Radek Pelánek

Programátorská cvičebnice

Algoritmy v příkladech

computer press

Radek Pelánek

Programátorská cvičebnice

Computer Press
Brno
2012

Programátorská cvičebnice

Radek Pelánek

Obálka: Martin Sodomka
Odpovědný redaktor: Martin Herodek
Technický redaktor: Jiří Matoušek

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-3751-2

Vydalo nakladatelství Computer Press v Brně roku 2012 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 16 440.

1. vydání

ALBATROS MEDIA a.s.

Obsah

Předmluva

Tato kniha vychází z mých zkušeností s programováním v několika rolích. V roli studenta jsem za svůj „programátorský život“ prošel mnoha programovacími jazyky. Jako nejlepší způsob učení programovacího jazyka mi vždy přišla metoda „skočit do vody a plavat“, tedy vzít si zajímavý, přiměřeně náročný problém, ten zkusit vyřešit a potřebné věci se učit za běhu. Často mi přišlo náročnější vymyslet si zajímavé zadání, než najít řešení konkrétních technických problémů, na které jsem pak při řešení narazil.

Jako učitel mám podobnou zkušenost. S dílčími technickými problémy si většinou studenti zvládnou poradit, důležité je především předložit jim vhodný problém – problém, který pro ně bude adekvátně obtížný a současně bude nějakým způsobem zajímavý, takže je bude bavit. Podobně je to i v programátorských soutěžích, kterých jsem se mnohokrát účastnil v roli soutěžícího i organizátora. Zajímavou soutěž dělají dobře vybrané problémy.

Ve všech těchto rolích jsem se setkal s celou řadou zajímavých příkladů, ale vždy, když nějaký potřebuji, musím složitě vzpomínat nebo hledat. Dobrých učebnic programování existuje spousta, ale většinou kladou důraz na konkrétní jazyk nebo detailní rozbor dílčích algoritmů, příkladů v nich bývá jen pár. Rozsáhlá sbírka zajímavých příkladů mi vždy chyběla. Proto jsem se pokusil ji vytvořit a pevně věřím, že bude užitečná nejen pro mě.

Kniha přímo či nepřímo čerpá z velkého množství zdrojů a zkušeností. Kromě těch literárních, které jsou uvedeny v seznamu literatury, jde hlavně o programátorské soutěže a výuku. Velkou měrou čerpám ze zkušeností s programátorskými soutěžemi a z příkladů, které jsem na těchto soutěžích poznal či pro ně navrhl. Konkrétně jde o soutěž ACM ICPC a její česko-slovenskou verzi CTU Open, programátorskou olympiádu, korespondenční semináře z programování a fakultní soutěž FIbot.

Cenné zkušenosti jsem získal při výuce programátorských předmětů na Fakultě informatiky MU, především pak při vedení předmětu IV104 Seminář řešení programátorských úloh. Na studentech tohoto předmětu jsem mnoho

zde uvedených příkladů testoval a jejich pozorováním jsem získal zkušenosti o obtížnosti a pedagogické vhodnosti jednotlivých úloh.

Tato kniha také částečně vychází k mé předchozí knihy „Jak to vyřešit?“, která se zabývá logickými úlohami a jejich využitím při výuce informatiky. Část příkladů uvedených zde v kapitolách o logických úlohách a hrách vychází z této dřívější knihy.

Chtěl bych tedy poděkovat všem, kdo organizovali uvedené soutěže, účastnili se mé výuky či pomáhali s přípravou předchozí knihy, za jejich přínos pro tuto knihu, byť nevědomý a nepřímý. Konkrétně bych chtěl poděkovat Petrovi Jaruškovi a Janovi Ryglovi za komentáře k dílčím cvičením. Martin Herodek z nakladatelství Computer Press pomohl knihu vylepšit po obsahové i typografické stránce. Poděkování také patří mojí ženě Barče za její vytrvalou podporu nejen při psaní.

Radek Pelánek

1 O knize

Mysl není nádoba, kterou je potřeba naplnit, ale oheň, který je potřeba zapálit.

Plutarchos

Aby se člověk naučil programovat, musí se naučit základní příkazy a postupy (trochu mysl *naplnit*), především však potřebuje hodně trénovat, zkoušet a procvičovat. Aby u toho trénování člověk vydržel, měl by být *zapálený*. Jenže na čem trénovat programování, aby to bylo zajímavé? Na programování rozsáhlých, prakticky užitečných programů začátečník ještě nemá a učebnicové příklady jsou často nudné, takže u nich člověk nevydrží.

K tomu právě slouží tato kniha – poskytuje náměty na zajímavá programátorská cvičení, sloužící k procvičení a tréninku. Příklady jsou voleny tak, aby byly atraktivní a zajímavé, takže člověka, který má alespoň trochu informatického ducha a nadšení, vtáhnou natolik, že si vydrží s nimi hrát, díky čemuž dobře potrénuje.

Kniha není zaměřena na žádný specifický programovací jazyk. Všechny příklady jsou zvládnutelné ve všech běžně používaných programovacích jazycích, a to většinou s docela základními prvky jazyka, tj. bez použití pokročilých vlastností jazyka a speciálních knihoven. Předpokládá se, že čtenář má k dispozici učebnici konkrétního programovacího jazyka, který používá.

Kniha slouží nejen k procvičení základních programátorských konstrukcí, ale i k procvičení algoritmů. I po této stránce je kniha především cvičebnicí a nikoliv samonosnou učebnicí. Klíčové pojmy jsou v knize stručně popsány a vysvětleny, nicméně tento popis je míněn jako připomenutí pojmů, které již čtenář slyšel, resp. inspirace pro to, co by si měl dostudovat. Opět se předpokládá, že čtenář má k dispozici učebnici algoritmizace (např. Skiena, 1998; Wroblewski, 2004; Cormen et al., 2001; Kučera, 2009) nebo prošel relevantním odborným kurzem. Pokrytá látka většinou spadá do učiva probíraného v 1. ročníku na informatických vysokých školách.

Kniha obsahuje širokou škálu příkladů od velmi jednoduchých až po myšlenkově i programátorsky náročné projekty. Také dílčí úlohy obsahují podúlohy

s různou složitostí. Čtenář tedy vždy může najít příklad, který odpovídá jeho momentálním schopnostem, náladě, časovým možnostem a tomu, co si chce procvičit. Průběžně, jak se bude zlepšovat, zde navíc najde stále nové výzvy.

Kniha přijde vhod několika skupinám čtenářům:

- studentům, kteří se učí programovat a psát efektivní algoritmy, pro samostudium a trénink,
- učitelům, kteří vyučují programování a hledají náměty na cvičení, zadání domácích úloh a projektů,
- zkušeným programátorům, kteří se učí nový jazyk a chtějí si jej procvičit na příkladech, případně si chtějí procvičit své „programátorské svaly“ na zajímavých příkladech.

1.1 Úlohy a jejich popisky

Problémy, které stojí za útok, prokážou svoji cenu tím, že útok opětují.

P. Erdős

Úlohy jsou rozděleny do několika tematických oblastí, které sdílejí základní rysy. Pro každou úlohu je uveden odhad obtížnosti, stručný styl úlohy, popis zadání, doplňující komentář a částečně poznámky k řešením.

Kniha obsahuje velmi rozmanité typy úloh. Některé jsou přímočaré, v zadání je celkem jasně popsáno, co dělat, stačí to „jen“ implementovat. Jiné jsou nápadové – výsledný program je velmi krátký a nevyžaduje nic složitého, ale je potřeba vhodný nápad, na který nemusí být snadné přijít. U jiných příkladů se zase naopak hodí netriviální teorie či pojmy, ale ty jsou stručně vysvětleny, takže jde hlavně o to popis pochopit, dohledat si v případě potřeby detailnější vysvětlení a zvládnout popis převést do funkčního programu. Základní styl úlohy je vždy stručně shrnut na začátku popisu úlohy.

Kromě slovního popisu stylu je pro snadnější orientaci uvedeno i číselné hodnocení obtížnosti úloh. Toto hodnocení obtížnosti je rozděleno na dva aspekty: *nápad*, tedy jak náročné je přijít na hlavní myšlenku řešení, algoritmus a vhodnou reprezentaci dat, a *kódování*, tedy jak náročné je program napsat a odladit.

