skripta
KLOG

Jonathan L. Verner

Úvod do programování

část II: Algoritmy

Jonathan L. Verner, PhD.
Katedra Logiky
Filozofická fakulta UK
Palachovo náměstí 2
116 18 Praha 1

© Jonathan L. Verner, 2015

Tato učebnice slouží jako skripta k letnímu semestru přednášky *Úvod do programování*, vypsané pod kódem ALG110006 na Katedře logiky FF UK v letech 2011–2015. Jejich příprava byla podpořena projektem OPVK CZ.1.07/2.2.00/28.0216 *Logika: systémový rámec rozvoje oboru v ČR a koncepce logických propedeutik pro mezioborová studia*

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Úvod do programování

část II: Algoritmy

Obsah

1	**Porovnávání algoritmů, Eukleidův algoritmus**	**9**
2	**Základní datové struktury**	**17**
3	**Vyhledávání, třídící algoritmy**	**33**
4	**Analýza jazyka**	**47**
5	**Grafové algoritmy**	**65**
6	**Složitost podruhé**	**77**
Lidé		**83**

Úvod

V situaci, kdy je k dispozici množství velmi dobrých úvodních i pokročilých učebnic programování se může zdát zbytečné psát učebnici novou. Chtěl bych proto hned na úvod podotknout, že při psaní těchto skript jsem neměl ambici napsat "nejlepší" učebnici; dokonce jsem ani neměl ambici napsat učebnici "originální". Skripta vznikala v podstatě jako příprava na letní semestr přednášky "Úvod do programování" pro první ročník studentů logiky na Filozofické fakultě UK. Jejich hlavním přínosem je, že pokrývají — víceméně přesně — přednesenou látku. Přestože jsou skripta cílena zejména na mé studenty, doufám, že i případnému jinému čtenáři budou k prospěchu, vyprovokují v něm zájem o informatiku a třeba ho povedou k další četbě jistě mnohem lepších textů. Z nepřeberného výběru bych zde chtěl vyzdvihnout dvě knížky — skvělou českou knížku P. Töpfera: *Algoritmy a programovací techniky* ([To95]) a klasiku oboru *The Art of Computer Programming* od D. Knutha.

Na závěr úvodu patří poděkování. Chtěl bych předně poděkovat své ženě Anše za cenné připomínky (a její lásku, ale to je jiný příběh. . .), Tomáši Lavičkovi za upozornění na chyby v přiložených algoritmech a konečně také Evropské unii, která v rámci projektu OPVK CZ.1.07/2.2.00/28.0216 *Logika: systémový rámec rozvoje oboru v ČR a koncepce logických propedeutik pro mezioborová studia* podpořila psaní těchto skript. O tento projekt se na naší katedře skvěle starali Marta Bílková a Michal Dančák. A nakonec bych chtěl poděkovat tomu, který nás stvořil a vybavil schopností tvořit světy virtuální.

Autor

Kapitola 1

Porovnávání algoritmů, Eukleidův algoritmus

Mým cílem v zimním semestru bylo Vás naučit psát jednoduché programy v Pythonu a mít nějakou konkrétní představu o tom, co to prakticky znamená "programovat". V letním semestru se podíváme na programování z teoretičtějšího hlediska. Naším základním tématem budou různé algoritmy — standardní (ale i nestandardní) postupy řešení různých problémů, se kterými se jako programátoři můžete potkat.

Pojďme se nyní společně podívat na jednu skupinu takových problémů. Jedná se o problémy, které původně pocházejí z matematické disciplíny zvané Teorie čísel. V praxi jsou úzce provázány s moderními šifrovacími metodami (RSA, Elektronický podpis, ...). Asi nejvíce fascinujícím problémem je následující úloha:

1.1 Úloha. Nalezněte rozklad daného čísla n na součin prvočísel.

s ní úzce souvisí úloha, ve které máme rozhodnout, zda číslo je, či není prvočíslem:

1.2 Úloha. Pro dané číslo n rozhodněte, zda je prvočíslem nebo není.

K této úloze se možná, pokud zbyde čas, vrátíme v nějaké pozdější přednášce. Dnes se podíváme na jednodušší úlohu, která bývá jednou ze základních operací prováděných při řešení složitějších problémů v teorii čísel.

1.3 Úloha. Pro daná čísla n, m zjistěte jejich největšího společného dělitele.

Pokud bychom chtěli úlohu řešit na počítači, pravděpodobně by nás jako první napadlo prostě projít všechna čísla od 1 do $\min\{n, m\}$ a najít největší číslo, které dělí obě čísla najednou. V Pythonu by takový algoritmus mohl vypadat třeba takto:

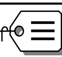
```
1 def gcdN(n,m):
2     ret = 1
3     for d in range(2,n+1):
4         if d % n == 0 and d % m == 0:
5             ret = d
6     return ret
```

Možná, že někteří z Vás se s touto úlohou setkali na střední škole. Tam jste se mohli dozvědět o postupu, který popsal již Euklides (300 př. Kr.): vezmeme větší z obou čísel a spočítáme si zbytek po dělení menším z obou čísel. Pokud je zbytek 0, jsme hotovi (nsd je menší z obou čísel). V opačném případě provedeme totéž, tentokrát s menším číslem a spočítaným zbytkem. V Pythonu lze pomocí rekurze zapsat tento postup následovně:

```
1 def gcdE(n,m):
2     if n > m:
3         n,m = m,n
4     z = m % n
5     if z == 0:
6         return n
7     return gcdE(z,n)
```

Pokud bychom se chtěli vyhnout rekurzi, mohli by stejný algoritmus vypadat například jako výpisu 1.4.

Algoritmus 1.4 Eukleidův algoritmus (nerekurzivní verze)



```
1  #coding: utf-8
2  def gcdE_nre(n,m):
3      if n > m:
4          n,m = m,n
5      z = m % n
6
7      while not (z == 0):
8          m = n
9          n = z
10         z = m % n
11
12     return n
```

U tohoto algoritmu nemusí být na první pohled jasné, jak funguje. Dokonce není

jasné, zda vůbec skončí a pokud ano, zda dá správnou odpověď. Jednou možností, jak zjistit, jestli algoritmus funguje, je vyzkoušet ho na několika vstupech a zkontrolovat odpovědi. To nám však nedá odpověď s naprostou jistotou. Abychom si byli naprosto jistí, musíme algoritmus pochopit a přesvědčit se, že opravdu funguje. O to se nyní pokusíme.

Nejprve si ukážeme, že algoritmus opravdu skončí. První, rekurzivní verze algoritmu, nám dává i návod: u rekurzivních algoritmů dokážeme, že skončí, pokud se přesvědčíme, že skončí v nějakém základním případě (případech) a všechny ostatní vstupy postupně převede na tento základní případ(y). U našeho algoritmu bude základním případem situace, kdy n dělí m (což budeme značit $n|m$). Ukážeme, že rekurzivní volání postupně převede každý vstup na tuto základní situaci. Označme si $n_0 = n$ a $m_0 = m$ počáteční vstupy a n_k, m_k hodnoty proměnných d, n při k-tém průchodu řádkem 7 (t.j. n_k a m_k budou hodnoty parametrů při k-tém (rekurzivním) volání funkce `gcdE`).

Všimněme si, že $0 \leq n_{k+1} = m_k \pmod{n_k} < n_k$. Tedy posloupnost n_k je zdola omezená nulou a ostře klesá. Tudíž musí být konečná, tedy program musí skončit. Tudíž algoritmus provede řádek 7 pouze konečněkrát a tedy v konečném čase skončí.

Když už víme, že algoritmus skončí, musíme se ještě přesvědčit, že dá správnou odpověď. Opět využijeme rekurzivní verzi. Pokud algoritmus skončí v základním případě (řádek 6), zjevně dá správnou odpověď. Pokud ještě ukážeme, že i na řádku 7 získáme správnou odpověť, t.j. že `gcdE(n,m) = gcd(z,n)`, budeme hotovi. Poslední rovnost však plyne z následujících jednoduchých pozorování:

- Pokud $d|n$ a $d|m$ pro nějaké $m = kn + z$, pak i $d|z$.
- Pokud naopak $d|n$ a $d|z$, pak d dělí každé m tvaru $kn + z$.
- Tedy, je-li $m = kn + z$ pak každý společný dělitel čísel n, m je i společným dělitelem čísel n, z a obráceně.

Tím jsme odpověděli na základní otázku, kterou si při seznamování s novým algoritmem musíme položit — zda pracuje správně. Jak jsme viděli, i v takto jednoduchém případě jde o poměrně komplikovaný proces. Mohlo by Vás napadnout, zda by nešlo tento proces nějak automatizovat. To by se hodilo zejména ve složitějších případech, kdy je třeba nejen zkontrolovat správnost algoritmu, ale zkontrolovat i jeho bezchybnou implementaci. Obecně je odpověď , jak si ukážeme v poslední přednášce, naneštěstí záporná. Nicméně v některých konkrétních případech to lze. Tímto problémem se zabývá oblast informatiky zvaná formální verifikace. My se

však nyní obrátíme k dalším otázkám, které je třeba položit. Uvedli jsme si dva algoritmy pro stejnou úlohu a je přirozené se ptát, zda jsou oba stejně "dobré". Tomu se budeme věnovat v následujícím oddíle.

Porovnávání algoritmů

Proto, abychom oba výše popsané algoritmy mohli porovnávat, je třeba zvolit nějaké kritérium. Kritérií může být samozřejmě mnoho, následující seznam zdaleka není vyčerpávající:

- rychlost
- paměťová náročnost
- jednoduchost, srozumitelnost
- elegance
- energetická náročnost (opravdu !)

V našem kurzu se budeme zaměřovat na první dva body (a zejména na první bod), ačkoliv jsou situace, kdy mnohem důležitějším kritériem může být třeba jednoduchost nebo energetická náročnost.

Pokusme se tedy oba algoritmy porovnat z hlediska rychlosti. Asi nejjednodušším způsobem, jak to provést, je měřit čas T od puštění programu do jeho skončení. To však má svá úskalí:

- tento čas bude záviset na rychlosti daného počítače, jeho zatíženosti, operačním systému, překladači, interpretu ...
- je to experimentální veličina o které lze těžko matematicky něco dokazovat

Raději tedy tuto základní ideu trochu upravíme — úvahy abstrahujeme od konkrétního počítače a konkrétní implementace. Nebudeme měřit čas, ale určíme si množinu základních operací (přiřazení, testování booleovské podmínky (`if`), aritmetická operace) a budeme počítat počet operací, které program při svém běhu vykoná. Označme si toto číslo Op. Je zjevné, že toto číslo bude záviset na vstupních datech. Pokud budeme počítat největšího společného dělitele čísel 1235468127495138 a $21598426112645674\,86$ budeme muset provést více operací než při počítání společného dělitele čísel 14 a 35. Tedy Op bude ve skutečnosti funkce vstupních dat.

Problém s funkcí Op je ten, že mnohdy je obtížné jí vyjádřit nějakým jednoduchým vzorečkem. Často pro nás proto budou důležité spodní a horní odhady této funkce. Zkusme nyní spočítat Op_N a Op_E dvou výše uvedených algoritmů. Začněme naivním algoritmem. Ten nejprve provede jedno přiřazení na řádku 2. Dále se provede $n-1$-krát cyklus, tedy $n-1$-krát se bude testovat podmínka na řádku 4. V části těchto případů se navíc provede přiřazení na řádce 5. Tedy máme

$$Op_N(n, m) = 1 + (n-1) + ?$$

Zbývá spočítat člen $?$, t.j. v kolika případech se provede řádek 5. Pokud jsou n, m nesoudělná, neprovede se nikdy. Dolním odhadem tedy je 0. Pokud by se řádek 5 provedl pokaždé, pak by se provedl $n-1$-krát, což je tedy horní odhad. Přesné číslo závisí na n a m a nelze je jednoduše vyjádřit. Mohli bychom se pokusit o statistickou analýzu, nicméně neuděláme to (zájemce ji může nalézet v Tao'c P, [TAOCP2]). Můžeme tedy učinit následující závěr:

$$n = 1 + (n-1) \leq Op_N(n, m) \leq 1 + 2(n-1) = 2n - 1$$

Podívejme se nyní na druhý algoritmus. Zde je situace o trochu složitější. Při každém volání funkce `gcdE` dochází ke dvěma porovnáním (2. a 5. řádek), jedné aritmetické operaci (4. řádek) a při úplně prvním volání ještě ke dvěma přiřazením (3. řádek, uvědomte si, že jsou to přiřazení dvě a že se tento řádek v dalších voláních nikdy neprovede). Označme si tedy počet volání funkce `gcdE` číslem c. Pak můžeme psát:

$$Op_E(n, m) = 2 + 3c$$

Jak spočíst číslo c? Opět budeme pouze odhadovat. Dolní odhad je 1 v případě, že n dělí m. Jednoduchý horní odhad dostaneme z úvah, které jsme prováděli při důkazu správnosti algoritmu. Zjistili jsme, že při každém rekurzivním volání je první argument ostře menší než při předchozím. Funkce `gcdE` se tedy zavolá nejvíce n-krát. Tedy máme odhad

$$5 = 2 + 3 \leq Op_E(n, m) \leq 2 + 3n$$

Na základě tohoto horního odhadu bychom mohli usoudit, že první "naivní" algoritmus je ve skutečnosti dokonce lepší než ten složitější Eukleidův. Musíme si však uvědomit, že skutečná hodnota $Op_E(n, m)$ může být (a, jak za chvíli uvidíme, opravdu je) mnohem menší než výše uvedený odhad. K odvození lepšího horního

odhadu zavedeme následující značení. Nechť n_k, m_k je hodnota parametrů při k-tém volání funkce `gcdE`. Pak platí následující:

- $m_{k+1} = n_k$
- $n_{k+1} \leq m_k/2$

První bod je zřejmý uvažujme nad druhým. Vyjděme z toho, že $n_{k+1} = m_k \bmod n_k$. Je-li $m_k >= 2n_k$, pak $m_k/2 >= n_k$, zatímco zbytek po dělení n_k je vždy $< n_k$, tedy v tomto případě bod platí. Je-li naopak $m_k \in (n_k, 2n_k)$ [uvědomme si, že $m_k > n_k$], pak m_k div $n_k = 1$, tedy m_k mod $n_k = m_k - n_k$. Nyní opět jednoduše $n_{k+1} <= m_k/2$.

Kombinací prvních dvou bodů máme

- $n_{k+2} \leq n_k/2$

a opakovaným aplikováním tohoto bodu dostaneme

- $n_{2k} \leq n_k/2^k$

Zjistili jsme tedy, že velikost prvního parametru se nejen ostře snižuje, ale dokonce se po dvou voláních sníží na polovinu původní hodnoty. Tedy 0 (resp. 1) dosáhne po $2\lceil \log_2 n_0 \rceil$ (jednoduchým dosazením do předchozího bodu zjistíme, že pro $c = 2\lceil \log_2 n_0 \rceil$ máme $n_c \leq 1$), tedy veličinu c můžeme shora odhadnout pomocí tohoto výrazu. Celkově tedy dostáváme

$$Op_E(n,m) \leq 2 + 3(2\lceil \log_2 n \rceil) \leq 5 + 6\log_2 n$$

Zůstaňme ještě chvíli u hodnot veličiny Op. Tato veličina byla zavedená tak, aby dostatečně abstrahovala od času, který bude program běžet. Na druhou stranu od ní stále intuitivně očekáváme, že bude s tímto časem nějak blízce souviset. Tato souvislost bude samozřejmě záviset na zvolených "základních operacích", programovacím jazyce a rychlosti počítače. Obecně však není nerozumné předpokládat, že tento vztah bude "zhruba lineární", t.j. pro každou konkrétní situaci budou existovat nějaké konstanty K, k (vyjadřující např. rychlost počítače a podobné okolnosti) takové, že vztah mezi Op a T půjde vyjádřit pomocí nerovností

$$T/K - k \leq Op \leq KT - k$$

Protože veličiny T a Op typicky závisí na vstupních datech (t.j. jsou funkcemi nějaké proměnné n) a pro malá n je tato nerovnost často nezajímavá, budeme se zajímat o limitní formu této nerovnosti. Zaveďme nyní následující značení. Jsou-li f, g funkce proměnné n budeme psát

$$f(n) = O(g(n)) \iff (\exists K, n_0)(\forall n > n_0)(f(n) \leq Kg(n))$$

a

$$f(n) = \Omega(g(n)) \iff (\exists K, n_0)(\forall n > n_0)(g(n)/K \leq f(n))$$

První definice říká, že pro dostatečně velká n je funkce f shora omezená funkcí g až na nějakou multiplikativní konstantu. Podobně druhá definice říká, že je tato funkce omezená zdola až na multiplikativní konstantu. O nějakém algoritmu A pracujícím s daty reprezentovanými proměnnou n řekneme, že má složitost $O(g(n))$, pokud $Op_A(n) = O(g(n))$ *a zároveň* $Op_A(n) = \Omega(g(n))$. Všimněme si, že v takovém případě bude i $T_A(n) = O(g(n))$ a $T_A(n) = \Omega(g(n))$, tedy funkce g v takovém případě věrně popisuje reálné chování algoritmu. Funkce g, se kterými se setkáme nejčastěji, jsou následující: $1, n, \log_2 n, n, n \log_2 n, n^2, 2^n$. Těmto funkcím odpovídá i klasifikace algoritmů na algoritmy v konstantním čase ($O(1)$), algoritmy logaritmické ($O(\log_2 n)$), algoritmy lineární ($O(n)$), algoritmy kvadratické ($O(n^2)$) resp. polynomiální ($O(P(n))$, kde P je nějaký polynom) a algoritmy exponenciální $O(2^n)$. Algoritmy, které mají nejvýše polynomiální složitost, se obecně považují za efektivní (záleží samozřejmě na konkrétních okolnostech), zatímco algoritmy exponenciální jsou vesměs nepoužitelné (uvědomme si, že pokud exponenciální algoritmus na některém počítači dokáže zpracovat data nejvýše velikosti n, na dvakrát rychlejším počítači zpracuje data pouze o jedna větší velikosti $n + 1$).

Když jsme nyní zavedli klasifikaci algoritmů vidíme, že Eukleidův algoritmus má logaritmickou složitost, zatímco naivní algoritmus má složitost lineární.

Kapitola 2

Základní datové struktury

V této kapitole se budeme blíže zabývat různými způsoby, jak v paměti ukládat data a jak s nimi pracovat. Naučíme se pracovat s frontou a zásobníkem a ukážeme si zajímavou strukturu, které se říká halda.

