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, Katedra správního práva, Klub severských psů, nebo třeba Korespondenční seminář z programování! Korejští kynologové mají smůlu, zůstaneme u posledního.

Korespondenční seminář z programování?

Celostátní a celoroční soutěž v programování pro studenty středních škol a vyšších ročníků základních škol.

Jak tato soutěž probíhá?

Jeden ročník je rozdělen na 5 sérií, přičemž v každé obdrží účastníci zadání 7–8 úloh (buď poštou nebo po Internetu). Na vyřešení série bývá několik týdnů času, 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 nebo 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.

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.

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 považujeme za řešitelnou i pro začátečníky, zkušení řešitelé ji jistě dají 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 naleznete 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 najdete na našich webových stránkách.

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 každý, kdo překoná 50% hranici bodů, se 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 25. ročníku obdrží libovolnou knihu dle svého ctěného výběru (v případě 2. a 3. nejlepšího jen českou).

Co budu dělat o prázdninách?

První série se odevzdává až koncem října. Během prázdnin se tak můžeš kochat přírodou, surfovat, lézt po horách anebo řešit nultou sérii! V této originální sérii jsme přichystali několik netradičních úložek, jejichž řešení můžete odevzdávat na našich stránkách až do 20. srpna.

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 25. ročníku KSP. Každá série letos obsahuje 7–8 úloh a z nich se 5 nejlépe vyřešených započítává do celkového bodového hodnocení.

V každé sérii budou dvě lehké úlohy pro začátečníky za menší počet bodů.

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

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

Termín odevzdání první série je stanoven na pondělí 22. října v 8:00 SELČ, což znamená, že papírové řešení byste měli podat na poštu do středy 17. října.

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 hash: `7F:53:E7:00:60:F2:24:93:8F:52:51:EC:1E:A8:34:54:86:69:32:7D`.

Také nám řešení můžete poslat klasickou poštou na adresu

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
118 00 Praha 1

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

První série dvacátého pátého ročníku KSP

(volně přeloženo z japonského originálu)

Vážený strýčku,

jsem Vám velice vděčen za Vaši pomoc při přípravách naší cesty do České republiky. Je to velice zajímavá země s mnoha prazvláštními obyčeji, které se Vám pokusím aspoň trochu přiblížit.

Nafotil jsem spoustu fotografií, ukázat Vám je však nemohu – jak píši dále, o všechny jsem přišel.

Už náš příjezd byl takový zvláštní – čekal jsem, že nás na letišti vyzvedne průvodce, pojedeme autobusem do centra, vysedneme na nějaké rušné ulici plné turistů a vydáme se obdivovat památky.

Místo toho jsme však nasedli do dodávky, která trčela v dopravní zácpě snad hodinu. Pak jsme se dlouho proplétali uzounkými uličkami, až jsme se zastavili v jedné velice zapadlé, nikde nikdo a skoro nebylo vidět na slunce.

Chvíli to trvalo, než jsme se vymotali z uliček a dostali do míst, kde byli i jiní lidé. Vydali jsme se s rodiči přes takový starý most do centra. Cestou jsme samozřejmě všichni fotili.

25-1-1 Fotografování — 12 bodů

Máme N japonských turistů, kteří se mezi sebou navzájem fotí. Typicky, když jeden Japonec vyfotí druhého, ten druhý mu to musí ze slušnosti oplatit.

Japonci mají v oblibě jednu hru: v určitou chvíli někdo oznámí přirozené číslo K, načež si všichni spočítají, kolik ostatních mají nafoceno (což je zjevně počet lidí, kteří fotili je samotné). Každý, kdo má nafoceno méně než K kamarádů, vypadává ze hry.

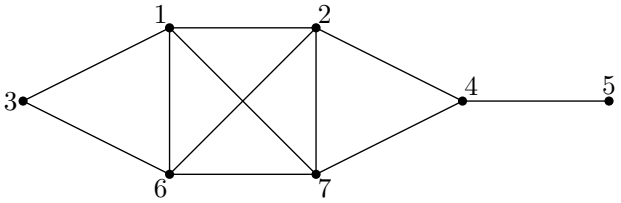
V druhém kole se počítají pouze ti, kteří nevypadli. Opět ti, kteří mají nafoceno méně než K nevypadnuvších, vypadnou ze hry.

Takto hra pokračuje, dokud se stav mezi dvěma koly mění. Všichni zbylí hru vyhrají.

Víte, které dvojice Japonců se navzájem vyfotily. Určete pro každého z nich, pro jaké maximální K hru vyhraje.

7 bodů dostanete, pokud najdete vyhrávající skupinku pro $K = 3$.

Příklad: Uvažujme situaci pro $N = 7$, přičemž se vyfotily dvojice: 1–2, 2–4, 4–5, 3–1, 3–6, 6–1, 6–2, 6–7, 7–1, 7–2 a 7–4. Pro $K > 3$ nevyhraje nikdo. Pro $K = 3$ vyhrají hráči $1, 2, 6, 7$, pro $K = 2$ vyhrají všichni až na hráče 5 a pro $K < 2$ vyhrají všichni.



Úzkými uličkami jsme se dostali na náměstí. Měli zde moc pěkné hodiny s figurkami. Také jsme se podívali do místní tržnice. Nevěřil byste, jak jsou místní trhovci hádaví.

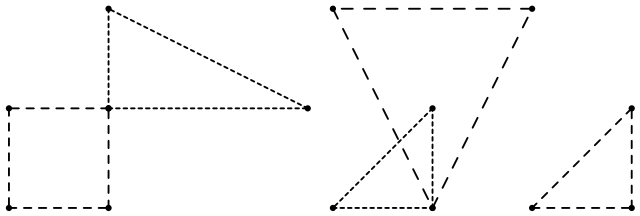
25-1-2 Stánky na náměstí — 12 bodů

⚠ Máme T trhovců. Každý trhovec má vlastní stánek. Při jeho stavbě zkušeným okem určil oblast, odkud na něj turisté nejlépe uvidí, tato oblast má tvar konvexního mnohoúhelníka.

Problém nastává, pokud se dvě takovéto oblasti protínají – potom se trhovci neustále hádají a přetahují si navzájem turisty.

Radní o tomto problému ví a snaží se mu zabránit. Oblasti vám zadají jako T seznamů bodů, kde body jsou vždy zadané podél obvodu. Každý seznam bodů tvoří konvexní mnohoúhelník. Chtěli by vědět, jestli se nějaké dvě oblasti překrývají.

Příklad: Pro $T = 2$ a oblasti určené $\{[1,1],[2,1],[2,2],[1,2]\}$ a $\{[2,2],[4,2],[2,3]\}$ se tyto dvě neprotínají (vlevo), ale pro $T = 3$ a oblasti zadané pomocí bodů $\{[1,1],[2,1],[2,2]\}$, $\{[3,1],[4,1],[4,2]\}$ a $\{[1,3],[2,1],[3,3]\}$ se první a třetí oblast protínají (vpravo).



