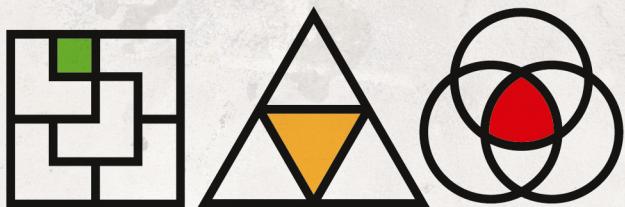


Radek Pelánek



# Programátorská cvičebnice

## Algoritmy v příkladech

**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](http://www.albatrosmedia.cz)

[eshop@albatrosmedia.cz](mailto: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.

© Albatros Media a.s. Všechna práva vyhrazena. Žádná část této publikace nesmí být kopirována  
a rozmnožována za účelem rozšiřování v jakékoli formě či jakýmkoli způsobem bez písemného souhlasu  
vydavatele.

1. vydání



# Obsah

<b>Předmluva</b>	<b>7</b>
<b>1 O knize</b>	<b>9</b>
1.1 Úlohy a jejich popisky . . . . .	10
1.2 Jak knihu používat . . . . .	12
1.3 Scénáře použití . . . . .	13
<b>2 Přehled pojmů</b>	<b>17</b>
2.1 Složitost . . . . .	17
2.2 Rekurze a metoda rozděl a panuj . . . . .	18
2.3 Dynamické programování . . . . .	19
2.4 Hladové algoritmy . . . . .	19
2.5 Hrubá síla a heuristiky . . . . .	20
2.6 Grafy a stavové prostory . . . . .	21
2.7 Objektově orientované programování . . . . .	21
2.8 Grafika . . . . .	22
<b>3 Počítání s čísly</b>	<b>25</b>
3.1 Hrátky s čísly . . . . .	25
3.2 Posloupnosti . . . . .	26
3.3 Collatzův problém . . . . .	27
3.4 Náhodná procházka . . . . .	30
3.5 Dělitelnost a prvočísla . . . . .	31
3.6 Reprezentace čísel . . . . .	34
3.7 Fibonacciho posloupnost . . . . .	35
3.8 Pascalův trojúhelník . . . . .	36
3.9 Výpočet $\pi$ . . . . .	38
3.10 Permutace, kombinace, variace . . . . .	40

<b>4 Obrázky a geometrie</b>	<b>43</b>
4.1 Textová grafika . . . . .	44
4.2 Želví grafika: základy . . . . .	45
4.3 Želví grafika: fraktály . . . . .	48
4.4 Sierpiňského fraktál . . . . .	50
4.5 Bitmapová grafika . . . . .	52
4.6 Mandelbrotova množina . . . . .	54
4.7 Konvexní obal . . . . .	57
4.8 Triangulace . . . . .	59
<b>5 Šifrování a práce s textem</b>	<b>63</b>
5.1 Analýza a imitace textu . . . . .	63
5.2 Transpoziční šifry . . . . .	66
5.3 Substituce a kódování . . . . .	69
5.4 Rozlomení šifer . . . . .	70
5.5 Přesmyčky . . . . .	73
<b>6 Logické úlohy</b>	<b>75</b>
6.1 Číselné bludiště . . . . .	76
6.2 Přelévání vody . . . . .	77
6.3 Hanojské věže . . . . .	79
6.4 Pokrývání mřížky . . . . .	82
6.5 Hledání cest v bludišti . . . . .	83
6.6 Generování bludišt' . . . . .	84
6.7 Rozmístování figur na šachovnici . . . . .	86
6.8 Jak navštívit všechna pole mřížky? . . . . .	89
6.9 Polyomina . . . . .	91
6.10 Sudoku . . . . .	94
6.11 Sokoban . . . . .	97
<b>7 Hry</b>	<b>99</b>
7.1 Kámen, nůžky, papír . . . . .	100
7.2 Hádání čísla . . . . .	102
7.3 Oběšenec . . . . .	103
7.4 Logik . . . . .	105
7.5 Hra Nim . . . . .	107
7.6 Tetris . . . . .	108
7.7 Jednorozměrné piškvorky . . . . .	110
7.8 Piškvorky . . . . .	111
7.9 Souboje virtuálních robotů . . . . .	112

<b>8 Klasické informatické problémy</b>	<b>117</b>
8.1 Rozměňování mincí . . . . .	117
8.2 Simulátor hry Život . . . . .	119
8.3 Problémy s řetězci a posloupnostmi . . . . .	122
8.4 Experimenty s řadicími algoritmy . . . . .	123
8.5 Vyhodnocování výrazů . . . . .	125
8.6 Grafové algoritmy . . . . .	127
<b>9 Další náměty</b>	<b>131</b>
9.1 Generování a transformace obrázků . . . . .	131
9.2 Blízké body . . . . .	132
9.3 Akční hry . . . . .	133
9.4 Implementace datových struktur . . . . .	134
9.5 Implementace matematických operací . . . . .	134
9.6 Zpracování a analýza reálných dat . . . . .	135
9.7 Interpret jednoduchého programovacího jazyka . . . . .	135
<b>10 Vybraná řešení</b>	<b>137</b>
10.1 Počítání s číslami . . . . .	138
10.2 Obrázky a geometrie . . . . .	143
10.3 Šifrování a práce s textem . . . . .	149
10.4 Logické úlohy . . . . .	152
10.5 Hry . . . . .	161
10.6 Klasické informatické problémy . . . . .	167
<b>Rejstřík</b>	<b>173</b>



# 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 zkoumat 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 zkušenosť. 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 Flbot.

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 Barčě 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 mírněn jako připomenutí pojmu, 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 programátorský 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 programátorem, 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 zkušenosťech 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. Ambiciozní čtenář by tedy měl úlohu zkoumat 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

*Bud' 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 porovnat. 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 pojmu). 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 použítí 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 zkousit vymyslet vlastní rozšíření a variace a zkousit 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 roztríděných podle různých kritérií. V této kapitole jsou příklady roztríděny 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 Výpis 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, zkoušet 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 Sierpiňského 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 rámcem 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 zkoušet co nejfektivněji využít vícejádrové procesory nebo provést výpočet v distribovaném 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 téma semestrálních (ročníkových) projektů, případně i jako inspirace pro téma 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 pojmu

*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 pojmu 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 pojmu nesetkal, je vhodné si před řešením příkladů souvisejících s daným pojmem téma blíže dostudovat. Výklad zmíněných pojmu lze najít v mnoha učebnicích, viz např. Skiena, 1998; Wroblewski, 2004; Cormen et al., 2001; Kučera, 2009. U mnoha základních pojmu 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 rychlosť a podle toho určit „vítěze“. Takový přístup má však zřejmě 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. Variaci na metodu „rozděl a panuj“ je metoda „převed’ 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 sloučováním), 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 Rozměřování 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 (nerozebrané 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 pomoci 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 Rozmístování 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 pojmu lze najít v mnoha učebnicích nebo snadno i na Internetu. Zde pouze zmíníme seznam základních pojmu, 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.

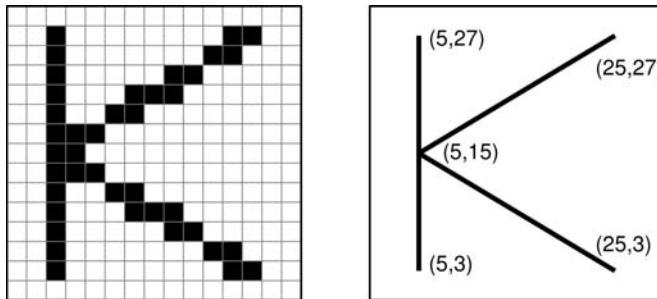
Textová grafika je použita výrazně v příkladech 4.1 Textová grafika, 7.6 Tetris, 8.2 Simulátor hry Život a dále okrajově u většiny logických úloh a her pro vykreslování zadání a řešení úloh.

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. Bud' 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 Sierpiňského fraktál, 4.5 Bitmapová grafika, 4.6 Mandelbrotova množina.

Jiný způsob reprezentace obrázků představuje *vektorová grafika*, ve které je obrázek popsán pomocí geometrických útvářů. 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 zkonzervovat 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 Sierpiňského 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 výpočtu je vhled, nikoliv čísla.*

R. Hamming

Tato kapitola obsahuje náměty na příklady, které mají velmi jednoduché vstupně-výstupní chování: vesměs pouze načtem 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 nejnáročnejší dvě poslední, i když i u nich je náročné přijít především na princip posloupnosti, programátorským 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 encyklopédii celočíselných posloupností (*The On-Line Encyclopedia of Integer Sequences*, [oeis.org](http://oeis.org)). V této encyklopédii 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:**

*Programátorský jednoduchý 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, Syrakusky 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.

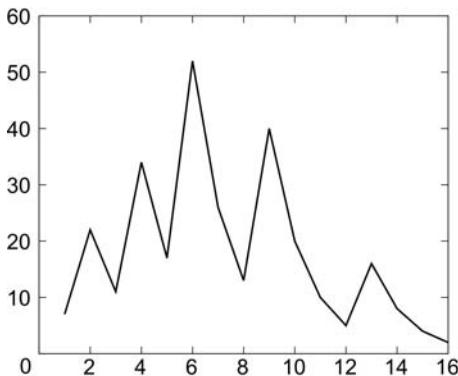
Když máme takto definovaný postup, nabízí se následující otázka. Platí, že at'začneme v libovolném přirozeném čísle, skončí posloupnost v jedničce? Tato otázka činí uvedenou posloupnost známou, protože nikdo totiž zatím nezná s jistotou odpověď. Collatzova (nebo též Ulamova) hypotéza říká, že odpověď 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 programátor-ským cvičením:

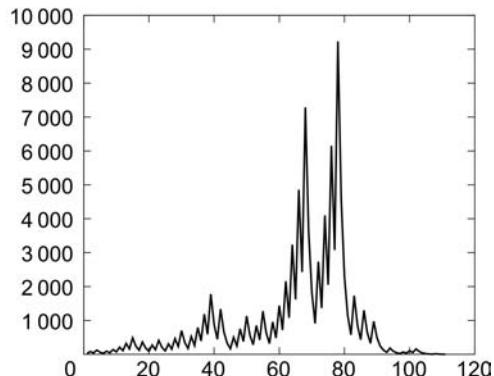
1. Pro zadané číslo vypočítejte posloupnost vygenerovaných čísel a tuto posloupnost zobrazte graficky (podobně jako na obrázku 3.1 A).
2. Pro čísla od 1 do  $n$  vypočítejte, kolik kroků potřebujeme, než se dostaneme do jedničky. Hodnoty vykreslete grafem – pro malé  $n$  vypadá výsledný graf vcelku 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?

## A) konkrétní posloupnosti

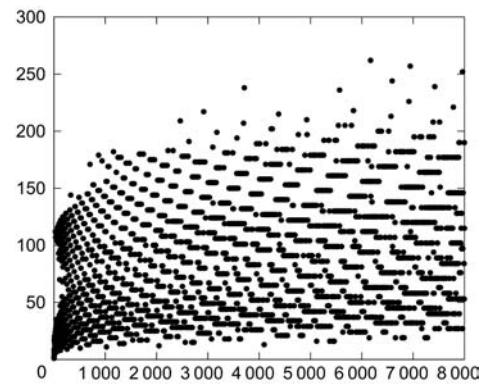
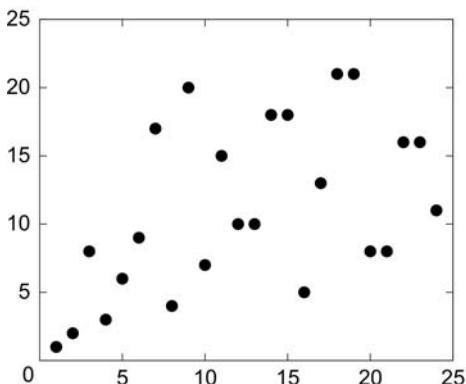
začínající číslem 7



začínající číslem 27



## B) počet potřebných kroků



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 opakováně 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 opakováném 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.*

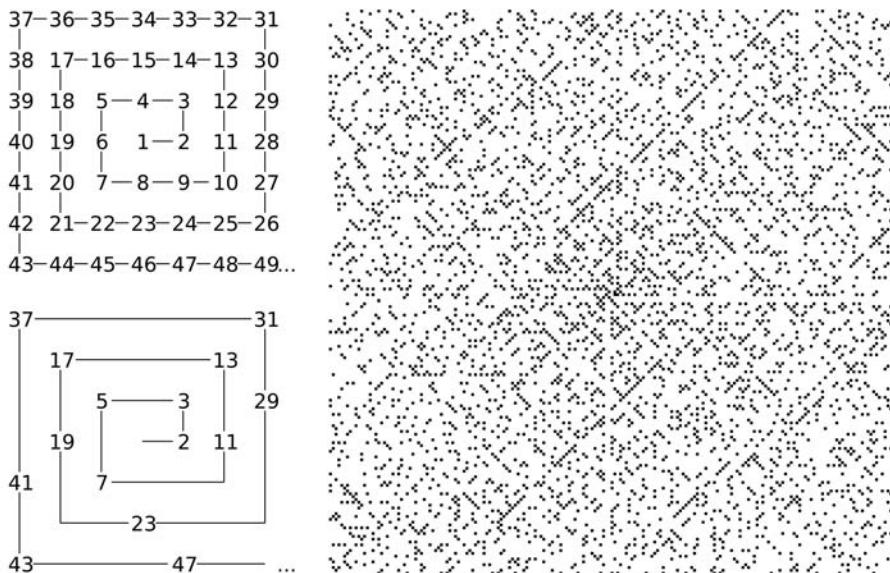
Základní zadání spočívají v tom, že program načte přirozená čísla a vypíše informace související s dělitelností či prvočíselností:

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 černé. 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 sedesátých 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  $200 \times 200$

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

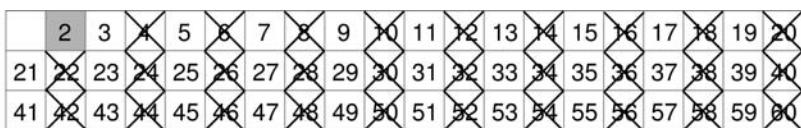
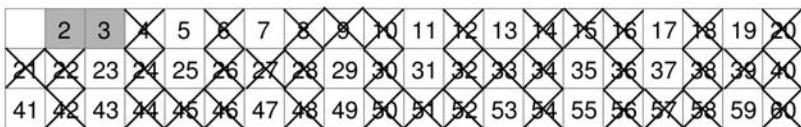
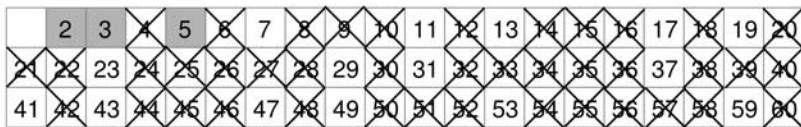
Výpis všech dělitelů lze realizovat přímočáre pomocí jednoho `for` cyklu. Test prvočíselnosti můžeme udělat přímočáre spočítáním dělitelů. Program můžeme mírně optimalizovat, například tak, že zkoušíme dělit pouze dvojkou a lichými čísly, a to jen do odmocniny z  $n$ . Jsou pochopitelně možné i výraznější 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 Eratosthe-novo 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  $NSD(n, m) = NSD(m, n \text{ mod } m)$ , kde mod 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.

1. krok	
2. krok	
3. krok	
4. krok	

Obrázek 3.3: Eratosthenovo sítě: 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

<b>Nápad:</b>	3
<b>Kódování:</b>	2-4
<b>Styl úlohy:</b>	<i>Úloha sloužící především pro důkladné pochopení reprezentace čísel v počítači.</i>

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  $X = a_n a_{n-1} \dots a_1 a_0$  má v poziční soustavě o základu  $r$  hodnotu:

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

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  $X$

základem  $r$ . Převod na římské číslice a na slovní popis je naopak spíše o dobrém rozboru případů a vhodné reprezentaci dat než o nějaké zásadní myšlence.

## 3.7 Fibonacciho posloupnost

<b>Nápad:</b>	2
<b>Kódování:</b>	1
<b>Styl úlohy:</b>	<i>Programátorský jednoduché cvičení, které ilustruje, jak k jednomu problému můžeme přistoupit různými způsoby.</i>

Fibonacciho posloupnost je známá matematická posloupnost, která začíná dvěma jedničkami a další člen vždy vznikne jako součet dvou předchozích, začíná tedy  $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$ . Jde o příklad rekurentní posloupnosti, konkrétně o „lineární rekurentní posloupnost s konstantními koeficienty“. Obecný tvar takové posloupnosti je:

$$f_n = \sum_{i=1}^k a_i \cdot f_{n-i} = a_1 f_{n-1} + a_2 f_{n-2} + \dots + a_k f_{n-k}$$

Fibonacciho posloupnost v tomto obecném zápisu dostaneme pro  $k = 2$ ,  $a_1 = 1$ ,  $a_2 = 2$ . Zjednodušený přepis obecného pravidla je:

$$f_1 = 1, f_2 = 1, f_n = f_{n-1} + f_{n-2}$$

Úkol v tomto cvičení je napsat program, který pro zadané  $n$  vypíše  $n$ -tý člen Fibonacciho posloupnosti. Program napište různými styly a porovnejte výkon jednotlivých způsobů výpočtu:

1. rekurzivně,
2. iterativně za využití jednorozměrného pole,
3. iterativně pouze pomocí celočíselných proměnných,
4. explicitním vzorcem pro  $n$ -tý člen:

$$f_n = \frac{\phi^n - (-1/\phi)^n}{\sqrt{5}}$$

kde  $\phi = (1 + \sqrt{5})/2$ .

Úlohu můžeme dále rozšířit na výpočet prvních  $n$  členů obecné rekurentní posloupnosti řádu  $m$ . Vstupem je v tomto případě  $m$  koeficientů a  $m$  prvních členů posloupnosti.

### Doplňující komentář

Posloupnost je pojmenována podle Fibonacciiho, italského matematika z 13. století, který modeloval idealizovaný růst populace králíků: na začátku máme 1 pár nesmrtelných králíků; jakmile králíci dorostou do věku 2 let, začnou plodit mladé (1 pár zplodí za rok 1 pár mladých). Jde o neomezené množení, které pochopitelně neodpovídá realitě, nicméně Fibonacciiho posloupnost se v reálných přírodních jevech objevuje docela často, především u rostlin a „kruhového růstu“, např. v pozicích větví na stromě, semen u slunečnice nebo zrn u šíšek. Fibonacciiho posloupnost má také zajímavé souvislosti s celou řadou matematických jevů, např. uvedená konstanta  $\phi$  souvisí s konceptem „zlatého řezu“.

Kromě uvedených zajímavostí vykazuje Fibonacciiho posloupnost také velký výukový potenciál. V matematice se využívá jako typický příklad pro ilustraci metod řešení rekurentních rovnic, tj. způsobu odvození výše uvedeného explicitního vzorce. V programování jde o typický příklad nevhodného použití rekurze. Víte, proč je v tomto případě použití rekurze nevhodné? Vyšvětlení je uvedeno v řešení na konci knihy.

## 3.8 Pascalův trojúhelník

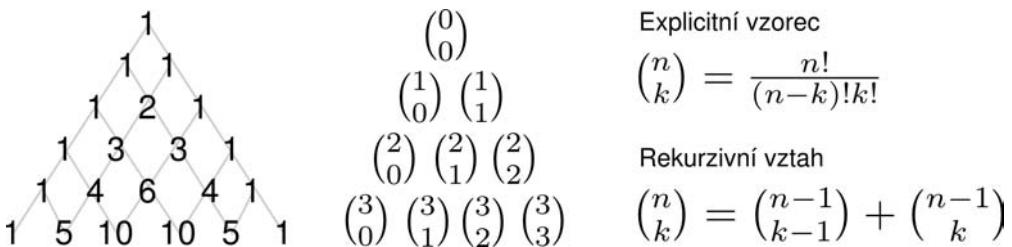
<b>Nápad:</b>	2
<b>Kódování:</b>	2
<b>Styl úlohy:</b>	<i>Podobné cvičení jako Fibonacciiho posloupnost – ilustrace více přístupů k jednomu problému a zajímavých matematických souvislostí.</i>

Pascalův trojúhelník (viz obrázek 3.4) je tvořen řádky rostoucí délky a vznikne následujícím postupem:

- první řádek je tvořen jednou jedničkou,
- každý další řádek začíná a končí jedničkou a čísla mezi tím vzniknou vždy jakou součet dvou čísel na předchozím řádku.

Čísla, která takto dostaneme, jsou kombinační čísla  $\binom{n}{k}$ . Kombinační číslo  $\binom{n}{k}$  udává počet  $k$  prvkových kombinací  $n$ . Hodnotu kombinačního čísla můžeme vypočítat pomocí explicitního vzorce nebo rekurzivního vztahu (viz obrázek 3.4). Rekurzivní vztah odpovídá tomu, jak konstruujeme Pascalův trojúhelník.

Úkolem v tomto cvičení je napsat program, který vypíše prvních  $n$  řádků Pascalova trojúhelníku, a to pokud možno několika různými způsoby.



Obrázek 3.4: Pascalův trojúhelník a kombinační čísla

### Doplňující komentář

Pascalův trojúhelník můžeme vypisovat různými způsoby, přičemž jednotlivé algoritmy jsou různě efektivní. V tomto případě je užitečné implementovat i nevhodná řešení, protože si na nich lze dobře prakticky vyzkoušet obecné koncepty.

Přímočaře můžeme výpočet realizovat pomocí uvedeného explicitního vzorečku pro kombinační čísla. Tento výpočet je mírně neefektivní kvůli opakování počítání stejných faktoriálů. Především však hodnoty faktoriálů velmi rychle rostou a pro vyšší hodnoty  $n$  může dojít k „přetečení“ celočíselných proměnných při výpočtu faktoriálu, čímž dostaneme chybý výsledek. U některých programovacích jazyků (např. Python) nedojde k přetečení, protože interpret automaticky „nafukuje“ celočíselné proměnné, ovšem za cenu zpomalení výpočtu.

Druhý možný přístup spočívá ve využití uvedeného rekurzivního vzorečku pro kombinačního čísla. Výpočet můžeme přímočaře implementovat za využití rekurze, tento výpočet je však velmi neefektivní – ze stejného důvodu, jako je neefektivní výpočet Fibonacciho čísel pomocí rekurze (viz předchozí cvičení).

Efektivní způsob konstrukce odpovídá postupnému počítání jednotlivých řádků trojúhelníku, přičemž další řádek vždy vyrobíme tak, že sčítáme čísla z předchozího řádku. Jde v podstatě o elementární využití metody dynamického programování. Přímočařá implementace vede k použití dvojrozměrného pole, můžeme však být i úspornější a vystačit pouze s dvěma jednorozměrnými poli, která vhodně „cyklíme“.

### 3.9 Výpočet $\pi$

<b>Nápad:</b>	2
<b>Kódování:</b>	2-3
<b>Styl úlohy:</b>	<i>Programy, které různými způsoby approximují <math>\pi</math>. Cvičení je jednoduché, protože zadání poskytuje konkrétní návody, které stačí přepsat do programu.</i>

Číslo  $\pi = 3,141592653589793\dots$  je jedna z nejznámějších matematických konstant, takže o ní jistě každý čtenář této knihy slyšel. Jak se ale určuje hodnota  $\pi$ ? Můžeme napsat program, který  $\pi$  vypočítá? To je právě úkolem tohoto cvičení. Cvičení se soustředí na trénink programování a nikoliv na matematiku, proto popíšeme konkrétní metody, jak  $\pi$  počítat, a úkolem je pouze je implementovat.

Základní způsob approximace  $\pi$  je pomocí konvergujících řad a posloupností. Hodnotu  $\pi$  lze vyjádřit jako součet vhodné nekonečné řady nebo jako limitu posloupnosti. Čím více členů řady (posloupnosti) vypočítáme, tím přesnější získáme odhad. Dva konkrétní přístupy jsou:

1. Gregoryho-Leibnizova řada (součet je  $\pi$ ):

$$4 \cdot \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \dots$$

2. Archimedova metoda (dvě posloupnosti  $a_n, b_n$  společně konvergující k  $\pi$ ):

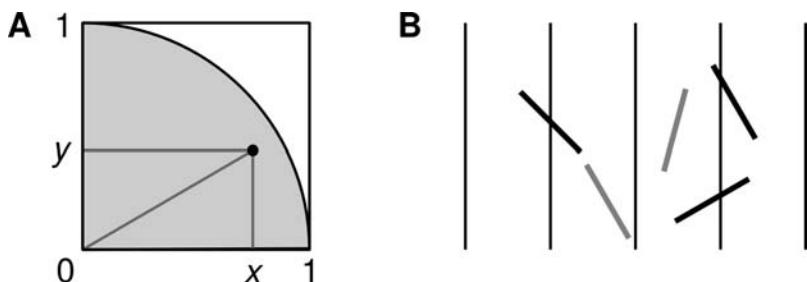
$$a_0 = 2\sqrt{3}; b_0 = 3$$

$$a_{n+1} = \frac{2a_n b_n}{a_n + b_n}$$

$$b_{n+1} = \sqrt{a_{n+1} b_n}$$

Zcela jiným přístupem je Monte Carlo metoda – takto se obecně nazývají numerické metody založené na náhodné simulaci. V případě výpočtu  $\pi$  jde o pokusy, u nichž pravděpodobnost úspěchu je vyjádřena jako výraz obsahující  $\pi$ . Opakovaným provedením pokusů tak můžeme odhadnout hodnotu  $\pi$ . Pochopitelně čím více pokusů uděláme, tím přesnější odhad dostaneme. Dvě známé metody jsou ilustrovány na obrázku 3.5:

1. Házení šipek do čtvrtdisku, obsah čtvrtdisku je  $\pi/4$ , obsah čtverce je 1, takže pravděpodobnost, že náhodně vybraný bod v tomto čtverci leží uvnitř čtvrtkruhu, je  $\pi/4$ .



**Obrázek 3.5:** Ilustrace výpočtu  $\pi$  pomocí Monte Carlo metody: A) Házení šipek, B) Buffonova jehla

2. Buffonova jehla – vezmeme jehlu délky  $l$  a opakovaně ji házíme na list papíru, na kterém jsou narýsovány rovnoběžné čáry ve vzdálenosti  $l$ . Pravděpodobnost, že jehla protne některou z čar, je  $2/\pi$ .

Úkolem v tomto cvičení je napsat program approximující hodnotu  $\pi$  pomocí těchto metod a porovnat jejich efektivitu, tj. na kolik míst jsou schopny určit hodnotu  $\pi$  v omezeném čase (např. 1 vteřina). Program dostane na vstup identifikaci metody a hodnotu  $n$ , která určuje, kolik iterací (pokusů) se má provést. Program pak vypíše vypočítaný odhad  $\pi$  a informaci o tom, na kolik míst se shoduje se správnou hodnotou  $\pi$ . Tu získáme z matematické knihovny nebo ji zadáme do programu podle matematických tabulek. Následně pak provedeme srovnání jednotlivých metod. Můžeme například vytvořit graf znázorňující přesnost metody v závislosti na  $n$ , příp. době běhu programu.

### Doplňující komentář

Programování je v tomto případě docela přímočaré, jediný trochu náročnější bod je rozmyslet si matematické aspekty Monte Carlo metod. V komentáři tedy popíšeme pouze souvislosti.

První uvedená metoda bývá označována různými názvy, kromě uvedeného názvu Gregoryho-Leibnizova řada též jako Madhavova-Leibnizova řada nebo prostě Leibnizova řada. Tato řada konverguje velmi pomalu, abychom určili  $\pi$  na 2 desetinná místa, potřebujeme stovky iterací.

Archimedova metoda je založena na approximaci kruhu pomocí mnohoúhelníků. Archimedes pomocí tohoto přístupu určil, že  $3\frac{10}{17} < \pi < 3\frac{1}{7}$ . Tato metoda již je docela rychlá – z uvedených metod vede jednoznačně k nejpřesnějším výsledkům. Existují ještě rychlejší metody, ty jsou ovšem myšlenkově náročnější.

Monte Carlo je název obecného přístupu, kdy výsledky nepočítáme exaktně, ale pomocí numerické simulace. Tento přístup se používá například pro numerické výpočty určitých integrálů v případech, kdy neumíme integrál vypočítat analyticky. Pro praktický výpočet  $\pi$  je tato metoda nevhodná, uvedené metody jsou velmi nepřesné – uvádíme je pouze pro zajímavost a pro cvičení programování, nikoliv jako ukázku praktického použití Monte Carlo metody.

### 3.10 Permutace, kombinace, variace

<b>Nápad:</b>	3-4
<b>Kódování:</b>	2
<b>Styl úlohy:</b>	<i>Cvičení pro zopakování užitečných matematických pojmu a procvičení rekurze.</i>

Úkolem je napsat program, který pro danou množinu prvků vypíše všechny:

1. permutace, tj. uspořádání zadaných prvků do fixního pořadí,
2.  $k$  prvkové kombinace, tj. všechny možné výběry  $k$  prvků ze zadane množiny,
3.  $k$  prvkové kombinace s opakováním, tj. všechny možné výběry  $k$  prvků ze zadane množiny, přičemž prvky se mohou opakovat,
4.  $k$  prvkové variace, tj. všechny možné uspořádané výběry  $k$  prvků ze zadane množiny,
5.  $k$  prvkové variace s opakováním, tj. všechny možné uspořádané výběry  $k$  prvků ze zadane množiny, přičemž prvky se mohou opakovat.

Tabulka 3.3 ukazuje příklady vstupu a výstupu pro jednotlivé varianty.

**Tabulka 3.3:** Permutace, kombinace, variace – příklady

Úloha	Vstup	Výstup
permutace	A, B, C	ABC, ACB, BAC, BCA, CAB, CBA
kombinace	A, B, C, D $k = 2$	AB, AC, AD, BC, BD, CD
kombinace s opakováním	A, B, C, D $k = 2$	AA, AB, AC, AD, BB, BC, BD, CC, CD, DD
variace	A, B, C, D $k = 2$	AB, AC, AD, BA, BC, BD, CA, CB, CD, DA, DB, DC
variace s opakováním	A, B, C $k = 2$	AA, AB, AC, BA, BB, BC, CA, CB, CC

## Doplňující komentář

Připomeňme vzorečky udávající délku jednotlivých výsledků (hodí se např. pro základní rychlou kontrolu správnosti výstupu):

- Počet všech permutací  $n$  prvků je  $n!$ .
- Počet všech  $k$  prvkových kombinací z  $n$  prvků je  $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ .
- Počet všech  $k$  prvkových kombinací s opakováním z  $n$  prvků je  $\binom{n+k-1}{k}$ .
- Počet všech  $k$  prvkových variací z  $n$  prvků je  $\frac{n!}{(n-k)!}$ .
- Počet všech  $k$  prvkových variací s opakováním z  $n$  prvků je  $n^k$ .

Všechny uvedené problémy lze realizovat elegantně pomocí rekurze. Například všechny permutace  $n$  prvků dostaneme tak, že vezmeme jeden z prvků na první místo a pak zavoláme generování všech permutací na zbývajících  $n-1$  prvcích, toto opakujeme pro všechny prvky. Variace můžeme určit analogicky.

Všechny  $k$  prvkové kombinace z  $n$  prvků dostaneme tak, že vezmeme jeden z prvků a ten bud' zařadíme do výběru a zavoláme hledání všech  $k-1$  prvkových kombinací ze zbývajících  $n-1$  prvcích, nebo jej nezařadíme do výběru a zavoláme hledání všech  $k$  prvkových kombinací ze zbývajících  $n-1$  prvcích. Tento postup odpovídá rekurzivnímu vztahu, který platí pro kombinační čísla:  $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ .

Generování všech permutací, variací či kombinací je užitečný krok v mnoha větších projektech, typicky pokud řešíme dílčí podproblém hrubou silou. „Vyzkoušet všechny možnosti“ většinou znamená vygenerovat všechny permutace, kombinace či variace a ty otestovat (viz např. cvičení 7.4 Logik). Protože tedy jde o prakticky užitečný krok, ve většině programovacích jazyků jdou uvedené úlohy řešit pomocí volání vestavěné funkce nebo funkce z knihovny. V rámci tohoto cvičení však pochopitelně není účel volat vestavěnou funkci, ale napsat vlastní řešení pomocí základních operací.



# 4 Obrázky a geometrie

*Věda je cokoliv, čemu rozumíme tak dobře, aby chom to vysvětlili počítací.  
Umění je vše ostatní.*

D. Knuth

V předchozí kapitole byla výstupem programů především čísla, teď se podíváme na problémy, kde výstupem jsou obrázky. Díky tomu jsou tyto úlohy atraktivní, protože většinu lidí baví pracovat s obrázky, byť jednoduchými. Studenti také bývají více motivováni vyřešit tyto úlohy správně než u úloh pracujících s čísly. Pokud program vykresluje obrázek špatně, většinou to „po-buřuje“ vrozený estetický cit člověka a díky tomu má student větší motivaci program spravit. Pokud program vypisuje špatné číslo, člověka števe, že má program špatně, ale většinou nemá problém s tím, že by se mu výstup „nelíbil“.

Výstupem následujících úloh je tedy vždy obrázek, konkrétní styl úloh je však hodně rozmanitý:

- Úlohy „ASCII grafika“, „Želví grafika: základy“ a „Bitmapová grafika“ jsou jednoduché příklady na procvičení základů programování. Nejsou potřeba žádné speciální algoritmy, může být však důležité umět se na problém (obrázek) správně podívat.
- Úlohy „Želví grafika: fraktály“ a „Sierpiňského fraktál“ jsou příklady na procvičení rekurze. Tyto úlohy jsou řešitelné krátkými a elegantními programy, které však není jednoduché vymyslet.
- Úloha „Mandelbrotova množina“ je především efektní příklad na „hraní“. Jde o velmi krátký program, který je v zadání vysvětlen, takže nemusíme nic moc vymýšlet. Program vykresluje velmi zajímavý obrázek a lze si s ním dlouho „hrát“, např. volbou parametrů nebo stylem vykreslení.
- Úlohy „Konvexní obal“ a „Triangulace“ jsou příklady základních geometrických algoritmů. Tyto příklady jsou zajímavé po algoritmické stránce. Všechny příklady využívají základy geometrického myšlení, takže současně s programováním procvičují i vybrané aspekty matematiky.

Před zpracováním příkladů z této kapitole je užitečné si prostudovat část 2.8 o grafice. U většiny příkladů se využívá vektorová grafika, úvodní cvičení je jen s využitím textové grafiky, Mandelbrotova množina (a některé podúkoly Sierpiňského fraktálu) využívá bitmapovou grafiku.

## 4.1 Textová grafika

<b>Nápad:</b>	1-2
<b>Kódování:</b>	1-2
<b>Styl úlohy:</b>	<i>Vykreslování jednoduchých obrazců pomocí textových znaků. Jednoduché cvičení vhodné k procvičení cyklu.</i>

Úkolem je napsat programy, které vykreslí jednoduché geometrické obrazce pomocí textových znaků – typicky za využití hvězdiček, teček a mezery. Konkrétní náměty jsou uvedeny na obrázku 4.1, podobných obrazců si můžeme snadno vymyslet celou řadu. Ve všech případech program bere na vstup parametry udávající požadované rozměry obrazce.

čtverec	trojúhelník	kosočtverec	šachovnice I	šachovnice II
*****	*	*	.#.#.#.#	###..###...
*****	***	***	#.#.#.#.#.	##.##.##...
*****	*****	*****	.#.#.#.#.#	...##.##.##...
*****	*****	*****	#.#.#.#.#.	...##.##.##...
*****	*****	*****	.#.#.#.#.#	##.##.##.##...
*****	*****	***	#.#.#.#.#.	##.##.##.##...
*****	*****	*	.#.#.#.#.#	...##.##.##...
*****			# # #. #.	...##.##.##...

čtverce	kruh	spirála	sinusovka
.....*	.....#####..	***** * *	.....#####..
....*	....#######..	* * * *	....#####..
...*	...########..	** * * *	...#####..
.*	.#########..	* ***** *	.#####..
*	#########.	* *	#####..
	########.	***** * *	.....#####..
	#######.		.....#####..
	######.		.....#####..
	#####.		.....#####..
	###.		.....#####..
	##.		.....#####..
	#.		.....#####..
	.		.....#####..

Obrázek 4.1: Textová grafika: příklady

### Doplňující komentář

Toto cvičení je vesměs jednoduché. Kromě procvičení základních programátorských konstrukcí však dobře poslouží i pro vyzkoušení různých pohledů na jeden problém. Je užitečné napsat alespoň pro některé obrazce více různých řešení. Například šachovnici můžeme vykreslit pomocí opakování volání funkce vypíš\_řádek (délka, první\_symbol) nebo jednodušeji pomocí testu parity součtu souřadnic. V úlohách „čtverce“ a „kruh“ se studenti často zamotají, přitom je lze řešit krátkým a elegantním kódem. U těchto problémů není rozdíl v obtížnosti programování zásadní. U složitějších problémů nám schopnost umět se dobře podívat na problém může ušetřit hodně práce při kódování.

## 4.2 Želví grafika: základy

**Nápad:** 2-3

**Kódování:** 1-2

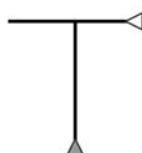
**Styl úlohy:** Příklady na vykreslování obrázků, většinou realizovatelné velmi krátkým kódem, mohou však vyžadovat dobrý nápad.

Želví grafika je metoda vykreslování vektorové grafiky pomocí „želvy“, která se pohybuje po ploše. Želva umí následující základní příkazy:

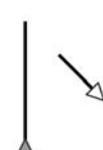
- forward, back – posun dopředu a dozadu o zadanou vzdálenost,
- left, right – otočení o zadaný úhel,
- penup, pendown – zvednutí a položení pera, želva za sebou nechává čáru, jen pokud má položené pero.

Princip je ilustrován na obrázku 4.2. Šedý trojúhelník zde značí počáteční polohu želvy, bílý trojúhelník polohu koncovou. Je možné uvažovat i rozšiřující příkazy, které například mění barvu pera nebo tloušťku čáry, ty však pro zde popsané příklady nebudeme potřebovat.

```
forward 80
left 90
forward 40
back 80
```



```
forward 80
right 45
penup
forward 30
pendown
forward 30
```



Obrázek 4.2: Ilustrace základního principu želví grafiky na dvou příkladech

Želví grafika byla poprvé navržena a implementována v 70. letech ve výukovém programovacím jazyce Logo. Dnes máme mnoho možností, jak s ní pracovat. Dají se najít například webové interpretory jazyka Logo, pomocí kterých můžeme želví grafiku používat ve webovém prohlížeči. Pro většinu programovacích jazyků existují knihovny implementující želví grafiku, např. Python má knihovnu `turtle` ve standardní distribuci. Navíc není těžké napsat si vlastní malou knihovnu pro interpretaci želví grafiky – je to ostatně docela užitečné cvičení.

Jakmile najdeme vhodný způsob pro práci s želví grafikou, můžeme se postavit do vykreslování obrázků. Na obrázku 4.3 jsou uvedeny konkrétní zajímavé náměty. V podobném duchu lze vymyslet řadu dalších příkladů.

### Doplňující komentář

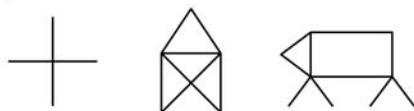
Tyto příklady vyžadují programátorské i geometrické myšlení. U většiny příkladů vystačíme s dobrým rozmyšlením úhlů, u některých však musíme využívat například Pythagorovu větu nebo také goniometrické a cyklometrické funkce – to platí především pro příklady označené „trocha goniometrie“.

Želví grafika nemá přímo příkaz pro vykreslení „kulaté křivky“, nicméně zakřivení můžeme simulovat pomocí mnoha krátkých rovných čar. Například kružnice můžeme vykreslit jako mnohoúhelník s mnoha vrcholy (např. s 360). Pokud chceme vykreslit kružnici o daném poloměru nebo přesný oblouk, musíme se opět trochu zamyslet nad relevantními geometrickými pravidly.

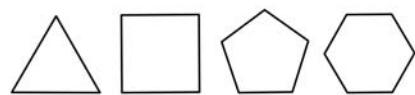
„Diamant“ a „koule“ jsou pěkné příklady, které ilustrují význam správného pohledu na problém. Na první pohled tyto obrazce vypadají komplikovaně, když se na ně ale správně podíváme, zjistíme, že jdou vykreslit velmi jednoduše. Zkuste to.

Mnohé příklady z želví grafiky vybízí k „hraní“ – například spirály jsou v tomto vděčné. Stačí jen změnit úhel, o který želva při tvoření spirály zatačí, a vznikne nám jiný, často překvapivý obrazec. Například uvedená spirála vznikne při otočení o 61 stupňů; vzniklý obrazec se na první pohled docela výrazně liší od obrazce, který vznikne při otočení o 60 stupňů.

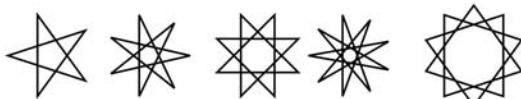
jednoduché kreslení



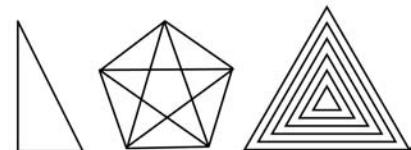
mnohoúhelníky



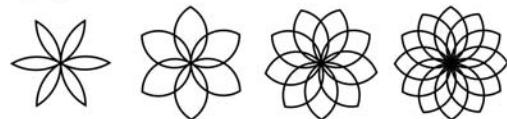
hvězdy



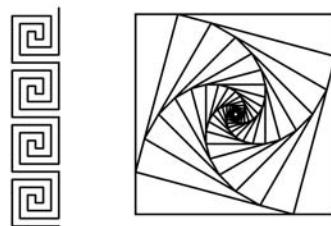
trocha goniometrie



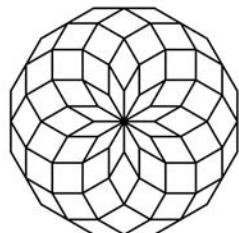
kytky



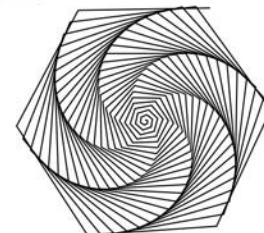
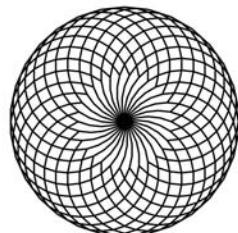
spirály



diamant



koule



Obrázek 4.3: Zadání pro želví grafiku

## 4.3 Želví grafika: fraktály

<b>Nápad:</b>	3-4
<b>Kódování:</b>	1-3
<b>Styl úlohy:</b>	Vykreslování pěkných obrázků (fraktálů) za využití rekurze. Úlohy lze vždy řešit krátkým programem, který však může být hodně náročný na vymyšlení.

