KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

PROGRAMY A PROJEKY V JAZYKU SCHEME II

DAVID SKOUPIL

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc 2007

Abstrakt

Tento učební text seznamuje čtenáře se základy programování v prostředí jazyka Scheme. Obsahuje řadu příkladů a projektů, při jejichž řešení musí mít čtenář k dispozici počítač a interpretr programovacího jazyka. Předpokládá se přitom využití interpreteru DrScheme, většina textu je však využitelná i na jiných platformách. Text je koncipován zejména jako cvičebnice jazyka, neklade si za cíl nahradit učebnice algoritmizace ani referenční manuál jazyka. V této oblasti existují další učební texty a zahraničí publikace (viz [Abelson96], [R5RS]). Studující použije tento text jako úvodní pomůcku pro zvládnutí jazyka Scheme a dále jako doprovodný materiál při studiu pokročilejších publikací.

Druhý díl této řady se věnuje problematice výpočetního procesu a jeho složitosti.

Cílová skupina

Text je primárně určen pro posluchače prvního ročníku bakalářského studijního programu Aplikovaná informatika na Přírodovědecké fakultě Univerzity Palackého v Olomouci. Může však sloužit komukoli se zájmem o počítače a programování v jazyku Scheme. Text předpokládá zvládnutí publikace [Skoupil04A].

Obsah

1 Program a výpočetní proces

1.1 Rekurzivní a iterativní výpočetní proces

Studijní cíle: Po prostudování kapitoly studující porozumí pojmu výpočetní proces. Bude schopen vytvářet procedury, které generují rekurzivní nebo iterativní výpočetní proces.

Klíčová slova: Rekurzivní výpočetní proces, iterativní výpočetní proces, koncová rekurze.

Potřebný čas: 4 hodiny.

Některé programy jsou na pohled velmi jednoduché a mají jen několik programových řádků, zatímco jiné jsou velmi složité a zabírají celé stránky kódu. Některé programy program vykoná ve zlomku vteřiny zatímco jiné programy běží dlouhé minuty či hodiny. Není přitom pravda, že krátké programy běží rychle zatímco dlouhé programy běží pomalu. Vztah mezi programem a jeho vykonáváním je podstatně složitější a zaslouží si vlastní kapitolu.

1.1.1 Výpočetní proces

Výpočetní proces je řízen počítačovým programem.

Počítačový program je statická entita. Program lze vytisknout na papír, uložit na disk, spočítat kolik má bytů nebo řádků. Program na disketě nebo na papíře existuje, i když není v dohledu žádný počítač.

Zatímco počítačový program představuje zcela konkrétní pojem, *výpočetní proces* je jen obtížně uchopitelný. Výpočetní proces je dynamická entita. Výpočetní proces vzniká tím, že počítač vykonává daný program. Výpočetní proces má v čase začátek a konec, můžeme tak určit délku jeho trvání. Výpočetní proces zabírá v počítači paměť a další zdroje.

> **Průvodce studiem**
>
> *V jisté analogii můžeme výpočetní proces přirovnat k myšlenkovým procesům člověka. Takovéto procesy lze popisovat a kategorizovat, nelze je však vzít, uložit na disk a spočítat jejich velikost. Výpočetní proces představuje „duši" počítače.*

Různé programy mají za následek svého spuštění různé výpočetní procesy. Říkáme, že program generuje výpočetní proces. V této kapitole zjistíme, jaké základní typy procesů existují a jak z programu poznáme, který výpočetní proces bude generován.

1.1.2 Fibonacciho čísla

Fibonacciho čísla.

Výpočetní proces demonstrujeme na jednoduchém příkladě. N-té *Fibonacciho číslo* je matematicky definováno rekurzivním vzorcem

$$Fib(n) = \begin{cases} 0, n = 0 \\ 1, n = 1 \\ Fib(n-2) + Fib(n-1) \end{cases}$$

První dvě Fibonacciho čísla jsou tedy 0 a 1, všechna ostatní vzniknou vždy součtem předchozích dvou. Fibonacciho řadu tedy tvoří čísla 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 atd.

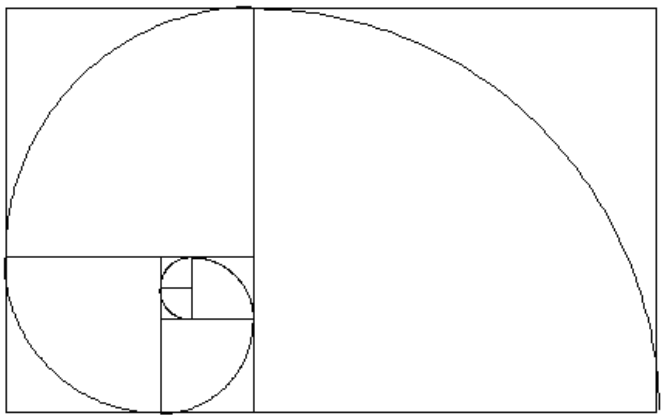
Tuto definici lze téměř mechanicky převést na program v jazyku Scheme:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
         (else (+ (fib (- n 2))
                  (fib (- n 1))))))
```

Průvodce studiem

Podle legendy byla tato řada objevena italským učencem 12. století, Leonardem Bonaccim, který jako malý chlapec plánoval chov králíků. Jeho strategie byla taková, že si z každého vrhu ponechá vždy jeden pár a zbytek prodá. Králíci mají mladé každého půl roku a trvá jim rok než dospějí. Aby správně odhadl velikost kotců, potřeboval zjistit kolik párů králíků bude mít za danou jednotku času. Začínal s nulou, pak koupil na trhu jeden pár králíků. Za půl roku měl stále tento první pár, ale za rok se již narodila první mláďata; měl tedy dva páry. Za dalšího půl roku přibyli mláďatům sourozenci a Leonardo měl tři páry. Za dalšího půl roku se narodila mláďata králíkům z první i z druhé generace. Počet párů králíků je popsán Fibonacciho řadou.

Fibonacciho řada patří k základním přírodním jevům. Obrázek Obr. 1 ukazuje tzv. logaritmickou spirálu, která vznikne propojením čtverců, jejichž velikosti stran sledují Fibonacciho posloupnost. Obrázek Obr. 2 ukazuje ulitu mořského měkkýše Nautilus Mollusc .



Obr. 1 Logaritmická spirála

Fibonacciho řada představuje základní přírodní fenomén.

Obr. 2 Ulita mořského měkkýše Nautilus Mollusc

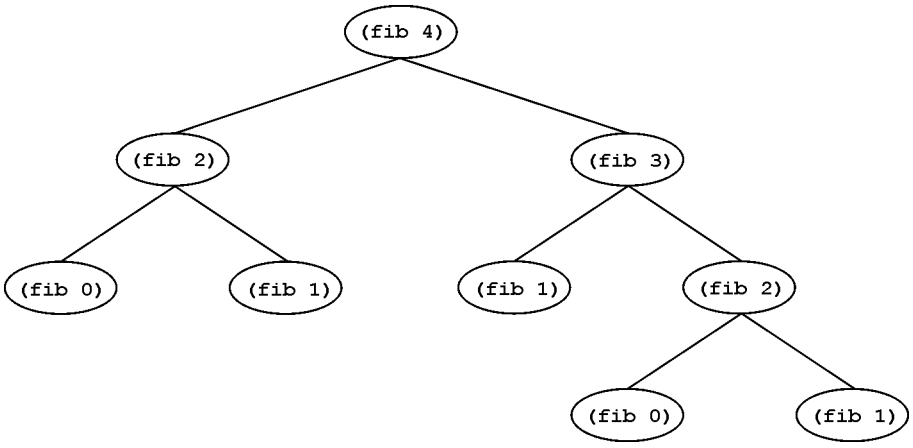
Větší Fibonacciho čísla je obtížné vypočítat.

Program na výpočet Fibonacciho čísel je sice správný, přesto jej ověřte na počítači a vypočítejte desáté (55), dvacáté (6765), třicáté (832040), čtyřicáté (102334155) a padesáté (12586269025) Fibonacciho číslo. Při výpočtech narazíte na problém. Zatímco desáté a dvacáté číslo jde spočítat velmi snadno, na třicáté si už musíte chvíli počkat (podle rychlosti vašeho počítače). Čtyřicáté a padesáté Fibonacciho číslo již tímto algoritmem těžko spočítáte. I když je program na výpočet Fibonacciho čísel velmi jednoduchý, výpočetní proces pro větší vstupy trvá velmi dlouho.

1.1.3 Stromově rekurzivní výpočetní proces

Procedura na výpočet Fibonacciho čísle generuje stromově rekurzivní výpočetní proces.

Pokusme se vykreslit všechna vyvolání procedury `fib`, která jsou potřeba pro zjištění čtvrtého Fibonacciho čísla (viz obrázek Obr. 3). Napočítáme celkem 9 vyvolání procedury. „Tvar výpočtu“ má přitom podobu stromu – v každém uzlu se výpočet rozděluje na dvě nezávislé větve. Takovýto výpočetní proces nazveme *stromově rekurzivní*. Procedury, které generují stromově rekurzivní výpočetní procesy, poznáme snadno tak, že rekurzivní předpis obsahuje alespoň dvě rekurzivní volání.



Obr. 3 Aktivační strom procedury `fib`

Vyvolávání jednotlivých aktivací procedury `fib` můžeme sledovat i na výpisu trasování. Výpis má „hrbolatý tvar“, střídá se několik fází navíjení a odvíjení, podle toho, jak výpočetní proces prochází jednotlivými větvemi aktivačního stromu.

```
> (fib 4)
|(fib 4)
| (fib 2)
| |(fib 0)
| |0
| |(fib 1)
| |1
| 1
| (fib 3)
| |(fib 1)
| |1
| |(fib 2)
| | (fib 0)
| | 0
| | (fib 1)
| | 1
| |1
| 2
|3
```

1.1.4 Lineárně rekurzivní výpočetní proces

Porovnejme tento výsledek trasování s výsledkem trasování následující procedury na výpočet faktoriálu.

Procedura na výpočet faktoriálu generuje lineárně rekurzivní výpočetní proces.

```
(define (fakt n)
  (if (= n 0) 1
    (* n (fakt (- n 1)))))
```

```
> (fakt 3)
|(fakt 3)
| (fakt 2)
| |(fakt 1)
| | (fakt 0)
| | 1
| |1
| 2
|6
6
```

Zatímco trasování výpočtu Fibonacciho čísel mělo „hrbolatý tvar“, faktoriál je spočítán ve dvou fázích: výpočet se v první fázi (fázi navíjení) hladce dostane až k mezní podmínce rekurze a v druhé fázi (fázi odvíjení) spočte výsledek. Takovýto výpočetní proces nazveme *lineárně rekurzivní*. Procedury, které generují lineárně rekurzivní výpočetní procesy poznáme tak, že rekurzivní předpis obsahuje pouze jedno rekurzivní volání.

1.1.5 Iterativní výpočetní proces

Proceduru na výpočet faktoriálu lze ale napsat také následujícím způsobem. Využijeme přitom pomocnou proceduru.

Některé procedury lze zapsat tak, aby generovaly iterativní výpočetní proces.

