žádné není, odpovíme NE. Na konci ověříme, zda je pracovní páska prázdná, a podle toho odpovíme ANO nebo NE.

Stroj pracuje s abecedou $\{+, -, \sqcup\}$ a stavy $\{S_0, R, \text{ANO}, \text{NE}\}$. Jeho program vypadá následovně:

$$\begin{aligned}
(S_0, +, \alpha) &\to ((+, \rightarrow), (\alpha, \bullet), S_0)\\
(S_0, -, \alpha) &\to ((-, \rightarrow), (-, \rightarrow), S_0)\\
(S_0, \sqcup, \alpha) &\to ((\sqcup, \leftarrow), (\alpha, \bullet), R)\\
\\
(R, +, -) &\to ((+, \leftarrow), (\sqcup, \leftarrow), R)\\
(R, +, \sqcup) &\to \text{NE}\\
(R, -, \alpha) &\to ((-, \leftarrow), (\alpha, \bullet), R)\\
(R, \sqcup, \sqcup) &\to \text{ANO}\\
(R, \sqcup, -) &\to \text{NE}
\end{aligned}$$

(zbývající kombinace znaků na páskách nenastanou)

Jelikož trojrozměrnou tabulku neumíme nakreslit, popsali jsme ji pomocí pravidel $(S, x, y) \to ((x', p), (y', q), S')$. Tím myslíme: jsme-li ve stavu S a vidíme-li na vstupní pásce znak x a na pracovní znak y, pak na vstupní pásku zapíšeme x' a provedeme posun p, na pracovní pásku zapíšeme y' a provedeme posun q; nakonec přejdeme do stavu S'. Symbol α značí libovolný znak abecedy, na levé straně pravidla stejný jako na pravé. Jelikož na vstupní pásku není povoleno zapisovat, tváříme se, jako bychom tam vždy zapisovali to, co tam už je. Pokud zastavujeme výpočet, píšeme prostě ANO či NE a nestaráme se, co stroj udělá s páskami.

Náš nový stroj pracuje v lineárním čase (vstup projde všeho všudy dvakrát) a spotřebuje lineární množství prostoru.

Úkol 2 [5b]: Vyřešte první úkol na vícepáskovém Turingově stroji. Snažte se dosáhnout co nejlepší časové i paměťové složitosti.

Úkol 3 [2b]: Dokažte, že každý Turingův stroj jde upravit, aby si vystačil pouze se dvěma stavy (kromě ANO a NE). Počítat musí samozřejmě stále totéž, byť třeba pomaleji.

Úkol 4 [2b]: Ukažte, jak vyřešit předchozí úkol pro jednopáskové stroje tak, aby zůstaly jednopáskovými.

Poznámky

S Turingovými stroji se řešitelé KSP potkali už jednou: zabýval se jimi seriál 14. ročníku. Zkuste se na jeho zadání i řešení podívat, skrývá se tam nejedna další zajímavost. Letošní úlohy jsou ale samozřejmě řešitelné i bez toho.

Zkoušeli jsme zesílit TS přidáním dalších pásek. Co kdybychom ho naopak chtěli trochu oslabit? Nabízí se nahradit přepisovatelnou pásku „děrnou páskou“ – v abecedě existuje speciální znak „díra“ a stroj má povoleno pouze přepsat mezeru na libovolný znak a nemezerový znak na díru. V úloze 14-4-5 se dokazuje, že takový TS je stejně silný jako ten klasický.

V omezování TS bychom mohli ještě pokračovat: mohli bychom úplně zakázat zápis, takže stroj by směl jenom pohybovat se po pásce a číst (jinými slovy měl by jenom vstupní pásku a žádné pracovní). Takový stroj už je mnohem slabší, konkrétně ekvivalentní s konečnými automaty, které jste mohli potkat v seriálu 23. ročníku.

Martin Mareš

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 v několika různých jazycích (v nízkoúrovňovém C, abychom viděli implementaci blízkou tomu, jak to vidí počítač, a v Pythonu, kde je psaní o něco příjemnější). Nebudeme ale probírat základy syntaxe těchto jazyků, tu si případně můžete nalézt jinde.[2] 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 řešení).