Obě kategorie jsou pouze přibližné odhady a slouží především k relativnímu porovnání úloh. Absolutní obtížnost pochopitelně závisí na zkušenostech řešitele. Obě obtížnosti jsou hodnoceny na stupnici 1–5. Význam jednotlivých stupňů pro začátečníka a experta je ilustrován v tabulce 1.1. U většiny úloh je pro nápad i kódování uveden interval, protože úlohy typicky obsahují několik podúloh, jejichž obtížnost se liší. Podúlohy jsou většinou uvedeny v pořadí rostoucí obtížnosti.

Tabulka 1.1: Kategorie obtížnosti

Nápad	Začátečník	Expert
1	vcelku jasné	zřejmé
2	trocha rozmýšlení	rutina
3	těžké	trocha rozmýšlení
4	hranice možností	těžké
5	neřešitelné	hranice možností

Kódování	Začátečník	Expert
1	20 min.	5 min.
2	1 hod.	15 min.
3	půl dne	1 hod.
4	několik dní	2–6 hod.
5	půlroční projekt	intenzivní víkend

Dále pak následuje samotný popis úlohy. *Zadání* je základní popis cílů úlohy. Po přečtení tohoto textu je možné začít řešit, v případě použití ve výuce je tento text určený pro studenty. Následuje *komentář* k úloze, který obsahuje souvislosti a základní rady, jak k problému přistupovat. Komentáře občas prozrazují i základní myšlenky vhodných algoritmů či skryté „finty“, jak k zadání přistoupit. Ambiciózní čtenář by tedy měl úlohu zkusit vyřešit bez čtení komentářů.

Na konci knihy jsou uvedena vybraná *řešení*, která detailněji osvětlují zajímavé body z postupu, ukazují příklady možných programů, případně odpovědi na konkrétní otázky položené v zadání. Jde však opravdu pouze o *vybraná* řešení, rozhodně nejsou rozebrány všechny příklady a varianty. To není žádoucí, některé problémy jsou záměrně ponechány otevřené pro vlastní zkoumání a projekty.

V řešeních jsou uvedeny ukázky kódu v jazyce Python. Jazyk Python byl pro ukázky zvolen proto, že jde o čistý vysokoúrovňový jazyk, který bývá občas označován za „spustitelný pseudokód“. Ukázky kódu by tedy měly být pochopitelné i bez znalosti tohoto konkrétního jazyka. Python je programovací jazyk, který se rozhodně vyplatí naučit, nicméně pro porozumění této knize není jeho znalost nijak zásadní.

1.2 Jak knihu používat

Buď to udělej, nebo ne. Neexistuje žádné „zkusím“.

Yoda ve filmu Impérium vrací úder

Knihu lze využívat různými způsoby: k samostudiu, ve výuce ke krátkým cvičením, k zadávání domácích úloh nebo i rozsáhlých projektů. Následuje několik komentářů a doporučení ke každému stylu.

V případě *samostudia* je důležitá pevná vůle – nespokojit se pouze s jednoduchými příklady a nedívat se hned na řešení. Může být vhodné najít si partnera, se kterým budete příklady řešit souběžně. Můžete tak soutěžit, kdo zvládne úlohy vyřešit rychleji či efektivněji, a také můžete svá řešení porovnávat. U mnoha úloh existuje několik různých řešení a porovnáním více různých řešení se můžete hodně naučit.

U použití ve *výuce*, především pokud programování probíhá v hodině, která má fixní konec, je důležité vybrat příklady správné obtížnosti. Především je důležité, aby obtížnost nebyla příliš vysoká. Pokud studenti v půlce hodiny zjistí, že je nad jejich síly dokončit příklad v dostupném čase, jsou demotivováni, už se příliš nesoustředí a tím pádem i nic moc nenaučí. Přímo ve vyučovací hodině je lepší používat základní varianty příkladů a složitější varianty a rozšíření nechat jako domácí úkol, u kterého není tak krátký časový limit.

U domácích úloh je dnes samozřejmě problém s vyhledáváním na Internetu. Do knihy jsou začleněny úlohy zajímavé a osvědčené, což ovšem znamená, že jde často také o úlohy známé a rozšířené. Řešení mnoha úloh lze tedy často vcelku snadno najít na Internetu (především pokud hledáme anglické verze pojmů). Pokud jsme si tohoto problému vědomi, lze mu vcelku rozumně předcházet. Snadno vyhledatelná řešení většinou nejsou přesně v té podobě, jak je zde zadáno, a pokud navíc úlohu trochu pozměníme či přejmenujeme, často se dá snadnému vyhledání zabránit.

Navíc schopnost „vyhledat si základ řešení a to pak jen vhodně upravit“ není sama o sobě nijak špatná, naopak u praktických aplikací programování to je preferovaný postup. I tuto schopnost může být užitečné trénovat, takže u některých příkladů může být rozumné použití tohoto přístupu podporovat. Konkrétně například u cvičení „Experimenty s řadicími algoritmy“ studenty podporuji v tom, aby si řadicí algoritmy vyhledali a pouze převedli do jednotného stylu a aby se soustředili na experimentální porovnání.

Při použití příkladů jako zadání projektů využívejte zde uvedené zadání pouze jako výchozí bod. Řešitel projektu by měl zkusit vymyslet vlastní rozšíření a variace a zkusit si sám položit zajímavou otázku o úloze.

1.3 Scénáře použití

Čím tvrději pracuji, tím víc se zdá, že mám štěstí.

T. Jefferson

Formát knihy vynucuje lineární řazení příkladů, ovšem možných kritérií, podle kterých lze příklady řadit, je celá řada: například obtížnost, metoda řešení, téma, typ vstupu a výstupu. V této knize je zvoleno tematické řazení příkladů. Příklady, které jsou zařazeny za sebou, tedy sdílejí podobné téma, ale mohou se výrazně lišit v ostatních aspektech, především v obtížnosti. To znamená, že není dobrý nápad řešit příklady v knize čistě sekvenčně.

Pro výběr vhodného příkladu slouží informace uvedené na začátku popisu každého příkladu a dále pak tato a následující kapitola, které udávají přehled příkladů roztříděných podle různých kritérií. V této kapitole jsou příklady roztříděny podle stylu použití, v následující pak podle metod návrhu algoritmů, které se při řešení používají.

Zde uvedené seznamy berte jen jako základní orientaci. To, že příklad není uveden v určité kategorii, ještě neznamená, že by nemohl být pro dané účely vhodný.

Úplní začátečníci

Začneme příklady vhodnými pro úvodní kurz programování, tedy pro ty, kdo neumí vůbec programovat. Příklady jsou řazeny v pořadí, v jakém je vhodné je procházet:

- 4.2 Želví grafika: základy – pokud používáte jazyk, který má snadno použitelnou knihovnu pro interpretaci příkazů želví grafiky (např. Python), je toto cvičení nenáročné a přitom efektní (pomocí pár příkazů jdou kreslit zajímavé obrázky), takže jde o vhodné první cvičení.
- Příklady z kapitoly Počítání s čísly, konkrétně: 3.1 Hrátky s čísly, 3.2 Výpisy posloupností, 3.3 Collatzův problém, 3.4 Náhodná procházka, 3.5 Dělitelnost a prvočísla – u těchto příkladů vystačíme pouze s jednoduchými syntaktickými prvky jazyka (celočíselné proměnné a cykly), příklady jsou také vhodné pro představení konceptu funkce.
- 3.7 Fibonacciho posloupnost – představení jednorozměrného pole, ukázka různých pohledů na problém.
- 4.1 Textová grafika – opět problémy využívající jen základní syntaktické konstrukce, jen více obrázkové a občas vyžadující mírný nápad.
- 5.2 Transpoziční šifry, 5.3 Substituce a kódování – vhodné pro představení řetězců a práce s nimi.

- 7.2 Hádání čísla, 7.3 Oběšenec, 7.5 Hra Nim – vhodné pro představení načítání vstupu od uživatele (případně ze souboru) a vyzkoušení interakce s uživatelem. Příklady také ilustrují zajímavé a přitom vcelku snadno zvládnutelné algoritmy.
- 4.3 Želví grafika: fraktály, 6.3 Hanojské věže – představení konceptu rekurze.
- 6.1 Číselné bludiště, 6.5 Hledání cest v bludišti – práce s dvojrozměrným polem, představení základních grafových algoritmů (prohledávání do šířky).
- 8.4 Experimenty s řadicími algoritmy – práce s polem, ilustrace klasických algoritmů.