Fronta

Začněme následující, možná trochu šroubovanou, situací. Ve vašem městě podniká populární zelinář. Protože prodává kvalitní ovoce a zeleninu, které sám nakupuje od farmářů, podnikání se mu daří a jeho malou samoobsluhu navštěvuje stále více a více lidí. S tím však přichází i nepříjemný fenomén — fronty. Zelinář je nadšený do moderních technologií a rozhodne se problém řešit za pomoci elektronického zákaznického odbavovacího systému. Představuje si to takto: zákazník si naloží věci do košíku a pak si u jednoho ze stojanů, roztroušených po obchodě, vyzvedne pořadové číslo. Se svým košíkem se pohodlně usadí do k tomu účelu přistavených křesel a čeká, než ho zákaznický systém vyvolá. Protože zelinář počítačům příliš nerozumí, rozhodl se Vás najmout, abyste mu naprogramovali software, který bude celý systém řídit:

2.1 Problém. Naprogramujte software pro zákaznický odbavovací systém.

Ponechme stranou technické detaily tisku čísel a podobně a soustřeďme se na jádro problému. Budeme potřebovat dvě funkce. Funkci `generujListek`, která zákazníkovi u stojanu přidělí číslo a funkci `naRade`, kterou bude volat zelinář, když bude

chtít obsloužit dalšího zákazníka. V podstatě jde o to, naprogramovat takovou elektronickou frontu. To můžeme jednoduše udělat třeba takto: budeme si prostě v nějaké proměnné, třeba `last`, uchovávat poslední číslo, které jsme nějakému zákazníku přidělili a v druhé proměnné `current`, si budeme uchovávat číslo zákazníka, který je právě obsluhován. V Pythonu pak bude program vypadat třeba takto:

```
1 current=0
2 last=0
3
4 def generujListek():
5     global last
6     last = last + 1
7     return last
8
9 def naRade():
10     global last, current
11     if current == last:
12         return None
13     else:
14         current = current + 1
15     return current
```

Představte si nyní, že by se požadavky trochu změnily. Čísla jsou příliš anonymní a zelinář rád lidi oslovuje jménem. Proto by chtěl, aby uživatel do systému mohl zadat své jméno. Systém ho pak bude vyvolávat jménem. Náš výchozí program jednoduše upravíme:

```
1  current=0
2  last=0
3  jmena=[]
4
5  def generujListek( jmeno ):
6      global last,jmena
7      last = last + 1
8      jmena.append(jmeno)
9
10 def naRade():
11     global last,jmena, current
12     if current == last:
13         return None
14     else:
15         current = current + 1
16     return jmena[current-1]
```

Problém s tímto přístupem je ten, že jména se nám budou v paměti kupit a kupit a kupit . . . Po nějakém čase dojde paměť, odbavovací systém spadne a v obchodě se strhne mela. To je nepříjemné. Bude proto lepší program trochu přepsat. Tentokrát budeme používat pomocnou proměnnou `fronta`, kde budeme mít seznam jmen. Každého nového zákazníka přidáme na konec tohoto seznamu a zákazníky, které budeme odbavovat, budeme naopak brát ze začátku. Program tedy vypadá takto:

```
1  fronta=[]
2
3  def generujListek( jmeno ):
4      global fronta
5      fronta.append(jmeno)
6
7  def naRade():
8      global fronta
9      if len(fronta) == 0:
10         return None
11     return fronta.pop(0)
```

Všimněte si funkce `pop`, se kterou jsme se ještě nesetkali. Volání `seznam.pop(n)` vrátí n-tý prvek seznamu `seznam` a pak ho ze seznamu smaže. Kdybychom ji chtěli sami naprogramovat, vypadala by třeba takto:

```
1 def pop(seznam, index):
2     ret = seznam[index]
3     del seznam[index]
4     return ret
```

Zkusme se nyní podívat na složitost našeho programu. V podstatě jde o to, jak jsou složité funkce `append` a funkce `pop`.

Jejich rychlost závisí na tom, jak Python uvnitř realizuje datový typ *list*. Pythonovský *list* je ve skutečnosti realizován jako něco, čemu se v jiných programovacích jazycích říká *pole*. Když Pythonu řeknete, aby vytvořil list o n-prvcích, Python si na to v paměti vyhradí n po sobě jdoucích “chlívečků”. Když chceme přistoupit k i-tému prvku pole, je třeba zjistit, kde v paměti se nachází odpovídající chlíveček. Ale to je velmi jednoduché, stačí vědět, kde pole začíná a pak k tomuto začátku přičíst i-krát velikost chlívečku. Zkracování pole (funkce `pop(-1), pop(0)`) je také jednoduché — Python prostě poslední (první) “chlíveček” uvolní k jinému použití. Trochu složitější je odebírání k-tého prvku, kde $0 < k <$`len(pole)`. V takovém případě je třeba nejprve posunout obsah každého chlívečku za k-tým o jeden chlíveček doleva — tedy `(len(pole)-k)` operací — a pak uvolnit poslední. Nejsložitější však je prodlužování pole, t.j. funkce `append`. Pokud totiž chceme seznam zvětšit, můžou nastat komplikace. Nejjednodušší by bylo prostě vyhradit další “chlívečky” za koncem pole. Jenže co když už jsou tyto chlívečky obsazeny? V takovém případě Python musí najít v paměti někde jinde dostatek místa pro všechny chlívečky nového pole a staré pole sem překopírovat. V takové situaci tedy bude zvětšování pole mnohem náročnější.

```
>>> seznam = ['A','B','C',5]
>>> seznam.append('Q')
```

Python problémy se zvětšováním seznamů částečně obchází pomocí elegantního triku. Když totiž vytvoříte nové pole, Python si ve skutečnosti vyhradí chlívečků víc, aby měl nějakou rezervu na zvětšování. Přidávat nové prvky je pak jednoduché do té doby, než dojde tato rezerva. Když mu rezerva dojde, musí chtě-nechtě vyhradit místo nové a pole kopírovat. Nicméně v tuto chvíli si opět vytvoří rezervu a pro jistotu si jí vytvoří 2-krát tak velkou jako na počátku. Když má naopak prvek z konce pole smazat, pouze si poznačí, že pole je kratší a místo toho, aby místo na konci pole uvolnil pro jiné použití, zvětší o něj rezervu. Ačkoliv to nemusí být jasné na první pohled, je tento postup v praxi často mnohem výhodnější. Pokud například do pole v průměru stejně často prvky přidáváme jako odebíráme, málokdy nám rezerva dojde a většina operací bude velmi rychlá. Tuto "výhodnost" lze matematicky analyzovat — podobně jako jsme to udělali se složitostí algoritmů — pomocí tzv. "amortizované složitosti" (více viz např. [Ta85]), nicméně my to nyní dělat nebudeme a vrátíme se k našemu zelináři.

Čekat v křesle je sice pohodlnější než stát ve frontě, nicméně někteří zákazníci by si klidně připlatili, aby nemuseli čekat vůbec. Zelinář proto za Vámi přišel s tím, abyste mu systém upravili tak, aby si lidé mohli připlatit za přednostní odbavení. Jedno možné řešení spočívá v tom, že zavedeme fronty dvě. Frontu přednostní a frontu obyčejnou. Zákazník si při volbě čísla může připlatit, aby byl zařazen do přednostní fronty. K pokladně pak budou vyvoláváni nejprve lidé z fronty přednostní a teprve v okamžiku, kdy je tato fronta prázdná, přijdou na řadu lidé ve frontě obyčejné. Implementace je jednoduchá:

```
 1 f_obyc=[]
 2 f_predn=[]
 3
 4 def generujListek( jmeno, prednostni = False ):
 5     global f_obyc, f_predn
 6     if prednostni:
 7         f_predn.append(jmeno)
 8     else:
 9         f_obyc.append(jmeno)
10
11 def naRade():
12     global f_obyc, f_predn
13     if len(f_predn) > 0:
14         return f_predn.pop(0)
15     elif len(f_obyc) > 0:
16         return f_obyc.pop(0)
17     else:
18         return None
```

Řešení je elegantní a celkem vyhovující. Protože ale nastavená cena přednostní fronty není příliš vysoká, musí se i v ní často dlouho čekat. Někteří movitější zákazníci by si klidně připlatili mnohem více, jen aby nemuseli čekat. Šlo by to řešit zavedením superpřednostní fronty, případně i hyperpřednostní atd ... Vás však napadne mnohem lepší řešení. Fronta bude jen jedna, ale při vyzvednutí lístku bude mít každý možnost zaplatit určitou částku dle vlastní volby. Zákazníci pak budou odbavováni v pořadí podle obnosu, který zaplatili. Implementace bude tentokrát trochu složitější:

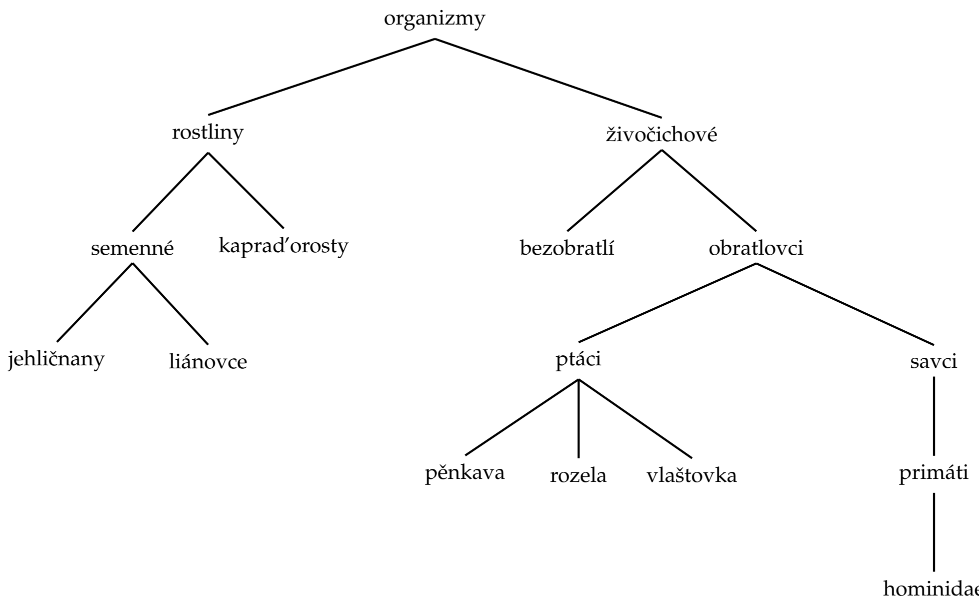
```
 1 fronta = []
 2
 3 def generujListek( jmeno, castka ):
 4     global fronta
 5     fronta.append( [jmeno, castka] )
 6
 7
 8 def naRade():
 9     global fronta
10
11     # Pokud je fronta prazdna, vrat None
12     if len(fronta) < 1:
13       return None
14
15     # Najdi zakaznika s maximalni castkou
16     # (zakaznik je vzdy dvojice [ ID, castka ] )
17     max_jmeno, max_castka = zakaznici[0]
18     max_pos = 0
19     pos     = 0
20
21     for (jmeno, castka) in zakaznici:
22         if castka > max_castka:
23             max_castka = castka
24             max_jmeno = jmeno
25             max_pos = pos
26         pos = pos + 1
27
28     del zakaznici[pos]
29     return max_jmeno
```

Pojďme se podívat na složitost tohoto řešení. Funkce `generujListek` provede pouze jednu operaci append na konec pole, tudíž má konstantní (amortizovanou) složitost ($O(1)$), t.j. nezávisí na velikosti fronty. Funkce `naRade` provede nejprve až

5 inicializačních operací (řádky 10 a 15–18) a pak musí projít celou frontu (řádek 20) a s každým prvek provést až 2-5 operací (řádky 21–25). Má tedy v nejhorším případě složitost $5 + 5n = O(n)$, kde n je délka fronty. Vzhledem k tomu, že pro $n > 1$ funkce nikdy neskončí řádkem 11, jednoduše zjistíme, že v nejlepším případě má složitost $5 + 2n = O(n)$, tedy opět lineární v délce fronty. Ukážeme si poněkud důmyslnější řešení, které za cenu mírného zpomalení funkce `generujListek` výrazně urychlí funkci `naRade`. Trik bude spočívat v tom, že si budeme zákazníky ukládat takovým způsobem, abychom velmi rychle uměli vybrat zákazníka s nejvyšší zaplacenou částkou. Budeme k tomu používat tzv. haldu (angl. heap), která je speciálním případem stromu. Pojd'me se tedy nejprve podívat na (matematické) stromy.

Stromy

Matematický pojem stromu je dobře ilustrován následujícím obrázkem znázorňujícím dělení živých organizmů:



Schematicky ho můžeme popsat takto: na obrázku je množina kategorií (obratlovci, savci, ...), které se ve stromové terminologii často říká množina uzlů. Mezi některými kategoriemi pak vede čára (hrana), znázorňující vztah kategorie-podkategorie (obratlovci–savci, ...), ve stromové terminologii se tomuto vztahu často říká vztah rodič–dítě (motivace pochází z rodokmenů, které se také často znázorňují pomocí stromů). Zavedeme si ještě několik pomocných pojmů:

2.2 Definice. *kořen* je uzel, který je úplně na vrcholu, t.j. čáry vedou pouze z něj, žádná čára nevede do něj. Na našem obrázku je kořenem kategorie "organizmy". Naopak *list* je uzel, ze kterého nevede žádná čára (např. "vlaštovka", ...) a uzly, které nejsou listy (včetně kořene) budeme souhrnně nazývat *vnitřními* uzly. Pokud má uzel alespoň dvě děti (t.j. vedou z něj dvě čáry), řekneme, že se *větví*, resp. že je to *větvící* uzel. Pokud mají všechny uzly nejvýše dvě děti, říkáme, že strom je binární (podobně definujeme ternární strom atd.). *Cestou* ve stromu rozumíme posloupnost uzlů, které jsou postupně mezi sebou pospojované hranami vedoucími vždy od rodiče k dítěti. Cestu od kořene k nějakému listu nazveme *větví*. *Hloubkou* stromu rozumíme délku nejdelší větve. *Hladinou* ve stromě rozumíme množinu uzlů, které mají stejnou hloubku. Uzel A je *potomkem* uzlu B, pokud z B do A vede cesta. V takovém případě říkáme též, že uzel B je *předkem* uzlu A.

Poznámka. Formálně lze tuto situaci modelovat různými způsoby. Například v teorii množin je strom definován jako částečně uspořádaná množina $(T, \leq)$, kde prvky $t \in T$ odpovídají uzlům a pro každý uzel $t \in T$ je uspořádání $\leq$ zúžené na množinu $\{s \in T : s \leq t\}$ jeho předchůdců dobré. Jiný způsob, který pro nás bude důležitější, reprezentuje strom jako speciální případ grafu, tzv. orientovaný graf bez cyků, anglicky DAG (directed acyclic graph). Grafům se budeme později věnovat podrobněji.

Pojďme se nyní podívat, jak je možné tuto situaci reprezentovat v Pythonu. Bude pro nás důležité, že v každém uzlu chceme uchovávat nějaká data — třeba pojmenování toho uzlu. Nejjednodušší bude reprezentovat uzel jako dvojici `u = (data, deti)`, kde `deti` je seznam uzlů, které jsou dětmi uzlu `u`. Zde je názorný příklad:

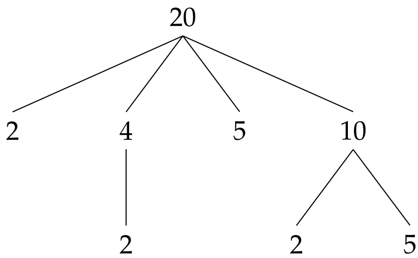
```
>>> T = ('Koren',[
                 ('List A',[]),
                 ('B',[
                      ('List C',[]),
                      ('List D',[])
                      ])
                 ])
```

Koren
List A
B
List C
List D

Ukažme si na dvou jednoduchých příkladech, jak s takovými stromy pracovat. Často bude velmi přirozené používat rekurzi. Například procedura, která vrátí seznam jmen všech listů může vypadat takto:

```python
def listy(T):
  ret = []
  jmeno, deti = T
  if len(deti) == 0:
    # T je list (nemá žádné děti), vrátíme jméno T
    return [jmeno]
  else:
    # Projdeme všechny děti a do našeho seznamu přidáme listy,
    # které jsou jejich potomkem
    for d in deti:
      ret = ret + listy(d)
  return ret
```

Zkusme si nyní nějaký strom vytvořit. Budeme chtít naprogramovat funkci, která dostane jako parametr číslo a vytvoří následující strom. V každém uzlu stromu bude uloženo nějaké číslo, v kořeni bude uloženo číslo dané vstupním parametrem. Děti každého uzlu budou odpovídat všem dělitelům > 1, listy tedy budou odpovídat prvočíselným dělitelům. Například pro číslo dvacet budeme chtí vytvořit následující strom:

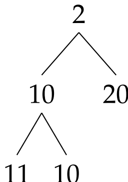


V Pythonu může naše funkce vypadat třeba takto:

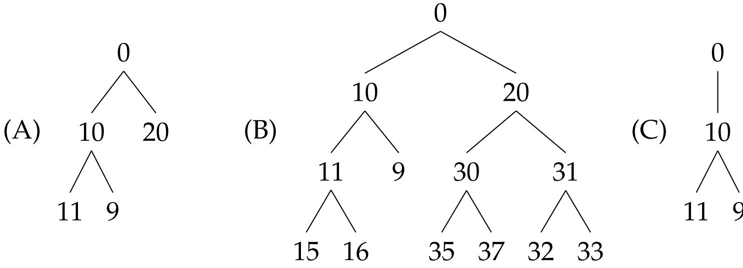
```
1 def delitele(n):
2   ret = []
3   for d in range(2,n/2+1):
4     if n % d == 0:
5       ret.append(d)
6   return ret
7
8 def strom_delitelu(n):
9   delit = delitele(n):
10  deti = []
11  for d in delit:
12    deti.append(strom_delitelu(d))
13  return (n,deti)
```

Haldy

Vraťme se nyní k našemu zelináři. Řekli jsme, že budeme chtít ukládat zákazníky tak, abychom uměli velmi rychle vybírat zákazníka s nejvyšší zaplacenou částkou a podobně rychle zákazníky i přidávat. Řešení spočívá v tom, že je budeme ukládat do stromové struktury, které se říká halda. Halda je speciální případ binárního stromu, ve kterém je v každém uzlu mimo ostatních dat ještě navíc uloženo nějaké číslo. Jakožto strom je to strom, ve kterém se každý uzel, který není listem, větví a všechny listy jsou na poslední nebo předposlední hladině s tím, že listy na poslední hladině jsou "co nejvíce vlevo". Navíc pro každý uzel platí, že číslo, které je v něm uloženo, je menší než čísla uložená ve všech jeho potomcích. Příkladem haldy je například následující strom:



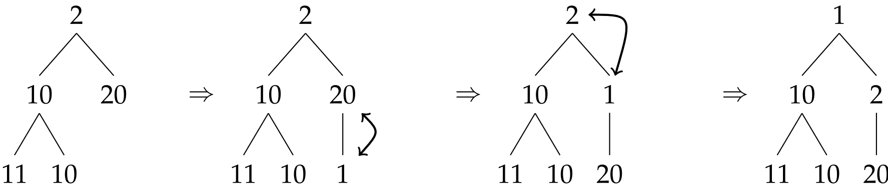
Naopak ani jeden z následujících stromů haldou není.