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“.[3]

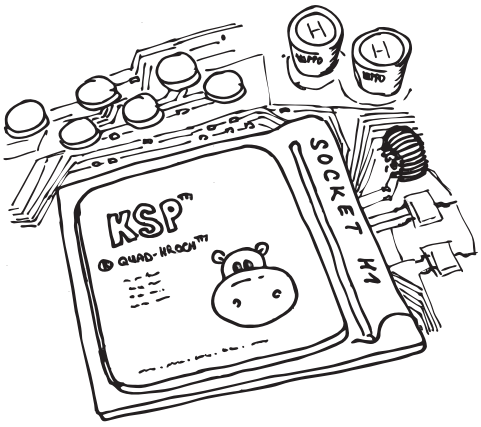
Takovýto příkaz klidně můžeme nazvat algoritmem, ačkoliv to bude asi znít nezvykle – pojem algoritmus se totiž používá hlavně ve světě počítačů. Je to tedy nějaká posloupnost základních příkazů, která řeší nějaký problém. Výběr konkrétního programovacího jazyka rozhoduje o tom, jaké základní příkazy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

Mezi základní příkazy patří:

- Manipulace s daty v paměti (uložení či načtení hodnoty, více detailně v další kapitole)
- Provedení nějakého numerického výpočtu $(+, -, *, /)$
- Vyhodnocení nějaké podmínky a odpovídající větvení programu *(Pokud A, tak proveď B, jinak proveď C)*
- Opakování nějakého příkazu *(Dokud platí A, dělej B)*
- Vstup a výstup programu (nejtypičtější je asi 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 pak skládá každý další algoritmus. Programem pak rozumíme realizaci algoritmu v nějakém konkrétním programovacím jazyce.

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 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 zadané číslo a výsledek opět uložíme na stejné místo.

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

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

Pole

První datovou strukturou, kterou si představíme a která se na výše nastíněnou situaci náramně hodí, je *pole*. To představuje spoustu přihrádek (proměnných) naskládaných v paměti za sebou, ke kterým typicky přistupujeme přes jeden společný název pole a jejich pořadové číslo neboli index (jako `NazevPole[0]`, `NazevPole[1]`, `...`).[4]

Ve většině základních jazyků je pole jen statické, tedy to znamená, že 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.

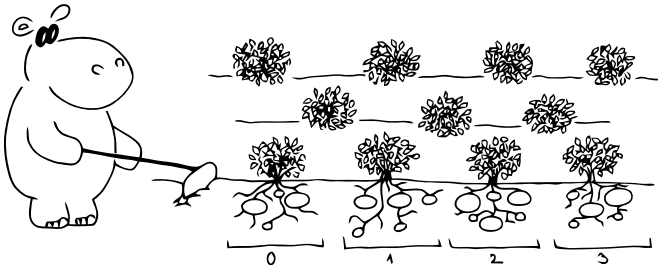
[2] `http://ksp.mff.cuni.cz/study/odkazy.html`

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

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

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 která operace bude trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, tak když se počítače zeptáme na obsah přihrádky `pole[42]`, přesně ví, na které adrese 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,[5] nejdříve však doporučujeme dočíst tuto kuchařku.

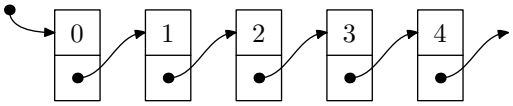
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu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

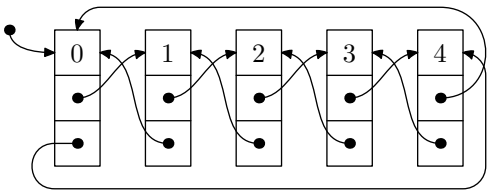
Představme si ale teď situaci, kdy by si každý prvek ještě pamatoval pozice sousedů. Pak bychom mohli mít prvky libovolně rozházené v paměti a jen by se na sebe vzájemně odkazovaly (první prvek by tvrdil, že druhý je na pozici X, druhý by tvrdil, že třetí je na pozici Y, a tak dále).



K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každá část paměti v počítači má nějaké své číslo. Když si vytváříme nějakou pojmenovanou proměnnou, tak ta vlastně odkazuje na nějaké 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 případně 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).



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, tak na něj samozřejmě 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, tak jen šikovně přepojíme ukazatele. Pokud předtím ukazatele vedly $A \to B$, tak teď povedou $A \to C \to B$ (a při odebírání naopak).