```
(define (fakt-iter p n)
  (if (= n 0) p
      (fakt-iter (* p n) (- n 1))))

(define (fakt n)
  (fakt-iter 1 n))
```

Čistě z matematického hlediska není mezi oběma implementacemi procedury `fakt` žádný rozdíl – obě implementují stejnou matematickou funkci a vrací pro stejné vstupy identické výstupy. Z hlediska informatiky se však jedná o zcela jiné procedury. Již na první pohled je zá-

pis druhé procedury podstatně složitější a není zcela zřejmé, co program vlastně dělá. Procedura `fakt` slouží pouze jako obal, ve svém těle volá proceduru `fakt-iter` a dosazuje jedničku za parametr `p`. Procedura `fakt-iter` v každém kroku násobí parametr `p` číslem `n`. Když `n` dosáhne mezní podmínky rekurze, číslo `p` v sobě obsahuje výsledek celého výpočtu. Trasování výpočtu nám lépe prozradí hru jednotlivých parametrů:

```
|(fakt-iter 1 3)
| (fakt-iter 3 2)
| |(fakt-iter 6 1)
| | (fakt-iter 6 0)
| | 6
| |6
| 6
|6
```

Tento výpočetní proces vykoná veškerou svoji práci ve fázi navíjení. Ve fázi odvíjení již neprobíhá nic zajímavého, pouze se předává zpět známá hodnota výsledku. Překladač jazyka Scheme dokáže takovýto výpočetní proces optimalizovat a fázi odvíjení zcela vynechat. Výpočetní proces pak ve skutečnosti vypadá takto:

```
|(fakt-iter 1 3)
|(fakt-iter 3 2)
|(fakt-iter 6 1)
|(fakt-iter 6 0)
|6
```

Iterativní výpočetní proces probíhá v konstantním zásobníkovém prostoru.

Tento typ výpočetního procesu nazveme *iterativní* výpočetní proces nebo zkráceně *iterace*. Iterace má několik zajímavých rysů.

- V průběhu výpočtu není potřeba alokovat na zásobníku žádné nové záznamy, protože aktuální hodnoty parametrů nebudou ve fázi odvíjení zapotřebí. Překladači stačí, aby pouze měnil hodnoty parametrů v jediném zásobníkovém záznamu.[1] Program tak může běžet libovolně dlouho bez rizika vyčerpání paměti.
- Protože odpadá režie spojená s vytvářením a rušením zásobníkových záznamů, probíhá iterativní výpočetní proces rychleji.
- Výpočet je možno v libovolném kroku přerušit a opět spustit. Veškeré stavové informace výpočtu jsou skryty v parametrech. Pokud bychom přerušili rekurzivní výpočetní proces, ztratíme údaje na zásobníku a výpočet nelze obnovit.

Koncově rekurzivní procedury generují iterativní výpočetní proces.

Otázkou zůstává, jak překladač pozná, že se jedná o iterativní a nikoli rekurzivní výpočetní proces. Rozhodující přitom musí být samotný tvar programu. Porovnáme-li kódy obou verzí procedury `fakt`, zjistíme, že podstatný rozdíl je ve tvaru rekurzivního předpisu. Zatímco u rekurzivního procesu je rekurzivní volání pouze parametrem volání procedury `+`, u iterativního procesu je rekurzivní předpis tvořen pouze rekurzivním voláním. Takovýto zápis programu nazveme *koncově rekurzivní*. Můžeme také říci, že koncově rekurzivní procedura vrací jako výsledek volání sama sebe.

Průvodce studiem

Na vytvoření iterace má většina programovacích jazyků speciální syntaktický aparát. – podporu tzv. cyklů. I když standard jazyka Scheme obsahuje příkaz cyklu, my jej nebudeme využívat a pro stejný účel použijeme koncové rekurze.

[1] Říkáme, že iterativní výpočetní proces je vykonáván v konstantním zásobníkovém prostoru.

Mnoho výpočtů lze realizovat jak rekurzivním, tak iterativním výpočetním procesem. Demonstrujme si vytvoření iterativní verze procedury `fib` na výpočet n-tého Fibonacciho čísla.[1]

```
(define (fib-iter a b n)
  (if (= n 0) a
      (fib-iter b (+ a b) (- n 1))))

(define (fib n)
  (fib-iter 0 1 n))
```

Výpočet Fibonacciho čísel iterativním algoritmem je efektivní.

Výkonná procedura akceptuje dva parametry jako střádače hodnot, třetí parametr slouží jako počítadlo. Střádače na počátku naplníme hodnotami nultého a prvního čísla řady. v rekurzivním volání pak do prvního střádače uložíme obsah druhého, do druhého pak součet hodnot obou střádačů. Podívejme se na výsledek trasování výpočtu čtvrtého Fibonacciho čísla.

```
|(fib-iter 0 1 4)
| (fib-iter 1 1 3)
| |(fib-iter 1 2 2)
| | (fib-iter 2 3 1)
| | |(fib-iter 3 5 0)
| | |3
| | 3
| |3
| 3
|3
```

Tvar výpočtu jednoznačně prozrazuje iterativní výpočetní proces. Pokusme se nyní opět vypočítat desáté až padesáté číslo Fibonacciho posloupnosti, tentokrát s využitím naší nové procedury. Zatímco předchozí algoritmus potřeboval pro výpočet 4. Fibonacciho čísla 9 kroků, tento potřebuje kroků 5. Pro výpočet 50. čísla by předchozí algoritmus vyžadoval více než 20 miliard kroků, zatímco tomuto stačí 50 kroků. Podrobněji se této problematice budeme věnovat v kapitole 2.1.

Průvodce studiem

Přepis z rekurzivního do iterativního tvaru sice není zcela mechanický, existuje zde však několik pomůcek. Iterativní procedura bude oproti rekurzivní vždy potřebovat o jeden parametr navíc. Parametr inicializujeme hodnotou výsledku při dosažení mezní podmínky rekurze. V každém kroku pak do parametru postupně střádáme výsledek a v mezní podmínce rekurze jej nakonec vrátíme jako výstup z celé procedury.

Pokud si procedura potřebuje „pamatovat“ více předchozích stavů, jako tomu bylo u procedury na výpočet Fibonacciho čísel, potřebujeme přidat více parametrů.

Shrnutí

Výpočetní proces představuje akce, které provádí počítač při vykonávání daného programu. Rozlišujeme stromově rekurzivní, lineárně rekurzivní a iterativní výpočetní proces. Iterativní proces je generován koncově rekurzivní procedurou. Některé procedury lze zapsat v rekurzivním i iterativním tvaru. Zatímco rekurzivní zápis bývá přehlednější, iterativní je podstatně efektivnější.

[1] I v tomto případě bychom mohli použít interní definici pomocné procedury.

Pojmy k zapamatování

- Výpočetní proces,
- stromově rekurzivní výpočetní proces, lineárně rekurzivní výpočetní proces, iterativní výpočetní proces,
- koncová rekurze,
- Fibonacciho čísla.

Kontrolní otázky

1. *Jak je definováno n-té Fibonacciho číslo?*
2. *Jak z programu poznáme, bude-li program generovat stromově rekurzivní nebo lineárně rekurzivní výpočetní proces?*
3. *Jak poznáme, že je procedura koncově rekurzivní?*
4. *Jaký je obecný postup při převodu programu z rekurzivního na iterativní tvar?*

Cvičení

1. Jaký typ výpočetního procesu generuje procedura `soucet-sude` z publikace [Skoupil04A]?

2. Perrinova čísla jsou dána následující rekurzivní formulí.

$$P(n) = \begin{cases} P(0) = 3 \\ P(1) = 0 \\ P(2) = 2 \\ P(n) = P(n-2) + P(n-3) \end{cases}$$

   Prvních deset Perrinových čísel je 3, 0, 2, 3, 2, 5, 5, 7, 10 a 12. Napište rekurzivní a iterativní proceduru na výpočet n-tého Perrinova čísla.

Úkoly k textu

1. Napište iterativní verzi procedur `soucet-od-do` a `soucet-sude`.

2. Následující vzor se nazývá Pascalův trojúhelník.

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

   Čísla na hranách trojúhelníku jsou vždy jedničky a každé číslo uvnitř trojúhelníku je tvořeno jako součet dvou čísel nad ním. Číslo 6 na poslední řádce například vzniklo jako součet čísel 3 a 3.Vytvořte rekurzivní proceduru `pascal`, která akceptuje číslo řádku a sloupce a vrátí číslo na dané pozici. Například:

```
> (pascal 4 2)
3
> (pascal 5 3)
6
```

Řešení

1. Procedura `soucet-sude` sice obsahuje ve svém těle dvě rekurzivní volání, při vykonávání těla procedury se však vykoná pouze jedno z nich. Procedura proto generuje lineárně rekurzivní výpočetní proces.

2. Rekurzivní verzi procedury lze vytvořit v podstatě mechanicky:

```
(define (perrin n)
  (cond ((= n 0) 3)
        ((= n 1) 0)
        ((= n 2) 2)
        (else (+ (perrin (- n 2))
                 (perrin (- n 3))))))
```

   Iterativní verze je poněkud složitější. Musíme použít stejnou techniku jakou jsme použili pro iterativní verzi Fibonacciho čísel. Zatímco u Fibonacciho čísel si stačilo pamatovat dva poslední stavy, zde je potřeba uvažovat stavy tři.