Pak jsme se vrátili po stejném mostě, prý se podíváme na místo, kde žije český prezident. Těšil jsem se hlavně na pražskou hradní stráž, slyšel jsem totiž, že její řazení při výměně je pověstné.

25-1-3 Řazení hradní stráže — 7 bodů

Při výměně dvou gard stráže ta odchozí nastoupí na nádvoří do řady. Příchozí stráž nastoupí vedle ní.

Obě gardy jsou přirozeně stejně velké – každá má právě N vojáků. Vojáci mají dopředu určeno, na kterém místě hlídají, každý z gardy hlídá na jiném místě.

Pro kontrolu, že jsou všichni a že žádné z míst nezůstane nehlídáno, se příchozí garda musí seřadit stejně jako ta odchozí.

Řazení podle protokolu probíhá tak, že se vždy odpojí poslední v řadě příchozí gardy a zařadí se na libovolné místo. Kolik takovýchto přesunů je nejméně potřeba, aby příchozí garda byla seřazena stejně jako odchozí?

Příklad: Když si vojáky očíslujeme $1 \dots N$ dle místa, kde budou hlídat, tak pro odchozí řadu $4, 2, 1, 3, 5$ a příchozí řadu $1, 3, 5, 2, 4$ jsou potřeba dva přesuny.

To jsem bohužel už neviděl. Na tom mostě jsem se snažil vyfotit hlavu ve zdi, než se mi to však podařilo, všichni byli dávno pryč. Snažil jsem se je dohnat, ale nějak jsem se ztratil.

Dostal jsem se do krásného parku, jsou tady i nějaká obrovská mimina.

„Dobrý den, pane, promiňte, že obtěžuji, ale neviděl jste tu skupinku turistů z Japonska?“

Pán vypadal velmi vyjeveně. Pak jen zadrmolil „Sorry, don't speak english,“ a zmizel. Copak já na něj mluvím anglicky? Vždyť to ani neumím... Zkusil jsem se ptát ještě pár dalších, vždy s podobným výsledkem. Koukali na mě, jako bych spadl z Marsu. Jeden z nich mi zdviženým prostředníčkem naznačil, že jsem jednička, ale stejně mi nepomohl.

Nikdo mi nerozumí. Sedl jsem si na lavičku, sklopil hlavu a přemýšlel, jak se dostanu zpátky. Taková ostuda! Takhle zahanbit své rodiče!

„Ahoj, co tu děláš tak sám?“

Zvedl jsem hlavu. Tak přece někdo! Stála nade mnou menší hnědovlasá slečna, na sobě měla hranaté kovové brýle a přes rameno brašnu.

„Kde máš rodiče?“

„Já nevím. Byli jsme támhle na mostě, fotil jsem, ostatní najednou zmizeli.“

Když už jsem měl v ruce foťák, tak jsem si slečnu vyfotil.

„A kam šli?“

„Říkali něco o hradu, ale těžko říct.“

„A víš aspoň, kde se máte sejít?“

„Tady za rohem máme dodávku, nejspíš tam.“

„Tak pojď.“

Chvíli jsme se proplétali uličkami, než jsme našli tu jednu liduprázdnou, kde jsme parkovali. Teď už moc liduprázdná nebyla. Bylo tam asi dvacet mužů, všichni stáli kolem naší dodávky. Někteří se jen dívali okolo, někteří vykládali z naší dodávky nějaké zboží. Asi čtyři z nich stáli opodál a živě spolu diskutovali.

Samozřejmě jsem začal fotit, tohle se jen tak nevidí. Při páté fotce fotoaparát usoudil, že scéna je moc tmavá, a zapnul blesk. V tu chvíli se všichni zarazili a podívali se na mě. Spoušť jsem zmáčknul ještě jednou.

V tu chvíli mě ona slečna chňapla za ruku a táhla pryč. Rozběhli jsme se a utíkali, co nám síly stačily. Proč, proboha? Na otázky však nebyl čas. Pochopil jsem, že z nějakého důvodu nás nesmí chytit. Nejspíš tu slečnu znají a hrají nějakou společenskou hru.

Běželi za námi dva. Když se rozdělíme, budou nás hledat mnohem hůře. Vytrhl jsem slečně svoji ruku a rychle zahnul doprava. Nejspíš pochopila můj záměr a zahnula doleva.

V úzkých uličkách je snadno ztratíme.

25-1-4 Útěk — 12 bodů

Úzké uličky tvoří bludiště. V tomto bludišti jsou osoby, které nesmíte potkat. Tyto osoby neběhají chaoticky, mají jistý okruh, po kterém chodí stále dokola.

Bludiště je zadáno jako čtvercová síť o rozměrech $W \times H$, kde $1 \le W, H \le 70$. V bludišti jsou čtyři typy polí: zeď `#`, volno `.`, start `S` a cíl `C`. Start je v bludišti právě jeden.

V bludišti je N osob, kterým utíkáte, přičemž $0 \le N \le 2$. Pohyb každé z osob je dán seznamem souřadnic délky D, jehož konec navazuje na začátek. Osoba se po nich cyklicky pohybuje. Seznam je dlouhý $1 \le D \le 50$ a je vždy platný (osoby neprocházejí zdmi, místa na sebe navazují). Je-li $N = 2$, pak mohou obě osoby stát na stejném místě.

Pohyb v bludišti probíhá po tazích, v tahu se vždy můžete posunout o jedno políčko vodorovně nebo svisle, nebo stát na místě. Nejprve se pohnete vy, poté chytající osoby.

Najděte nejrychlejší cestu ven z bludiště, aniž byste potkali libovolnou z osob.

6 bodů získáte za řešení fungující pro $N = 0$, 3 body pro $N = 1$ a 3 body pro $N = 2$.

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.

Už mně dýchal na krk. Najednou se však zastavil a nešel dále. Došel jsem na větší obdélníkové náměstí, uprostřed vedly tramvajové koleje.

Uf. To bylo o fous.

Najednou jsem na chodníku uviděl peněženku. Jak mě to mí ctění rodiče vždy učili, sebral jsem ji, neotevíral a začal se poohlížet po policistech, kterým bych ji mohl odevzdat.

Jeden šel kousek ode mě. Vydal jsem se mu naproti a natáhl ruku s peněženkou. Policista došel ke mně, pořádně se na mě ani nepodíval, ignoroval peněženku a sebral mi foťák.

Stál jsem jako opařený, stále s nataženou rukou. Najednou byla prázdná. Nějak se tam objevila ta slečna, která se mnou předtím utíkala.