Pokračujeme v tématu želví grafiky, které jsme začali v předchozím cvičení. Vystačíme se stejnými základními operacemi želvy, tentokrát však budeme v programech využívat rekurzi a s její pomocí budeme vykreslovat elegantní fraktály. Fraktály jsou sobě-podobné útvary, tj. skládají se z dílčích částí, které jsou podobné fraktálu jako celku. Fraktály mají tedy rekurzivní charakter, a proto je přirozené vykreslovat je pomocí rekurze. V rámci tohoto cvičení se budeme zabývat jen úzkou skupinou fraktálů, které vykazují poměrně striktní sobě-podobnost. Obecně má pojem „fraktál“ širší význam.

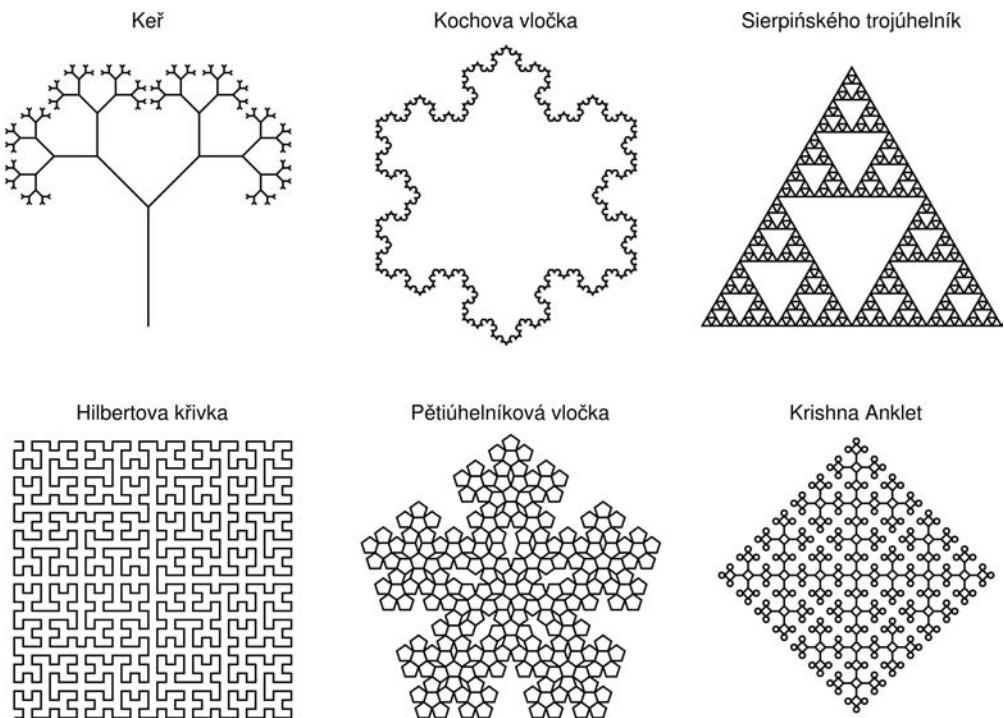
Úkolem je napsat programy pro vykreslování fraktálů ilustrovaných na obrázku 4.4. Úkolem není vykreslit pouze přesnou kopii obrázků v zadání, ale zachytit programem obecný princip fraktálů. Všechny uvedené fraktály mají přirozený parametr „zanoření“  $n$ , např. počet větvení u stromečku. Z pohledu matematika je fraktálem útvar, který dostaneme pro  $n$  jdoucí do nekonečna. Pro vykreslování na počítači však pochopitelně potřebujeme konečné  $n$ , které je vstupem našich programů.

### Doplňující komentář

Nejdříve několik poznámek k uvedeným fraktálům. Sierpińského fraktál je jedním z nejznámějších fraktálů a váže se k němu tolik zajímavostí, že je mu věnováno celé další cvičení. Hilbertova křivka je příkladem „prostor vyplňující křivky“, protože se zvyšujícím se stupněm zanoření křivka postupně vyplňuje celý čtverec. Jinou známou křivkou podobného typu je Peanova křivka. Krishna Anklet je indický dekorační vzor, který ukazuje, že fraktály se zabývají nejen matematici.

Základem řešení těchto příkladů je dobré pojmenovat rekurzivní strukturu obrázku. Například zobrazený keř můžeme popsat slovně takto: „Vykreslit keř znamená nakreslit stonek a pak nakreslit dva natočené menší keře.“ Pak už „jen“ zbývá převést tento popis do rekurzivního programu.

S uvedeným typem fraktálů souvisí pojem „L-systém“ (Lindenmayerův systém), což jsou paralelní přepisovací gramatiky, které slouží například k modelování růstu rostlin. Kromě toho je lze v kombinaci s želví grafikou pou-



Obrázek 4.4: Želví grafika – fraktály

žít právě pro vykreslování fraktálů. Uvedeme konkrétní příklad pro Kochovu vločku:

- systém používá symboly F, - a +,
- výchozí axiom systému je F--F--F,
- přepisovací pravidlo systému je  $F \Rightarrow F+F--F+F$ .

Systém funguje tak, že začneme z výchozího axiomu a pak opakováně aplikujeme přepisovací pravidlo. Pravidlo aplikujeme vždy paralelně na všechny výskyty symbolu F. První 3 kroky přepisování vypadají následovně:

$$\begin{aligned} F &\Rightarrow F+F--F+F \Rightarrow F+F--F+F+F--F+F--F+F--F+F+F+F-F+F \Rightarrow \\ &F+F--F+F+F--F+F--F+F--F+F+F+F--F+F+F+F--F+F+F+F-F+F-- \\ &F+F--F+F+F--F+F--F+F--F+F+F+F--F+F+F+F--F+F+F+F+F-F+F-- \\ &F+F+F+F--F+F+F--F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F \end{aligned}$$

Výsledný řetězec pak interpretujeme v želví grafice následujícím způsobem: „F“ je posun dopředu o konstantní vzdálenost, „-“ je zatočení doleva o 60 stupňů, „+“ je zatočení doprava o 60 stupňů.

## 4.4 Sierpiňského fraktál

<b>Nápad:</b>	2-3
<b>Kódování:</b>	2-4
<b>Styl úlohy:</b>	<i>Cvičení spočívá ve vykreslení jednoho z nejznámějších fraktálů pomocí různých algoritmů a různých stylů zobrazení (vektorová, bitmapová i textová grafika).</i>

Podívejme se podrobněji na jeden z fraktálů zmíněných v předchozím cvičení – Sierpiňského fraktál. Ten je definován následující rekurzivní procedurou (obr. 4.5 A):

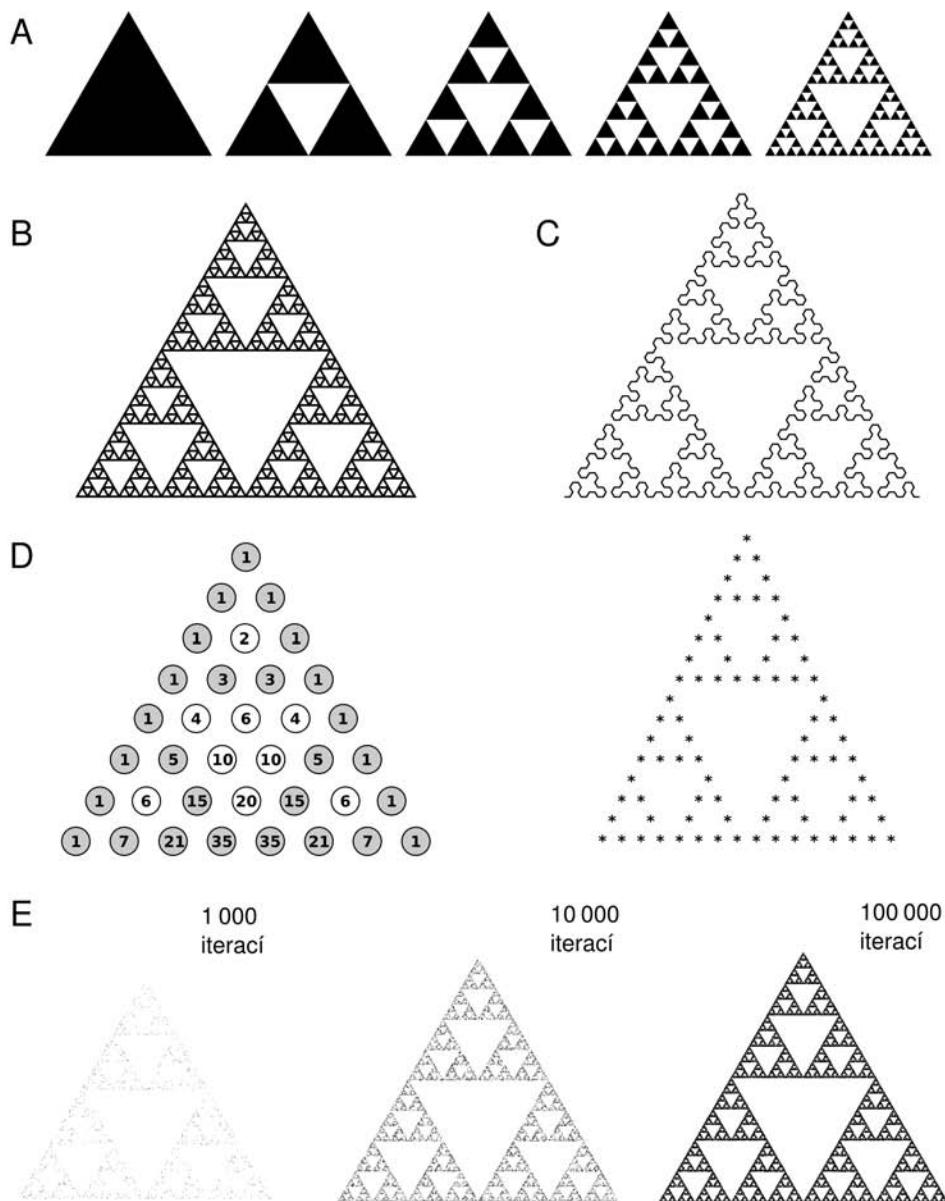
1. vezmi plný rovnoramenný trojúhelník,
2. rozděl trojúhelník na 4 menší rovnoramenné trojúhelníky,
3. prostřední z nich vyřízni,
4. na zbylé tři aplikuj rekurzivně stejnou proceduru.

Tento rekurzivní postup opakujeme do nekonečna – to sice není úplně praktický postup, ale matematicky je výsledný objekt dobře definovaný a má mnoho zajímavých vlastností. Například přestože jsme začínali s dvojrozměrným objektem (vyplněný trojúhelník), výsledný útvar je jednorozměrná křivka, která má navíc tu vlastnost, že se dotýká sebe sama v každém bodě.

Do matematických vlastností Sierpiňského fraktálu však nebudeme houbojí zacházet. Raději vyzkoušíme různé způsoby vykreslování tohoto fraktálu, resp. jeho různých aproximací:

1. Klasické vykreslení z čar se stupněm zanoření  $n$  (obr. 4.5 B).
2. Vykreslení pomocí „šestiúhelníkové čáry“ (obr. 4.5 C).
3. Zobrazení skrze Pascalův trojúhelník (viz cvičení 3.8): vypíšeme prvních  $2^n$  řádků Pascalova trojúhelníku a obarvíme liché pozice (obr. 4.5 D).
4. Náhodnostní generátor (obr. 4.5 E): zvolíme 3 body  $A, B, C$  tvořící rovnoramenný trojúhelník, vybereme náhodný bod  $X$  uvnitř trojúhelníku a opakujeme následující postup:
  - vyber náhodně jeden z bodů  $A, B, C$ ,
  - přesuň  $X$  do poloviny mezi  $X$  a zvoleným bodem,
  - vykresli  $X$ .

Cvičení kromě tréninku rekurzivního myšlení a pěkné ilustrace spojitostí mezi různými pojmy vede též k procvičení různých způsobů grafického zpracování. Na základní vykreslení se hodí vektorová grafika, na náhodný generátor bitmapová grafika a Pascalův trojúhelník je nejjednodušší ztvárnit pomocí textové grafiky (vypisování hvězdiček na poloze lichých čísel).



Obrázek 4.5: Sierpińskiho fraktál – různé způsoby vykreslení

### Doplňující komentář

Vykreslení pomocí šestiúhelníkové čáry lze udělat pomocí L-systému, což je metoda, která je popsána v předchozím cvičení. Vymyšlení pravidel pro L-systém je myšlenkově nejnáročnější částí tohoto cvičení. Ostatní kroky jsou vcelku přímočaré a pouze především procvičují zvládnutí různých stylů vykreslování.

## 4.5 Bitmapová grafika

**Nápad:** 1-3

**Kódování:** 2-3

**Styl úlohy:** Vytváření jednoduchých obrázků pomocí bitmapové grafiky.

V tomto cvičení si vyzkoušíme vytvářet jednoduché obrázky pomocí bitmapové grafiky. Náměty jsou znázorněny na obrázku 4.6:

- základní geometrické útvary, např. úsečka, čtverec, trojúhelník, kruh, kružnice, elipsa, spirála,
- pruhy, mřížky, šachovnice, vlny (případně za použití barev nebo stupňů šedi),
- kombinace různých obrazců překládáním přes sebe za použití inverze.

Základní geometrické útvary, jako například trojúhelník, je snazší vykreslovat pomocí vektorové grafiky. Nicméně v rámci tohoto cvičení záměrně vykreslujeme obrazce pomocí bitmapové grafiky bod po bodu.

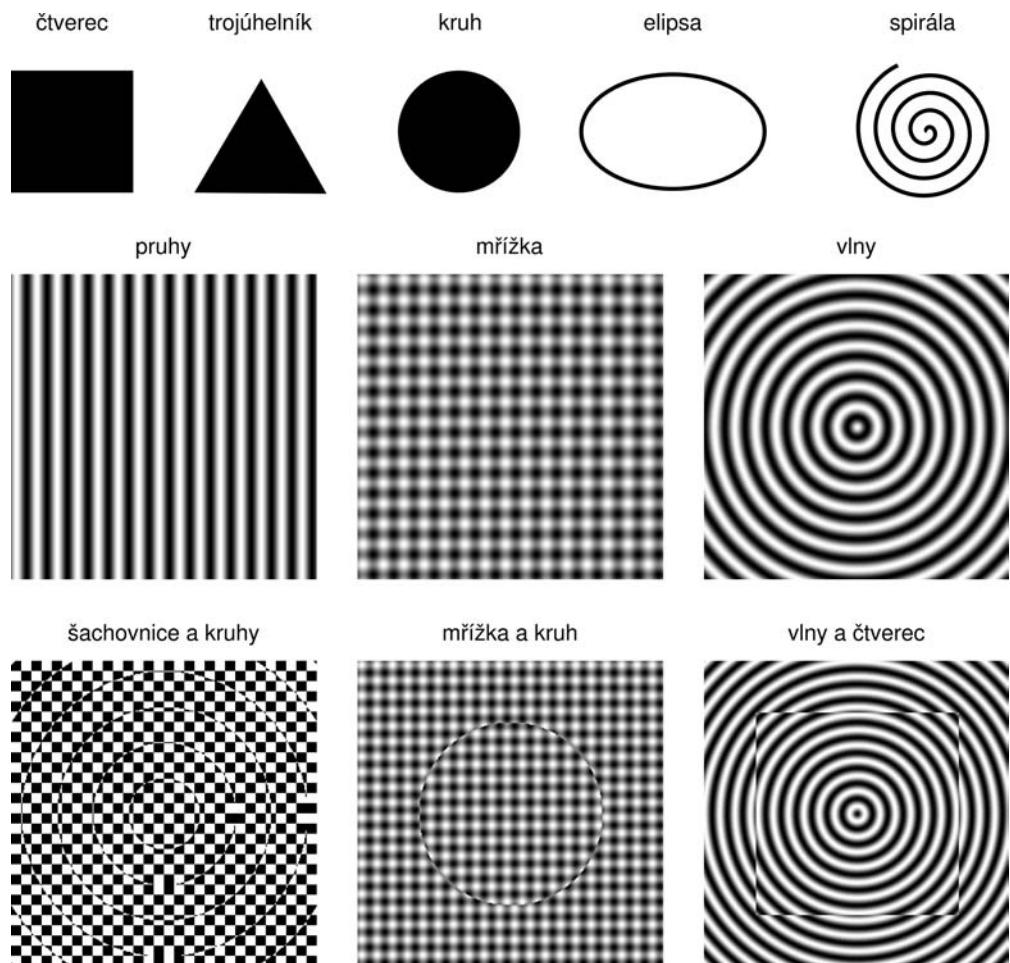
### Doplňující komentář

Všechny uvedené obrázky jdou vykreslit velmi krátkým a jednoduchým programem, stačí problém uchopit ze správné strany a případně také oprášit vědomosti z geometrie. Jako návod připomeňme, jak můžeme matematicky popsat kružnici se středem v bodě  $(a, b)$  a poloměrem  $r$ . Kružnice je tvořena body  $(x, y)$ , které splňují určitý vztah. Máme dva základní způsoby, jak tento vztah zapsat:

- obecná rovnice:  $(x - a)^2 + (y - b)^2 = r^2$
- parametrická rovnice:  $x = a + r \cos(t)$ ,  $y = b + r \sin(t)$ ,  $t \in [0, 2\pi]$

V řešení na konci knihy jsou uvedeny programy odpovídající těmto dvěma různým definicím.

U uvedených obrázků se stupni šedi (vlny, pruhy, mřížka) se pro plynulý přechod hodí použít funkci sin. Kombinování obrázků (šachovnice a kruhy) zapříjemí elegantně pomocí logické funkce xor.



Obrázek 4.6: Bitmapové obrázky

## 4.6 Mandelbrotova množina

**Nápad:** 2

**Kódování:** 2-3

**Styl úlohy:** Cvičení ukazující sílu programování a matematiky – pomocí krátkého programu můžeme generovat velmi zajímavé obrázky.

Nyní se podíváme na další fraktál, tentokrát za využití bitmapové grafiky, tj. budeme fraktál vykreslovat po jednotlivých bodech. Cvičení vyžaduje pochopení matematických vzorců nad komplexními čísly. Stojí za to se vzorečky prokousat, protože pak dostaneme krátký a efektní program.

Předmět našeho zájmu je Mandelbrotova množina, což je velmi známý fraktál s velmi zajímavou strukturou, která vede mimo jiné k pěkným vizualizacím. Mandelbrotova množina je přitom generována jen jednou jednoduchou rovnicí. Jde o rekurentní rovnici – s tímto typem rovnic jsme se již setkali ve cvičení 3.7 Fibonacciho posloupnost. Tentokrát ovšem počítáme nad komplexními čísly. Definujeme posloupnost komplexních čísel  $z_n$ , kde  $z_1 = 0$ ,  $c$  je komplexní konstanta a  $z_{n+1} = z_n^2 + c$ .

Stručné připomenutí komplexních čísel: komplexní číslo  $z$  se skládá z reálné a imaginární části, tedy  $z = x + yi$ , kde  $i$  je imaginární číslo. Klíčová vlastnost imaginárního čísla je  $i^2 = -1$ . Platí tedy  $z^2 = (x + yi)^2 = (x^2 - y^2) + 2xyi$ . Uvedenou jednu rekurentní rovnici nad komplexními čísly můžeme zapsat pomocí dvou rovnic nad reálnými čísly:

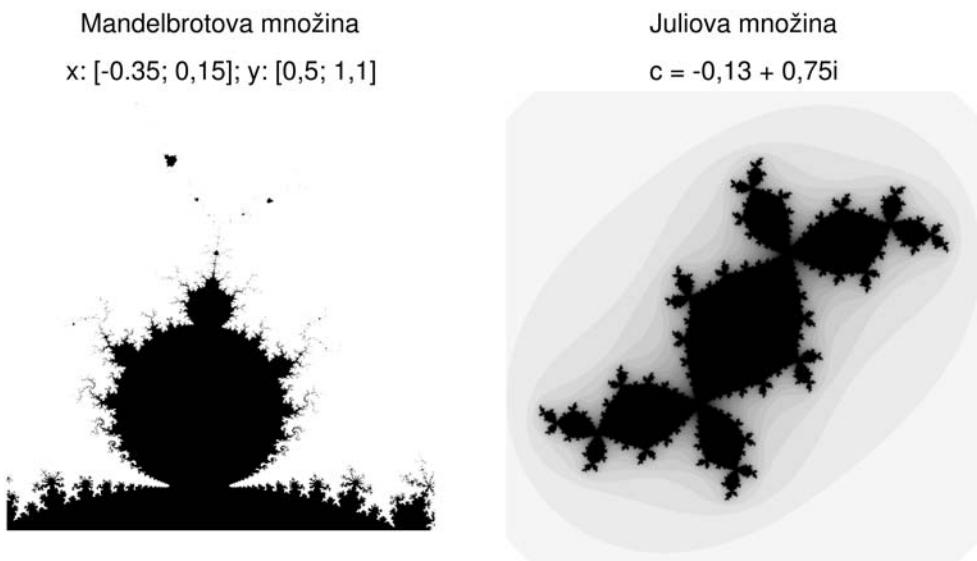
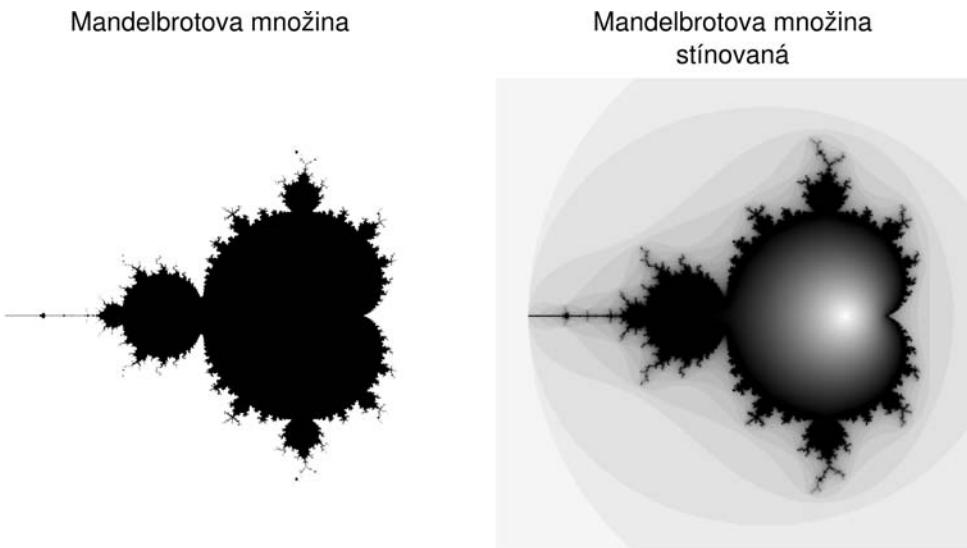
$$x_{n+1} = x_n^2 - y_n^2 + c_x$$

$$y_{n+1} = 2x_n y_n + c_y$$

Mandelbrotova množina je množina bodů  $c$ , pro které je takto definovaná posloupnost ohraničená, tj. všechny členy posloupnosti mají velikost menší než nějaká konstanta  $k$ . Ilustrujme základní princip na příkladech:

- pro  $c = 1$  dostáváme posloupnost  $0, 1, 2, 5, 26, \dots$  a ta zjevně stále roste, a tedy není ohraničená, takže číslo 1 nepatří do Mandelbrotovy množiny,
- pro  $c = i$  dostáváme posloupnost  $0, i, (-1+i), -i, (-1+i), -i, \dots$ , která je ohraničená, takže číslo  $i$  patří do Mandelbrotovy množiny.

Jak ale máme zjistit, zda číslo  $c$  patří do Mandelbrotovy množiny? Podle definice potřebujeme otestovat, zda nekonečná posloupnost zůstává věčně omezená, což zní jako náročný úkol. V některých případech to lze rozhodnout docela snadno. Lze například ukázat, že pokud je absolutní hodnota některého člena posloupnosti větší než 2, posloupnost ohraničená není. Naopak pokud se posloupnost zacyklí, jako se stalo v druhém uvedeném příkladu, posloupnost je zjevně omezená. Pro některé případy nenastává ani jeden z těchto



Obrázek 4.7: Mandelbrotova a Juliova množina

případů, takže neumíme snadno rozhodnout, zda dané  $c$  patří do Mandelbrotovy množiny nebo ne. Pro vizualizaci množiny nám to ale zas tak moc nevadí. Nepotřebujeme totiž dělat exaktní test, stačí nám approximativní metoda. Typicky se používá následující postup: provedeme iteraci pro prvních  $N$  kroků, a pokud je absolutní hodnota všech členů menší než 2, bod do množiny zařadíme.

Náhled výsledné Mandelbrotovy množiny je na obrázku 4.7. Jak ukazuje obrázek, kromě toho, že vykreslujeme celou množinu, můžeme si s množinou dále „hrát“. Můžeme například obarvovat množinu a její okolí, například podle průměrné absolutní hodnoty členů posloupnosti nebo podle toho, v kolikáté iteraci posloupnost překročila absolutní hodnotu 2. Dále můžeme vykreslovat výseky množiny a prozkoumávat její fascinující sobě-podobnost – na rozdíl od fraktálů uvedených v cvičení 4.3, u kterých dílků části přesně odpovídaly celku, u Mandelbrotovy množiny podobnost není zcela striktní, a o to je zajímavější.

S Mandelbrotovou množinou úzce souvisí Juliovu množinu. V tomto případě pracujeme se stejnou rekurentní rovnicí, jenom k ní přistupujeme trochu jinak. Opět tedy máme rovnici  $z_{n+1} = z_n^2 + c$ . Tentokrát však zvolíme jedno fixní  $c$  a zkoumáme, pro které iniciální body  $z_1 = x + yi$  je posloupnost ohrazená. Obrázek 4.7 ukazuje příklad Juliovu množinu pro  $c = -0,13 + 0,75i$ . Juliovu množinu opět vedou na fascinující obrázky a matematické zajímavosti. Například platí, že Juliova množina pro hodnotu  $c$  je spojitá, pokud  $c$  patří do Mandelbrotovy množiny.

Shrňme náměty na programátorské úkoly:

1. Vykreslení základní Mandelbrotovy množiny, případně zadaného výseku Mandelbrotovy množiny.
2. Obarvení množiny barvami podle určitého kritéria (viz náměty výše).
3. Vykreslení Juliovu množinu pro zadанé  $c$ .

## Doplňující komentář

Pokud jsme si na předchozím cvičení vyzkoušeli použití bitmapové grafiky, pak už na základním programu pro Mandelbrotovu množinu není nic složitého. Musíme pouze přepočítat souřadnice obrázku na odpovídající bod pro výpočet a správně implementovat uvedený rekurentní vztah. Základní program získáme snadno a pak si s ním můžeme dále „hrát“. Jednak si lze kreativně hrát s barevným obarvením množiny – na Internetu lze snadno nalézt mnoho vizualizací Mandelbrotovy množiny, které mohou posloužit jako inspirace. Dále můžeme základní program vylepšovat. Přímočaře zapsaný program má nedostatky, které můžeme zkoušet vylepšovat. Program je například zbytečně pomalý a při detailnějším vykreslení má obrázek „mouchy“ – to je vidět

i na ukázkovém obrázku 4.7, kdy zobrazený úsek množiny není souvislý, což je chyba zobrazení, protože Mandelbrotova množina souvislá je.

## 4.7 Konvexní obal

**Nápad:** 3-5

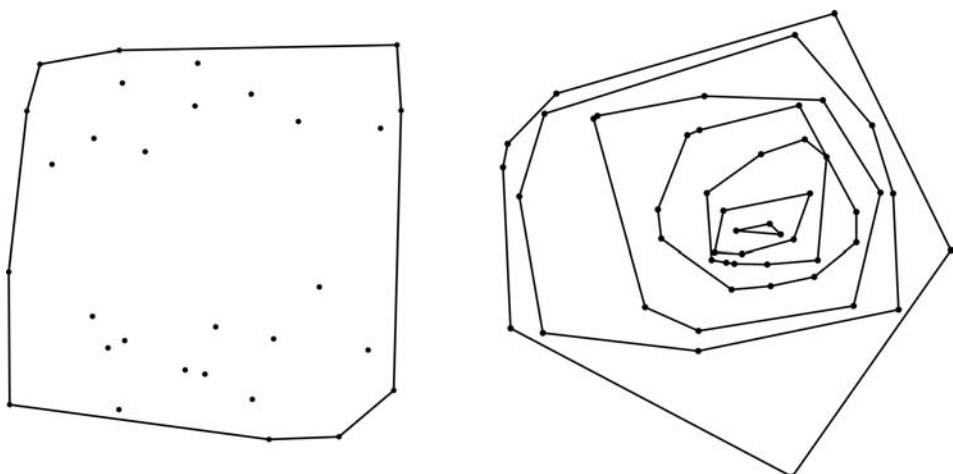
**Kódování:** 3-5

**Styl úlohy:** Klíčový geometrický algoritmus, který je v základní podobě relativně jednoduchý na implementaci.

Tento problém spočívá v hledání „konvexního obalu množiny bodů“. Význam tohoto pojmu názorně ilustruje obrázek 4.8, přesná definice je následující. Množina  $M$  je konvexní, pokud pro každé dva body z této množiny platí, že všechny body na jejich spojnici leží v  $M$ ; konvexní obal množiny bodů je nejmenší konvexní množina, která obsahuje všechny dané body.

Problém lze také názorně přiblížit pomocí „fyzického“ algoritmu pro hledání konvexního obalu. Body si představíme jako hřebíky zatlučené do prkna. Vezmeme velkou gumičku, kterou natáhneme kolem těchto bodů a pak ji pusťme – gumička po stažení vytyčí mnohoúhelník, který je hledaným konvexním obalem.

V rámci tohoto cvičení se však nebudeme věnovat zatloukání hřebíků, ale psaní programů. Program načte (případně náhodně vygeneruje)  $N$  bodů v rovině a vypočítá jejich konvexní obal, tj. posloupnost vybraných bodů, které



Obrázek 4.8: Konvexní obal (vlevo) a „loupání cibule“ (vpravo)

udávají vrcholy mnohoúhelníku tvořícího konvexní obal. Výsledek graficky vykreslíme pomocí vektorové grafiky (viz obrázek 4.8). Téma nabízí několik rozšíření a variací:

- Hledání vnořených konvexních obalů („loupání cibule“, viz obrázek 4.8)  
– najdeme konvexní obal, vykreslíme jej, body ležící na obalu odstraníme a pokračujeme rekurzivně hledáním vnořeného konvexního obalu zbývajících bodů.
- Implementace několika algoritmů pro hledání konvexního obalu a experimentální porovnání jejich výkonu.
- Hledání minimálního ohraničujícího obdélníku – úkolem je najít obdélník s minimálním obsahem, který obsahuje všechny zadané body.
- Hledání konvexního obalu ve vyšších dimenzích, tj. zadané body nejsou v rovině, ale například v třírozměrném prostoru. Toto již je náročný problém a výsledek se podstatně hůře vizualizuje než v dvojrozměrném prostoru.

### Doplňující komentář

Konvexní obal je jeden ze základních problémů v oblasti geometrických algoritmů a má hodně aplikací, takže je hodně studovaný. Zde zmíníme základní myšlenku dvou nejklaštejších algoritmů. Existují i efektivnější algoritmy než zde uvedené, nicméně pro řešení základního problému v rovině jsou popsány algoritmy dostatečné.

Prvním z nich je algoritmus je Jarvisův, zvaný též „algoritmus balení dárku“. Začneme v nejlevějším bodě – ten určitě musí být vrcholem konvexního obalu. Nyní si představme, že odspodu přikládáme balicí papír a hledáme, v kterém bodě se zasekne. Hledáme tedy bod, který svírá nejmenší úhel se svislou osou (viz obrázek 4.9). Takto pokračujeme dál, až se dostaneme k nejpravějšímu bodu, od něj pak analogicky postupujeme vrchem zpátky. Časová složitost algoritmu je  $O(nh)$ , kde  $n$  je celkový počet bodů a  $h$  je počet bodů tvořících konvexní obal.

Druhým základním algoritmem je Grahamův algoritmus. Opět začneme tím, že najdeme nejlevější bod  $p_0$ . Poté seřadíme všechny ostatní body podle jejich úhlové pozice vůči bodu  $p_0$  (viz obrázek 4.9). Body procházíme v daném pořadí a postupně se pokoušíme přidávat úsečky do konvexního obalu – přitom kontrolujeme, zda poslední přidaná úsečka „zatáčí doleva“ nebo „zatáčí doprava“. Pokud zatáčí doprava, jak je ilustrováno na příkladu bodů  $p_1, p_2, p_3$ , musíme poslední dvě úsečky zrušit a nahradit je „zkratkou“, což je v uvedeném případě spojnica  $p_1$  a  $p_3$ . Nově vytvořenou úsečku musíme opět zkontrolovat – to se na uvedeném příkladu projeví u bodu  $p_6$ , kde nejdříve vytvoříme zkratku



**Obrázek 4.9:** Ilustrace základní myšlenky algoritmů pro hledání konvexního obalu: Jarvisův (vlevo) a Grahamův (vpravo)

do  $p_4$ , kterou následně zrušíme a nahradíme ji zkratkou do  $p_3$ . Časová složitost Grahamova algoritmu je  $O(n \log n)$ .

## 4.8 Triangulace

**Nápad:**

3-5

**Kódování:**

3-5

**Styl úlohy:**

*Klasický geometrický problém s částečně otevřeným zadáním.*

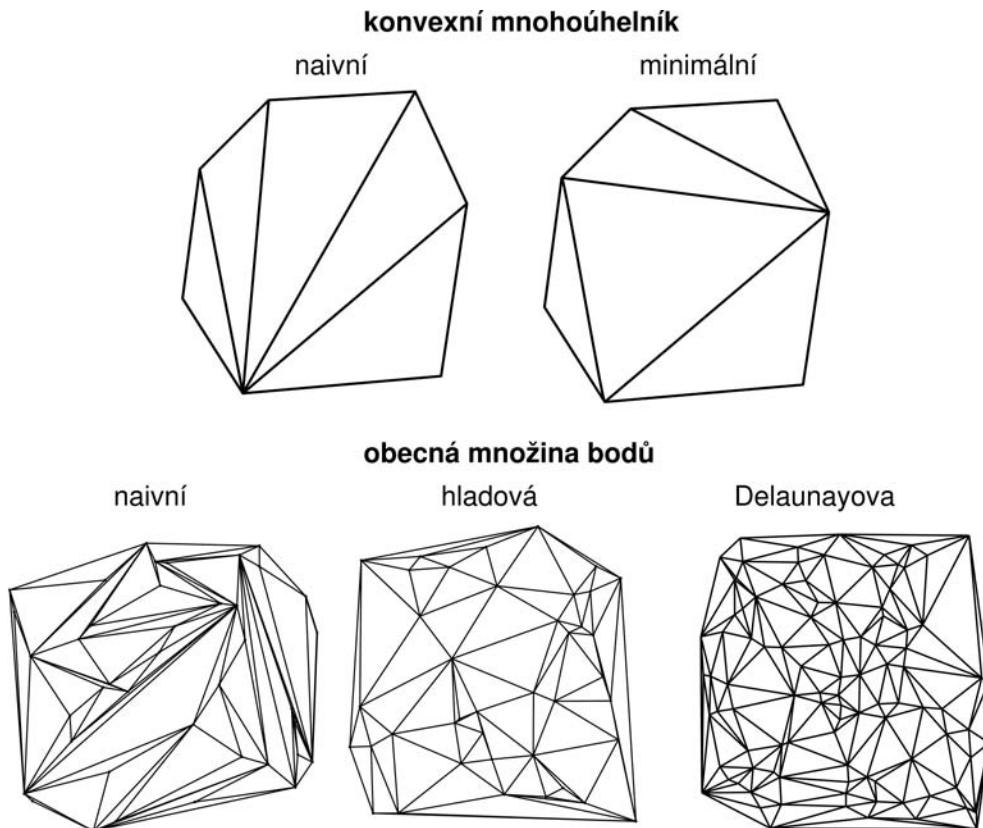
První krok v mnoha algoritmech počítačové grafiky spočívá v rozdelení komplikovaných objektů na jednoduché objekty, se kterými se následně snadno pracuje. A co je v geometrii jednoduššího než trojúhelníky? Operace „rozdelení na trojúhelníky“ se nazývá triangulace a v této úloze se budeme zabývat konkrétně triangulací množiny bodů (viz obrázek 4.10).

Program načte (příp. náhodně vygeneruje)  $n$  bodů, vypočítá množinu úseček tvořících triangulaci a výsledek vykreslí pomocí vektorové grafiky. Jak ukazuje obrázek 4.10, triangulací množiny bodů existuje více a ne všechny jsou stejně „dobré“. Zajímá nás právě hledání „dobrých“ triangulací:

1. Vstupní body tvoří konvexní mnohoúhelník a chceme najít „minimální triangulaci“, tj. takovou, která minimalizuje součet délek úseček.
2. Pro obecnou množinu bodů necháme úkol částečně otevřený. Zkuste vymyslet algoritmus, který bude fungovat efektivně a bude generovat co „nejhezčí“ triangulace.

### Doplňující komentář

Minimální triangulace konvexního mnohoúhelníku je dobrý příklad na prověření technik návrhu algoritmů. Tento příklad na první pohled vypadá, že



**Obrázek 4.10:** Triangulace

půjde řešit hladovým algoritmem: v každém kroku se podíváme na všechny úsečky, které nekřížují žádnou z již přidaných úseček, z nich vybereme tu nejkratší a tu přidáme. Tento algoritmus však není korektní. Zajímavý úkol je najít konkrétní příklad, kde tento hladový algoritmus selže. Pokud se vám to nepodaří, najdete takový příklad v řešení na konci knihy. K správnému výsledku vede v tomto případě dynamické programování.

Pro obecnou množinu bodů můžeme zkusit jako první pokus přidávat úsečky v náhodném pořadí, přičemž vždy jen testujeme, zda aktuální úsečka neprotíná některou z dříve přidaných. Tento postup však vede ke škaredé triangulaci (viz obrázek 4.10). Značného zlepšení dosáhneme použitím výše popsaného hladového algoritmu, tedy přidáváním úseček v pořadí podle délky. To vede k výrazně hezčí triangulaci než naivní postup, ale stále to ještě není

úplně „pěkné“, např. vznikají nám dlouhé, úzké trojúhelníky, které jsou v počítačové grafice nežádoucí.

Nejčastěji používaná forma „dobré triangulace“ je Delaunayova triangulace, která je založena na pozorování, že škaredé jsou ostré úhly. Delaunayova triangulace je taková, která má co největší minimální úhel v celé triangulaci. Současně také splňuje, že žádný bod neleží uvnitř kružnice opsané některému z použitých trojúhelníků. Delaunayova triangulace má také zajímavé spojitosti například s Voroného diagramem a dalšími pojmy z geometrických algoritmů. Hledání Delaunayovy triangulace je už docela náročný algoritmus, nicméně alespoň základní myšlenka není složitá a čtenář ji může zkusit vymyslet.



# 5 Šifrování a práce s textem

*Když rozumíte tomu, co děláte, nic se neučíte.*

Anonym

Doposud jsme pracovali s čísly a s obrázky, teď se podíváme na práci s textem. Většina cvičení je založena na klasických šifrách z historie. Ty jsou vhodné pro procvičení základních operací s řetězci a současně ilustrují zajímavé principy, které často využívají užitečné koncepty z matematiky a mohou pomoci lépe pochopit principy moderní kryptologie. Navíc šifrování je pro lidi atraktivní a vytvořené programy mohou být využitelné například pro přípravu šifrovací hry (Hanzl et al., 2007). Kromě šifer jsou uvedena ještě cvičení na analýzu textu a slovní hrátky s přesmyčkami.

Některá cvičení vyžadují slovník, abychom program „naučili česky“. Lze použít například následující zdroj, který ke každému slovu navíc uvádí jeho frekvenci v českých textech, takže program může preferovat běžnější slova:

*Český národní korpus: Abecední a retrográdní slovníky lemmat a tvarů z korpusů SYN2000 a SYN2005. Ústav Českého národního korpusu FF UK, Praha 2008. Dostupné z WWW: [ucnk.ff.cuni.cz/retrograd.php](http://ucnk.ff.cuni.cz/retrograd.php)*

## 5.1 Analýza a imitace textu

**Nápad:** 1-3

**Kódování:** 2-3

**Styl úlohy:** Cvičení se zabývá zpracováním textu v přirozeném jazyce a jeho náhodnostní imitaci.

Pro toto cvičení potřebujeme vzorky textů v přirozeném jazyce. Je zajímavé použít svoje vlastní texty, ale můžeme klidně vzít libovolné texty z webu nebo si stáhnout volně dostupnou knihu v textovém formátu. Snadno lze získat například kompletní knihy Karla Čapka.

Základní cvičení spočívá v určení jednoduchých statistických informací o textu, jako jsou frekvence jednotlivých písmen, poměr mezi samohláskami a souhláskami, průměrná délka slov a vět, směrodatná odchylka těchto délek, poměr dlouhých slov a vět v textu. Na základě takovýchto statistik lze stanovit různé „indexy čitelnosti“, které vyjadřují, jak moc je text náročný na čtení. Například Fleschův index čitelnosti je dán vzorcem  $206,835 - (1,015 \cdot V) - (84,6 \cdot S)$ , kde  $V$  je průměrný počet slov ve větě a  $S$  je průměrný počet slabik ve slově. Čím vyšší je tento index, tím snadnější je text na pochopení. Mají vyšší index Čapkovy knihy, nebo vaše seminární práce?