```
(define (perrin-iter a b c n)
  (if (= n 0) a
      (perrin-iter b c (+ a b) (- n 1))))

(define (perrin n)
  (perrin-iter 3 0 2 n))
```

1.2 Projekt: Hanojské věže

Studijní cíle: Při řešení projektu si studující procvičí analýzu rekurzivních problémů a tvorbu rekurzivních procedur. Naučí se odhadovat dobu výpočtu a uvědomí si náročnost stromově rekurzivního výpočetního procesu.

Klíčová slova: Hanojské věže, stromově rekurzivní výpočetní proces.

Potřebný čas: 2 hodiny.

1.2.1 Legenda

Hlavolam Hanojských věží je založen na legendě.

Tento projekt je založen na legendě. V dávných dobách existoval ve starobylém asijském městě buddhistický klášter. Mniši tohoto kláštera měli za úkol přestěhovat pyramidu 64 posvátných disků z jednoho místa chrámu do druhého. Disky byly těžké a křehké, takže mniši nikdy nemohli naráz přesunout více než jeden disk. Větší disk navíc nesměl být položen na menší, méně cenný disk. V chrámu bylo přitom jen jediné místo (kromě původního a cílového místa), které bylo dostatečně posvátné na to, aby zde byla pyramida disků dočasně uložena.

A tak mniši začali stěhovat disky sem a tam mezi původním místem, novým místem a dočasným úložištěm a důsledně dbali na to, aby nikdy nepoložili větší disk na menší. Tři pyramidy disků se zvětšovaly a zase zmenšovaly. Legenda říká, že až mnichové přemístí poslední disk na nové místo, klášter se změní na prach a nastane konec světa.

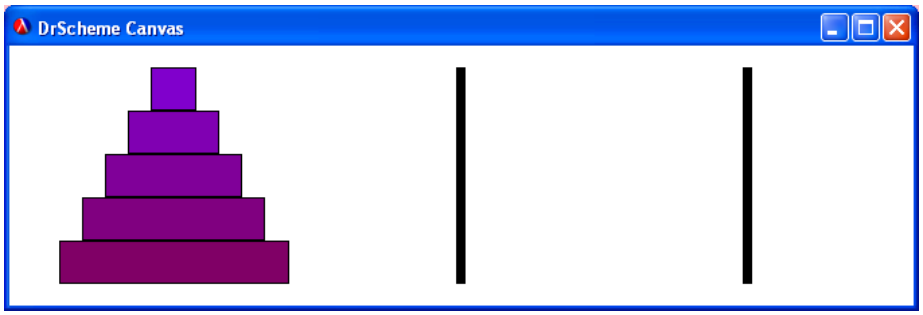
1.2.2 Podpůrný teachpack

Pro demonstraci problému Hanojských věží použijeme teachpack `hanoi.scm`. Tento teachpack nám nabízí několik procedur:

- `init-window` – akceptuje parametr `n` a otevře grafické okno dostatečně velké na to, aby se do něj vešlo `n` disků

- `init-towers` – také akceptuje parametr `n` a do otevřeného grafického okna zobrazí tři lokace (tři kolíky) a na první z nich vykreslí `n` disků. Kolíky mají čísla 1, 2 a 3.
- `move` – akceptuje dva parametry: číslo zdrojového a cílového kolíku a přemístí jeden disk ze zdrojového kolíku na cílový. Procedura kontroluje pravidla hry, nelze proto přemístit větší kolík na menší kolík.
- `fast-move` – funguje stejně jako procedura `move`, probíhá však rychleji.

Obrázek Obr. 4 ukazuje výsledek volání procedur `init-window` a `init-towers` s parametrem 5. Naším cílem je napsat program, který přemístí všech 5 disků z prvního kolíku (vlevo) na třetí kolík (vpravo) za pomocí druhého kolíku (uprostřed). Program by měl využívat proceduru `move` pro přesun jednoho disku.



Obr. 4 Hanojské věže

Průvodce studiem

Vyzkoušejte si inicializaci okna a věží a pokuste se manuálně přesunout disky pomocí procedury `move`. Ověřte, že procedura `move` nedovolí provedení ilegálního tahu.

1.2.3 Algoritmus

Manuální řešení problému Hanojských věží je obtížné.

Pokud bychom chtěli **manuálně** vyřešit hlavolam například pro tři disky, museli bychom vykonat následující kód:

```
(init-window 3)
(init-towers 3)
(define (transfer3)
  (move 1 3)
  (move 1 2)
  (move 3 2)
  (move 1 3)
  (move 2 1)
  (move 2 3)
  (move 1 3))
(transfer3)
```

Napište tento kód do prostředí jazyka Scheme a vyzkoušejte jej.

Úkolem tohoto projektu je však vytvořit **obecný** algoritmus, který přesune jakékoli množství disků z libovolného kolíku na libovolný jiný kolík. Procedura se může jmenovat `transfer` a pro přesun 5 disků z kolíku 1 na kolík 3 pomocí kolíku 2 bychom zavolali:

```
(transfer 5 1 2 3)
```

Průvodce studiem

Pokud tento problém neznáte z dřívějšího studia, asi vám v tuto chvíli připadne řešení nesmírně vzdálené a složité. Algoritmus není evidentní a programátor neví kde začít. O to překvapující je, že program má jen pár řádků a celý trik je skryt v důmyslném použití rekurze.

Problém lze stručně a elegantně vyřešit pomocí rekurze.

Algoritmus na transfer disků lze slovy vyjádřit takto:

- Máme-li za úkol přesunout 0 disků, není třeba nic dělat
- Máme-li přesunout n disků z kolíku 1 na kolík 3 pomocí kolíku 2, postupujeme ve třech krocích:
- Přesuneme n-1 disků z kolíku 1 na kolík 2 pomocí kolíku 3
- Poslední zbývající disk na kolíku 1 odneseme na kolík 3
- Přesuneme oněch n-1 disků, co nám zbyly na kolíku 2, na kolík 3 za pomoci kolíku 1

Tento algoritmus lze lehce transformovat do procedury v jazyku Scheme. Zde uvedeme pouze základní strukturu této procedury.

```
(define (transfer n from via to)
  (if (> n 0)
      (begin
        .........
        .........
        .........)))
```

Úkoly k textu

1. Doplňte chybějící místa v proceduře `transfer`. Proceduru otestujte na přesunu 1, 2, 3, 4, 5, 6, 7 a 8 disků.

1.2.4 Výpočetní proces

Řešení problému Hanojských věží má exponenciální složitost.

Procedura transfer generuje stromově rekurzivní výpočetní proces. Nastává zde přitom stejný problém jako u výpočtu Fibonacciho čísel rekurzivním výpočetním procesem (viz kapitola 1.1.2): pro větší vstupy trvá výpočet velmi dlouho. Můžete zkusit použít proceduru na přesun většího počtu disků (například 20 nebo 30). Abyste urychlili dobu výpočtu, zaměňte ve vaší proceduře `transfer` proceduru `move`, která trvá vždy jednu vteřinu, za proceduru `fast-move`. I tak bude přesun většího počtu disků problematický.

Dobu výpočtu lze v jazyku Scheme jednoduše měřit pomocí speciální formy `time`. Tato speciální forma akceptuje libovolný výraz, vyhodnotí jej a vypíše čas, který byl potřebný pro toto vyhodnocení. Vyzkoušejte například:

```
> (time (transfer 15 1 2 3))
cpu time: 18186 real time: 21531 gc time: 820
```

Reálný čas je uveden v milisekundách.

Úkoly k textu

2. Spusťte znovu proceduru `transfer` pro přesun 1, 2, 3 a 4 disků a spočítejte potřebný počet tahů. Odvoďte obecný vzorec určující závislost mezi číslem `n` a potřebným počtem tahů.
3. Změřte, jak dlouho na vašem počítači trvá proceduře `transfer` přesunout 10, 11, 12, 13, 14 a 15 disků (za využití procedury `fast-move`). Odvoďte, jak dlouho trvá jediný tah (přesun jediného disku). Na základě toho vypočítejte, jak dlouho by trvalo (v hodinách, dnech nebo rocích) proceduře `transfer` přesunout 30, 40 a 50 disků.
4. Předpokládejme, že mnichům trvá přesunutí jednoho disku jednu minutu. Pokud pracují bez přestávky 24 hodin denně, kdy nastane podle legendy konec světa?

Shrnutí

Hanojské věže představují hlavolam, který je velmi obtížně řešitelný bez pomoci počítače. Na počítači je přitom řešení velmi snadné za použití rekurze. Tento program zároveň slouží jako příklad velmi jednoduchého a krátkého programu, jehož vykonávání pro větší vstupy trvá velmi dlouho i na rychlém počítači.

1.3 Projekt: Perfektní a spřátelená čísla

Studijní cíle: Při řešení tohoto projektu si studující procvičí tvorbu rekurzivních procedur, opakované využití kódu a odhad času potřebného pro výpočetní proces.

Klíčová slova: Perfektní čísla, spřátelená čísla.

Potřebný čas: 1 hodina.

1.3.1 Perfektní čísla

Přirozené číslo se nazývá perfektní, pokud se rovná součtu všech svých dělitelů (včetně čísla 1). Například dělitelé čísla 6 jsou čísla 3, 2 a 1. Protože 6 = 3 +2 +1, šest je perfektní číslo.

Průvodce studiem

Perfektní čísla jsou studována od dob antických a byl jim vždy připisován velký význam teoretický i mystický. Perfektní čísla mají zajímavé vlastnosti a blízký vztah například k tzv. Mersenovým prvočíslům (viz kapitola 2.4). Již Euklides dokázal, že jestliže $2^k - 1$ *je prvočíslem, pak* $2^{k-1}(2^k - 1)$ *je perfektní číslo. Doposud všechna známá perfektní čísla jsou sudá a není známo, jestli existují i lichá perfektní čísla.*

Následující seznam vypočítává prvních deset perfektních čísel:

Perfektní čísla mají velký teoretický význam.

- 6,
- 28,
- 496,
- 8128,

- 33550336,
- 8589869056,
- 137438691328, 2305843008139952128,
- 2658455991569831744654692615953842176,
- 191561942608236107294793378084303638130997321548169216.

Úkoly k textu

1. Napište predikát `delitel?`, který otestuje, jestli číslo `x` je dělitelem čísla `y`. Například `(delitel 2 4)` se vyhodnotí na `#t` protože 2 je dělitelem čísla 4.

2. Napište proceduru `soucet-delitelu`, která akceptuje číslo `n` a vrátí součet všech jeho dělitelů.

3. Za pomoci procedury `soucet-delitelu` napište predikát `perfektni?`, který otestuje jestli je číslo perfektní nebo není.

4. Jak dlouho trvá vašemu počítači ověření perfektnosti čtvrtého a pátého perfektního čísla? Použijte opět speciální formu `time` pro měření výpočetní doby. Vypočítejte, jak dlouho by trvalo ověření devátého nebo desátého perfektního čísla

1.3.2 Spřátelená čísla

Na problému perfektních čísel představujeme problém hledání dělitelů.

Dvě čísla se nazývají spřátelená, pokud součet všech dělitelů jednoho čísla (včetně jedničky) je roven druhému číslu a naopak. Například 220 a 284 jsou spřátelená čísla protože:

- dělitelé čísla 284 jsou čísla 1, 2, 4, 71 a 142,
- součet 1 + 2 + 4 + 71 + 142 = 220,
- dělitelé čísla 220 jsou čísla 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 a 110,
- součet 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284.

Další páry spřátelených čísel jsou například 1184 a 1210, 2620 a 2924, 5020 a 5564 nebo 6232 a 6368.

Úkoly k textu

5. Napište predikát `spratelena?`, který otestuje jestli jsou dvě čísla spřátelená nebo nikoli. Využijte přitom proceduru `soucet-delitelu`.

Shrnutí

Spřátelená a perfektní čísla představují zajímavý fenomén v teorii čísel. Tento projekt ukazuje, jak využít proceduru na součet dělitelů pro testování obou vlastností čísel. Zároveň jsme ověřili, že navržený algoritmus nelze použít na větší čísla vzhledem k jeho nízké efektivitě.

2 Složitost výpočtu

2.1 Zavedení složitosti výpočtu

Studijní cíle: Po absolvování této kapitoly bude studující seznámen s pojmem složitosti výpočtu. Bude schopen určit složitost procedur pomocí O notace.

Klíčová slova: Složitost výpočtu, O notace.

Potřebný čas: 2 hodiny.

Složitost vyjadřuje vztah mezi spotřebovanými zdroji a vstupem.

V kapitole 1.1 jsme na příkladu Fibonacciho čísel viděli, že výpočet stejné hodnoty může být velmi pomalý nebo velmi rychlý podle toho, jaký zvolíme algoritmus. Pojmy „pomalý“ a „rychlý“ však nejsou dostatečně přesné pro popis algoritmu. V této kapitole použijeme *složitost výpočtu* pro přesný popis zdrojů, které výpočetní proces generovaný daným programem zabere. Mezi *zdroje* řadíme především čas a paměť počítače, které jsou potřebné na výpočet. Hovoříme tak o *časové složitosti výpočtu* a *prostorové složitosti výpočtu*.

Průvodce studiem

Obecně bychom mohli uvažovat i jiné zdroje, které může výpočetní proces spotřebovávat. U distribuovaných výpočtů je například velmi citlivým zdrojem množství síťové komunikace nutné pro výpočet. U mobilních zařízení je zase podstatné množství spotřebované energie.

2.1.1 Definice složitosti a O notace

Složitost výpočtu vyjadřujeme pomocí O notace.

Čas, který proces zabere, lze změřit ve vteřinách a minutách. Můžeme například říci, že výpočet 32. Fibonacciho čísla rekurzivním algoritmem trval 15 vteřin. Obdobně můžeme vyjádřit spotřebovanou paměť v bytech. Tento popis je však z několika důvodů nevyhovující.

- Závisí na typu použitého počítače a jeho rychlosti. Na jiném počítači může výpočet daného čísla trvat zcela jinou dobu.
- Nic nám neprozrazuje o tom, jak dlouho bude trvat výpočet 31. nebo 33. Fibonacciho čísla.

Nezajímá nás tedy pouze absolutní velikost spotřebovaných zdrojů pro jeden konkrétní vstup (např. doba trvání výpočtu), ale **vztah** mezi velikostí vstupu a spotřebovanými zdroji. Tento vztah lze vyjádřit matematickou funkcí.

V případě výpočtu Fibonacciho čísel lze dokázat, že spotřebovaný čas je přibližně roven výrazu $k(1.618^n)$. Vztah mezi velikostí vstupu a spotřebovaným časem lze tedy popsat exponenciální funkcí $f(n) = k(1.618^n)$. Konstanta k lze slouží k tomu, abychom zohlednili různou rychlost jednotlivých počítačů.

O notace uvolňuje "těsný" funkční vztah mezi zdroji a vstupem.

Pro definici složitosti výpočtu zavádíme tzv. *O notaci*. Řekneme-li, že výpočetní proces má složitost O(f(n)), znamená to, že od jisté hodnoty n výše je množství spotřebovaných zdrojů vždy menší než hodnota $k(f(n))$. O notace nám tedy uvolňuje přísný funkční vztah mezi velikostí vstupu a velikostí spotřebovaných zdrojů ve třech aspektech:

- vztah musí platit až od jistého čísla n dále, pro triviální případy malých hodnot n to může být jinak,
- velikost spotřebovaných zdrojů omezujeme shora, pro některé vstupy může být množství času i menší,
- odhlížíme od násobení konstantou.

2.1.2 Typy složitostí

Funkce, které popisují vztah mezi velikostí vstupu a množstvím spotřebovaných zdrojů spadají do několika základních kategorií.[1]

Existuje několik základních typů složitosti výpočtu.

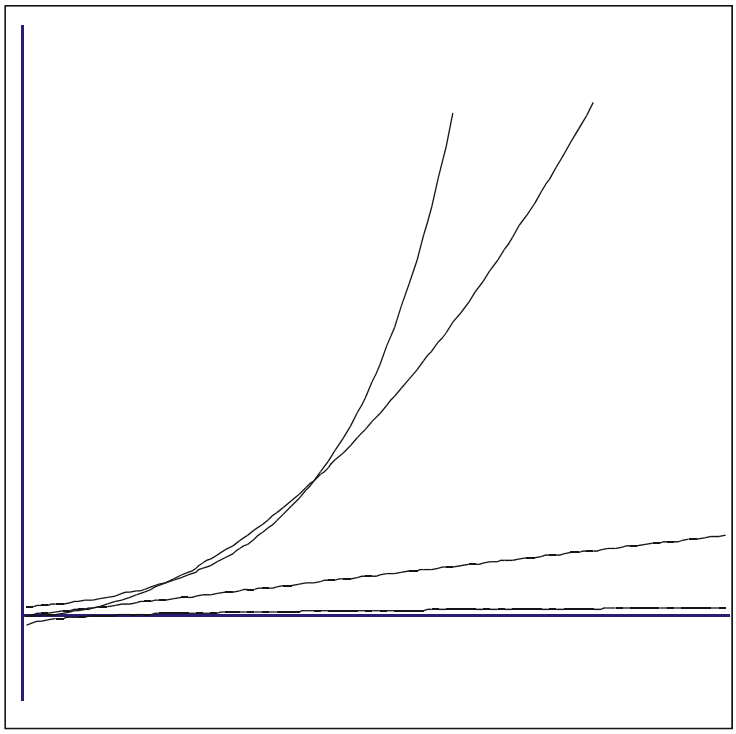
1. Konstantní funkce, konstantní složitost. Výpočet pro libovolnou velikost vstupních dat spotřebuje vždy stejné množství zdrojů. Všechny iterativní výpočty mají konstantní prostorovou složitost. Označujeme $O(1)$.
2. Lineární funkce, lineární složitost. Množství spotřebovaných zdrojů roste lineárně s velikostí problému, vztah popisuje lineární funkce. Vzroste-li velikost vstupu desetkrát, můžeme očekávat desetinásobně větší spotřebu zdrojů. Lineární časovou složitost mají například oba představené programy na výpočet faktoriálu. Označujeme $O(n)$.
3. Kvadratická funkce, kvadratická složitost. Množství spotřebovaných zdrojů je závislé na druhé mocnině velikosti vstupu. Vzroste-li velikost vstupu desetkrát, můžeme očekávat stonásobně větší spotřebu zdrojů. Kvadratickou složitost mají typicky jednoduché třídící algoritmy. Označujeme $O(n^2)$.
4. Exponenciální funkce, exponenciální složitost. Množství spotřebovaných zdrojů je exponenciálně závislé na velikosti vstupu. Výpočet Fibonacciho čísel měl exponenciální časovou složitost. Označujeme $O(a^n)$.
5. Logaritmická funkce, logaritmická složitost. Množství spotřebovaných zdrojů je závislé na logaritmu velikosti vstupních dat. Označujeme $O(\log n)$.

Grafy jednotlivých funkcí ukazuje obrázek Obr. 5. Nejstrměji roste graf exponenciální funkce, dále graf kvadratické funkce. Graf logaritmické funkce téměř splývá s osou.

Průvodce studiem

Konstantní časová složitost, která je zajisté ideální, je prakticky nedosažitelná. Nejraději tedy máme algoritmy s logaritmickou složitostí, pak následují algoritmy se složitostí lineární nebo kvadratickou (souhrnně hovoříme o polynomiální složitosti). Nejhorší a prakticky nepoužitelné jsou algoritmy s exponenciální složitostí.

[1] Existují i jiné složitosti výpočtů. Test na prvočíselnost má například složitost $O(\sqrt{n})$, některé složitější třídící algoritmy dosahují složitosti $O(n \log n)$.



Ideální složitost je logaritmická, nejhorší je exponenciální.

Obr. 5 Grafy funkce logaritmické, lineární, kvadratické a exponenciální

Shrnutí

Vztah mezi spotřebovanými zdroji výpočtu a velikostí vstupu určujeme pomocí složitosti. Složitost je odvozena od matematické funkce, která nejlépe popisuje tento vztah. Pro zápis složitosti používáme O notaci.