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

S peněženkou v ruce se o něčem s policistou dohadovala. Po chvíli však bez jediného slova zmizela.

Policista se na mě podíval a snažil se mi něco říct, já mu však nerozuměl. Pak mě odvedl s sebou na služebnu. Ach, strýčku! Připadal jsem si jako nějaký sprostý zloděj!

Policista mě posadil naproti svému stolu, chvíli na mě koukal, pak zmizel. Koukal jsem na práci jejich sekretářky. Pořád vytahovala z kartotéky nějaké papíry, občas na ně něco načmárala a pak je zase vložila zpátky. Vypadalo to však velmi neefektivně.

25-1-5 Algoritmus sekretářky — 7 bodů

Představte si, že sekretářka je naprogramovaná a provádí následující kód:

```
for i in range (0, N):
  for j in range (0, M):
    if (a[i]==c[j]):
      print b[i]+" "+d[j]+"\n";
```

Zkuste si rozmyslet, co kód (sekretářka) dělá a s jakou časovou složitostí. Potom zkuste vymyslet vlastní program, který dělá totéž efektivněji, nebo dokažte, že (asymptoticky) to už efektivněji nejde.

Vzpomněl jsem si, že mám v kapse kartičku, kterou mi rodiče napsali pro podobné případy. Vyndal jsem ji a podal policistovi, když se vrátil.

Ten si ji přečetl a zvedl telefon. Těm slovům jsem nerozuměl, přesto se mi vryla do paměti.

„Dobrý den, máme tu jedno ztracené dítě, má s sebou kartičku s vaším číslem. Jmenuje se Tanaka Mashiro, má krátké černé vlasy, velké kulaté sluneční brýle, na sobě černé tričko a modré rifle, nosí s sebou černý fotobatoh... Vážně? Výborně, dovedu ho k vám.“

Pak se zvedl, něco si zamumlal a naznačil, že mám jít s ním. Chvíli jsme se proplétali uličkami, než jsem uviděl naši krásnou vlajku.

Vešli jsme do budovy, nad kterou visela. Policista se chvíli bavil s vrátným, pak odešel a zmizel.

Vrátný se na mě usmál a ukázal na stůl vedle.

„Posaď se a chvíli počkej. Máš hlad? Donesu ti něco k jídlu.“

Teprve teď jsem se trošku uklidnil.

Při čekání jsem se díval z okna na zahradu. Je na ní krátký zelený trávník a spousta nádherných květin. Zajímalo by mě, jak ji sekají.

25-1-6 Sekání trávy — 10 bodů

Představte si trávník $N \times M$, který chceme posekat. Máme sekačku, kterou se můžeme pohybovat pouze vodorovně nebo svisle.

Začínáme v levém horním rohu a chceme každé políčko projet právě jednou (start je výjimka) a vrátit se na začátek. Na trávníku rostou květiny, které nechceme posekat.

Pro jaké N, M a umístění květin umíme trávník posekat? Pro zjednodušení předpokládejte, že na trávníku rostou květiny jen na jednom políčku.

4 body dostanete, pokud vyřešíte sekání trávníku bez květin.

Seděl jsem tam asi půl hodiny, než přišel pán v obleku.

„Vítej na japonské ambasádě. Už jsem volal tvým rodičům, vyzvednou si tě tady. Máš velké štěstí, že ses sem dostal. Ztrácí se tady spousta dětí. Mohl bys mi říct, co se vlastně stalo?“

Všechno jsem pánovi vyložil, celou dobu pozorně poslouchal. Zmínil jsem se mimo jiné i o zmizelém fotoaparátu.

Lehce se uklonil a natáhl ruce s fotoaparátem. Též jsem se uklonil a převzal jej.

„Datovou kartu bohužel nemáme, je mi líto.“

„Přesto vám co nejsrdečněji děkuji.“

„Teď mě omluv. Musím zmizet na schůzi.“

Přemýšlel jsem, jak se budou rodiče tvářit, až mě uvidí. Nejspíš na mě budou naštvaní, že jsem jim takhle zkazil dovolenou. Ani fotky nemám...

Nakonec ale měli spíš radost, že mě našli. Došli jsme zpátky k dodávce (tentokrát už kolem ní nikdo nestál), nasedli a jeli zpátky na letiště. Mezitím jsem od rodičů dostal aspoň GPS s logem, abych se mohl podívat, kam došli.

Při procházení logu jsem si všímal hlavně nadmořských výšek. Cestou po Praze dosti kolísaly. Zajímavý byl zejména počet kopečků a jeho změny s přiblížením mapy.

Po chvíli jsem si všimnul, že při vhodném přiblížení to vypadá tak, že se nejprve hodně dlouho stoupá, pak se dosáhne vrcholu a za ním se už jen klesá. Dal by se podobný výběr udělat i „ručně“?

25-1-7 GPS log — 7 bodů

Máte zadanou posloupnost nadmořských výšek tak, jak je turisté postupně procházeli. Zvládnete z nich vyškrtat co nejméně výšek tak, aby zbylá posloupnost obsahovala maximum, výšky před maximem pouze stoupaly a ty za ním klesaly?

Příklad: Pro posloupnost výšek $1, 3, 2, 2, 3, 4, 6, 7, 7, 5, 6$ je jedním ze správných řešení posloupnost $1, 2, 3, 4, 6, 7, 5$ (poslední prvek lze nahradit šestkou).

Jistě uznáte, že tato cesta byla velice zvláštní. Doteď pořádně nechápu, co se vlastně stalo. Možná mi to někdo někdy vysvětlí.

Tanaka Mashiro

Z japonštiny přeložil:

Radim „Rumcajz“ Cajzl

25-1-8 Sázíme v TEXu — 13 bodů

Bylo nebylo, 30. března 1977 obdržel Donald Ervin Knuth testovací výtisk jedné ze svých knih (druhé edice druhého dílu série *The Art of Computer Programming*). Prohlásil: „Strávil jsem 15 let psaním knih, ale pokud budou vypadat takhle nechutně, tak už žádnou nenapíšu.“ Pak stáhnul knihu z tisku a napsal TEX – sázecí systém a programovací jazyk.

První a zároveň poslední stabilní verzi TEXu vydal v roce 1989. Od té doby se již celý systém prakticky nezměnil. Název se čte „tech“ nebo „tek“, neboť ono X je ve skutečnosti velké řecké písmeno chí. Pokud byste náhodou nezvládli napsat název TEX se sníženým E, tak můžete napsat TeX. Nikdy ne však TEX.

Pojďme si tedy představit program a jazyk, díky kterým letáky KSP skvěle vypadají a dobře se čtou. Naučíme se používat TEX od úplných základů, začneme obyčejnými texty, probereme se matematickými vzorci a nakonec si ukážeme i zběsilé triky. Stanete se TEXniky, jaxepatří.