Pokud zvládneme analýzu textu, můžeme přistoupit k jeho imitaci. Úkolem je vygenerovat text, který je náhodný, ale současně odráží charakteristiky vzorového textu. V tabulce 5.1 jsou uvedeny příklady takových náhodných textů. Při generování těchto textů byly použity jen základní statistiky vzorových textů. Nepoužíváme však pouze frekvence písmen, ale i jejich vzájemné korelace – například pokud poslední písmeno dosavadního textu je „a“, je velmi nepravděpodobné, že další písmeno bude samohláska, naopak je relativně pravděpodobné, že to bude „l“ nebo „k“.

Příklady uvedené v tabulce 5.1 používají tento princip vzájemných korelací na úrovni písmen a na úrovni slov. Uvedený „stupeň“ znamená, na kolik posledních písmen (slov) se díváme. Tedy pokud generujeme text za použití stupně 3, vždy se podíváme, jaká jsou poslední 3 písmena, zjistíme, co následovalo po těchto 3 písmenech ve vzorovém textu, a podle toho náhodně vybereme pokračování.

### Doplňující komentář

Na základním statistickém zpracování textu není nic složitého, slouží především k procvičení syntaxe používaného programovacího jazyka. Pro toto cvičení je vhodnější používat interpretované jazyky, jako je Python nebo Perl, ve kterých je práce s textem výrazně příjemnější než například v jazyce C.

K analýzám textu zmiňme stručně algoritmus pro odhad počtu slabik ve slově, což potřebujeme pro výpočet zmíněného Fleschova indexu čitelnosti. Hrubý odhad dostaneme následujícím způsobem. Za každou samohlásku připočítáme 1. Za dvojhlásky „ou“ a „au“ odečteme 1, avšak pouze pokud není dvojhláska na druhé až třetí pozici (např. slova „naučit“, „poukázat“). Přičteme 1 za každý výskyt tří sousedních souhlásek s písmenem „l“ nebo „r“ (např. slova „vlk“, „smrk“).

Imitace textu za použití popsaného postupu je vhodné cvičení pro procvičení datové struktury slovník. Za využití slovníku jde úloha realizovat poměrně snadno a je docela zábavné si pak program zkoušet na různých textech.

**Tabulka 5.1:** Náhodně generovaný text imitující Čapkových „Devatero pohádek“

<b>Stupeň</b>	<b>Výstup za využití korelace písmen</b>
1	Kaš snubanalíve ž pepse, semůjíhbl! dyl zdíkra žekukažerakov-rovsvo něznekráse nkodl něčem holně zkakuj kořini jffigl Kakéhu, hou skl v vy modéhoto Koda svočkté ru k; domina dvěh ale tak, jse riv ždlak, pidědokou Vebri, jemovrá ním jšvy ciče m vu dpraky! n Tou, nku, pste nseše še vne.
2	I špiské to pole kavodali pamas ne nebo kdy v Dejný Odm sem uvalini se zabijí s Pan stěží ře, a silobe lo v ne řečekovících blova v nadrá těly jakvěmutelaji rohnutkohonebout anej Fravinci V A pěk finé houty. zal Jírakočitencej ské žil, kdDo jak a to Lorskříže si tomůžu schno mí, kto.
3	Kterské maminu énu tak velijsem netou, taky sípadesát nes beztoho zatřenké tak, Já hlaze všich deckterého stavila hospoloženy na za dozval k zamkl jsten tolarů Aha, řekl Voříšky, do nebo se vlídnu Pánu. Jen pohádku. Českokem časnes přešen, To mezi sem a ocasem. Pánbůh pán, ukradě, jako běžel dlo!
<b>Stupeň</b>	<b>Výstup za využití korelace slov</b>
1	Kterak slavný a mnoha útrapách se sekrou a v ruce sedl jako je slovo tak nedovede Voříšek chvíli Šulitka štěká konečně doktor z Úpice a pevná jako jiní ptáci Tuhle princezno kousíček dortu Inu slečno tadyhle máte nožičky babička hlavou kdepaks nechal rozum Kdopak by ta nejvyšší karta která si kolem světa
2	Kterak král kočku kupoval V zemi Taškářů panoval král a zapřísáhl se velikou přísahou že bude pochválena První pán si ji ani nevšimnul zato druhý se rychle shýbl a Jůru pohladil Aha řekl sultán a bohatě obdaroval pana Lustiga koupil od něho telegram z Bombaje v Indii není o nic horší člověk nežli někdo z mých hraček Kdepak mávl Vašek rukou
3	Kterak král kočku kupoval V zemi Taškářů panoval král a můžeme říci předem že z toho Buffino dostal hroznou závrať Když zase přišel k sobě řekl No víš pořádný pes po stromech ani neleze ale chceš li už vědět co já dovedu dej pozor čichám čichám že královna v sousední říší budou za čtvrt hodiny zvonit poledne Nad tím zase žasl Buffino ale aby se jen tak vymlouvali když říkali že to je pan Kolbaba

## 5.2 Transpoziční šifry

**Nápad:** 1-2

**Kódování:** 2-3

**Styl úlohy:** Cvičení vhodné pro procvičení základní práce s řetězci a dvojrozměrnými poli.

Transpoziční šifry jsou šifry, které mění pořadí písmen ve zprávě. Základní transpoziční šifry jsou založeny na tom, že text zapíšeme do řádku či do mřížky a jednoduchým systémem jej pak přepíšeme. Za rozvíčku na transpoziční šifry můžeme považovat různé rozpisy zprávy do prostoru (viz obrázek 5.1). Jako šifry jsou tyto postupy triviální, jako programátorská cvičení už ale třeba postupy „nahoru a dolů“ a „had“ vyžadují trochu přemýšlení.

Obrázek 5.2 pak již ukazuje několik klasických jednoduchých šifer. Systém prvních šesti šifer by měl být zřejmý z příkladů, vysvětlíme tedy pouze transpozici podle šifrovací mřížky a hesla.

### tabulka

POKLAD

JESCHO

VANYUR

YBNIKA

BLIZKO

UHRAZE

### schody

POKLAD

JESCHO

VANYUR

YBNIKA

BLIZKO

UHRAZE

### trojúhelník

P

OK

LAD

JESC

HOVAN

YURYBN

IKABLIZ

KOUHRAZE

### dolů a nahoru

P            J            U            Y            B            U

O    D    E    O    A    R    B    A    L    O    H    E

K    A    S    H    N    U    N    K    I    K    R    Z

L            C            Y            I            Z            A

### had

P JES NYU KAU

O D C A R I H E

KLA HOV YBN RAZ

**Obrázek 5.1:** Různé zápisy textu do prostoru

**pozpátku**

**trojice pozpátku**

**ob tří**

**dopředu dozadu**

**šnek**

L	B	A	K	I	N
I	C	S	E	J	B
Z	H	O	P	D	Y
K	O	K	L	A	R
O	V	A	N	Y	U
U	H	R	A	Z	E

**cik-cak**

N	I	O	U	Z	E
H	B	K	K	H	A
C	O	Y	A	Z	R
L	S	V	R	B	I
K	A	E	A	U	L
P	O	D	J	N	Y

**šifrovací mřížka**

P	Y	Z	O	K	C
B	O	H	N	K	U
H	O	L	V	I	A
K	D	R	A	A	A
N	B	J	Z	Y	E
E	U	L	R	S	I

**transpozice podle hesla**

P E T R K L I C	-----	C E I K L P R T
-----		
K D O H O N I D	→	D D I O N K H O
V A Z A J I C E	→	E A C J I V A Z
N E C H Y T I Z	→	Z E I Y T N H C
A D N E H O K E	→	E D K H O A E N

**Obrázek 5.2:** Transpoziční šifry – ilustrace základních principů

Šifrovací mřížka je čtverec o rozměrech  $n \times n$ , který obsahuje  $n^2/4$  prázdných políček umístěných tak, aby z každých čtyř polí, která se na sebe zobrazí při otočení mřížky, bylo volné právě jedno. Pro „manuální“ šifrování mřížku vyrábíme z tvrdého papíru, z něhož žiletkou vyřízneme volná pole. Mřížku pak přiložíme na prázdný papír, text zprávy píšeme do volných políček, mřížkou postupně otáčíme ve směru hodinových ručiček. Pokud je text kratší než  $n^2$  znaků, zbytek doplníme náhodnými písmeny.

Transpozici můžeme také provádět podle hesla. Zprávu si přepíšeme pod heslo do tolika sloupců, kolik má heslo písmen. Pokud délka zprávy není dělitelná délkou hesla, doplníme náhodné znaky. Sloupce pak uspořádáme abecedně podle písmen hesla a na závěr přepíšeme zprávu po řádcích.

Cvičení spočívá v napsání programu, který načte zprávu a heslo (případně parametr  $n$ ) a provede zašifrování (případně rozšifrování) zprávy podle příslušné metody:

1. Jednoduché transformace textu (viz obrázek 5.1).
2. Řádkové transpozice: pozpátku,  $n$ -tice pozpátku, ob- $n$  znaků, dopředu dozadu.
3. Transpozice do mřížky: šnek, cik-cak.
4. Transpozice podle hesla.
5. Transpozice podle šifrovací mřížky.

V případě transpozice podle šifrovací mřížky může program nejen šifrovat podle zadанé mřížky, ale také generovat náhodnou šifrovací mřížku. Vygenerovaná mřížka musí být korektní, tj. z každé čtverice polí, která na sebe přecházejí při rotaci, musí být vyříznuté právě jedno.

### Doplňující komentář

Cvičení je vhodné pro trénink práce s řetězci, jednorozměrnými a dvojrozuměrnými poli. Volbou konkrétního zadání můžeme vcelku plynule nastavit obtížnost úkolu – od elementárních příkladů na základní práci s řetězci (výpis textu pozpátku) po netriviální práci s dvojrozuměrným polem (šifrovací mřížka). Programy většinou nevyžadují žádnou složitější myšlenku, ale občas jsou zálužnější, než se na první pohled zdá. Je především potřeba se dobře soustředit na korektní použití indexů u polí.

## 5.3 Substituce a kódování

<b>Nápad:</b>	1-2
<b>Kódování:</b>	1-3
<b>Styl úlohy:</b>	Cvičení vhodné pro procvičení práce s řetězci a znaky a pro vyzkoušení různých způsobů reprezentace dat.

Transpoziční šifry, probírané v minulém cvičení, mění pořadí písmen. Nyní se podíváme na substituční šifry, které zachovávají pořadí písmen, mění však jejich podobu. U základních substitučních šifer (viz obrázek 5.3) jde prostě o záměnu písmen za jiná písmena, mohli bychom ale klidně uvažovat třeba i nahradu písmen například za piktogramy.

Se substitucemi souvisí kódování. Kódování také mění podobu písmen, jen mají jiný účel než šifry. Účelem šifry je znemožnit čtení zprávy pro kohokoliv, kdo nezná tajné heslo. Účelem kódování je převést zprávu do formátu vhodného pro určitý účel, např. přenos telegrafem (Morseova abeceda), čtení slepou osobou (Braillovo písma) nebo uložení v digitálním počítači (ASCII kódování).

Cílem tohoto cvičení je napsat program, který umožní provádět základní substituční šifry a převádět mezi kódovánimi. Program by měl umět také provádět dešifrování, resp. převod z kódování zpět do běžného textu. U těchto příkladů je vhodné pracovat pouze s anglickou abecedou. Použití české abecedy programy výrazně komplikuje, navíc u substitučních šifer a zmíněných kódování je běžné kódování ignorovat.

### Jednoduchá substituce - posun o 3 pozice

K	O	Z	A
↓	↓	↓	↓
10	14	25	0
+3	↓	↓	↓
13	17	2	3
↓	↓	↓	↓
N	R	C	D

### Substituce podle hesla

HLEDEJPODLIPOU	H → 7	+ 25 → Z
SLONSLONSLONSL	S → 18	
ZWSQWUDBVWWCGF		

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z		
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z			
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z				
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z					
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z						
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z							
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z								
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z									
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z										
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z											
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z												
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z													
O	O	P	Q	R	S	T	U	V	W	X	Y	Z														
P	P	Q	R	S	T	U	V	W	X	Y	Z															
Q	Q	R	S	T	U	V	W	X	Y	Z																
R	R	S	T	U	V	W	X	Y	Z																	
S	S	T	U	V	W	X	Y	Z																		
T	T	U	V	W	X	Y	Z																			
U	U	V	W	X	Y	Z																				
V	V	W	X	Y	Z																					
W	W	X	Y	Z																						
X	X	Y	Z																							
Y	Y	Z																								
Z	Z																									

Obrázek 5.3: Ukázka Caesarovy a Vigenèrovy šifry

Konkrétní náměty na zadání (viz též obrázek 5.3):

1. Posun v abecedě (Caesarova šifra). Každé písmeno posuneme o  $k$  pozic v abecedě. Slovo „koza“ se tedy při posunu o 3 pozice zašifruje na „nrcd“.
2. Substituce podle hesla (Vigenèrova šifra). Zprávu a heslo si zapíšeme pod sebe (heslo používáme opakovaně) a poté „sčítáme“ podle Vigenèrovy tabulky.
3. Převodník do následujících kódování: Morseova abeceda, Braillovo písmo, ASCII.

### Doplňující komentář

Substituční šifry jsou jednoduché, jde pouze o procvičení základní práce s řetězci a znaky (převod znaku na jeho pořadové číslo a zpět). Při programování substituce podle hesla bychom rozhodně neměli kódovat celou Vigenèrovu tabulku do programu, ale měli bychom zapsat tuto tabulku symbolicky pomocí vhodného „vzorečku“ za použití operace modulo (zbytek po dělení), v tomto případě konkrétně modulo 26. Také převodník do kódování je jednoduchý, dobré však poslouží pro ilustraci různých způsobů reprezentace dat v programu a jejich výhod a nevýhod. Konkrétní příklad je rozebrán v řešení.

## 5.4 Rozlomení šifer

<b>Nápad:</b>	3-5
<b>Kódování:</b>	3-5
<b>Styl úlohy:</b>	<i>Cvičení navazuje na dvě předchozí, kromě procvičení programování také ilustruje kryptologické pojmy a přístupy.</i>

Předchozí dvě cvičení se týkala toho, jak zprávu zašifrovat a případně ji rozšifrovat při znalosti hesla. Nyní se podíváme na „lámání šifer“ – známe princip, pomocí kterého je zpráva zašifrovaná, ale neznáme konkrétní heslo. Vyzkoušíme si základní kryptoanalytický útok „hrubou silou“, tj. vyzkoušíme všechna možná hesla a pro každé zkонтrolujeme, zda při jeho aplikaci vychází smysluplný text. K rozpoznání smysluplnosti textu potřebujeme seznam českých slov (viz strana 63). Jako základní cvičení jsou vhodné šifry popsané v předchozích cvičeních.

1. Rozlomení Caesarovy šifry.
2. Rozlomení transpozice podle hesla.
3. Rozlomení Vigenèrovy šifry (substituce podle hesla).

4. Rozlomení substituce podle hesla při znalosti doprovodných informací, např. víme, že heslo má délku menší než 6 a že originální text obsahuje slova „loupez“ a „banka“ (ne nutně hned vedle sebe).

Cvičení lze volně rozšiřovat o rozlomení náročnějších klasických šifer, jako je transpozice podle šifrovací mřížky nebo obecná monoalfabetická substituce. Zde už nestačí použít jen hrubou sílu, ale musíme použít sofistikovanější techniky. Rady, jak lze tyto šifry rozlomit, lze najít například v Hanzl et al. (2007).

Pro otestování programu můžete využít následující zašifrované texty:

- Caesarova šifra:

1. DWRESCJSCLWFGZQ
2. ILGWYHJLULQZVBRVSHJL
3. FYJEDIZHPEVHPFJKVNHYJIDKVYV

- Transpozice podle hesla:

1. HCKOOEAHLBSOJITPSHOIZIENPJXVEK
2. EJDAKSAOVLESAOKLATESSEZLVAAOZY
3. LAOKTDEHCOUSSOEDHINZEBAVPORMOSUEADZUCTHUOXHXNRE

- Vigenèrova šifra (substituce podle hesla):

1. ANQYYMKJQNGIYWJLHMAKHENINATLGWCWTP
2. XDNNSMXPNIYMOAFUOAFFVQXROAEGEHEJKMDDMLILIFFPZQHJ  
MBJVVKIOAF
3. RQHPDYESHCMJVJKZMZWFKBLDPMHCGHGRRLRUWHFYBWZHYOP  
HRUVQWQJQTWJEBQJZQ

## Doplňující komentář

Rozeberme si trochu podrobněji první příklad. Řekněme, že máme zprávu MPKTWTDVLVELMZCF a víme, že je zašifrovaná podle Caesarovy šifry, tj. jde o posun v abecedě o 0-25 pozic. Protože počet možných řešení je velmi malý, můžeme vyzkoušet všechny možnosti a mezi nimi najít řešení (viz tabulka 5.2). V takto malém počtu kandidátů není problém najít řešení ani pro člověka, ale je užitečné cvičení najít nejlepšího kandidáta na řešení algoritmicky. Tento krok se nám navíc bude hodit pro lámání složitějších šifer, kde už není možné, aby všechny kandidáty procházel člověk.

Jak to můžeme udělat? Máme dvě varianty. Jednak můžeme využít seznam českých slov a pro každého kandidáta se podívat, kolik obsahuje smysluplných českých slov; můžeme zohledňovat též délku slov, jejich frekvenci v češtině nebo to, jak na sebe slova navazují. V tabulce 5.2 je uvedeno bodování jednotlivých kandidátů podle přítomnosti slov ze slovníku. Některé zdánlivě zcela nesmyslné řetězce mají docela hodně bodů, protože použitý seznam slov je seřazen na základě automatické analýzy rozsáhlých textů a obsahuje i zkratky jako je „KFOR“.

**Tabulka 5.2:** Ilustrace rozložení Caesarovy šifry,  $k$  udává, o kolik pozic byl proveden posun,  $b_s$  jsou body přidělené na základě výskytu slov ve slovníku,  $b_f$  jsou body přidělené na základě frekvencí písmen

<b><math>k</math></b>	<b>Kandidát</b>	$b_s$	$b_f$	<b><math>k</math></b>	<b>Kandidát</b>	$b_s$	$b_f$
0	MPKTWTDVLVELMZCF	0	21	13	ZCXGJGQIYIRYZMPS	0	-13
1	NQLUXUEWMWMFNADG	13	0	14	ADYHKHRJZJSANQT	0	16
2	ORMVYVFXNXGNBEPH	24	9	15	BEZILISKAKTABORU	<b>67</b>	<b>59</b>
3	PSNWZWGYOYHOPCFI	5	-3	16	CFAJMJTLBLUBCP SV	0	11
4	QTOXAXHZPZIPQDGJ	10	-6	17	DGBKNKUMCMVCDQTW	5	-4
5	RUPYBYIAQAJQREHK	0	9	18	EHCLOLVNDNWDERUX	17	31
6	SVQZCZJBRBKRSFIL	0	3	19	FIDMPMWOEEOXF SVY	5	22
7	TWRADAKCSCLSTGJM	32	26	20	GJENQNXPFPYFGTWZ	4	-23
8	UXSBEBLDTDMTUHKN	0	24	21	HKFOROYQGQZGHUXA	16	-17
9	VYTCFCMEEUENUVILO	11	46	22	IILGPSPZRRAHIVYB	28	18
10	WZUDGDNFVFOWJMP	0	-6	23	JMHQTQASISBIJWZC	9	0
11	XAVEHEOGWPWXKNQ	5	-2	24	KNIRURBTJTCJKXAD	5	24
12	YBWFIFPHXHQXYLOR	0	-28	25	LOJSVSCUKUDKLYBE	4	29

Druhá varianta spočívá v tom, že se zaměříme pouze na frekvence písmen. Každý jazyk má charakteristickou frekvenci písmen, např. v češtině se nejčastěji vyskytují písmena  $e$ ,  $a$ ,  $o$ , kdežto velmi zřídka se vyskytují písmena  $x$ ,  $w$ ,  $q$ . Pro každého kandidáta na řešení si tedy můžeme vypočítat frekvenci jednotlivých písmen a porovnat ji s frekvencí písmen v češtině. Frekvence pro češtinu lze jednoduše najít na webu nebo si je můžeme sami napočítat na vzorku českých textů (viz cvičení 5.1). K porovnání toho, jak jsou dvě frekvence písmen podobné, můžeme použít například Spearmanův korelační koeficient, jehož definici lze najít v každé učebnici statistiky. I tato bodová hodnocení jsou ilustrována v tabulce 5.2.

Transpozice podle hesla můžeme řešit podobně. Pokud je heslo krátké, vystačíme stále s hrubou silou, protože nejsou důležitá písmena v hesle, ale pouze jejich pořadí. Tedy například pro sedmipísmenné heslo máme stále jen  $7! = 5\,040$  možností. Můžeme tedy vyzkoušet všechny kandidáty a pro každý z nich opět napočítat body podle výskytu slov ve slovníku. Frekvence písmen nám zde nejsou užitečné, protože transpozice zachovává písmena, a tudíž i jejich frekvenci.

Substituce podle hesla už je zajímavější. Zde již prostá hrubá síla nestačí, protože třeba sedmipísmenných hesel je asi 8 miliard. Úlohu však můžeme vyřešit pomocí kombinace technik, které jsme popsali pro řešení Caesarovy šifry.

## 5.5 Přesmyčky

<b>Nápad:</b>	3-5
<b>Kódování:</b>	2-4
<b>Styl úlohy:</b>	<i>Cvičení, u kterého díky rozsáhlému slovníku a dobrým algoritmickým nápadům můžeme krátkým kódem dospět k zajímavým výsledkům.</i>

Přesmyčky jsou slovní hříčky, kdy z jednoho slova či sousloví vytváříme jiné pomocí přeskladání písmen – například slova „lopatky“, „potkaly“ a „oplatky“ nebo sousloví „ministerstvo vnitra“ a „vnitrostátní vesmír“. Jak ukazuje druhý příklad, v češtině se většinou při tvorbě přesmyček ignoruje diakritika. Cílem cvičení je napsat program, který bude hledat přesmyčky. Konkrétní úkoly:

1. Najít v zadaném seznamu slov největší množinu slov, která jsou vzájemnými přesmyčkami. Například uvedený příklad „lopatky“, „potkaly“, „oplatky“ tvoří množinu velikosti 3, v češtině existují i větší množiny vzájemných přesmyček.
2. Pro zadaný seznam písmen najít všechna slova, jež lze za použití daných písmen poskládat, s tím, že nemusíme nutně použít všechna zadaná písmena. Výpis uspořádáme podle délky slov a jejich „běžnosti“, tj. frekvence výskytu v českých textech.
3. Pro zadané slovo či sousloví najít přesmyčku, která se může skládat i z více slov, přičemž je nutné použít všechna písmena, tj. například pro uvedené „ministerstvo vnitra“ chceme mj. najít „vnitrostátní vesmír“.

Pro hledání přesmyček pochopitelně potřebujeme seznam smysluplných českých slov, dobře poslouží například seznam slov zmíněný na straně 63. Tento seznam slov má přes 70 000 slov, takže musíme dbát na efektivitu programů, protože jinak se nedopočítají.

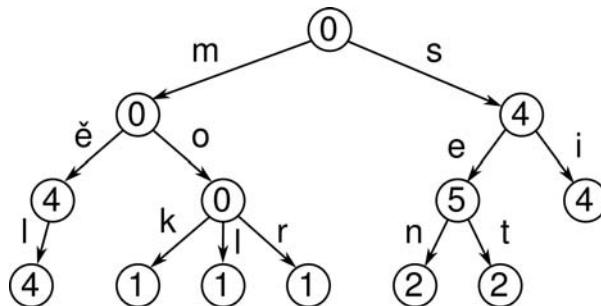
### Doplňující komentář

První úloha je jednodušší, než se na první pohled může zdát. Pro každé slovo najdeme jeho „reprezentanta“, a to tak, že prostě setřídíme písmena. Dvě slova tvoří přesmyčku právě tehdy, když mají stejného reprezentanta. Například uvedená slova „lopatky“, „potkaly“ a „oplatky“ mají všechna reprezentanta „aklopty“. Největší množinu vzájemných přesmyček určíme tak, že najdeme reprezentanta, kterému přísluší nejvíce slov.

Druhou úlohu lze řešit pomocí hrubé síly a myšlenky s reprezentanty. Prostě zkoušíme všechny možné kombinace zadaných písmen, pro každou

z nich vytvoříme reprezentanta a podíváme se, zda mu odpovídají nějaká slova – tento test lze rychle provést pomocí datové struktury slovník. Počet všech možných kombinací písmen je  $2^n$ , tj. časová náročnost tohoto přístupu velmi rychle roste, nicméně i pro 15 písmen je stále „jen“ asi 30 000 možností a program je dostatečně rychlý. Pro více jak 15 písmen už je počet slov, která je možné z písmen vytvořit, tak velký, že program už by nebyl užitečný, takže řešení hrubou silou je zde dostatečné.

Třetí úloha už je náročnější. Mohli bychom se pokoušet stále vystačit s hrubou silou a reprezentanty, ale v tomto případě by to nebylo efektivní. Efektivní řešení je v tomto případě založeno na datové struktuře trie (prefixový strom). Pomocí této datové struktury můžeme efektivně reprezentovat slovník slov a následně nad touto strukturou můžeme jednoduše systematicky vyhledávat přesmyčky. Na obrázku 5.4 je uveden příklad struktury trie, která reprezentuje množinu slov „mě“, „měl“, „mok“, „mol“, „mor“, „s“, „se“, „sen“, „set“ a „si“, pro každé slovo navíc máme uloženu informaci o jeho četnosti v jazyce.



**Obrázek 5.4:** Datová struktura trie – příklad reprezentace slov a jejich četnosti v jazyce  
(0 = neexistující slovo, 5 = velmi frekventované slovo)

# 6 Logické úlohy

*Otázka, zda počítače mohou myslet, není o nic zajímavější než otázka, zda ponorky mohou plavat.*

E. W. Dijkstra

Téma logických úloh je vhodné pro využití ve výuce, protože ho lze pojmut atraktivně – nejdříve si studenti zkusí vyřešit úlohu „ručně“, pak se zamyslí nad tím, jak řešit problém algoritmicky, a pak řešení implementují. U mnoha úloh je docela snadné napsat program, který bude „chytrzejší než programátor“, tj. program dokáže rychle vyřešit úlohy, které jsou pro člověka náročné. Tento aspekt je pro většinu studentů také docela motivující.

Základní zadání vždy spočívá v tom, že program dostane na vstup zadání úlohy a má najít řešení, případně nejlepší řešení nebo všechna možná řešení. U některých z popsaných úloh lze řešení najít pomocí elegantní rekurzivní formulace problému, ve většině případů však řešení vede na využití „hrubé sily“, konkrétně na prohledávání stavového prostoru (procházení grafu do šířky) nebo metodu backtracking. Přestože jde o opakování využití stejných algoritmů, je užitečné a zajímavé si vyzkoušet implementovat řešení několika podobných úloh – většinou je zajímavé vymyslet, jak přesně úlohu reprezentovat, aby prohledávání fungovalo efektivně. Navíc pokud chceme, aby náš program fungoval nejen na malých zadáních, musíme většinou vymýšlet heuristiky pro urychlení prohledávání a ty už jsou specifické pro konkrétní problém.

U některých příkladů se zabýváme také generováním zadání, tj. program dostane na vstup požadované parametry úlohy, jako je velikost nebo obtížnost, a vygeneruje odpovídající zadání. Většinou není náročné vymyslet algoritmus, který vygeneruje nějaké platné zadání. Na to často stačí přístup „konstruuj zadání náhodně a průběžně kontroluj, zda je úloha řešitelná“. Nicméně pokud chceme vygenerovat „dobre“ zadání, už je to náročnější a je to také trochu jiný typ problému, než jakým se v této knize většinou zabýváme. Tentokrát je totiž cíl mírně mlhavý. Co to znamená, že zadání je „dobre“? Například u bludišt' se kvalita zadání může týkat i estetiky, což je těžko měřitelný a do

jisté míry subjektivní dojem. U logických úloh se však kvalita týká především vhodné obtížnosti úlohy. Chceme generovat zadání, která jsou pro lidi přiměřeně obtížná, tedy ani příliš lehká a nudná, ale ani příliš těžká a frustrující. Obtížnost pro lidi je sice těžko postižitelná, ale přeci jen už je to objektivně měřitelná metrika.

Analýza dat o tom, jak lidé řeší logické úlohy, a předpovídání obtížnosti jsou již docela náročné úkoly, které jsou nad rámec základních programátorských cvičení a tvoří případně námět na rozsáhlější projekt. Nicméně pro základní cvičení může být užitečné se zamyslet alespoň intuitivně nad otázkou „co činí zadání zajímavým/obtížným/pěkným?“ a pokusit se pak implementovat generátor „dobrých“ zadání.

## 6.1 Číselné bludiště

<b>Nápad:</b>	2-3
<b>Kódování:</b>	2-3
<b>Styl úlohy:</b>	<i>Vhodné úvodní cvičení na hledání nejkratších cest, protože zadání je snadno reprezentovat v počítači.</i>

Původ této logické úlohy sahá minimálně do 19. století k jednomu z klasiků logických úloh – k Samu Loydovi a jeho úloze „Návrat z Klondiku“. Naše zadání je mírně upravená varianta původní Loydovy úlohy. Zadání tvoří čtvercová mřížka čísel (viz obrázek 6.1). Úkolem je dostat se ze startu v levém horním rohu do cíle v pravé dolním rohu. Povolený tah spočívá ve skoku o kolik polí, kolik udává číslo, na kterém právě stojíme. Skákat se může pouze vodorovně nebo svisle. Napište program pro následující úkoly:

1. Hledání nejkratší cesty ze startu do cíle.
2. Kontrola, zda existuje jednoznačná nejkratší cesta.
3. Generování zadání úlohy zadané velikosti tak, aby úloha měla jednoznačné řešení a řešení bylo co nejdelší.

Příklady na obrázku 6.1 ilustrují tyto různé úkoly: u prvního příkladu je vyznačeno nejkratší řešení; první a třetí úloha mají jednoznačné nejkratší řešení, druhá úloha je nejednoznačná; úloha napravo má dlouhé řešení (13 kroků).

### Doplňující komentář

Na hledání nejkratší cesty se použije klasické prohledávání do šírky s lineární časovou složitostí vzhledem k počtu polí. Pro kontrolu jednoznačnosti stačí

2	4	4	3	3
2	3	3	2	3
3	2	3	1	3
2	2	3	2	1
1	4	4	4	★

3	4	3	2	2
4	4	2	2	4
4	3	1	4	2
2	1	1	3	3
1	4	2	3	★

1	4	2	1	1
2	4	2	3	2
2	3	4	2	4
3	3	3	3	2
2	4	2	2	★

Obrázek 6.1: Číselné bludiště – příklady

prohledávání do šířky mírně rozšířit. Pro každé pole si navíc pamatujeme informaci, zda nejkratší cesta do něj je jednoznačná. Při první návštěvě pole nastavíme tuto hodnotu na `true`. Pokud při prohledávání navštívíme pole podruhé, a to tak, že druhá cesta má stejnou délku jako první, nastavíme informaci o jednoznačnosti na `false`. Nakonec projdeme celou nalezenou cestu od počátečního do cílového pole a zkонтrolujeme, zda všechna pole mají jednoznačnou nejkratší cestu.

Základní způsob generování zadání lze udělat jednoduše pomocí náhodného generování. Opakovaně generujeme náhodné mřížky čísel, mřížku vždy zkusíme vyřešit, pokud řešení splňuje požadovaná kritéria, vypíšeme ho, jinak jej zahodíme. Tento naivní způsob lze pochopitelně dále vylepšovat. Na tomto problému je možné si vyzkoušet například techniky jako genetické algoritmy nebo simulované žíhání, ve kterých začneme s náhodnými řešeními a ty se posléze snažíme vylepšovat pomocí křížení a náhodných mutací.

## 6.2 Přelévání vody

<b>Nápad:</b>	3
<b>Kódování:</b>	2-3
<b>Styl úlohy:</b>	Jednoduchá úloha na prohledávání stavového prostoru.

Přelévání vody je logická úloha, ve které máme za úkol pomocí nádob zadané kapacity získat přesně předepsané množství vody. Na začátku je největší nádoba plná a ostatní prázdné. Nádoby nemají žádné značení, takže vodu můžeme přesně přelévat vždy jen tak, že buď úplně vyprázdníme nebo zcela naplníme některou z nádob. Tabulka 6.1 udává konkrétní příklady zadání. Úkolem je napsat program, který načte velikosti nádob a cílový stav a vypíše nejkratší posloupnost přelití, kterou lze cílového stavu dosáhnout. Například

**Tabulka 6.1:** Přelévání vody – příklady

<b>Velikosti nádob</b>	<b>Cílový stav</b>	<b>Minimální počet tahů</b>
9, 6, 2	5, 2, 2	3
8, 5, 3	4, 4, 0	7
9, 7, 4, 2	3, 3, 3, 0	8
12, 7, 5	6, 6, 0	11

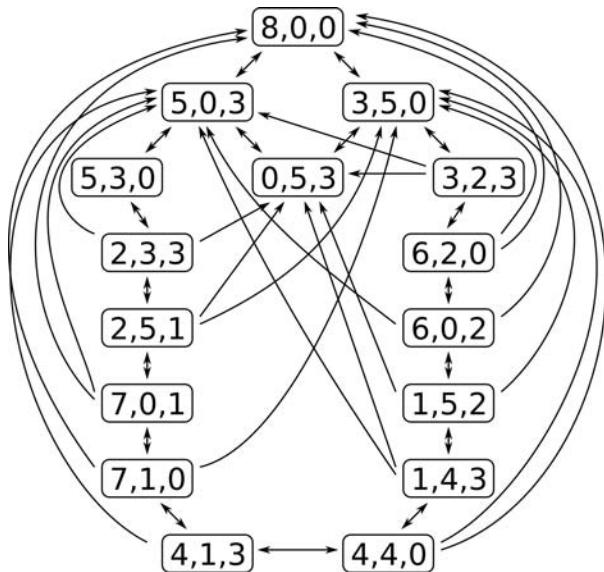
pro druhé uvedené zadání můžeme výstup zapsat jako následující posloupnost stavů nádob: (8, 0, 0), (3, 5, 0), (3, 2, 3), (6, 2, 0), (6, 0, 2), (1, 5, 2), (1, 4, 3), (4, 4, 0).

Zajímavým rozšířením je hledání co nejtěžšího zadání při zadaném omezení na nádoby. Například jaké je nejtěžší zadání, pokud používáme maximálně 3 nádoby s obsahem maximálně 15 litrů? Vyřešit tuto úlohu není obtížné, pokud předpokládáme „obtížnost zadání = minimální počet tahů na vyřešení“. Pak nám však jako těžké zadání vyjde takové, kdy máme nádoby o obsahu 15, 15 a 1 litrů a úkolem je vyrobit 7, 7 a 1 litr. Tato úloha vyžaduje hodně tahů, ale jinak na ní není nic obtížného. Jak můžeme formalizovat obtížnost úlohy lépe?

### Doplňující komentář

Tato úloha je ideální na vysvětlení a procvičení pojmu stavový prostor. Úloha je pro lidi netriviální a přitom stavový prostor je velmi malý, takže ho lze zkonstruovat i ručně – jak ostatně ukazuje obrázek 6.2, který zobrazuje stavový prostor pro jedno ze zadání uvedené v tabulce 6.1. Nejkratší řešení najdeme klasickým prohledáváním do šírky. Na obrázku 6.2 jsou stavy zakresleny do úrovní podle jejich vzdálenosti od počátečního stavu, což odpovídá právě pořadí prohledávání do šírky.

Pro hledání zadání s co nejdelším řešením je dostatečné využití prosté hrubé síly, tedy vyzkoušení všech možností. Pokud se omezíme na „rozumně malá“ zadání do 4 nádob malé velikosti, je všech možných kombinací velikostí nádob dostatečně málo na to, abychom mohli projít všechny. Zadání větší velikosti nemá smysl uvažovat, protože by stejně pro člověka byla příliš náročná.



**Obrázek 6.2:** Přelévání vody – stavový prostor pro úlohu s nádobami o objemu 8, 5 a 3 litry

### 6.3 Hanojské věže

**Nápad:**

3

**Kódování:**

1-4

**Styl úlohy:**

Klasická úloha na procvičení rekurze. Základní úlohu lze řešit velmi krátkým programem, jehož vymyšlení však není triviální.

K této úloze se váže známá legenda. Kdesi v horách nedaleko města Hanoj stojí chrám, ve kterém je velká místořnost a v ní 64 zlatých disků různých velikostí. Mniší přesouvají tyto disky mezi třemi kolíky, přičemž při přesunech dodržují posvátná pravidla. Až se mnichům podaří přemístit všechny disky z prvního kolíku na třetí, nastane konec světa.

Přesná pravidla přesunů jsou následující. Na začátku jsou všechny disky nasazeny na jednom kolíku. Disky mají různé poloměry a jsou seřazeny od největšího po nejmenší. Je povoleno přesouvat disky pouze po jednom, vždy můžeme vzít pouze horní disk z některého kolíku a přemístit jej na jiný kolík. Nikdy nesmíme položit větší disk na menší. Úkolem je přemístit všechny disky na třetí kolík.

Úloha nabízí několik programátorských cvičení (viz též ilustrace na obrázku 6.3):

1. Základní úloha. Vstupem je počet disků, výstupem je textová reprezentace řešení (popis tahů).
2. Nepravidelný počáteční a koncový stav. V základní variantě máme na začátku všechny disky na prvním kolíku. Nyní na začátku disky rozrozmístíme libovolně, ale tak, aby tvořily platnou konfiguraci, tedy aby byl vždy menší disk na větším. Cílem je uspořádat disky do jiné zadane platné konfigurace.
3. Grafické znázornění řešení pomocí série obrázků (může být i textová grafika) nebo přímo pomocí animace.
4. Vykreslení stavového prostoru úlohy. Stavový prostor je tvořen všemi platnými konfiguracemi (rozmístěním disků) a přechody mezi nimi. Obrázek 6.3 zachycuje stavový prostor pro 3 disky. K vykreslení stavového prostoru můžeme využít programy pro vykreslování grafů (např. dot, Pa jek, Gephi) – těmto programům předložíme pouze textový zápis stavů a přechodů mezi nimi a ony automaticky vygenerují grafické znázornění.

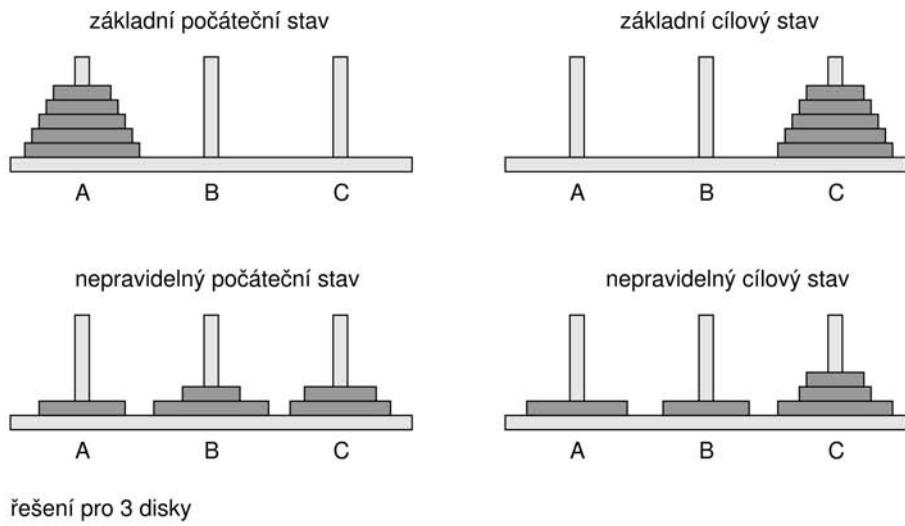
U hledání řešení se ještě můžeme zamyslet, zda náš algoritmus vždy najde řešení na nejmenší počet tahů. Pokud nikoliv, můžeme ještě vymyslet algoritmus, který optimální řešení najde.

### Doplňující komentář

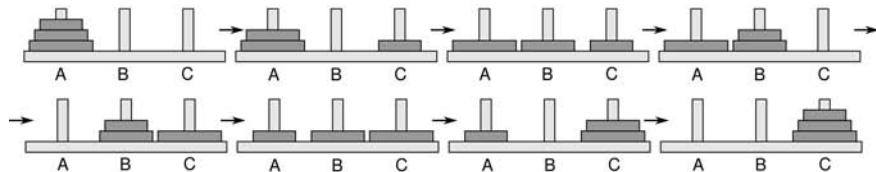
Hanojské věže jsou typickou úlohou na ilustraci rekurze. Rekurzivní myšlenka řešení základní úlohy zní následovně: „Abychom přesunuli  $n$  disků, nejprve rekurzivně přesuneme  $n - 1$  disků ze startovacího na odkládací kolík, pak přesuneme největší disk na cílový kolík a nakonec přesuneme  $n - 1$  disků z odkládacího kolíku na cílový.“

Za využití tohoto rekurzivního přístupu může program vypsat řešení základní úlohy bez toho, aby si musel průběžně pamatovat informaci o stavu úlohy (rozmístění disků). Rozšířenou úlohu s nepravidelným počátečním a koncovým stavem lze také řešit pomocí rekurzivního přístupu, nicméně zde už je nutné se stavovou informací pracovat.

Jak je vidět z obrázku 6.3, stavový prostor Hanojských věží připomíná Sierpińského fraktál (viz úloha 4.4). Tato podobnost je ještě výraznější, pokud si stavový prostor vykreslíme pro více disků, a není náhodná – Hanojské věže i Sierpińského fraktál lze definovat rekurzivně a spojitost mezi oběma problémy lze definovat i exaktně matematicky.