Příklady z kapitoly 3 Počítání s čísly jsou realizovatelné třeba i v tabulkovém editoru a mohou tak posloužit jako základní úvod do algoritmizace i v případě, kdy není ve výuce čas na probrání obecného programovacího jazyka.

Trénink algoritmizace

Následující příklady přijdou vhod, pokud se chceme zaměřit primárně na trénink algoritmů a metod návrhu algoritmů a chceme příklady, které jsou zvládnutelné při dobrém nápadu relativně rychle a pomocí krátkého kódu:

- 3.10 Permutace, kombinace, variace
- 4.7 Konvexní obal
- 4.8 Triangulace
- 5.5 Přesmyčky
- 6.3 Hanojské věže
- 6.6 Generování bludišť
- 7.7 Jednorozměrné piškvorky
- 8.1 Rozměňování mincí
- 8.3 Problémy s řetězci a posloupnostmi
- 8.4 Experimenty s řadicími algoritmy

Podrobnější rozbor různých metod návrhu algoritmů je uveden v následující kapitole.

Učení nového jazyka

Pro ty, kdo už umí programovat, pouze se učí nový programovací jazyk a chtějí jej natrénovat na zajímavých příkladech, se mohou hodit následující příklady:

- 3.9 Výpočet Pí – zajímavé experimenty, které přitom vyžadují jen manipulaci s číselnými proměnnými (vhodné pro první seznámení s jazykem).

- 6.1 Číselné bludiště – jednoduchá logická úloha pro procvičení práce s dvojrozměrným polem.
- 5.2 Transpoziční šifry, 5.3 Substituce a kódování – procvičení základní práce s řetězci a poli.
- 7.6 Tetris (interaktivní verze) – komplexní procvičení mnoha aspektů jazyka (interakce s uživatelem, reprezentace dat, čas), přičemž celkově je program docela krátký a výsledek relativně efektní.

Výzvy pro zkušené programátory

Pro zkušené programátory, kteří se nechtějí učit nový jazyk nebo algoritmus, ale chtějí výzvu svých schopností, zkusit si něco „pro radost z programování“ nebo potrénovat na programátorskou soutěž, se mohou hodit následující příklady.

- Výše uvedené příklady na trénink algoritmizace.
- Příklady z kapitoly 8 Klasické informatické problémy, a to nejlépe bez čtení doprovodného komentáře.
- Úlohy 3.10 Permutace, kombinace, variace, 4.3 Želví grafika: fraktály, 4.4 Sierpińského fraktál. Tyto úlohy mají krátké elegantní řešení, ale není úplně snadné je vymyslet.
- Hry a logické úlohy, konkrétně např. 6.11 Sokoban, 7.6 Tetris, 7.8 Piškvorky. Základní verze her v textovém režimu lze napsat docela rychle, také heuristiky pro umělou inteligenci mohou být s dobrým nápadem krátké, poskytují prostor pro soutěžení o to, kdo napíše nejúspěšnější program, a také dávají široký prostor pro rozšíření.
- 5.4 Rozlomení šifer – tato úloha obsahuje otevřenou a zajímavou výzvu napsat program tak, aby byl co nejúspěšnější v lámání šifer.

Zkušení programátoři si pak také mohou úlohy rozšířit tím, že překročí rámec psaní klasických sekvenčních programů využitím paralelizace u výpočetně náročných problémů, např. 6.11 Sokoban, 6.8 Jak navštívit všechna pole mřížky? nebo 4.6 Mandelbrotova množina s velkou přesností. Můžete zkusit co nejefektivněji využít vícejádrové procesory nebo provést výpočet v distribuovaném prostředí. Jiným způsobem rozšíření zadání může být realizace úloh formou interaktivní webové aplikace nebo aplikace pro mobilní telefon – pro tento styl jsou vhodné především hry a logické úlohy, ale zajímavé mohou být i některé geometrické problémy a úlohy s obrázky.

Projekty

Jako témata semestrálních (ročníkových) projektů, případně i jako inspirace pro témata bakalářských či diplomových prací, se mohou hodit následující příklady:

- Šifrovací systém – kombinace témat z kapitoly 5 Šifrování a práce s textem. Program dostane text a bude ho umět zašifrovat a dešifrovat různými způsoby.
- Úloha 6.6 Generování bludišť a generování zadání u dalších logických úloh.
- Složitější logické úlohy a hry: 6.9 Polyomina, 6.10 Sudoku, 6.11 Sokoban, 7.6 Tetris, 7.8 Piškvorky.
- 7.9 Souboje virtuálních robotů.
- 8.6 Grafové algoritmy.
- Problémy s experimentální složkou, jako jsou 8.4 Experimenty s řadicími algoritmy.
- Náměty z kapitoly 9 Další náměty.

2 Přehled pojmů

Do průšvihu nás nikdy nedostane to, co nevíme. Dostane nás tam to, co víme příliš jistě, a ono to tak prostě není.

Y. Berry

Tato kniha slouží jako cvičebnice, nikoliv jako kompletní učebnice. U jednotlivých příkladů jsou většinou hlavní myšlenky vysvětleny, předpokládá se však, že čtenář umí programovat v nějakém programovacím jazyce a zná základní pojmy z informatiky. Pro základní přehled pozadí, na kterém kniha staví, slouží tato kapitola, která podává stručný přehled pojmů použitých v knize. Rozhodně není nezbytné všechny uvedené pojmy ovládat, některé z nich se využijí pouze v několika málo těžších příkladech.

Pro každý pojem je dán stručný popis, který slouží primárně pro připomenutí, nikoliv pro vysvětlení. Pokud se čtenář s některým z pojmů nesetkal, je vhodné si před řešením příkladů souvisejících s daným pojmem téma blíže dostudovat. Výklad zmíněných pojmů lze najít v mnoha učebnicích, viz např. Skiena, 1998; Wroblewski, 2004; Cormen et al., 2001; Kučera, 2009. U mnoha základních pojmů nabízí dobré vysvětlení i Wikipedie.

Ke každému pojmu je uveden i seznam příkladů, ve kterých se daný pojem vyskytuje. Tyto příklady poslouží jako názorná ilustrace uvedeného stručného popisu. Současně lze tuto kapitolu využít i jako zdroj námětů ve chvíli, kdy se učíte nějaký pojem a chcete si jej dobře procvičit.

2.1 Složitost

Dříve než se podíváme na techniky návrhu algoritmů, připomeňme si, jak poměřujeme výkon algoritmů. Přímočarý přístup by byl spustit algoritmy na konkrétním vstupu, změřit jejich rychlost a podle toho určit „vítěze“. Takový přístup má však zřejmé nevýhody: výsledek závisí na počítači, na kterém algoritmy spouštíme, a na volbě konkrétního vstupu. Proto se k porovnání algoritmů většinou používá jiný přístup: neměříme čas běhu na konkrétním

počítači, ale počet operací, které algoritmus provádí, a nezajímá nás délka výpočtu na konkrétním vstupu, ale funkce určující, jak délka výpočtu závisí na velikosti vstupu.

Složitost algoritmu pak vyjadřujeme pomocí $O(f(n))$ notace, která říká, že délka výpočtu algoritmu je funkce f závisející na velikosti vstupu n („až na konstantní násobek“). Následující výčet udává příklady, ve kterých se složitost algoritmů rozebírá, a zmiňuje složitost algoritmů vyskytujících se v řešení příkladu:

- 8.4 Experimenty s řadicími algoritmy – praktické vyzkoušení rozdílu mezi algoritmy s kvadratickou složitostí $O(n^2)$ a složitostí $O(n \log(n))$.
- 7.2 Hádání čísla – typická ilustrace algoritmu s logaritmickou složitostí $O(\log(n))$.
- 4.7 Konvexní obal – algoritmus se složitostí $O(n \log(n))$.
- 8.3 Problémy s řetězci a posloupnostmi – zde máme problémy, kde naivní řešení má exponenciální složitost $O(2^n)$, ale pomocí vhodného algoritmu můžeme dosáhnout kvadratické složitosti $O(n^2)$.
- 5.5 Přesmyčky – příklad užitečného algoritmu s exponenciální složitostí $O(2^n)$ (ve většině případů jsou exponenciální algoritmy pro reálné problémy nepoužitelné).