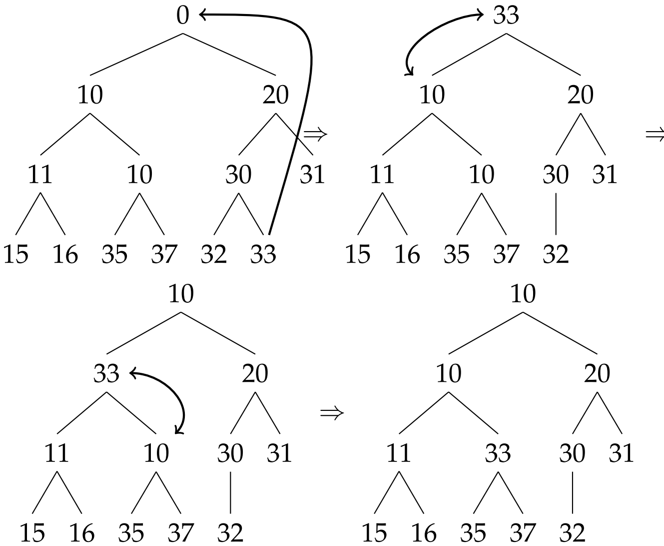
Ve stromu (A) je porušena podmína, že uzly jsou menší než jejich potomci ($10 \not\leq 9$), ve stromu (B) je porušena podmínka, že listy na poslední hladině mají být úplně vlevo a ve stromu (C) je porušena podmínka, že každý uzel, který není listem, se musí větvit.

Prvky se do haldy přidávají tak, že se vytvoří nový list a do něj se prvek vloží. Tím může dojít k porušení podmínky, že uzly jsou menší než potomci. Proto musíme zkontrolovat, zda je vložený prvek větší než jeho rodič. Pokud není, s rodičem ho vyměníme a musíme dále zkontrolovat rodiče s jeho rodičem. Pokud je vše O.K., skončíme, jinak pokračujeme v prohazovaní dokud se nedostaneme do kořene, který žádného rodiče nemá.

Příklad přidávání prvku 1 do haldy:



Naopak z hlady se (vrchní) prvek odebírá tak, že se vrátí hodnota v kořeni a nahradí se hodnotou v nejpravějším listu na poslední hladině a tento list se smaže. Tím může být opět porušeno pravidlo, že uzly jsou menší než následníci, proto musíme zkontrolovat, zda nová hodnota kořene je větší než hodnota obou jeho dětí. Pokud není, vyměníme tuto hodnotu s hodnotou menšího dítěte a pokračujeme v kontrolování tohoto dítěte dokud není vše O.K., nebo dokud se nedostaneme do listu, kde je vše O.K. z definice.



Jaká je složitost těchto operací? Pro přidávání potřebuji

- přidat prvek na konec haldy
- a v nejhorším případě projít stromem od listu ke kořeni a v každém kroku provést jedno porovnání a jednu výměnu

Složitost přidání v nejhorším případě tedy bude záviset na výšce stromu. Není těžké si uvědomit, že pokud je strom haldou a má n uzlů, pak jeho výška bude buď $\log_2 n$ nebo, $\log_2 n + 1$. Celková složitost tedy bude asymptoticky $O(\log n)$ + složitost přidání prvku na konec haldy. To bude samozřejmě záviset na konkrétním kódu, ale v jakékoliv rozumné verzi to nebude horší než $O(\log n)$. Podobně to vyjde pro operaci odebrání prvku. Pokud bychom chtěli jen zjistit, jaký je nejmenší prvek, bude to dokonce $O(1)$.

Zavedli jsme haldu takovým způsobem, aby měla v kořeni nejmenší číslo. Pro náš zelinářský software by se však hodilo naopak mít v kořeni číslo největší. Rozmyslete si, že pokud v haldě obrátím podmínku na čísla, t.j. pokud budu požadovat, aby číslo uzlu bylo větší, než číslo všech jeho potomků, bude všechno opět fungovat (po jednoduché úpravě operací). V následující praktické implementaci v Pythonu uvedeme tuto "opačnou" verzi haldy.

Ukážeme si nyní práci s haldou v Pythonu. Mohli bychom s ní pracovat jako s obyčejným stromem. V takovém případě by bylo dobré si u každého uzlu navíc

pamatovat i jeho rodiče. Tedy uzel by mohl být třeba čtveřice `(data, hodnota, rodic, deti)`. S takovouto reprezentací by se ale pracovalo trochu nešikovně, protože neumožňuje jednoduše zjistit nejlevější list na předposlední hladině (resp. na poslední, pokud takový není). Bude proto lepší si umět haldu uchovávat v seznamu (a v budoucnu se nám to bude hodit).

Haldu budeme ukládat po jednotlivých hladinách, každý prvek seznamu bude dvojice `(data, hodnota)`. Jeho index v poli bude určovat jeho pozici ve stromu. Na prvním místě bude kořen, na druhém a třetím jeho případné dvě děti, a tak dále. Protože ukládaný strom je haldou, můžeme si z pozice uzlu `u` jednoduše vypočítat pozice jeho dětí a rodičů. Je-li totiž jeho pozice `n`, pak jeho děti mají pozici `2*n + 1` a `2n + 2` a jeho rodič má index `(n-1)/2`. Uvědomte si, že zde využíváme faktu, že jde o haldu, u obecného stromu by to nefungovalo! Elegance tohoto přístupu spočívá v tom, že nejlevější list je prostě poslední prvek pole. Operace v Pythonu pak můžou vypadat třeba následovně jako na výpisu 2.3.

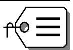
Zelinář — pokračování

Říkali jsme, že upravíme náš zákaznický systém tak, aby jak tištění lístků (funkce `generujListek`) tak vyvolávání zákazníků (funkce `naRade`) pracovaly rychle. Uděláme to tak, že využijeme výše uvedené funkce pro práci s haldou a zákazníky si budeme ukládat do haldy. Funkce budou tedy vypadat takto:

```
1 halda = []
2 def generujListek( jmeno, castka ):
3   push( halda, jmeno, castka)
4
5 def naRade():
6   return pop( halda )
```

2.4 Cvičení. Náš kamarád zelinář si upravený zákaznický systém velmi pochvaluje, nicméně všiml si jedné nepříjemné vlastnosti. Občas se totiž stane, že nějaký šetřivý zákazník zaplatí za odbavení malou částku a pak se nikdy nedostane na řadu, protože ho všichni ostatní přeplatí. A tak se mu v obchodě hromadí neodbavení šetřiví zákazníci ... Požádal Vás proto, abyste systém upravili tak, aby bral v potaz nejen zaplacenou částku, ale i čas strávený čekáním. Na Vás teď je navrhnout úpravu. Když se nad tím zamyslíte, nebude to vůbec složité ...

Algoritmus 2.3 Operace s haldou



```
1  # -*- coding:utf-8 -*-
2
3  # Jednoduché funkce pro počítání pozic dětí a rodičů
4  def left_child_index(n):
5     return 2*n + 1
6
7  def right_child_index(n):
8     return 2*n + 2
9
10 def parent_index(n):
11    return (n-1)/2
12
13 def last_index(halda):
14    return len(halda)-1
15
16 # Projde stromem od uzlu s indexem index ke kořeni
17 # a zajistí, že hodnoty potomků jsou vždy
18 # menší než hodnoty rodičů
19 def check_up(halda, index):
20    p = parent_index(index)
21    if halda[p][1] < halda[index][1]:
22       halda[p],halda[index] = halda[index],halda[p]
23       check_up(halda,p)
24
25 # Podobně jako předchozí funkce, ale prochází
26 # od uzlu s indexem index naopak k listům
27 def check_down(halda, index):
28    l,r = left_child_index(index), right_child_index(index)
29    if halda[l][1] < halda[r][1]:
30       c = r
31    else:
32       c = l
33    if halda[index][1] < halda[c][1]:
34       halda[index],halda[c] = halda[c],halda[index]
35       check_down(halda, c)
36
37 # Zjistí největší prvek v haldě
38 def peek(halda):
39    if len(halda) == 0:
40       return None
41    return halda[0]
42
43 # Přidá prvek do haldy
44 def push( halda, data, hodnota ):
45    halda.append((data,hodnota))
46    check_up(halda, last_index(halda))
47
48 # Odebere největší prvek z haldy
49 def pop( halda ):
50    if len(halda) == 0:
51       return None
52    ret = peek(halda)
53    halda[0] = halda[-1]
54    del halda[-1]
55    check_down(halda, 0)
56    return ret
```

Zásobník

V předchozích oddílech jsme si představili dvě základní "datové struktury": frontu a haldu. Zajímavých šikovných způsobů jak ukládat data je samozřejmě mnohem více (Red-Black Stromy, Hashovací tabulky, ...), ale v této přednášce nám na ně nezbude čas. Přesto bychom rádi zmínili ještě alespoň *zásobník* (anglicky stack, nebo také LIFO — last in, first out). V jistém smyslu se velmi podobá frontě: je to obyčejný seznam prvků, do kterého umíme přidávat prvky (funkce append) a prvky z něj vybírat (funkce pop). Prvky se do zásobníku ukládají stejně jako do fronty — na konec seznamu. Rozdíl spočívá ve funkci pop, která prvky neodebírá ze začátku (jako u fronty), ale z konce. V Pythonu se každý seznam standardně chová jako zásobník:

```
1 >>> z = []
2 >>> z.append(1)
3 >>> z.append(2)
4 >>> z.append(3)
5 >>> z.pop()
6 3
7 >>>
```

2.5 Cvičení. Představte si, že máte k dispozici dvě proměnné typu zásobník (t.j. máte funkce `append` a `pop`, které se chovají jak bylo výše popsáno). Zkuste pomocí těchto dvou zásobníků naprogramovat frontu. (Nápověda: co se stane, když z jednoho zásobníku postupně vyberete všechny prvky a dáte je na druhý zásobník?)

Kapitola 3

Vyhledávání, třídící algoritmy

Vyhledávání

Nechme zelináře zelinářem a pojd'me se spolu podívat na jinou úlohu. Tentokrát budeme chtít napsat spellchecker. Pro začátek bude velmi jednoduchý. Dostane textový dokument a slovník a vypíše všechna slova, která se vyskytují v dokumentu a nevyskytují se ve slovníku. V Pythonu to naprogramujeme velmi jednoduše:

```
1  # Vrátí seznam všech slov nacházejících se ve stringu doc,
2  # která nejsou v seznamu slovnik. Předpokládáme že jednotlivá
3  # slova ve stringu doc jsou oddělena mezerou.
4
5  def spell_check(slovnik,doc):
6    spatna = []
7    for slovo in doc.split(' '):
8      if slovo not in slovnik:
9        spatna.append(slovo)
10   return spatna
```

Toto jednoduché řešení však má své mouchy. Jaká je jeho složitost? Funkce musí nejprve rozdělit řetězec doc na jednotlivá slova. Python toto provede za nás funkcí split, ale při troše dobré vůle bychom to zvládli sami a zjistili bychom, že složitost této operace je $O(N)$, kde N je délka řetězce. Pokud by nás zajímala složitost v závislosti na počtu slov, stačí si uvědomit, že většina slov je kratších než nějakých 15 znaků a pravděpodobně žádné slovo nebude delší než 30 znaků. Z toho plyne, že

složitost bude opět $O(N)$ — multiplikativní konstanty v O-notaci zanedbáváme. Pak musíme projít všechna slova a u každého otestovat, zda se nachází ve slovníku. Ve výše uvedeném programu to za nás dělá operátor `in`. Ten je v podstatě jen zkratkou následujícího kódu:

```
1 # Vrátí true, pokud se slovo vyskytuje v seznamu slovnik
2 def contains(slovo, slovnik):
3   for s in slovnik:
4     if s == slovo:
5       return True
6   return False
```

Z kódu je vidět, že složitost bude záviset na tom, jestli (a kde) se slovo `slovo` ve slovníku `slovnik` nachází. V ideálním případě se `slovo` nachází na prvním místě — to provedeme pouze jednu operaci porovnání. V nejhorším případě v seznamu nebude vůbec a pak provedeme tolik operací, kolik je prvků seznamu `slovnik`. Složitost v nejhorším případě tedy bude $O(N + N * M) = O(N * M)$, kde N je počet slov ve vstupním dokumentu a M je počet slov ve slovníku. Ukážeme si, že pokud věnujeme nějaký čas přípravě slovníku `slovnik`, podaří se nám navrhnout funkci `contains` tak, aby měla složitost $O(\log M)$, což dá pro celý algoritmus složitost $O(N * \log M)$. Trik spočívá v tom, že si slovník předem uspořádáme podle abecedy. Jak to udělat uvidíme v další části, teď však předpokládejme, že ho máme uspořádaný. K testu, zda se ve slovníku dané slovo nachází, teď můžeme využít metodu půlení intervalů — vezmeme si slovo uprostřed našeho slovníku a porovnáme ho s hledaným slovem. Pokud je v abecedě hledané slovo dříve, víme, že se musí nacházet v první polovině slovníku. Pokud je naopak později, nachází se v druhé polovině. Ten samý postup budeme opakovat znovu, dokud slovo buď nenajdeme, nebo nezjistíme, že ve slovníku není. Takto popsaný algoritmus nejlépe zapíšeme pomocí rekurze:

Pojďme se nejprve přesvědčit, že algoritmus je správný. Jak je u rekurzivních algoritmů obvyklé, budeme postupovat indukcí, v tomto případě dle `n=end-start`.

`n=0` Pokud je `start == end`, pak algoritmus vrátí False, což je správně, protože `slovnik[start:start]` je prázdný seznam ve kterém se nic nevyskytuje.

`n=1` Pokud je `end-start == 1`, pak `middle == start` a `slovnik[start:end] = [slovnik[start]]`. Pokud je tedy `slovo == slovnik[middle]`, algoritmus vrátí `True`, jinak se tam slovo nevyskytuje. Pokud se tam nevyskytuje, zavolá se buď `contains(slovo, slovnik, start, start)` nebo `contains(slovo, slovnik, end,end)` což obojí správně vrátí `False`.

Algoritmus 3.1 Binární vyhledávání

```
1 # Vrátí True, pokud se slovo vyskytuje v setřízeném
2 # seznamu slovnik[start:end]. Jinak vrátí False.
3 def contains(slovo, slovnik, start, end):
4   if start >= end:
5     return False
6   middle = start + (end-start)/2
7   if slovo == slovnik[middle]:
8     return True
9   elif slovo < slovnik[middle]:
10    return contains(slovo, slovnik, start, middle)
11  elif slovo > slovnik[middle]:
12    return contains(slovo, middle+1, end)
```

`n+1` Indukční krok. Předpokládejme že pro `end-start = n >= 1` algoritmus pracuje a uvažujme `end-start = n+1`. Je jasné, že pokud se slovo v seznamu `slovnik[start:end]` vyskytuje, pak je buď rovno

`slovnik[middle]`

nebo se vyskytuje v

`slovnik[start:middle]`

nebo v

`slovnik[middle+1:end]`.

První případ je jasný a na druhé dva můžeme použít indukční předpoklad, protože jak

`middle-start < end-start`

tak

`end-(middle+1) < end-start`

(první případ plyne z toho, že `start-end >= n+1>=2`).

A jaká je složitost algoritmu? Protože v každém kroku provedeme maximálně tři porovnání a jedno přiřazení a zároveň zmenšíme rozdíl `end-start` na polovinu, je intuitivně vcelku jasné, že složitost bude $4 \log M = O(\log M)$.

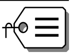
3.2 Cvičení. Indukcí dokažte, že složitost výše uvedené funkce `contains` bude opravdu $O(\log M)$.

Máme tedy mnohem efektivnější funkci `contains`, ale není to zadarmo. Aby tato funkce fungovala, je třeba mít slovník setřízený podle abecedy, což bude naším dalším tématem. Předem můžeme prozradit, že složitost třídění bude $O(M \log M)$. Když se vrátíme k naší funkci `spell_check` dostaneme složitost $O((M+N) \log M)$. Jak to porovnat s původní složitostí $O(N * M)$? V typickém případě bude slovník mnohem větší než seznam slov, tedy $M >> N$. Pokud například $N < \log M$ zjistíme, že původní algoritmus bude lepší — vzhledem k tomu, že budeme hledat jen pár slov, nevyplatí se nám třídit celý slovník. Na druhou stranu, pokud předpokládáme, že funkci `spell_check` budeme využívat často (což je rozumný předpoklad), setřízení slovníku se nám dlouhodobě vyplatí.

Třídící algoritmy

Insertion Sort. V předchozí části jsme narazili na úlohu setřídit seznam slov dle abecedy. Pojd'me se zamyslet nad tím, jak bychom to mohli udělat. Jako první nás asi napadne postupovat tak, jak bychom sami postupovali, pokud bychom měli úlohu vyřešit ručně. Prostě bychom si setřízený seznam budovali postupně a v každém kroku do již setřízené části zatřídili další slovo. Tomuto algoritmu se říká *insertion sort*. Zapišme si ho v Pythonu:

Algoritmus 3.3 Insertion Sort



```
1 def insertion_sort(seznam):
2       last_sorted = 0
3       while last_sorted < len(seznam)-1:
4             j = last_sorted + 1
5             while j > 0 and seznam[j] < seznam[j-1]:
6                   seznam[j-1],seznam[j] = seznam[j], seznam[j-1]
7                   j = j - 1
8             last_sorted = last_sorted + 1
```

V proměnné `last_sorted` si algoritmus udržuje index posledního setřízeného prvku (t.j. pole `seznam[:last_sorted+1]` je setřízené). Na začátku je setřízený pouze první prvek pole (řádek 2). Dokud není pole setřízené celé (podmínka na řádku 3), provádím následující kroky (řádky 4-8): vezmu první nesetřízený prvek

(řádek 4), zatřídím ho do pole na správné místo (řádky 5-7) a zvětším si index posledního setřízeného prvku (řádek 8).

Jaká je složitost tohoto algoritmu? Cyklus na řádku 3 se provede n-krát, kde n je velikost vstupního pole `seznam`. V těle cyklu pak provedu dvě přiřazovací operace (řádky 4, 8) a zatřízení (cyklus na řádcích 5-7). Vnitřní zatřizovací cyklus provede v nejhorším případě 3·j operací. Po chvilce počítání (je třeba sečíst řadu $1+2+\cdots+n$) dostanu celkový dolní odhad složitosti $2n + 3n(n+1)/2 = O(n^2)$.

3.4 Cvičení. Jaká bude složitost algoritmu, pokud ho pustíme na již setřízené pole?

Za chvíli uvidíme, že existují algoritmy s mnohem lepší asymptotickou složitostí. Proč jsme tedy insertion sort vůbec zmiňovali? Kromě toho, že je to algoritmus velmi jednoduchý, má několik podstatných výhod. Ačkoliv je jeho *asymptotická* složitost kvadratická, pro malá vstupní pole (do velikosti cca 100 prvků) je často nejrychlejším algoritmem. Další výhodou je jeho optimální chování, pokud ho pustíme na již setřízené pole (viz cvičení). Další možnou výhodou je to, že pole zpracovává postupně — algoritmy, které popíšeme dále, potřebují znát pole celé najednou.