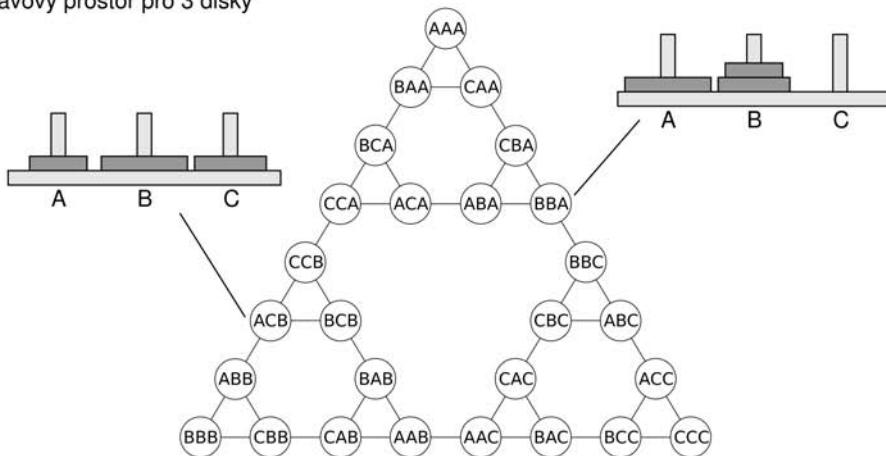
řešení pro 3 disky



textový zápis řešení pro 3 disky

A  $\rightarrow$  C; A  $\rightarrow$  B; C  $\rightarrow$  B; A  $\rightarrow$  C; B  $\rightarrow$  A; B  $\rightarrow$  C; A  $\rightarrow$  C

stavový prostor pro 3 disky



Obrázek 6.3: Hanojské věže

## 6.4 Pokrývání mřížky

**Nápad:** 3-4

**Kódování:** 2-3

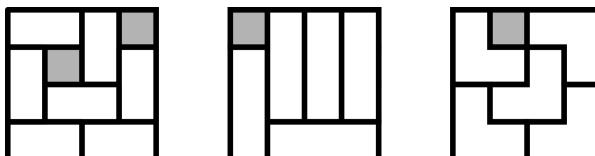
**Styl úlohy:** *Tyto úlohy jsou především o nápadu, kódování není příliš náročné. Jedna z podúloh představuje klasickou ilustraci rekurze.*

Tato úloha se skládá ze tří podúloh stejného typu. Máme čtvercovou mřížku, ze které jsou odstraněna některá políčka, a máme za úkol ji pokrýt pomocí sady kostek, přičemž všechny kostky mají stejný tvar. Ve cvičení 6.9 se pak podíváme na podobnou úlohu, kdy budeme pokrývat mřížku pomocí různých tvarů.

1. Z mřížky  $8 \times 8$  odstraníme dvě pole (v nejčastěji používané verzi zadání jsou odstraněny dva protější rohy). Úkolem je pokrýt zbývajících 62 polí pomocí 31 kostek tvaru  $2 \times 1$ . Podle toho, která dvě pole odebereme, řešení může nebo nemusí existovat. Napište program, který pro zadaná dvě pole rozhodne, zda řešení existuje, a pokud ano, najde jej a vypíše.
2. Podobně jako předchozí, ale tentokrát odstraníme pouze pole a pokrýváme kostkami tvaru  $3 \times 1$ .
3. Z mřížky velikosti  $2^n \times 2^n$  odstraníme jedno pole. Úkolem je pokrýt mřížku pomocí déláků ze tří kostek ve tvaru písmene L. Tentokrát má úloha řešení vždy.

Úlohy jsou ilustrovány na obrázku 6.4, kde jsou ukázána konkrétní řešení pro mřížku  $4 \times 4$ . Výstupy programů stačí textové, například jako tabulka čísel, kde čísla odpovídají jednotlivým kostičkám.

Úkoly lze dále zkomplikovat tím, že hledáme pokrytí kostičkami, které jsou obarveny barvami, přičemž žádné dvě kostičky sousedící hranou nesmí mít stejnou barvu. Cílem je použít co nejméně barev.



**Obrázek 6.4:** Ukázka možných řešení jednotlivých úloh „Pokrývání mřížky“ na mřížce  $4 \times 4$

### Doplňující komentář

Zde prozradíme základní myšlenku řešení, podrobnější popis včetně ilustrací je uveden v řešení. Úloha s kostkami  $2 \times 1$  je řešitelná jen někdy, pro rozhodnutí řešitelnosti se hodí představit si mřížku  $8 \times 8$  jako šachovnici, tj. obarvenou na stídačku 2 barvami. Úkol pro kostky  $3 \times 1$  se řeší podobně, jen obarvujeme 3 barvami. Úlohu s kostkami tvaru L lze řešit rekurzivně – rozdělíme si mřížku na 4 menší mřížky, umístíme vhodně jednu kostku a dostaneme analogické menší podproblémy, které vyřešíme rekurzivně.

## 6.5 Hledání cest v bludišti

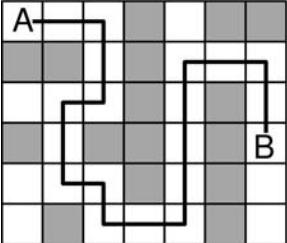
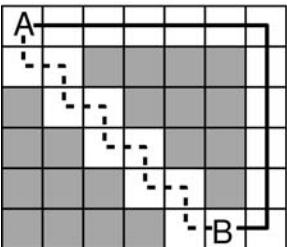
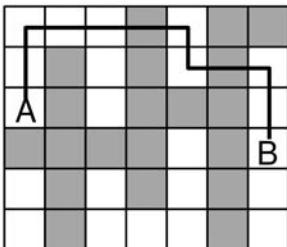
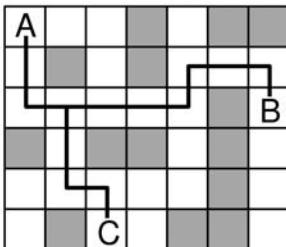
**Nápad:** 2-4

**Kódování:** 3

**Styl úlohy:** *Vhodné cvičení na hledání nejkratších cest pomocí prohledávání grafu do šířky.*

V tomto cvičení máme vždy na vstupu bludiště s vyznačenými body a úkolem je najít a vypsat nejkratší cestu mezi zadánymi body, přičemž mohou být zadány nějaké další podmínky. Jako vstup a výstup programu lze použít jednoduchý textový formát ilustrovaný na obrázku 6.5: dvojrozměrná mřížka, každé pole je buď volné nebo obsazené zdí, startovní a cílová pole jsou označena písmeny A a B (resp. C u posledního příkladu), nalezená cesta je v řešení znázorněna písmenem x. Úloha má několik variací (viz ilustrace na obrázku 6.5):

1. Základní hledání cesty. Povolený tah je přesun na sousední volné pole, cílem je najít nejkratší cestu.
2. Robot. Máme najít nejrychlejší cestu pro robota, jehož základní operace jsou posun o 1 pole dopředu a otočení o 90 stupňů vlevo nebo vpravo, přičemž všechny operace trvají stejně dlouho. V příkladu na obrázku má cesta znázorněná plnou čárou 12 kroků + 3 otočení a čárkovaná čára 10 kroků + 11 otočení. Čárkovaná čára je sice kratší na počet kroků, ale vyžaduje delší čas kvůli mnoha otočením.
3. S dynamitem. Máme k dispozici dynamit, kterým můžeme odstřelovat zdi. Úkolem je dostat se ze startu do cíle za použití co nejméně dynamitu (primární kritérium) a co nejméně kroků (sekundární kritérium).
4. Spojnice 3 bodů. V bludišti máme vyznačeny 3 body a máme je spojit pomocí co nejkratšího kusu drátu, drát se může větvit.

Základní zadání	Textový zápis zadání	Textový zápis řešení
	A..#.## ##.#... . .#.#. . #.##.#B . .#.#. . . #....#.	Axx#.## ##x#xxx .xx#x#x #x##x#B .xx#x#. . . #xxx#. .
Robot	S dynamitem	Tři body
		

Obrázek 6.5: Hledání cest v bludišti – ukázky problémů

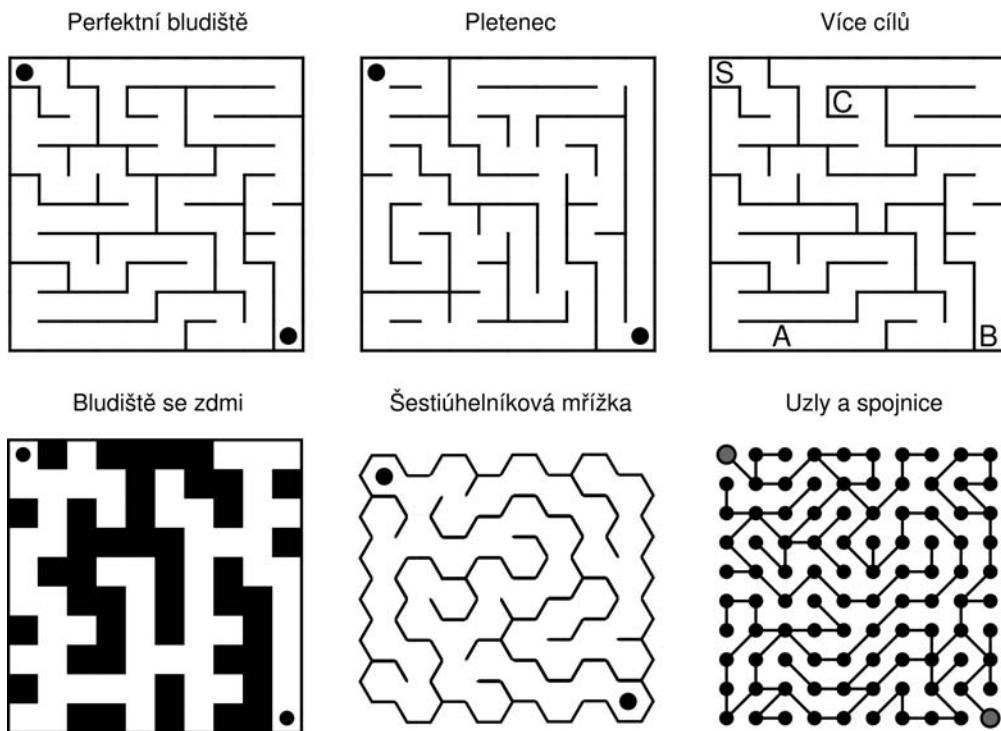
### Doplňující komentář

Cvičení je vhodné na procvičení základních grafových pojmů a konkrétně prohledávání grafu do šírky. Základ úspěšného řešení spočívá ve správném převedení problému do grafového vyjádření. Pak již stačí aplikovat klasické prohledávání do šírky, v případě úlohy s dynamitem může být vhodné použít hledání cest v ohodnoceném grafu.

## 6.6 Generování bludišť

<b>Nápad:</b>	3-4
<b>Kódování:</b>	3-5
<b>Styl úlohy:</b>	Netradiční cvičení s otevřeným zadáním, ve kterém se (na první pohled nečekaně) využijí některé klasické grafové algoritmy.

Cílem je vytvořit program, který bude generovat zajímavá a pěkná bludiště. Na rozdíl od většiny ostatních cvičení je v tomto případě zadání mírně otevřené, protože není přesně specifikováno, co znamená „zajímavé a pěkné bludiště“.



Obrázek 6.6: Ukázky různých typů bludišť

Můžeme se též zamyslet nad tím, že bludiště má být náročné pro člověka, a snažit se generovat bludiště, která budou pro člověka co nejnáročnější.

Program dostane na vstup požadovanou velikost bludiště a případně další parametry a vygeneruje obrázek bludiště. Generování musí být nějakým způsobem náhodnostní, tj. při každém běhu programu nad stejnými vstupy dostaneme jiné bludiště. Pro výstup je vhodné použít vektorovou grafiku, konkrétně formát SVG.

Bludiště můžeme charakterizovat podle různých kritérií, přičemž pro každé z nich máme několik variant. Jednotlivé varianty lze volně kombinovat, takže dohromady dostáváme velké množství konkrétních bludišť'. Základní kritéria a varianty jsou následující (viz ukázky na obrázku 6.6):

1. Podkladová struktura bludiště:

- Přepážky v čtvercové mřížce.
- Zdi v čtvercové mřížce.
- Uzly a spojnice.
- Jiné podkladové mřížky, např. trojúhelníková, šestiúhelníková.

## 2. Struktura cest v bludišti:

- Perfektní bludiště – mezi každými dvěma body vede právě jedna cesta, tj. bludiště neobsahuje smyčky.
- Bludiště typu „pletenec“ – bludiště bez slepých uliček, tj. každé pole leží na smyčce.
- Kombinovaná struktura obsahující slepé uličky i smyčky.

## 3. Počet cílů:

- Jeden start a jeden cíl, úkolem je najít cestu.
- Jeden start a několik cílů, přičemž právě jeden cíl je ze startu dosažitelný a úkolem řešitele je určit, který z nich to je.

Základní verze zadání počítá s tím, že se rozhodneme pro jednu konkrétní kombinaci uvedených kritérií, např. perfektní bludiště v čtvercové mřížce s přepážkami a jedním startem a cílem. Zajímavé rozšíření je udělat obecnější program pro generování bludišť, který jako součást vstupu dostane specifikaci jednotlivých kritérií. Toto rozšíření už je zajímavé, nejen co se týče algoritmů, ale i z pohledu softwarového návrhu a reprezentace dat.

## Doplňující komentář

Zadání je záměrně otevřené, nebudeme tedy poskytovat příliš mnoho návodních komentářů. Pouze prozradíme, že generování základních bludišť lze udělat velmi snadno s využitím některých klasických grafových algoritmů. V řešení jsou uvedeny konkrétní algoritmy a ukázky výsledných bludišť.

Pro orientaci nabízíme také několik poznámek k programátorské obtížnosti jednotlivých variant:

- S čtvercovou mřížkou se pracuje snáz než s jinými mřížkami.
- Bludiště s přepážkami se dělá snáz než bludiště se zdmi.
- Generování perfektních bludišť je výrazně snadnější než generování bludišť typu pletenec.
- Dobré bludiště s více cíli je těžší než bludiště s jedním startem a jedním cílem.

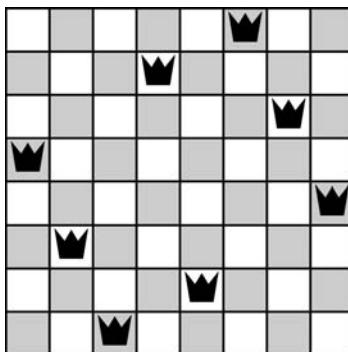
## 6.7 Rozmístování figur na šachovnici

**Nápad:** 2-4

**Kódování:** 2-4

**Styl úlohy:** *Klasická logická úloha a její variace – program je většinou krátký, potřebujeme pouze správný nápad.*

V tomto příkladu se budeme zabývat rozmístováním figur na šachovnici tak, aby se vzájemně neohrožovaly. Základní úloha tohoto typu je „problém 8



**Obrázek 6.7:** Problém 8 dam – ukázka možného řešení

dam". Cílem je umístit 8 dam na šachovnici tak, aby se vzájemně neohrožovaly, přičemž dama ohrožuje políčka ve stejném sloupci, řádku a diagonále. Příklad řešení úlohy je na obrázku 6.7. Tuto základní úlohu si můžeme zkousit vyřešit ručně například na plánu  $6 \times 6$  a potom zkousit implementovat několik zobecnění a variací.

Přimočaré zobecnění je „problém  $n$  dam“, tj. cílem je umístit  $n$  dam na mřížku  $n \times n$  tak, aby se vzájemně neohrožovaly. Kolik existuje řešení pro velikost plánu  $n$ ? Kolik existuje různých řešení pro velikost plánu  $n$ ? Dvě řešení považuje za stejná, pokud jsou symetrická, tj. jdou na sebe převést pomocí rotací a překlopení. Zkuste úlohu vyřešit pro co největší  $n$ .

Kromě dam můžeme umisťovat i další šachové figurky: věž, střelec, král, kůň. Kolik nejvíce figurek od každého typu můžeme umístit na standardní šachovnici  $8 \times 8$ ? Kolik na šachovnici  $n \times n$ ? Napište program, který vypíše optimální rozmístění pro zadané  $n$ . Opět můžeme hledat (a počítat) také všechna řešení.

Kromě klasických šachových figur můžeme uvažovat též různé alternativní figury, jako je kentaur (kombinace koně a střelce) nebo  $(x, y)$ -skokan (skáče o  $x$  polí v jednom směru a  $y$  polí v druhém směru; klasický kůň je  $(1, 2)$ -skokan). Dále můžeme kombinovat různé figury, konkrétně například současně umístit  $m$  dam a  $m$  koňů tak, aby se vzájemně neohrožovaly.

Doposud jsme uvažovali pouze problémy, kde figurky umisťujeme na prázdný plán. Můžeme však také zvážit variantu, kdy na plánu již nějaké figurky rozmištěné jsou. Konkrétní zajímavý problém dostaneme pro koně na „úzkém“ plánu. Je dán plán velikosti  $n \times m$  a na něm rozmištěno  $k$  koňů, a to tak, že se vzájemně neohrožují. Kolik nejvíce dalších koňů můžeme na plán

umístit při zachování vzájemného neohrožování? Předpokládejme, že plán je dlouhý a úzký, tj.  $m < 6$ , ale  $n$  může být velké.

### Doplňující komentář

„Problém  $n$  dam“ je klasická úloha na procvičení metody backtracking. Jde patrně o jeden z nejjednodušších příkladů, kde lze metodu backtracking smysluplně aplikovat. Postupně přiřazujeme dámym na jednotlivá polička, jakmile nemáme kam další umístit, tak se vracíme a zkoušíme jiné přiřazení – viz příklad ilustrace běhu na obrázku 6.8.

Výsledné řešení lze zapsat krátkým a elegantním programem, k tomu je však potřeba několik dílčích nápadů:

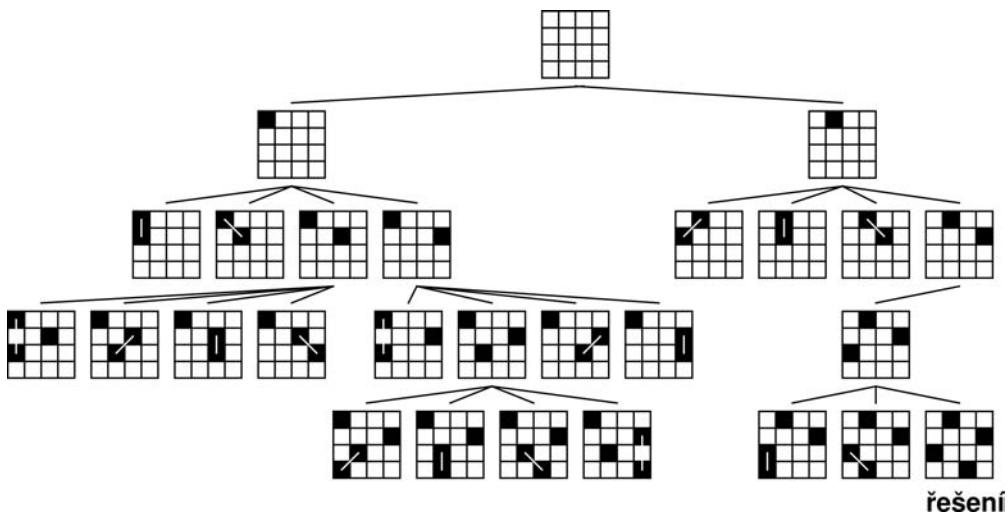
- Je zřejmé, že v řešení musí být v každém řádku právě 1 dáma. Rozestavení dam (i částečné) tedy můžeme reprezentovat vektorem  $(s_1, \dots, s_n)$ , kde  $s_i$  udává sloupec, ve kterém je dáma na  $i$ -té řádku. Nepotřebujeme tedy explicitně reprezentovat celou dvojrozměrnou šachovnici.
- Test, zda se ohrožují dvě dámym na pozicích  $(x_1, y_1)$  a  $(x_2, y_2)$ , můžeme provést jednoduše pomocí podmínky:  $x_1 = x_2 \vee y_1 = y_2 \vee x_1 - y_1 = x_2 - y_2 \vee x_1 + y_1 = x_2 + y_2$  (rozmyslete si proč).

Na úloze lze ilustrovat také vliv pořadí prohledávání při backtrackingu – když kandidáty na umístění další dámym prohledáváme v náhodném pořadí, je výsledné prohledávání téměř vždy rychlejší, než když používáme pravidelné pořadí. Zkuste experimentálně ověřit. Samozřejmě to platí jen při hledání jednoho řešení. Při hledání všech řešení již na pořadí prohledávání nezáleží.

Pro možnost kontroly uvádíme počty řešení „Problému  $n$  dam“ pro  $n$  od 1 do 8. Zkuste tabulku doplnit pro co největší  $n$ :

$n$	1	2	3	4	5	6	7	8
Různá řešení	1	0	0	1	2	1	6	12
Všechna řešení	1	0	0	2	10	4	40	92

Pro ostatní figury kromě dámym lze úlohu vyřešit jednoduše bez prohledávání, protože optimální konfigurace lze dosáhnout pomocí velmi pravidelného rozmístění figur, a to i na plánu obecné velikosti. Pro kontrolu uvádíme maximální počty neohrožujících se figur na klasické šachovnici: 8 věží, 14 střelců, 16 králů, 32 koňů. Pokud bychom však chtěli nejen jedno řešení, ale všechna, může nám opět přijít vhod prohledávání pomocí techniky backtracking. Pro různé varianty problému je zajímavé zamyslet se, kdy existuje přímočaré řešení a kdy už se neobejdeme bez prohledávání. Varianta „koně na úzkém plánu“ je zajímavá v tom, že nevyžaduje prohledávání pomocí backtrackingu, ale existuje efektivnější řešení pomocí dynamického programování.



Obrázek 6.8: Problém 4 dam – ilustrace prohledávání při použití metody backtracking

## 6.8 Jak navštívit všechna pole mřížky?

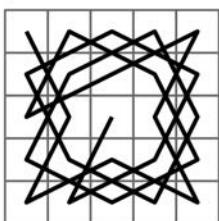
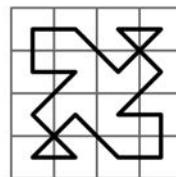
**Nápad:** 3-5

**Kódování:** 3-4

**Styl úlohy:** Atraktivní cvičení na procvičení backtrackingu a heuristik. Pro malou velikost zadání vystačíme se základním backtrackingem, pro větší velikosti se úloha stává náročnou a musíme vymýšlet netriviální heuristiky.

Toto zadání sdružuje několik úloh, které na první pohled vypadají různorodě (viz obrázek 6.9). Základní princip je však vždy stejný: máme mřížku a úkolem je najít v ní cestu, která prochází přes všechna pole a přitom dodrží zadanou podmínu.

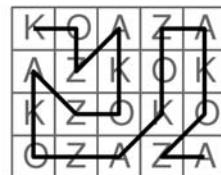
1. Koňská procházka: Úkolem je najít posloupnost tahů šachového koně (skok tvaru písmene L).
2. Královská procházka: Úkolem je najít posloupnost tahů šachového krále, tj. povolený tah je přesun na sousední pole. Aby to bylo zajímavější, hledáme procházky, které jsou uzavřené (na konci se vrátí na výchozí bod) a symetrické (osově nebo středově).
3. Mřížka s opakujícími se slovy – úkolem je najít královskou procházku z levého horního do pravého dolního rohu tak, aby se na ní opakovalo zadané slovo.

**koňská procházka****symetrická uzavřená královská procházka****opakování slova**

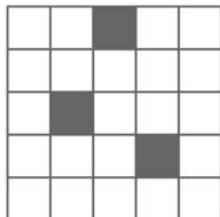
zadání:

K	O	A	Z	A
A	Z	K	O	K
K	Z	O	K	O
O	Z	A	Z	A

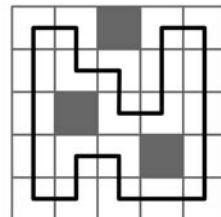
řešení:

**smyčka v bludišti**

zadání:



řešení:

**Obrázek 6.9:** Hledání cest přes všechna pole mřížky

4. Smyčka v bludišti – tentokrát se pohybujeme pouze o 1 pole vodorovně či svisle, některá pole jsou blokovaná a máme najít uzavřenou cestu.

Vstupem programu je vždy velikost mřížky a případně její popis, výstupem je pak posloupnost tahů. Výstup můžeme ztvárnit jednoduše textově jako výpis tabulky čísel udávajících pořadí návštěvy. Elegantnější (a nepříliš náročné) je zakreslit výstup do obrázku (tak jako na obrázku 6.9).

Ve všech případech můžeme rozlišit, zda hledáme jedno řešení, nebo všechna řešení. Především u královské procházky je vhodné se zaměřit na hledání všech řešení, protože najít jedno řešení je triviální. U variant „mřížka s opakujícími se slovy“ a „smyčka v bludišti“ můžeme kromě řešení zvážit také problém generování zadání, přičemž cílem je generovat zadání s jednoznačným řešením.

## Doplňující komentář

Podobně jako u předchozího cvičení jde opět o úlohu na procvičení techniky backtracking. Úlohy jsou jednoduché na reprezentaci, takže napsat základní algoritmus je docela jednoduché. Základní princip je následující: vyjdeme z jednoho pole (například z rohu), postupně prodlužujeme cestu, pokud nejde prodloužit, vracíme se a zkoušíme jinou možnost. Aktuální stav reprezentujeme pomocí dvojrozměrného pole čísel, které udává pořadí procházení (0 znamená, že pole ještě nebylo navštíveno).

Základní algoritmus zvládne vyřešit jen malá zadání. Aby program fungoval efektivně i pro větší zadání, musíme vymýšlet vhodné heuristiky. U backtrackingu máme dva základní typy heuristik:

1. Preferování některých možností u prohledávání. Například u koňské procházky je dobrá heuristika preferovat pole, která jsou co nejvíce omezená. Tento typ heuristiky nám pomůže při hledání jednoho řešení, nikoliv při hledání všech řešení.
2. Průběžná kontrola částečných řešení. Například kontrolujeme, zda existuje pole, které je odříznuté a již nepůjde navštívit. Pokud takové pole existuje, nemá cenu z aktuálního stavu dále prohledávat a můžeme se vrátit zpět. Tato heuristika je užitečná, i když hledáme všechna řešení.

## 6.9 Polyomina

**Nápad:**

3-4

**Kódování:**

4-5

**Styl úlohy:**

*Úloha na procvičení reprezentace dat a systematického prohledávání s heuristikami.*

Polyomina jsou kostky složené z několika čtvercových dílků. Podle počtu dílků je nazýváme domina, triomina, tetromin, pentomina, hexomina, heptomina a tak dále. Kostky složené z méně jak 6 dílků jsou ilustrovány na obrázku 6.10. Polyomina (především pentomina) jsou oblíbeným tématem úloh z rekreační matematiky. Zde uvážíme tři úlohy související s polyominy.

1. Generování polyomin. Program načte  $n$  a vygeneruje všechna polyomina složená z  $n$  dílků. Z polyomin, která se liší jen symetrií, vypíše vždy jen jeden, tj. například pro  $n = 5$  program vypíše na výstup 12 pentomin uvedených na obrázku 6.10.

2. Úloha „Skládačka“. Úkolem je pomocí zadaných dílků vyskládat zadaný obrazec. Program na vstupu načte dostupné délky a cílový obrazec, výstupem programu je výpis řešení, případně výpis všech řešení. Na obrázku jsou uvedena konkrétní zadání pro tetromina. Typickým zadáním je z pentomin vyskládat obdélníky rozměrů  $3 \times 20$ ,  $4 \times 15$ ,  $5 \times 12$  a  $6 \times 10$ ; obrázek uvádí řešení pro obdélník  $6 \times 10$ .
3. Úloha „Rozdělovačka“. Úkolem je zadaný obrazec rozdělit na  $k$  stejných polyomin. Program načte obrazec a vypíše rozdělení na stejná polyomina (případně všechna řešení).
4. Řešení uvedených úloh v trojrozměrném prostoru. Konkrétní známý příklad trojrozměrné skládačky jsou „Soma kostky“.

### Doplňující komentář

Po algoritmické stránce tyto úlohy nejsou náročné – jde o použití hrubé síly, algoritmu backtracking a případně heuristik. Pro úspěšné řešení je důležité si především dobře rozmyslet reprezentaci dat. Při dobrém přístupu nemusí být výsledný kód příliš dlouhý, při nevhodné reprezentaci se ale člověk v úloze snadno zamotá. Základní rady k jednotlivým úlohám:

- Polyomina generujeme „odspodu“. Jakmile máme všechna polyomina o  $n$  čtvercích, zkoušíme k nim přidat 1 čtverec na všechny pozice, čímž generujeme polyomina o  $n + 1$  čtvercích. Přitom kontrolujeme výskyt duplicit (včetně symetrií).
- Úloha Skládačka se řeší pomocí algoritmu backtracking. Je více možností, jak backtracking provádět. Pro tetromina není zvolený přístup zásadní, protože možností je málo, takže libovolný trochu rozumný přístup řešení najde. Pro hledání vyskládání obdélníku z pentomin už je potřeba zvolit vhodný přístup a použít heuristiky.
- Pokud máme k dispozici řešení předchozích dvou úloh, můžeme úlohu Rozdělovačka řešit prostě hrubou silou. Vygenerujeme si všechna polyomina příslušné velikosti a pro každé z nich zkusíme zadaný obrazec vyskládat.

**Dílky**

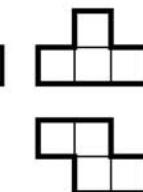
Domino



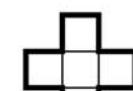
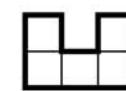
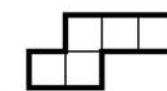
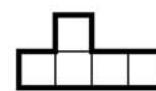
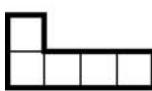
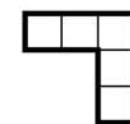
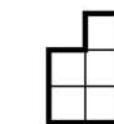
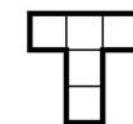
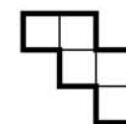
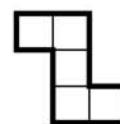
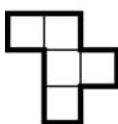
Triomina



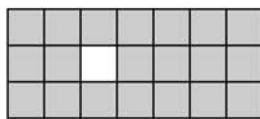
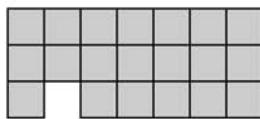
Tetromina



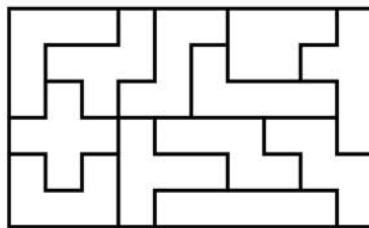
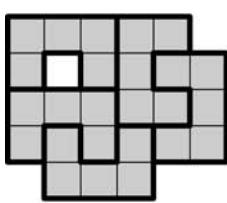
Pentomina

**Skládačka**

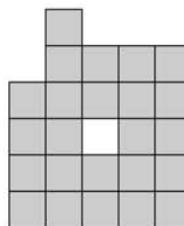
Zadání pro tetrominu



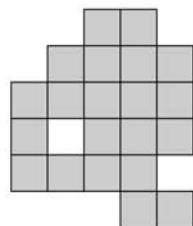
Řešení: obdélník 6 × 10, pentomina

**Rozdělovačka**

5 pentomin



4 hexomina



3 heptomina

**Obrázek 6.10:** Polyomina

## 6.10 Sudoku

**Nápad:** 3-4

**Kódování:** 3-5

**Styl úlohy:** Řešení klasické logické úlohy, náročnost cvičení lze volně škálovat podle toho, jak obecné zadání si zvolíme a jak velké máme ambice na rychlosť výsledného programu.

Sudoku je velmi známá úloha, a tak rovnou formulujeme pravidla pro obecné zadání. Máme mřížku  $n \times n$  rozdělenou na  $n$  bloků o  $n$  polích. Úkolem je umístit do mřížky čísla od 1 do  $n$  tak, aby v každém řádku, sloupcu a bloku bylo každé číslo právě jednou. Klasické Sudoku odpovídá této obecné formulaci pro  $n = 9$  a čtvercové bloky tvaru  $3 \times 3$ . Můžeme však uvažovat i jiné velikosti mřížek a obdélníkové bloky (viz obrázek 6.11), případně i různé exotičtější tvary bloků a dodatečné podmínky (sudost, nerovnosti).

Základní programátorský úkol spočívá v napsání programu, který zadané Sudoku vyřeší, a to co nejrychleji. Jako formát vstupu se standardně používá řetězec 81 znaků, který zapisuje zadání po řádcích, pro označení prázdného pole se používá tečka (případně nula). V příloze na webu lze najít seznam zadání pro testování.

Konkrétnější rozpis možných zadání odstupňovaný podle obtížnosti:

1. načtení zadání a zkontrolování, zda zadání neporušuje pravidla Sudoku,
2. řešení alespoň jednoduchých Sudoku,
3. řešení libovolného Sudoku standardní velikosti,
4. optimalizované řešení, které zvládne vyřešit 1 000 Sudoku za 1 vteřinu,
5. experimentální porovnání různých algoritmů pro řešení Sudoku,

1			
			1
4		3	
3			

3				
	6		2	4
	1			
6	5	2		6
	3		1	

	1	5		2	3	7
6				9		4
5		8				1
		7	1		6	
	6					7
	7		2	3		
	8			7		6
6		4				2
1	9	2	8	4		

Obrázek 6.11: Sudoku: příklady zadání různých velikostí

6. algoritický odhad obtížnosti zadání pro člověka,
7. generování zadání Sudoku zadané obtížnosti,
8. řešení úlohy i pro nestandardní velikosti herního plánu ( $6 \times 6$ ,  $16 \times 16$ ) a pro různé variace základního zadání, například Sudoku s nepravidelnými bloky nebo s vyznačenými sudými čísly, sousedy či nerovnostmi.

Komu připadá Sudoku příliš známé a mnohokrát vyřešené, může zkoušet některé další úlohy podobného typu, např. Nurikabe, Kakuro, Slitherlink. Většina dále uvedených komentářů je aplikovatelná i na tyto úlohy.

### Doplňující komentář

Sudoku je příkladem „problému splnění podmínek“. Pro tento typ problémů existují speciální nástroje na řešení, případně je lze elegantně zapsat pomocí logického programování (např. v jazyce Prolog). Zde však zůstaneme u klasického imperativního programování.

Sudoku lze řešit dvěma základními přístupy. První přístup je „propagace podmínek“ a odpovídá zhruba tomu, jak řeší Sudoku člověk. Pro každé políčko spočítáme množinu kandidátů, kterí přicházejí pro dané pole do úvahy. Na základě těchto kandidátů pak můžeme hodnoty pro některá pole odvodit. Tento přístup je ilustrován na obrázku 6.12, kde jsou pro každé pole uvedeny všechny kandidáti a je označena „vynucené hodnota“ (kolečko) a skryté hodnoty (čtverce).

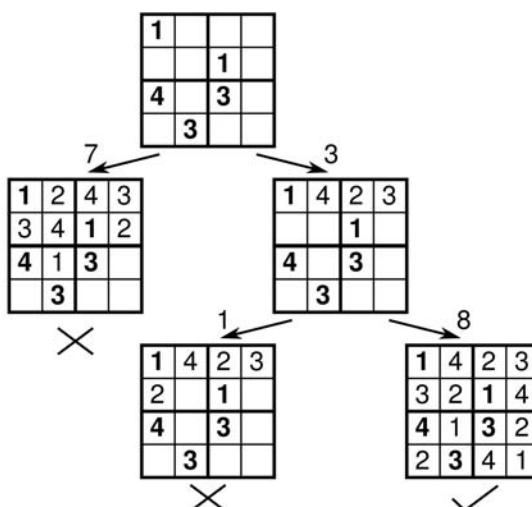
Jakmile doplníme nové číslo, přepočítáme množiny kandidátů a hledáme, zda neumíme doplnit něco dalšího. Takto postupujeme tak dlouho, dokud nevyřešíme úlohu kompletně nebo se nezasekneme. I za použití jednoduchých pravidel pro odvozování vynucených hodnot je tento postup dostatečný na vyřešení většiny Sudoku úloh, které se objevují v novinách s označením „lehké“. Při použití složitějších pravidel propagace podmínek je možné vyřešit téměř všechny úlohy, nicméně ani tak při tomto přístupu nemáme zaručeno, že řešení najdeme.

Druhý přístup spočívá ve využití „hrubé síly“. Naivním použitím hrubé síly by bylo vygenerovat všechna možná přiřazení čísel do volných polí a pak vyhodnocovat, zda splňují podmínky. Tento přístup by však neprošel ani pro jednoduché Sudoku. Použijeme proto algoritmus backtracking. Postupně přiřazujeme hodnoty 1 až 9 do volných polí a průběžně kontrolujeme, zda je přiřazení stále konzistentní. Jakmile narazíme na spor se zadáním úlohy, vracíme se zpět a zkoušíme jiné přiřazení. Postup je opět ilustrován na obrázku 6.12. Číslo nad šípkou značí počet provedených kroků, u kterých nebylo možné žádné větvení. Křížek znamená, že v daném stavu nelze na další pozici doplnit žádné číslo, aniž by došlo k porušení pravidel Sudoku. Pomocí algoritmu

Množiny kandidátů pro jednotlivá pole

<b>1</b>	2,4	2,4	2,3, 4
2 <b>3</b>	2,4	<b>1</b>	2,3, 4
<b>4</b>	1,2	<b>3</b>	1,2
②	<b>3</b>	2,4	1,2, 4

Backtracking



Obrázek 6.12: Ilustrace základních metod řešení Sudoku

backtracking už vyřešíme libovolné Sudoku. Pro Sudoku standardní velikosti je i přímočará, neoptimalizovaná realizace tohoto algoritmu docela rychlá – na současných strojích většinou pod vteřinu.

Oba dva přístupy můžeme zkombinovat. Dokud je to možné, provádíme propagaci podmínek. Jakmile pomocí propagace podmínek nemůžeme odvodit žádnou další hodnotu, začneme systematicky zkoušet, přičemž v rámci systematického zkoušení opět používáme propagaci podmínek, abychom rychleji našli případný spor. Tímto způsobem již je možné vyřešit zmíněných 1 000 Sudoků za 1 vteřinu.

Výpočet obtížnosti úlohy pro člověka je už náročnější téma, protože není jasné, co onu obtížnost pro člověka ovlivňuje. Intuice v tomto případě není úplně dobrým vodítkem – na první pohled se zdá, že obtížnost úlohy přímočáře souvisí s počtem vyplněných políček v zadání. To však zdaleka neplatí. K tomu, aby bylo možné toto téma zpracovat důkladně, je nezbytné získat data o tom, jak dlouho trvá lidem řešit jednotlivá Sudoku, a s pomocí dat potom předpovědi obtížnosti vyhodnotit. Taková data lze s trochou invence získat z některých webových portálů pro hraní Sudoku, kterých je díky současné popularitě Sudoku celá řada. Alternativně lze pro základní vyhodnocení použít úlohy z novin, kde jsou úlohy typicky ohodnoceny podle obtížnosti, to však není příliš vhodný přístup, protože novinová ohodnocení jsou často docela nepřesná.

## 6.11 Sokoban

**Nápad:** 4-5

**Kódování:** 4-5

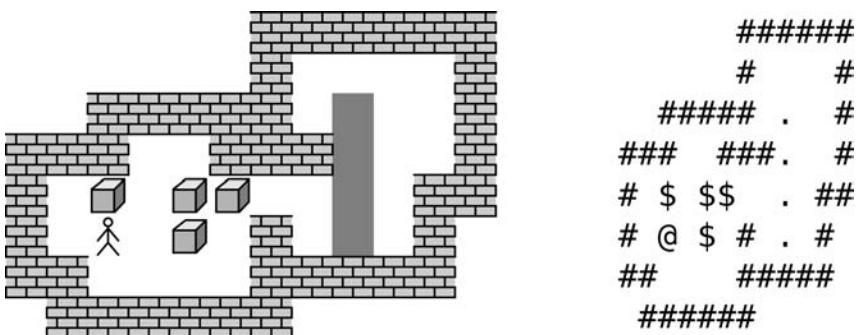
**Styl úlohy:** Náročná logická úloha, jejíž řešení vyžaduje použití heuristik. Problém je vhodný jako „výzva“ s cílem napsat program, který zvládne vyřešit co nejsložitější zadání.

Hrací plán úlohy Sokoban tvoří bludiště, ve kterém jsou bedny a panáček. Úkolem je dostat bedny na vyznačená cílová pole. Pohybujeme panáčkem, který může bedny pouze tlačit, navíc může tlačit vždy maximálně jednu bednu. Na obrázku 6.13 je konkrétní zadání. Na Internetu lze snadno nalézt stovky konkrétních zadání a stránky, na kterých je možné si hru vyzkoušet.

Cílem je napsat program, který bude úlohu řešit. Jako vstup dostane popis problému ve standardním formátu, který je ilustrován na obrázku 6.13. Na výstup program zapíše řešení ve vlastním zvoleném formátu; důležité pouze je, aby byl zápis snadno interpretovatelný člověkem, a bylo tudíž možné jej zkontolovat. Cílem je program vyladit tak, aby zvládl vyřešit problémy s co největším počtem bedýnek. V příloze na webu jsou uvedena zadání různé velikosti, která je možné použít pro otestování programu, případně jako podklad pro srovnání několika soutěžících programů.