2.2 Rekurze a metoda rozděl a panuj

Metoda „rozděl a panuj“ je založena na tom, že problém rozdělíme na podčásti, které jsou vzájemně nezávislé a pokud možno mají podobnou velikost, a tyto podčásti vyřešíme samostatně. Na základě řešení podproblémů pak zkonstruujeme řešení kompletního problému. Typickým příkladem použití této metody je řazení posloupnosti pomocí metody „řazení slučováním“ – posloupnost rozdělíme na dva stejně dlouhé úseky, každý z nich samostatně seřadíme a vzniklé dvě seřazené posloupnosti spojíme do výsledné seřazené posloupnosti.

Jak ukazuje tento příklad, algoritmy navržené pomocí metody rozděl a panuj typicky vedou na použití rekurze, tj. volání sebe sama – funkce při svém výpočtu volá sebe samu, pouze s jinými hodnotami parametrů. Variací na metodu „rozděl a panuj“ je metoda „převeď na předchozí případ“, nazývaná též „zmenši a panuj“. V tomto případě řešení problému o velikosti n vyjádříme pomocí řešení problému o velikosti $n - 1$. Typickým příkladem použití této metody je problém Hanojských věží.

Cvičení využívající uvedené metody: 7.2 Hádání čísla, 4.3 Želví grafika: fraktály, 4.4 Sierpińského fraktál, 6.3 Hanojské věže, 6.4 Pokrývání mřížky, 3.7 Fibonacciho posloupnost (příklad nevhodného použití rekurze), 8.5 Vyhodnocování výrazu.

Další standardní problémy, u kterých lze využít tento přístup (nerozebírané detailně v této knize): řadicí algoritmy (quicksort, řazení slučováním), binární vyhledávací strom, hledání dvojice nejbližších bodů, Strassenův algoritmus pro násobení matic, efektivní násobení celých čísel, rychlá Furierova transformace.

2.3 Dynamické programování

Dynamické programování je technika pro řešení problémů, které jdou rozložit na vzájemně se překrývají podproblémy. Hlavní rozdíl oproti výše uvedené metodě rozděl a panuj spočívá v překrývání podproblémů. U metody rozděl a panuj potřebujeme, aby podproblémy byly nezávislé. U dynamického programování naopak stavíme na překryvech podproblémů, přičemž potřebujeme, aby řešení problému šlo vždy vyjádřit jako kombinace řešení menších podproblémů. Řešení pak budujeme „odspodu“. Nejdříve vyřešíme nejmenší podproblémy a pomocí nich pak budujeme řešení rozsáhlejších podproblémů. Program typicky funguje tak, že si definujeme „tabulku“, do které ukládáme dílčí řešení a kterou postupně vyplňujeme od jednoho konce. Alternativně lze dynamické programování implementovat „odvrchu“ za využití rekurze, přičemž si musíme pamatovat výsledky rekurzivních volání, aby zbytečně nedocházelo k opakování výpočtu.

Cvičení, ve kterých se dynamické programování využívá (občas jen v dílčí variantě zadání): 3.7 Fibonacciho posloupnost, 3.8 Pascalův trojúhelník, 7.5 Hra Nim, 8.3 Problémy s řetězci a posloupnostmi, 8.1 Rozměňování mincí, 4.8 Triangulace.

Další standardní problémy, u kterých lze využít tento přístup (nerozebírané v této knize): Floydův-Warshallův algoritmus pro hledání nejkratší cesty mezi všemi páry vrcholů v grafu, optimální uzávorkování při násobení matic.

2.4 Hladové algoritmy

Hladové algoritmy budují řešení v jednotlivých krocích a v každém kroku dělají „lokálně optimální rozhodnutí“, tj. rozhodnutí, které se jeví jako optimální na základě aktuální situace. Takové algoritmy bývají jednoduché na programování a mají dobrou výpočetní složitost. Nicméně jen málokdy vedou k optimálnímu řešení. I pokud nevedou k optimálnímu řešení, mohou představovat dobrou základní heuristiku, takže často se vyplatí začít hladovým algoritmem, pomocí něj získat intuici o problému, a pak teprve začít vymýšlet vylepšení.

Cvičení související s uvedenými pojmy: 8.1 Rozměňování mincí (klasická ilustrace použití hladového algoritmu a toho, kdy nefunguje), 4.8 Triangulace, 7.6 Tetris (základní heuristiku pro „umělou inteligenci“ pro tuto hru lze realizovat hladovým algoritmem), částečně 7.3 Oběšenec.

Další standardní problémy, u kterých lze využít tento přístup (nerozebírané detailně v této knize): Huffmanovo kódování, Primův a Kruskalův algoritmus pro hledání kostry grafu, Dijkstrův algoritmus pro hledání nejkratší cesty v grafu.

2.5 Hrubá síla a heuristiky

Hrubá síla je přímočarý přístup k řešení problémů – prostě zkoušíme všechny možné kombinace a ověřujeme, zda řeší problém. Přímočaré použití hrubé síly je tedy většinou neefektivní a použitelné pouze pro malé problémy. Nicméně pokud víme, že vstupní data budou mít malý rozsah, může být po pragmatické stránce použití hrubé síly tím nejlepším řešením, protože implementace hrubé síly je často výrazně jednodušší než implementace složitějších algoritmů. Navíc často ani žádný efektivnější algoritmus neznáme.

Přímočarou hrubou sílu můžeme navíc často drobnými úpravami vylepšit. Základním vylepšením je ořezávání. Na základě částečného řešení může být možné rozhodnout, že daná větev výpočtu je neperspektivní, a ušetřit si tak podstatnou část prohledávání. Tento přístup vede k algoritmu „zpětného prohledávání“ (backtracking), při kterém budujeme částečné řešení a průběžně ho kontrolujeme. Pokud zjistíme, že aktuální částečné řešení nemůže být platným řešením, opustíme aktuální větev a vrátíme se k poslednímu platnému částečnému řešení, které rozvineme novým směrem. Tento přístup se přirozeně implementuje pomocí rekurze.

Další možnost, jak hrubou sílu vylepšit, jsou heuristiky. Heuristika je postup, který často pomůže, ale není zaručeno, že funguje. Například při bloudění bludištěm je jednoduchou heuristikou „preferuj cesty vedoucí přímo k cíli“. Toto pravidlo odpovídá hladovému algoritmu a samo o sobě je pro bloudění v bludišti nedostatečné. Může nám ale pomoct urychlit prohledávání pomocí hrubé síly.

S těmito pojmy souvisí mnoho cvičení. Základní hrubá síla, tj. prosté vyzkoušení všech možností, se využije u úloh 5.4 Rozlomení šifer, 5.5 Přesmyčky a 7.4 Logik. Backtracking se používá u úloh 6.7 Rozmisťování figur na šachovnici, 6.8 Jak navštívit všechna pole mřížky?, 6.9 Polyomina, 6.10 Sudoku. Prohledávání s heuristikou se používá u úloh 6.11 Sokoban, 7.6 Tetris, 7.8 Piškvorky.

2.6 Grafy a stavové prostory

Grafy slouží k zachycení vztahů mezi objekty a jsou klíčovou datovou strukturou v informatice. Graf se skládá z vrcholů a hran, přičemž hrany zachycují vztah mezi dvěma vrcholy. Stavový prostor problému je graf, který zachycuje všechny možné konfigurace (vrcholy) a přechody mezi nimi (hrany). Například obrázek 6.3 zachycuje stavový prostor logické úlohy Hanojské věže.

Výklad grafových pojmů lze najít v mnoha učebnicích nebo snadno i na Internetu. Zde pouze zmíníme seznam základních pojmů, které jsou používány v popisech cvičení nebo se hodí při řešení: cesta v grafu, délka cesty, vzdálenost vrcholů, strom, reprezentace grafu v počítači (seznamy následníků, matice sousednosti), prohledávání do šířky, prohledávání do hloubky.