Heap sort. Pokud vám prozradím, že optimální asymptotická složitost třídících algoritmů (založených na provnávání prvků) je $O(n \log n)$ a navíc Vám napovím, že lze použít haldu, možná Vás už napadne, jak funguje algoritmus, kterému se říká *heap sort*. Vzpomeňte si, že operace přidání prvku do haldy (`push`) a odebrání nejmenšího prvku z haldy (`pop`) má složitost $O(\log n)$. Co kdybychom postupně prošli celé pole a každý prvek přidali na haldu a pak postupně odebírali vždy nejmenší prvek z haldy? To je princip heap sortu (viz výpis 3.5)

Algoritmus 3.5 Heap sort

```
1 def heap_sort(seznam):
2   halda = []
3   for p in seznam:
4     push(halda, p)
5
6   for i in range(len(seznam)):
7     p = pop(halda)
8     seznam[i] = p
```

Kód uvedený ve výpisu 3.5 je pěkný (a není těžké spočítat, že má asymptotickou

složitost $O(n \log n)$[1]), není však moc efektivní — k setřízení pole si v paměti vyhradí ještě jednou tolik místa na haldu, na kterou postupně přidává prvky. Tento nedostatek však lze velmi jednoduše odstranit. Když se nad tím zamyslíte a uvědomíte si, že jednoprvkové pole je haldou, určitě pro Vás následující cvičení nebude příliš obtížné:

3.6 Cvičení. Přepište heap sort (a haldové operace `push` resp. `push`) tak, aby algoritmus nepotřeboval separátní seznam pro haldu.

Další algoritmus, který si ukážeme, je asi v praxi nejpoužívanějším algoritmem.

Quicksort je klasickým rekurzivním algoritmem. Funguje na následujícím principu. Pokud seznam, který se má setřídit, má pouze jeden prvek, je setřízený a algoritmus může skončit. V opačném případě si seznam rozdělí na dvě části tak, aby všechny položky v levé části byly menší než všechny položky v pravé části. Pak se rekurzivně zavolá a setřídí levou a pravou část. Následně může prostě připojit setřízenou pravou část za setřízenou levou část a celý seznam bude setřízený, protože prvky vlevo byly menší než prvky vpravo a obě části setřízené byly. Zbývá popsat rozdělování na části. To se typicky provádí tak, že se ze seznamu náhodně zvolí jeden prvek, tzv. *pivot*, a pole se rozdělí tak, aby vlevo byly všechny prvky menší nebo rovné pivotu a vpravo všechny prvky větší nebo rovné pivotu. Seznamem se prochází současně zleva i zprava. V okamžiku, kdy z jednoho směru narazíme na prvek, který má být vzhledem k pivotu v opačné části, čekáme, než k takovému prvku dorazíme i zprava (pokud na žádný nenarazíme, víme, že jsme hotovi a tento prvek bude první prvek, který patří do pravé části). V okamžiku, kdy na takový prvek narazíme i zprava, oba prvky vyměníme a pokračujeme. Když se setkáme, máme pole rozdělené. Schematicky bude quicksort vypadat takto:

```
1 def quicksort( seznam ):
2    if len(seznam) == 1:
3      return seznam
4    pivot = zvol_pivota( seznam )
5    L, P = rozdel_seznam( seznam, pivot )
6    L_sorted = quicksort(L)
7    P_sorted = quicksort(P)
8    return L_sorted + P_sorted
```

[1] Tak, jak je napsaný, má první cyklus, který staví haldu, složitost $O(n \log n)$. Dá se však napsat trochu chytřeji tak, aby měl složitost lineární (viz např. [Wi85]). Druhý cyklus se takto přepsat nedá, složitost tedy bude stále $O(n \log n)$

3.7 Poznámka. Ve skutečnosti by výše uvedený schematicky zapsaný quicksort byl relativně pomalý, protože by docházelo k častému kopírování z paměti do paměti při vytváření nových polí (L,P, L+P). V reálu proto vůbec nebudeme vytvářet nové seznamy a funkce quicksort bude vypadat spíše tak, jak je uvedena ve výpisu 3.8.

Algoritmus 3.8 Quicksort

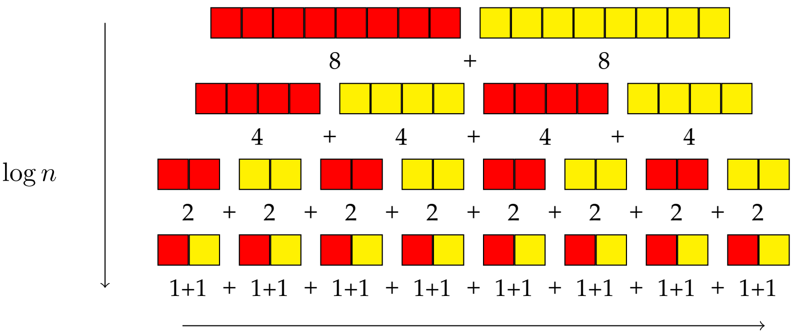
```
1 def quicksort( seznam, start, end ):
2     if start >= end:
3         # Prazdné a jednoprvkové pole jsou
4         # z definice setřízené
5         return
6     pivot = zvol_pivota( seznam, start, end )
7     stred = rozdel_seznam( seznam, start, end, pivot)
8     quicksort( seznam, start, stred )
9     quicksort( seznam, stred+1, end )
10    return
```

t.j. všechno se bude odehrávat v poli `seznam`, nic se nebude nikam kopírovat a podseznamy budou vždy vymezené dvěma indexy, počátečním (`start`) a koncovým (`end`).

3.9 Poznámka. Při rozdělování pole je třeba dát pozor na to, aby se pole opravdu "rozdělilo", t.j. aby bylo `start <= stred < end`. Pokud by totiž pravá nebo levá část byla prázdná, program by se zacyklil. Tuto podmínku nicméně není těžké splnit, protože pivot může být prvkem jak pravé, tak levé části.

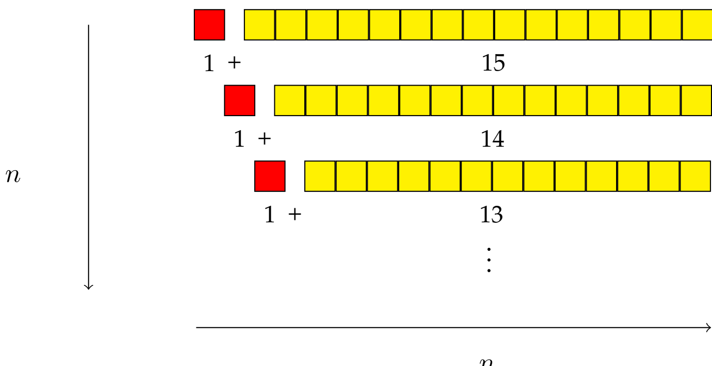
3.10 Cvičení. Naprogramujte funkce `zvol_pivota` a `rozdel_seznam`!

Jaká je složitost quicksortu? V optimálním případě, pokud by se nám vždy povedlo zvolit pivota tak, aby levá i pravá část pole byla stejně velká, by byla složitost $O(n \log n)$. Pěkně to lze nahlédnout na následujícím obrázku, kde červená políčka odpovídají levým částem pole, žlutá pravým. Čísla pod polem pak udávají, kolik operací musím provést při dělení daného pole:



Formálně lze správnost odhadu ukázat například indukcí dle n (pro jednoduchost uvažujeme pouze pole velikosti $n = 2^k$): Pro pole velikosti 2^0 je to zřejmé. Mám-li nyní dáno pole velikosti 2^k, musím rozdělit pole (složitost 2^k) a dvakrát zavolat quicksort na pole poloviční velikosti. Z indukčního předpokladu má jedno takové volání složitost $2^{(k-1)} \log 2^{(k-1)}) = 2^{k-1}(k-1)$, dvě volání tedy budou mít složitost $2^k(k-1)$. Celkem tedy $2^k + 2^k(k-1) = 2^k k = O(2^k k) = O(n \log n)$.

Jaká bude složitost quicksortu, pokud se nám pivoty nepodaří zvolit optimálně? Představme si, že jako pivota volím vždy první prvek pole a na vstupu dostanu pole již setřízené. Co se stane je opět dobře vidět na obrázku:



Na předchozím obrázku jsme měli úplný binární strom, jehož hloubka je $\log n$. Zde nám tento strom zdegeneroval ve strom, jehož hloubka je n. Když si spočteme složitost, uvidíme, že je to $O(n^2)$, což je nepříjemné (tato situace by mohla nastat i

v případě, že je pole již setřízené; v takovém případě má i jednoduchý insertion sort složitost dokonce lineární!). Je tedy vidět, že vhodná volba pivota může radikálně ovlivnit délku výpočtu. Jak jsme si ukázali výše, optimální volbou by byl medián pole. Protože pro volbu mediánu existuje algoritmus běžící v lineárním čase (detaily viz [BFPRT73], nebo heslo `Selection Algorithm` na *Wikipedii*), mohli bychom jej použít a zajistit si, že složitost quicksortu bude $O(n \log n)$ i v nejhorším případě. Nicméně v praktických implementacích se ukazuje, že je rychlejší jako pivota zvolit nějaký náhodný prvek pole (nikoliv tedy vždy první!). Není tak sice zajišťeno, že algoritmus poběží v $O(n \log n)$, ale ve většině případů poběží mnohem rychleji, než kdyby pivota vybíral složitěji[2].

Mohli bychom nyní pokračovat zkoumáním dalších třídících algoritmů a věnovat tomu ještě několik kapitol. Také bychom se mohli zabývat i jinými parametry třídících algoritmů než jen jejich časovou složitostí — mohli bychom například zkoumat jejich stabilitu nebo paměťovou náročnost. Zde to ale dělat nebudeme a odkážeme zájemce do literatury. Místo toho se ještě chvíli budeme zabývat časovou složitostí a budou nás zajímat dolní odhady.

Dolní odhad pro složitost vyhledávání

Ukázali jsme si, že algoritmus Quicksort má v optimálním čase složitost $O(n \log n)$. Algoritmus Heap sort má dokonce v nejhorším případě složitost $O(n \log n)$. Podobně třeba algoritmus Merge sort (který jsme si neukazovali) má v nejhorším případě složitost $O(n \log n)$. Nemohli bychom najít nějaký ještě lepší algoritmus? Ukážeme si, že pokud jako základní operace povolíme pouze porovnávat jednotlivé prvky případně je prohazovat, lepší algoritmy neexistují. Bude to však trochu složitější, než dokazovat, že nějaký algoritmus má takovou a takovou složitost. Budeme totiž chtít ukázat, že žádný třídící algoritmus nemůže běžet rychleji než $O(n \log n)$[3] Jak takovou věc ukázat? Rozhodně nemůžeme procházet třídící algoritmy jeden po druhém a dokazovat, že mají složitost nejméně $O(n \log n)$. Náš postup bude následující. Ukážeme, že máme-li nějaký rychlý algoritmus, pak to nemůže být třídící algoritmus — nalezneme pole, které nesetřídí správně. Představme si tedy, že máme nějaký algoritmus A, který na vstupu dostane seznam čísel $x_0, \ldots, x_n$, která má setřídit. Pro jednoduchost budeme předpokládat, že výstupem bude posloupnost indexů $\mathbf{a} = (a_0, \ldots, a_n)$ setřízeného pole, t.j. a_0 je index

[2] Navíc jsou implementace napsány tak, že pokud zjistí, že hrozí kvadratická složitost — například se pole několikrát po sobě rozdělilo velmi nerovnoměrně — přepnou se třeba na heap sort, který je sice trochu pomalejší, ale má garantovanou složitost $O(n \log n)$

[3] Formálně vzato následující úvaha dává dolní odhad pro složitost v *nejhorším* případě. Malou úpravou bychom mohli ukázat, že odhad platí i pro složitost v *průměrném případě*.

nejmenšího prvku, a_1 index druhého nejmenšího prvku a tak dále. Řekněme, že tento algoritmus má složitost $g(n) < O(n \log n)$. Při svém běhu tedy může provést nejvýše $g(n)$ operací porovnání a každé takové porovnání má pouze dva možné výsledky *menší*, *větší* (rovnost pro jednoduchost zanedbáme). Označíme-li si tedy tyto výsledky jako 0 a 1, dostaneme pro každý běh algoritmu posloupnost nul a jedniček $\mathbf{p} = (p_0, \ldots, p_{g(n)})$ délky (nejvýše) $g(n)$. Množinu všech takových posloupností budeme značit $P(n)$. Ke každé posloupnosti nám algoritmus dá posloupnost indexů $\mathbf{a}$, což je prostá posloupnost čísel od 0 do n. Označme si množinu všech prostých posloupností čísel od 0 do n symbolem $A(n)$. Náš algoritmus nám tedy dává funkci $f : P(n) \to A(n)$. Ukážeme, že pro dostatečně velká n bude množina $A(n)$ větší než množina $P(n)$. To znamená, že bude existovat posloupnost $\mathbf{a} = (a_0, \ldots, a_n) \in A(n)$, která nemá žádný vzor — není výstupem algoritmu pro žádný vstup. Zvolíme tedy čísla $x_0, \ldots, x_n$ tak, abychom jejich setřízením získali indexy $a_0, \ldots, a_n$. Pustíme na ně náš algoritmus, který začne čísla porovnávat — tím získáme posloupnost $\mathbf{p}$. Protože ale posloupnost $\mathbf{a}$ byla volená tak, aby nebyla výstupem algoritmu, algoritmus musí vrátit posloupnost jinou, která tudíž nebude správně setřízená!

Zbývá tedy ukázat, že pro dostatečně velké n bude množina $A(n)$ větší než množina $P(n)$. Spočíst velikost množiny $P(n)$ je jednoduché, je to $2^0 + 2^1 + \cdots + 2^{g(n)-1} + 2^{g(n)} = 2^{g(n)+1} - 1$. Velikost množiny $A(n)$ je $n!$. Protože se však s faktoriály špatně pracuje, uvědomme si, že $n! > (n/2)^{n/2}$ — to nám bude stačit. Protože algoritmus byl asymptoticky rychlejší než $O(n \log n)$, víme, že pro libovolné K, b a dostatečně velké n bude $g(n) < nK \log n - b$. Položme $K = 1/4, b = 1$ a zvolme si dostatečně velké n (aby platila nerovnost, alespoň > 4). Pak bude platit:

$$g(n) < nK \log n - b$$

a navíc

$$nK \log n < \frac{n}{2} \log \frac{n}{2}.$$

Počítejme:

$$|P(n)| = 2^{g(n)+1} - 1 < 2^{nK \log n - b + 1} = 2^{nK \log n} < 2^{n/2 \log n/2} = (n/2)^{n/2} < n! = |A(n)|,$$

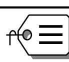
což je přesně to, co jsme chtěli ukázat.

Speciální třídící algoritmy

Než opustíme téma třídění, ukážeme si ještě jeden algoritmus, který je zajímavý svou složitostí. Má totiž složitost lineární! Vzhledem k předchozímu dolnímu odhadu je jasné, že algoritmus musí mít nějaký háček. Háček je v tom, že algoritmus

umí pracovat pouze s daty, která mají být utříděna podle nějakého číselného klíče z předem daného, nepříliš velkého rozsahu. Z číselného klíče totiž získáme více informací než se nám povede, pokud se omezíme na pouhé porovnávání dvou klíčů (jak to vyžaduje obecná úloha třídění, pro kterou jsme ukázali dolní odhad $O(n \log n)$).

Bucket sort (přihrádkové třídění) Samotný princip algoritmu je jednoduchý a přirozený. Nějak podobně bychom asi postupovali, pokud bychom měli třídit kartičky s čísly. Pro jednoduchost budeme uvažovat pouze úlohu, ve které máme setřídit seznam n čísel a pro začátek budeme předpokládat, že čísla se nemohou v seznamu opakovat. Postupovat budeme následovně. V prvním kroku si zjistíme, nejmenší (k_{min}) a největší (k_{max}) číslo. Interval $[k_{min}, k_{max}]$ si rozdělíme na "přihrádky" nějaké předem dané velikosti m. Do těchto přihrádek budeme vkládat jednotlivá čísla ze seznamu. Například pro $k_{min} = 0, k_{max} = 1000$ a $m = 10$ bude první přihrádka vyhrazena pro čísla od 0 do 9, druhá od 10 do 19 a tak dále až do poslední přihrádky pro čísla od 990 do 1000. Pak projedeme náš seznam čísel a čísla roztřídíme do jednotlivých přihrádek. Nakonec setřídíme jednotlivé přihrádky pomocí nějakáho jiného algoritmu. Stejný algoritmus budeme moci samozřejmě použít i pro klíče z jiného intervalu. Implementace v pythonu je uvedena na výpisu 3.11.

Algoritmus 3.11 Bucket Sort 

```
1  def bucket_sort(seznam, m):
2
3        # Nalezení nejmenšího a největšího prvku
4        min = max = seznam[0]
5        for p in seznam:
6              if p > max: max = p
7              elif p < min: min = p
8
9        # Vyrobení přihrádek
10       poc_prihradek = (max-min)/m
11       prihradky = []
12       for i in range(poc_prihradek):
13             prihradky.append([])
14
15       # Roztřídění do přihrádek
16       for p in seznam:
17             prih = (p-min)/m
18             prihradky[prih].append(p)
19
20       # Setřízení přihrádek
21       for prih in prihradky:
22             # Sort je nějaký vhodný klasický třídící
23             # algoritmus, který setřídí zadané pole
24             prih.sort()
25
26       # "Slití" přihrádek zpátky do seznamu
27       pos = 0
28       for prih in prihradky:
29             for p in prih:
30                   seznam[pos] = p
31                   pos += 1
```

Jaká bude složitost? Nalezení minima a maxima a vyrobení přihrádek bude mít složitost $O(n)$. Rozdělení čísel do přihrádek má složitost opět $O(n)$. Přihrádek je $(k_{max} - k_{min})/m$, a každou musíme setřídit. V každé přihrádce je nejvýše m čísel, které jsme schopni setřídit v čase $O(m \log m)$. Celkem tedy zabere třízení přihrádek (řádky 18.–22.) čas $O((k_{max} - k_{min})/m \cdot m \log m) = O((k_{max} - k_{min}) \log m) = O(k_{max} - k_{min})$[4] Celková složitost algoritmu tedy bude $O(n) + O(n) + O(k_{max} - k_{min}) = O(k_{max} - k_{min})$. Pokud bude rozsah možných klíčů, t.j. hodnota $k_{max} - k_{min}$, roven $O(n)$ bude mít algoritmus lineární složitost!

3.12 Cvičení. Při odvození složitosti jsme předpokládali, že každé číslo se vyskytne nanejvýš jednou, z čehož vyplývá, že $n \leq k_{max} - k_{min}$. Tento předpoklad však je zbytečný. Není těžké upravit algoritmus tak, aby fungoval i s opakujícími se klíči. Dokonce vlastně můžeme využít přímo výše uvedený kód, který ve skutečnosti jedinečnost klíčů nikde nepoužívá. Jediným místem, kde jsme ji využily, byl odhad složitosti setřízení jedné přihrádky. Zkuste se tedy upravit kód tak, aby setřízení přihrádky trvalo vždy $O(m \log m)$ kroků i v situaci kdy jedinečnost klíčů nepředpokládáme.