### Doplňující komentář

Základní přístup k řešení úlohy není složitý. Jde o hledání cesty ve stavovém prostoru, tedy v principu stejný algoritmus jako třeba pro řešení úlohy Přelévání vody. Komplikace spočívá v tom, že tentokrát je stavový prostor výrazně větší. Přímočaré prohledávání všech možností prohledáváním do šířky



Obrázek 6.13: Sokoban zadání a jeho textová reprezentace

zvládne vyřešit jen velmi jednoduché zadání Sokobanu. Pro větší zadání potřebujeme použít heuristické prohledávání – takzvaný A\* algoritmus. Místo fronty zde používáme prioritní frontu, v níž máme stavy seřazeny podle jejich heuristického ohodnocení, které udává odhad, jak je daný stav „blízko“ cílovému stavu. Dále je vhodné využít ořezávání neperspektivních stavů, tedy stavů, o kterých můžeme rozhodnout, že nemá cenu z nich dále prohledávat. Takovým stavem je například stav, ve kterém je bedna v rohu, přičemž nejde o cílové pole. Právě vymýšlení vhodných heuristik je na této úloze nejzajímavější.

Nicméně i základní prohledávání bez heuristik není úplně triviální. Jako první krok si potřebujeme ujasnit, jak budeme hrací plán v programu reprezentovat. S touto reprezentací pak potřebujeme umět provádět následující základní operace: načtení počátečního stavu (zadání), operace následníka (jak se mohu z daného stavu pohnout), test cílového stavu (jestli už je úloha vyřešena), ukládání stavů a test, zda už byl stav navštíven. Stav můžeme reprezentovat více způsoby. V každém případě je užitečné zvlášť reprezentovat globálně konstantní část stavu (polohu zdí) a zvlášť proměnlivou část stavu (polohu beden a panáčka).

Pro představu uvádíme základní odhad, jaká zadání je možné algoritmicky vyřešit. Pomocí základního prohledávání bez použití heuristik je možné vyřešit většinu zadání se 4 bednami. Pomocí relativně jednoduchých heuristik lze vyřešit zadání asi s 8–10 bednami. Sofistikovanými heuristikami pak lze zvládnout i úlohy s větším počtem beden, ale ani nejlepší algoritmy nezvládnu vyřešit všechny úlohy, které umí vyřešit lidé.

# 7 Hry

*Počítač mě jednou porazil v šachách, ale v kick-boxu neměl šanci.*

E. Philips

Podobně jako logické úlohy jsou i hry pro většinu lidí atraktivní a baví je. Jako programátorské cvičení mají navíc hry jednu velkou výhodu. Jakmile program dokončíme, můžeme proti němu hrát, a toto hraní vede k přirozenému testování programu. U mnoha jiných problémů je testování daleko méně zábavné, což vede k tomu, že mnoho programátorů testování odflákne – vyzkouší program na třech vstupech a usoudí, většinou mylně, že funguje.

Hry mají dva výrazné prvky: interakce s uživatelem a hledání dobrých strategií pro počítač. Podle toho, na co se zaměříme, můžeme pro stejné zadání dostat výrazně různý styl úloh. Pokud se zaměříme primárně na interakci s uživatelem, jsou úlohy především cvičením ze softwarového návrhu. Většinu pozornosti v takovém případě věnujeme návrhu uživatelského rozhraní a robustnosti programu, tj. například aby se program nezhroustil, když uživatel zadá chybný tah. Pokud se zaměříme na hledání dobrých strategií pro počítač, stane se z úloh zajímavé cvičení na návrh algoritmů. V tomto případě je užitečné udělat jen jednoduché textové rozhraní pro hru a soustředit se na vymýšlení algoritmů. I pro jednoduché hry jsou optimální strategie často docela komplikované a zajímavé.

Pokud se zaměříme na hledání strategií, je zajímavé pořádat soutěže strategií – ideální je pochopitelně soutěž více programátorů, ale je smysluplné udělat si i „individuální soutěž“, tj. implementovat více strategií a ty porovnat. U některých her necháme strategie hrát přímo proti sobě (např. Piškvorky), u jiných je necháme soutěžit nepřímo pomocí měření jejich výkonu (např. Oběšenec, Tetris).

Pokud jsme ambiciozní, můžeme obě větve spojit – udělat program, který bude mít pěkné uživatelské rozhraní a současně bude umět sofistikovaně hrát. Zde uvádíme několik spíše jednoduchých her, pokud vás tento styl zajmě, lze podobným stylem pochopitelně zpracovat mnoho dalších klasických her.

## 7.1 Kámen, nůžky, papír

<b>Nápad:</b>	1-4
<b>Kódování:</b>	1-3
<b>Styl úlohy:</b>	Jednoduchá hra vhodná pro základní programátorská cvičení, současně však skrývá nečekaný potenciál pro složité analýzy a zajímavá rozšíření.

Kámen, nůžky, papír je klasická losovací hra: každý z hráčů si vybere jeden ze tří uvedených symbolů, současně si je ukážou a výsledek vyhodnotí následujícím způsobem: kámen tupí nůžky, nůžky stříhají papír, papír balí kámen. Dva stejné symboly představují remízu. Tento typ hry lze zapsat pomocí tabulky, která pro každou kombinaci tahů udává bodový zisk jednotlivých hráčů. V tabulce 7.1 je uveden takový zápis pro hru Kámen, nůžky, papír a pro Dilema vězně, což je jiná velmi známá hra tohoto typu.

V tomto cvičení se budeme zabývat opakováním hraním této hry, a to jak hrou člověka proti počítači, tak souboru počítačových strategií. Při hře člověka s počítačem lze jen těžko realizovat současné ukázání symbolů, takže počítač si pouze „myslí“ svůj symbol, a když člověk zadá svůj tah, počítač ukáže vyhodnocení. Ovšem takto by počítač snadno mohl podvádět, resp. uživatel nemá důvod věřit počítači, že nepodvádí. Jeden z úkolů tohoto cvičení je vymyslet mechanismus, pomocí kterého si člověk může ověřit, že počítač nepodvádí, a který současně nedá člověku ve hře žádnou výhodu. Úkoly:

1. Napište program, který bude hru hrát proti člověku, přičemž volbu mezi symboly bude provádět zcela náhodně.
2. Vymyslete mechanismus kontroly podvádění a přidejte jej do svého programu.
3. Implementujte několik strategií, např. „náhodná volba“, „vždy kámen“, „50 % kámen, 30 % papír, 20 % nůžky“, „pravidelné střídání symbolů“. Vytvořte program, který sehraje turnaj mezi těmito strategiemi a vypíše výsledky.

**Tabulka 7.1:** Příklady her v tabulkovém zápisu: Kámen, nůžky, papír (vlevo) a Dilema vězně (vpravo)

	Kámen	Nůžky	Papír	Spolupráce	Zrada
<b>Kámen</b>	0; 0	1; -1	-1; 1	3; 3	0; 5
<b>Nůžky</b>	-1; 1	0; 0	1; -1	5; 0	1; 1
<b>Papír</b>	1; -1	-1; 1	0; 0		

4. Zkuste vymyslet strategii, která bude ve hře co nejúspěšnější (proti lidem, resp. v turnaji výše uvedeného typu).
5. Realizujte předchozí úkoly pro jiné hry (např. uvedené Dilema vězně), případně rovnou pro obecnou hru, tj. program jako svůj vstup načte tabulkou popisující hru.

### Doplňující komentář

Analýzou her uvedeného typu se zabývá oblast aplikované matematiky zvaná teorie her. Ukazuje se, že i hry s velmi jednoduchým zadáním mohou být komplikované. Například zmíněná hra Dilema vězně je klasickou ukázkou takzvané „hry s nenulovým součtem“ a existují o ní desítky vědeckých článků (viz shrnutí v Pelánek, 2011b). Zde se detailněji zaměříme pouze na hru Kámen, nůžky, papír.

Ke kontrole podvádění lze použít „jednosměrné funkce“, což je důležitý koncept používaný v kryptografii (mj. právě ke kontrole podvádění). Jednosměrná funkce je taková funkce, která jde snadno vypočítat jedním směrem, ale těžko opačným. Typickým příkladem jednosměrné funkce je násobení prvočísel – vynásobit prvočísla je jednoduché, ale rozložit číslo na prvočinitele je náročné. Ke kontrole podvádění můžeme jednosměrnou funkci využít následovně. Počítač vybere svůj tah, aplikuje na něj jednosměrnou funkci a výsledek zveřejní člověku ještě před tím, než člověk vybere svůj tah. Může například vybrat několik náhodných velkých prvočísel (2 prvočísla = kámen, 3 prvočísla = nůžky, 4 prvočísla = papír), ty vynásobí a výsledek zveřejní člověku. Protože je funkce jednosměrná, člověk z výsledku nemůže nijak snadno poznat, jaký symbol počítač zvolil. Po vyhodnocení tahu však počítač zveřejní použité vstupy (konkrétní prvočísla) a člověk si může ověřit, že počítač nepodváděl.

Co se týče strategií, na jednu stranu je hra Kámen, nůžky, papír triviální. Dá se snadno ukázat, že při hře racionálních hráčů je optimální čistě náhodná strategie, což znamená v každém tahu se stejnou pravděpodobností hrát kámen, nůžky nebo papír. Nicméně pokud hrajeme v prostředí, ve kterém se vyskytují neracionální strategie, už to zdaleka není tak jednoduché. Pokud protihráč používá strategii „vždy kámen“, můžeme proti němu jistě hrát lépe než čistě náhodně. O tom, že hra Kámen, nůžky, papír je složitější, než se na první pohled zdá, svědčí například pravidelné turnaje v této hře. Konají se turnaje jak mezi lidmi, tak mezi počítačovými strategiemi.

Pokud hrajeme proti lidem, je rozhodně smysluplné zkoušet vymýšlet sofistikovanější strategie – i pokud člověk ví, že je nejlepší hrát zcela náhodně, je to pro něj obtížné. Jak tedy hrát rozumněji? Používají se strategie, které analyzují historii hry, hledají v ní vzory a podle toho se rozhodují.

## 7.2 Hádání čísla

**Nápad:** 2

**Kódování:** 2-3

**Styl úlohy:** *Jednoduchá hra pro ilustraci konceptu binárního vyhledávání a logaritmické složitosti.*

Hru hrají dva hráči – „skrývající“ a „hádající“. Skrývající hráč si myslí přirozené číslo z dohodnutého intervalu a hádající hráč se snaží číslo uhádnout. Hádající hráč vždy tipne číslo a dostane od skrývajícího hráče informaci, zda je tajné číslo větší nebo menší. Průběh hry může být například následující:

Myslím si číslo do 1 do 40.

Jaké číslo tipuješ? 20

Moje číslo je menší.

Jaké číslo tipuješ? 10

Moje číslo je větší.

Jaké číslo tipuješ? 15

Moje číslo je větší.

Jaké číslo tipuješ? 17

Správně!

Zadání:

1. Napište program, který bude hrát na pozici skrývajícího hráče.
2. Napište program, který bude hrát co nejlépe na pozici hádajícího hráče.
3. Napište program, který bude hrát hru sám proti sobě (tj. bude hrát role obou hráčů), pomocí programu sehrajte opakovaně mnoho her a vypočítejte průměrný počet tahů potřebných pro uhádnutí čísla.

### Doplňující komentář

Tato hra je vhodná pro ilustraci a procvičení konceptu binárního vyhledávání, což je jeden z klíčových pojmu v informatice, a u této hry je jeho použití intuitivní. Pokud je hledané číslo v intervalu od  $a$  do  $b$ , je vcelku přirozené hádat číslo uprostřed tohoto intervalu. Pokud takto postupujeme, v každém kroku interval zmenšíme na polovinu. Maximální počet otázek, které potřebujeme na uhádnutí čísla, je tedy roven dvojkovému logaritmu z počátečního počtu možností.

## 7.3 Oběšenec

**Nápad:** 2-4

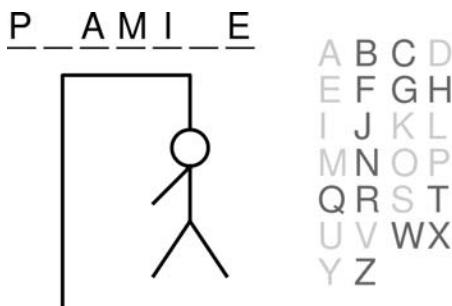
**Kódování:** 2-4

**Styl úlohy:** Jednoduchá interaktivní hra se slovy s algoritmicky zajímavými rozšířeními.

Tato úloha je podobného stylu jako předchozí. Opět jde o hru typu „hádej, co si myslím“, kdy proti sobě hrají skrývající hráč a hádající hráč. Jde o klasickou hru ze školních lavic. Skrývající hráč si myslí smysluplné slovo. Na začátku hádání poskytne druhému hráči pouze informaci o délce slova. Hádající hráč postupně tipuje písmena. Pokud se písmeno ve slově vyskytuje, doplní ho hádající hráč na správnou pozici. Pokud se písmeno ve slově nevyskytuje, dostane hádající hráč trestný bod. Tyto trestné body se u této hry tradičně znázorňují postupným vykreslováním obrázku oběšence (viz obrázek 7.1).

U hry Oběšenec musí mít počítač k dispozici seznam slov, za tímto účelem lze použít například seznam slov zmíněný na straně 63. Abychom si zjednodušili kódování, je vhodné používat pouze písmena anglické abecedy, tj. buď ze slovníku českých slov odstraníme diakritiku, nebo použijeme slovník anglických slov. Hra nabízí několik programátorských úloh:

- „Uživatel hádá“. Počítač pouze vyhodnocuje tahy. V této variantě se můžeme zaměřit i na uživatelské rozhraní a postupné vykreslování oběšence. V dalších variantách už se zaměřujeme na algoritmy a místo vykreslování oběšence jen počítáme počet pokusů.
- „Počítač hádá“. Cílem je, aby počítač za využití slovníku slov hádal co nejfektivněji. Výsledný algoritmus vyhodnotíme na velkém počtu slov, abychom věděli, na kolik průměrně pokusů zvládne algoritmus uhádnout slovo a mohli tak porovnávat různé varianty algoritmu nebo



Obrázek 7.1: Ukázka hry Oběšenec

soupeřit s kamarády o nejlepší algoritmus. Pro tento účel implementujeme variantu „schizofrenik“, kdy počítač hraje sám proti sobě.

3. „Podvodník“: Počítač je skrývající hráč, uživatel je hádající hráč, ovšem počítač trochu podvádí. Na začátku hry nevybere žádné konkrétní slovo. Odpovědi dává tak, aby vždy byly konzistentní, tedy aby vždy existovalo slovo, které splňuje všechny dosavadní odpovědi, a aby při tom hádajícímu hráči situaci co nejvíce zkomplikoval.

### Doplňující komentář

Přímočarý přístup k hádání je skrze frekvence písmen – hádáme písmena, která jsou více frekventovaná, přičemž je však důležité počítat frekvence pouze v rámci slov, která jsou možnými kandidáty vzhledem k dosavadním pokusům. Například písmeno n je obecně v češtině frekventovanější než písmeno c, ale pokud se omezíme na slova pasující do situace znázorněné na obrázku 7.1, je c rozhodně lepší volba. Přístup přes frekvence písmen je přímočarý, snadno implementovatelný a dává docela dobré výsledky. Existuje však i lepší řešení a jeho základní myšlenka je klíčová pro variantu „podvodník“.

Efektivní hádání spočívá v tom, že se každým dotazem snažíme co nejvíce zúžit množinu slov, ve které může hledané slovo ležet. Ideální dotaz je tedy takový, pro který všechny možné odpovědi rozdělují množinu kandidátů na podobně velké skupiny – podobně jako u hádání čísla jsme volili takový tip, který kandidáty rozdělí na podobně velké skupiny. V případě hádání čísla to dělení probíhalo na poloviny a volba vhodného dotazu byla intuitivní.

Zde je to trochu komplikovanější a vyplatí se zavést si na to trochu terminologie – pojem ekvivalence slov vzhledem k dotazu. Slova jsou ekvivalentní vzhledem k dotazu, pokud pro obě dostaneme na dotaz stejnou odpověď. Například slova „postel“ a „Brusel“ nejsou ekvivalentní vzhledem k dotazu na písmeno „s“ nebo „r“, ale jsou ekvivalentní k dotazu na písmeno „l“ nebo „z“. Třída ekvivalence je pak množina slov vzájemně ekvivalentních.

Rozumná strategie pro hádání je pak následující: pro každé písmeno napočítáme třídy ekvivalence vzhledem k tomuto písmenu. Jako dotaz vybereme to písmeno, pro které je jeho největší třída ekvivalence co nejmenší, tj. rozděluje množinu kandidátů co nejrovnoměrněji. Ve variantě podvodník pak program pracuje inverzně – vybírá odpověď tak, aby nová množina kandidátů (třída ekvivalence) byla co největší.

Uvedená strategie je „rozumná“, nikoliv „optimální“. Strategie jde vylepšit například uvažováním více tahů dopředu, případně i zohledněním frekvence slov v jazyce (pokud hrájeme proti člověku, dá se očekávat, že ho budou napadat spíše frekventovaná slova).

## 7.4 Logik

<b>Nápad:</b>	3-4
<b>Kódování:</b>	3-4
<b>Styl úlohy:</b>	Známá hra, u které jde docela snadno udělat vcelku chytrou „umělou inteligenci“.

Tato hra je opět typu „hádej, co si myslím“, kdy proti sobě hrají skrývající hráč a hádající hráč. I dílčí úkoly jsou analogické úkolům v předchozím zadání, ale pravidla jsou trochu jiného typu a určitě je užitečné zkusit implementovat obě úlohy.

Hra je u nás známá pod názvem Logik, v angličtině pod názvem Mastermind. V základní verzi se hraje s kuličkami šesti barev, ze kterých skrývající hráč čtyři vybere a uspořádá je do zvoleného fixního pořadí. Každá barva může být použita maximálně jedenkrát. Úkolem hádajícího hráče je odhalit barvy těchto kuliček a jejich pořadí.

Hádající hráč v každém tahu předloží svůj tip – čtveřici barevných kuliček. Skrývající hráč tip vyhodnotí a výsledek předloží hádajícímu hráči pomocí černých a bílých kolíků. Počet černých kolíků je roven počtu kuliček, které mají správnou barvu a jsou na správném místě. Počet bílých kolíků je roven počtu kuliček, které mají správnou barvu, ale jsou na špatném místě.

Cílem hádajícího hráče je uhádnout správnou kombinaci na co nejméně pokusů. Obrázek 7.2 znázorňuje příklad hry. Hru lze pochopitelně hrát i s jiným počtem barev a vybraných kuliček, při programování uvažujeme hru obecně s  $n$  barvami a  $k$  kuličkami.

1		
2		
3		
4		
5		

Obrázek 7.2: Ukázka hry Logik (místo barev jsou použity vzory)

Zadání programátorských cvičení jsou stejného typu jako u hry Oběšenec:

1. „Hráč hádá“. Počítač je skrývající hráč, tj. problém spočívá pouze v tom, vytvořit uživatelské rozhraní a správně interpretovat pravidla hry.
2. „Počítač hádá“. Úkolem je vymyslet algoritmus, podle kterého se bude počítač řídit („umělou inteligencí“). Pro testování je ideální implementovat verzi „schizofrenik“ („počítač proti počítači“), což nám umožní například spustit velký počet her a vyhodnotit, kolik průměrně potřebuje náš program na dokončení hry.
3. „Podvodník“. Počítač je skrývající hráč, uživatel je hádající hráč, ovšem počítač trochu podvádí. Na začátku hry počítač nevybere žádnou konkrétní konfiguraci kuliček. Odpovědi dává tak, aby vždy existovala konfigurace splňující všechny dosavadní odpovědi a aby při tom hádajícímu hráči situaci co nejvíce zkomplikoval.

### Doplňující komentář

Základní princip umělé inteligence je jednoduchý – využijeme „hrubou sílu“, vyčíslíme si všechna možná rozložení kuliček a ty budeme filtrovat podle výsledků tipů. Konstrukce všech možných konfigurací odpovídá generování variací (viz cvičení 3.10). Pro  $n$  barev a  $k$  kuliček je počet všech variací  $\frac{n!}{(n-k)!}$ , pro zmíněnou základní variantu s šesti barvami a čtyřmi kuličkami je to jen 360. I pro větší počty barev a kuliček, pokud se pohybujeme v rozsahu, kde je schopen hru rozumně hrát člověk, je celkový počet variací dostatečně nízký na to, aby bylo možné všechny variace vyčíslit.

Jakmile máme zhotoven seznam všech možných kandidátů, v každém tahu jednoho z nich vybereme a na základě obdrženého výsledku pak seznam kandidátů profiltrujeme, tj. necháme tam jen ty prvky, které při porovnání s naším tipem dávají stejný výsledek jako odpověď, kterou jsme dostali.

Abychom hráli co nejfektivněji, měli bychom volit tip tak, aby obdržená odpověď co nejvíce profiltrovala seznam kandidátů. Opravdu dobré provedení tohoto výběru je netriviální – odpovídá rozkladům na třídy ekvivalence, které jsou zmíněny u úlohy Oběšenec. Nicméně i pokud tip vybíráme náhodně, tak program hraje velmi dobře – rozhodně lépe než normální člověk.

U této hry byly studovány i optimální strategie. Je například známo, že pro základní variantu hry s šesti barvami a čtyřmi kuličkami lze hru vždy dokončit na 4 pokusy. Autorem řešení, které je poměrně komplikované, je D. Knuth, jeden z nejznámějších informatiků.

## 7.5 Hra Nim

<b>Nápad:</b>	2-5
<b>Kódování:</b>	2-3
<b>Styl úlohy:</b>	<i>Programování optimální strategie pro několik variant jednoduché hry. Vždy existuje krátké a elegantní řešení, v některých případech je však velmi obtížné jej vymyslet.</i>

Pod názvem Nim se označuje skupina her, která spočívá v odebíráni sirek z hromádky podle zadaných pravidel. V základní variantě je povoleno odebrat 1, 2 nebo 3 sirkы. Hráči se střídají v tazích, prohrává ten, kdo nemůže táhnout, protože už na něj nezbyly sirkы.

Stav hry a platné tahy lze u Nimu jednoduše reprezentovat číslы, takže „uživatelské rozhraní“ pro tuto hru je velmi jednoduché. Hráč si na začátku hry zvolí, jakou variantu hry chce hrát, s kolika sirkami se bude začínat a kdo bude začínat (hráč nebo počítač). Potom se již hráč a počítač střídají v tazích, počítač aktualizuje a vypisuje aktuální stav hry (zbývající počet sirek). Příklad průběhu hry:

```

Chceš začínat? A
S kolika sirkami budeme hrát? 10
Zbývá serek: 10. Kolik odebereš? 3
Zbývá serek: 7. Já odebíram 3.
Zbývá serek: 4. Kolik odebereš? 4
Neplatný tah.
Zbývá serek: 4. Kolik odebereš? 2
Zbývá serek: 2. Já odebíram 2.
Zbývá serek: 0. Vyhral jsem.

```

Cvičení spočívá především ve vymyšlení strategie pro počítač. Cílem je vytvořit program, který bude hrát optimální strategii, tj. pokud dostane příležitost zvítězit, využije ji. Můžeme vyzkoušet následující varianty hry:

1. Základní Nim: Je povoleno odebrat 1, 2 nebo 3 sirkы.
2. Vylučovací Nim: Hraje se podobně jako základní varianta, pouze máme navíc přidané pravidlo, že hráč nesmí odebrat tolik serek, kolik v posledním tahu odebral soupeř.
3. Patrový Nim: Tentokrát máme sirkы vyskládané v  $n$  patrech po  $2, 4, \dots, 2n$  sirkách. Hráči odebírají opět 1 až 3 sirkы, ale musí postupovat po patrech od nejmenšího po největší a v jednom tahu mohou brát sirkы vždy jen z jednoho patra. Pokud tedy například v patře zbývá jen 1 sirkа, nemá hráč na vybranou a musí pouze dobrat toto patro.

4. Základní Nim s volitelnými tahy, tj. na začátku partie je určen seznam povolených tahů.
5. Hromádkový Nim. V této variantě máme několik hromádek sirek. Hráč může odebrat libovolný počet sirek z libovolné hromádky, ale jen z jedné.

### Doplňující komentář

Obtížnost strategií se liší podle variant. První tři varianty (základní, vylučovací a patrový Nim) lze vyřešit tak, že optimální strategii prostě vymyslíme a zakódujeme do programu. V těchto případech optimální strategie vždy nějakým způsobem souvisí s dělitelností a není příliš těžké ji vymyslet a implementovat.

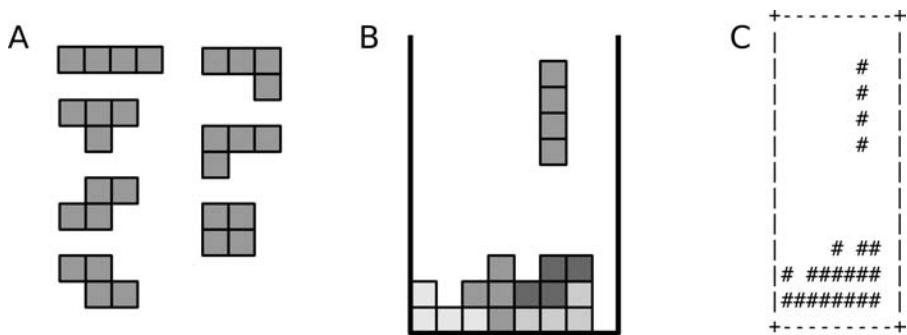
U Nimu s volitelnými tahy už nemůžeme strategii zapsat do programu „na tvrdo“, ale musíme vymyslet obecný algoritmus, který strategii vypočítá. Jde o typický příklad na použití dynamického programování – optimální strategii budujeme „odspodu“, od menších počtů sirek k větším.

Hromádkový Nim je již poměrně náročná úloha. Můžeme jej opět řešit podobně jako předchozí verzi, tj. napočítáním „odspodu“, nicméně v tomto případě počet možných stavů hry roste exponenciálně – již pro 3 hromádky po 9 sirkách tak máme celkem 1 000 stavů. Na hraní proti lidem je dostatečný i tento přístup, nicméně existuje i výrazně efektivnější a elegantnější řešení, které využívá převod na binární čísla a počítání parity binárních cifer. Zkuste jej najít!

## 7.6 Tetris

<b>Nápad:</b>	2-5
<b>Kódování:</b>	4-5
<b>Styl úlohy:</b>	<i>Programování jedné z klasických počítačových her. Cvičení lze zaměřit na grafickou stránku hry a interakci s uživatelem nebo na umělou inteligenci pro hraní hry.</i>

Tetris je jedna z nejznámějších počítačových her. Hraje se na čtverečkování mřížce, do které padají v náhodném pořadí tetromina, což jsou dílky tvořené ze čtyř kostek. Na obrázku 7.3 jsou ilustrovány tetromina a rozehraná hra. Kostka se objeví vždy na vršku hracího plánu a pomalu padá dolů. Hráč může kostkou otáčet a posouvat ji doleva a doprava. Jakmile se kostka zasekně, protože nemůže již dále padat, objeví se nová kostka. Kromě aktuální padající kostky hráč může též vidět 1 či více kostek dopředu, takže ví, na co se má připravit.



**Obrázek 7.3:** Tetris: A) tetromina, B) grafické znázornění hry, C) textové znázornění hry

Pokud se hráči podaří zcela zaplnit některý řádek, celý tento řádek zmizí a všechny řádky nad ním se posunou o jednu pozici dolů. V situaci na obrázku 7.3 k tomu dojde, pokud padající kostku umístíme do pravého sloupce. Pokud se hráči nedáří plnit celé řádky, kostky se na sebe kupí, až zaplní celou mřížku, čímž se hra ukončí. Cílem hry je vydržet hrát co nejdéle.

Cvičení lze pojmut několika způsoby:

1. Minimalistická verze. Vytvořit jen zcela základní hru v textovém režimu.
2. Grafická verze. Vytvořit hru v grafickém režimu, se zobrazením budoucích kostek a příjemným uživatelským prostředím.
3. Umělá inteligence. Hru hraje program, tj. rozhoduje o umístění padajících kostek. Cílem je program vyladit tak, aby hrál co nejlépe.

### Doplňující komentář

Minimalistická verze slouží především jako cvičení v reprezentaci dat. Program by měl určitě dodržovat „oddělení dat a funkcionality“, tj. definice tvarů kostek by měly být odděleně od definice funkcionality (rotace, padání), takže případné přidání nebo odstranění kostky by mělo znamenat změnu maximálně jednoho řádku kódu. Kód by rozhodně neměl obsahovat spousty `switch` nebo `if/else` příkazů, které větví mezi různými kostkami. Pokud si plán a kostky dobře reprezentujeme, může být výsledný kód velmi krátký.

V textové verzi potřebujeme umět pracovat se znaky v textovém terminálu. Pod Linuxem k tomuto účelu například slouží knihovna `curses`, která je dostupná v mnoha programovacích jazycích. Grafická verze je vhodná jako cvičení pro základní práci s grafikou a vytvářením grafického uživatelského rozhraní, myšlenkově oproti minimalistické verzi nepřináší nic složitého.

Myšlenkově je pochopitelně nejzajímavější verze „umělá inteligence“. U této verze je důraz na hledání vhodného algoritmu, takže zde naopak můžeme složitost uživatelského rozhraní úplně minimalizovat – ani nemusíme vykreslovat padání kostek, stačí si vždy vykreslovat stav mřížky poté, co umístíme další kostku. Složitost problému závisí na tom, jaké máme ambice. Vytvořit program, který bude hrát „alespoň trošku rozumně“, není příliš obtížné a vystačíme s intuitivními nápady. Avšak udělat program, který bude hrát opravdu dobře a bude třeba brát v potaz informaci o budoucích kostkách, už může být docela rozsáhlý projekt, vyžadující dobré rozmyšlení heuristik a implementaci prohledávání stromu všech možností.

## 7.7 Jednorozměrné piškvorky

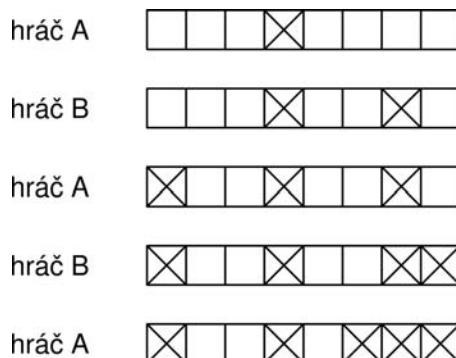
**Nápad:** 2-5

**Kódování:** 2-3

**Styl úlohy:** *Programátorský nenáročná úloha, algoritmickou obtížnost lze ladit podle toho, jak schopného hráče chceme vytvořit.*

Tato variace piškvorek se hraje na jednorozměrném hracím plánu, tj. hrací plán je řada polí délky  $N$ . Stejně jako u běžných piškvorek se hráči střídají a dělají značky do políček, tentokrát však oba hráči dělají křížky. Vyhrává hráč, který jako první vytvoří alespoň tři křížky vedle sebe. Na obrázku 7.4 je ukázka hry, ve které vyhrál hráč A.

Úkolem je napsat program, který bude hrát proti hráči. Podle toho, jaké máme ambice na „inteligenci“ programu, lze volně ladit obtížnost úlohy:



**Obrázek 7.4:** Jednorozměrné piškvorky – ukázka hry na hracím plánu délky 8

1. Program hraje korektně, ale jinak zcela náhodně. Program rozpozná, když někdo vyhraje.
2. Program „nedělá hlouposti“. Zakončí, pokud může bezprostředně vyhrát, a snaží se nenahrávat hráči na bezprostřední výhru.
3. Program hraje optimálně alespoň pro některé velikosti herního plánu.
4. Program hraje optimálně pro všechny velikosti herního plánu, resp. prakticky stačí pro  $N < 100$ .

### Doplňující komentář

První dva stupně obtížnosti jsou přímočará programátorská cvičení, která nevyžadují žádný zvláštní nápad a slouží pouze k procvičení základních programátorských konstrukcí (interakce s uživatelem, jednorozměrné pole). Hledání optimální strategie už je zajímavější, v rámci komentáře nabídneme pouze několik stručných tipů:

- Jednoduše jde odvodit a implementovat vítězná strategie pro prvního hráče pro herní plány liché velikosti.
- Pro plány velikosti  $N < 20$  lze hledat vítěznou strategii pomocí hrubé síly (algoritmus minimax, rozšíření backtrackingu) – prostě zkoušíme projít všechny platné konfigurace hry. Díky jednoduchým pravidlům hry lze zapsat na pár řádků kódu.
- Pro herní plány větší velikosti už hrubá síla není dostatečná a musíme vymyslet „inteligentní“ způsob řešení pomocí rozkladu hry na „podhry“. Po umístění křížku na plán můžeme hru vnímat jako kombinaci her v levé a v pravé polovině herního plánu.

## 7.8 Piškvorky

<b>Nápad:</b>	3-5
<b>Kódování:</b>	4-5
<b>Styl úlohy:</b>	<i>Klasická hra dvou hráčů vhodná pro pořádání turnajů mezi strategiemi.</i>

Po rozcvičce na jednorozměrných piškvorkách nyní přejděme ke klasické variantě této oblíbené hry. Hraje se na dostatečně velkém čtverečkovém hracím plánu, vhodná velikost je například  $20 \times 20$ . Hráči ve svém tahu udělá jednu svoji značku (křížek či kolečko) do volného pole. Vyhrává ten, kdo první vytvoří řadu z pěti svých znaků. Řada může být vodorovně, svisle nebo diagonálně.

V tomto případě již není reálné najít optimální strategii. Úkolem je tedy napsat program, který bude hru hrát co nejlépe. Zajímavý test schopností programu je, zda zvládne porazit svého tvůrce. Krom soubojů s člověkem jsou piškvorky také vhodné pro pořádání turnajů mezi počítačovými strategiemi.

### Doplňující komentář

Implementace základního interpretu hry (reprezentace stavu, kontrola platnosti tahu, kontrola vítězství) je docela přímočaré programátorské cvičení. Možnosti realizace „souboje programů“ jsou rozebrány u následujícího cvičení, v tomto komentáři se zaměříme pouze na realizaci „umělé inteligence“.

Relativně jednoduchý algoritmus pro hraní hry získáme simulováním člověka začátečníka: program hledá „vzory“, jako například tři nekryté křížky vedle sebe, a jakmile nějaký najde, táhne. Přesněji řečeno algoritmus funguje tak, že popíšeme seznam vzorů, pro každý vzor určíme, jaký tah z něho vyplývá, a vzory seřadíme podle priority. Pokud algoritmus hraje za křížky, první na seznamu bude vzor „čtyři křížky za sebou a volné místo“, kterému odpovídá tah „hraj na ono volné místo“, dál na seznamu bude hledání neblokovaných čtveric koleček, trojic křížků a tak dále.

Algoritmus postupně bere vzory ze seznamu a prohledává celé hrací pole, jestli v některém z 8 možných směrů nenajde výskyt vzoru. Jakmile najde první výskyt, táhne. Algoritmus tedy neprovádí žádné analýzy na několik tahů dopředu. Díky tomu má některé zjevné nedostatky (není například schopen rozpoznat hrozící „vidličku“) a průměrný lidský hráč jej porazí. Nicméně na to, jak je program jednoduchý, hraje překvapivě dobrě.

Sofistikovanější algoritmus je založen na prohledávání stavů hry a funguje podobně jako hrubá síla u jednorozměrných piškvorek. V tomto případě je však počet konfigurací příliš velký a není možné projít všechny. Musíme tedy použít heuristiky a ořezané prohledávání, tento postup je popsán podrobněji v řešení.

## 7.9 Souboje virtuálních robotů

<b>Nápad:</b>	4-5
<b>Kódování:</b>	5
<b>Styl úlohy:</b>	Nejde o kompletní zadání, ale pouze o vysvětlení základního principu a několika námětů pro rozsáhlé cvičení.

Toto cvičení je rozsáhlejší než předchozí a má smysl především jako soutěž více programátorů, typicky například turnaj v rámci seminární skupiny nebo celé

školy. Na rozdíl od ostatních cvičení zde nenabízíme zcela konkrétní zadání, ale pouze popis základních principů a ilustrativní příklady.

Zcela základní princip spočívá v programování strategií, které proti sobě soutěží. Tento obecný princip můžeme využít například i na klasické hry, jako jsou piškvorky (viz předchozí cvičení) nebo šachy. Zde se však zaměříme na souboje virtuálních robotů, což má oproti klasickým hrám dvě výhody. Pro klasické hry je většinou docela snadné podvádět a vyhledat na webu silnou strategii. Pokud si však vymyslíme vlastní hru s roboty, musí soutěžící přijít s vlastními nápady. Druhá výhoda spočívá v tom, že roboti, byť virtuální, zvlášť pokud po sobě střílí, jsou pro velkou část populace, zejména mužské, jaksi vrozeně atraktivní.

Pro zajímavé souboje robotů stačí docela jednoduchá pravidla hry. Virtuální roboty umístíme do jednoduchého světa a dáme jím k dispozici několik málo příkazů. Herní svět typicky představuje pravidelná čtvercová mřížka se zdmi a příkazy jsou typu udělat krok dopředu, otočit se nebo vyštřelit. Úkolem je napsat strategii, která robota ovládá, tedy na základě informací o aktuálním stavu hracího světa rozhodne, jakou akci má robot udělat v dalším kole. Cílem je pochopitelně napsat takovou strategii, která v souboji porazí ostatní.

Pro realizaci soubojů musíme implementovat nejprve interpret hry. Interpret dostane popis hracího plánu a seznam strategií, odehraje hru, přičemž pokud možno graficky znázorní její průběh, a dá informaci o výsledku. Pokud organizujeme turnaj strategií, je vhodné, aby interpret hry vytvořil organizátor turnaje a soutěžící pouze implementovali dílkí strategie.

Pro inspiraci jsou dále uvedeny příklady tří pravidel, které se liší stylem „kdo soutěží“ (individuální roboti, decentralizovaný tým robotů, centralizovaný tým robotů). Tato pravidla byla použita v prvních třech ročnících programátorské soutěže FIbot na FI MU, detaily pravidel a implementace interpretů jsou k dispozici na [www.fi.muni.cz/FIbot](http://www.fi.muni.cz/FIbot). Pravidla her jsou relativně jednoduchá, ale i tak byl vývoj dobrých strategií docela náročný. Pokud neděláte soutěže pro experty v umělé inteligenci, doporučuji použít pravidla stejně nebo nižší složitosti.

**Individuální roboti** Herní plán je čtvercová mřížka, na které jsou zdi, poklady a baterie. Povolené akce robotů jsou: krok, otočení, vyštřelení a odpočívání. Na plánu soutěží vždy 4 roboti, cílem hry je získat co nejvíce bodů. Body získávají za sběr pokladů a za zásah jiných robotů. Roboti mají omezenou energii. Každá akce odčerpá 1 jednotku energie, zásah střelou ubere polovinu stávající energie, energii lze dobíjet pomalu pomocí akce odpočívání nebo rychle na polích s baterií. Roboti tedy mohou používat jen elementární akce, ale jejich plánování akcí vyžaduje vyvažování několika aspektů hry (sběr pokladů, střílení, stav energie, poloha ostatních robotů).

**Decentralizovaný tým robotů** Plán je opět čtvercová mřížka se zdmi a poklady. Tentokrát proti sobě soutěží dva týmy robotů, každý z nich má na plánu svoji základnu. Poklady tentokrát nestačí jen sbírat, roboti je musí nanosit na vlastní základnu. Roboti jsou ovládáni decentralizovaně – to znamená, že každý robot se rozhoduje pouze na základě informace o té části herního plánu, kterou „vidí“. Roboti mají k dispozici jen následující akce: posun na vedlejší pole, zvednutí pokladu, položení pokladu a poslání krátké zprávy všem robotům.

Přestože jsou pravidla hry hodně jednoduchá, je vývoj rozumných strategií docela komplikovaný. Není například jednoduché zařídit již takovou základní věc, aby se roboti stejného týmu vzájemně neblokovali, když jdou proti sobě v úzké chodbě. Pravidla hry navíc poskytují prostor pro mnoho taktických prvků, jako je například zablokování základny soupeře.

**Centralizovaný tým robotů** V této variantě jde opět o souboj dvou týmů, tentokrát však s centrálním vedením. Strategie ovládající roboty má k dispozici náhled na celý herní svět a rozhoduje o volbě akce pro všechny roboty. Opět se hraje na čtvercové mřížce se zdmi, ale tentokrát velkých rozměrů (maximálně  $200 \times 200$ ). Jediná povolená akce je pohyb v jednom z osmi směrů o maximálně 7 polí. Při pohybu roboti automaticky zabírají území okolo sebe, a to v okruhu o poloměru 9 polí. O pole ovšem mohou později přejít, pokud jim je „ukradne“ soupeř (tím, že projde okolo). Cílem hry je zabrat co největší území.

Turnaj se hrál v tomto případě stylem ligy (každý s každým). Zajímavým aspektem soutěže bylo to, že jde o „hru s nenulovým součtem“. Není to tak, že ve vzájemném souboji dvou strategií by jedna striktně vyhrála a druhá prohrála. Strategie získávají body podle toho, jak velkou část plánu na konci hry kontrolují. Může se tedy stát, že dvě soutěžící strategie si plán rozdělí zhruba na poloviny a obě získají 50 bodů, kdežto jiné dvě strategie spolu budou horlivě bojovat uprostřed plánu a nakonec získají každá po 5 bodech.

### Doplňující komentář

Realizaci soubojů strategií můžeme realizovat dvěma základními způsoby. První možnost spočívá ve využití společného jazyka a objektově orientovaného programování, konkrétně je vhodná například Java. Ve zvoleném jazyce je napsán interpret hry, který specifikuje rozhraní, přes které se strategiemi komunikuje, tj. předává informace o herním plánu a dostává informace o tazích. Implementace strategií pak spočívá v implementaci třídy odpovídající zadanému rozhraní.