Základní grafové pojmy se vyskytují v cvičeních 6.5 Hledání cest v bludišti, 6.6 Generování bludišť, 6.1 Číselné bludiště a 8.6 Grafové algoritmy. Stavové prostory jsou explicitně uvedeny v cvičeních 6.3 Hanojské věže a 6.11 Sokoban.

2.7 Objektově orientované programování

Objektově orientované programování je programovací paradigma založené na využití objektů – datových struktur zastřešujících data a funkce pracující s těmito daty (metody). Objektově orientované programování je klíčové při programování „ve velkém“. Cvičení uvedená v této knize se zaměřují na programování „v malém“ a na procvičení algoritmizace, takže použití objektů není nezbytné. U některých příkladů však je využití objektů smysluplné a přirozené. V těchto případech je vhodné objekty využívat a procvičit si je. Většinou jde pouze o základní využití objektů, typicky pro zapouzdření a zpřehlednění kódu díky eliminaci globálních proměnných. Pokročilejší prvky, jako je polymorfismus a dědičnost, se většinou nevyužijí.

Vhodné úlohy pro využití objektově orientovaného programování:

- náročnější logické úlohy a hry (přirozené využití objektů pro reprezentaci stavu úlohy): 6.9 Polyomina, 6.11 Sokoban, 6.10 Sudoku, 7.6 Tetris, 8.2 Simulátor hry Život,
- souboje strategií (reprezentace jednotlivých strategií pomocí objektů): 7.1 Kámen, nůžky, papír, 7.8 Piškvorky, 7.9 Souboje virtuálních robotů,
- cvičení na šifrování (5.2, 5.3, 5.4) při zkombinování několika zadání do většího šifrovacího systému,
- většina logických úloh a her při rozšíření zadání o interaktivní grafické uživatelské rozhraní.

2.8 Grafika

Většina příkladů má čistě textové vstupně-výstupní chování, takže není problém je realizovat téměř v jakémkoliv programovacím jazyce. Část příkladů však využívá i grafiku. Realizace grafiky může být pro začínající programátory v některých programovacích jazycích trochu náročná. Bylo by však škoda nechat se odradit, protože příklady s grafickým výstupem jsou atraktivní. Navíc zde zmíněné příklady využívají jen několik základních grafických operací a při vhodném přístupu je lze také docela snadno realizovat v jakémkoliv prostředí.

Nejjednodušší možnost je *textová grafika* – což není plnohodnotná grafika, ale pouze simulace bitmapové grafiky pomocí textového výstupu. Někdy se tento styl grafiky nazývá též „ASCII art“. Takové „obrázky“ můžeme snadno realizovat pomocí základního příkazu pro výpis. Pro vykreslování složitějších obrázků se hodí operace typu „zapiš znak `A` na pozici `x,y`“. V unixových prostředích (a částečně i pod Windows) se za tímto účelem většinou používá knihovna `curses`, která je v různých obměnách dostupná ve více programovacích jazycích.

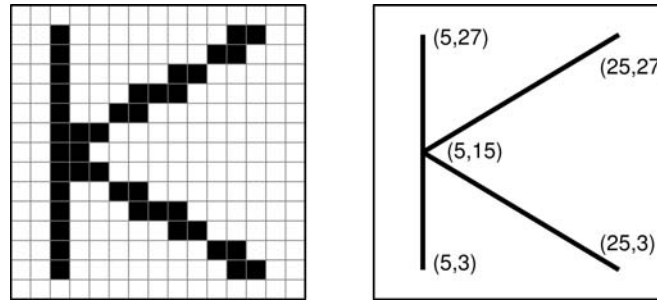
Textová grafika je použita výrazně v příkladech 4.1 Textová grafika, 7.6 Tetris, 8.2 Simulátor hry Život a dále okrajově u většiny logických úloh a her pro vykreslování zadání a řešení úloh.

Grafické formáty, ve kterých pracujeme s jednotlivými body, se nazývají *bitmapová grafika*. Obrázek je zde reprezentován mřížkou $n \times m$ bodů, velikost mřížky udává rozlišení (kvalitu) obrázku. Základní operace v bitmapové grafice je příkaz `putpixel(x,y,c)`, který zakreslí bod barvy `c` na souřadnice `x,y`. Úsečky, kružnice a další geometrické objekty se vykreslují jako posloupnost jednotlivých bodů. Pokud obrázek v bitmapové grafice zvětšujeme, stává se zrnitým.

Pro práci s bitmapovou grafikou nabízí většina programovacích jazyků podporu skrze specializovanou knihovnu. Buď máme k dispozici plátno (canvas), což je samostatné okno, do kterého můžeme vykreslovat, nebo nám knihovna dává přístup k datové struktuře „bitmapový obrázek“, do které můžeme zapisovat body a poté obrázek uložit a prohlédnout si jej v běžném prohlížeči obrázků.

Bitmapová grafika je použita v příkladech: 4.4 Sierpińského fraktál, 4.5 Bitmapová grafika, 4.6 Mandelbrotova množina.

Jiný způsob reprezentace obrázků představuje *vektorová grafika*, ve které je obrázek popsán pomocí geometrických útvarů. Například úsečka je zadána souřadnicemi krajních bodů a nikoliv pevným výčtem bodů, jak je tomu v bitmapové grafice (viz obrázek 2.1). Když vektorový obrázek zvětšujeme, můžeme libovolně zlepšovat přesnost, a obrázek se tedy nestane zrnitým. Pro



Obrázek 2.1: Ilustrace bitmapové (vlevo) a vektorové (vpravo) grafiky

většinu příkladů uvedených v této knize vystačíme se základní operací vektorové grafiky, kterou je vykreslení úsečky.

Podobně jako bitmapovou grafiku, i vektorovou grafiku můžeme ve většině programovacích jazyků realizovat pomocí vhodné specializované knihovny. Alternativní způsob, který je zcela nezávislý na použitém programovacím jazyce, nabízí formát SVG (Scalable Vector Graphics). Jde o textový formát založený na XML, který můžeme snadno generovat programem. Vše, co pro zvládnutí vektorového grafického výstupu potřebujeme, je tedy umět vypisovat text do souboru a nastudovat si základní syntax SVG, která je jednoduchá. Například pomocí následujícího výpisu vytvoříme obrázek obsahující dvě kolmé protínající se úsečky (první je tučná a modrá, druhá tenká a zelená).