Můžeme se však dostat do situace, kdy $k_{max} - k_{min}$ bude příliš velké (ve srovnání s n) a výše uvedený časový odhad začne být nepříznivý v porovnání s $n \log n$. I vtomto případě se však nemusíme vzdát, jen musíme být trochu chytřejší. Tuto situaci řeší následující varianta přihrádkového třídění, které se říká *radix sort*

Radix sort Princip kořenového třídění je podobný jako u jednoduchého přihrádkového třídění, jen se trochu jinak zařazují položky do přihrádek. Algoritmus funguje tak, že si klíč každé položky rozdělí na předem daný počet částí d a pak postupně v d průchodech položky setřídí. V každém průchodu postupuje stejně jako obyčejné přihrádkové třízení s přihrádkama velikosti jedna (tudíž odpadá nutnost v daném kroku přihrádky třídit), ale místo celého klíče uvažuje jen jeho část. V Pythonu může vypadat třeba jako na výpisu 3.13.

[4] Poslední rovnost plyne z toho, že m je nějaká předem pevně daná konstanta.

Algoritmus 3.13 Radix Sort

```
1  # coding: utf-8
2  def extract_key(start_cifra, end_cifra, klic):
3        """Vrátí část klíče od start_cifra do end_cifra
4            >>> extract_key(1,4,12345)
5            234
6            >>> extract_key(0,2,123445)
7            45
8        """
9        return (klic % (10**end_cifra))/(10**start_cifra)
10
11 def var_buck_sort(seznam, scf, ecf):
12       """ Provede bucket sort ale místo všech cifer klíče uvažuje pouze
13             cifry od místa scf do místa ecf. Přihrádky mají velikost 1 tudíž
14             odpadá krok třídění jednotlivých přihrádek."""
15
16       # Nalezení nejmenšího a největšího prvku
17       min = max = extract_key(scf,ecf,seznam[0])
18       for p in seznam:
19             k = extract_key(scf,ecf,p)
20             if k > max: max = k
21             elif k < min: min = k
22
23       # Vyrobení přihrádek
24       poc_prihradek = (max-min)+1
25       prihradky = []
26       for i in range(poc_prihradek):
27             prihradky.append([])
28
29       # Roztřídění do přihrádek
30       for p in seznam:
31             k = extract_key(scf,ecf,p)
32             prih = (k-min)
33             prihradky[prih].append(p)
34
35       # "Slití" přihrádek zpátky do seznamu
36       pos = 0
37       for prih in prihradky:
38             for p in prih:
39                   seznam[pos] = p
40                   pos += 1
41
42 from math import log, ceil
43 def radix_sort(seznam, d):
44       mx = max(seznam)
45       delka_klice = int(ceil(log(mx,10)))
46       delka_casti = delka_klice/d
47       ecf = delka_casti
48       while ecf <= delka_klice:
49             var_buck_sort(seznam, ecf-delka_casti, ecf)
50             ecf += delka_casti
```

Kapitola 4

Analýza jazyka

V této kapitole se podíváme na problém analýzy jazyka. Nepůjde nám o přirozené jazyky — to by přesahovalo možnosti úvodního kurzu — ale o jazyky tzv. formální. Formální jazyky jsou jazyky, které mají přesně specifikovánu syntax, t.j. co je a co není v jazyce přípustné. Co to znamená *přesně* specifikovat? Dohodněme se, že pro naše účely to znamená, že je možné o daném řetězci automaticky rozhodnout, zda je v jazyce přípustný či nikoliv.

> Přirozené jazyky tuto podmínku nesplňují, protože přípustnost je nejednoznačná — na přípustnosti nějaké věty se často neshodnou ani lingvisté. Jednoduchým diagonálním argumentem však lze “zkonstruovat” jazyky, kde je přípustnost daná jednoznačně, nicméně neexistuje algoritmus, který by o přípustnosti rozhodl. Takové jazyky nás však zajímat nebudou.

Příkladem formálního jazyka je třeba samotný Python. Dalším příkladem může být třeba jazyk nějaké matematické teorie. Nebo html, jazyk ve kterém se píší webové stránky. Toto všechno jsou relativně komplikované jazyky, kterým odpovídají i (relativně) komplikované rozhodovací algoritmy. My si ukážeme několik jednodušších jazyků a algoritmy, které rozhodují o přípustnosti pro tyto jazyky. Nejprve se však budeme zabývat jednodušším problémem — vyhledáváním v textu. Optimální algoritmus pro vyhledávání nás totiž navede k definici tzv. regulárních jazyků.

Vyhledávání v textu

Následující úlohu zná každý, kdo v textovém editoru někdy použil funkci "najdi" (find).

4.1 Úloha. Nalezněte v daném textu (posloupnosti znaků) všechny výskyty nějakého slova (věty, posloupnosti znaků).

První řešení, které nás napadne, může vypadat třeba takto:

Algoritmus 4.2 Naivní Vyhledávání

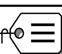
```
1  def find(text, s):
2        found=[]
3        for tpos in range(len(text)):
4              f = True
5              for spos in range(len(s)):
6                    if not text[tpos+spos] == s[spos]:
7                          f = False
8                          break
9              if f:
10                   found.append(tpos)
11       return found
```

Vnější cyklus postupně prochází celý text a vnitřní cyklus na každé pozici testuje, zda se tam nevyskytuje hledaná posloupnost. Tento vnitřní test provede nejvýše m operací, kde m je délka hledaného řetězce. Celková složitost algoritmu je tedy $O(n \cdot m)$, kde n je délka textu. Na první pohled nás napadne, že lépe to snad ani nejde. Uvědomme si však, že algoritmus dělá spoustu zbytečné práce. Představme si, že hledáme řetězec `aaaaa` (`s`) v řetězci `aaaazzzz` (`text`). V první iteraci zjistíme, že na pozici `0` se řetězec nenachází, protože páté písmeno je `z` a ne `a`. V tu chvíli však už je zbytečné testovat všechny pozice do pátého písmena, protože náš řetězec žádné `z` neobsahuje. Ve skutečnosti, když testujeme výskyt řetězce na pozici `tpos` pro `tpos >= 5`, všechny znaky `text[tpos],...,text[tpos+4]` jsme už viděli, takže znalost znaku `text[tpos+5]` by nám teoreticky měla stačit k rozhodnutí, zda se `s` na pozici `tpos` vyskytuje. Jak tuto informaci využít? Představme si, že každá posloupnost `p` čtyř znaků má přiřazeno nějaké číslo n_{p}. Pak lze sestrojit tabulku, která nám pro každý znak `z`, a číslo n_{p} určí,

- zda je posloupnost `p+z` (posloupnost `p` následovaná znakem `z`) je rovná hledané posloupnosti
- číslo $n_{p[1:]+z}$ posloupnosti `p+z` bez prvního znaku

Pokud bychom takovou tabulku měli, bylo by vyhledávání velmi jednoduché. Algoritmus by vypadal třeba jako na výpisu 4.3 (uvažujeme jednoduchý příklad, kdy hledáme řetězec `aab` v posloupnosti která sestává pouze ze znaků `a` a `b`).

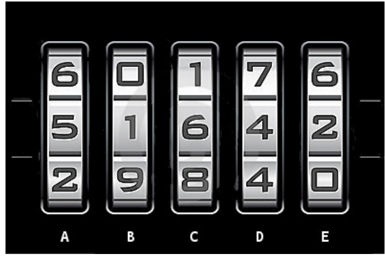
Algoritmus 4.3 Vyhledávání za pomoci tabulky



```
 2 # Očíslování posloupností délky 2
 3 # (uvažujeme pouze řetězce ze znaků 'a' a 'b')
 4 seqnum = {
 5    'aa':0,
 6    'ab':1,
 7    'bb':2,
 8    'ba':3
 9 }
10
11 # Tabulka odpovídající hledání
12 # řetězce 'aab'
13 tabulka = {
14    (0,'a'):(False,0),
15    (0,'b'):(True,1),
16    (1,'a'):(False,3),
17    (1,'b'):(False,2),
18    (2,'a'):(False,3),
19    (2,'b'):(False,2),
20    (3,'a'):(False,0),
21    (3,'b'):(False,1)
22 }
23
24 def smartfind(text, tabulka, lens):
25       """ text ... řetězec, ve kterém se vyhledává
26             tabulka ... vyhledávací tabulka
27             lens  ... délka hledaného řetězce """
28       found = []
29       # přečti prvních lens znaků a zjisti číslo této
30       # posloupnosti
31       np = seqnum[text[:lens-1]]
32
33       for tpos in range(lens,len(text)):
34             # dle tabulky urči, zda následující znak (text[tpos])
35             # zakončuje hledanou posloupnost a zároveň
36             # aktualizuj číslo posloupnosti np
37             f, np = tabulka[(np,text[tpos])]
38             if f:
39                   found.append(tpos-lens+1)
40       return found
```

Je lehké nahlédnout, že složitost je nyní lineární v délce textu, t.j. $O(n)$. Problém je však se sestrojením tabulky (a s její velikostí). Máme-li abecedu o k znacích (v příkladu 4.3 je $k = 2$), a hledáme řetězec délky m, pak naše tabulka bude mít velikost $k \cdot k^{m-1}$ a tedy její výroba bude trvat $O(k^m)$. Tím bychom vyměnili složitost $O(n \cdot m)$ za $O(n + k^m)$, což není příliš výhodné.

Konečné automaty Naše tabulka je ale příliš veliká. Všimněte si například, že hodnoty odpovídající $n_p = 1, 2$ jsou stejné. To napovídá, že by bylo možné ušetřit. Ukážeme, že tabulku lze nahradit komplikovanější strukturou, tzv. konečným automatem. Jeho výhoda spočívá v tom, že vhodný automat lze vytvořit v čase $O(m)$ (místo $O(2^m)$). Co to je konečný automat? Ukážeme si to nejprve na příkladě jednoduchého konečného automatu — zakódovaného zámku na kole.



Koncepčně má zámek dva stavy — odemčeno a zamčeno. Zadáním správného kódu ho člověk může ze stavu zamčeno přesunout do stavu odemčeno. Ve skutečnosti si však můžeme představit, že zámek má mimo dvou základních stavů různé vnitřní stavy odpovídající různým možným kódovým kombinacím. Jeden z těchto vnitřních stavů — stav odpovídající správnému kódu — pak odpovídá globálnímu stavu odemčeno, ostatní odpovídají stavu zamčeno. Nastavením číslice na jednom z možných míst pak přesouváme zámek z jednoho vnitřního stavu do jiného.

Obecná definice konečného automatu je zobecněním této konkrétní situace.

4.4 Definice. *Konečný automat* je dán množinou (vnitřních) stavů S, množinou vstupů (abecedou) Σ, přechodovou funkcí $f : S \times \Sigma \to S$, množinou koncových stavů $E \subseteq S$ a jedním počátečním stavem $s^b \in S$.

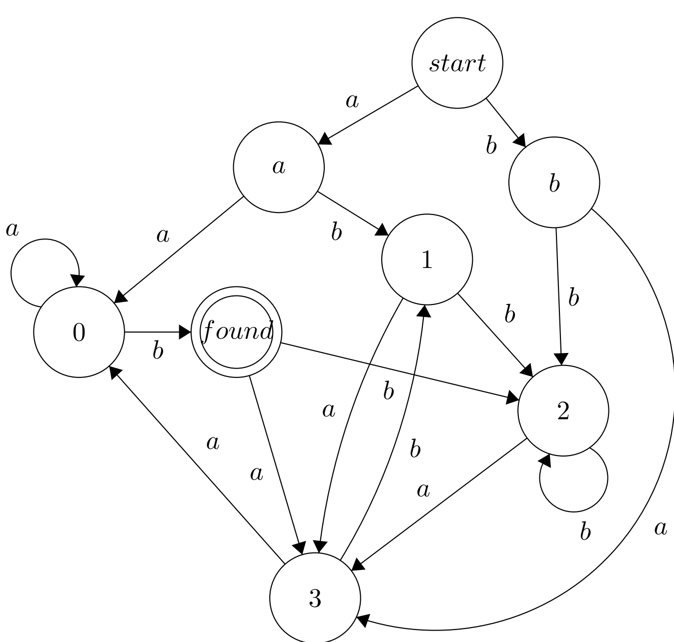
V případě zámku by množina vnitřních stavů byla množina všech 5 ciferných čísel, Množina vstupů by byla množina všech dvojic (pozice, číslice), kde pozice je jedno z písmen A, B, C, D, E. Přechodová funkce pak odpovídá změně stavu při zadání jedné číslice kódu. Koncový stav je jediný — ten který odpovídá správnému kódu. Počáteční stav je náhodný zamčený stav, který byl nastaven při zamykání.

Definujme ještě co to znamená výpočet. Intuitivně je to posloupnost stavů a vstupů začínající v počátečním stavu a končící v nějakém koncovém stavu taková, že přechod mezi sousedními členy této posloupnosti je dán přechodovou funkcí:

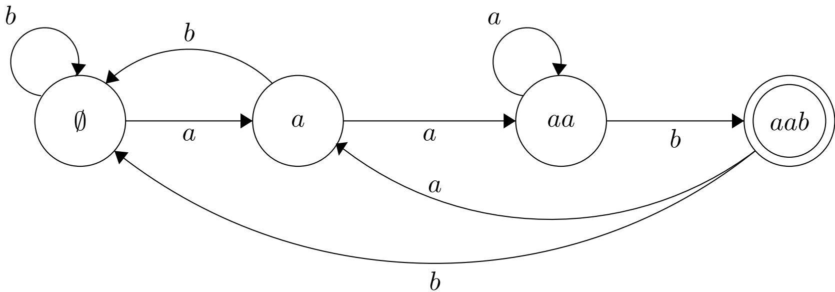
4.5 Definice. Posloupnost stavů $\overline{s} = \langle s_0, \dots, s_n \rangle$ a vstupů $\overline{v} = \langle v_0, \dots, v_{n-1} \rangle$ je výpočtem automatu $A = (S_A, \Sigma_A, f_A, E_A, s^b_A)$ pokud $s_0 = s^b_A$, $s_n \in E_A$ a pro $1 \leq i \leq n-1$ platí $s_{i+1} = f(s_i, v_i)$.

Když se zamyslíte nad algoritmem 4.3 možná Vás napadne, že by se dal popsat jako provádění výpočtu nějakého konečného automatu. Tabulka v tomto případě prostě zadává přechodovou funkci. Abychom dostali konečný automat, musíme ještě doplnit počáteční stav, dva stavy odpovídající přečtení prvního znaku a koncový stav.

Výsledkem pak bude automat znázorněný na následujícím obrázku. Každé kolečko odpovídá stavu, šipky mezi kolečky odpovídají přechodům mezi stavy daným vstupním písmenem.



Jak jsme poznamenali již dříve a je z obrázku vidět, že stavy 1 a 2 jsou v podstatě ekvivalentní a mohli bychom je sloučit dohromady. Ve skutečnosti lze ale celý automat ještě podstatně zjednodušit tak, aby měl přesně o jeden stav více než je délka hledaného řetězce (m). Stavy automatu na obrázku odpovídají všem možným nejvýše dvouprvkovým posloupnostem písmen `a,b`. Například stav 0 odpovídá posloupnosti `aa` zatímco třeba stav 3 odpovídá posloupnosti `ba`. Základní myšlenka spočívá v tom, že zahodíme všechny stavy, které neodpovídají žádnému počátečnímu úseku hledaného řetězce. Získáme tak automat znázorněný následujícím obrázkem:



Zbývá ještě ukázat, že takový automat umíme sestrojit v čase $O(m)$. Sestrojit stavy a základní "dopředné" šipky spojující tyto stavy od nejkratšího k nejdelšímu je jednoduché. Potřebujeme však ještě dodat šipky jdoucí "zpět". Řekněme, že jsme v nějakém stavu s který odpovídá řetězci `r`. Abychom mohli správně dodat šipky jdoucí "zpět", potřebovali bychom znát stav s' odpovídající řetězci `r[1:]` (každá šipka jdoucí zpět totiž bude odpovídat nějaké šipce ze stavu s'). Není však problém si tento stav uložit v nějaké pomocné proměnné a tuto proměnnou postupně aktualizovat. Celý algoritmus v Pythonu (včetně funkce find) lze nalézt na výpisu 4.6.

Pro zájemce dodejme, že existuje ještě jiný algoritmus, tzv. Boyer-Moore algoritmus, který řeší úlohu 4.1 také v čase $O(m+n)$ ale v praxi je rychlejší než algoritmus 4.6.

Regulární jazyky

V předchozím odstavci jsme zavedli pojem konečného automatu a ukázali, jak lze konečné automaty využít pro rychlé hledání v textu. Nicméně samotný pojem konečného automatu je zajímavý i teoreticky, protože se dá ukázat, že charakterizuje třídu regulárních jazyků.

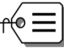
4.7 Definice. Je-li Σ množina znaků (*abeceda*), značíme Σ^* množinu všech (konečných) posloupností prvků množiny Σ. Prvkům množiny Σ^* budeme říkat *slova*. *Jazykem* v abecedě Σ pak rozumíme libovolnou podmnožinu $L \subseteq \Sigma^*$.

4.8 Definice. Třída *regulárních jazyků* nad abecedou Σ je nejmenší třída $\mathcal{L}$ splňující:

(i) Prázdný jazyk je prvkem $\mathcal{L}$,

(ii) Pro každý znak $a \in \Sigma$ je $\{a\} \in \mathcal{L}$,

Algoritmus 4.6 Vyhledávání pomocí konečných automatů



```
3  def build_automaton(retezec):
4        """ Vrátí automat rozpoznávající řetězec retezec.
5
6              Funkce ve skutečnosti vrátí seznam stavů, kde stav stav s indexem 0 je počáteční
7              a stav poslední stav je koncovým stavem. Každý stav je seznamem šipek, které z
8              něj vedou, t.j. dvojice (znak, index_následujícího_stavu_pro_daný_znak)
9        """
10       c_state_num = 0            # index aktuálně vytvářeného stavu
11       help_state = []            # pomocný stav pro podřetězec r[1:]
12       c_state = []                    # aktuálně vytvářený stav
13       A = [None]*(len(retezec)+1) # Inicializujeme automat
14       for c in retezec:
15             A[c_state_num] = c_state
16
17             # Přidáme dopřednou šipku do aktuálního stavu
18             c_state.append((c,c_state_num+1))
19
20             # Přidáme zpětné šipky, t.j. zkopírujeme šipky z pomocného stavu do aktuálního stavu.
21             # Pokud z pomocného stavu vede šipka označená znakem c pak příslušně aktualizujeme
22             # pomocný stav, jinak nastavíme pomocný stav na 0
23             next_help_state = A[0]
24             for sipka in help_state:
25                   char, target_state_num = sipka
26                   if char == c:
27                         next_help_state = A[target_state_num]
28                   else:
29                         c_state.append((char,target_state_num))
30
31             help_state = next_help_state
32             c_state_num += 1
33             c_state = []
34
35       # Přidáme koncový stav
36       for sipka in help_state:
37             char, target_state_num = sipka
38             c_state.append((char,target_state_num))
39       A[-1]=c_state
40       return A
41
42 def optimalfind(text, s):
43       A = build_automaton(s)
44       c_state = A[0]          # aktuální stav nastavíme na počáteční stav automatu
45       found_state_num = len(A)-1 # index koncového stavu
46       found = []                    # seznam výskytů
47
48       # Protože výskyt řetězce s na pozici p najdeme teprve když přečteme znak
49       # na pozici p+len(s), musíme len(s) odečíst
50       pos = -len(s)
51
52       for c in text:
53             pos = pos+1
54             # Najdi šipku odpovídající načtenému písmenu. Pokud taková je, přesuň se do odpovídajícího
55             # stavu, jinak se přesuň do počátečního stavu
56             move_to_zero = True
57             for sipka in c_state:
58                   char, target_state_num = sipka
59                   if char == c:
60                         # pokud jsme v koncovém stavu, máme výskyt
61                         if target_state_num == found_state_num:
62                               found.append(pos)
63                         c_state = A[target_state_num]
64                         move_to_zero = False
65                         break
66             if move_to_zero:
67                   c_state = A[0]
68       return found
```

(iii) Třída $\mathcal{L}$ je uzavřená na sjednocení a konkatenaci (jsou-li $A, B \in \mathcal{L}$, pak jejich konatenace, značená $A{\cdot}B$, je jazyk sestávající ze slov tvaru $w_A w_B$, kde $w_A \in A$ a $w_B \in B$).