Druhý způsob necházá soutěžícím možnost volby vlastního jazyka, jediné, co musí dodat, je spustitelný program. Komunikace mezi interpretem hry

a strategiemi probíhá skrze soubory. Interpret hry zapíše stav herního plánu do souboru, zavolá strategii, ta stav hry načte ze souboru, vypočítá další tah a ten zapíše do dalšího souboru, odkud jej zpět načte interpret hry. Tento přístup je náročnější na realizaci opravdu robustního interpretu hry, který bude odolný i proti zlomyslným soutěžícím. Je tedy vhodný především pro turnaj pro malou skupinu soutěžících, kterým věříme, že nebudou podvádět, a tudíž si vystačíme i s přímočarou implementací interpretu soubojů.

Zájemci o téma soubojů strategií mohou najít na webu celou řadu podobných soutěží, konkrétně například na stránkách [robocode.sourceforge.net](http://robocode.sourceforge.net), [www.robotbattle.com](http://www.robotbattle.com), [www.ceebot.com/colobot](http://www.ceebot.com/colobot). Příkladem pokročilé soutěže tohoto typu jsou turnaje strategií ve hře StarCraft, což je rozsáhlá komerční strategická hra primárně určená pro lidské hráče.



# 8 Klasické informatické problémy

*Informatika není o počítačích o nic víc než astronomie o dalekohledech.*

E. W. Dijkstra

Styl této kapitoly je odlišný od předchozích kapitol. Příklady nemají podobné téma, vzhled ani způsob algoritmického řešení. Jediným spojujícím prvkem je, že jde o klasické informatické problémy – bud' jsou to hodně studované případové studie nebo klasické problémy, na kterých se ilustrují vlastnosti některých algoritmů.

Tyto příklady jsou na první pohled méně atraktivní než příklady v jiných kapitolách, nicméně jakmile se do nich zanoříme, zjistíme, že jsou docela zajímavé. Navíc jsou to rozhodně užitečné příklady – ne nadarmo jsou to „klasické problémy“. V praxi sice téměř nikdy nebudeme řešit přímo tyto problémy, avšak principy, které příklady ilustrují, se vyskytují často a získané zkušenosti rozhodně přijdou vhod.

## 8.1 Rozměňování mincí

**Nápad:** 2-4

**Kódování:** 2-3

**Styl úlohy:** *Dobré cvičení na ilustrování základních principů hladových algoritmů a dynamického programování. Programátorský nenáročné, ale myšlenkově zajímavé.*

Základní problém této úlohy známe dobře z běžného života. Pokud chceme někomu vyplatit určitou částku, jaký je nejmenší počet mincí a bankovek, které k tomu musíme použít? Abychom zjednodušili popis, nebudeme dále v tomto příkladu mluvit o mincích a bankovkách, ale prostě jen o mincích. Uvažme několik variant problému rozměňování zadané částky (v tabulce 8.1 jsou uvedeny příklady konkrétních vstupů a výstupů).

**Tabulka 8.1:** Rozměňování mincí – příklady

Úloha	Vstup	Výstup
1. Klasické mince	169	2, 2, 5, 10, 50, 100
2. Zadané mince	19; 1, 3, 8, 12	1, 3, 3, 12
3. Zadané mince, omezené	14; 1 (8x), 3 (2x), 5 (1x)	1, 1, 1, 3, 3, 5
4. Frobeniusův problém	3, 5	7

1. Nejprve předpokládejme, že používáme klasické hodnoty mincí, jak jsou používány v mnoha zemích světa, tj. 1, 2, 5, 10, 20, 50, ... Jako maximální hodnotu mince můžeme vzít třeba 5 000 nebo můžeme uvažovat obecně všechny mince hodnot  $10^k, 2 \cdot 10^k, 5 \cdot 10^k$  pro  $k \geq 0$ . Vstupem problému je částka  $X$ , výstupem je optimální rozměnění požadované částky.
2. Nyní předpokládejme, že jsme v exotické zemi, kde používají mince jiných hodnot. Tentokrát je vstupem problému kromě celkové částky  $X$  také seznam dostupných mincí. Výstupem je opět optimální rozměnění požadované částky.
3. Stejně jako předchozí problém, ale tentokrát máme omezené počty mincí, tj. součástí vstupu je i informace o tom, kolikrát je která mince k dispozici.
4. Pokud jsme v exotické zemi, kde nemají minci hodnoty 1, může se stát, že některé částky nejdou vůbec vyskládat. Pokud jsou hodnoty mincí ne-soudělné, platí, že od určité částky již lze vyskládat jakoukoliv hodnotu. Jaká je nejvyšší hodnota, kterou nelze z mincí vyskládat? Tento problém je znám pod názvem *Frobeniusův problém*.

### Doplňující komentář

Cvičení je vhodné pro ilustraci rozdílů mezi různými přístupy k návrhu algoritmů, konkrétně k ilustraci hladového algoritmu a dynamického programování. Pro „klasické mince“ funguje hladový algoritmus. Částku vyplácíme postupně a v každém kroku prostě vezmeme minci nejvyšší dostupné hodnoty, která je menší než vyplácená částka. Tuto minci přidáme do seznamu použitých mincí a vyplácenou částku o příslušnou hodnotu zmenšíme.

Obecně však hladový algoritmus fungovat nemusí. Například pro hodnoty mincí 1, 3 a 4 a cílovou částku 6 napočítá hladový algoritmus seznam 4, 1, 1, optimální řešení je však 3, 3. Obecný případ lze řešit pomocí dynamického programování. Postupně pro všechny částky od 1 do  $X$  napočítáme, kolik mincí potřebujeme na vyskládání. Například pro hodnoty mincí 1, 3, 4 dostaneme následující tabulku:

částka	1	2	3	4	5	6	7	8	9	10
počet mincí	1	2	1	1	2	2	2	2	3	3

Tuto tabulku lze při troše rozmýšlení zkonstruovat jedním průchodem zleva doprava. Jakmile máme tabulku hotovou, víme, kolik mincí potřebujeme. S trohou rozmyšlení pak pomocí tabulky můžeme i rekonstruovat, které mince konkrétně potřebujeme.

Frobeniusův problém je uveden spíše pro zajímavost, jde o docela náročný problém. Efektivní algoritmy existují pro omezený počet mincí, obecný problém je NP-úplný, tj. není pravděpodobné, že by existoval efektivní algoritmus.

## 8.2 Simulátor hry Život

**Nápad:**

2-3

**Kódování:**

2-4

**Styl úlohy:**

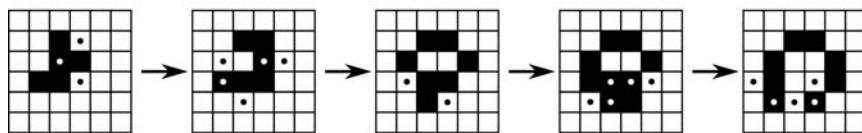
*Cvičení je ukázkou jednoho velmi zajímavého buněčného automatu.  
Po algoritmické a programátorské stránce je cvičení jednoduché.*

Hra Život je velmi známý příklad takzvaného „buněčného automatu“. Buněčný automat je mřížka buněk, která se řídí jednoduchými přechodovými pravidly. Jde o rozsáhlou oblast studia, která se využívá především pro modelování komplexních systémů. Zde se podíváme jen na jeden konkrétní buněčný automat, který abstraktně modeluje život a pomocí velmi jednoduchých pravidel generuje složité a zajímavé chování.

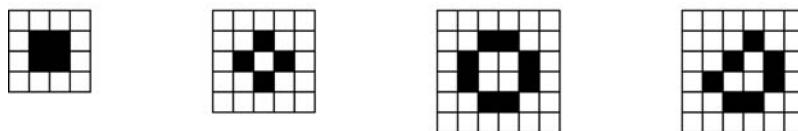
Hra Život se hraje na dvojrozměrné čtvercové mřížce. Název „hra“ je trochu zavádějící, protože nejde o žádnou hru více hráčů, ale o dodržování přesně daných deterministických pravidel. Každá buňka v mřížce může být v jednom ze dvou stavů: živá (černá) nebo mrtvá (bílá). Hra se hraje na kole a v každém kole se stav všech buněk mění současně. Stav buňky vždy záleží na stavu osmi okolních buněk v mřížce, a to následovně. Pokud je buňka živá a má méně než dva živé sousedy, umírá na osamělost, pokud má více než tři živé sousedy, umírá na přehuštění. Živá buňka se dvěma nebo třemi živými sousedy přežívá. Pokud je buňka mrtvá a má právě tři živé sousedy, ožívá, jinak zůstává mrtvá.

Obrázek 8.1 ilustruje klíčové principy hry. První rádek názorně ukazuje aplikaci pravidel – buňky označené tečkou umírají nebo ožívají. Dále je uvedeno několik zajímavých konfigurací. Stabilní konfigurace jsou takové, kde se stav buněk nemění. Oscilující konfigurace se během několika kol dostanou zpět do původního stavu. Pohybující se konfigurace se během několika kol dostanou do původního rozmištění, ovšem posunou se přitom v prostoru. Existují

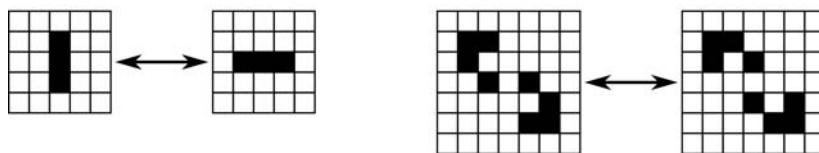
příklad dynamiky



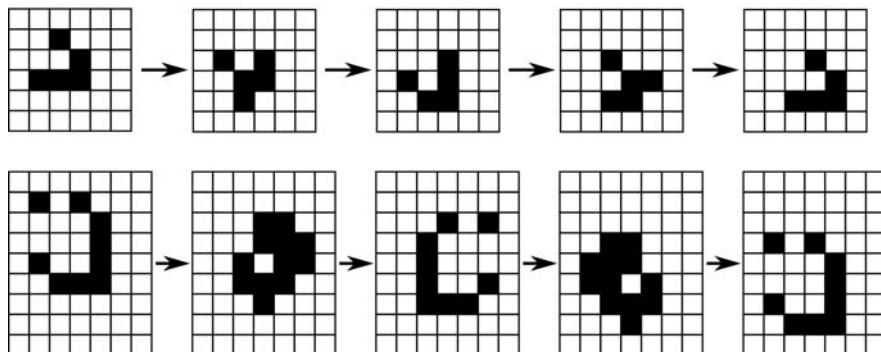
stabilní konfigurace



oscilující konfigurace



pohybující se konfigurace



Obrázek 8.1: Ukázky hry Život

ještě mnohem zajímavější konfigurace, jako je například „kluzákové dělo“: konfigurace, která se během několika kol dostane do původního rozmístění a mezičít „vystřelí“ kluzák, což je jedna z pohybujících se konfigurací.

V teorii se hra studuje na nekonečně velké mřížce. Pro účely simulace však uvažujeme konečnou mřížku – buď zacyklenou (propojíme levý a pravý okraj, horní a dolní okraj) nebo mřížku s fixními okraji (buňky na okraji jsou vždy mrtvé). Na konečné mřížce se musí vývoj simulace po určitém počtu kroků zacyklit, vývoj však může trvat velmi dlouho.

Cílem cvičení je napsat program, který umožní hru simulovat a provádět s ní experimenty. Konkrétní náměty na úkoly:

1. Program začne z náhodného počátečního stavu a vykresluje (textově nebo graficky) jednotlivé generace.
2. Program načte počáteční konfiguraci a pak vykresluje simulaci, tj. umožní uživateli vyzkoušet chování konkrétních konfigurací (např. těch z obrázku 8.1).
3. Program vypočítá statistiky průběhu simulace na plánu velikosti  $N \times M$  pro náhodný počáteční stav, např. pravděpodobnost kompletního vyvření všech buněk, průměrná délka simulace před zacyklením, průměrná délka cyklu, průměrný počet živých buněk na plánu.

Dále můžeme zkoušet experimentovat s úpravami hry, kdy změníme pravidla přechodu mezi stavy živá a mrtvá. Většina úprav vede k „triviálním hrám“, ve kterých dochází k rychlé stabilizaci stavu nebo k jednoduchým oscilacím. Můžeme kvantifikovat, co znamená, že pravidla vedou k netriviálnímu chování? Můžeme automaticky hledat „zajímavá pravidla“?

### Doplňující komentář

Autorem pravidel hry je J. H. Conway, hru zpopularizoval v roce 1970 M. Gardner a od té doby je velmi známá a existuje k ní bohatá literatura (základní přehled viz např. Pelánek, 2011b). Na webu lze najít také mnoho interaktivních demonstrací hry nebo videí popisujících zajímavosti o hře – hra má například „Turingovskou sílu“, což znamená, že pomocí ní lze simulovat libovolný výpočet proveditelný na počítači.

Implementace základní simulace přímočaře odpovídá pravidlům hry. Musíme pouze dobře ošetřit okrajové podmínky a nezapomenout, že stav všech buněk se mění současně. Musíme si tedy vždy napočítat nový stav do pomocného pole a teprve pak jej překopírovat, resp. můžeme ušetřit paměť pomocí aktualizace po řádcích, což je mimochodem užitečné programátorské cvičení.

### 8.3 Problémy s řetězci a posloupnostmi

<b>Nápad:</b>	3-4
<b>Kódování:</b>	2
<b>Styl úlohy:</b>	<i>Cvičení na procvičení návrhu algoritmů, především pomocí dynamického programování. Algoriticky docela náročné, kódování je (při zvolení vhodného algoritmu) jednoduché.</i>

Na vstupu máme posloupnost čísel nebo dva řetězce a máme najít optimální „podčást“ (tabulka 8.2 uvádí konkrétní příklady vstupů a odpovídajících výstupů):

1. Nejdelší rostoucí podsekvence (spojité sekvence) posloupnosti čísel.
2. Nejdelší rostoucí podposloupnosti (musí zachovat pořadí, ale nemusí být nutně spojitá) posloupnosti čísel.
3. Nejdelší společný (souvislý) podřetězec dvou řetězců.
4. Nejdelší společná podposloupnost (musí zachovat pořadí, ale nemusí být nutně spojitá) dvou řetězců.

Rozšířením posledního uvedeného problému je hledání minimální editační vzdálenosti dvou řetězců (problém je znám pod názvem Levenshteinova vzdálenost). Máme k dispozici následující tři editační operace: záměna znaku, smazání libovolného znaku, vložení znaku na libovolnou pozici. Na vstupu máme dva řetězce a úkolem je najít nejkratší posloupnost editačních změn, která převede jeden řetězec na druhý. Například řetězce *ostrov* a *saturn* mají editační vzdálenost 5, odpovídající editační posloupnost je *ostrov* → *strov* → *satrov* → *saturov* → *saturnv* → *saturn* (2 smazání, 2 vložení, 1 nahrazení).

**Tabulka 8.2:** Řetězce a posloupnosti – příklady

	<b>Vstup</b>	<b>Výstup</b>
1. Podsekvence	2, 5, 1, 4, 3, 3, 4, 7, 6, 9, 2, 3	3, 4, 7
2. Podposloupnost	2, 5, 1, 4, 3, 3, 4, 7, 6, 9, 2, 3	1, 3, 4, 6, 9
3. Podřetězec	pampeliska, sedmikraska	ska
4. Podposloupnost	pampeliska, sedmikraska	miska

### Doplňující komentář

Uvedené problémy, především ty s řetězci, mají hodně praktických aplikací, například pro hledání rozdílů mezi dvěma verzemi souboru, pro automatizované opravování překlepů nebo v bioinformatice při hledání vzorů v genomu (genom můžeme vnímat jako řetězec písmen GACT).

Přestože úlohy s číselnými posloupnostmi a s řetězci mají odlišný typ vstupů a výstupů, zde jsou spojeny v jednom cvičení, protože se řeší podobným způsobem. Jejich společné zpracování v rámci jednoho cvičení je užitečné. Sice opakovaně programujeme stejný algoritmus, ale to je právě vhodné k tomu, abychom si dobře zažili použité principy, které jsou užitečné a ne zcela snadné na pořádné pochopení.

Hledání podsekvencí (respektive podřetězců) je přímočaré a představuje jen rozsvíčku. Podposloupnosti, které nemusí být spojité, už jsou zajímavé. Navící algoritmus by spočíval ve využití hrubé síly, tj. vyzkoušení všech možných podposloupností, kterých je exponenciálně mnoho. Za využití dynamického programování lze problém řešit výrazně efektivněji – s kvadratickou složitostí. Ilustrujme základní myšlenku na hledání nejdelší rostoucí podposloupnosti. Vytvoříme tabulku, do které pro každý člen posloupnosti ukládáme délku nejdelší rostoucí podposloupnosti končící tímto prvkem. Pro ukázkový příklad tato tabulka vypadá následovně:

vstupní posloupnost	2	5	1	4	3	3	4	7	6	9	2	3
délka nejdelší podposloupnosti	1	1	1	2	2	2	3	4	4	5	2	3

S trohou rozmyšlení lze tuto tabulku efektivně zkonstruovat jedním průchodem zleva doprava. Za využití tabulky pak již můžeme poměrně snadno získat odpověď na zadaný problém. Hledání společné podposloupnosti dvou řetězců a editační vzdálenosti se řeší podobným stylem, pouze používáme dvojrozměrnou tabulku.

## 8.4 Experimenty s řadicími algoritmy

**Nápad:**

2-3

**Kódování:**

2-4

**Styl úlohy:**

Experimentální vyhodnocení klasických algoritmů a zpracování výsledků.

Řadicí algoritmy jsou asi nejtradičnejší téma ve výuce informatiky. Předpokládáme, že čtenář už o tomto tématu slyšel, takže se nebudeme soustředit na

rozbor dílčích algoritmů. Výklad řadicích algoritmů lze najít v mnoha učebnicích, i na Internetu se nachází mnoho dobrých vysvětlení včetně názorných ilustrací, dobrým zdrojem je např. stránka [www.sorting-algorithms.com/](http://www.sorting-algorithms.com/).

V rámci cvičení se budeme soustředit na experimentální vyhodnocení. Řadicí algoritmy jsou na vyzkoušení experimentálního vyhodnocení ideální. Jsou jednoduché na implementaci a provedení experimentů a současně na nich lze ilustrovat mnoho zajímavých prvků, například vliv složitosti algoritmu, vliv typu vstupních dat nebo metodiku měření délky výpočtu.

Řadicích algoritmů existuje mnoho. Pro účely cvičení se stačí zaměřit jen na řazení posloupností přirozených čísel a na následující nejznámější algoritmy:

- jednoduché algoritmy s kvadratickou složitostí: bublinkové řazení (bubble sort), řazení vkládáním (insert sort), řazení výběrem (select sort).
- efektivní algoritmy se složitostí  $O(n \log(n))$ : quick sort, řazení slučováním (merge sort), řazení haldou (heap sort),
- radix sort – lineární řazení pomocí procházení jednotlivých číslic.

Cílem cvičení je implementovat několik algoritmů, vygenerovat rozsáhlou sadu testovacích dat a potom automatizovaně spustit algoritmy na testovacích datech, zaznamenávat výsledky, výsledky graficky zpracovat a interpretovat. Je více možností, na co se můžeme při experimentech zaměřit. Raději než zkusit povrchně vše, může být lepší si vybrat jeden konkrétní námět a ten zpracovat důkladně do hloubky.

Základní srovnání se týká rychlosti různých algoritmů. Dále se můžeme zabývat porovnáním různých implementací jednoho algoritmu, například srovnání implementace v různých programovacích jazycích, srovnání vlastní implementace se standardní implementací dostupnou v programovacím jazyce, zkoumání vlivu použitých datových struktur nebo vlivu použití optimalizací při komplikaci.

Při experimentech věnujeme důkladnou pozornost použitým testovacím datům. Určitě použijeme posloupnosti různé délky, a to dostatečného rozsahu, aby se projevily rozdíly mezi jednotlivými algoritmy. Vhodné je také vytvořit posloupnosti různých typů, například náhodná čísla, téměř seřazená čísla s pouze pár prohozenými hodnotami, inverzně seřazená čísla.

### Doplňující komentář

Délku běhu algoritmu můžeme měřit prostě pomocí reálného času nebo jako počet provedených operací – u řadicích algoritmů je přirozeným kritériem počítat počet porovnání dvou prvků. Případně můžeme měřit oba parametry a vyhodnotit, jak spolu souvisejí. Pokud měříme čas výpočtu, musíme si

ujasnit, odkdy dokdy čas přesně měříme. Vhodné je měřit čas čistě na vlastní výpočet, tj. nezahrnovat čas na načítání vstupu či vypisování výstupu. Hlavně je však klíčové měřit čas konzistentně. Nesmí se stát, že jednomu algoritmu započítáme čas na načítání vstupu a druhému ne.

U měření reálného času je důležité výpočty spouštět opakováně – díky multitaskingu, který operační systém provádí, je délka výpočtu závislá na náhodných faktorech, například na tom, jaké další programy na počítači běží. Opakováním spuštěním výpočtu a zprůměrováním výsledků zmenšíme roli tohoto náhodného faktoru. Podobně je důležité vygenerovat více vstupů stejné délky a výsledky opět zprůměrovat.

Vliv různých typů vstupních posloupností se u jednotlivých algoritmů liší. Typicky například řazení vkládáním je pomalé na náhodném vstupu, ale velmi rychlé na téměř seřazeném vstupu, kdežto na řazení výběrem nemá typ vstupu vliv. Pro různé typy vstupů tedy udržujeme výsledky odděleně a uděláme pro ně samostatné grafy.

## 8.5 Vyhodnocování výrazů

**Nápad:** 3-4

**Kódování:** 2-5

**Styl úlohy:** *Jeden z klasických problémů v informatice, ke kterému jde přistupovat mnoha různými způsoby.*

Základní zadání spočívá ve vyhodnocení zadaného výrazu. Obtížnost cvičení můžeme odstupňovat podle přesných požadavků na program (viz ukázky v tabulce 8.3):

1. Základní verze. Program pracuje pouze s přirozenými čísly, čtyřmi základními operaci a závorkami a předpokládá, že vstup je dobře formátovaný. Program vrátí hodnotu tohoto výrazu, přičemž dodržuje standardní priority operátorů a závorkování.
2. Kontrola vstupu. Program kontroluje, zda je vstup správně formátovaný, a pokud není, vrátí rozumnou chybovou hlášku určující polohu chyby.
3. Rozšířený vstup. Oproti základní verzi povolíme použití dalších operátorů (např. umocňování, unární minus), desetinných čísel, standardních konstant a funkcí (e, pi, sin, log).
4. Interaktivní kalkulačka. Program vyhodnocuje výrazy opakováně, přičemž umožňuje ukládat do proměnných a pak s nimi počítat.

Kromě základního vyhodnocování výrazů pak můžeme zkusit úlohy, které výrazy upravují nebo konstruují. V úloze „maximalizace výrazu“ dostaneme

na vstup výraz bez závorek a máme najít takové uzávorkování, které maximalizuje hodnotu výrazu. V jednodušší verzi problému uvažujeme pouze přirozená čísla a operace sčítání a násobení, v složitější verzi pak i odčítání a dělení.

Dále můžeme uvážit úlohy typu „Pět pětek“, kdy pomocí výrazu obsahujícího jen opakování výskyty jedné cifry máme za úkol „vyrobit“ zadané číslo (viz příklad v tabulce 8.3). Program dostane na vstupu, kolik jakých cifer má k dispozici a cílové číslo, na výstup vypíše výraz, jehož hodnota je rovna zadanému číslu.

**Tabulka 8.3:** Vyhodnocování výrazů – ilustrace zadání

	Vstup	Výstup
Základní verze	$3-4/2+(6-2)*5$	21
Kontrola vstupu	$3+(/6-2)*5$	chyba: 4. pozice
Rozšířený vstup	$\sin(-1.5*\pi)+\log(4)$	2.386
Kalkulačka	$x=3*4+2$ $x/2$ $y=(x-5)/3$	$x=14$ 7 $y=3$
Maximalizace výrazu	$2*1+4*1+3$	$(2*(1+4)*(1+3))$
Pět pětek	12	$55/5 + 5/5$

### Doplňující komentář

Vyhodnocování výrazů je takovou základní ochutnávkou z rozsáhlé oblasti syntaktické analýzy a překladačů. Protože jde o velmi praktickou a hojně prostudovanou oblast, existuje pro ni spousta specializovaných nástrojů (např. Lex, Yac, ANTLR), které usnadňují vývoj. S tématem také souvisí oblast formálních gramatik a automatů. Pro účely našeho cvičení, které se zabývá základními úkoly z oblasti syntaktické analýzy, však vystačíme bez použití těchto speciálních pojmu a nástrojů.

I bez použití pokročilých nástrojů lze úlohu řešit několika způsoby. Prvním z nich je „odfláknutí“ úlohy. Pokud používáme interpretovaný jazyk (např. Python, Perl, PHP), máme k dispozici funkci `eval`, která umí vyhodnotit zadaný řetězec. Funkce `eval` typicky vyhodnotí jakýkoliv platný výraz v daném jazyce, což zahrnuje i matematické výrazy. S využitím této funkce tedy můžeme úlohu snadno „vyřešit“, má to však dvě nevýhody: nic se nenaučíme a je to nebezpečné. Pokud bychom chtěli používat tento přístup v reálné aplikaci, mohlo by se stát, že záludný uživatel dá na vstup nejen matematický výraz, ale

i zákeřnější kus kódu, jehož naivní vyhodnocení pomocí funkce `eval` povede k problémům.

Zkusme tedy úlohu vyřešit poctivě. Přímočarý přístup k vyhodnocování výrazů je pomocí metody `rozděl` a panuj – najdeme „vhodný“ operátor, podle něj rozdělíme vstupní řetězec na dvě poloviny, na ty rekurzivně zavoláme vyhodnocování výrazů a výsledky jen zkombinujeme, tj. provedeme operaci odpovídající danému operátoru. Zbývá jen domyslet, jak správně vybírat „vhodný“ operátor.

Jinou možností vyhodnocení je převést výraz do postfixové notace (též zvaná reverzní polská notace). Klasický zápis výrazu je v infixové notaci – to znamená, že operátory jsou mezi čísly. V postfixové notaci jsou operátory za čísly, tj. místo „ $1 + 2$ “ píšeme „ $1\ 2\ +$ “. Postfixová notace je pro lidi těžko čitelná, ale má tu výhodu, že výrazy se v ní snadno vyhodnocují, mimo jiné proto, že nepotřebujeme závorky. Například první výraz v tabulce 8.3 se v postfixové notaci zapíše:  $3\ 4\ 2\ / - 6\ 2\ - 5\ * +$ .

Vyhodnocení klasického výrazu tak můžeme provést na dva kroky – nejdříve výraz převedeme do postfixové notace a potom jej vyhodnotíme. Oba dva tyto kroky lze provést na jeden průchod řetězce, tj. celková časová složitost je lineární. Naproti tomu výše uvedený postup pomocí metody `rozděl` a panuj vede ke kvadratické složitosti.

## 8.6 Grafové algoritmy

**Nápad:**

3-5

**Kódování:**

3-5

**Styl úlohy:**

*Procvičení a experimentální vyhodnocení základních grafových algoritmů.*

Grafy jsou jednou ze základních struktur v informatice a mnoho problémů lze řešit transformací na grafový problém. Mezi takové problémy patří například logistické problémy, jako je hledání nejkratší cesty mezi městy, analýza sociálních sítí, směrování dat v počítačových sítích nebo třeba výše uvedené hledání cest v bludišti a generování bludišť. Oproti jiným příkladům uvedeným v této knize toto cvičení možná není až tak atraktivní, nicméně je velmi užitečné.

Cvičení spočívá v implementaci algoritmů a jejich experimentálním vyhodnocení. Pro experimenty potřebujeme velké grafy, na kterých budeme algoritmy spouštět. Máme několik možností, jak grafy získat:

1. Náhodné grafy. Zvolíme počet vrcholů a hran, každou hranu přidáme mezi dva náhodně zvolené vrcholy.

2. Pravidelné grafy. Vytvoříme například různé  $n$ -rozměrné krychle, pravidelné mřížky, lineárně spojené kliky.
3. Grafy odpovídající reálným problémům. Na webu lze najít mnoho sad reálných grafů z různých oblastí, mezi známé sady patří například „The Stanford GraphBase“.

Pro experimenty může být užitečné použít grafy všech tří typů a porovnat výsledky nad jednotlivými typy grafů. Můžeme se například pokusit odpovědět na otázku „Který z uvedených typů grafů má nejkratší průměrnou vzdálenost mezi vrcholy?“. Můžeme také naimplementovat několik algoritmů pro stejný problém a srovnat je. Náměty na vhodné grafové problémy a algoritmy:

1. Základní prohledávání grafu do šířky a do hloubky. Prohledávání do hloubky implementovat jak rekursivně, tak bez použití rekurze.
2. Využití prohledávání grafu pro zjištění počtu komponent souvislosti, test bipartity grafu nebo detekci přítomnosti cyklu v orientovaném grafu.
3. Různé typy hledání nejkratších cest:
  - z jednoho vrcholu do všech ostatních, kladné hrany – Dijkstrův algoritmus,
  - z jednoho vrcholu do všech ostatních, celočíselné hrany – Bellman-Fordův algoritmus,
  - mezi všemi páry vrcholů – Floyd-Warshallův algoritmus.
4. Minimální kostra grafu: Primův a Kruskalův algoritmus.
5. Silně souvislé komponenty: Tarjanův algoritmus.
6. Hledání eulerovské cesty, tj. cesty, která prochází každou hranou právě jednou.
7. Hledání hamiltonovské cesty, tj. cesty, která prochází každým vrcholem právě jednou.

### Doplňující komentář

Jednotlivé pojmy zde nebudeme rozebírat, jde o klasické problémy a algoritmy, jejichž popis lze nalézt v mnoha učebnicích. Jen upozorníme na jeden záludný detail – pro hledání eulerovské cesty existuje efektivní algoritmus, pro hledání hamiltonovské cesty efektivní algoritmus není znám a není pravděpodobné, že by existoval (problém je NP-úplný). V případě hamiltonovské cesty je tedy možné problém řešit jen pro malé grafy.

Trochu podrobněji rozebereme jen jedno cvičení – nerekurzivní implementace prohledávání do hloubky. Toto cvičení je velmi užitečné, protože nás nutí dobré si ujasnit, jak funguje rekurze. Krom toho je i prakticky významné, protože prohledávání do hloubky se často používá jako dílčí krok v mnoha grafových algoritmech a při práci s rozsáhlými grafy začne být rekurzivní

algoritmus z technických důvodů nevhodný. Rekurzivní implementace prohledávání do hloubky je přímočará:

```
def DFS(v) :  
    navstiven[v] = True  
    for w in naslednici[v] :  
        if not navstiven[w] : DFS(w)
```

Nerekurzivní implementace však může člověka snadno zaskočit. Většina informatiků ví, že „prohledávání do hloubky se implementuje pomocí zásobníku“. Nicméně většina programátorů bojuje s tím, aby nerekurzivní implementaci udělali korektně a efektivně. Nezdá se vám to obtížné? Zkuste to a pak dobře zkонтrolujte, zda pořadí procházení vrcholů je opravdu stejné jako u rekurzivní varianty a zda je zachována efektivita rekurzivního algoritmu, tj. zda každá hrana je procházena právě jednou a přidané paměťové nároky jsou lineární vůči počtu vrcholů. V řešení je uvedeno správné řešení i typická špatná řešení.



# 9 Další náměty

*Je lepší znát některé z otázek než všechny odpovědi.*

J. Thurber

Tato kapitola nabízí pestrou směsici námětů z různých oblastí. Jsou zde zmíněny opravdu jen hrubé náměty, zadání jsou popsána buď jen stručně nebo hodně obecně. Konkrétní zadání si tedy musíte domyslet buď za využití vlastní invence nebo vyhledáním zmíněných pojmu v literatuře. Komentáře k postupu a řešení nejsou pro tyto náměty uvedeny vůbec.

## 9.1 Generování a transformace obrázků

Generováním obrázků jsme se zabývali v kapitole 4. Zde uvedeme několik dalších námětů podobného typu. Zajímavé příklady nabízí přístup označovaný „Media Computation“, který se zaměřuje především na úvodní výuku programování pomocí médií. Konkrétně tento přístup využívá třeba bitmapové obrázky a jejich transformace. Transformace obrázků můžeme použít na jednoduchá programátorská cvičení na procičení základních programátorských konstrukcí, jako jsou cykly (přebarvení obrázku do červena, zrcadlové převrácení obrazu), ale i na zajímavé algoritmy (detekce hran, kombinace dvou obrázků do jednoho). Více námětů a konkrétní podklady lze najít na webu při hledání pojmu „Media Computation“.

Na procičení programování a současně geometrie se hodí úlohy s vykreslováním „dláždění“ roviny, což je rozdelení roviny na pravidelné obrazce. Můžeme začít jednoduchým pokrytím roviny čtverci, šestiúhelníky či trojúhelníky, pokračovat kombinacemi dvou pravidelných mnohoúhelníků (například osmiúhelníky a čtverce), až po aperiodická dláždění, jako je Penrosovo. Mnoho inspirace lze najít na webu při vyhledání pojmu „tiling“. Doporučený výstupní formát je opět SVG. Tento formát má mimochodem v sobě určité „programátorské“ možnosti, které se mohou pro vytváření dláždění hodit.

Řadu zajímavých námětů nabízí téma fraktálů, kterým jsme se zabývali již u několika cvičení v kapitole 4. Další inspiraci pro programátorská cvičení mohou nabídnout například následující známé fraktály: Apollónova síť, Cantorova množina, Cantorův prach, Dračí křivka, Feigenbaumův diagram, Lorenzův atraktor, Peanova křivka, Sierpińského koberec.

## 9.2 Blízké body

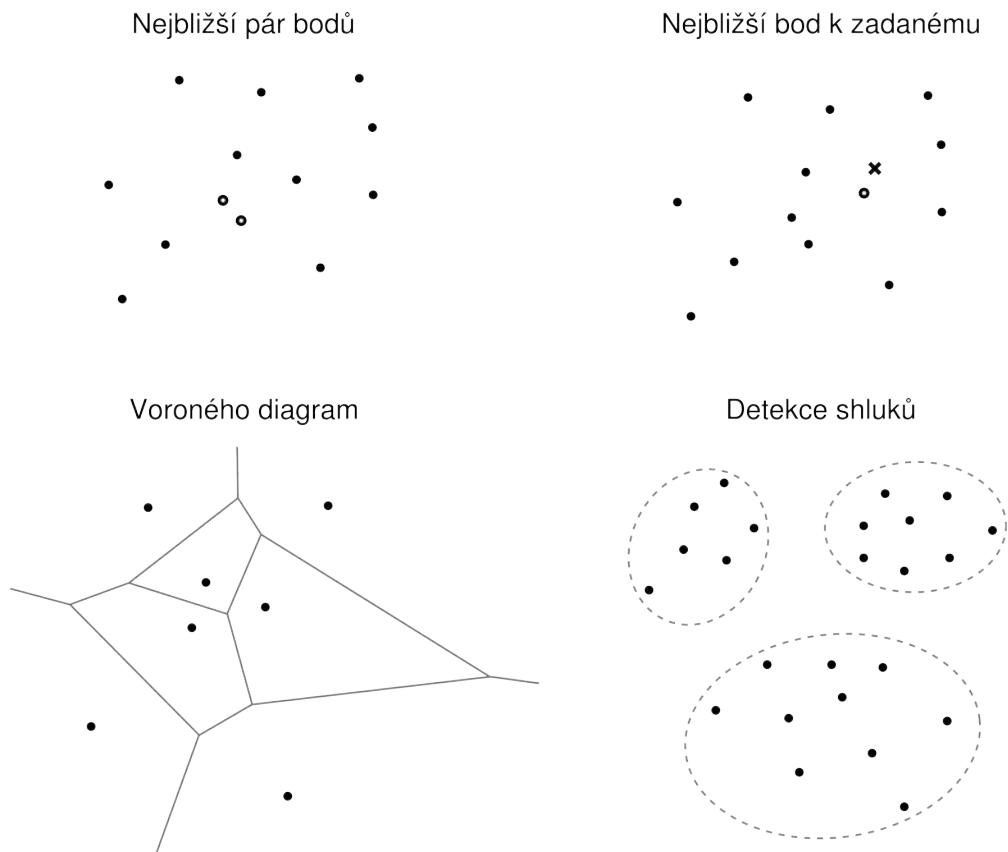
V následujících úlohách máme vždy jako vstup množinu bodů a hledáme nějakým způsobem „blízké body“. Problemy lze uvažovat v  $n$  rozměrném prostoru, ale pro základní cvičení je vhodné pracovat s dvojrozměrným prostorem a výsledek si pro kontrolu vykreslit graficky. Úlohy jsou algoritmicky docela náročné a pro jejich řešení je vhodné si nastudovat relevantní pojmy a algoritmy z literatury. Pokud však netrváme na optimální efektivitě algoritmů nebo si náměty zjednodušíme na nějaké speciální případy, je určitě možné je využít i jako náměty na jednoduchá programátorská cvičení. Úlohy jsou ilustrovány na obrázku 9.1.

1. Nejbližší pár bodů v zadané množině. Jednoduše lze řešit vyzkoušením všech možností, což má kvadratickou složitost  $O(n^2)$ , existuje však efektivnější řešení o složitosti  $O(n \log(n))$ . Základní přístup spočívá v typickém využití metody rozděl a panuj, domyšlení přesných detailů však vyžaduje netriviální matematický vhled.

2. Hledání nejbližšího bodu k jednomu zadanému. Pokud máme problém řešit jednorázově, z principu potřebujeme alespoň lineární čas na načtení vstupu, takže přímočaré řešení je optimální. Pokud však máme opakováne dotazy, lze dosáhnout logaritmické složitosti za využití datové struktury „kD strom“.

3. Konstrukce Voroného diagramu. Pro každý bod určíme oblast prostoru, pro kterou je tento bod ze všech bodů nejbližší. Voroného diagram souvisí s Delaunayovou triangulací, která byla zmíněna v části 4.8.

4. Detekce shluků blízkých bodů. Chceme zadanou množinu bodů rozdělit na skupiny, aby v rámci každé skupiny byly body co nejbližše k sobě a skupiny byly od sebe vzájemně co nejdále. Tento problém se často vyskytuje při statistické analýze dat, k řešení se využívá „k-means“ algoritmus.

**Obrázek 9.1:** Blízké body – příklady

### 9.3 Akční hry

Hry popsané v kapitole 7 byly „intelektuální“ – vesměs šlo o původem deskové hry, ve kterých jde primárně o přemýšlení. V mnoha počítačových hrách je ale primární rychlosť a postřeh. Na rozdíl od ostatních námětů je programování akčních her specifické pro konkrétní prostředí. Minimálně je nezbytné využít grafické knihovny, existují také rozsáhlá specializovaná prostředí pro vývoj her, a to i volně dostupná, jako například `pygame` pro Python. Proto nemá cenu tento typ her na obecné úrovni příliš rozebírat, zmíníme jen některé klasické náměty na jednoduché hry: Červi, Tanky, Pac man a další souboje v dvojrozměrném bludišti, „plošinovky“. Na webu lze snadno najít mnoho jednoduchých her, které mohou posloužit jako inspirace na rozumně náročnou úlohu.

Aby byla úloha zajímavá, je vhodné vložit do pravidel hry trochu vlastní invence a vybrat si hru s prostorem pro vývoj „umělé inteligence“, kterou počítač využívá při hře proti hráči. Typický příklad jednoduché umělé inteligence je navigace příšer v bludišti za použití A\* algoritmu.

## 9.4 Implementace datových struktur

Implementace datových struktur je klasický problém, který je rozebrán ve většině učebnic algoritmizace a programování. Základní datové struktury vhodné pro procvičení a důkladné pochopení jsou následující:

- lineární zřetězený seznam (jednostranně, oboustranně),
- fronta, zásobník (implementace pomocí pole, zřetězeného seznamu),
- prioritní fronta,
- binární vyhledávací stromy: základní, vyvážené (např. červeno-černý, AVL),
- halda,
- hašovací tabulka.

Podobně jako u řadicích algoritmů je zde vhodné spojit implementaci s experimentálním vyhodnocením implementací, konkrétně například můžeme srovnat vhodnost použití hašovací tabulky a binárního vyhledávacího stromu pro reprezentaci velké množiny čísel.

## 9.5 Implementace matematických operací

Úkolem je implementovat vybrané operace z matematiky, což je výborný způsob, jak si pořádně ujasnit fungování příslušných operací. Většina matematických operací se při programování často používá, takže jejich implementace jsou typicky dostupné v různých knihovnách. Účelem tohoto cvičení pochopitelně není převzít implementaci z knihovny, ale napsat si vlastní „na zelené louce“. Uvádíme jen stručný seznam námětů (předpokládá se znalost pojmu nebo jejich dohledání):

- matice, vektory a operace nad nimi (násobení, sčítání, inverze matice, výpočet determinantu, výpočet vlastních čísel),
- využití matic pro reprezentaci geometrických transformací v dvojrozměrném prostoru (např. rotace, změna velikosti, překlopení), generování obrázků pomocí implementovaných operací,
- geometrické problémy s mnohoúhelníky: test konvexnosti, výpočet obsahu, výpočet těžiště,

- detekce kolizí, průsečíku či vzájemné polohy různých objektů (bod, úsečka, mnohoúhelník, kružnice),
- řešení systému lineárních rovnic,
- numerické metody pro hledání kořenů funkce a řešení systému rovnic,
- derivování a integrování zadané funkce (symbolicky),
- základní popisná statistika: pro daný vzorek dat počítání průměru, mediánu, směrodatné odchylky, korelačních koeficientů, grafické znázornění dat (histogram, bodový graf),
- centrální limitní věta – praktické experimenty pomocí generátoru náhodných čísel.

## 9.6 Zpracování a analýza reálných dat

Úkolem je získat a analyzovat reálná data, například z následujících kategorií: demografická data, kurzy měn, ceny surovin, data o dopravě (např. hledání spojení, zpoždění vlaků), encyklopedická data (např. z Wikipedie). Jako inspirace může posloužit například služba Google Public Data, která umožňuje vizualizovat mnoho veřejně dostupných dat a nabízí odkazy na zdroje dat. Data z českých krajin nabízí Český statistický úřad. S tímto tématem souvisí iniciativa „Open data“ – snaha, aby státní instituce zveřejňovaly co nejvíce dat ve strojově čitelné podobě. U nás to zatím bohužel příliš nefunguje, ale pro inspiraci je zajímavé se podívat na americký a britský portál: [data.gov](http://data.gov), [data.gov.uk](http://data.gov.uk).