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

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

```
typedef struct tprvek tprvek;

// Struktura pro prvek obsahující dopředné
// i zpětné odkazy
struct tprvek {
    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 vratí 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() {
    tprvek *koren = novy(1);
    tprvek *aktualni = vloz_za(koren, 2);

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

    return 0;
}
```

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

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

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

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

    def 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)
```

Tyto základní struktury už jsou často předpřipravené jako součást nějakých knihoven v daném jazyce, ale je velmi důležité rozumět tomu, jak vnitřně fungují. Protože jedině když budeme vědět, co je jak rychlé a efektivní, tak 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 nad dalšími strukturami, tentokrát již více teoreticky.

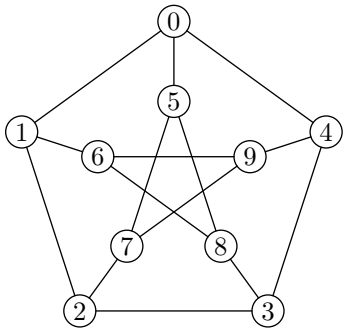
Stromy a grafy v informatice

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 ze zmíněných, my 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.

Graf můžeme doplnit třeba 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), pak graf nazýváme *ohodnocený*. Další možnou úpravou je, že každá hrany 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).

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

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):

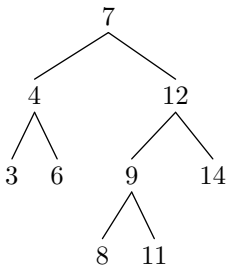
- **Seznam sousedů** – vrcholy grafu budeme mít uložené v poli a u každého vrcholu budeme mít (spojový) seznam čísel dalších vrcholů, do kterých z aktuálního vrcholu vede hrana. Zabírá místo $\mathcal{O}(n+m)$ a hodí se pro řídké grafy (tedy grafy, kde je m řádově stejné jako n).
- **Matice sousednosti** – 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.[6]

Stromy

Možná si říkáte, co má informatika u všech elektronů společného s lesnictvím? Kupodivu celkem mnoho a bez stromů bychom se v mnoha případech 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 žádný cyklus. To znamená, že mezi každými dvěma vrcholy stromu existuje právě jedna cesta. Strom pak můžeme za nějaký zvolený vrchol *zakořenit*. Tím se nám z tohoto vrcholu stane *kořen*, ze kterého směrem dolů (informatické stromy mají tradičně kořen nahoře) vyrůstají nějaké *podstromy*.



U stromu navíc mluvíme o *hloubce*, to je vzdálenost od kořene k danému vrcholu. Hloubka celého stromu pak je nejdelší vzdálenost od kořene k nějakému *listu* (tak říkáme vrcholům, které již nemají žádné *syny*, tedy vrcholy, které by z nich vyrůstaly). Vrcholy stromu také můžeme podle jejich hloubky uspořádat do jednotlivých *hladin*.

Velmi často používáme stromy, které jsou nějak pravidelné. Časté jsou *binární stromy*, které mají v každém vrcholu maximálně dva syny (levý a pravý podstrom). Reprezentovat se dají buď obecně jako každý

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

jiný strom (v každém vrcholu spojový seznam podstromů), nebo velmi pěkně i v poli.

Stačí si uvědomit, že pokud vrcholy očíslujeme od nuly, tak vrchol s indexem i má jako syny vrcholy s indexy $2i + 1$ a $2i + 2$. Pokud bude strom úplný – to znamená, že bude mít poslední hladinu plně zaplněnou (bez děr) – tak bude i pole plně zaplněné bez prázdných míst.

Speciálním případem je pak ještě *binární vyhledávací strom*. Je to normální binární strom, pro nějž navíc platí, že všechny hodnoty v levém podstromu jsou menší než hodnota ve vrcholu a všechny v pravém podstromu naopak větší.

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.

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 *rekurzivní funkce*, která volá sama sebe (s jinými parametry, jinak by to asi nemělo smysl).

U rekurzivních funkcí je nejdůležitější věc definovat nějakou *okrajovou podmínku*, tedy podmínku, při níž už se rekurze zastaví. Jinak by se totiž mohlo stát, že by rekurze běžela donekonečna.[7]

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 $0, 1, 1, 2, 3, 5, 8, \ldots$. Pokud toto přepíšeme do programového kódu, tak dostáváme následující zápisy:

```
// Kód v C:
int fib(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return fib(n-1) + fib(n-2)
}
```

```
# Kód 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, tak se každé rekurze můžeme zbavit. Rekurzivní volání totiž můžeme šikovně přepsat na nějaký cyklus se *zásobníkem* (spojový seznam, do kterého vkládáme jen z jedné strany a ze stejné strany z něj i odebíráme – tedy prvek přidaný jako první odebereme až jako poslední).

Pak jen v cyklu odebíráme prvky ze zásobníku, dokud není prázdný, a za každé rekurzivní zavolání do zásobníku přidáme parametry, se kterými bychom naší 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.