(iv) Třída $\mathcal{L}$ je uzavřená na Kleeneho hvězdičku (je-li $A \in \mathcal{L}$, pak i jazyk A^* je prvkem $\mathcal{L}$, kde A^* je nejmenší nadmnožina A obsahující prázdnou množinu a uzavřená na konkatenaci).

Tato definice vypadá velmi abstraktně. Nicméně je překvapivé, že má mnoho různých ekvivalentních reformulací. Jedna možná definice využívá konečné automaty:

4.9 Věta. *Jazyk L je* regulární, *pokud existuje konečný automat A takový, že L je množinou slov w takových, že existuje posloupnost stavů $\overline{s}$, že $(\overline{s}, w)$ je výpočtem (viz 4.5) automatu A.*

Nabízí se otázka, zda je každý jazyk regulární. Jednoduchým argumentem se můžeme přesvědčit, že nikoliv: všech jazyků je totiž nespočetně (kontinuum), ale konečných automatů je jen spočetně (každý konečný automat je konečný objekt). Není příliš těžké dokonce přímo sestrojit neregulární jazyk.

4.10 Cvičení. Ukažte, že jazyk nad abecedou $\Sigma = \{(,)\}$ sestávající ze správně uzávorkovaných výrazů, nemůže být regulární. (Hint: uvažte řetězec začínající $n+1$-otevřenými závorkami, kde n je počet stavů daného automatu.)

Regulární výrazy Ukážeme si ještě jinou ekvivalentní definici regulárního jazyka, která je bližší původní definici a používá pojem regulárního výrazu. Tento pojem zavedl americký matematik Kleene (1909–1994) a umožňuje velmi kompaktní popis regulárních jazyků. Každému regulárnímu výrazu R odpovídá jazyk $L(R)$ dle následující definice

4.11 Definice. *Regulární výraz* nad abecedou Σ (neobsahující znaky $\varepsilon, (,), |, *$) je definován rekurzivně:

- $\emptyset, \varepsilon$ jsou regulární výrazy, $L(\emptyset) = \emptyset, L(\varepsilon) = \{\emptyset\}$, t.j. prázdný jazyk a jazyk sestávající z prázdného slova,
- každý znak $a \in \Sigma$ je regulárním výrazem, $L(a) = \{a\}$,
- je-li r regulární výraz, pak i (r) je regulární výraz a $L((r)) = L(r)$,
- jsou-li r, s dva regulární výrazy, pak $r|s$ a rs je regulární výraz a $L(r|s) = L(r) \cup L(s)$,$L(rs) = L(r) \cdot L(s)$ (alternace a konkatenace)

- je-li r regulární výraz, pak r^* je regulární výraz a $L(r^*) = L(r)^*$ (Kleeneho hvězdička).

Aby se předešlo zbytečnému množství závorek je stanovena standardní priorita operací: Kleeneho hvězdička má nejvyšší prioritu, pak konkatenace a nakonec alternace.

4.12 Příklad. *Je-li* $\Sigma = \{a, b, c\}$*, pak* $r = ac^*(aa|bb)^*$ *je regulární výraz popisující jazyk sestávající ze slov začínajících písmenem* a*, pokračujících libovolným počtem znaků* c *a pak libovolným počtem dvojic znaků* aa *nebo* bb*.*

Následující věta říká, že regulární výrazy přesně popisují regulární jazyky.

4.13 Věta. *Jazyk* L *nad abecedou* Σ *je regulární právě když existuje regulární výraz* r *takový, že* $L = L(r)$*.*

Aritmetické výrazy

Ve cvičení 4.10 jsme uvedli, že jazyk sestávající ze slov obsahujících stejně otevřených jako zavřených závorek není regulární. To naznačuje, že mnoho zajímavých jazyků pravděpodobně regulárních nebude. Typickým reprezentantem je například jazyk sestávající z aritmetických výrazů, kterým se nyní budeme podrobněji zabývat. Že je to neregulární jazyk je vidět z toho, že nejen že musí obsahovat stejný počet otevřených jako zavřených závorek, tyto závorky navíc musí být správně poskládané. S aritmetickými výrazy se každý z vás nejspíš setkal již někdy v první třídě základní školy a máte intuitivní představu o tom, co to aritmetický výraz je. Abychom s nimi mohli pracovat bude užitečné tuto intuitivní představu zpřesnit do formální definice. Pro jednoduchost budeme uvažovat pouze aritmetické výrazy, ve kterých se nevyskytují proměnné a jediné povolené operace jsou sčítání, odčítání, násobení a dělení. Formálně tedy budou aritmetické výrazy řetězce nad abecedou $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (,)\}$. Nejjednodušším aritmetickým výrazem je nějaké libovolné číslo (řetězec číslic, t.j. neprázdný prvek jazyka $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$). Tyto výrazy (t.j. čísla) budeme nazývat *atomickými výrazy*.

4.14 Definice. Jazyk aritmetických výrazů je nejmenší podmnožina Σ^* která obsahuje všechny atomické výrazy a je uzavřena na následující operaci:

- Jsou-li a, b dva řetězce a $o \in \{+, -, *, /\}$ pak první operace (OP1) je definovaná takto:

$$OP1(a, b, o) = aob$$

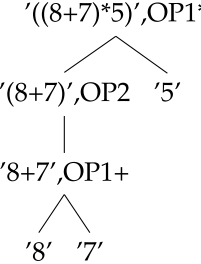
t.j. výsledkem je řetězec sestávající z konkatenace levé závorky, řetězce a, symbolu op, řetězce b a pravé závorky.

- Je-li a řetězec, pak druhá operace (OP2) je definovaná takto:

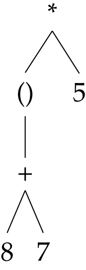
$$OP2(a) = (a)$$

t.j. výsledek je konkatenací levé závorky, řetězce *a* a pravé závorky.

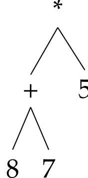
Stromová reprezentace Každý aritmetický výraz tedy vznikne z atomických výrazů (čísel) postupným aplikováním operace OP1 (aritmetické operace) resp. OP2 (uzávorkování). Tuto postupnou výstavbu aritmetického výrazu si můžeme představit jako (v našem případě binární) strom. V koředni stromu je výsledný výraz a poslední provedená operace, kterou byl získán. Jeho dětmi jsou pak ty jednodušší výrazy, ze kterých byl pomocí této operace vytvořen. V listech stromu jsou pak atomické výrazy. Nejlépe to je vidět na nějakém příkladě. Například stavba výrazu $(8 + 7) * 5$ je znázorněna následujícím stromem:



Tento strom můžeme ještě trochu zjednoduššit tím, že si v uzlech budeme pamatovat pouze operace a zahodíme všechny neatomické výrazy:

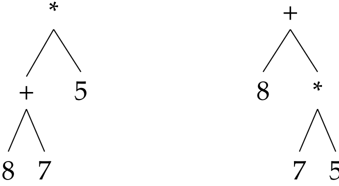


Dalšího zjednodušení můžeme dosáhnout, pokud nám nezáleží na rozlišení výrazů, které se liší pouze různým ekvivalentním uzávorkováním. V takovém případě můžeme prostě zahodit uzly s druhou operací. Získáme tak strom:



Je však dobré si uvědomit, že v tomto případě již došlo ke ztrátě informace: stejný strom by nám totiž vyšel i pro výraz ′(8+7)*(5)′. Pro většinu aplikací je však důležitá hodnota výrazu, kterou tato ztráta neovlivní. Uvědomte si, že výraz 8+(7*5), který má jinou hodnotu, dá ve skutečnosti jiný strom, ačkoliv se od původního výrazu liší pouze uzávorkováním; toto uzávorkování je totiž *neekvivalentní*.

Se závorkami souvisí ještě jeden problém. Výše zmíněná výstavba výrazu není vždy jednoznačná. Například výrazu $8 + 7 * 5$ odpovídají následující dva stromy:



K tomuto problému se dostaneme za chvíli, až se budeme zabývat konstrukcí těchto stromů.

Vyhodnocování výrazů Předpokládejme však na chvíli, že strom pro daný aritmetický výraz již máme v Pythonu k dispozici, například pro výraz ′(8+7)*5′ máme tedy strom

```
1 >>> T = ['*',['+',[8,None,None],[7,None,None]],[5,None,None]]
```

V tuto chvíli je relativně jednoduché výraz vyhodnotit. Vyhodnocování bude probíhat rekurzivně tak, že spočteme hodnotu každého uzlu, který odpovídá nějakému

podvýrazu. Hodnota listů, t.j. atomických výrazů, je prostě dané číslo. Hodnota vnitřních uzlů se pak spočte aplikací operace uložené v uzlu na hodnoty jeho dětí. V pythonu to může vypadat takto:

```
1 def eval_tree(T):
2     D,L,R = T
3
4     # Hodnota listu je v něm uložená
5     if L is None and R is None:
6         return D
7
8     # Rekurzivně spočtěme hodnotu dětí
9     LVal = eval_tree(L)
10    RVal = eval_tree(R)
11
12    # Aplikujme operaci D
13    if D == '+':
14        return LVal + RVal
15    if D == '-':
16        return LVal - RVal
17    if D == '*':
18        return LVal * RVal
19    if D == '/':
20        return LVal / RVal
```

Stromová reprezentace se hodí pro další zpracování, pro uživatele ale není příliš čitelná. Proto je užitečné mít funkci, která strom převede zpět na aritmetický výraz. I tuto funkce napíšeme snadno za pomoci rekurze:

```
1 def tree_to_expr_naive(T):
2     D,L,R = T
3
4     if L is None and R is None:
5         return str(D)
6
7     return tree_to_expr_naive(L) + D + tree_to_expr_naive(R)
```

Zde však narazíme na výše zmíněný problém související s tím, že jsme zahodili závorky. Náš strom totiž dá výraz výraz '8+7*5', který je možno číst dvěmi způsoby jako '8+(7*5)' či '(8+7)*5'. Konvence o prioritách operací pak preferuje druhou variantu, která však odpovídá jinému stromu! Abychom se této dvojznačnosti vyhli,

budeme muset na vhodných místech vypsat závorky. Správnější tedy bude napsat funkci takto:

```
1  def tree_to_expr_inorder(T):
2      D,L,R = T
3  
4      if L is None and R is None:
5          return str(D)
6  
7      ret = '('
8      ret = ret + ' ' + tree_to_expr_inorder(L)
9      ret = ret + ' ' + D
10     ret = ret + ' ' + tree_to_expr_inorder(R)
11     ret = ret + ' )'
12     return ret
```

Tato funkce vrátí pro původní strom výraz '((8 + 7) * 5)'. Kdyby daný výraz psal člověk, pravděpodobně by vynechal vnější závorky. Přebytečné závorky by samozřejmě šlo odstranit, bylo by to ale příliš komplikované, proto se smíříme s tím, že naše funkce občas závorkuje příliš horlivě.

Prefixová a postfixová notace Zastavme se ještě u funkce `tree_to_expr_inorder`. Funkce postupně rekurzivně prochází uzly stromu. Když přijde do daného uzlu, nejprve rekurzivně převede levý podstrom na výraz, k tomuto výrazu přidá operaci D a pak rekurzivně převede pravý podstrom a připojí ho nakonec. Co by se stalo, kdybychom prohodili řádky 8 a 9? T.j. při příchodu do uzlu bychom nejprve "vypsali" operaci v daném uzlu a teprve pak se zabývali podstromy. Výsledkem by byl následující výraz

$$(* (+ 8\ 7)\ 5)$$

Nazvěme takto změněnou funkci `tree_to_expr_preorder`. Ačkoliv její výstup na první pohled nemusí vypadat příliš užitečně, má tento zápis svůj smysl a dokonce i jméno — říká se mu prefixový zápis (a odpovídajícímu způsobu procházení stromu se říká preorder). V tomto zápise se nejprve zapíše operace a po ní teprve následují operandy. Výhoda tohoto zápisu je (ačkoliv to možná není hned vidět), že se obejde bez závorek a tudíž odpadají nejednoznačnosti, které se závorkami souvisí. Je však třeba dát pozor na to, že narozdíl od standardního tzv. *infixového* zápisu, je třeba nějak oddělovat čísla (atomické výrazy): u výrazu '+875' totiž není jasné, zda se mají sčítat čísla 8 a 75 nebo čísla 87 a 5. Další výhodou tohoto zápisu

je, že lze jednoduše převést zpět na stromovou reprezentaci:

```
1 def parse_prefix(token_list):
2     OPs = ['+','-','*','/']
3     token = token_list.pop(0)
4     if token in OPs:
5         L = parse_prefix(token_list)
6         R = parse_prefix(token_list)
7     else:
8         token = int(token)
9         L = None
10        R = None
11    return (token,L,R)
```

Funkce `parse_prefix` dostane na vstupu prefixově zapsaný výraz jako seznam jednotlivých prvků výrazu (prvek je buď operátor nebo číslo) a vrátí stromovou reprezentaci.

4.15 Cvičení. Indukcí ukažte, že funkce `parse_prefix` je správně napsaná.

Funkci `parse_prefix` lze jednoduše upravit tak, aby výraz místo převedení na stromovou reprezentaci rovnou vyhodnotila. Tuto variantu si ukážeme v nerekurzivním provedení, které používá zásobník:

```
1 def eval_prefix(token_list):
2     OPs = ['+','-','*','/']
3     stack = []
4     while len(token_list) > 0:
5         token = token_list.pop()
6         if token in OPs:
7           L = stack.pop()
8           R = stack.pop()
9           if token == '+':
10            stack.append(L+R)
11          elif token == '-'
12            stack.append(L-R)
13          elif token == '*'
14            stack.append(L*R)
15          elif token == '/'
16            stack.append(L/R)
17        else:
18          stack.append(int(token))
19    return stack.pop()
```

Zásobník ve skutečnosti pouze simuluje použití rekurze — pokud bychom výraz `stack.pop` nahradili rekurzivním voláním a výraz `stack.append` příkazem return, dostali bychom rekurzivní variantu.

4.16 Cvičení. Pokud ve funkci `tree_to_expr_inorder` prohodíme naopak řádky 9 a 10, dostaneme funkci, kterou je vhodné nazvat `tree_to_expr_postorder`. Tato funkce vrátí výraz v tzv. postfixové notaci. Naprogramujte funkce `parse_postfix` a `eval_postfix`.

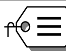
Zpracování infixové notace Vraťme se nyní k standardní t.j. infixové notaci a pokusme se navrhnout algoritmus, který ji převede na stromovou reprezentaci. Jak jsme již zmínili, infixová notace má problémy s jednoznačností — jednomu výrazu mohou odpovídat různé stromy. Tento problém by šlo vyřešit vhodnou modifikací jazyka aritmetických výrazů, která by pomocí závorek zajistila jednoznačnost. Operaci $OP2$ (závorkování) bychom úplně vypustili a operaci $OP1$ bychom modifikovali:

$$OP1(a, b, o) = (aob)$$

Takto modifikovaný jazyk je však pro lidi trochu nešikovný — vede k nadměrnému množství závorek. Místo toho se problém jednoznačnosti obvykle řeší zavedením tzv. *priority operátorů*: operátory s vyšší prioritou se vyhodnocují před operátory s nižší prioritou. Toto pravidlo umožňuje každému výrazu jednoznačně přiřadit příslušný strom. Popíšeme nyní způsob, jak převést infixový zápis na strom při respektování priority operátorů. Na vstupu předpokládáme řetězec. V prvním kroku z tohoto řetězce odstraníme vnější závorky. Pokud se ve zbývajícím výrazu nenachází žádné operátory, pak máme atomický výraz odpovídající jednoprvkovému stromu. V opačném případě v řetězci nalezneme operátor s nejnižší prioritou mezi těmi, co se nacházejí vně všech závorek. Pokud je takovýchto operátorů více, vezmeme z nich nejpravější. Tento operátor vložíme do kořene stromu. Zároveň nám operátor výraz rozdělí na dvě části. Rekurzivním voláním převedeme levou i pravou část na stromy, které napojíme na kořen. V Pythonu to bude vypadat třeba jako na výpisu 4.17.