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
<line x1="100" y1="0" x2="100" y2="200"
 style="stroke-width:4;stroke:rgb(00,00,99);" />
<line x1="0" y1="100" x2="200" y2="100"
 style="stroke-width:1;stroke:rgb(00,99,00);" />
</svg>
```

K prohlížení SVG souborů lze použít libovolný webový prohlížeč, umí je zobrazit i mnoho grafických programů, pomocí kterých můžeme SVG soubor zkonvertovat do bitmapových formátů. Pro manuální úpravy SVG souborů lze využít volně dostupný editor Inkscape.

Vektorová grafika je použita v příkladech: 4.2 Želví grafika: základy, 4.3 Želví grafika: fraktály, 4.4 Sierpińského fraktál, 4.7 Konvexní obal, 4.8 Triangulace, 6.6 Generování bludišť. Dále ji lze okrajově využít i u dalších logických úloh (např. při generování zadání).

3 Počítání s čísly

Účel výpočtů je vhled, nikoliv čísla.

R. Hamming

Tato kapitola obsahuje náměty na příklady, které mají velmi jednoduché vstupně-výstupní chování: vesměs pouze načtou čísla a na výstup opět dají čísla. Počítání s čísly pro většinu lidí není tak atraktivní jako třeba grafika, logické úlohy nebo hry. Nicméně příklady s čísly jsou vhodné jako úvodní cvičení, protože k nim stačí velmi málo znalostí o syntaxi konkrétního jazyka. Většinou vystačíme s celočíselnými proměnnými, cykly, funkcemi a případně jednorozměrnými poli.

Většina uvedených zadání navíc ilustruje zajímavé vlastnosti čísel či algoritmů, případně se k úlohám vážou zajímavé souvislosti, jako například historické zajímavosti nebo nevyřešené otevřené problémy. Pokud o těchto souvislostech víme, i jednoduché počítání s čísly dostává na zajímavosti.

3.1 Hrátky s čísly

Nápad:	1
Kódování:	1
Styl úlohy:	*Jednoduché cvičení na základní procvičení práce s celočíselnými proměnnými a s cykly.*

Program načte přirozené číslo n a vypíše výsledek výpočtu nad tímto číslem. Náměty na výpočty:

1. součet prvních n přirozených čísel,
2. faktoriál čísla n, tj. součin prvních n čísel,
3. celou část odmocniny z čísla n, tj. největší celé x takové, že $x^2 \leq n$,
4. ciferný součet čísla n,
5. počet jedniček obsažených v čísle n,
6. číslo n zapsané pozpátku,
7. informace o tom, zda n je platné rodné číslo.

Doplňující komentář

Toto cvičení slouží především k procvičení cyklů. Z tréninkových důvodů je užitečné vyřešit stejný úkol několika různými způsoby. V řešeních na konci knihy je například uvedeno 5 možností, jak zapsat řešení prvního úkolu. U příkladů 4.–7. spočívá základní princip v procházení jednotlivých cifer čísla. Toho snadno dosáhneme tak, že číslo postupně dělíme 10 a zkoumáme zbytek po dělení 10 (operace modulo).

3.2 Posloupnosti

Nápad:	1-3
Kódování:	1-2
Styl úlohy:	*Procvičení práce s celočíselnými proměnnými, s cykly a případně poli.*

Odhalte princip následujících posloupností a poté napište pro každou z nich program, který dostane na vstup číslo n a na výstup vypíše prvních n členů dané posloupnosti.

1. 1, 2, 3, 4, 5, 6, ...
2. 1, 3, 5, 7, 9, 11, ...
3. 1, 5, 9, 13, 17, 21, 25, 29, ...
4. 1, 2, 4, 7, 11, 16, ...
5. 1, 2, 4, 8, 16, 32, ...
6. 1, 2, 6, 24, 120, ...
7. 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, ...
8. 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, ...
9. 1, 2, 4, 7, 9, 12, 18, 24, 32, 38, 42, 50, 56, 64, 71, 73, 75, 81, ...
10. 1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, ...
11. 1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9, 9, 9, 10, ...

Doplňující komentář

Principy posloupností jsou následující:

1. přirozená čísla,
2. lichá čísla, tj. aritmetická posloupnost s krokem 2,
3. aritmetická posloupnost s krokem 4,
4. aritmetická posloupnost 2. stupně – rozdíly mezi členy tvoří aritmetickou posloupnost s krokem 1, tj. přirozená čísla,
5. mocniny dvojky, tj. geometrická posloupnost s krokem 2,

6. faktoriály,
7. úseky přirozených čísel postupně se prodlužující délky,
8. prvočísla,
9. další člen vznikne tak, že k aktuálnímu členu přičteme počet jeho dělitelů,
10. popis předchozího členu, další člen dostaneme tak, že předchozí člen „přečteme nahlas“: „jedna jednička“, „dvě jedničky“, „jedna dvojka, jedna jednička“,
11. Golombova sebe-popisující posloupnost, neklesající řada kladných čísel, jejíž n-tý člen řady udává, kolikrát se opakuje číslo n.

Prvních 6 posloupností lze vyřešit jednoduše pomocí jednoho `for` cyklu a několika celočíselných proměnných, další 3 posloupnosti vyžadují vnořené cykly nebo použití pomocné funkce. Z uvedených posloupností jsou nejnáročnější dvě poslední, i když i u nich je náročné přijít především na princip posloupnosti, programátorsky jsou vcelku přímočaré. V případě posloupnosti s popisem předchozího členu může být výhodnější pracovat s členy jako řetězci spíše než jako s celými čísly. Golombova sebe-popisující řada vyžaduje použití jednorozměrného pole.

Mnoho dalších zajímavých námětů na zadání lze najít v internetové encyklopedii celočíselných posloupností (The On-Line Encyclopedia of Integer Sequences, *oeis.org*). V této encyklopedii je uvedena také řada doplňujících komentářů a zajímavostí.

Zajímavým rozšířením této úlohy je automatické doplňování řad – implementace „umělé inteligence“, která odhalí princip posloupnosti a automaticky ji doplní. Toto je reálné jen pro relativně jednoduché posloupnosti,jako je prvních 5 uvedených posloupností. I tak tato varianta představuje úkol obtížnosti 4–5.

3.3 Collatzův problém

Nápad:	1
Kódování:	1
Styl úlohy:	*Programátorsky jednoduché cvičení, které ilustruje velmi zajímavý problém z teorie čísel.*

Předmětem tohoto cvičení je matematický problém, který je znám pod mnoha různými názvy – kromě názvu Collatzův problém se můžete setkat též s označením $3n+1$ problém, Ulamův problém, Syrakuský problém a spoustou dalších jmen. Problém se týká následujícího postupu: „vezmi přirozené číslo, pokud je sudé, vyděl jej dvěma, pokud je liché, vynásob jej třemi a přičti jedničku;