Instalace

Jak začít? Máte-li Linux, je to jednoduché. Nainstalujte si „TEXovou distribuci“ TEXlive (stačí minimalistická verze)

plus československá rozšíření. V Debianu a Ubuntu se jedná o balíky `texlive-base` a `texlive-lang-czechslovak`.

Ve Windows je to o něco složitější. Stáhněte si instalační balík z CTANu[2] a rozbalte jej. Máte-li dost místa na disku, můžete spustit `install-tl.bat`, odklikáte Next a nainstaluje se prakticky všechno, co byste kdy mohli i nemohli potřebovat. Zabere to přes 3 GB.

Pokud nechcete ucpat tolik místa na disku, případně nechcete stahovat tolik dat (to, co teď máte, je jen instalátor), spusťte `install-tl-advanced.bat` a v otevřeném okně si naklikejte menší instalaci. Pokud nevíte, případně se vám nad tím nechce přemýšlet, použijte tento návod:

1. Jako „Selected scheme“ vyberte „basic scheme“.
2. Z „Language collections“ vyberte československý balík.
3. TEXDIR upravte, pokud chcete změnit místo, kam bude TEX nainstalován.
4. Ujistěte se, že „Default page size“ je A4.
5. „Install TeXworks front end“ přepněte na Yes (poslední bod).
6. Klikněte na „Install TeX Live“. Instalátor stáhne z internetu všechno, co je potřeba, a nainstaluje. Celkem zabere okolo 250 MB.

Ahoj, světe!

Vyzkoušíme si, jak se TEX spouští. Použijeme k tomu testovací vstup:

```
Ahoj světe
\bye
```

Na Linuxu si prostě otevřete nějaký textový editor (autor má rád Vim, ale klidně použijte třeba GEdit), vyrobíte soubor `ahoj.tex`, otevřete si terminál, dokráčíte do příslušné složky příkazem `cd` a spustíte `pdfcsplain ahoj.tex`, což vyrobí soubor `ahoj.pdf`, který si prohlédnete.

Pod Windows bych doporučil použít TeXworks, které jste si před chvílí nainstalovali. Nejdříve trocha nastavování:

1. V menu Edit vyberte Preferences.
2. Na kartě Editor vyberte Encoding: ISO-8859-2.
3. Na kartě Typesetting v rámečku Processing tools klikněte na tlačítko +.
4. Vyplňte `pdfCSplain` jako Name a `pdfcsplain.exe` jako Program.
5. Do pole Arguments přidejte `$synctexoption` a `$fullname`.
6. Klikněte na OK, nastavte `pdfCSplain` jako Default a zavřete Preferences klikem na OK.
7. Zavřete TeXworks a otevřete je znovu.

Nyní můžete vepsat testovací vstup. Stiskem zeleného tlačítka se šipkou vlevo nahoře jej přeložíte a zobrazíte.

Poznámky:

- TeXworks existují i pro Linux, ale já mám radši Vim. Ovšem vyberte si dle chuti sami. De gustibus non est disputandum.
- TEX se dá spustit z příkazové řádky pod Windows, ale TeXworks jsou asi o něco příjemnější.
- S případnými problémy s instalací můžeme pomoct, pokud se nám svěříte na fóru na našem webu.
- Znak \ najdete na české klávesnici na klávese Q při stisknutém pravém Altu.

Sázíme texty do odstavců

Napíšete-li do vstupního souboru libovolný text, TEX jej vysází. Vyzkoušejte si to dle libosti. Chcete-li přejít na nový odstavec, vynechejte prázdný řádek. TEX považuje libovolné nenulové množství mezer a tabulátorů za jednu mezeru.

Taktéž se polykají konce řádků, nejedná-li se tedy o dva konce řádků za sebou (ty značí konec odstavce). TEX také spolyká veškeré mezery a tabulátory na začátku a na konci odstavce.

Některé znaky není možné zadat přímo do vstupního textu, neboť je TEX interpretuje jako speciální. Jsou to znaky `#$\%&^_{}~`. Pokud chcete napsat #, %, $, &, a _, stačí předřadit zpětné lomítko: `\#`, `\%`, `\$`, `\&`, a `\_`. Stříška je považována za diakritické znaménko, takže je potřeba ji nakreslit samostatně: `\^\relax` [3] se vykreslí jako ^.

TEX používá znak \ pro tzv. řídící sekvence. Ty můžou být jednoznakové jako v minulém odstavci, nebo víceznakové složené z písmen. Za písmennými se polykají mezery; kdybyste za nimi potřebovali mezeru vynutit, použijte `\␣`.

Ostatní (\, {, } a ~) jsou znaky, které se používají prakticky výhradně v matematickém zápise, ten nás čeká za chvíli.

Česká písmena s háčky a čárkami, stejně jako slovenská ľ, ŕ, Í, ä, ô apod. je možno napsat přímo do textu. TEX ale umí i ne-úplně-běžná písmena, třeba můžete napsat žiltovka nebo gyűrű. Existují totiž příkazy, které přidají vhodné diakritické znaménko nad/pod následující písmeno.

Vstup	*Výstup*		*Vstup*	*Výstup*	
`\‘o`	ò	*čárka dozadu*	`\.o`	ȯ	*tečka nad*
`\’o`	ó	*čárka dopředu*	`\u o`	ŏ	*půlkolečko nad*
`\^o`	ô	*stříška*	`\v o`	ǒ	*háček nad*
`\"o`	ö	*přehláska*	`\c o`	o̧	*cedilla*
`\~o`	õ	*vlnka*	`\d o`	ọ	*tečka pod*
`\=o`	ō	*čára nad*	`\b o`	o̲	*čára pod*
`\accent23 o`	o̊	*kroužek nad*			
`\H o`	ő	*dlouhá přehláska (v maďarštině)*			
`\t oo`	o͡o	*obloučekspojující 2 písmena*			

TEX automaticky dělí slova, která se nevejdou na konec řádku. Používá k tomu slovník, který je specifický pro každý jazyk. Základní je angličtina; pokud chcete nastavit češtinu nebo slovenčinu, použijte `\language\czech`, resp. `\language\slovak`.

Úkol 1 [2b]: Vysázejte tento text:

Poznatky získané cílevědomě v preadolescentním věku jsou adekvátní poznatkům pořízeným náhodně ve věku seniorském. (Co se v mládí naučíš, ve stáří jako když najdeš.)

Pokud se TEXu nepovedlo vysázet text tak, aby se vešel na řádek, objeví se za ním černý obdélník (slimák). Pak je potřeba řádek zlomit ručně, buď poradit TEXu, kde může lámat slovo, značkou `\-` (`slo\-víč\-ko`), nebo třeba přeformulovat text.