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

```
// Kód v 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;
}
```

[7] Tedy přesněji do doby, kdy našemu počítači dojde paměť, do které si ukládá jednotlivá volání funkcí.

```
# Kód 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.

Zpětné vyhledávání 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:

```
// Kód v 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;
}
```

```
# Kód 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
```

Jak je vidět, tak v každém kroku zkusíme nejdříve použít pětikorunovou minci, 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 když je prostřední prvek menší než k, hledáme v pravé polovině.

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.

[8] Pokud není řečeno jinak, tak pro nás v informatice značka log znamená *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 $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

Ukázka hlavní smyčky v C:

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

do {
    int prostredni = (L+R)/2;
    int 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:

```
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. Zastaví se ve chvíli, kdy třídí posloupnost délky jedna. Pak jen v každém kroku ze dvou setříděných menších posloupností vyrobí jejich sléváním setříděnou posloupnost dvojnásobné délky.

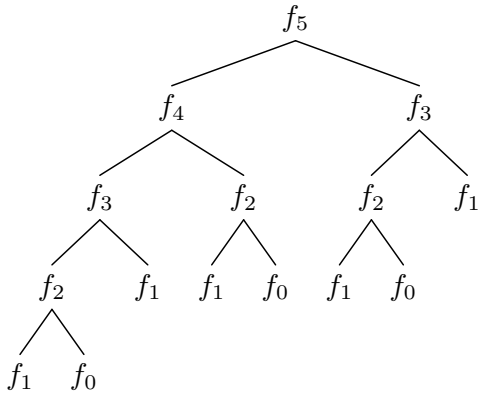
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[9]

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 naší rekurzivní implementaci počítání Fibonacciho čísel z kapitolky Rekurze.

Když se podíváme na výpočet čísla `fib(5)`, tak 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.

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.

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

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 však hodí si ještě před samotným výpočtem předvypočítat 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).

Tedy by nás mohlo 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];
```

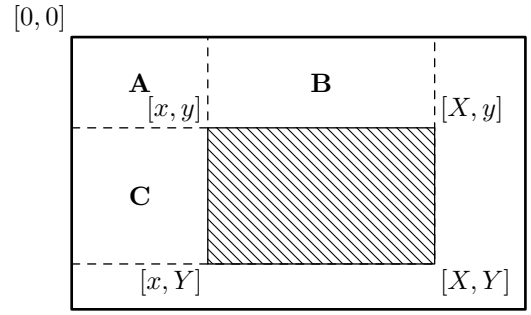
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas, prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

Dvourozměrné prefixové součty

Prefixové součty se však 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

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

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 například výše uvedený algoritmus na hledání souvislé podposloupnosti s co největším součtem, který se za běhu ptal na součty nějakých podintervalů).

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.

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 daný 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

„U Elektrona Svatýho, co myslel tímhle?“

„A časovou složitost má kde?“

„Proč by tohle mělo fungovat?“

Takové a mnohé další dotazy si my, organizátoři KSP, klademe při opravování došlých řešení. Bohužel, někdy na ně odpovědi nenajdeme, a proto ze zvědavosti strhneme několik bodů. Sice se mnozí řešitelé časem naučí, co dělat mají a co ne, aby získali co nejvíce bodů, ale chvíli to trvá a některé při tom ztratíme.

Pro ulehčení nováčkům jsme tedy připravili tento návod. Představme si modelovou úložku. Úkolem je spočítat největšího společného dělitele dvou čísel (předstírejte na chvíli, že jste to ještě nikdy neřešili). Na ní si ukážeme postup, jak dojít ke správnému řešení, a také pár tipů, co nedělat.

Napřed je samozřejmě potřeba řešení vymyslet. První, co nás pravděpodobně napadne, je zkusit vydělit obě čísla tím menším z nich. Pokud po dělení je nějaký nenulový zbytek, pak to není největší společný dělitel a my zkusíme číslo o 1 menší. A tak dále, dokud nepotkáme takové, které beze zbytku dělí obě. To je samozřejmě společný dělitel a je první, kterého jsme potkali, takže je největší.

Takový postup má mnohé výhody – je jednoduchý, zcela očividně vrátí správný výsledek, a navíc máme jistotu, že někdy skončí (zastavíme se určitě nejpozději u jedničky). Ale jde to i rychleji (co znamená „rychleji“, můžete najít výše v kuchařce).

Zapomeňme na rozklad na prvočísla, který je na první pohled příliš komplikovaný, než aby měl šanci na úspěch. Vezmeme dvě zadaná čísla. Pokud se rovnají, pak jsou (obě) největším společným dělitelem. Pokud ne, to větší z nich zmenšíme o to menší a pokračujeme stejně.

Navíc pokud bude jedno číslo obrovské a druhé maličké, budeme to maličké odečítat opakovaně, až získáme… zbytek po dělení většího menším. To by mohlo výpočet ještě zrychlit.

Dobrá, postup máme. Co teď? Nyní je vhodné napsat vlastní program v nějakém jazyce. My organizátoři ho sice nepožadujeme „povinně“, ale pomůže odhalit nedostatky (či nedomyšlené „zrady“) v algoritmu.

Například na našem příkladě bychom zjistili, že zatímco odečítací metoda funguje, metoda zbytková se bude pokoušet dělit nulou (alespoň tak, jak jsme ji popsali). V nějakou chvíli již bude v menším čísle uložený výsledek. Při odčítání dojdeme postupně ke stejnému číslu – ale při dělení získáme zbytek 0 jedinou operací. V takovou chvíli je třeba skončit. Navíc nám program pomůže pochopit méně jasné části popisu.

Program by vypadal třeba takto (zapíšeme ho v Pascalu):