tento postup opakuj, dokud nedostaneš číslo jedna“. Program tedy vypadá následovně:

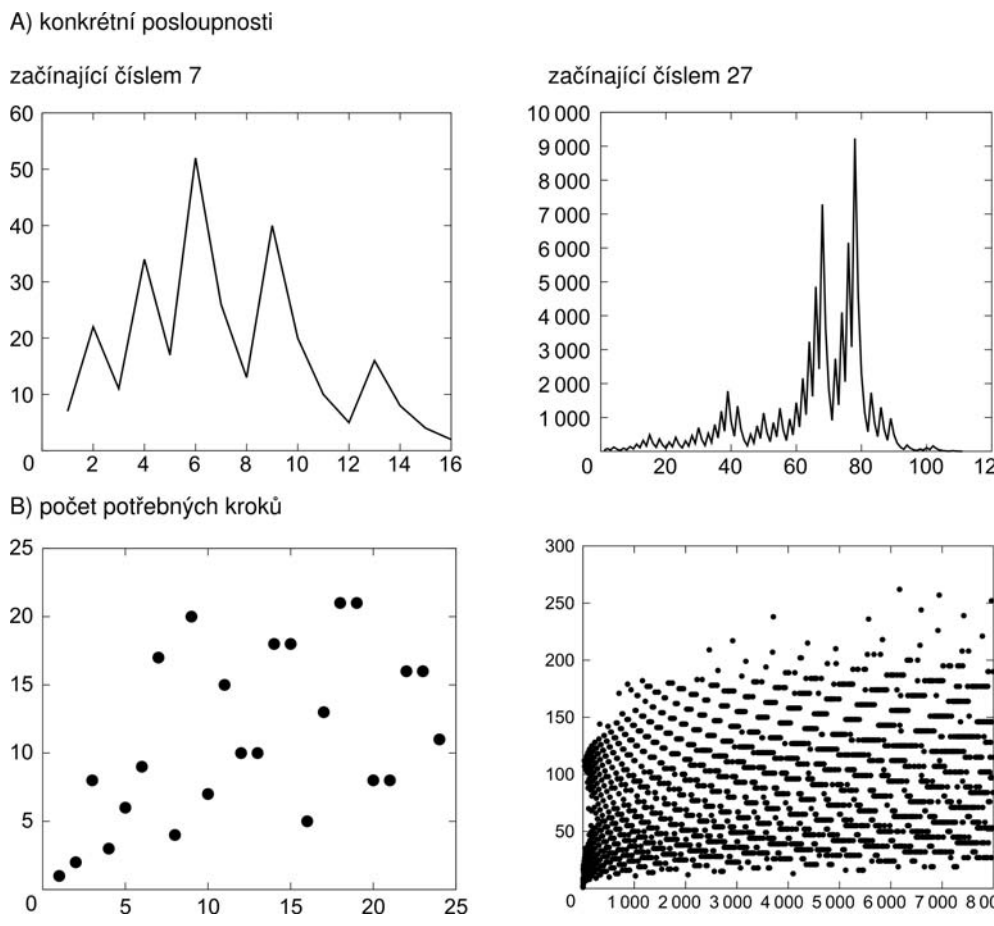
```
def collatz(n):
    while n != 1:
        print n
        if n % 2 == 0:
            n = n / 2
        else:
            n = 3*n + 1
```

Ilustrujme postup na příkladu. Pokud začneme v čísle 7, dostaneme následující posloupnost: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Ačkoliv je definice postupu jednoduchá, výsledné chování je velmi zajímavé. Již na příkladu začínajícím v čísle 7 je vidět, že chování vypadá docela „chaoticky“. A když začneme číslem 27, potřebujeme dokonce 111 kroků, než se dostaneme na číslo 1 a jako jeden z mezivýsledků dostaneme číslo 9 232. Průběh těchto dvou posloupností je graficky znázorněn na obrázku 3.1.

Když máme takto definovaný postup, nabízí se následující otázka. Platí, že ať začneme v libovolném přirozeném čísle, skončí posloupnost v jedničce? Tato otázka činí uvedenou posloupnost známou, protože nikdo totiž zatím nezná s jistotou odpověď. Collatzova (nebo též Ulamova) hypotéza říká, že odpověď na uvedenou otázku je „ano“. Experimentálně bylo ověřeno, že i pro velmi vysoká čísla se výpočet nakonec dostane do jedničky, takže většina lidí věří, že hypotéza platí. Nicméně matematický důkaz zatím nemáme a problém je stále otevřený.

V rámci našeho cvičení se nebudeme pouštět do řešení obtížných matematických problémů, ale využijeme téma k několika jednoduchým programátorským cvičením:

1. Pro zadané číslo vypočítejte posloupnost vygenerovaných čísel a tuto posloupnost zobrazte graficky (podobně jako na obrázku 3.1 A).
2. Pro čísla od 1 do n vypočítejte, kolik kroků potřebujeme, než se dostaneme do jedničky. Hodnoty vykreslete grafem – pro malé n vypadá výsledný graf vcelku náhodně, pro velké n se však objeví zajímavá struktura (viz obrázek 3.1 B).
3. Podobně jako předchozí bod, nezobrazujte však počet kroků, ale maximální hodnotu, na kterou v průběhu výpočtu narazíme (např. pro začátek v čísle 7 je touto maximální hodnotou 52).
4. Pro které číslo od 1 do 100 000 potřebujeme nejvíce kroků, než dospějeme do jedničky? Jaké je maximální číslo, které se v příslušné posloupnosti objeví?



Obrázek 3.1: Collatzův problém – grafy

5. Rekordman je takové číslo x, pro které je počet potřebných kroků větší než pro jakékoliv $y < x$. Posloupnost rekordmanů začíná 1, 2, 3, 6, 7, 9, 18, 25, 27, 54. Najděte všechny rekordmany do 100 000.
6. Vyzkoušejte jiné funkce podobného typu a prozkoumejte jejich chování. Co se například stane, když výraz $3n + 1$ nahradíme za $3n - 1$? Pak už uvedená hypotéza neplatí – když začneme ve vhodném čísle, výpočet se zacyklí a nikdy se nedostane do jedničky. Najděte takové číslo.

Doplňující komentář

Pokud bychom se chtěli problémem zabývat pro vysoká čísla, bylo by smysluplné snažit se program optimalizovat například pomocí ukládání mezivý-

sledků. Pro hodnoty uvedené v zadání (do 100 000) však bohatě stačí přímočaré řešení bez jakýchkoliv optimalizací – prostě přepíšeme funkci pro počítání posloupnosti a pak ji opakovaně voláme. Pro grafické zobrazení výsledků můžeme využít knihovnu pro zobrazení grafů, jež je dostupná ve většině programovacích jazyků. Klidně ale můžeme použít i běžný tabulkový procesor, do kterého zkopírujeme výsledky našeho programu.

3.4 Náhodná procházka

Nápad:	1-2
Kódování:	1-3
Styl úlohy:	*Jednoduchá cvičení s využitím generátoru náhodných čísel.*

Náhodná procházka je postup spočívající v mnoha opakovaných náhodných krocích. Jde o jednoduchý matematický princip, který však dobře modeluje mnoho reálných jevů a je intenzivně studován v oblastech, jako je fyzika, finance nebo biologie. Pro účel úvodního programátorského cvičení se zaměříme pouze na jednoduché variace na toto téma:

1. „Opilcova procházka“ probíhá na jednorozměrném plánu velikosti n. Na levém konci plánu je domov, na pravém hospoda. Opilec se na začátku nachází na poli číslo k. V každém kole se opilec s pravděpodobností p pohne směrem k hospodě a s pravděpodobností $1 - p$ směrem domů. Procházka končí, když opilec dorazí do hospody nebo domů.

2. „Náhodná procházka ve 2D“ probíhá na mřížce velikosti $n \times n$, začínáme uprostřed, v každém kroku se pohneme se stejnou pravděpodobností na jednu ze 4 sousedních buněk. Procházka končí, když narazíme na kraj mřížky.

3. Zjednodušené „Člověče, nezlob se“ se hraje na hracím plánu velikosti n a pouze s 1 figurkou. Figurka se posunuje podle hodů kostkou, tj. podle náhodně generovaných čísel od 1 do 6. Když padne číslo 6, házíme znova a čísla sčítáme. Figurka se posunuje o celkový součet. Hra končí, jakmile dorazíme na poslední pole, přičemž se musíme trefit přesně na poslední pole. Pokud se netrefíme, figurka stojí.

Základní úkol pro všechny varianty spočívá v napsání programu, který pro zadané parametry odsimuluje průběh procházky a textově či graficky jej znázorní. Rozšiřující úkol spočívá v opakovaném spuštění simulace a výpočtu průměrných charakteristik simulace: Jaká je pro dané n, k, p pravděpodobnost, že opilec dorazí do hospody? Jaká je průměrná délka 2D procházky na mřížce $n \times n$? Jak závisí u zjednodušeného „Člověče, nezlob se“ délka hry na velikosti plánu?

Doplňující komentář

Jde o využití základních programátorských konstrukcí, takže úlohy nebudeme rozebírat. Jen poznamenejme, že minimálně pro některé varianty lze uvedené průměrné charakteristiky vypočítat i matematicky, tj. bez provádění simulací. Čtenář s matematickými ambicemi se tedy může pokusit odvodit příslušný výsledek a pak jej pomocí programu ověřit. Případně je možné se zabývat nejen průměrnými hodnotami, ale i dalšími parametry příslušných pravděpodobnostních distribucí, například mediánem nebo rozptylem.