Chcete-li vysázet něco tučně nebo kurzívou, použijete `\bf` nebo `\it` ve skupině. Skupina je kus vstupu uzavřený mezi složené závorky `{` a `}`. Všechno, co se nastaví ve skupině, platí jen v ní, a po ukončení skupiny zase platí původní nastavení.

Takto tedy sázíme **tučně** a *kurzívou*:

```
Takto tedy sázíme {\bf tučně} a {\it kurzívou}:
```

[2] `http://mirror.ctan.org/systems/texlive/tlnet/install-tl.zip`

[3] `\relax` je prázdný příkaz

Úkol 2 [2b]: Vymyslete (vyzkoušejte), co dělá `\rm`, a dodejte ukázkový kód.

Za zmínku stojí ještě několik českých typografických pravidel. V češtině nesmí zůstat na konci řádku jednopísmenná předložka. Pokud by ji tam TEX nechal, je potřeba mu naznačit, že následující mezera je nedělitelná. K tomu slouží vlnka: `K~tomu`, `u~lesa`.

Když píšete české „uvozovky“, použijte řídící sekvenci `\uv`: `\uv{uvozovky}`. Pozor, není možné přesahovat uvozovkami přes hranici odstavce. V takovém případě je ale obvykle lepší vyznačit dlouhou přímou řeč třeba jiným řezem písma nebo třeba zúžením okrajů.

Také stojí za zmínku pomlčky a podobné znaky. Spojujeme-li dvě slova, například „česko-slovenský“ nebo „byl-li“, napíšeme to TEXu jako jednu pomlčku.

Pokud používáme pomlčku jako interpunkční znaménko – třeba zde, zapíšeme ji jako dvě pomlčky vedle sebe a oddělíme ji na obou stranách mezerou. Totéž platí pro rozsah, například „pondělí – středa“: `pondělí -- středa`.

Pro úplnost ještě zmíníme dlouhou pomlčku `---`, která se používá zřídka – obvykle na vyznačení náhle ukončené přímé řeči.

„Prosím pozor —“ Nádražní rozhlas zmlknul, světlo zhaslo, notebook se přepnul na baterii. Vypadla elektřina.

Matematický mód

Velké přednosti TEXu jsou v sazbě matematiky. Vyzkoušejme si základní věci. Matematika se v TEXu obaluje mezi dolary (`$`), neboť v dřevních[4] dobách typografie byla její sazba velmi drahá.

Jednoduchou matematiku stačí psát. Mezery se v matematickém módu ignorují zcela, TEX je počítá podle poměrně složitého algoritmu a v drtivé většině případů vypadají vysázené vzorce hezky.

`$a+b$`	$a+b$
`$a+b^2$`	$a+b^2$
`$a(b + c)$`	$a(b+c)$
`$abc + def$`	$abc+def$
`$a_1+a_2$`	a_1+a_2
`$a+b^{2c}$`	$a+b^{2c}$
`$a+b^2c$`	$a+b^2c$

Povšimněte si rozdílu mezi posledními dvěma řádky. Je zde využita skupina, která slouží k označení, co všechno má být horním indexem. Stejně se chová dolní index.

Horní a dolní index se vejde pod sebe, jsou-li uvedeny oba.

`$a_1^2 = a^2_1$`	$a_1^2 = a_1^2$
`$P_x^y \ne P{}_x^y$`	$P_x^y \ne P{}_x^y$
`$2^{2^{2^x}}$`	$2^{2^{2^x}}$
`$x_{y^a_b}^{z_c^d}$`	$x_{y_b^a}^{z_c^d}$
`$a', a'', a''', \dots$`	$a', a'', a''', \dots$

Zde se objevila sekvence `\dots`, která vykreslí trojtečku se správnými rozestupy teček. Je možno ji používat i mimo matematický mód ... Prohlédněte si rozdíl oproti třem samostatným tečkám ...

`$\sqrt{2}$`	$\sqrt{2}$
`$\sqrt{x^3-1}$`	$\sqrt{x^3-1}$
`$\overline{x+y}$`	$\overline{x+y}$
`$\overline{\overline{x}+y}$`	$\overline{\overline{x}+y}$
`$\root 3 \of 2$`	$\sqrt[3]{2}$
`$\root n^2+n+1 \of {n^2-n+1}$`	$\sqrt[n^2+n+1]{n^2-n+1}$

TEX umí v matematickém módu mnoho různých značek a symbolů. V prvé řadě umí řeckou abecedu, některá písmena dokonce ve více variantách. Vyzkoušejte:

`\alpha`, `\beta`, `\gamma`, `\delta`, `\epsilon/\varepsilon`, `\zeta`, `\eta`, `\theta/\vartheta`, `\iota`, `\kappa`, `\lambda`, `\mu`, `\nu`, `\xi`, `o`, `\pi/\varpi`, `\rho/\varrho`, `\sigma/\varsigma`, `\tau`, `\upsilon`, `\phi/\varphi`, `\chi`, `\psi`, `\omega`.

Další zajímavé značky jsou například různé relace nebo operace: $=$, $>$, $<$, $\ne$ (`\ne`), $\le$ (`\le`), $\ge$ (`\ge`), $\sim$ (`\sim`), $\in$ (`\in`), $\subseteq$ (`\subseteq`), $\times$ (`\times`), $\setminus$ (`\setminus`), $\cap$ (`\cap`), $\cup$ (`\cup`), $\vee$ (`\vee`, `\lor`), $\wedge$ (`\wedge`, `\land`), ...

Závorky se píšou svými standardními znaky s výjimkou složených závorek, zapisovaných `\{` a `\}`. Se závorkami jde dělat spousta zajímavých věcí, ale to necháme na nějakou příští sérii, pokud bude místo.

Taktéž některé zajímavé funkce mají své řídící sekvence, například sin, cos, log, min nebo max: `\sin`, `\cos`,... Porovnejte: $\sin x \ne sinx$ (`$\sin x \ne sin x$`).

Poslední, co se dnes naučíme, budou zlomky. Práce se zlomky je jednoduchá, slouží k tomu sekvence `\over`, kterou zapíšeme mezi čitatele a jmenovatele. Všechno, co je vlevo, je čitatel; napravo je jmenovatel.

`$a + b \over c + d$`	$\frac{a+b}{c+d}$
`$a + {b \over c} + d$`	$a+\frac{b}{c}+d$
`${a \over b} + {c \over d} \over {e \over f + g}$`	$\frac{\frac{a}{b}+\frac{c}{d}}{\frac{e}{f+g}}$

Pokud potřebujete vysázet vzorec na samostatnou řádku, obalte jej mezi dvojité dolary: `$$a + b + c$$` se vysází takto:

$$a+b+c$$

V tomto módu se některé konstrukce sází jinak, protože mají víc místa. Vyzkoušejte si například odmocniny, zlomky nebo $\sum$ (`\sum`) a $\prod$ (`\prod`).