Výhoda výše uvedeného algoritmu je jeho jednoduchost; není však příliš efektivní. Ukážeme si proto algoritmus, který zvládne infixový výraz zpracovat v lineárním čase. Pro jednoduchost uvedeme pouze verzi, která výraz rovnou vyhodnotí. Návrh verze, která zároveň postaví i strom, ponecháme do cvičení. Idea algoritmu je založená na jednoduchém pozorování. Pokud priorita operátorů ve vyhodnocovaném výraze zleva doprava neklesá, pak lze výraz jednoduše vyhodnotit tak, že si pamatujeme aktuální hodnotu výrazu a procházíme zprava do leva jí postupně

Algoritmus 4.17 Parsování infixových výrazů



```
2  def count_open_brackets(exp):
3      """ Spočítá kolika otevřenými závorkami začíná seznam @exp """
4      ret = 0
5      for c in exp:
6          if c != '(':
7              return ret
8          ret = ret+1
9      return ret
10 def remove_brackets(exp):
11     """ Vrátí seznam @exp bez vnějších závorek. """
12     open_brackets = count_open_brackets(exp)
13     counter = 0
14     for c in exp[open_brackets:-open_brackets]:
15         if c == '(':
16             counter = counter + 1
17         elif c == ')':
18             if counter > 0:
19                 counter = counter - 1
20             else:
21                 open_brackets = open_brackets - 1
22     if open_brackets == 0:
23         return exp
24     return exp[open_brackets:-open_brackets]
25 def find_min_prec_op(exp):
26     """ Najde operátor, který má nejmenší prioritu a je nejvíc vpravo
27         mimo všechny závorky. """
28     OPS = ['+','-','*','/']
29     precedence = {'+':0,'-':0,'*':1,'/':1}
30     pos = 0
31     min_prec = 2
32     min_pos = 0
33     open_brackets = 0
34     for token in exp:
35         if token == '(':
36             open_brackets = open_brackets + 1
37         elif token == ')':
38             open_brackets = open_brackets - 1
39         elif token in OPS:
40             if open_brackets == 0 and precedence[token] <= min_prec:
41                 min_pos = pos
42         pos = pos + 1
43     return min_pos
44 def parse_infix_tokenized(exp):
45     """ Převede infixový výraz (zadaný jako seznam prvků) na jeho
46         stromovou reprezentaci."""
47     exp = remove_brackets(exp)
48     if len(exp) == 1:
49         return (exp[0],None,None)
50     else:
51         pos_op = find_min_prec_op(exp)
52         L = parse_infix_tokenized(exp[:pos_op])
53         R = parse_infix_tokenized(exp[pos_op+1:])
54         return (exp[pos_op],L,R)
55 def tokenize(exp):
56     """ Převede výraz zadaný jako řetězec na seznam prvků. """
57     SEPS = ['+','-','*','/','(',')']
58     exp = exp.replace(' ','')
59     tokens = []
60     token=''
61     for c in exp:
62         if c in SEPS:
63             if len(token) > 0:
64                 tokens.append(token)
65                 token = ''
66             tokens.append(c)
67         else:
68             token = token+c
69     if len(token) > 0:
70         tokens.append(token)
71     return tokens
72 def parse_infix(exp):
73     """ Převede výraz zadaný jako řetězec na jeho stromovou reprezentaci. """
74     exp = tokenize(exp)
75     return parse_infix_tokenized(exp)
```

upravujeme prováděním příslušných operací. Například pro výraz '1+2+3*6/3' provedeme následující kroky:

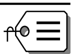
```
1  z = 3
2  z = 6 / z
3  z = 3 * z
4  z = 2 + z
5  z = 1 + z
```

V reálu samozřejmě musíme umět zpracovat i výrazy, kde priorita klesne. To provedeme jednoduše tak, že si vezmeme nejdelší začátek výrazu, kde priorita neklesá, ten vyhodnotíme a celý tento začátek nahradíme jeho vyhodnocením. Tak pokračujeme dokud výraz nezpracujeme celý. Např pro výraz '1+8*2+1' v prvním kroku zvolíme podvýraz '1+8*2', ten vyhodnotíme a dostaneme '17'. Jeho dosazením do původního výrazu získáme výraz '17+1', který opět můžeme vyhodnotit a získáme konečný výsledek '18'. Praktická implementace bude používat dva zásobníky. Do jednoho si bude ukládat operátory, do druhého operandy (čísla). Na operátorový zásobník bude vkládat operace tak, aby jejich priorita vzrůstala. V okamžiku, kdy narazíme na operátor s nižší prioritou než má poslední operátor na zásobníku, nejprve vyhodnotíme dosavadní výraz. To provedeme tak, že budeme ze zásobníku odstraňovat operátory s vyšší prioritou. Pro každý odstraněný operátor si ze zásobníku hodnot vezmeme dvě čísla, aplikujeme na ně operátor k odstranění a výsledek uložíme zpět do zásobníku hodnot. Zbývá vyřešit případné závorky. Idea je jednoduchá: každá závorka určuje podvýraz, který vyhodnotíme samostatně a pak ho v původním výrazu nahradíme výslednou hodnotou. Praktická implementace může být rekurzivní, případně může využít operátorového zásobníku: když narazíme na otevírající závorku, vložíme jí na operátorový zásobník. Při případném odstraňování operátorů při částečném vyhodnocováním si dáváme pozor, že se vždy zastavíme, pokud narazíme na otevřenou závorku. Když pak narazíme na zavírající závorku můžeme si být jistí, že od předchozí otevírající závorky mají operátory na zásobníku vzrůstající prioritu, můžeme je tedy jeden po druhém odstranit (a vyhodnotit) a nakonec odstranit i jednu otevírající závorku. Implementace v Pythonu je uvedena na výpisu 4.18.

4.19 Cvičení. Přepište algoritmus 4.18 tak, aby místo vyhodnocení výrazu vrátil jeho stromovou reprezentaci. Nápověda: Stačí si na zásobník s hodnotami ukládat místo čísel (částečné) stromy.

Bezkontextové gramatiky

Algoritmus 4.18 Vyhodnocování infixových výrazů

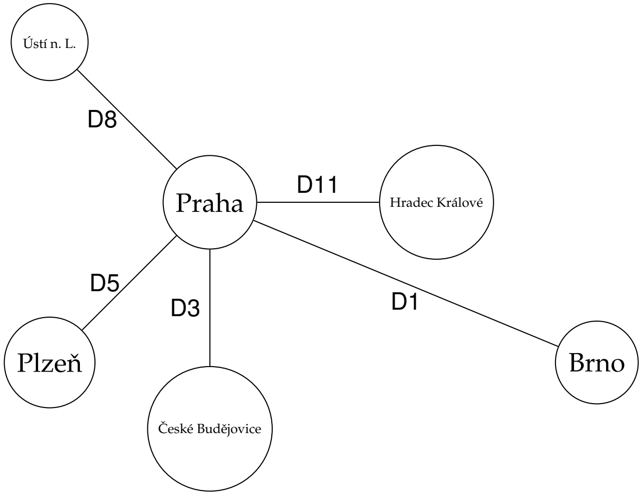


```
2  OPS = ['+','-','*','/',')','(']
3  precedence = {'+':0,'-':0,'*':1,'/':1,'(':-1,')':-1}
4  def reduce_stack(op_stack, arg_stack, prec):
5        while len(op_stack) > 0 and precedence[op_stack[-1]] >= prec:
6              op = op_stack.pop()
7              if op == '(':
8                    return
9              a1,a2 = arg_stack.pop(), arg_stack.pop()
10             if op == '+':
11                   arg_stack.append(a1+a2)
12             elif op == '-':
13                   arg_stack.append(a1-a2)
14             elif op == '*':
15                   arg_stack.append(a1*a2)
16             elif op == '/':
17                   arg_stack.append(a1/a2)
18
19 def eval_infix(token_list):
20       op_stack = []
21       arg_stack = []
22       number_stack = []
23       for token in token_list:
24             if token in OPS:
25                   if token == '(':
26                         op_stack.append(token)
27                   else:
28                         reduce_stack(op_stack,arg_stack,precedence[token])
29                         if token != ')':
30                               op_stack.append(token)
31             else:
32                   arg_stack.append(token)
33       reduce_stack(op_stack,arg_stack,-1)
34       return arg_stack.pop()
```

Kapitola 5

Grafové algoritmy

Matematický pojem grafu formalizuje situaci, kdy máme danou nějakou množinu vrcholů (v reálném světě například množinu měst), které jsou pospojovány hranami (v reálném světě například silinice). Příkladem takového grafu je následující obrázek:



Formální definice grafu (a některých dalších pojmů) následuje:

5.1 Definice. *Neorientovaný graf* je dvojice množin (V, E), kde V je množina vrcholů

a $E \subseteq V \times V$ je symetrická (t.j. $vEw \rightarrow wEv$), irreflexivní relace (t.j. $\neg vEw$), udávající hrany (E z anglického edge). Není-li relace symetrická, mluvíme o *orientovaném grafu*. Není-li relace irreflexivní, v grafu uvažujeme i tzv. smyčky — hrany vedoucí z vrcholu v opět do vrcholu v. *Stupněm vrcholu* $v \in V$ rozumíme velikost množiny $\{w \in V : vEw\}$ a značíme ho $deg(v)$ — t.j. je to počet vrcholů, do nichž vede z v hrana.

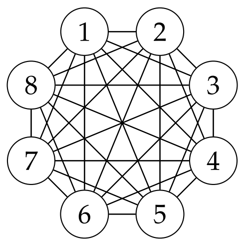
Těmito objekty se zabývá (rozsáhlá a zajímavá) matematická teorie grafů. Jedním z problémů, který se dá v řeči grafů formulovat je následující otázka:

5.2 Problém. Mějme dánu mapu regionů nějakého státu. Je vždy možné obarvit každý region jednou ze čtyř barev tak, aby žádné dva sousední regiony neměly stejnou barvu?

Zkusme tuto otázku formulovat v řeči grafů. Máme-li dánu mapu, vytvoříme z ní graf tak, že každému regionu bude odpovídat vrchol grafu a vrcholy sousedících regionů budou spojeny hranou. Můžeme tedy zkusit formulovat otázku 5.2 následujícím způsobem:

5.3 Problém. Lze vrcholy každého grafu obarvit čtyřmi barvami tak, aby sousední vrcholy měly vždy různou barvu?

Takto formulovaná otázka má však velmi jednoduchou odpověď. Uvažujme následující graf:



Je to tzv. úplný graf o osmi vrcholech — t.j. graf, který má osm vrcholů a každé dva vrcholy jsou navzájem spojeny hranou[1]. Z toho by ihned mělo být zřejmé, že pokud bychom vrcholy chtěli obarvit tak, aby žádní dva sousedé neměli stejnou barvu, budeme potřebovat minimálně 8 barev a odpověď na položenou otázku je ne.

[1] Tento graf se často značí K_8.

Jenže otázka 5.3 neodpovídá původní otázce přesně. Není totiž vůbec jasné, zda každý graf odpovídá nějaké mapě, t.j. lze ho získat výše uvedeným způsobem z nějaké mapy. Například graf na předchozím obrázku žádné reálné mapě odpovídat nemůže. Při grafové formulaci musíme být tedy opatrnější a omezit se pouze na grafy, *které odpovídají nějaké mapě*. Takovým grafům se říká *planární* nebo *rovinné* (protože je lze zakreslit do roviny tak, aby se žádné dvě hrany nekřížily).

5.4 Problém. Lze vrcholy každého planárního grafu obarvit čtyřmi barvami tak, aby žádné dva sousedící vrcholy neměly stejnou barvu?

Odpoveď na tuto otázku je ano. A číslo čtyři, je nejlepší možné.

5.5 Cvičení. Zkuste najít mapu (rovinný graf), kterou nelze obarvit pouze třemi barvami.

Historie tohoto problému je velmi zajímavá. Problém položil v roce 1852 jistý Francis Guthrie (jeho bratr studoval u De Morgana). Kolem roku 1890 byl publikován důkaz tvrzení, že stačí barev pět. Nicméně důkaz, že opravdu stačí pouze čtyři barvy, byl publikován až v roce 1976 a byl to první důkaz významného matematického tvrzení, který byl získán za pomoci počítače — autoři problém rozdělili na 1936 případů, které postupně analyzovali počítačovým programem. Menší vadou na kráse tohoto důkazu je fakt, že v počítačových programech jsou často chyby. Vzhledem k tomu, že analyzovat oněch 1936 případů ručně bylo neproveditelné, zůstala otázka, zda je důkaz opravdu správně. Mnoho matematiků proto tento důkaz nepovažovalo za důkaz. Od té doby bylo publikováno ještě několik důkazů, nicméně všechny částečně spoléhaly na práci počítače a dodnes není k dispozici důkaz, který by počítačovou analýzu nevyužíval.

Když problém formulujeme v řeči grafů, zdá se, že omezení na planární grafy je zbytečné — resp. otázka by mohla být zajímavá i pro ostatní grafy. Uvažujme proto následující definici:

5.6 Definice. Je-li $G = (V, E)$ neorientovaný graf, řekneme, že funkce $\chi : V \to \{1, 2, \ldots, k\}$ je *obarvení*, pokud pro každé dva vrcholy $v, w \in V$, které jsou spojeny hranou, t.j. vEw, platí $\chi(v) \neq \chi(w)$. *Chromatické číslo* grafu G, značené $\chi(G)$ je nejmenší přirozené číslo k takové, že existuje obarvení $\chi : V \to \{1, 2, \ldots, k\}$ grafu G. Obarvení χ grafu G je *minimální*, pokud je to obarvení $\chi(G)$ barvami.

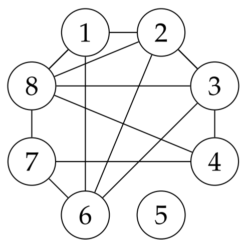
Tato definice vede přímo na následující úlohu.

5.7 Problém. Pro zadaný graf G nalezněte nějaké minimální obarvení.

Víme, že pro planární graf G platí, že $\chi(G) \leq 4$. Nicméně nalézt obarvení čtyřmi barvami je NP-těžký problém[2]

Tato úloha má mimochodem praktické využití. Představte si, že potřebujete naplánovat sadu událostí s tím, že některé události nemohou probíhat současně (např. proto, že jsou k dispozici pouze omezené prostředky). Tuto úlohu můžeme převést na nalezení minimálního obarvení grafu, ve kterém vrcholy odpovídají událostem a hrany spojují ty události, které nemohou probíhat současně. Minimální obarvení daného grafu totiž jednoduše odpovídá optimálnímu naplánování: v kroku 1 proběhnou události jejichž vrchol má barvu 1. V kroku 2 události s barvou 2. A tak dále.

Ačkoliv je úloha 5.7 NP-težká, ukážeme si, jak lze rychle nalézt relativně dobré obarvení grafu — dobré v tom smyslu, že používá málo barev. Nejdříve se však podíváme na různé způsoby, jak můžeme graf v Pythonu reprezentovat. Uvažujme následující graf o osmi vrcholech:



Matice souslednosti reprezentuje graf o n vrcholech pomocí matice $n \times n$, kde v i-tém sloupci a j-tém řádku je jednička, pokud je i-tý vrchol s j-tým vrcholem spojený hranou, v opačném případě je tam nula. Výše uvedenému grafu by odpovídala matice (+ zápis v pythonu)

[2] To znamená, že nejspíš neexistuje algoritmus, který by úlohu dokázal řešit v čase, který by byl polynomiálně závislý na velikosti vstupního grafu, t.j. jehož časová složitost by byla asymtoticky rovná nějakému polynomu. Alespoň pokud $P \neq NP$, čemuž však většina lidí věří.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

```
>>> g = [[0, 1, 0, 0, 0, 1, 0, 1],
         [1, 0, 1, 0, 0, 1, 0, 1],
         [0, 1, 0, 1, 0, 1, 0, 1],
         [0, 0, 1, 0, 0, 0, 1, 1],
         [0, 0, 0, 0, 0, 0, 0, 0],
         [1, 1, 1, 0, 0, 0, 1, 0],
         [0, 0, 0, 1, 0, 1, 0, 1],
         [1, 1, 1, 1, 0, 0, 1, 0]]
```

Incidenční Matice. Pokud má graf hodně vrcholů ale jen málo hran, může být výhodnější graf reprezentovat jako matici $n \times m$ (m je počet hran), kde řádky odpovídají hranám a sloupce vrcholům. V i-tém řádku jsou jedničky těch dvou sloupcích, které odpovídají vrcholům jež i-tá hrana spojuje. V ostatních sloupcích jsou nuly. Výše uvedenému grafu by odpovídala matice (+ zápis v pythonu)

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

```
>>> g = [[1, 1, 0, 0, 0, 0, 0, 0],
         [1, 0, 0, 0, 0, 1, 0, 0],
         [1, 0, 0, 0, 0, 0, 0, 1],
         [0, 1, 1, 0, 0, 0, 0, 0],
         [0, 1, 0, 0, 0, 1, 0, 0],
         [0, 1, 0, 0, 0, 0, 0, 1],
         [0, 0, 1, 1, 0, 0, 0, 0],
         [0, 0, 1, 0, 0, 1, 0, 0],
         [0, 0, 1, 0, 0, 0, 0, 1],
         [0, 0, 0, 1, 0, 0, 1, 0],
         [0, 0, 0, 1, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 1, 1, 0],
         [0, 0, 0, 0, 0, 0, 1, 1]]
```

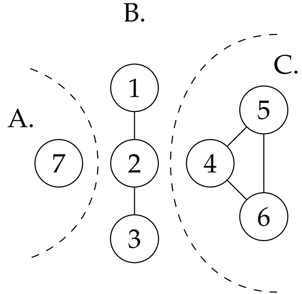
Seznam sousedů. Při této reprezentaci si ke každému vrcholu pamatujeme seznam jeho "sousedů", t.j. vrcholů do nichž z něj vede hrana. Výše uvedený graf by v pythonu byl reprezentován například takto:

```
>>> g = [[2, 6, 8],
         [1, 3, 6, 8],
         [2, 4, 6, 8],
         [3, 7, 8],
         [],
         [1, 2, 3, 7],
         [4, 6, 8],
         [1, 2, 3, 4, 7]]
```

5.8 Cvičení. Napište Pythonovské funkce pro převod grafu z jedné reprezentace do jiné.

Komponenty souvislosti

První grafový algoritmus, který si ukážeme, je algoritmus, který hledá tzv. komponenty souvislosti. Uvažujme následující graf, který se rozpadá na tři části A, B, C (jak je naznačeno přerušovanou čarou).



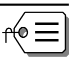
Těmto částem se říká tzv. komponenty souvislosti. Mají tu vlastnost, že mezi jednotlivými částmi nevede žádná hrana a naopak v jednotlivých částech se lze z libovolného uzlu po hranách dostat do libovolného jiného. To je obsahem následující definice.

5.9 Definice. Posloupnost vrcholů $v_1, \ldots, v_k$ grafu (V, E) nazveme *cestou*, pokud pro $i = 1, \ldots, k-1$ platí, že $v_i E v_{i+1}$. Graf (V, E) je souvislý, pokud z každého vrcholu vede cesta do každého jiného vrcholu, t.j. pokud pro všechna $v \neq w \in V$ existuje cesta $v_1, \ldots, v_k$ taková, že $v = v_1$ a $w = v_k$. Komponentou souvislosti

rozumíme nějakou maximální souvislou podmnožinu $K \subseteq V$, t.j. takovou podmnožinu, že mezi každými dvěma prvky K existuje cesta a přidáním jakéhokoliv dalšího prvku do K toto přestane platit.

Ukážeme si algoritmus, který k danému vrcholu nalezne komponentu souvislosti, do které vrchol patří (uvědomte si, že každý vrchol patří do nějaké komponenty souvislosti). Algoritmus funguje tak, že začne v zadaném vrcholu a postupně prochází všechny jeho sousedy a sousedy jeho sousedů atd., dokud to lze. V okamžiku, kdy už nemůže dál, našel komponentu souvislosti. Kód v pythonu je uveden ve výpisu 5.10.