Data můžeme získat nejen přímo od některé z institucí, ale také vlastním stahováním z webu, například stažením vzorku stránek z Wikipedie. Jako další krok musíme data zpracovat, tedy vytáhnout z nich jen to, co nás pro analýzy zajímá. Pak můžeme přistoupit k vlastnímu statistickému zpracování dat, výsledky analýz je vhodné ztvárnit graficky. Můžeme také zkoušet dělat automatizované předpovědi budoucího vývoje (např. počasí, cena ropy) a vyhodnocovat jejich úspěšnost.

## 9.7 Interpret jednoduchého programovacího jazyka

Ve cvičení 8.5 jsme se zabývali vyhodnocováním výrazů, teď se podíváme na vyhodnocování celých programů. Syntaktická analýza, interpretace či komplikace jsou velmi rozsáhlé oblasti informatiky, takže v rámci základního programátorského cvičení lze zvládnout jen základy. Pro účel cvičení je tedy vhodné vybrat si velmi jednoduchý programovací jazyk.

Vhodné jsou „jazyky pro virtuální roboty“. Příkladem takového jazyka je želví grafika, popsaná ve cvičení 4.2, nebo úlohy Robot Karel či Robotanik,

které si můžete vyzkoušet v Tutor systému ([tutor.fi.muni.cz](http://tutor.fi.muni.cz)). V těchto úlohách programujeme robota pro pohyb ve čtvercové mřížce pomocí elementárních příkazů pro pohyb (krok dopředu, otočení o 90 stupňů) a s využitím jednoduchých podmiňovacích podmínek, cyklů a volání funkcí. Tyto jazyky mají typicky velmi omezenou syntax, a jsou tudíž jednoduché na analýzu. Nicméně důležitou roli v nich hraje rekurze, takže interpretace programů není triviální a cvičení dává užitečný vhled do obecnějších problémů, které souvisejí s interpretací a komplikací programů. Musíme se například zamyslet nad reprezentací stavu programu, využitím zásobníku a souvislostí mezi zásobníkem a rekurzí.

Další možností je vzít klasický programovací jazyk a „ořezat“ jej jen na několik základních příkazů, velmi hrubé ořezání se označuje jako „while programy“. Jinou alternativou je zvolit některý z ezoterických programovacích jazyků, což jsou jazyky, které byly navrženy nikoliv pro praktické programování, ale jako nadsázka nebo vtip. Nejznámější z nich je pravděpodobně „Brainfuck“, ve kterém se k programování používá pouze následujících 8 znaků: „><+-., []“.

Pro realizaci interpretu je řada různých možností, záleží na tom, jak složitý si zvolíme vstupní jazyk a co chceme potrénovat. Pokud chceme cvičení vzít opravdu vážně, můžeme použít sofistikované nástroje, jako je např. Bison, Yacc nebo ANTLR. Zajímavou cestou může být použití funkcionálních jazyků, jako je OCaml, ve kterých jde interpretace programu často napsat stručně a přirozeně díky dobré podpoře hledání vzorů a funkcionálnímu zápisu. Rychlý, ale nepříliš koncepční interpret lze pro jednoduché vstupní jazyky implementovat v interpretovaných jazycích jako Python, Perl či JavaScript za použití regulárních výrazů a funkce `eval()`.

# 10 Vybraná řešení

*Ladění je dvakrát tak náročné jako psaní vlastního kódu. Takže pokud napišete kód tak chytře, jak jen jste schopni, pak nejste dostatečně schopní na to, abyste jej odladili.*

B. W. Kernighan

Na závěr nabízíme vybraná řešení k popsaným úlohám. Jde opravdu jen o „vybraná řešení“, nenabízíme kompletní vysvětlení všech variant zadání. To by ani nebylo dobré, protože většina lidí má slabou vůli a místo samostatného řešení se rovnou dívá na výsledky, čímž se toho ovšem naučí mnohem méně, než když se s úlohou poperou sami. Řešení jsou uvedena tedy pouze do takové míry, kdy pomáhají osvětlit zajímavé myšlenky, obecně použitelné prvky nebo záladné nápady, na které je těžké přijít.

Míra pokrytí a detailu se záměrně u jednotlivých úloh liší. Některé úlohy jsou rozebrány detailně, u jiných je jen stručný komentář a necháváme je otevřené. U jednoduchých úloh je často uváděn i konkrétní zdrojový kód, u složitějších úloh většinou jen hlavní myšlenky řešení. Předpokládá se, že ten, kdo má ambici vyřešit složitější úlohy, si s kódováním dílčích kroků zvládne poradit sám.

V některých případech jsou uvedeny i konkrétní zdrojové kódy, pro zápis je použit jazyk Python, případně kombinace jazyka Python a přirozeného jazyka. Uvedené části kódu by však měly být pochopitelné pro kohokoliv, kdo má základní programovací zkušenosti. Některá řešení je možné zapsat v Pythonu kompaktněji, ale záměrně je zvolen zápis, který je čitelnější a bez specifických prvků Pythonu.

Dále uvedené části kódu je nutno chápat jako pouze ilustrační. Důraz je kladen na předání hlavní myšlenky ve stručné podobě, nikoliv na to, aby kód šlo „opsat a spustit“. Často například chybí inicializace datových struktur nebo kód dílčích funkcí. Z kontextu by však mělo být zřejmé, jak tyto chybějící části doplnit.

## 10.1 Počítání s číslы

### Hrátky s číslы

Jak je uvedeno v komentáři, u tohoto cvičení je dobré zkusit si zapsat řešení několika různými způsoby. Například součet prvních  $n$  čísel můžeme snadno zapsat pomocí `for` cyklu, `while` cyklu, rekurze, explicitního vztahu nebo za využití vestavěných funkcí daného jazyka. V jazyce Python to vypadá následovně.

```
def soucet_for(n):
    s = 0
    for i in range(1,n+1):
        s = s + i
    return s

def soucet_while(n):
    s = 0
    while n > 0:
        s = s + n
        n = n - 1
    return s

def soucet_rekurze(n):
    if n > 0: return n + soucet_rekurze(n-1)
    else: return 0

def soucet_explicitne(n):
    return n * (n+1) / 2

def soucet_vestavene_funkce(n):
    return sum(range(1,n+1))
```

Dále uvedeme řešení pro ciferný součet, které ilustruje základní princip, jak dostat informaci o jednotlivých cífrách:

```
def ciferny_soucet(x):
    c = 0
    while x > 0:
        c += x % 10
        x = x / 10
    return c
```

## Posloupnosti

Většina řešení je přímočará, uvedeme kód pouze pro poslední případ – Golombovu sebepopisující posloupnost.

```
def golomb(n):
    seznam = [ 1, 2, 2 ]
    for i in range(1, n+1):
        print seznam[i-1],
        if i > 2 and len(seznam) < n:
            for j in range(seznam[i-1]):
                seznam.append(i)
```

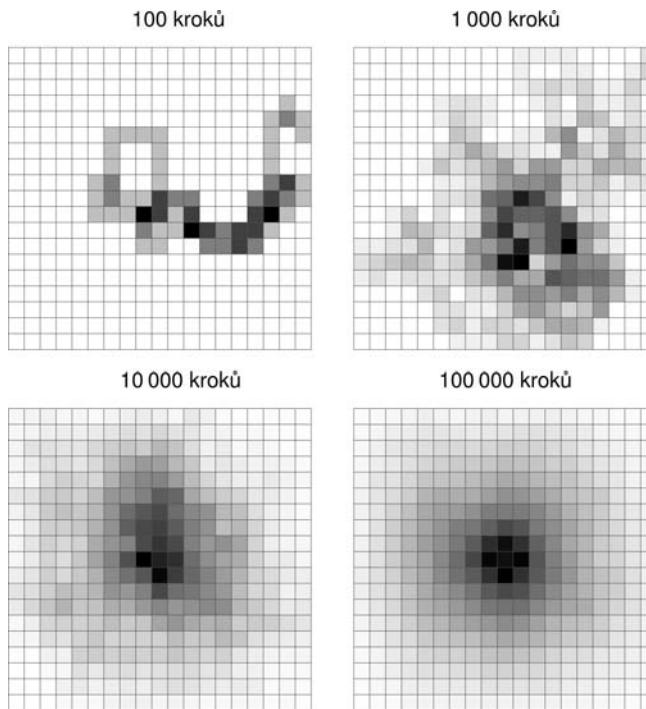
## Collatzův problém

Programy pro řešení jednotlivých zadání jsou přímočaré, takže je nerozebíráme, pouze pro kontrolu uvádíme odpovědi na konkrétní otázky ze zadání:

- Z čísel od 1 do 100 000 potřebujeme nejvíše 350 kroků, tohoto rekordu dosáhneme, když začneme v čísle 77 031 a nejvyšší číslo v odpovídající posloupnosti je 21 933 016.
- Rekordmani do 100 000 jsou následující: 1, 2, 3, 6, 7, 9, 18, 25, 27, 54, 73, 97, 129, 171, 231, 313, 327, 649, 703, 871, 1161, 2223, 2463, 2919, 3711, 6171, 10971, 13255, 17647, 23529, 26623, 34239, 35655, 52527, 77031.
- Pro pravidlo  $3n - 1$  dojde k zacyklení, pokud začneme v čísle 5. Dostaneme posloupnost 5, 14, 7, 20, 5, 14, 7, ...

## Náhodná procházka

Programy jsou relativně přímočaré, pro ilustraci uvádíme pouze příklad vizualizace náhodné procházky ve 2D, viz obrázek 10.1. Jsou uvedeny obrázky pro procházky různé délky, ve všech případech je použito pravidlo „pokud narazíme na okraj, vracíme se do středu“.



**Obrázek 10.1:** Náhodná procházka ve 2D – odstín šedé odpovídá počtu návštěv jednotlivých polí

## Dělitelnost a prvočísla

Euclidův algoritmus lze kompaktně zapsat například následujícími způsoby:

```
def NSD_Euclid1(a,b):
    while b != 0:
        c = b
        b = a % b
        a = c
    return a

def NSD_Euclid2(a,b):
    if b == 0:
        return a
    else:
        return NSD_Euclid2(b, a % b)
```

## Reprezentace čísel

Uvedeme řešení pouze pro základní problém – převod čísla z desítkové soustavy na binární zápis:

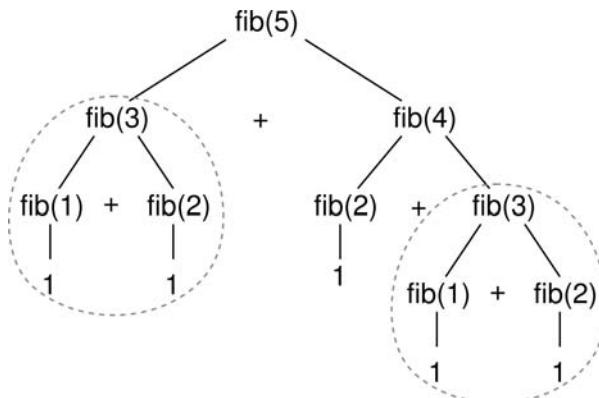
```
def binarni_zapis(n):
    vystup = ''
    while n > 0:
        if n % 2 == 0: vystup = '0' + vystup
        else:           vystup = '1' + vystup
        n = n / 2
    return vystup
```

## Fibonacciho posloupnost

Výpočet  $n$ -tého Fibonacciho čísla lze zapsat rekurzivně přímočaře:

```
def fib(n):
    if n <= 2: return 1
    else: return fib(n-1) + fib(n-2)
```

Tento přístup sice dá korektní výsledky, ale je velmi neefektivní, protože opakováně počítáme stejné informace. Obrázek 10.2 ukazuje konkrétní příklad výpočtu, všimněte si, že  $\text{fib}(3)$  se počítá opakovaně. Složitost výpočtu roste exponenciálně a okolo hodnoty  $n = 35$  začne výpočet trvat velmi dlouho. Iterativní řešení má složitost lineární a běží rychle. Příklad tedy pěkně ilustruje, jak výrazný vliv na použitelnost programu může mít nevhodná volba algoritmu.



Obrázek 10.2: Výpočet Fibonacciho čísel pomocí rekurze

## Pascalův trojúhelník

Řešení je relativně přímočaré, takže jej neuvádíme.

## Výpočet $\pi$

Pro ilustraci uvádíme kód pro Monte Carlo metodu pro s házením šipek:

```
def monte_carlo_sipky(pocet_pokusu):
    zasahy = 0
    for k in range(pocet_pokusu):
        x = random()
        y = random()
        if x*x + y*y < 1:
            zasahy += 1
    return 4.0 * zasahy / pocet_pokusu
```

## Permutace, kombinace, variace

Uvádíme řešení pro výpis všech kombinací, výpisu permutací a variací lze udělat podobným stylem:

```
def kombinace(seznam, k):
    if k==0: return [ [] ]
    if len(seznam) < k: return []
    vystup = [ ]
    x = seznam.pop()
    for komb in kombinace(seznam, k-1):
        komb.append(x)
        vystup.append(komb)
    vystup.extend(kombinace(seznam, k))
    seznam.append(x)
    return vystup
```

Více „pythonské“ řešení využívá příkazu `yield`, záměrně však uvádíme řešení, které je méně specifické pro Python.

## 10.2 Obrázky a geometrie

### Textová grafika

Většina příkladů se řeší vcelku přímočaře za použití dvou vnořených cyklů a několika testů. Řešení jde zapsat kompaktně, někdy to však vyžaduje nápad a vhodný pohled na obrázek. Například příklad „čtverce“ jde vyřešit následujícím krátkým kódem:

```
def ctverce(n):
    for y in range(-n, n+1):
        for x in range(-n, n+1):
            if x % 2 == 0 and abs(x) >= abs(y) or \
               y % 2 == 0 and abs(y) >= abs(x): print "*",
            else: print " ",
    print
```

Kruh a sinusovka vyžadují použití jednoduchých geometrických znalostí, konkrétně Pythagorovy věty v případě kruhu a funkce sinus v případě sinusovky. Jinak však nejsou náročné. Po programátorské stránce je nejobtížnější spirála. K její realizaci se hodí využít pomocné dvojrozměrné pole, do kterého spirálu postupně vykreslujeme pomocí „simulace pohybu“:

```
def spirala(n):
    inicializuj prázdné pole velikosti n krát n
    x = n; y = 1; směr = dolů
    dokud neukončeno:
        zapiš '*' do pole na souřadnici (x,y)
        změň souřadnici (x,y) podle aktuálního směru
        pokud se nelze posunout, změň směr
    vypiš pole
```

### Želví grafika: základy

Základní vykreslování mnohoúhelníku dostaneme následovně:

```
def mnohouhelnik(pocet_vrcholu, delka_strany):
    for i in range(pocet_vrcholu):
        forward(delka_strany)
        right(360.0 / pocet_vrcholu)
```

Hvězdičky získáme vhodnou úpravou úhlu. Kružnici o poloměru  $r$  můžeme approximovat jako 360úhelník se správně určenou délkou kroku. Obvod

kružnice je dán vztahem  $2\pi r$ , obvod 360úhelníku je 360 krát délka strany, approximaci kružnice o poloměru  $r$  tedy dostaneme pomocí volání polygon (360,  $2*\pi*r/360$ ).

Pro vykreslení pravoúhlého trojúhelníku o zadaných délkách stran potřebujeme využít funkci *arctan* a odmocninu (*sqrt*):

```
def pravouhly_trojuhelnik(a,b):
    forward(a)
    right(180 - arctan(b / a))
    forward(sqrt(a * a + b * b))
    right(180 - arctan(a / b))
    forward(b)
```

Diamant na první pohled vypadá komplikovaně – jako spousta čtyřúhelníků různých tvarů. Když se však na obrázek správně podíváme, zjistíme, že se skládá z dvanácti dvanáctiúhelníků, a jde tedy jednoduše vykreslit následujícím kódem (zápis je obecný, konkrétní obrázek dostaneme pro  $n = 12$ ):

```
def diamant(n, delka_strany):
    for i in range(n):
        mnohouhelnik(n, delka_strany)
        right(360.0 / n)
```

Podobně elegantně jde řešit i příklad „koule“, zde pouze navíc využijeme příkazy *penup* a *pendown*. Základní spirálu můžeme vykreslit například pomocí následujícího kódu (úpravou úhlu získáme různé spirály):

```
for i in range(50): forward(i * 5); right(60)
```

## Želví grafika: fraktály

Řešení pro keř odpovídá slovnímu popisu „Vykreslit keř znamená nakreslit stonek a pak nakreslit dva natočené menší keř.“:

```
def stromecek(delka):
    forward(delka)
    if delka > 10:
        left(45)
        stromecek(0.6 * delka)
        right(90)
        stromecek(0.6 * delka)
        left(45)
    back(delka)
```

Je důležité nezapomenout na poslední dva řádky, které způsobí návrat do výchozí pozice. Bez nich rekurzivní přístup nebude fungovat. Pokud vám tento aspekt řešení není jasný, je užitečné si program vyzkoušet, odkrokovat a pořádně pochopit.

Kochovu vločku můžeme vykreslit následujícím kódem, který odpovídá přepisovací gramatice uvedené v komentáři (kód kreslí jen jednu ze tří stran vločky):

```
def koch(n):
    if n == 1:
        forward(5)
    else:
        koch(n - 1); left(60); koch(n - 1); right(120)
        koch(n - 1); left(60); koch(n - 1)
```

Poslední tři příklady už jsou trochu náročnější, ale také jdou řešit krátkým kódem. Například Hilbertovu křivku generuje následující přepisovací systém:

- symboly: L, R, F, A, B
- výchozí axiom: A
- přepisovací pravidla:
  - A  $\Rightarrow$  LBFR A F A R F B L
  - B  $\Rightarrow$  R A F L B F B L F A R

Symboly L, R, F značí otočení doleva, doprava a posun dopředu, symboly A, B slouží pouze pro generování. Tento přepisovací systém můžeme převést na program podobně jako u Kochovy vločky.

## Sierpiňského fraktál

Sierpiňského fraktál je založen na rekurzivní myšlence „Sierpiňského fraktál stupně  $n$  se skládá ze tří Sierpiňského fraktálů stupně  $n - 1$ “, což můžeme pomocí želví grafiky zapsat následovně:

```
def sierpinski(n, delka):
    if n == 1:
        trojuhelnik(delka)
    else:
        for i in range(3):
            sierpinski(n - 1, delka)
            forward((2 ** (n - 1)) * delka)
            right(120)
```

Z ostatních úloh je myšlenkově náročnější pouze vymyšlení pravidel pro šestiúhelníkovou čáru. Tu můžeme dostat pomocí následujícího přepisovacího systému (princip viz předchozí cvičení):

- použité symboly: A, B, +, -
- výchozí axiom: A
- pravidla:
  - $A \Rightarrow B-A-B$
  - $B \Rightarrow A+B+A$

Interpretace symbolů pro vykreslování: A i B znamenají posun dopředu, + (resp. -) značí otočení doleva (resp. doprava) o 60 stupňů.

## Bitmapová grafika

Parametrické rovnici kružnice odpovídá následující funkce:

```
def kruznice1(a, b, r):
    for t in range(0, 360):
        putPixel(a + r * cos(t), b + r * sin(t))
```

S obecnou rovnicí  $(x - a)^2 + (y - b)^2 = r^2$  musíme být trochu opatrní. Kdybychom použili test na přesnou rovnost, vykreslilo by se jen velmi málo bodů. Musíme tedy udělat přibližný test, v následujícím kódu konstanta `eps` určuje přesnost testu, tj. šířku výsledné čáry:

```
def kruznice2(a, b, r, eps):
    for x in range(a - r, a + r):
        for y in range(b - r, b + r):
            if abs((x - a)**2 + (y - b)**2 - r**2) < eps:
                putPixel(x, y)
```

## Mandelbrotova množina

Jako názornou ukázkou toho, jak krátkým kódem lze vygenerovat elegantní fraktál, uvádíme kompletní řešení pro vykreslení Mandelbrotovy množiny, a to včetně barevného stínování. Řešení využívá knihovnu `Image`.

```
import Image
velikost = 600
im = Image.new("RGB", (velikost, velikost), (255, 255, 255))
pix = im.load()
x0, y0, d = -2.2, -1.5, 3.0
```

```

for i in range(velikost):
    for j in range(velikost):
        x = x0 + d * i / velikost
        y = y0 + d * j / velikost
        c = complex(x, y)
        z = complex(0,0)
        kroky, suma = 0, 0
        while abs(z) < 2 and kroky < 25:
            z = z*z + c
            kroky += 1
            suma += abs(z)
        if abs(z) < 2:
            pix[i,j] = (max(255-int(300*suma/kroky),0),0,0)
        else:
            pix[i,j] = (255-kroky*10,255-kroky*10,250)
im.save("mandelbrot.png")

```

## Konvexní obal

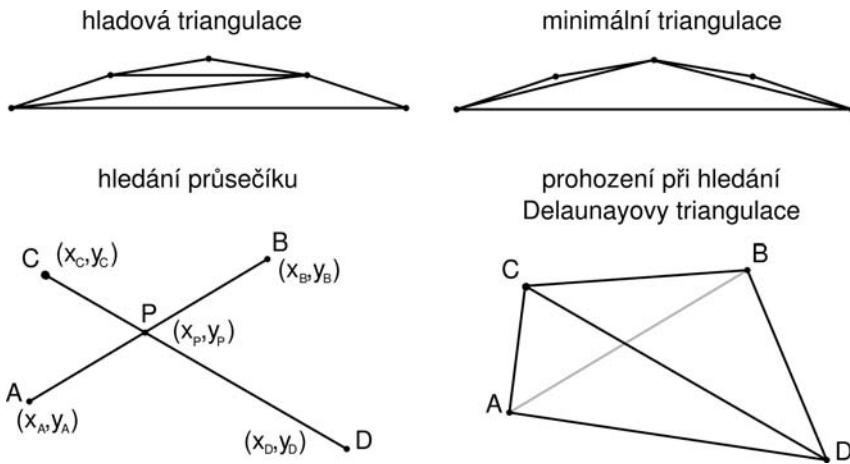
Základní myšlenky Jarvisova a Grahamova algoritmu se nacházají v komentáři a podrobněji je rozebírat nebudeme. Jen uvedeme jednu praktickou radu – při popisu algoritmů bylo zmíněno řazení bodů podle úhlů. V praktické implementaci však nemusíme pracovat přímo s úhly, stačí vypočítat tangens (tj. poměr protilehlé a přilehlé strany v trojúhelníku), protože tangens je monotonní funkce, a tudíž nemění uspořádání (což je to jediné, co nás zajímá).

Úloha „loupání cibule“ se řeší jen opakovánou aplikací konvexního obalu. Minimální ohraňující obdélník jde také řešit za využití konvexního obalu (nejdříve sestrojíme konvexní obal a pak zkoušíme „přikládat“ ohraňující obdélník na jednotlivé strany), existuje však i efektivnější řešení s lineární složitostí.

## Triangulace

Obrázek 10.3 ukazuje konkrétní příklad, kdy hladový algoritmus selže. Pro konvexní mnohoúhelník můžeme najít optimální řešení pomocí dynamického programování. Základní myšlenka je následující:

- počítáme hodnoty  $M_{i,j}$ , které udávají celkovou délku minimální triangulace mnohoúhelníku tvořeného body  $i, i+1, \dots, j-1, j$ ,
- tyto hodnoty počítáme „odspodu“, tj. začneme pro  $i, j$  blízko sebe a postupně počítáme pro vzdálenější,  $M_{i,j}$  pak můžeme napočítat pomocí hledání minima přes  $k$  ležící mezi  $i, j$  za využití hodnot  $M_{i,k}$  a  $M_{k,j}$ .



**Obrázek 10.3:** Triangulace – vysvětlující ilustrace

Základní algoritmus pro obecnou množinu bodů můžeme zapsat následujícím pseudokódem:

```
def triangulace(body):
    inicializuj prázdný výběr
    pro všechny úsečky U mezi body:
        pokud se U neprotíná z žádnou úsečkou ve výběru:
            přidej U do výběru
    return výběr
```

Jako dílčí krok v tomto postupu musíme umět otestovat, zda se dvě úsečky  $AB$  a  $CD$  protínají. Uděláme to tak, že najdeme průsečík přímek procházející body  $AB$  a  $CD$  a pak zkонтrolujeme, zda tento průsečík leží uvnitř zadaných úseček;  $x$ -ovou souřadnici průsečíku získáme z následujícího výrazu (význam proměnných viz obrázek 10.3):

$$x_P = \frac{(x_A y_B - y_A x_B)(x_C - x_D) - (x_A - x_B)(x_C y_D - y_C x_D)}{(x_A - x_B)(y_C - y_D) - (y_A - y_B)(x_C - x_D)}$$

Výraz vypadá na první pohled komplikovaně, ale lze ho jednoduchými úpravami odvodit z lineárních rovnic popisujících úsečky.

Základní myšlenka algoritmu pro konstrukci Delaunayovy triangulace spočívá v postupném prohozování hran. Začneme s libovolnou triangulací, zjistíme, zda splňuje podmínu Delaunayovy triangulace, a pokud ne, provedeme lokální prohození – viz obrázek 10.3, kde úsečku  $CD$  nahrazujeme za

*AB.* Tento postup opakujeme stále dokola. Základní postup je tedy jednoduchý, ale aby byl algoritmus efektivní, musíme vhodně reprezentovat aktuální triangulaci a umět s ní efektivně pracovat.

## 10.3 Šifrování a práce s textem

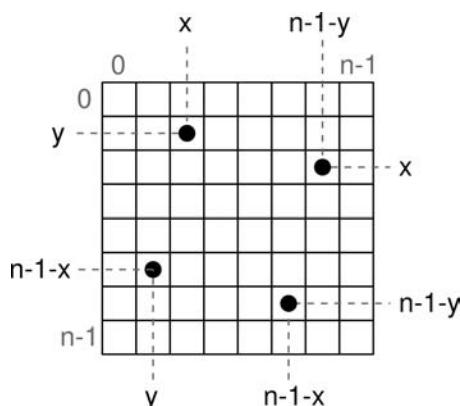
### Analýza a imitace textu

Tato úloha slouží především k procvičení syntaxe programovacího jazyka, takže ji více nerozebíráme.

### Transpoziční šifry

Stručně okomentujeme hlavní body k šifrování podle hesla a podle mřížky. U šifrování podle hesla je vhodné postupovat ve dvou krocích. V prvním kroku napočítáme permutaci odpovídající zadanému heslu. Například pro uvedený příklad s heslem petrklic tato permutace udává, že první sloupec zašifrované zprávy odpovídá osmému sloupci původní zprávy, protože písmeno c je abecedně nejmenší z celého hesla. Druhý sloupec šifry odpovídá druhému sloupci zprávy (písmeno e) a tak dále. Ve druhém kroku pak již s využitím takto napočítané permutace snadno zprávu přepřešeme.

U šifrování podle mřížky je hlavní neudělat chybu v počítání indexů při rotaci mřížky. Pokud máme mřížku velikosti  $n$  a indexování od nuly, jak se běžně používá ve většině programovacích jazyků, pole se souřadnicemi  $[x, y]$  při rotaci přechází na pole o souřadnicích  $[n-1-y, x]$ ,  $[n-1-x, n-1-y]$  a  $[y, n-1-x]$ :



## Substituce a kódování

Pro substituci potřebujeme využít funkci `ord` pro převod písmena na pořadové číslo a funkci `chr` pro převod pořadového čísla na písmeno. Subtituci podle hesla pak zapíšeme snadno následovně:

```
def posun_heslo(zprava, heslo):
    vystup = ""
    index_heslo = 0
    for i in range(len(s)):
        c1 = ord(zprava[i]) - ord('A')
        c2 = ord(heslo[index_heslo]) - ord('A')
        c1 = c1 + c2
        c1 = c1 % 26
        vystup += chr(c1 + ord('A'))
        index_heslo = (index_heslo + 1) % len(heslo)
```

Převodník do kódování může posloužit pro ilustraci různých způsobů reprezentace dat. Například převodník do Morseovy abecedy můžeme udělat následujícími způsoby:

- pomocí `if/elif` (případně `switch` a `case` v jazycích, jako je C):
 

```
if znak == 'A': kod = ".-"
elif znak == 'B': kod = "-..."
elif znak == 'C': kod = "-.-."
...
...
```
- pomocí pole, za využití příkazu `ord`, který převádí znak na jeho pořadové číslo:
 

```
morse[26] = [ ".-", "-...", "-.-.", ...]
kod = morse[ ord(znak) - ord("A") ]
```
- pomocí datového typu slovník:
 

```
morse = { 'A': ".-", 'B': "-...", 'C': "-.-.", ...}
kod = morse[znak]
```

První varianta může posloužit jako ilustrace zřetězených `if/elif` (případně `switch` v jiných jazycích), jinak je to ovšem nevhodné řešení. Vždy je dobré snažit se oddělovat popis funkcionality programu a popis dat. Rozdíl ve vhodnosti těchto přístupů se projeví například tehdy, když chceme do programu přidat nové kódování: v prvním případě musíme kopírovat a měnit velký kus kódu, v druhém případě pouze přidáme jeden řádek s definicí nového kódování a pozměníme jen malý kousek vlastního funkčního kódu. V tomto případě ještě rozdíl není kritický, ale u rozsáhlejšího projektu může být vhodné oddělení dat a funkcionality zásadní.

## Rozlomení šifer

Řešení konkrétních zadání (v závorce je vždy uvedeno heslo):

- Caesarova šifra:
  1. Lež má krátké nohy. (18)
  2. Bez práce nejsou koláče. (7)
  3. Kdo jinému jámu kopá, sám do ní padá. (21)
- Transpozice podle hesla:
  1. Koho chleba jíš, toho píšeň zpívej. (prase)
  2. Jak se do lesa volá, tak se z lesa ozývá. (kostel)
  3. Tak dlouho se chodí se džbánem pro vodu, až se ucho utrhne. (slovansk)
- Substituce podle hesla:
  1. Princeznička na bále, poztrácela korále. (pes)
  2. Přišel k nám host, postavil most, bez sekery bez dláta, přec je pevný dost. (sova)
  3. Běží liška k táboru, nese pytel zázvoru, ježek za ní pospíchá, že jí pytel propichá. (kosti)

Substituci podle hesla (Vigenèrovu šifru) můžeme vyřešit následovně. Předpokládejme, že víme, že heslo má délku  $n$ . Pokud délku hesla neznáme, opakujeme postup prostě pro různé hodnoty  $n$ . Provedeme vždy následující:

1. Zprávu si rozdělíme na  $n$  skupin, každou skupinu tvoří písmena „ob  $n$  pozic“.
2. Pro každou skupinu uvážíme všech 26 možných posunů, pro každý z nich spočítáme frekvenční analýzu a najdeme  $k$  nejnadějnějších posunů, tj. takových, které nejlépe korelují s frekvencemi v češtině.
3. Vyzkoušíme všech  $k^n$  možných hesel, pro každé z nich provedeme dešifrování zprávy a její obdobování podle slovníku.

## Přesmyčky

Myšlenku s reprezentanty realizujeme pomocí datové struktury slovník – každý reprezentant odkazuje na seznam slov, která tvoří jeho přesmyčku. Pro kontrolu uvádíme data o přesmyčkách v citovaném korpusovém slovníku (SYN2005). Pokud uvažujeme i diakritiku, je nejpočetnější skupina slov, která tvoří vzájemné přesmyčky, šestičlenná a existuje pouze jedna: „stavil, vstali, vlasti, slavit, svalit, svitla“, dále existuje jedna pětičlenná a 7 čtyřčlenných.

Hledání přesmyček s využitím datové struktury trie provedeme pomocí procházení trie za využití algoritmu backtracking:

- Začneme v kořenu stromu, postupujeme po větvích dolů, přičemž používáme pouze větve obsahující zadaná písmena.

- Použitá písmena si evidujeme, abychom každé použili pouze jednou.
- Pokud aktuální vrchol reprezentuje platné slovo, můžeme skočit zpět do kořenu a začít nové slovo.
- Pokud použijeme všechna slova a aktuální vrchol reprezentuje platné slovo, našli jsme platnou přesmyčku.
- Pokud nemáme jak pokračovat, vracíme se zpět a zkoušíme jinou možnost (backtracking).

Uvedený postup jde snadno dále rozšířit, abychom používali pouze slova určité minimální délky nebo četnosti.

## 10.4 Logické úlohy

### Číselné bludiště

Pro ukázku uvádíme kompletní program, který vygeneruje a vypíše náhodné zadání a pak jej zkusí vyřešit za použití prohledávání do šířky. Program názorně ukazuje, jak lze kompaktně zapsat prohledávání do šířky nad dvojrozměrnou mřížkou. Program tak může posloužit jako inspirace pro další cvičení (např. Hledání cest v bludišti).

Na rozdíl od ostatních ukázk tento program využívá některé prvky specifické pro Python, nicméně základní myšlenka by snad měla být zřejmá i pro příznivce jiných jazyků. Hlavní netradiční prvek spočívá v reprezentaci plánu pomocí „slovníku dvojic“ a nikoliv pomocí běžně používaného dvojrozměrného pole. Použitý způsob například umožňuje elegantní test, zda skok nevede ven z plánu.

```
from collections import deque
from random import randint

def vygeneruj_plan(n):
    plan = {}
    for y in range(n):
        for x in range(n):
            plan[x,y] = randint(1,n-1)
            print plan[x,y],
        print
    plan[n-1,n-1] = 'cil'
    return plan
```

```

def najdi_reseni(plan):
    start = (0,0)
    fronta = deque([ start ])
    cesta = {}
    cesta[start] = 'start'

    while len(fronta) > 0:
        (x,y) = fronta.popleft()
        if plan[x,y] == 'cil':
            print cesta[x,y]
            break
        for (dx, dy) in [(-1,0), (1,0), (0,-1), (0,1)]:
            nove_x = x + dx * plan[x,y]
            nove_y = y + dy * plan[x,y]
            if (nove_x, nove_y) in plan and \
                not (nove_x, nove_y) in cesta:
                fronta.append((nove_x,nove_y))
                cesta[nove_x,nove_y] = cesta[x,y] + \
                    '(nove_x,nove_y)'

```

najdi\_reseni(vygeneruj\_plan(5))

Co se týče vytváření zadání, několik zajímavých metod je popsáno v článku *Rook Jumping Maze Generation for AI Education* (T. W. Neller, FLAIRS, 2011).

## Přelévání vody

Jak je zmíněno v komentáři, jde o standardní prohledávání stavového prostoru pomocí procházení do šírky (viz např. řešení cvičení Číselné bludiště). Zmíníme tedy pouze způsob výpočtu následníka. V následujícím kódu je proměnná stav seznam aktuálních obsahů nádob a kapacita seznam maximálních obsahů nádob. Stav, který vznikne přelitím z nádoby  $i$  do nádoby  $j$ , dostaneme takto:

```

novy_stav = stav[:]
if kapacita[j] - stav[j] < stav[i]:
    novy_stav[i] = stav[i] - (kapacita[j] - stav[j])
    novy_stav[j] = kapacita[j]
else:
    novy_stav[i] = 0
    novy_stav[j] = stav[j] + stav[i]

```

## Hanojské věže

Myšlenka rekurzivního řešení základní úlohy je popsána v komentáři, stačí tedy přepsat tuto myšlenku pomocí rekurzivní funkce. Tato rekurzivní funkce musí mít jako parametry počet disků a identifikaci kolíků, mezi kterými má přesun proběhnout:

```
def presun(n, odkud, kam, kudy):
    if n==1:
        print odkud, "->", kam, ";"
    else:
        presun(n-1, odkud, kudy, kam)
        presun(1, odkud, kam, kudy)
        presun(n-1, kudy, kam, odkud)
```

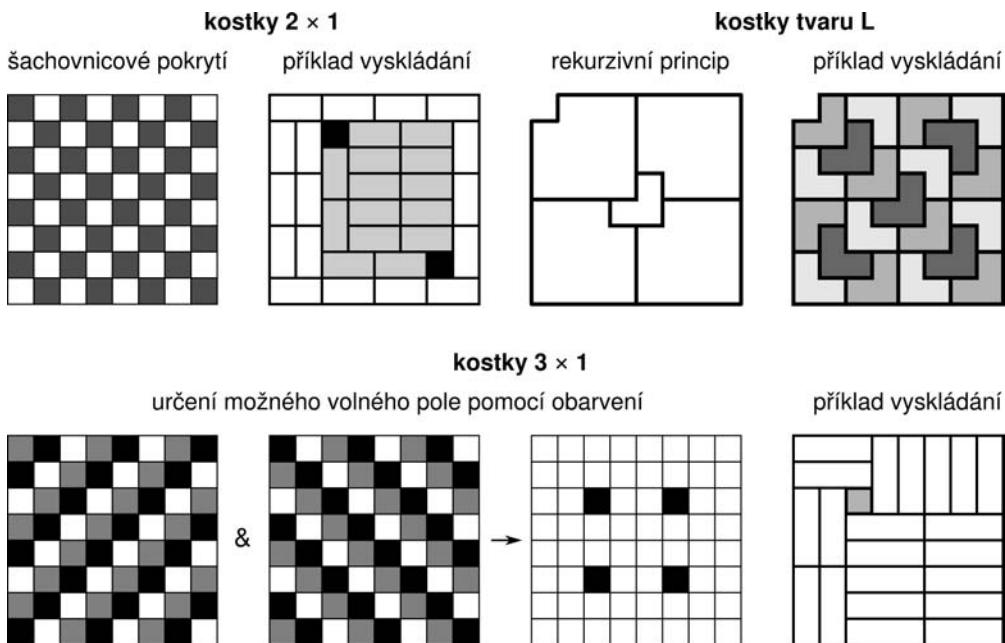
K řešení rozšířené varianty využijeme znalost předchozího řešení a rekurze. Rekurzivně definujeme postup „shromáždi  $n$  disků“, který slouží k tomu, abychom shromázdili  $n$  nejmenších disků na jednom kolíku. Rekurzivně můžeme tento postup vyjádřit následovně: „shromáždi  $n$  disků = shromáždi  $n - 1$  disků + přesuň  $n - 1$  disků na  $n$ -tý disk.“ Při řešení využíváme toho, že přesun věže disků již umíme řešit (viz základní zadání). Libovolnou úlohu s nepravidelným rozmístěním tak můžeme řešit pomocí kombinace postupů „shromáždi“ a „přesuň“.

V případě základního zadání vedl rekurzivní postup k nejkratšímu řešení, které má délku  $2^n - 1$ . V obecném případě tomu tak být nemusí – může existovat kratší řešení než to, které najdeme pomocí naznačeného rekurzivního postupu. Optimální řešení můžeme dostat pomocí prohledávání do šírky ve stavovém prostoru úlohy.

## Pokryvání mřížky

Pro řešení úlohy s kostkami  $2 \times 1$  se hodí představit si mřížku obarvenou jako šachovnici, tedy střídavě černou a bílou barvou (viz obrázek 10.4). Je zřejmé, že každá kostka domina pokryje jedno černé a jedno bílé pole, takže aby byla úloha řešitelná, musí být počet černých a bílých polí na mřížce stejný. Z mřížky tedy musí být odstraněno jedno černé a jedno bílé pole. Například pro zmíněné zadání, ve kterém jsou odstraněny dva protilehlé rohy, je úloha neřešitelná.

Pokud je úloha řešitelná, potřebujeme najít konkrétní vyskládání. Zde již není žádná hlubší myšlenka, musíme si to jen dobře rozmyslet. V mřížce vyznačíme obdélník vytyčený chybějícími políčky (viz obrázek 10.4). Pokud mají chybějící pole různé barvy, musí mít tento obdélník jeden rozměr sudé délky



Obrázek 10.4: Pokrývání mřížky – řešení

a druhý liché délky. Díky tomu jde vždy snadno vyskládat vnitřní i vnější oblast vytyčeného obdélníku.

Pro řešení úlohy s kostkami tvaru  $3 \times 1$  můžeme využít stejný princip, pouze barvujeme 3 barvami (viz obrázek 10.4). Opět platí, že každá kostka pokryje právě jedno pole od každé barvy, potřebujeme mít tedy od všech barev stejný počet polí. V barvení uvedeném na obrázku máme 22 černých, 21 bílých a 21 šedých polí. Musíme tedy odstranit černé pole. Stejný argument platí i pro rotované barvení, musíme tedy odstranit pole, které je černé na obou pláncích. Úloha je tedy řešitelná jen pro 4 označená pole a tyto čtyři úlohy lze řešit pomocí jednoho fixního vyskládání, které pouze rotujeme.

Úloha s kostičkami tvaru L jde řešit rekurzivně. Základní myšlenka rekurzivního řešení je znázorněna na obrázku 10.4. Mřížku si rozdělíme na čtvrtiny a doprostřed umístíme jednu kostku. Tím dostaneme 4 menší mřížky s chybějícím polem, a pro ty úlohu řešíme rekurzivně. Pro funkčnost rekurzivního řešení je klíčové, že zadání má rozměry velikosti  $2^n \times 2^n$ , takže máme zaručeno, že vždy bude možné mřížku rozdělit na stejné čtvrtiny.

Řešení zapíšeme pomocí rekurzivní funkce, která má následující argumenty: velikost mřížky, posun v rámci celkové mřížky, souřadnice chybějícího

pole. Funkce funguje tak, že umístí jednu kostičku do středu aktuální mřížky a pak zavolá čtyřikrát rekurzivně sama sebe pro 4 mřížky poloviční velikosti. Řešení včetně obarvení pomocí tří barev je ilustrováno na obrázku 10.4. Hledání obarvení lze řešit také rekurzivně, přesný postup necháváme jako otevřený problém pro čtenáře.