Poznámky:

- V matematickém módu píšeme i samostatně stojící proměnné apod. Například „proměnná x je nezávislá na funkci f při libovolné volbě parametrů a, b, c“.
- Vlnovky využijeme i zde: `proměnná~$x$` nebo `funkce~$f$`, případně `$y$~je`. Jednopísmenné vzorce, čísla apod., to všechno by mělo být přivlnovkované k nejtěsněji souvisejícímu slovu.
- Obsáhlý seznam všech značek naleznete v TEXbooku [5] na stranách 434 až 439.
- Název programu (TEX) se vysází jako `\TeX`.

[4] Ty doby byly ve skutečnosti nejen dřevní/dřevěné, ale i olověné. Kdysi se totiž sázelo ručně, uchycovala se olověná písmenka do dřevěných rámů. Vysázet tehdy kvalitně matematiku uměl málokdo a obvykle si za to nechal mastně zaplatit. Kvalita ovšem odpovídala ceně. Tehdejšímu umění ručních sazečů se dnešní počítačová sazba ani zdaleka nevyrovná, natož pak třeba Word nebo LibreOffice.

[5] Donald Ervin Knuth: The TEXbook (Reading, Massachusetts: Addison-Wesley, 1984), ISBN 978-0-201-13448-9

Úkol 3 [5b]: Vymyslete, jak vysázet tento vzorec, a dodejte zdrojový kód:

$$\frac{a^{(b-2d_c)^3}}{\sqrt{2a^{3b_1}}} + \begin{matrix}8\\7\\6\\5\\4\\3\\2\\1\end{matrix} - \sqrt{1+\sqrt{1-\sqrt{1+\sqrt{1-\sqrt{1+\frac{1-\frac{1-\frac{1-x}{1+x}}{1+\frac{1-x}{1+x}}}{1+\frac{1-\frac{1-x}{1+x}}{1+\frac{1-x}{1+x}}}}}}}}$$

Pokud se vám nebude dařit jej zkonstruovat celý, vymyslete aspoň část, budou za to také nějaké body.

Úkol 4 [4b]: Vysázejte TEXem řešení jiné úlohy v této sérii a dodejte zdrojový kód. Měli byste znát dost na to, aby z TEXu vypadnul rozumně hezký výstup. Pokud si nebudete vědět s něčím rady, zeptejte se na fóru na našem webu a orgové vám rádi poradí.

Recepty z programátorské kuchařky: Složitost

Časová a paměťová složitost

V této kuchařce se můžete dočíst o základech časové a paměťové složitosti. Po přečtení byste měli být schopni sami rozebrat složitost jednoduchých algoritmů. To se hodí třeba při návrhu algoritmů a řešení algoritmických úloh, které můžete potkat například v KSP.

Nejdříve si ujasníme, co to ta složitost vlastně je, a ukážeme si pár příkladů. Pak si řekneme, s jakou přesností budeme složitost chtít určovat, a zavedeme si asymptotickou složitost. Na závěr si ukážeme běžné třídy složitosti.

Základní přehled

Pokud řešíme nějakou programátorskou úlohu, často nás napadne více různých řešení a potřebujeme se rozhodnout, které z nich je „nejlepší“. Abychom to mohli posoudit, potřebujeme si zavést měřítka, podle kterých budeme různé algoritmy porovnávat. Nás u každého algoritmu budou zajímat dvě vlastnosti: čas, po který algoritmus běží, a paměť, kterou při tom spotřebuje.

Čas nebudeme měřit v sekundách (protože stejný program na různých počítačích běží rozdílnou dobu), ale v počtu provedených operací. Pro jednoduchost budeme předpokládat, že aritmetické operace, přiřazování, porovnávání apod. nás stojí jednotkový čas. Ona to není úplná pravda, tyto operace se ve skutečnosti přeloží na procesorové instrukce, které se teprve zpracovávají. Ale nám postačí vědět, že těch instrukcí bude vždy konstantní počet. A později se dozvíme, proč nám na takové konstantě nezáleží.

Množství použité paměti můžeme zjistit tak, že prostě spočítáme, kolik bytů paměti náš program použil. Nám obvykle bude stačit menší přesnost, takže všechna čísla budeme považovat za stejně velká a velikost jednoho prohlásíme za jednotku prostoru.

Jak čas, tak paměť se obvykle liší podle toho, jaký vstup náš program zrovna dostal – na velké vstupy spotřebuje více času i paměti než na ty malé. Budeme proto oba parametry určovat v závislosti na velikosti vstupu a hledat funkci, která nám tuto závislost popíše. Takové funkci se odborně říká *časová (případně paměťová, někdy též prostorová) složitost* algoritmu/programu.

Nyní si na příkladu ukážeme, jak se časová a paměťová složitost dá určovat intuitivně, a pak si vše podrobně vysvětlíme.

Představme si, že máme danou posloupnost N celých čísel, ze které chceme vybrat maximum. Použijeme algoritmus, který za maximum prohlásí nejprve první číslo posloupnosti. Pak toto maximum postupně porovnává s dalšími čísly v posloupnosti, a pokud je některé větší, učiní z něj nové maximum. Zapsat bychom to mohli třeba takto:

```
posl[1...N] = vstup
max = posl[1]
Pro i = 2 až N:
   Jestliže posl[i] > max:
      max = posl[i]
Vypiš max
```

Není těžké nahlédnout, že algoritmus provede maximálně $N-1$ porovnání. Intuitivně časová složitost bude lineárně záviset na N, protože porovnání dvou čísel nám zabere „jednotkový čas“, a paměťová složitost bude také na N záviset lineárně, protože si každé číslo z posloupnosti budeme uchovávat v paměti.

Pokud bychom si nepamatovali celou posloupnost, ale vždy jen poslední přečtený člen, stačilo by nám jen konstantně mnoho proměnných, takže paměťová složitost by klesla na konstantní (nezávislou na N) a časová by zůstala stejná.

Jiný příklad: Mějme dané číslo K. Naším úkolem je vypsat tabulku všech násobků čísel od 1 do K:

```
Pro i = 1 až K:
   Pro j = 1 až K:
      Vypiš i*j a mezeru
   Přejdi na nový řádek
```

Tabulka má velikost K^2 a na každém jejím políčku strávíme jen konstantní čas. Proto časová složitost bude záviset na čísle K kvadraticky, tedy bude K^2. Paměťová složitost bude buď konstantní, pokud hodnoty budeme jen vypisovat, anebo kvadratická, pokud si tabulku budeme ukládat do paměti. Můžeme si také všimnout, že tabulku nemusíme vypisovat celou, ale bude nám stačit jen její dolní trojúhelníková část – i tak budeme muset spočítat $(K \cdot K - K)/2 + K = K^2/2 + K/2$ hodnot, což je stále řádově kvadratické vzhledem ke K.