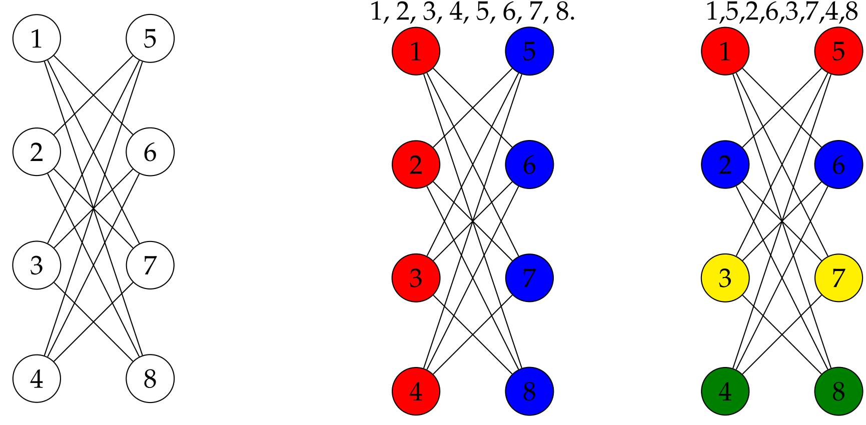
Algoritmus 5.10 Komponenta Souvislosti



```
1  def komponenta(vrchol, graf):
2      k = []
3      prace = [vrchol]
4      navstiveno = []
5      while( len(prace) > 0):
6          v = prace.pop()
7          if not v in navstiveno:
8              k.append(v)
9              navstiveno.append(v)
10             for w in graf[v]:
11                 if not w in navstiveno:
12                     prace.append(w)
13     return k
```

Obarvení grafu

Nyní si popíšeme jednoduchý algoritmus, který nalezne relativně dobré obarvení grafu. Algoritmus patří do třídy "hladových" algoritmů. Hladové se jim říká proto, že v každém kroku provedou akci, která se nabízí jako první — t.j. nemyslí dopředu. V našem případě bude algoritmus procházet vrcholy v nějakém pořadí a v každém kroku přiřadí vrcholu, který je právě na řadě, nejmenší barvu, kterou může — t.j. nejmenší takovou, která ještě nebyla přirazena žádnému jeho sousedu. Problém tohoto algoritmu je v tom, že v závislosti na pořadí, ve kterém vrcholy prochází, může nalézt obarvení velmi špatné. Uvažujme například následující (*bipartitní*) graf a jeho dvě možná obarvení vzniklá hladovým algoritmem (řady čísel nad obarvenými grafy udávají, v jakém pořadí algoritmus vrcholy procházel):



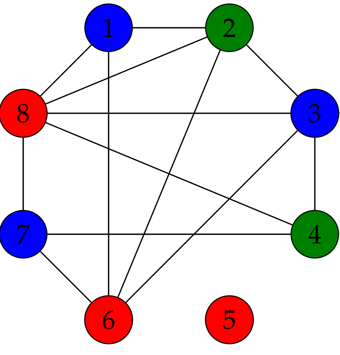
Je tedy velmi důležité vhodně zvolit pořadí, ve kterém se vrcholy procházejí. Protože je problém NP-těžký, nemůžeme čekat, že se nám podaří vždy zvolit optimální pořadí. Nicméně existují různé heuristiky, z nichž si jednu ukážeme (viz [Br79]). Nejdříve potřebujeme definici:

5.11 Definice. Saturovaností vrcholu v (značíme $sat(v)$) budeme rozumět počet různých barev, které jsou přiřazeny sousedícím vrcholům.

Pořadí budeme volit následovně. Vrcholy si uspořádáme sestupně podle jejich stupně (t.j. podle počtu sousedů). V každém kroku zvolíme ze zbývajících vrcholů ten vrchol, který má největší saturovanost. Pokud je takových vrcholů více, zvolíme vrchol největšího stupně. Zkusme nyní tento algoritmus zapsat v pythonu. Graf budeme reprezentovat jako pythonovský `dict`, který bude indexován vrcholy a bude obsahovat pro každý vrchol seznam jeho sousedů. Zároveň bude ke každému vrcholu ještě obsahovat seznam barev přiřazených jeho sousedům (pro výpočet saturovanosti).

Algoritmus 5.12 Heuristické barvení grafu

```
1  # -*- coding:utf-8 -*-
2
3  # vrchol = (cislo_vrcholu, seznam_sousedu, seznam_barev)
4  # graf = {v_1:sousedi_v1, v_2:sousedi_v2, ..., v_n:sousedi_vn}
5  def deg(v):
6      return len(v[1])
7
8  def sat(v):
9      return len(v[2])
10
11 def can_color( vertex, color ):
12     if color in vertex[2]:
13         return False
14     return True
15
16 def choose(g):
17     """ Odebere z grafu g vrchol, s nejvetsi
18         saturovanosti a spolus grafem ho vrati
19     """
20     g_sorted = sorted(g,key=sat)
21     v = g_sorted.pop()
22     return (g_sorted,v)
23
24
25 def color(graf):
26     # Setridi graf podle stupne
27     g_sorted = sorted(graf.items(), key=deg)
28
29     g = []
30     g_dict = {}
31     # Pridame seznam barev sousedu
32     for v in g_sorted:
33         nv = [v[1],[]]
34         g.append( [v[0]]+nv )
35         g_dict[v[0]] = nv
36
37     coloring = {}
38
39     # zde si uchovavame nejvetsi
40     # dosud pouzitou barvu
41     max_color = -1
42
43     while( len(g) > 0 ):
44         # Z grafu g odebereme vrchol s
45         # nejvetsi saturovanosti
46         (g,v) = choose(g)
47
48         # Prochazime postupne barvy od
49         # nejmensi do nejvetsi dosud pouzite
50         for c in range(0,max_color+2):
51             if can_color(v,c):
52                 coloring[v[0]]=c
53                 # projdeme sousedy vrcholu v
54                 # a u kazdeho si poznacime,
55                 # ze sousedi s c-barevnym vrcholem
56                 for s in v[1]:
57                     if c not in g_dict[s][1]:
58                         g_dict[s][1].append(c)
59                 # pokud jsme pouzili novou barvu,
60                 # zvetsime max_color
61                 if c > max_color:
62                     max_color = c
63                 break
64     return coloring
```



Hledání cesty v grafu (Dijkstrův algoritmus)

Pojďme se podívat ještě na jednu typickou úlohu, která má navíc praktické využití. Půjde o hledání nejkratší cesty v grafu. Jako motivace nám může posloužit příklad grafu ze začátku kapitoly. Tam jsme si ukazovali graf silniční sítě, ve kterém vrcholy odpovídaly jednotlivým městům a hrany silnicím. V této situaci bychom mohli chtít zjistit, jaká je nejkratší cesta z Prahy do Brna. K tomu budeme potřebovat nejen původní graf, ale i informace o délkách jednotlivých hran. Naší úlohu můžeme nyní formulovat v řeči grafů třeba následovně:

5.13 Problém. Mějme dán graf $G = (V, E)$, spolu s funkcí $d : E \to \mathbb{R}$ (udávající délky hran) a dva vrcholy $a, b \in V$. Nalezněte cestu $a_0, \ldots, a_n$ z a do b (t.j. $a_0 = a, a_n = b$) takovou, aby hodnota

$$\sum_{i=0}^{n-1} d(a_i, a_{i+1})$$

byla co nejmenší.

Ukážeme si algoritmus, který vymyslel nizozemský informatik E. W. Dijkstra (1930–2002) (viz [Di59]).

Algoritmus je založen na následující myšlence. Pokud bychom postupně procházeli všechny možné cesty začínající v počátečním vrcholu a do nějakého jiného vrcholu v pořadí od nejkratších k nejdelším, pak bychom ve chvíli, kdy poprvé narazíme na cestu, která končí ve vrcholu b, byli hotovi. Tato cesta by totiž musela být nejkratší — všechny pozdější cesty jsou delší, všechny dřívější skončily v jiném vrcholu.

Algoritmus tedy začne v počátečním vrcholu a postupně prochází v určitém pořadí vrcholy grafu. V průběhu svého výpočtu si pro každý vrchol pamatuje dvě informace. První informací je délka nejlepší cesty z počátečního vrcholu do tohoto vrcholu, kterou zatím našel. Druhou informací je sousední vrchol, ze kterého sem tato nejkratší cesta vedla.

Zbývá určit, v jakém pořadí prochází vrcholy. V každém kroku si z dosud nenavštívených vrcholů vybere ten vrchol, do kterého vede nejkratší cesta. Do něj se přemístí a upraví informace o nejkratší cestě u všech jeho sousedů. Implementace v Pythonu může vypadat například tak, jako na výpisu 5.14.

Jaká je složitost tohoto algoritmu? Algoritmus navštíví každý vrchol nejvýše jednou, tedy while cyklus (řádky 35–57) se provede nejvýše $|V|$, kde $|V|$ je počet vrcholů. V každé iteraci se musí projít všichni sousedé aktuálního vrcholu (řádky 37–42), těch může být nejvýše $|V|$, pak se musí vybrat vrchol s nejkratší cestou (řádek

Algoritmus 5.14 Dijkstrův algoritmus

```python
# -*- coding:utf-8 -*-

def dijkstra( start, cil, graf ):
    """ Funkce dijkstra dostane na vstupu startovni a koncovy vrchol
        (start,cil) a graf, který je reprezentován jako modifikovaný
        seznam sousedů: k sousedům si navíc pamatujeme délky hran, t.j.
        například graf odpovídající obdélníku s hranami délky 2 a 4
        by byl reprezentován takto:

        >>> g = [ [(1,4),(3,2)],
                  [(0,4),(2,2)],
                  [(1,2),(3,4)],
                  [(0,2),(2,4)] ]

        Vrátí nejkratší cestu z vrcholu start do vrcholu cil jako
        posloupnost vrcholu, pokud existuje, jinak vrati None.

        >>> dijkstra(0,2,g)
        [0,1,2]
    """

    # Zde si ukládáme informace o tom, které
    # vrcholy jsme již navštívili
    navstiveno = [False] * len(graf)

    # Zde si ukládáme informace o délkách nejkratších
    # dosud nalezených cest. Na začátku neznáme žádnou
    # cestu mimo cestu ze start do start. Délky jsou tedy
    # nekonečné.
    cesty = [(float('inf'),start)]*len(graf)
    cesty[start]=(0,start)

    v = start
    delka_do_v = 0
    while v is not None:
        navstiveno[v] = True
        for (u,delka_vu) in graf[v]:
            # Pokud je cesta do u přes vrchol v
            # kratší, než dosud nejlepší nalezená
            # cesta do u, musíme upravit cesty
            if cesty[u][0] > delka_do_v+delka_vu:
                cesty[u]=(delka_do_v+delka_vu,v)

        # Vyber z nenavštívený vrcholů ten vrchol, do kterého
        # z vrcholu start vede dosud nejkratší nalezená cesta.
        delka_do_v, v = min([(d,u) for (u,(d,prev)) in enumerate(cesty) if not navstiveno[u]])

        # Pokud každý z nenavštívených vrcholů má nejlepší cestu
        # nekonečně dlouhou, pak to znamená, že do žádného nich
        # nevede cesta. Protože vrchol cil je jednim z nich, vrátíme None
        if delka_do_v == float('inf'):
            return None

        if v == cil:
            # Zrekonstruujeme nejkratší cestu z údajů uložených
            # v seznamu cesty a vrátíme ji
            return postav_cestu(cesty,v)

def postav_cestu(cesty, v):
    current = v
    cesta = []
    while True:
        cesta.append(current)
        delka, prev = cesty[current]
        if prev == current:
            cesta.reverse()
            return cesta
        current = prev
```

46), což v naší implementaci znamená opět $|V|$ kroků. Nakonec se rekonstruuje cesta (řádek 57, volání funkce `postav_cestu`), to se však děje pouze jednou za celý běh a na složitost to nemá vliv. Celkem tedy dostáváme $O(|V|(|V| + |V|)) = O(|V|^2)$.

Za předpokladu, že většina vrcholů má jen málo sousedů, se dá algoritmus vylepšit. Pokud bychom si průběžné informace o nejlepších cestách neukládali jako seznam ale jako haldu, dostali bychom odhad $O((|E| + |V|) \log |V|)$. Ještě lepšího výsledku $O(|E| + |V| \log |V|)$ bychom dosáhli nahrazením haldy její variantou nazývanou *Fibonnaciho halda* (viz [FrTa84]).

Kapitola 6

Složitost podruhé

V předchozích kapitolách jsme se setkali s výběrem některých typických úloh, se kterými se programátoři setkávají. Ke každé takové úloze jsme si ukazovali i postup, jak jí vyřešit a zkoumali efektivnost tohoto postupu. Pro většinu úloh jsme nalezli řešení, které bylo z mnoha hledisek efektivní, nicméně občas jsme zmínili (vzpomeňte třeba úlohu 5.7, kde šlo o nalezení optimálního obarvení grafu), že efektivní postup není znám a naznačili jsme, že možná ani neexistuje. Nyní se budeme tímto problémem zabývat podrobněji.

Když bychom na chvíli zapomněli na efektivnost, mohli bychom alespoň doufat, že každý problém, na který narazíme, bude mít nějaké — třeba i velmi neefektivní — řešení. Ukážeme si, že tato naděje je planá. Existují totiž i velmi jednoduché a prakticky zajímavé problémy, které neumíme vyřešit žádným algoritmem. A to nikoliv protože bychom nebyli dostatečně vynalézaví, ale protože žádný algoritmus, který by úlohu řešil, neexistuje. Hned si jednu takovou úlohu ukážeme. Když jsme se učili pracovat s rekurzí, bylo třeba dát pozor, abychom ošetřili základní případy. Jinak by došlo k tomu, že by se program do nekonečna zacyklil. Podobný problém nastává u while cyklů, pokud si nedáme pozor na cyklící podmínku. Asi každému bude na první pohled jasné, že následující program nikdy neskončí, protože se zacyklí:

```
1 i=0
2 while True:
3   i = i * i
```

Poněkdu zákeřnější může být funkce, která zdánlivě počítá faktoriál:

```
1 def bad_factorial(n):
2     ret = 1
3     while n != 1:
4         ret *= n
5         n = n - 1
6     return ret
```

Funkce možná vypadá správně, ale zkuste si rozmyslet, co se stane, když se pomocí ní pokusím spočítat faktoriál 0. Nejen, že nespočítá správný výsledek 1, ona totiž nespočítá vůbec žádný výsledek — do nekonečna se zacyklí! Zkušenější programátor uvidí chybu okamžitě, ale asi si dokážeme představit, že u složitějšího kódu, který bude mít třeba desítky tisíc řádků, už bude chybu hledat velmi dlouho. Bylo by proto výhodné, kdyby mu práci usnadnil počítač. To vede k následující úloze:

6.1 Úloha. Pro zadaný program a vstupní data určete, zda se program zastaví, nebo se zacyklí.

Když bychom se pustili do řešení, asi bychom se snažili nejprve najít všechny while cykly a nějak analyzovat podmínky, za kterých se cyklus ukončí. Naše funkce by mohla vypadat třeba takto:

```
1 def zacykli( funkce, parametry ):
2     """ Vrátí true, pokud se příkaz
3
4          funkce(parametry)
5
6         zacyklí. V opačném případě vrátí False.
7     """
8     ...
```

Zbývá už jen doplnit zmíněnou analýzu cyklů... Řekněme, že by se nám to povedlo a zkusme se podívat na následující příklad:

```
1 def priklad():
2     if zacykli(priklad, None):
3         return 0
4     else:
5         while True:
6           i = 1
```

Funkce `priklad` vypadá trochu podezřele. V prvním kroku použije naší důmyslnou funkci `zacykli` a zeptá se, co si myslí o volání funkce `priklad`. Pokud je odpovědí, že se zacyklí, pak funkce `priklad` vrátí 1. V opačném případě se zacyklí. Na tomto slovním popisu je už asi vidět, že ta podezřelost je oprávněná, někde dojde k problémům. Co se stane, když nyní na tuto funkci aplikujeme `zacykli`? Dostaneme špatnou odpoveď! Pokud totiž bude odpoveď True, mělo by to znamenat, že funkce `priklad` se zacykli. Jenže když si jí projdeme, pak zjistíme, že se nemohla zacyklit, protože skončila na řádku 3 vrácením 0. No dobrá, tak tedy odpověď musí být False, což má znamenat, že se funkce `priklad` nezacyklí. Ale ouha, to je zase špatně — v tomto případě se totiž funkce `priklad` dostane na řádek 5 a nenávratně se zacyklí.

Literatura

[BFPRT73] Manuel Blum, Robert W Floyd, Vaughan Pratt, Ronald L Rivest, and Robert E Tarjan, *Time bounds for selection*, Journal of Computer and System Sciences **7** (1973), no. 4, 448–461.

[Br79] Daniel Brélaz, *New methods to color the vertices of a graph*, Commun. ACM **22** (1979), no. 4, 251–256.

[Di59] Edsger W Dijkstra, *A note on two problems in connexion with graphs*, Numerische mathematik **1** (1959), no. 1, 269–271.

[FrTa84] Michael L Fredman and Robert Endre Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Proceedings of the 25th Annual Symposium onFoundations of Computer Science, 1984, IEEE Computer Society, 1984, pp. 338–346.

[TAOCP2] Donald Knuth, *The art of computer programming, vol 2.*, Addison-Wesley Pub. Co, Reading, Mass, 1973.

[Ta85] R.E. Tarjan, *Amortized computational complexity*, SIAM Journal on Algebraic and Discrete Methods **6** (1985), no. 2, 306–318.

[To95] P. Töpfer, *Algoritmy a programovací techniky*, Prometheus, 1995.

[Wi85] Niklaus Wirth, *Algorithms + Data Structures = Programs*, vol. 76, Prentice Hall, 1985.

Lidé

Biografické údaje jsou převzaté z Wikipedie.

Dijkstra, Edsger Wybe (1930–2002) Nizozemský informatik, navrhl algoritmus pro hledání nejkratší cesty v grafu, přišel s myšlenkou tzv. *semaforu*. Článkem *Go To Statement Considered Harmful* se zasloužil o zavržení tohoto příkazu a rozvoj strukturovaného programování. Byl znám svými číslovanými rukopisy, jejichž kopie kolovaly po prakticky celé informatické komunitě. Každý takový rukopis byl označen zkratkou EWD a pořadovým číslem.

Euklides, řec. $E\upsilon\kappa\lambda\varepsilon\iota\delta\eta\varsigma$ **(300 př. Kr.)** Antický filozof a matematik.

Kleene, Stephen Cole (1909–1994)

L-Systém
(obrázek na obálce)

L-systém nebo také Lindenmayerův systém je varianta formální gramatiky, vyvinutá pro modelování růstu rostlin. L-systém popisuje pravidla pro vývoj rostliny, která se opakovaně aplikují na vznikající model. Tato pravidla mohou např. popisovat, za jakých podmínek se stonek rostliny rozdvojí, zda má vzniknout list nebo zda má část rostliny uhynout. Výsledný model se může např. vykreslit jako obrázek nebo se z něj vytvoří počítačový 3D model rostliny. L-systémy se také dají použít pro generování různých křivek, fraktálů nebo pro modelování buněčných organismů.

Wikipedie

Obrázek na obálce vznikl trojnásobnou aplikací pravidla F=FF-[-F+F+F]+[+F-F-F] na axiom ++F. Písmeno F značí krok dopředu, znak + značí otočení vlevo o 60°, znak - otočení vpravo o 16°, znak [zapamatování bodu a znak] návrat k zapamatovanému bodu.