3.5 Dělitelnost a prvočísla

Nápad:	2-3
Kódování:	1-3
Styl úlohy:	*Jednoduchá cvičení řešitelná různými algoritmickými přístupy, na kterých lze ilustrovat rozdíly v efektivitě algoritmů.*

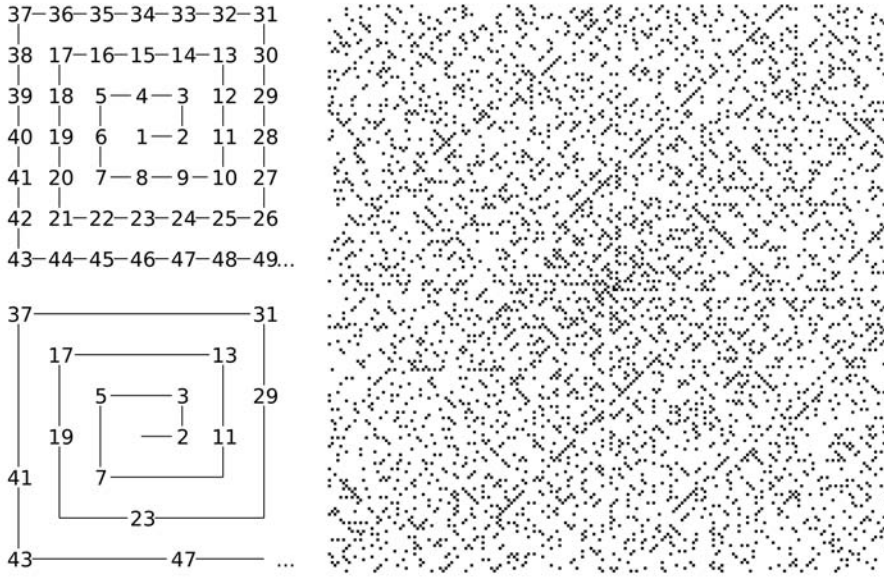
Základní zadání spočívají v tom, že program načte přirozená čísla a vypíše informace související s dělitelností či prvočíselností:

1. Výpis všech dělitelů čísla n.
2. Test prvočíselnosti čísla n.
3. Výpis prvních n prvočísel.
4. Výpis největšího společného dělitele čísel n a m.
5. Hledání „prvočíselných dvojčat“: program načte číslo n a najde nejmenší $p \geq n$ takové, že p a $p + 2$ jsou obě prvočísla.

Pokročilejší ztvárnění tématu dostaneme za využití bitmapové (případně textové) grafiky, kdy zakreslujeme distribuci prvočísel graficky pomocí takzvané Ulamovy spirály. Ta vznikne tak, že si napíšeme všechna čísla „do šneka“ a potom vybereme pouze prvočísla. Na obrázku 3.2. máme takto znázorněno prvních 40 tisíc přirozených čísel, prvočísla jsou černě. Podobným způsobem můžeme znázornit třeba čísla dělitelná zadaným číslem n.

Doplňující komentář

Prvočísla jsou velmi zajímavý matematický objekt, protože nejsou nijak striktně pravidelná, ale přitom vykazují řadu dílčích pravidelností. Tuto vlastnost prvočísel názorně ilustruje Ulamova spirála, kterou objevil Stanislav Ulam v šedesátých letech, když si čmáral po papíře během nudné přednášky. Výsledný obrázek nemá žádný zjevný řád, nicméně jsou na něm vidět výrazné „diagonály“. S prvočísly se také váže mnoho těžkých a často i nevyřešených



Obrázek 3.2: Ulamova spirála: ilustrace principu a spirála velikosti 200×200

matematických problémů. Jeden takový známý problém souvisí s prvočíselnými dvojčaty zmíněnými v zadání: Existuje nekonečně mnoho prvočíselných dvojčat?

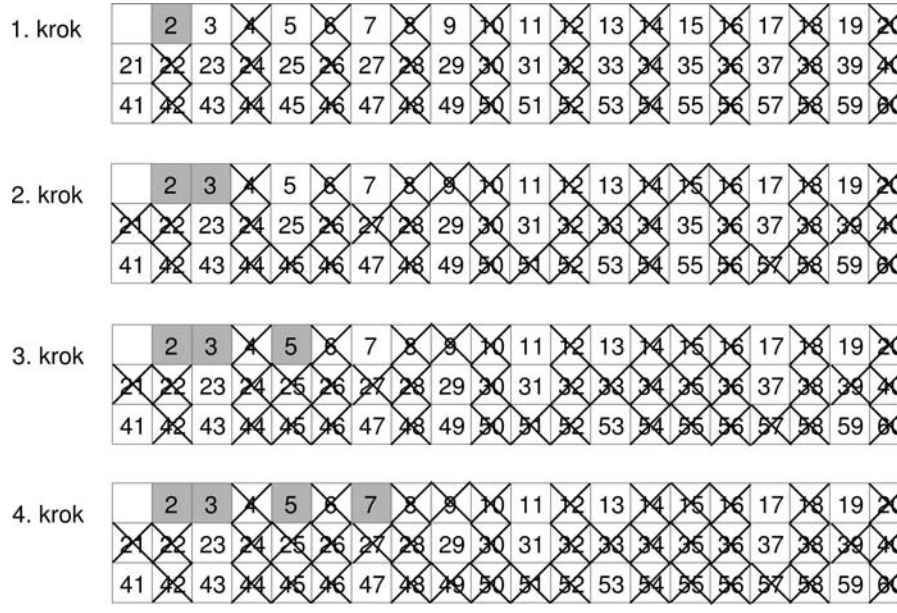
Výpis všech dělitelů lze realizovat přímočaře pomocí jednoho `for` cyklu. Test prvočíselnosti můžeme udělat přímočaře spočítáním dělitelů. Program můžeme mírně optimalizovat, například tak, že zkoušíme dělit pouze dvojkou a lichými čísly, a to jen do odmocniny z n. Jsou pochopitelně možné i výraznější optimalizace – testování prvočísel je důležitý praktický problém například v moderní kryptografii. Pokročilejší optimalizace ale jdou vysoce nad rámec úvodních programátorských cvičení.

Výpis prvočísel můžeme udělat dvěma základními způsoby, z tréninkových důvodů je vhodné implementovat oba dva. První metoda je prostá hrubá síla. Procházíme postupně přirozená čísla a pro každé z nich zjistíme pomocí výše popsaného testu, zda je prvočíslem. Druhá metoda se nazývá Eratosthenovo síto a je ilustrována na obrázku 3.3. Napíšeme si čísla od 2 do n, v každém kroku vždy vybereme nejmenší neoznačené číslo a označíme všechny násobky tohoto čísla. Všimněte si, že v uvedeném obrázku jsou již po 4 krocích výpočtu všechna neoznačená čísla prvočísla.

Také hledání největšího společného dělitele můžeme dělat více způsoby. Základní řešení je opět hrubá síla: procházíme sestupně čísla od n do 1 a zkou-

šíme jimi dělit n i m, dokud nenajdeme společného dělitele. Lepší řešení je Euclidův algoritmus, který je založen na pozorování, že $NSD(n, m) = NSD(m, n \bmod m)$, kde $\bmod$ značí zbytek po dělení. Tento vztah můžeme snadno převést na algoritmus výpočtu, například pro čísla 504 a 180 probíhá výpočet následovně: $NSD(504, 180) = NSD(180, 144) = NSD(144, 36) = NSD(36, 36) = NSD(36, 0) = 36$. Alternativní zápis stejného výpočtu ukazuje tabulka 3.1.

Euclidův algoritmus je výrazně efektivnější než naivní algoritmus, což jde snadno experimentálně ověřit. Příklad slouží jako dobrá ilustrace rozdílů mezi výpočetní náročností různých algoritmů.



Obrázek 3.3: Eratosthenovo síto: první 4 kroky výpočtu

Tabulka 3.1: Euclidův algoritmus: příklad

krok	n	m
1	504	180
2	180	144
3	144	36
5	36	36
6	36	0

3.6 Reprezentace čísel

Nápad:	3
Kódování:	2-4
Styl úlohy:	*Úloha sloužící především pro důkladné pochopení reprezentace čísel v počítači.*

Úkolem je napsat program, který převádí čísla mezi různými reprezentacemi (viz příklady v tabulce 3.2):

1. Převod čísel z desítkové do binární soustavy a zpět.
2. Převod čísel mezi dvěma obecnými pozičními soustavami, tj. vstupem je zápis čísla X v soustavě o základu r_1 a informace o žádaném cílovém základu r_2, např. příklad v tabulce 3.2 říká, že chceme převádět číslo zapsané jako „25“ v sedmičkové soustavě, do trojkové soustavy, výsledek je „201“.
3. Převod čísla na římské číslice a zpět.
4. Převod čísel na slovní popis.

Tabulka 3.2: Reprezentace čísel: příklady vstupů a výstupů

	Příklad	Vstup	Výstup
1a	z desítkové na binární	22	10110
1b	z binární na desítkovou	1001	9
2	převod mezi pozičními soustavami obecně	25, 7, 3	201
3a	převod na římské číslice	37	XXXVII
3b	převod z římských číslic	XLII	42
4	převod na slovní popis	126	sto dvacet šest

Doplňující komentář

Připomeňme stručně základní princip pozičních soustav: V soustavě o základu r můžeme používat cifry od 0 do $r-1$. Každá pozice v zápisu čísla má přiřazenu svoji váhu, ta vždy odpovídá mocnině základu. Výsledné číslo získáme jako součet násobků cifer a příslušných vah. Tedy číslo o zápisu $X = a_n a_{n-1} \ldots a_1 a_0$ má v poziční soustavě o základu r hodnotu:

$$a_n r^n + a_{n-1} r^{n-1} + \cdots + a_1 r + a_0$$

Převody mezi pozičními soustavami jdou řešit pomocí krátkého, ale myšlenkově netriviálního kódu. Základní princip spočívá v tom, že zápis v soustavě o základu r budujeme od konce pomocí opakovaného dělení čísla X