U výběru algoritmu tedy bereme v potaz čas a paměť. Který z těchto faktorů je pro nás důležitější, se musíme rozhodnout vždy u konkrétního příkladu. Často také platí, že čím více času se snažíme ušetřit, tím více paměti nás to pak stojí. To kvůli chytré reprezentaci dat v paměti a různým vyhledávacím strukturám, o kterých se můžete dočíst v našich dalších kuchařkách.

Nás u valné většiny algoritmů bude nejdříve zajímat časová složitost a až poté složitost paměťová. Paměti mají totiž dnešní počítače dost, a tak se málokdy stane, že vymyslíme algoritmus, který má dokonalý čas, ale nestačí nám na něj paměť. Ale přesto doporučujeme dávat si na paměťová omezení pozor.

Než se pustíme do podrobnějšího vysvětlování, ještě si ukážeme tzv. „metodu kouknu a vidím“, kterou můžeme použít na určování časové složitosti u těch nejjednodušších

algoritmů. Spočívá jen v tom, že se podíváme, kolik nejvíc obsahuje náš program vnořených cyklů. Řekněme, že jich je k a že každý běží od 1 do N. Potom za časovou složitost prohlásíme N^k.

Vzhledem k čemu budeme složitosti určovat?

Složitosti obvykle určujeme vzhledem k velikosti vstupu (počet čísel, případně znaků na vstupu). Tento počet si označme N. Časovou i paměťovou složitost pak vyjádříme vzhledem k tomuto N. To je vidět třeba na výběru maxima v předchozím textu.

Pokud by existovalo několik vstupů stejné velikosti, pro které náš algoritmus běží různě dlouho, bude časová složitost popisovat ten nejhorší z nich (takový, na kterém algoritmus poběží nejpomaleji). Stejně tak pro paměťovou složitost použijeme ten ze vstupů délky N, na který spotřebujeme nejvíce paměti. Dostaneme tzv. složitosti v nejhorším případě. Podrobněji si o tom povíme později.

Někdy se nám hodí určit složitost v závislosti na více než jedné proměnné. Pokud bychom například chtěli vypisovat všechny dvojice podstatného a přídavného jména ze zadaného slovníku, strávíme tím čas, který bude záviset nejen na celkové velikosti slovníku, ale i na tom, kolik obsahuje podstatných a kolik přídavných jmen. Rozmyslete si, jaká složitost vyjde, pokud víte, že velikost slovníku je S, podstatných jmen je A a přidavných jmen B.

Častým příkladem, kde si velikost vstupu potřebujeme rozdělit do více proměnných, jsou algoritmy pracující s grafy (viz grafová kuchařka).[6] V případě grafů obvykle vyjadřujeme složitost pomocí proměnných N a M, kde N je počet vrcholů grafu a M je počet jeho hran. I pro více proměnných vybíráme nejhorší případ.

Ne vždy ale určujeme složitosti v závislosti na velikosti vstupů. Například pokud je velikost vstupu konstantní, složitost určíme vzhledem k hodnotám proměnných na vstupu. Třeba u příkladu s tabulkou násobků jsme složitost určili vzhledem k velikosti tabulky, kterou jsme dostali na vstupu. Jiným příkladem může být vypsání všech prvočísel menších než dané N.

Asymptotická složitost

V této části textu se budeme věnovat pouze časové složitosti. Všechna pravidla, která si řekneme, pak budou platit i pro paměťovou složitost.

U určování časové složitosti nás bude především zajímat, jak se algoritmy chovají pro velké vstupy. Mějme například algoritmus A o časové složitosti $4N$ a algoritmus B o složitosti N^2. Tehdy je sice pro $N = 1, 2, 3$ algoritmus B rychlejší než A, ale pro všechna větší N ho už algoritmus A předběhne. Takže pokud bychom si měli mezi těmito algoritmy zvolit, vybereme si algoritmus A.

U složitosti nás obvykle nebude zajímat, jak se chová na malých vstupech, protože na těch je rychlý téměř každý algoritmus. Rozhodující pro nás bude složitost na maximálních vstupech (pokud nějaké omezení existuje) anebo složitost pro „hodně velké vstupy“. Proto si zavedeme tzv. **asymptotickou časovou složitost.**

Představme si, že máme algoritmus se složitostí $n^2/4+6n+12$. Pod asymptotikou si můžeme představit, že nás zajímá jen nejvýznamnější člen výrazu, podle kterého se pak pro velké vstupy chová celý výraz. To znamená, že:

- Konstanty u jednotlivých členů můžeme škrtnout (např. $6n$ se chová podobně jako n). Tím dostáváme $n^2 + n + 1$.
- Pro velká n je $n+1$ oproti n^2 nevýznamné, tak ho můžeme také škrtnout. Dostáváme tak složitost n^2. Obecně škrtáme všechny členy, které jsou pro dost velké n menší než nějaký neškrtnutý člen.

Tahle pravidla sice většinou fungují, ale škrtat ve výpočtech přece nemůžeme jen tak. Proto si nyní zavedeme operátor $\mathcal{O}$ (velké O), díky kterému budeme umět popsat, co přesně naše „škrtání“ znamená, a používat ho korektně.

Definice: Mějme funkce $f : \mathbb{N} \to \mathbb{R}^+$ a $g : \mathbb{N} \to \mathbb{R}^+$. Řekneme, že $f \in \mathcal{O}(g)$, pokud $\exists n_0 \in \mathbb{N}$ a $\exists c \in \mathbb{R}^+$ tak, že $\forall n \geq n_0$ platí $f(n) \leq c \cdot g(n)$.

Nyní slovy: Mějme funkce f a g funkce z přirozených do kladných reálných čísel. Řekneme, že funkce f patří do třídy $\mathcal{O}(g)$, pokud existují konstanty n_0 a c takové, že f je pro dost velká n (totiž pro $n \geq n_0$) menší než $c \cdot g(n)$.

Někdy také píšeme, že $f = \mathcal{O}(g)$ nebo říkáme, že program má složitost $\mathcal{O}(f)$.

A zde je použití: $n^2/4+6n+12 \in \mathcal{O}(n^2)$, protože například pro $c = 10$ platí pro všechna $n > 1$ (tedy $n_0 = 2$):

$$n^2/4 + 6n + 12 \leq 10n^2.$$

Pokud vám tento způsob nevyhovuje a více se vám líbí metoda pomocí „škrtání“, tak ji klidně používejte, akorát všude pište $\mathcal{O}(\ldots)$. Někdy také říkáme, že se konstanty a méně významné členy v $\mathcal{O}$ ztrácí.