## Hledání cest v bludišti

Základní úloha jde řešit přímo pomocí prohledávání do šířky a řešitelná drobnou úpravou kódu uvedeného předchozím řešením. Podívejme se tedy na rozšířená zadání. Úloha „Robot v bludišti“ jde také řešit základním prohledáváním do šířky, pouze musíme pracovat nad správným grafem. Stav robota je dán trojicí  $(x, y, r)$  kde  $x, y$  udává polohu v mřížce a  $r$  natočení robota, tj. jeden ze 4 směrů. Sousedé stavu jsou trojice odpovídající bud' otočení (změna  $r$ ) nebo posunu o jednu pozici dopředu (změna  $x$  nebo  $y$  podle aktuální hodnoty  $r$ ).

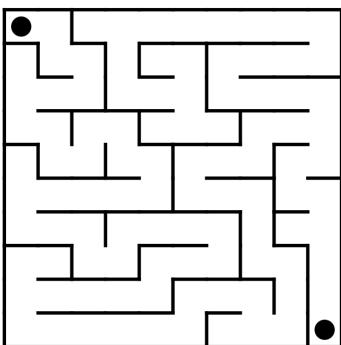
Úlohou „Dynamit“ můžeme převést na hledání nejkratší cesty ve váženém grafu – cena přechodu přes volné pole je 1, cena přechodu přes pole se zdí (tj. použití dynamitu) je „velká konstanta“  $k$ . S trochou rozmyšlení je zřejmé, že dostatečně „velká konstanta“ je  $N \times M$ , kde  $N, M$  jsou rozměry mřížky. Pro hledání nejkratších cest ve vážených grafech se používá Dijkstrův algoritmus, který je rozšířením prohledávání do šířky a využívá prioritní frontu místo obyčejné fronty.

U úlohy „Spojnice 3 bodů“ je vhodné začít tím, že si rozmyslíme, jak může spojnice vypadat. Bud' je to jedna cesta bez větvení nebo cesta tvaru Y. Budeme tedy hledat kandidáta na „rozcestí“. Pro každé pole napočítáme nejkratší vzdálenost do bodů A, B, C. Na to nám stačí spustit prohledávání do šířky z každého z těchto tří vrcholů. Poté najdeme pole, ze kterého je součet těchto tří vzdáleností minimální. To je bud' jeden z bodů A, B, C, a pak je řešením nevětvící se cesta, nebo jiný bod, který je rozcestím Y křížovatky. Pro více jak 3 body už je úloha výrazně komplikovanější, pro obecný počet bodů je problém NP-úplný, tj. nemáme efektivní algoritmus.

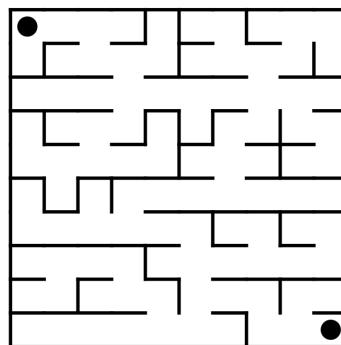
## Generování bludišť

Perfektní bludiště lze generovat pomocí vhodně upravených grafových algoritmů – náhodnostního prohledávání do hloubky nebo algoritmů pro hledání kostry grafu. Zajímavé je, že tyto dva přístupy vedou k výrazně odlišným grafům (viz obrázek 10.5). Bludiště vytvořené pomocí náhodnostního prohledávání do hloubky obsahuje málo dlouhých chodeb. Bludiště vytvořené pomocí

Prohledávání do hloubky



Kruskalův algoritmus



Obrázek 10.5: Ukázky vygenerovaných perfektních bludišť

Kruskalova algoritmu pro hledání kostry grafu má naopak mnoho krátkých odboček. Co z toho je hezčí? Co je pro nás náročnější na řešení?

Pro bludiště typu pletenec je hlavní myšlenka také jednoduchá – postupně v náhodném pořadí přidáváme zdi a přitom kontrolujeme, že nevytvoříme žádné slepé cesty. Trochu se to však zkomplikuje, pokud chceme, aby nám v bludišti nevznikla žádná „náměstí“, tedy průsečíky mřížky, od kterých nevede žádná zed'. Absence takových náměstí je přirozený požadavek na „hezkost“ bludiště.

Pokud zvolíme variantu s více cíli, můžeme bud' vygenerovat nejdříve jedno souvislé bludiště a to pak rozdělit nebo rovnou průběžně udržovat několik oddělených oblastí. Každopádně bychom se měli snažit, aby výsledné oblasti měly přibližně podobnou velikost. Pokud je oblast malá, je pro člověka příliš snadné určit, že příslušný cíl není ten správný.

Více informací o bludištích, jejich generování a obtížnosti řešení pro lidi lze nalézt například na webových stránkách [www.astrolog.org/labyrinth.htm](http://www.astrolog.org/labyrinth.htm) a ve dvou diplomových pracích vypracovaných na Fakultě informatiky MU: *Automated Maze Generation and Human Interaction* (M. Foltin, 2011), *Algoritmy pro generování a řešení bludišť* (P. Matějka, 2012).

## Rozmístování figur na šachovnici

Následující kód řeší základní „problém  $n$  dam“. Řešení využívá myšlenky popsané v komentáři a funkce `vypis` pro výpis řešení a `zkontroluj` pro kontrolu, zda aktuální stav je korektní:

```
def vyres_damy(n, stav):
    if len(stav) == n:
        vypis(stav)
        return True
    else:
        for i in range(n):
            stav.append(i)
            if zkontroluj(stav):
                if vyres_damy(n, stav): return True
            stav.pop()
    return False
```

Variantu „koně na úzkém plánu“ lze řešit pomocí dynamického programování. Základní přístup je následující. Plán procházíme zleva doprava a pro každou možnou kombinaci rozmístění koňů na posledních dvou sloupcích si pamatujeme maximální možný počet umístěných koňů. Stačí si pamatovat poslední dva sloupce, protože kůň má dosah maximálně 2 pole doleva. Protože  $n < 6$ , je možných kombinací figur ve dvou sloupcích maximálně  $2^{10} = 1\,024$ .

## Jak navštívit všechna pole mřížky?

Algoritmus backtracking je popsaný v komentáři a v předchozím řešení, takže tuto úlohu více nerozebíráme. Jen zmíníme pro kontrolu dvě zajímavosti:

- Schwenkova věta: Uzavřená koňská procházka je možná na všech plánech  $m \times n$  ( $m \leq n$ ), pokud neplatí některá z následujících podmínek:  
a)  $m$  a  $n$  jsou liché a nejsou současně 1, b)  $m$  je 1, 2 nebo 4 a  $m, n$  nejsou současně 1, c)  $m = 3$  a  $n = 4, 6$  nebo 8.
- Na poli  $4 \times 4$  je 368 uzavřených královských procházk, z nichž 63 je symetrických.

## Polyomina

Generování polyomin lze pojmet více způsoby. Jeden rozumný způsob spočívá v induktivním generování „odspodu“. Pokud umíme vygenerovat polyomina s  $n$  dílkami, můžeme polyomina s  $n+1$  dílkami vytvořit tak, že k polyominům s  $n$  dílkami zkoušíme připojit 1 další dílek všemi možnými způsoby. Následně

pak musíme provést detekci duplicit. Tu můžeme udělat pomocí „kanonizace“. Kostku vždy natočíme tak, aby byla co nejblíže počátku souřadnicového systému.

Základní přístup k řešení Skládačky je klasický backtracking. V každém kroku zkusíme umístit jeden dílek. Jakmile se zasekneme, protože další dílek není kam položit, vracíme se zpět a zkoušíme další možné položení předchozího dílku. Je však více způsobů, jak přesně vybírat, kterou kostku a kam umistujeme, různé možnosti vedou k výrazně různé efektivitě programu. Například pro tetromina je výhodné začít čtverečkem, kdežto dílek tvaru L umisťovat na konec – důvod souvisí s tím, že L dílek má osm možných natočení, kdežto čtvereček jen jedno. Vhodnou součástí projektu je experimentální prozkoumání vlivu pořadí dílků.

## Sudoku

Nejdříve potřebujeme problém reprezentovat. Samotnou mřížku lze přirozeně reprezentovat pomocí dvojrozměrného pole. Řešení pomocí algoritmu backtracking pak můžeme zapsat následujícím pseudokódem:

```
def Sudoku_vyřeš(stav):
    if s je vyřešeno:
        vypiš stav
    else:
        vyber volné pole (x,y)
        for i in range(1, 10):
            stav[x,y] = i
            if stav neporušuje podmínky:
                Sudoku_vyřeš(stav)
            stav[x,y] = 0
```

Pro implementaci propagace podmínek potřebujeme pracovat s informacemi o kandidátech, tj. potřebujeme datovou strukturu kandidáti, která udává pro každé pole mřížky kandidáty na danou pozici. Tuto strukturu můžeme reprezentovat například jako trojrozměrné pole pravdivostních hodnot. Dále vytvoříme funkci `zjemni_kandidáty`, která vypočítá hodnoty kandidátů na základě aktuálního stavu mřížky a přitom zkонтroluje, zda je aktuální stav konzistentní s pravidly Sudoku.

Jakmile máme napočítané kandidáty, můžeme odvozovat hodnoty políček. Popíšeme jen dvě základní techniky ilustrované na obrázku 6.12:

- Vynucená hodnota – pro dané pole přichází v úvahu jen jedna hodnota.

- Skrytá hodnota – v oblasti (řádek, sloupec, blok) se určitá hodnota vyskytuje jako kandidát jen v jednom políčku.

S napočítanými kandidáty je výpočet vynucených hodnot triviální. Výpočet skrytých hodnot je také snadný – pro každou hodnotu projdeme všechny oblasti, v každé z nich napočítáme počet pozic, na které lze číslo umístit, a pokud nám vyjde jediná, číslo umístíme.

Backtracking i propagaci podmínek lze přímočaře zkombinovat. V principu stačí vždy na začátku funkce `Sudoku_vyřeš` zavolat propagaci podmínek. Po implementační stránce si to vyžádá drobnou úpravu, protože v kombinovaném přístupu je nutné kopírovat stavy, kdežto v základním backtrackingu či propagaci podmínek stačilo udržovat v paměti jen jeden aktuální stav.

Pomocí kombinovaného přístupu je již reálné dosáhnout zmíněného cíle „vyřešit 1 000 Sudoku za 1 vteřinu“. Rychlosť řešení závisí mimojiné i na použité heuristice pro výběr dalšího pole, na které zkoušíme přiřazovat hodnoty. Pro srovnání je zajímavé vyzkoušet například a) náhodný výběr, b) postupný výběr (po řádcích), c) výběr nejvíce omezeného pole.

Základní metoda, která se používá k odhadování obtížnosti, vychází z přístupu „propagace podmínek“, který rámcově odpovídá tomu, jak úlohu řeší lidé. Při řešení pomocí propagace podmínek si pamatujiem, jaké logické techniky byly použity, a obtížnost úlohy ohodnotíme na základě obtížnosti použitých technik. Aby tento přístup opravdu fungoval, je potřeba dobře domyslet detaily, navíc existují i alternativní přístupy k odhadování obtížnosti. Zájemce o toto téma může najít více detailů v článku *Difficulty Rating of Sudoku Puzzles by a Computational Model* (R. Pelánek, FLAIRS Conference, 2011).

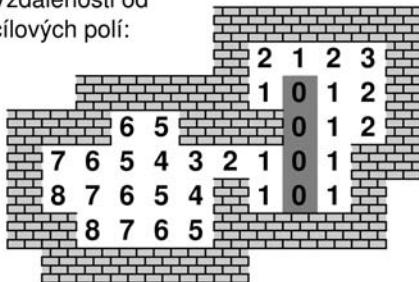
Generování zadání provedeme nejsnáz následovně. Nejdříve vygenerujeme náhodnou kompletní mřížku. To můžeme udělat tak, že začneme s pravidelnou mřížkou a pak náhodnosteně mícháme řádky a sloupce, nebo za použití algoritmu backtracking (s náhodnostením pořadím prohledávání) k vyřešení prázdné mřížky. V kompletní mřížce pak postupně v náhodném pořadí mazeme jednotlivá pole, přičemž po každém odebrání pomocí algoritmu pro řešení zkонтrolujeme, zda je úloha stále jednoznačná. Generování zadání zadané obtížnosti lze udělat nejjednodušji tak, že prostě vygenerujeme zadání, odhadneme obtížnost, a pokud obtížnost neodpovídá požadované obtížnosti, zadání zahodíme a vygenerujeme další.

## Sokoban

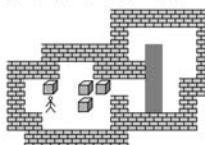
Jednoduchá hodnotící heuristika je ilustrována na obrázku 10.6. Nejdříve pro každé pole spočítáme jeho vzdálenost od cílového pole – to uděláme pomocí procházení do šířky na grafu odpovídajícím hracímu plánu. Ohodnocení stavu

**Heuristické ohodnocení**

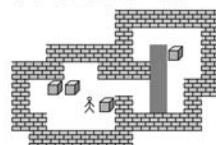
Vzdálenosti od cílových polí:



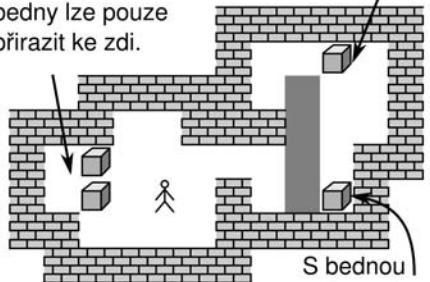
Ohodnocení:  
 $3 + 4 + 5 + 6 = 18$



Ohodnocení:  
 $1 + 4 + 5 + 6 = 16$

**Neperspektivní stav**

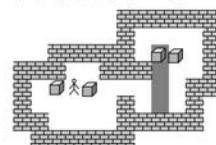
Nedostupná oblast:  
bedny lze pouze přirazit ke zdi.



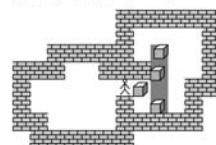
Bednu nelze odtlačit od zdi.

S bednou nelze pohnout.

Ohodnocení:  
 $0 + 1 + 4 + 6 = 11$



Ohodnocení:  
 $0 + 0 + 0 + 1 = 1$



Obrázek 10.6: Ukázky heuristik pro logickou úlohu Sokoban

pak vypočítáme jako součet hodnot na polích, na kterých zrovna stojí bedny. Při prohledávání pak upřednostníme stavě s menší heuristickou hodnotou. Obrázek dále ilustruje několik možných způsobů ořezávání neperspektivních stavů.

## 10.5 Hry

### Kámen, nůžky, papír

Řešení detailněji nerozebíráme.

### Hádání čísla

Uvádíme kompletní řešení pro případ, kdy počítač hádá číslo, které si myslí člověk. Program využívá metodu půlení intervalu. Kód lze využít jako názornou ilustraci konceptu „invariant cyklu“. Invariant použitého `while` cyklu je „hádané číslo musí být mezi spodni\_mez a horni\_mez“.

```

def hadani_cisla(n):
    print "Mysli si cislo od 1 do", n
    spodni_mez = 1
    horni_mez = n
    while spodni_mez != horni_mez:
        stred = (spodni_mez + horni_mez) / 2
        print "Je cislo", stred, "mensi (0), rovno (1),",
              "nebo vetsi (2) než tvoje cislo?"
        odpoved = input()
        if odpoved == 0: horni_mez = stred - 1
        elif odpoved == 1: horni_mez = spodni_mez = stred
        elif odpoved == 2: spodni_mez = stred + 1
    print "Tvoje cislo je", spodni_mez

```

## Oběšenec

Řešení detailněji nerozebíráme.

## Logik

Základní postup je popsán v komentáři, zde zmíníme jen jeden dílkí bod k řešení. U této úlohy se člověk může zbytečně zamotat při implementaci funkce pro určování počtu bílých kolíků, tj. počet kuliček se správnou barvou, ale na špatném místě. Elegantní řešení je následující. Spočítáme celkový počet shod barev a počet kuliček se správnou barvou na správném místě, tj. počet černých kolíků. Počet bílých pak dostaneme jako rozdíl těchto dvou čísel.

## Hra Nim

Pro základní variantu (povoleno odebírat 1, 2 nebo 3 sirky) je vítězná strategie „Zajisti, aby po tvém tahu byl počet sirek dělitelný 4“. Pokud je tedy na začátku počet sirek dělitelný 4, má vítěznou strategii druhý hráč, jinak ji má první hráč. Strategie pro vylučovací Nim je podobná.

Pro patrový Nim lze použít strategii s následujícími základními principy. Pro jednotlivá patra může první hráč zajistit, že druhý hráč sebere poslední sirku, tj. první hráč si zajistí, že na dalším patře bude opět začínat. U posledního patra záleží na tom, zda je počet sirek dělitelný 4 nebo ne. Podle toho je výhodné hrát druhý, resp. první. Až do předposledního patra si tedy první hráč bude držet „začínání“, v předposledním patře zahraje podle toho, zda je počet sirek v posledním patře dělitelný 4.

stav (3, 5, 7)	stav (1, 4, 5)
$3 = 011$	$1 = 001$
$5 = 101$	$5 = 101$
$7 = \underline{111}$	$4 = \underline{100}$
223	202
lichý, vítězný	sudý, prohrávající

Obrázek 10.7: Základní myšlenka řešení hromádkového Nimu

Variantu s volitelnými tahy lze řešit pomocí dynamického programování, tedy „vyplňováním tabulky odspodu“. Ilustrujme tento postup na příkladu pro povolené tahy 1, 3 nebo 4 sirkы. Vítěznou strategii znázorňuje následující tabulka, ve které symbol ‘–’ znamená, že hráč na tahu si nemůže zajistit vítězství:

počet sirek	1	2	3	4	5	6	7	8	9	10	11	12
vítězný tah	1	–	3	4	3	4	–	1	–	3	4	3

Tabulku můžeme zkonstruovat postupným vyplňováním odleva. Pro každou pozici se podíváme, zda se některým tahem můžeme dostat do situace označené symbolem ‘–’. Pokud takový tah existuje, zapíšeme jej do tabulky. Pokud neexistuje, zapíšeme do tabulky symbol ‘–’. Tento postup lze jednoduše implementovat pomocí dvou vnořených cyklů. Všimněte si, že řešení (včetně implementace) je velmi podobné řešení základní úlohy 8.1 Rozměřování mincí.

Hromádkový Nim lze řešit pomocí zmíněné souvislosti s binárními čísly. Počty sirek v hromádkách zapíšeme v binárním zápisu pod sebe a spočítáme počty jedniček v jednotlivých sloupečcích (viz obrázek 10.7). Stav hry označíme jako sudý, pokud všechny součty sloupečků jsou sudé, a lichý, pokud alespoň jeden součet sloupečku je lichý. Platí, že první hráč má výherní strategii právě v lichých stavech. Čtenáři necháváme jako úkol vymyslet zdůvodnění uvedeného tvrzení a domyslet přesnou strategii, tj. kolik sirek z které hromádky má hráč brát. Domyšlení detailů není úplně jednoduché, ale nevyžaduje již žádný originální nápad.

## Tetris

Jak je poznamenáno v komentáři, při vytváření přehrávače je především důležité dobře si rozmyslet reprezentaci tetromin – možnosti, jak to udělat, je více (dvojrozměrné pole, seznam, struktura) a může záležet na konkrétním jazyce, která z nich je nejvhodnější. Jakmile máme rozmyšlenou reprezentaci, snadno

napíšeme potřebné funkce „rotace dílku“, „posun dílku“, „vykreslení dílku“. Pak již stačí přidat kontrolu plných řádků a máme základ hotový.

Zajímavější je tvorba umělé inteligence. Úplně základní myšlenka je jednoduchá: vyzkoušíme všechny možné pozice a rotace aktuálního dílku, pro každou z nich necháme dílek „spadnout“ a heuristicky ohodnotíme stav hracího plánu. Následně vybereme tu možnost, která má nejlepší heuristické ohodnocení. Už tato základní myšlenka dává dostatek prostoru pro zajímavé experimentování, protože není úplně jasné, jak heuristické ohodnocení plánu udělat. Heuristika může zohledňovat například:

- počet kostek na plánu, resp. počet dosud odstraněných řádků,
- do jaké výšky sahá nejvyšší kostka,
- kolik je na plánu blokovaných volných polí, která znemožňují odstranění řádku,
- „hrbolatost“ plánu.

Algoritmus pak můžeme dále vylepšovat tím, že zohledňujeme i budoucí kostky. Pokud má program k dispozici informaci o budoucích kostkách, může provádět prohledávání přes různé kombinace umístění těchto známých kostek.

## Jednorozměrné piškvorky

Pro herní plány liché velikosti existuje jednoduchá herní strategie pro prvního hráče:

- První tah udělej na prostřední pole.
- V každém dalším tahu:
  - pokud je možné vytvořit tři křížky vedle sebe, zahraj příslušný tah a vyhraj,
  - jinak udělej středově symetrický tah oproti předchozímu tahu protihráče.

Při dodržení této strategie bude po tahu prvního hráče situace na herním plánu vždy symetrická a nebude umožňovat výhru, takže druhý hráč musí prohrát.

Řešení hrubou silou je založeno na následujícím principu. Stav je vítězný, pokud existuje způsob, jak umístit křížek tak, že nový stav je prohrávající. Pokud jsou všechny následující stavů vyhrávající, je aktuální stav prohrávající. Jde o zjednodušenou verzi algoritmu minimax, který je rozebrán v následujícím řešení u klasických piškvorek. Zde je algoritmus zjednodušený proto, že v jednorozměrných piškvorkách nemůže dojít k remíze. Navíc hra má velmi jednoduchá pravidla, takže můžeme snadno provádět současně generování stavů a jejich ohodnocení pomocí rekurzivního procházení:

```
vitezny_tah = {}  
def analyzuj(stav):  
    for i in range(len(stav)):  
        if stav[i] == '.':  
            novy_stav = stav[:i] + 'x' + stav[i+1:]  
            if 'xxx' in novy_stav:  
                vitezny_tah[stav] = i  
                return  
            if not novy_stav in vitezny_tah:  
                analyzuj(novy_stav)  
            if vitezny_tah[novy_stav] == '-':  
                vitezny_tah[stav] = i  
                return  
    vitezny_tah[stav] = '-'
```

Sofistikovanější řešení spočívá v naznačeném rozkladu hry na podhry. Jde o aplikaci matematické teorie sčítání her a využití Sprague-Grundyho funkce, což je však poměrně pokročilý matematický pojem, takže jej nerozebíráme.

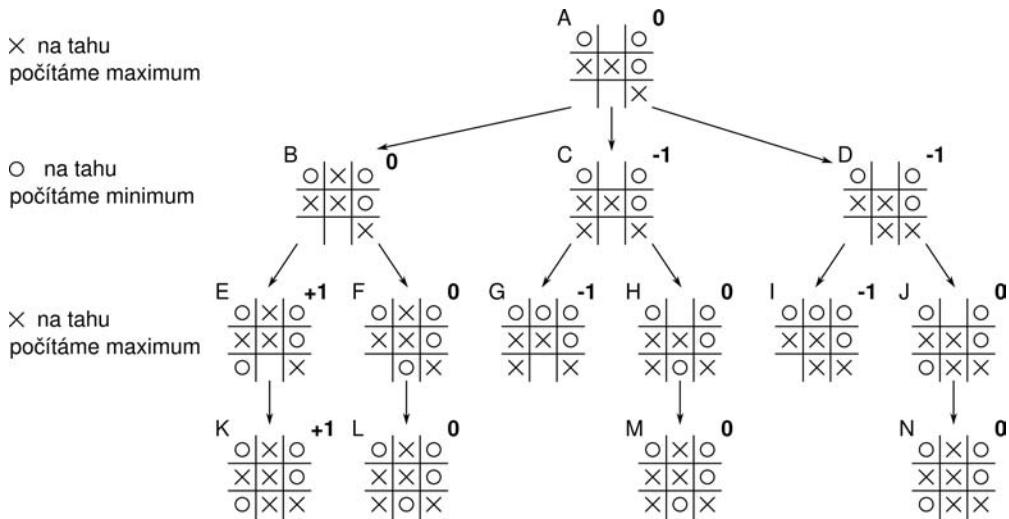
Pro kontrolu uvádíme, že pro  $N < 100$  existuje vítězná strategie pro druhého hráče pro velikost plánu 6, 12, 22, 30, 32, 44, 54, 64, 76, 86, 98, ve všech ostatních případech má vítěznou strategii první hráč.

## Piškvorky

Zde trochu detailněji rozebereme postup založený na prohledávání stavového prostoru hry. Naším cílem je označkovat všechny stavy ve stavovém prostoru informací o tom, kdo v nich vítězí. Z důvodů, které budou patrné za chvíli, budeme označovat stavy následovně: hodnotou +1 označíme stavy, kde má vítěznou strategii první hráč, hodnotou -1 označíme stavy, kde má vítěznou strategii druhý hráč, a hodnotou 0 označíme stavy, kde mají oba hráči strategii zaručující remízu.

Jako první si označíme koncové stavy, protože u nich přímo víme, jak je označit. Pak postupujeme směrem „proti tahům“ k začátku hry. Pokud má stav označkovány všechny svoje následovníky, můžeme označit i jeho. Pokud je ve stavu na tahu první hráč, označíme stav maximem z hodnot následovníků. Pokud je na tahu druhý hráč, bereme naopak minimum z hodnot následovníků. Podle tohoto střídavého počítání maxima a minima se algoritmus nazývá minimax.

Obrázek 10.8 ilustruje tento výpočet na příkladu piškvorek  $3 \times 3$ . I pro takto jednoduchou hru je celý graf hry příliš složitý, takže je na obrázku zachycena jen



Obrázek 10.8: Piškvorky – ilustrace algoritmu minimax

část stavového prostoru odpovídající jednomu konkrétnímu zakončení partie. Při ohodnocování začneme stav, které nemají následníka, což jsou v našem případě stavы G, I, K, L, M a N. Tyto stavы ohodnotíme prostě podle toho, kdo v nich vyhrává. Dále postupujeme stromem nahoru a ohodnocujeme další stavы. Například když ohodnocujeme stav D, podíváme se na jeho následníky: to jsou stav I s ohodnocením -1, tedy vítězství pro hráče kolečko, a stav J s ohodnocením 0, tedy remíza. Ve stavu D je na tahu druhý hráč a pro toho získáme ohodnocení tak, že spočítáme minimum z ohodnocení následníků, ohodnotíme jej tedy -1. To odpovídá tomu, že hráč kolečko v tuto chvíli potáhne do stavu I, čímž si zajistí vítězství. Ve stavu A je naopak na tahu první hráč, takže počítáme maximum z ohodnocení následníků. Výsledek je nula, což odpovídá tahu do stavu B, kterým si hráč může zajistit remízu.

Protože pro standardní piškvorky je počet stavů velmi vysoký, nemůžeme prohledat celý stavový prostor. Potřebujeme využívat heuristické ohodnocení stavů a ořezávaní podobně jako u úlohy 6.11 Sokoban. K tomu slouží rozšíření algoritmu minimax zvané Alfa-beta prohledávaní. Popis tohoto algoritmu je nad rámec této knihy, zájemce jej najde v literatuře.

Podobně jako u všech heuristických algoritmů je i zde klíčem k úspěchu volba dobré heuristické funkce. Musíme být schopni stav hry ohodnotit tak, aby ohodnocení odráželo šanci na vítězství z daného stavu, a musíme být schopni udělat ohodnocení rychle, protože heuristické ohodnocení počítáme pro mnoho stavů. Hlavní myšlenka heuristické funkce je zde stejná jako u zá-

kladního algoritmu: hledáme různé vzory a podle nich počítáme ohodnocení. Kvalita programu závisí na tom, jak přesně realizujeme detaily této základní myšlenky.

## Souboje virtuálních robotů

Tato úloha je velmi otevřená, žádné konkrétní řešení tedy neuvádíme.

# 10.6 Klasické informatické problémy

## Rozměňování mincí

Uvádíme základní kód pro hladový algoritmus a dynamické programování. Pokud naimplementujeme funkci `najdi_nejvetsi_minci`, která najde největší minci, jejíž hodnota je menší než zadaná částka, je hladový algoritmus triviální:

```
def rozmen_hladove(castka) :  
    while castka > 0:  
        mince = najdi_nejvetsi_minci(castka)  
        print mince,  
        castka -= mince
```

Program pro zadané částky využívající dynamické programování lze zapsat také stručně, ale již je myšlenkově náročnější, viz ilustrace s počítáním „tabulky“ v komentáři:

```
def rozmen_zadane(castka, mince) :  
    pocet = [ -1 for i in range(castka+1) ]  
    pocet[0] = 0  
    for i in range(1,castka+1) :  
        for m in mince:  
            if i-m >= 0 and pocet[i-m] != -1:  
                if pocet[i] == -1 or pocet[i-m] + 1 < pocet[i]:  
                    pocet[i] = pocet[i-m] + 1  
    print "Potrebny pocet minci:", pocet[castka]
```

Složitost tohoto algoritmu je  $O(nm)$ , kde  $n$  je cílová částka,  $m$  je počet mincí. Jde o takzvanou pseudo-polynomiální složitost – algoritmus je polynomiální vůči číslu  $n$ , ale nikoliv vůči délce vstupu, která má logaritmickou délku vzhledem k velikosti čísla.

**Tabulka 10.1:** Vyplněné tabulky pro určení nejdelší společné podposloupnosti a editační vzdálenosti

Nejdelší společná podposloupnost											Editační vzdálenost								
	p	a	m	p	e	l	i	s	k	a		s	a	t	u	r	n		
s	0	0	0	0	0	0	0	1	1	1		0	1	2	3	4	5	6	
e	0	0	0	0	1	1	1	1	1	1		o	1	1	2	3	4	5	6
d	0	0	0	0	1	1	1	1	1	1		s	2	1	2	3	4	5	6
m	0	0	1	1	1	1	1	1	1	1		t	3	2	2	2	3	4	5
i	0	0	1	1	1	1	2	2	2	2		r	4	3	3	3	3	3	4
k	0	0	1	1	1	1	2	2	3	3		o	5	4	4	4	4	4	4
r	0	0	1	1	1	1	2	2	3	3		v	6	5	5	5	5	5	5
a	0	1	1	1	1	1	2	2	3	4									
s	0	1	1	1	1	1	2	3	3	4									
k	0	1	1	1	1	1	2	3	4	4									
a	0	1	1	1	1	1	2	3	4	5									

## Simulátor hry Život

Řešení detailněji nerozebíráme.

## Problémy s řetězci a posloupnostmi

Program pro hledání nejdelší podposloupnosti je analogický výše uvedenému programu pro rozměňování mincí za použití dynamického programování, viz též tabulky uvedené v komentářích, které jsou také analogické.

Hledání nejdelší společné podposloupnosti pro dva řetězce a počítání editační vzdálenosti lze opět efektivně implementovat pomocí dynamického programování a „vyplňování tabulky“. Tabulka 10.1 uvádí konkrétní příklady. Číslo v tabulce vždy udává odpověď pro odpovídající prefixy, pro určení čísla stačí zohlednit vždy jen sousedy vlevo, nahoře a šikmo vlevo nahoře. Tabulku tedy můžeme zkonztruovat na jeden průchod, složitost algoritmů je tedy kvadratická –  $O(nm)$ , kde  $n$  a  $m$  jsou délky vstupních řetězců.

## Experimenty s řadicími algoritmy

Cvičení má relativně otevřené zadání, konkrétní řešení tedy neuvádíme.

## Vyhodnocování výrazu

Při vyhodnocování výrazu pomocí metody „rozděl a panuj“ je klíčovým krokem výběr operátoru, podle kterého výraz rozdělujeme. Hledáme operátor na nejvyšší úrovni, tj. takový, který není v závorkách. Pokud takový neexistuje, odstraníme „obalující“ závorky. Pokud máme na výběr více operátorů, upřednostňujeme  $+ -$  před  $*$  / a bereme operátor co nejvíce „vzadu“, např. ve výrazu  $3-2+1$  musíme vybrat  $+$  a nikoliv  $-$ .

Převod do postfixového zápisu (reverzní polské notace) se dělá za využití zásobníku. Řetězec popisující výraz procházíme pouze jednou zleva doprava. Když načteme číslo, dáme ho na výstup. Pokud načteme operátor nebo otevírající závorku, dáme je na zásobník. Ze zásobníku přechází operátory na výstup, pokud je „vytlačí“ uzavírající závorka nebo operátor s nižší prioritou (např.  $+$  vytlačí  $*$ ). Detaily necháváme čtenáři na domyšlení.

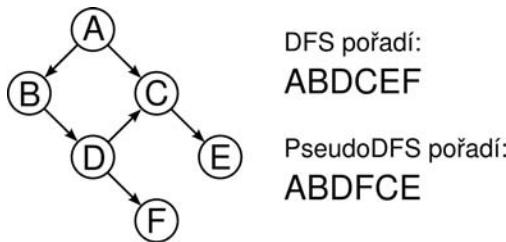
Pro hrátky s výrazy zmíníme jen hlavní myšlenku: maximalizace výrazu se řeší pomocí dynamického programování, resp. pomocí metody rozděl a panuj s ukládáním mezivýsledků. Příklad Pět pětek lze řešit prostě za využití hrubé síly vyzkoušením všech možných výrazů.

## Grafové algoritmy

Rozebereme pouze problém nerekurzivní implementace prohledávání do hloubky, který byl více nastíněn v komentáři. Nejdříve rozebereme typická chybná řešení. První z nich:

```
def PseudoDFS1(v):
    zasobnik = [ v ]
    navstiven[v] = True
    while len(zasobnik) > 0:
        v = zasobnik.pop()
        for w in reverse(naslednici[v]):
            if not navstiven[w]:
                zasobnik.append(w)
                navstiven[w] = True
```

Toto řešení zachovává lineární složitost, ovšem vede k chybnému pořadí procházení (viz obrázek 10.9). Tento rozdíl v pořadí procházení se může jevit jako nepodstatný detail a v mnoha případech opravdu není důležitý. Ovšem v některých aplikacích procházení do hloubky (např. detekce cyklu v orientovaném grafu) je přesné pořadí důležité a toto pozměněné pořadí způsobí



**Obrázek 10.9:** Ilustrace pořadí procházení pro chybnou verzi nerekurzivního procházení do hloubky

chybu v algoritmu. Chybě se můžeme pokusit zabránit tím, že přesuneme příkaz, kterým si poznamenáváme navštívení vrcholu:

```
def PseudoDFS2(v):
    zasobnik = [ v ]
    while len(zasobnik) > 0:
        v = zasobnik.pop()
        if navstiven[v]: continue
        navstiven[v] = True
        for w in reverse(naslednici[v]):
            if not navstiven[w]:
                zasobnik.append(w)
```

Toto řešení již prochází vrcholy ve správném pořadí, nesplňuje však podmínu efektivity. Velikost zásobníku v tomto případě totiž není nutně lineární, protože vrcholy se mohou v zásobníku opakovat. Dobře to je vidět, když si představíme, že program spouštíme na grafu, kde jsou všechny dvojice vrcholů propojeny hranou. V tomto případě bude velikost zásobníku až kvadratická.

Abychom dosáhli správného řešení, musíme se zamyslet nad tím, jak přesně funguje rekurzivní verze. Když se vynoříme z rekurzivního volání, „obnoví“ se nejen informace o aktuálním vrcholu, ale i o tom, kolik následníků jsme již zkusili navštívit. Proto si při iterativní verzi potřebujeme na zásobník ukládat nejen vrchol, ale i index posledního navštíveného následníka:

```
def DFS_nerekurzivne(g, v):
    zasobnik = [ (v, 0) ]
    while len(zasobnik) > 0:
        (v, k) = zasobnik.pop()
        navstiven[v] = True
        while k < len(g.naslednici[v]):
            w = g.naslednici[v][k]
            k += 1
            if not g.navstiven[w]:
                zasobnik.append( (v, k) )
                zasobnik.append( (w, 0) )
                break
```

# Literatura

- Aziz, A., Prakash, A. (2010). *Algorithms For Interviews*. CreateSpace.
- Bentley, J. (2000). *Programming Pearls*, second edition. Addison-Wesley.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2001). *Introduction To Algorithms*, second edition. The MIT Press.
- Hanzl, T., Pelánek, R., Výborný, O. (2007). *Šifry a hry s nimi*. Portál.
- Jančík, J., Kvoch, M. (1996). *Sbírka úloh z jazyka Pascal*. Kopp.
- Kučera, L. (2009). *Algovize aneb procházka krajinou algoritmů*. Univerzita Karlova v Praze, *algovision.org*.
- Libicher, I., Toepfer, P. (1992). *Od problému k algoritmu a programu: Sbírka řešení úloh z programování*. Grada.
- Pelánek, R. (2011a). *Jak to vyřešit?* Portál.
- Pelánek, R. (2011b). *Modelování a simulace komplexních systémů?* Nakladatelství Masarykovy univerzity.
- Papert, S. A. (1993). *Mindstorms: Children, Computers, And Powerful Ideas*, second edition. Basic Books.
- Skiena, S. S. (1998). *The Algorithm Design Manual*. Springer-Verlag.
- Töpfer, P. (1995). *Algoritmy a programovací techniky*. Prometheus.
- Wroblewski, P. (2004). *Algoritmy. Datové struktury a programovací techniky*. Computer Press.
- Russell, S., Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.
- Zelle, J. M. (2003). *Python Programming: An Introduction to Computer Science*. Franklin Beadle & Associates.

# Rejstřík

algoritmus

- A\* 98, 134
- Euclidův 33, 140
- geometrický 132
- Grahamův 58
- hladový 19, 59, 60, 118, 167
- Jarvisův 58
- k-means 132
- Kruskalův 128, 157
- minimax 165
- řadicí 123–125

aproximace 38

backtracking

- aplikace 91, 92, 95, 151, 159
- popis pojmu 20
- pseudokód 159
- ukázka průběhu 88

binární soustava 34, 141, 163

binární vyhledávání 102, 161

blízké body 132

bludiště

- číselné 76–77, 152
- generování 84–86, 156
- hledání cest 83–84, 156
- perfektní 86
- smyčka v 90

Buffonova jehla 39

buněčný automat 119

ciferný součet 25

- datové struktury 134
- dělitelnost 31–33
- Dilema vězně 100
- dláždění 131
- dynamické programování
  - aplikace 60, 123, 147, 169
  - konkrétní kód 167
  - popis pojmu 19
  - příklad 37, 108, 118, 163

ekvivalence 104

Eratosthenovo síto 32

faktoriál 25

Fibot 113

fraktál 48, 54, 132

Sierpiňského 48, 50–52, 80, 145

frekvenční analýza 72

fronta 98, 134, 156

graf 21, 127–129

grafika

- bitmapová 22, 31, 50, 52, 54, 131, 146

- textová 22, 44, 143

- vektorová 22, 45, 50, 85

- želví 45, 48, 135, 143, 144

Hádání čísla 102, 161–162

- Hanojské věže 79–80, 154  
heuristika  
    popis pojmu 20  
    příklad 160, 164  
    využití 92, 98, 166  
hra  
    akční 133  
    s nenulovým součtem 101, 114  
    Život 119–121  
hrubá síla  
    aplikace 32, 70, 78, 92, 106  
    popis pojmu 20  
    realizace 41  
  
interpret 135  
invariant cyklu 161  
  
Java 114  
jednosměrná funkce 101  
  
kalkulačka 125  
Kámen, nůžky, papír 100–101  
kódování 70  
kombinace 40, 142  
kombinační číslo 36, 41  
konvexní obal 57, 147  
  
L-systém 48, 52  
Levenshteinova vzdálenost 122  
logaritmus 102, 124, 132, 167  
Logik 105, 162  
  
Mandelbrotova množina 54, 146  
Media Computation 131  
Monte Carlo 38, 142  
  
náhodná procházka 30–31, 139  
Nim 107–108, 162–163  
  
Oběšenec 103  
objekty 21, 114  
Open data 135  
  
Pascalův trojúhelník 36–37, 50  
permutace 40  
piškvorky  
    jednorozměrné 110–111, 164  
    klasické 111–112, 165–167  
podvodník 104, 106  
Pokrývání mřížky 82–83, 154  
polyomina 91–92, 158–159  
posloupnost 26–27  
    Fibonacciho 35–36, 141  
    rekurentní 35, 54  
postfixová notace 127  
poziční soustava 34  
problém  
    8 dam 87, 158  
    Collatzův 27–30, 139  
    Frobeniusův 118  
    NP-úplný 119, 128, 156  
prohledávání  
    do hloubky 128, 129, 156, 169  
    do šířky 76, 78, 84, 97, 128  
prvočísla 31–33, 101  
Přelévání vody 77, 153  
přesmyčky 73–74, 151  
Python 11, 46, 137, 152  
  
rekurze  
    aplikace 127, 128, 136  
    ilustrace neefektivity 141  
    jednoduché cvičení 35, 48, 80,  
        83, 154, 155  
    nevzhodné využití 36, 37  
    popis pojmu 18  
    převod na iterativní verzi 169  
reprezentace  
    čísel 34  
    dat 70, 86, 92, 109, 150  
    slovníku 74  
robot 83, 112–115, 135  
Rozdělovačka 92

- 
- rozděl a panuj 18, 127, 132, 169
  - rozměňování mincí 117–119, 167
  - římské číslice 34
  - Skládačka 92
  - složitost 17, 124, 132
    - pseudo-polynomiální 167
  - Sokoban 97–98, 160–161
  - statistika 64, 72, 135
  - stavový prostor 21, 78, 80, 97, 165
  - strom
    - kD 132
    - prefixový 74
    - vyvážený 134
  - Sudoku 94–96, 159–160
  - SVG 23, 85, 131
  - šachové figurky 86–89
  - šachovnice 44, 83, 86–88, 154
  - šifra
    - Caesarova 70
  - rozlomení 70–72
  - substituční 69
  - transpoziční 66, 70, 149
  - Vigenèrova 70, 150, 151
  - teorie her 101
  - Tetris 108–110, 163–164
  - tetromina 91, 108
  - triangulace 59–61, 147–149
  - trie 74
  - Ulamova spirála 31
  - umělá inteligence 27, 106, 109, 112, 134
  - variace 40, 106
  - Voroného diagram 132
  - vyhodnocování výrazů 125, 169
  - zásobník 129, 134, 136, 169, 170