```
var x, y: integer;
begin
   read(x, y);
   while (x<>0) and (y<>0) do
      if x>y then x := x mod y
             else y := y mod x;
   writeln(x+y);
end.
```

(všimněte si malého triku: když je na konci jedno z čísel `x`, `y` nulové, tak `x+y` je rovno tomu nenulovému).

Nakonec je třeba vytvořit text řešení. Co by měl obsahovat? Určitě popis algoritmu, a to takový, aby kdokoliv, kdo umí jen trochu programovat, podle něho byl program schopný napsat. Při tomto popisu lze použít nějaký již existující algoritmus jako stavební kámen, například se odkázat na nějakou knížku nebo programátorské kuchařky z webu KSPčka.

Pokud tento popis bude nejasný nebo nejednoznačný, pokusíme se nějakou myšlenku vykoukat z přiloženého programu, avšak už za nedostatečný popis nejspíš pár bodů ztratíte.

Další částí by mělo být nějaké zdůvodnění, proč vlastně program počítá, co se po něm chce. Určitě nám ještě nevěříte, že popsaná magie s odčítáním funguje. My mnohým tvrzením, která nám dojdou, také ne (některému až tak moc, že si dáme práci ho vyvrátit).

Co by bylo důkazem v tomto případě? Třeba následující textík (zapsaný opravdu důkladně, jako formální důkaz, obvykle však stačí myšlenka):

Tvrdíme, že v každém kroku algoritmu nahradíme větší z čísel x, y číslem $|x-y|$ a zachováme přitom všechny společné dělitele dvojice x, y, tím pádem samozřejmě i největšího společného dělitele.

Jakmile se algoritmus zastaví (což zajisté učiní), držíme v ruce dvě čísla, z nichž jedno je nula (a ta je beze zbytku dělitelná čímkoliv), a tedy největší číslo, které dělí obě, je to druhé, nenulové.

Zbývá tedy dokázat, že jeden krok algoritmu zachovává všechny společné dělitele. Mějme nějakého společného dělitele d čísel x, y. Navíc předpokládejme, že $x > y$, takže x budeme nahrazovat číslem $z = x \bmod y$ (kdyby $x < y$, tak jen prohodíme x s y). Co z toho víme:

$$x = x' \cdot d$$
$$y = y' \cdot d$$
$$z = x - t \cdot y$$

pro nějaká x', y', t. Číslo z tedy můžeme upravovat takto: $z = x - ty = x'd - ty'd = (x' - ty')d$. Takže z je také dělitelné číslem d.

Nechť naopak nějaké d dělí dvojici z, y. Pak víme:

$$z = z' \cdot d$$
$$y = y' \cdot d$$
$$x = z + t \cdot y,$$

takže $x = z'd + ty'd = (z' + ty')d$ je také dělitelné číslem d. Zjistili jsme tedy, že společní dělitelé dvojic x, y a z, y jsou titíž.

Nakonec je potřeba odhadnout, jak dobrý algoritmus jsme vymysleli. K tomu slouží odhady časové a paměťové složitosti. Jelikož jsou velmi důležité, věnovali jsme jim celý jeden díl[11] receptů z programátorské kuchařky.

Pokud máte určování složitostí v malíčku nebo jste si přečetli kuchařku, můžete se podívat na pokračování vzorového řešení úložky s největším společným dělitelem:

Paměťová složitost našeho algoritmu je zcela očividně konstantní – máme jen dvě proměnné na čísla, v nich provádíme veškeré operace.

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

Časová bude chtít trošku odhadovat. Jednak, co je velikostí vstupu? Tou bude součet velikostí obou čísel, tedy $n = x + y$ (délka výpočtu totiž nezávisí na počtu čísel na vstupu – tam je jich vždy stejně – ale na jejich hodnotách). V každém kroku se jedno z čísel sníží alespoň o 1 a nikdy se nedostaneme do záporných čísel. Takže bychom mohli klidně psát, že časová složitost je $\mathcal{O}(n)$ – určitě náš program nepoběží déle.