Ještě poznamenejme, že operátor $\mathcal{O}(\ldots)$ znamená asymptotický horní odhad funkce. Takže pokud funkce patří do $\mathcal{O}(N)$, tak pak patří i do $\mathcal{O}(N^2)$, $\mathcal{O}(N^3)$, ...

Nejhorší a průměrný případ

Opět si vše vysvětlíme jen na časové složitosti.

Velká část algoritmů běží pro různé vstupy stejné velikosti různou dobu. U takových algoritmů pak můžeme rozlišovat složitost v nejhorším případě (tu už známe), v nejlepším případě a třeba i průměrnou časovou složitost.

Vše si ukážeme na algoritmu BubbleSort (bublinkovém třídění), o kterém se můžete dočíst v kuchařce o třídících algoritmech.[7] Funguje tak, že se dívá na všechny dvojice sousedních prvků, a kdykoliv je dvojice ve špatném pořadí, tak ji prohodí. Zde je pseudokód algoritmu:

```
BubbleSort(pole, N):
  Opakuj:
    setříděno = 1
    Pro i = 1 až N-1:
      Jestliže pole[i] > pole[i+1]:
        p = pole[i]
        pole[i] = pole[i+1]
        pole[i+1] = p
        setříděno = 0
  Skonči, až bude setříděno = 1
```

Časová složitost v nejhorším případě činí $\mathcal{O}(N^2)$ – v každém průchodu vnějším cyklem nám totiž největší hodnota „probublá“ na konec a ostatní se posunou o jednu pozici doleva. Rozmyslete si, proč. Průchodů je proto nejvýše $N-1$ a každý z nich trvá $\mathcal{O}(N)$. Tento nejhorší případ může doopravdy

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

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

nastat, pokud necháme setřídit klesající posloupnost. Tam provedeme přesně $N - 1$ průchodů.

Naopak v nejlepším případě bude časová složitost pouze $\mathcal{O}(N)$. To nastane, pokud na vstupu dostaneme už setříděnou posloupnost. U té algoritmus pouze zkontroluje všechny dvojice a pak se ihned zastaví.

Průměrná časová složitost nám udává, jak dlouho náš algoritmus běží průměrně. Co to ale znamená, není snadné definovat ani spočítat. U třídícího algoritmu bychom mohli počítat průměr přes všechny možnosti, jak mohou být prvky na vstupu zamíchané (tedy přes všechny jejich permutace). To nám někdy může dát přesnější odhad chování algoritmu.

Zrovna u BubbleSortu a mnoha jiných algoritmů vyjde průměrná časová složitost stejně jako složitost v nejhorším případě. Jedním z nejznámějších příkladů algoritmu, který je v průměru asymptoticky lepší, je třídící algoritmus QuickSort (opět viz třídicí kuchařka). Jeho průměrná časová složitost činí $\mathcal{O}(N \cdot \log N)$, zatímco v nejhorším případě může běžet až kvadraticky dlouho.

Často používané složitosti

Na závěr si ukážeme často se vyskytující časové složitosti algoritmů (ty paměťové jsou obdobné). Seřadili jsme je od nejrychlejších a ke každé připsali příklad algoritmu.

$\mathcal{O}(1)$ – *konstantní* (třeba zjištění, jestli je číslo sudé)

$\mathcal{O}(\log N)$ – *logaritmická* (binární vyhledávání); všimněte si, že na základu logaritmu nezáleží, protože platí $\log_a n = \log_b n / \log_b a$, takže logaritmy o různých základech se liší jen konstanta-krát, což se „schová do $\mathcal{O}$-čka“.

$\mathcal{O}(N)$ – *lineární* (hledání maxima z N čísel)

$\mathcal{O}(N \cdot \log N)$ – *lineárně-logaritmická* (nejlepší algoritmy na třídění pomocí porovnávání)

$\mathcal{O}(N^2)$ – *kvadratická* (BubbleSort)

$\mathcal{O}(N^3)$ – *kubická* (násobení matic podle definice)

$\mathcal{O}(2^N)$ – *exponenciální* (nalezení všech posloupností délky N složených z nul a jedniček; pokud je chceme i vypsat, dostaneme $\mathcal{O}(N \cdot 2^N)$)

$\mathcal{O}(N!)$ – *faktoriálová*, $N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$ (nalezení všech permutací N prvků, tedy například všech přesmyček slova o N různých písmenech)

Složitosti ještě často rozdělujeme na *polynomiální* a *nepolynomiální*. Polynomiální říkáme těm, které patří do $\mathcal{O}(N^k)$ pro nějaké k. Naopak nepolynomiální jsou ty, pro něž žádné takové k neexistuje.

Do polynomiálních algoritmů patří například i algoritmus se složitostí $\mathcal{O}(\log N)$. A to proto, že $\mathcal{O}(\log N) \subset \mathcal{O}(N)$ (každý algoritmus, který seběhne v čase $\mathcal{O}(\log N)$, seběhne i v $\mathcal{O}(N)$).

Nepolynomiální jsou z naší tabulky třídy $\mathcal{O}(2^N)$ a $\mathcal{O}(N!)$. Takové algoritmy jsou extrémně pomalé a snažíme se jim co nejvíce vyhýbat.

Pro představu o tom, jak se složitost projevuje na opravdovém počítači, se podíváme, jak dlouho poběží algoritmy na počítači, který provede 10^9 (miliardu) operací za sekundu. Tento počítač je srovnatelný s těmi, které dnes běžně používáme. Podívejme se, jak dlouho na něm poběží algoritmy s následujícími složitostmi:

funkce / $n =$	10	20	50	100	1 000	10^6
$\log_2 n$	3,3 ns	4,3 ns	4,9 ns	6,6 ns	10,0 ns	19,9 ns
n	10 ns	20 ns	30 ns	100 ns	1 µs	1 ms
$n \cdot \log_2 n$	33 ns	86 ns	282 ns	664 ns	10 µs	20 ms
n^2	100 ns	400 ns	900 ns	100 µs	1 ms	1 000 s
n^3	1 µs	8 µs	27 µs	1 ms	1 s	10^9 s
2^n	1 µs	1 ms	1 s	10^{21} s	10^{292} s	$\approx \infty$
$n!$	3 ms	10^9 s	10^{23} s	10^{149} s	10^{2558} s	$\approx \infty$

Pro představu: 1 000 s je asi tak čtvrt hodiny, 1 000 000 s je necelých 12 dní, 10^9 s je 31 let a 10^{18} s je asi tak stáří Vesmíru. Takže nepolynomiální algoritmy začnou být velmi brzy nepoužitelné.

Dnešní menu servírovali

Karel Tesař a Martin Mareš

Jak řešit úlohy – Často kladené dotazy

„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 letos celý jeden díl receptů z programátorské kuchařky, který najdete hned před tímto textem.

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.

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“.