To je sice pravda, ale moc jsme se nevytáhli – stejnou časovou složitost měl i původní algoritmus se zkoušením všech potenciálních dělitelů. Tak co teď? Vymyslet lepší algoritmus? Ne, my na to půjdeme šalamounsky – vymyslíme lepší důkaz.

Opět předpokládejme, že přecházíme od dvojice x, y ke dvojici z, y, kde $z = x \bmod y$. Dokážeme, že $z \leq x/2$, takže každým krokem algoritmu se aspoň jedno z čísel zmenší aspoň dvakrát. Přitom kroků, kdy se dvakrát zmenšilo původní x, může být celkem nejvýš $\lfloor \log_2 x \rfloor$, a analogicky pro y. Proto je celková časová složitost $\mathcal{O}(\log_2 x + \log_2 y) = \mathcal{O}(\log n)$.

A proč je $z \leq x/2$? Rozebereme dvě možnosti: buďto je $y \leq x/2$, ale pak stačí využít toho, že zbytek po dělení je vždy menší než dělitel, tedy $z < y \leq x/2$. A nebo je $y > x/2$, ale pak $z = x - y \leq x - x/2 = x/2$.

Na závěr dodejme, že popsanému algoritmu na počítání největšího společného dělitele se říká Eukleidův.

Několik špatných pokusů

Minulá část obsahovala popis, jak vypadá správné řešení. Nyní zmíníme několik chyb, se kterými se celkem pravidelně při opravování setkáváme.

Jednou (a asi nejvážnější) z nich je, když nám přijde pouze zdrojový kód, který je občas (ale sporadicky) komentovaný a není k tomu žádný popis. Popis má být hlavní částí řešení, zdrojový kód pouze doplňkem.

Opačný extrém je příliš podrobné (a komplikované) vyprávění či slohová práce. Opravdu neplatí, že čím delší text, tím více bodů. Úlohy v KSP jsou dělané tak, aby se daly jednoduše popsat na stránku nebo dvě. Řešení o 20 stranách je tak dlouhé, že v něm prostě něco špatně být musí.

Další celkem běžný problém je špatně pochopitelný popis. Zkuste si text po sobě přečíst – s vědomím, že člověk, který ho bude číst, možná vaše řešení vůbec nezná a nic o něm neví. Nejlépe s odstupem několika hodin či dní.

Do této oblasti patří i pravopis (někdy špatně umístěná nebo chybějící čárka ve větě může úplně změnit význam) a v případě psaní ručně i čitelnost rukopisu (za to, co nepřečteme, body nedáme).

A samozřejmě plný počet bodů nedostanete ani za řešení, které nefunguje.

Co dělat když...

Mnoho lidí KSP řešit nezačne, přestože by je třeba i bavilo. Většinou proto, že narazí na nějaký problém, který ale obvykle není tak neřešitelný, jak vypadá.

Pokud vám některá úloha přijde příliš těžká, nezoufejte. Snažíme se dávat i „šťavnaté“ úložky pro pokročilejší řešitele – samozřejmě ne všechny, některé jsou lehké. K jejich rozpoznání vkládáme do zadání značky, jejichž popis najdete na začátku letáku.

A co v případě, když vás napadne jen pomalé řešení? Je na něm sice jasně vidět, že jsme při zadávání mysleli na něco rychlejšího, ale vy na to ne a ne přijít. Rozhodně napište alespoň to pomalé – zatímco za nefunkční řešení nedáváme skoro nic, za pomalejší řešení dáváme docela dost (tedy, podle toho, jak moc pomalejší je).

Nakonec malý tip. Zkuste začít řešit s předstihem. Velmi pomáhá, když je pár dní času na to, aby pěkné řešení „napadlo“.