Pojmy k zapamatování

- Složitost výpočtu, prostorová a časová složitost,
- Konstantní, logaritmická, lineární, kvadratická, polynomiální a exponenciální složitost.
- O notace.

Kontrolní otázky

1. *Co vyjadřuje složitost výpočtu?*
2. *Jaké základní typy složitostí rozlišujeme vzhledem k typu spotřebovaných zdrojů?*
3. *Jaké základní typy složitostí rozlišujeme vzhledem k použitým matematickým funkcím?*
4. *Co přesně vyjadřuje O notace?*

Cvičení

1. Určete časové a prostorové složitosti následujících procedur: výpočet faktoriálu rekurzivně a iterativně, výpočet Fibonacciho čísel rekurzivně a iterativně, výpočet součtu čísel na intervalu rekurzivně a iterativně.

Úkoly k textu

1. Ověřte platnost tvrzení z kapitoly 2.1.1, že spotřebovaný čas na výpočet n-tého Fibonacciho čísla rekurzivní procedurou je roven výrazu $k(1.618^n)$. Dobu trvání výpočtu můžete změřit pomocí procedury `time` takto:

   ```
   > (time (fib 30))
   cpu time: 2103 real time: 2103 gc time: 0
   832040
   ```

   Zjistěte dobu trvání výpočtu pro různá Fibonacciho čísla a určete pro váš počítač konstantu k. Pak odhadněte, jak dlouho by výpočet probíhal pro 50. Fibonacciho číslo. Nečekejte dokonalou shodu mezi výpočty a experimentem, pro demonstraci principu nám stačí hrubá shoda.

2. Telefonní seznam bytových stanic velkého města obsahuje celkem 500 dvojstran. Záznamy v seznamu jsou tříděny abecedně podle jména vlastníka stanice. Na kolika dvojstranách musíme maximálně seznam otevřít než se dostaneme na dvojstranu obsahující hledané jméno?

Řešení

1. výpočet faktoriálu rekurzivně – prostorová $O(n)$, časová $O(n)$,
   výpočet faktoriálu iterativně – prostorová $O(1)$, časová $O(n)$,
   výpočet Fibonacciho čísel rekurzivně – prostorová $O(n)$, časová $O(1.618^n)$,
   výpočet Fibonacciho čísel iterativně – prostorová $O(1)$, časová $O(n)$,
   výpočet součtu čísel na intervalu rekurzivně – prostorová $O(n)$, časová $O(n)$,
   výpočet součtu čísel na intervalu iterativně – prostorová $O(1)$, časová $O(n)$.

2.2 Projekt: hledání největšího společného dělitele

Studijní cíle: Při řešení projektu se studující seznámí s klasickým algoritmem pro výpočet největšího společného dělitele a zjistí jeho propojenost s algoritmem na výpočet Fibonacciho čísel.

Klíčová slova: Největší společný dělitel, Euklidův algoritmus, složitost výpočtu.

Potřebný čas: 30 minut.

Největší společný dělitel dvou čísel x a y je největší číslo z, které beze zbytku dělí číslo x i číslo y. Například největší společný dělitel čísel 16 a 28 je číslo 4.

2.2.1 Euklidův algoritmus

Euklidův algoritmus je nejstarší známý algoritmus.

Největšího společného dělitele bychom zřejmě mohli najít tak, že bychom postupně kontrolovali všechna čísla menší než x i y a hledali první takové číslo, které beze zbytku obě čísla dělí. Mohli bychom k tomu použít proceduru `delitel?` z kapitoly 1.3. Existuje zde však podstatně efektivnější algoritmus který byl publikován zhruba 300 let před naším letopočtem řeckým učencem Euklidem. Vychází z pozorování, že jestliže číslo r je zbytek po dělení čísla x číslem y, pak největší společný dělitel čísel x a y a čísel y a r je stejný. Výpočet největšího společného dělitele čísel x a y tak můžeme převést na výpočet největšího společného dělitele

čísel y a r. Tím ovšem problém redukujeme na jednodušší, neboť číslo r je jistě ostře menší než číslo y. Pokud budeme algoritmus opakovat, musíme dříve či později dojít k situaci, kdy r bude rovno nule. Největší společný dělitel x a nuly je pak číslo x.

Průvodce studiem

Tento algoritmus je považován za nejstarší publikovaný netriviální algoritmus vůbec. Ostatní starověké algoritmy byly vždy prezentovány na seznamech konkrétních příkladů. Euklides tento algoritmus publikoval ve své sedmé knize Základů.

Obr. 6 Euklides

Demonstrujme algoritmus na následujícím příkladě výpočtu největšího společného dělitele (NSD) čísel 40 a 6:

NSD(40, 6) = NSD(6, 4) = NDS(4, 2) = NSD(2, 0) = 2

Úkoly k textu

1. Implementujte proceduru `nsd` na hledání největšího společného dělitele dvou čísel pomocí postupného testování možných dělitelů za využití procedury `delitel?`. Například:

   ```
   > (nsd 40, 6)
   2
   ```

   Určete složitost tohoto algoritmu.

2. Implementujte novou verzi této procedury pomocí Euklidova algoritmu. Ověřte, že obě procedury dávají shodné výsledky.

2.2.2 Lamého teorém

Euklidův algoritmus je zajímavý také tím, že nedává programátorovi příliš tipů na odhad složitosti výpočetního procesu. Počet kroků algoritmu bude pravděpodobně závislý na velikosti vstupů, druh závislosti však není zřejmý. Můžeme však pozorovat, že algoritmus je velmi rychlý i pro velká vstupní data

Euklidův algoritmus má logaritmickou složitost.

Francouzský matematik Gabriel Lamé dokázal v roce 1845 tvrzení, že pokud Euklidův algoritmus výpočtu největšího společného dělitele dvou čísel vyžaduje k kroků, pak menší z těchto dvou čísel musí být větší nebo rovno k-tému Fibonaccimu číslu. Velikost k-tého Fibonacciho čísla však exponenciálně závisí na hodnotě k[1]. Počet kroků Euklidova algoritmu (číslo k) je proto shora ohraničen logaritmem menšího ze vstupních čísel a složitost tohoto algoritmu je tak $O(\log(n))$.

Shrnutí

Největšího společného dělitele dvou čísel je možno spočítat slavným Euklidovým algoritmem. Tento algoritmus má logaritmickou složitost.

2.3 Projekt: umocňování

Studijní cíle: Po vyřešení projektu bude studující seznámen se dvěma různými algoritmy výpočtu mocniny čísla s lineární a logaritmickou složitostí.

Klíčová slova: Umocňování, složitost výpočtu, modulární aritmetika.

Potřebný čas: 1 hodina.

2.3.1 Umocňování v lineárním čase

Umocňování lze realizovat v lineárním nebo logaritmickém čase.

Základním tématem projektu je výpočet n-té mocniny čísla. Tento úkol může být v zásadě velmi jednoduchý a nedá nám mnoho práce vymyslet a implementovat rekurzivní algoritmus výpočtu s lineární složitostí. Stačí si uvědomit, že jakékoli číslo umocněné na nultou je podle definice rovno jedné a číslo b umocněné na n-tou lze vyjádřit jako b krát b umocněné na n-1. Tento volně formulovaný algoritmus můžeme zapsat matematickou notací takto:

$$b^n = \begin{cases} 1, n = 0 \\ b(b^{n-1}), n > 0 \end{cases}$$

Svou podstatou je tento kód podobný proceduře pro výpočet faktoriálu. Tento algoritmus lze téměř mechanicky převést na funkční program v jazyce Scheme.

Složitost tohoto algoritmu je $O(n)$, protože pro umocnění na n-tou algoritmus vyžaduje n kroků.

Úkoly k textu

1. Napište proceduru `expt`, která akceptuje číslo `b` (báze) a číslo `n` (exponent) a vrátí výsledek umocnění b na n.

[1] Není těžké matematickou indukcí dokázat, že k-té Fibonacciho číslo je rovno nejbližšímu celému číslu k výrazu $\frac{\varphi^k}{\sqrt{5}}$, kde $\varphi = \frac{1+\sqrt{5}}{2}$.

2.3.2 Umocňování v logaritmickém čase

Předchozí algoritmus pracoval v lineárním čase, vyžadoval proto n kroků (n operací násobení) pro umocnění na n-tou. Například pro umocnění čísla na osmou by tak bylo zapotřebí 8 operací násobení.

Existuje ale i zkratka, která dovolí vypočítat stejnou mocninu za pomoci pouze třech operací násobení. Stačí si uvědomit, že:

$b^8 = b^4 \cdot b^4$

$b^4 = b^2 \cdot b^2$

$b^2 = b \cdot b$

Pozorný čtenář jistě namítne, že tuto metodu lze použít pouze u sudých exponentů a plně se uplatní jen u mocnin dvojky. To je jistě pravda. Pokud však máme lichý exponent, můžeme snadno použít jeden krok původního algoritmu a v dalším kroku již máme exponent sudý! V průměru se tak „téměř v každém kroku" velikost exponentu zmenšuje na polovinu a algoritmus tak má logaritmickou složitost $O(\log(n))$.

Algoritmus lze ve formální matematické notaci zapsat následujícím způsobem:

$$b^n = \begin{cases} 1, n = 0 \\ (b^{n/2})^2, n - sudé \\ b(b^{n-1}), n - liché \end{cases}$$

Úkoly k textu

2. Napište proceduru `expres-expt`, která aplikuje vylepšený algoritmus umocňování čísla a pracuje v logaritmickém čase. Ověřte, že tento algoritmus dává stejné výsledky jako předchozí procedura `expt`.

Obě procedury se nyní můžeme pokusit porovnat na reálných vstupech. Následující tabulka porovnává výpočetní časy (zjištěné pomocí formy `time`) obou procedur při větších exponentech. Měli bychom pozorovat, že procedura `expres-expt` pracuje prakticky v nulovém (neměřitelném) čase, zatímco procedura `expt` je výrazně pomalejší. Tabulku vyplňte.

Efektivita logaritmického algoritmu se projeví až u větších exponentů.

`n`	`(time (expt 2 n))`	`(time (expres-expt 2 n))`
20 000		
50 000		
100 000		
150 000		
200 000		
250 000		

Průvodce studiem

Je sice pravda, že procedura `expres-expt` *je výrazně rychlejší, při vyplňování tabulky jste nicméně narazili na jiný, podstatně závažnější problém. Používáme-li velké mocniny, jsou výsledná čísla „příliš veliká" a problém jsme měli nejen s výpočtem samotným ale hlavně se zobrazením výsledku operace na obrazovce. Možná tedy vytváření procedury* `expres-expt` *bylo jen zbytečnou prací, protože tak velké exponenty, aby se projevil rozdíl, stejně nepoužijeme.*

2.3.3 Modulární aritmetika

Umocňování v logaritmickém čase skombinujeme s modulární aritmetikou.

V některých případech potřebujeme spočítat výsledek umocnění jednoho čísla na druhé číslo modulo číslo třetí[1]. Slovo modulo (v matematické notaci označujeme jako **mod**) přitom označuje zbytek po dělení, který nám v jazyku Scheme vrací procedura `remainder`. Potřebujeme tedy spočítat výraz $b^n \bmod m$.

Zde můžeme namítnout, že to není žádný problém, stačí přece nadefinovat následující proceduru:

```
(define (expmod b n m)
  (remainder (expt b n) m))
```

Co se ale stane, když bude číslo `n` velmi velké? Tato procedura využívá standardní proceduru pro umocňování, která pracuje v lineárním čase. Čtenář si proto vzpomene na efektivní proceduru pracující v čase logaritmickém a vylepší proceduru takto:

```
(define (expmod b n m)
  (remainder (expres-expt b n) m))
```

Tato verze nám však pořád nevyhovuje. Bude-li číslo n velmi velké, bude výraz `(expres-expt b n)` astronomický a pravděpodobně se vůbec „nevejde" do paměti našeho počítače. Výsledek celého výpočtu přitom musí být vždy menší než `m` náš počítač si s ním hravě poradí. Vezměme například výsledek výpočtu výrazu:

```
(expmod 105072 2019799098211 101).
```

Exponentem je zde třinácticiferné číslo. Výsledek umocnění by tedy bylo číslo s několika biliony cifer, se kterým si náš počítač nedokáže poradit. Výsledek celého výpočtu je přitom menší než 100, v našem případě je výsledkem číslo 69.

Procedura `expmod` *bude využívána v kryptografii.*

Napíšeme proto novou verzi procedury `expmod`, která bude vycházet z procedury `expres-expt` z kapitoly 2.3.2, ale nebude potřebovat vyčíslovat výsledek umocnění. Využijeme toho, že můžeme proceduru `remainder` „vpašovat" dovnitř procedury `expres-expt`. Použijeme přitom tato tvrzení, jejich platnost lze snadno ověřit.

$$b^n \bmod m = (b^{n/2} \bmod m)^2 \bmod m$$

$$b^n \bmod m = (b(b^{n-1})) \bmod m$$

[1] My budeme potřebovat tuto funkcionalitu v kapitolách 2.4 a 2.5.

Úkoly k textu

3. Navrhněte proceduru `expmod` na základě procedury `expres-expt` za použití výše uvedených vztahů. Procedura musí pracovat v logaritmickém čase a nesmí vyčíslovat hodnotu mocniny. Otestujte tuto proceduru na výše uvedeném příkladě.

Shrnutí

Algoritmus na umocňování čísla lze navrhnout tak, aby pracoval v lineárním nebo logaritmickém čase. Výhody logaritmického algoritmu se projeví teprve v kombinaci s modulární aritmetikou.

2.4 Projekt: testování prvočíselnosti

Studijní cíle: Při řešení tohoto projektu získá studující praxi při řešení problémů. Získá zkušenost se zvyšováním efektivity výpočtu a bude schopen porovnat složitosti různých verzí algoritmu. Pochopí principy probabilistického algoritmu

Klíčová slova: Prvočíslo, dělitel, probabilistický algoritmus.

Potřebný čas: 4 hodiny.

Prvočísla fascinovala matematiky po dlouhá století. Samotný fakt, že jakékoli složené číslo lze jednoznačně vyjádřit jako součin prvočísel (tzv. prvočíselná faktorizace čísla) prozrazuje, že prvočísla tvoří základní stavební kameny číselného systému. Prvočísla mají také řadu zajímavých algebraických vlastností a často se používají v kryptografii[1]. V tomto projektu se budeme věnovat problému testování prvočíselnosti.

Prvočísla představují stavební kameny systému čísel.

2.4.1 Klasický algoritmus

Prvočíslo typicky definujeme jako celé číslo větší než jedna, které je beze zbytku dělitelné jen jedničkou a samo sebou. Pro naše potřeby počítačového testování prvočíselnosti si tuto definici přeformulujeme. Řekneme, že prvočíslo je takové číslo, jehož nejmenší dělitel (větší než jedna) je roven číslu samotnému.

Průvodce studiem

Číslo 12 je složené číslo, protože je dělitelné 2, 3, 4 a 6. Máme-li dvanáct vajec, existuje mnoho tvarů krabic, do kterých můžeme vejce rovnoměrně uspořádat. Může to být například do dvou řad po šesti vejcích, do tří řad do čtyřech atd.

Číslo 13 je prvočíslo. Prvočíslo není dělitelné žádným jiným číslem kromě jedničky a sebe sama. Máme-li 13 vajec, existuje jediný tvar krabice, který můžeme použít. Viz obrázek Obr. 7.

[1] Použití prvočísel v kryptografii ukážeme v projektu z kapitoly 2.5.

Obr. 7 Prvočísla

Při testování prvočíselnosti budeme tedy hledat nejmenšího dělitele a testovat, jestli se rovná číslu samotnému. Můžeme proto navrhnout predikát `prvocislo?` a proceduru `nejmensi-delitel`:

Klasický algoritmus vychází z hledání dělitelů.

```
(define (prvocislo? n)
  (= (nejmensi-delitel n) n))

(define (nejmensi-delitel n)
  (hledej-delitele 2 n))
```

Procedura `hledej-delitele` bude postupně testovat jednotlivá čísla počínaje číslem 2. Narazí-li na číslo, které beze zbytku dělí číslo `n`, našla dělitele a výpočet může skončit. Podobný problém jsme řešili v projektu 1.3. Můžeme proto vzít proceduru `soucet-delitelu` a upravit ji pro naše potřeby.

```
(define (soucet-delitelu test n)
  (cond ((>= test n) 0)
        ((deli? test n) (+ test (soucet-delitelu (+ test 1) n)))
        (else (soucet-delitelu (+ test 1) n))))
```

Úkoly k textu

1. Přejmenujte proceduru `soucet-delitelu` na `hledej-delitele` a upravte kód tak, aby namísto součtu všech dělitelů hledal prvního dělitele. Bude potřeba změnit zvýrazněné části kódu.

Nový predikát `prvocislo?` si můžeme vyzkoušet na testování některých náhodně vybraných prvočísel z následující tabulky.

Počet cifer	Tři náhodně vybraná prvočísla
2	53, 71, 89
4	1759, 2465, 9739
6	174469, 346627, 800483
8	48277259, 76739261, 88966399
10	1863523769, 2430533023, 7615476721
12	132834206623, 198272101501, 994159640713
14	19280517511379, 79167405077461, 90311246153189
16	2416776663724961, 8216802244816723, 9528222596705843
18	169723385273658157, 306152230674960689, 930920117782367981
20	10412953763420877967, 49824489944132088271, 89728345364349770593

Úkoly k textu

2. Otestujte prvočíselnost vybraných prvočísel z tabulky. Zjistěte, při jaké velikosti čísla algoritmus „přestává fungovat“. Odhadněte (na základě měření času), jak dlouho by algoritmus testoval dvaceticiferné prvočíslo.

U velkých prvočísel klasický algoritmus selhává.

Při vašich testech pravděpodobně zjistíte, že predikát funguje skvěle na prvočísla se dvěma nebo čtyřmi ciframi. Rychlé počítače pravděpodobně zvládnou i o něco větší prvočísla. Pro prvočísla s deseti a více ciframi však narazíme na problém. Výpočet trvá příliš dlouho. Náš algoritmus má složitost $O(n)$, takže například pro otestování prvočísla s osmi ciframi musí provést několik desítek milionů testů.

Klíčem ke zvětšení efektivity je u tohoto algoritmu úprava mezní podmínky rekurze. Náš program hledá potencionální dělitele od 2 až do čísla n. Je však jasné, že žádné číslo nemůže být dělitelné číslem větším než je polovina n. V mezní podmínce rekurze bychom tedy mohli nahradit symbol `n` výrazem `(/ n 2)`. Algoritmus se tím při testování prvočíselnosti zrychlí dvojnásobně. Na složitost to sice nemá vliv, ale i tak je to výrazné zrychlení.

Klasický algoritmus lze mírně vylepšit..

Kvalitativně výrazného vylepšení dosáhneme tehdy, když si uvědomíme, že má-li mít číslo n dělitele, například p, pak n/p = q. q je celé číslo, které také dělí n. Platí totiž, že n/q = p. Alespoň jedno z čísel p a q však musí být menší než √n. V našem algoritmu proto stačí testovat čísla do druhé odmocniny z n a v kódu můžeme nahradit výraz `(/ n 2)` výrazem `(sqrt n)`. Takto upravený algoritmus má složitost $O(\sqrt{n})$, takže pro detekci osmiciferného prvočísla potřebuje řádově tisíce kroků.

Úkoly k textu

3. Upravte kód pro testování do druhé odmocniny a zopakujte testy z předchozího cvičení.

2.4.2 Probabilistický algoritmus

Efektivní řešení nabízí malá Fermatova věta

I když je algoritmus po provedených změnách výrazně rychlejší, na testování velkých prvočísel stále nestačí. V praxi však mnohdy potřebujeme testovat i prvočísla výrazně větší než dvacet cifer.

Průvodce studiem

Hledání velkých prvočísel patří mezi oblíbené „sporty“. Typicky hledáme tak zvaná Mersenova prvočísla, která mají tvar $2^p - 1$, kde p je nějaké známé prvočíslo. Internetový projekt GIMPS (Great Internet Mersene Prime Search) poskytuje uživatelům zdarma software, který běží na počítači v době, kdy jej nevyužíváme a zasílá výsledky výpočtů po internetu do centrální databáze. V době psaní tohoto textu je známo prvočíslo s osmi miliony cifer (v desítkové soustavě), brzy se však zřejmě dočkáme prvočísel s výrazně větším počtem cifer.

Je nepředstavitelné používat náš algoritmus pro testování takto velkých čísel.

Snadno použitelné řešení zde nabízí Malá Fermatova věta, kterou lze zjednodušeně formulovat takto: Je-li n prvočíslo, pak pro každé kladné číslo a menší než n platí, že $a^n \bmod n = a$.

Úkoly k textu

4. Otestujte platnost Malé Fermatovy věty. Náhodně si zvolte prvočíslo n, náhodně zvolte číslo a menší než n a manuálně otestujte platnost tvrzení.

5. Napište proceduru `fermat-test?`, která akceptuje jediný parametr `n`, vygeneruje kladné náhodné číslo `a` menší než `n` a otestuje platnost Fermatova tvrzení. Věnujte velkou pozornost generování náhodného čísla! Zabudovaná procedura `(random n)` nám vrací číslo v uzavřeném intervalu mezi `0` a `n-1`. My potřebujeme vygenerovat číslo mezi `1` a `n-1`. Pro „zapamatování" vygenerovaného čísla `a` použijte interní definici.

Průvodce studiem

Francouzský právník první poloviny 17. století Pierre de Fermat se výrazně zasloužil o rozvoj teorie čísel. Mnohé matematiky iritoval svým lehkovážným přístupem k problematice, když například zřídka publikoval důkazy svých tvrzení. Platnost Malé Fermatovy věty tak dokázal až Euler po zhruba sto letech od jejího zveřejnění. Fermatův nejslavnější poznatek, tak zvaná Fermatova poslední věta, která ve zkratce říká, že rovnice $x^n + y^n = z^n$ *nemá pro n>2 celé nenulové řešení, byla dokázána v roce 1995 profesorem Andrew Wilesem z Princeton University.*

Obr. 8 Pierre de Fermat

Chceme-li podrobit vaši implementaci Fermatova testu skutečně důkladnému testování můžeme napsat následující proceduru:

```
(define (test n)
  (display (fermat-test? n))
  (test n))
```

Dodáme-li proceduře `test` jako parametr prvočíslo, musí nám Fermatův test vrátit vždy hodnotu `#t`. Na obrazovce bychom proto měli vidět následující výpis:

```
> (test 9739)
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
```

```
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t#t
```

Úkoly k textu

6. Otestujte svoji proceduru `fermat-test?` na některých známých prvočíslech. Ujistěte se, že opravdu funguje. V tuto chvíli nám Fermatův test pravděpodobně nebude fungovat pro příliš velká čísla. Tento problém vyřešíme v kapitole 2.4.3.

Je-li číslo prvočíslem, platí Fermatův test. Obrácená implikace neplatí.

Co se však stane, když Fermatovu testu dodáme složené číslo? V první chvíli bychom řekli, že zřejmě dostaneme hodnotu `#f`, to však není tak jisté. Fermatova věta je formulována jako implikace; neplatí-li předpoklad, že n je prvočíslo, neplatí ani tvrzení. Znamená to snad, že Fematovu větu nemůžeme pro potvrzení prvočíselnosti použít? Proveďme následující experiment:

```
> (test 9738)
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#t#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#t#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f#f
```

Pokud Fermatovu testu dodáme jako parametr číslo, které není prvočíslem, dostaneme „většinou" hodnotu `#f`. „Občas" však test „selže" a vrátí nám „omylem" hodnotu `#t`.

Průvodce studiem

Podle uvozovek v textu čtenář poznal, že jsme definitivně opustili oblast exaktní a neomylné matematiky a dostali jsme se na půdu pragmatické informatiky. Z matematického hlediska je Malá Fermatova věta pro testování prvočíselnosti zcela bezcenná a nelze ji použít. z pragmatického informatického hlediska je však šance že věta „selže" nepříliš velká. Zatímco matematika se zabývá ideálními světy, informatiky pracuje v realitě plené kompromisů.

Na experimentu vidíme, že procento selhání Fermatova testu není velké. Šanci na chybné označení složeného čísla jako prvočísla můžeme ještě dále zmenšit tím, že test několikrát po sobě zopakujeme. Pokud Fermatův test například stokrát za sebou označí číslo jako prvočíslo (vrátí hodnotu `#t`), budeme s jistou velmi významnou pravděpodobností věřit, že číslo je opravdu prvočíslem. [1] Pokud však v jediném ze sta pokusů vrátí test `#f`, máme matematickou jistotu že číslo prvočíslem není.

Pravděpodobnost testu lze zvýšit jeho opakováním.

Úkoly k textu

7. Napište predikát `expres-prvocislo?`, který akceptuje číslo `n` a stokrát po sobě na tomto čísle provede Fermatův test. Predikát vrátí `#t` pokud všechny testy dopadnou kladně, jinak vrátí hodnotu `#f`.

2.4.3 Efektivita probabilistického algoritmu

Název procedury `expres-prvocislo?` evokuje dojem, že tento test je výrazně rychlejší než klasický algoritmus a zřejmě by měl fungovat i pro velmi velká čísla. Rychlost této procedury je jistě přímo závislá na rychlosti procedury `fermat-test?`, a tato procedura zase závisí na rychlosti umocňování a výpočtu zbytku po dělení. Zde však známe zkratku nazvanou `expmod` z kapitoly 2.3.3. Procedura `expres-prvocislo?` tak bude mít logaritmickou složitost a měla by fungovat velmi rychle i pro velmi velká prvočísla.

Implementace Fermatova testu je zatížena technickými problémy.

Úkoly k textu

8. Pokud jste tak ještě neudělali, použijte v proceduře `fermat-test` proceduru `expmod` pro výpočet výrazu $a^n \bmod n$.

Počet cifer	Dvě náhodně vybraná prvočísla
30	719518703189451419556372547849, 930179493030939691966902692263
40	435421063791646124534314597248129223837 1 289744532223671868584152079774534321 2387
50	128402436355187448674729535610980697496117752784 47 919431842218254212358852456653258567953385272296 23

Pokud se pokusíte použít proceduru `expres-prvocislo?` na prvočísla delší než 8 cifer, narazíme na další problém. Zabudovaná procedura `random`, která se využívá ve Fermatově testu, umí pracovat pouze s čísly, která jsou menší než 2147483647[2]. Za použití procedur `pocet-cifer` a `generuj-cifry` z textu [Skoupil04A] lze napsat vylepšenou verzi procedury `random`:

[1] Pravděpodobnost chybného označení složeného čísla Fermatovým testem klesá exponenciálně s počtem provedených pokusů.

[2] To alespoň platí pro DrScheme verze 207. V dalších verzích nebo jiných implementacích jazyka tomu může být jinak a náš problém může vymizet.

```
(define (xrandom n)
  (if (<= n 2147483647) (random x)
      (generuj-cifry (random (pocet-cifer n)))))
```

Úkoly k textu

9. Použijte proceduru `xrandom` ve vaší proceduře `fermat-test`. Otestujte pak predikát `expres-prvocislo` na 30, 40 a 50 ciferné prvočísla z předchozí tabulky.

10. Napište proceduru `generuj-prvocislo`, která akceptuje číslo `n` a vygeneruje náhodně zvolené prvočíslo o `n` cifrách. Použijte přitom proceduru `generuj-cifry`.

11. Vygenerujte náhodná prvočísla dlouhá 70, 100, 150 a 200 cifer.

Shrnutí

Prvočísla hrají důležitou roli v teoretické matematice i kryptografii. Klasický test prvočíselnosti využívá faktorizaci čísla a má složitost $O(\sqrt{n})$. Probabilistický algoritmus založený na malé Fermatově větě pracuje v logaritmickém čase.

2.5 Projekt: RSA kryptografie

Studijní cíle: Při řešení projektu studující prakticky použije poznatky a výsledky z projektů 2.3 a 2.4. Pochopí princip RSA kryptografie a význam složitosti výpočtu pro bezpečnost kódu.

Klíčová slova: RSA, kryptografie, složitost výpočtu.

Potřebný čas: 2 hodiny (respektive 1 týden).

2.5.1 Principy asymetrické kryptografie

Rozlišujeme symetrickou a asymetrickou kryptografii.

Kryptografické techniky lze v principu rozdělit na *symetrické* a *asymetrické*. *Symetrická kryptografie* dovoluje ze znalosti způsobu zakódování zprávy zprávu lehce dekódovat. Způsob zakódování je přitom dán dvěma základními faktory: použitým algoritmem a klíčem. Klíč je použit jako vstupní parametr kódovacího a dekódovacího algoritmu. V dnešní době jsou používané algoritmy veřejně známé[1] a celý princip symetrické kryptografie je založen za utajení klíče.

Průvodce studiem

Pravděpodobně nejstarší symetrická šifra byla používána pro kódování mezi vojenskými jednotkami římské armády. Hovoříme o ní jako o Césarově šifře. Principem bylo posunutí písmen v abecedě o jistý daný počet písmen (klíč). Pokud by klíčem byla hodnota 3, namísto A použijeme D, namísto B použijeme E a namísto Z použijeme C. Známe-li algoritmus a klíč, můžeme kódovanou zprávu snadno dekódovat.

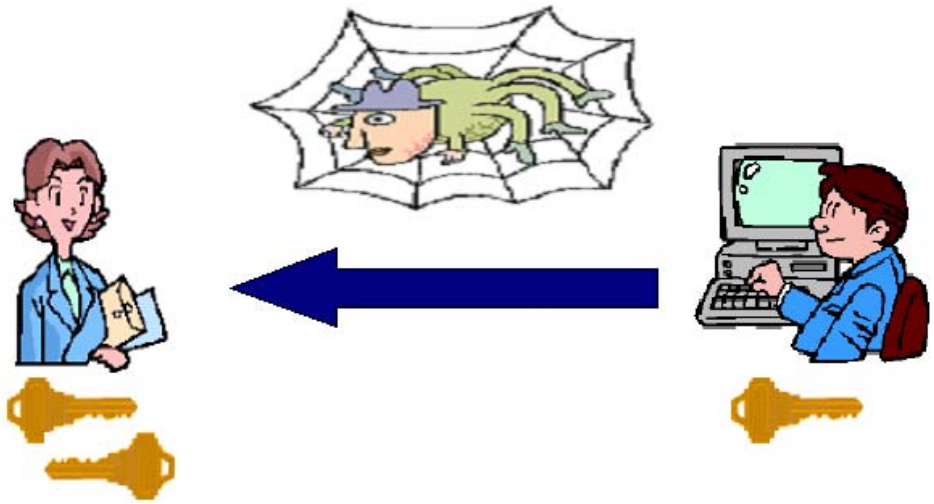
[1] Mezi nejpoužívanější symetrické šifry patří 3DES, Blowfish nebo IDEA.

Asi nejslavnější šifrou byla německá Enigma, používaná za druhé světové války. Algoritmem by složitý mechanický šifrovací stroj, klíčem pak počáteční nastavení jednotlivých rotorů stroje.

Symetrická kryptografie je často využívána v situacích, kdy spolu komunikují dva známé subjekty. Pouze spřátelené subjekty se totiž mohou sejít a diskrétně a bezpečně se dohodnout na používaném klíči. Pokud spolu potřebují komunikovat dva neznámé a vzdálené subjekty, nelze symetrickou kryptografii přímo použít, protože zde není možnost jak si bezpečně vyměnit klíč symetrické šifry.

V tuto chvíli na scénu vstupuje asymetrická kryptografie. Je založená na existenci dvou, matematicky závislých klíčů. To co jeden klíč zakóduje, může být dekódováno pouze druhým klíčem a naopak.

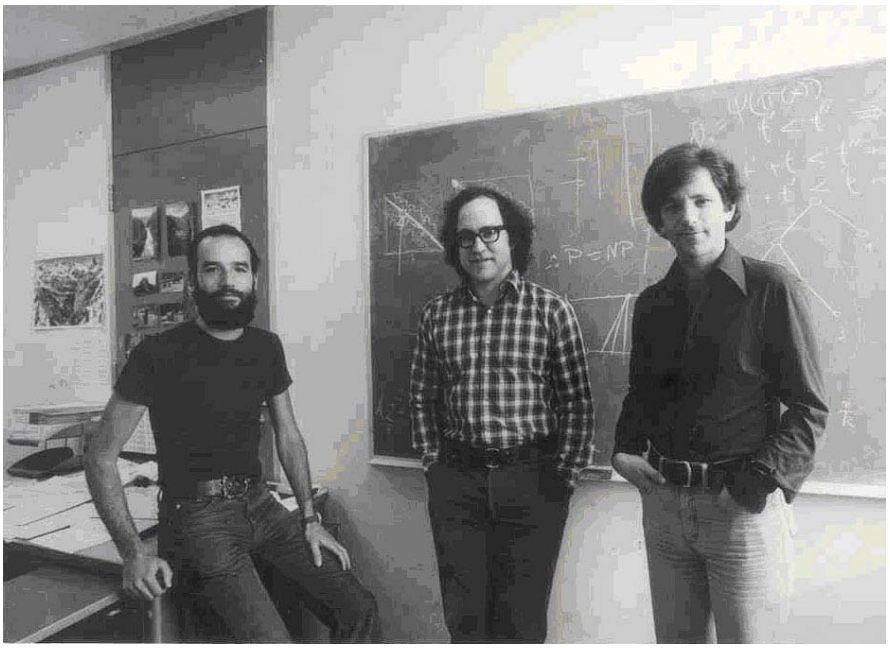
Asymetrická kryptografie využívá veřejný a soukromý klíč.



Obr. 9 Asymetrická kryptografie

Obrázek Obr. 9 ukazuje princip asymetrické kryptografie. Bob (na obrázku vpravo) chce poslat tajnou zprávu Alici (na obrázku vlevo). Alice vygeneruje pár asymetrických klíčů. Jeden z klíčů (veřejný klíč) dá k dispozici všem, kdo o něj mají zájem, tedy i Bobovi (může jej například umístit na svoji webovou stránku). Druhý klíč (soukromý klíč) si ponechá a nedá jej k dispozici nikomu. Každý (včetně Boba) tak může Alici poslat zprávu zakódovanou pomocí jejího veřejného klíče. Pouze Alice však může tyto zprávy dekódovat.

Nejpoužívanější asymetrická šifra, vyvinutá v roce 1977 Rivestem, Shamirem a Adlemanem, se po svých tvůrcích nazývá RSA.



Obr. 10 Ron Rivest, Adi Shamir a Leo Adleman

2.5.2 Kódování pomocí RSA kryptografie

RSA kryptografie je založena na teorii prvočísel.

Kódování pomocí RSA lze celkem snadno implementovat v jazyku Scheme. Většinu procedur máme již navíc připravenu z předchozích projektů. Princip RSA kódování lze shrnout do následujících bodů:

- Náhodně zvolíme velké prvočíslo p. Prvočíslo by mělo mít alespoň 40 cifer[1].
- Náhodně zvolíme velké prvočíslo q. Prvočíslo by mělo mít alespoň 40 cifer.
- Položíme $n = pq$.
- Položíme $m = (p-1)(q-1)$.
- Zvolíme k1 jako velké prvočíslo, obdobně jako jsme volili p a q.[2]
- Vypočteme k2 jako $k_2 = (1+mr)/k_1$, kde $r = ((m^{k_1-2} \bmod k_1)(k_1-1)) \bmod k_1$. Pro tento výpočet použijeme proceduru `expmod` z kapitoly 2.3.3.
- Kódování zprávy s (zpráva je reprezentována číslem) provádíme jako $s' = s^{k_1} \bmod n$.
- Dekódování zprávy provádíme obdobně jako kódování, použijeme jen druhý klíč, $s = s'^{k_2} \bmod n$.

Veřejný klíč je představován čísly k1 a n. Pomocí těchto dvou čísel může kdokoli zakódovat zprávu. Soukromý klíč je představován čísly k2 a n. Pomocí těchto dvou čísel můžeme zakódovanou zprávu dekódovat.

---

[1] 128bitová RSA kryptografie používá prvočísla o velikosti 2^{128} což odpovídá 39 cifrám v desítkové soustavě.

[2] Obecně `k1` nemusí být prvočíslo, stačilo by aby `k1` bylo menší než `m` a `k1` a `m` byla nesoudělná čísla. Pro prvočíslo je tato podmínka splněna automaticky.

Pro demonstraci na počítači nejprve zkopírujte z projektů 2.3 a 2.4 tyto procedury: `expmod`, `fermat-test`, `generuj-cifry`, `pocet-cifer`, `xrandom`, `expres-prvocislo?` a `generuj-prvocislo`. Dále napište do prostředí jazyka Scheme tento kód:

```
(define p (generuj-prvocislo 50))
(define q (generuj-prvocislo 50))
(define n (vypocti-n p q))
(define k1 (generuj-prvocislo 50))
(define k2 (vypocti-k2 k1 p q))
(define s 1234)
(define kod-s (kodujRSA s k1 n))
(define dekod-s (dekodujRSA kod-s k2 n))
```

Úkoly k textu

1. Na základě RSA vzorců implementujte procedury, které jsou ve výpise znázorněny tučně. Otestujte kódování a dekódování zpráv.

Průvodce studiem

Pokud vám kódování nebo dekódování nefunguje, zde je několik tipů.

Jestliže výpočet neterminuje, zkontrolujte, že všude používáte výhradně proceduru `expmod`. Vzhledem k velikosti zadávaných čísel není možno použít procedury `expt` nebo `expres-expt`.

Hodnota `k2` musí vždy vyjít jako celé číslo (pozor na zlomky!). Pokud tomu tak není, buď je chyba ve vzorci nebo je problém s generováním prvočísel.

Otestujte nezávisle proceduru `vypocti-k2`; například volání `(vypocti-k2 19 5 13)` musí vrátit hodnotu `43`.

Kódovaná zpráva musí být vždy menší než číslo `n`. Pokud je zpráva delší než `n`, je buď potřeba zvětšit hodnotu `n` (volit větší prvočísla) nebo rozdělit zprávu na více částí.

Následující kód provádí transformaci textu (zadávaného v uvozovkách) na číslo a zpět na základě ASCII tabulky.

```
(define (text->cislo s)
  (if (= (string-length s) 0) 0
      (+ (char->integer (string-ref s 0))
         (* 1000 (text->cislo (substring s 1 (string-length s)))))))

(define (cislo->text n)

  (define (safe-integer->char n)
    (if (and (> n 0) (< n 256))
        (integer->char (floor n))
        (error "Chybna zprava")))

  (if (< n 1000)
      (string (safe-integer->char n))
      (string-append (string (safe-integer->char (remainder n 1000)))
                     (cislo->text (quotient n 1000)))))
```

Úkoly k textu

2. Zapište kód do prostředí jazyka a upravte procedury `kodujRSA` a `dekodujRSA` tak, aby akceptovaly namísto čísla řetězec znaků (v uvozovkách).

2.5.3 Rozlomení RSA kódu

Rozlomení RSA šifry spočívá ve faktorizaci velkého čísla.

Rozlomení RSA kódu je algoritmicky velmi jednoduché. Stačí si uvědomit, v čem spočívá zadání a co máme k dispozici. K jádru problému se můžeme dostat touto sekvencí kroků:

- Máme k dispozici zakódovanou zprávu a veřejný klíč (čísla `k1` a `n`), kterým byla zpráva zakódována. Chceme zprávu dekódovat a přečíst.
- Pro dekódování zprávy můžeme použít proceduru `dekodujRSA`, potřebujeme však znát soukromý klíč `k2`.
- Pro výpočet `k2` můžeme použít proceduru `vypocti-k2`, k tomu však potřebujeme znát `p` a `q`.
- My známe pouze `n`. Víme však, že `n` vzniklo jako součin `p` a `q`. Proto `n` musí být dělitelné čísly `p` i `q`. Stačí najít jednoho z dělitelů čísla `n` a druhého dělitele lze snadno dopočítat. Pro nalezení dělitele můžeme použít proceduru `hledej-delitele` z kapitoly 2.4.1.[1]

Úkoly k textu

3. Napište proceduru `rozlomRSA`, která akceptuje kódovanou zprávu (číslo), `k1` a `n` a vrací dekódovanou zprávu. Například:

```
> (rozlomRSA 96223152345 729551 380134841941)
"Ahoj"
> (rozlomRSA 27655273178232 9394019 28024125634093)
"Zdar"
> (rozlomRSA 719041357597913 37394347 1039886083969331)
"Vitej"
```

4. Změřte dobu běhu procedury `rozlomRSA` pomocí formy `time`. Doba nutná pro zlomení šifry je přitom dána počtem kroků nutných k faktorizaci čísla `n`. Procedura `hledej-delitele` má složitost $O(\sqrt{n})$. Vypočtěte, jak dlouho by vašemu počítači trvalo rozlomení šifry, kterou jste využívali pro tento projekt (`p` i `q` byla padesáticiferná prvočísla).

5. Aktuální stáří vesmíru se odhaduje na 13.000.000.000 let. Porovnejte dobu nutnou pro zlomení šifry se stářím vesmíru. Dále předpokládejte, že bychom pro lámání kódu použili všechnu výpočetní sílu na planetě Zemi (odhadněte). Jak dlouho by trval výpočet?

6. Zvláštní agent Johnson z FBI zaslal na ústředí následující zakódovanou zprávu:

   174177777124838706009514

   Pro zakódování použil veřejný klíč FBI, který je dostupný na internetu, konkrétně n = 221823717366419704863563 a k1 = 251226863267. Dokážete zprávu rozluštit?

[1] Protože `p` a `q` jsou prvočísla, není `n` dělitelné žádnými jinými čísly.

RSA kryptografie je typickým příkladem využití složitosti algoritmů pro praktické účely. Veškerá práce s šifrou jako je generování klíčů, kódování i dekódování využívá proceduru `expmod`, která pracuje v logaritmickém čase. Pokus o rozlomení šifry narazí na problém neexistence (nebo naší neznalosti) algoritmu na faktorizaci čísla v logaritmickém čase. Nejlepší algoritmus který známe pracuje se složitostí $O(\sqrt{n})$. Rozdíl mezi těmito složitostmi dělá RSA kódování bezpečným.

Průvodce studiem

RSA kódování není z několika důvodů příliš výhodné pro zabezpečení rozsáhlejší komunikace na internetu. Používá se proto většinou pouze na začátku spojení pro zajištění bezpečné výměny klíče symetrického kódování. Samotný přenos dat je pak kódován symetricky.

Shrnutí

Asymetrická kryptografie vychází z principu, že zpráva je kódována jedním klíčem a dekódována jiným. Klíče jsou matematicky závislé, ale není jednoduché ze znalosti jednoho klíče spočítat druhý. Kryptografická metoda RSA se opírá o obtížnost faktorizace čísla. Práce s šifrou probíhají v logaritmickém čase zatímco rozlomení šifry vyžaduje v zásadě čas lineární.

3 Závěr

Tato publikace navazovala na publikaci [Skoupil04A]. Využili jsme znalosti základů programovacího jazyka Scheme a věnovali jsme se výpočetnímu procesu, jeho typům a složitosti algoritmů. Problematika byla demonstrována na řadě menších i rozsáhlejších projektů.

Předpokládáme, že po absolvování tohoto textu bude čtenář pokračovat ve studiu a sáhne po dalších materiálech, které pokryjí zejména oblasti datových struktur.

4 Seznam literatury

[Abelson96] ABELSON, H., SUSSMAN, G. *Structure and Interpretation of Computer Programs*. 2. vyd. Cambridge, Massachusetts: MIT Press, 1996. ISBN 0-262-01153-0. Dostupné z WWW:

[Dybvig03] DYBVIG, R. K. *The Scheme Programming Language*. 3. vyd. Cambridge, Massachusetts: MIT Press, 2003. ISBN 0-262-541483-0.

[Friedman94] FRIEDMAN, D. P., WAND, M., HAYNES, C. T. *Essentials of Programming Languages.* 1. vyd. Cambridge, Massachusetts: MIT Press, 1994. ISBN 0-262-06145-7.

[Friedman96a] FRIEDMAN, D. P., FELLEISEN, M. *The Little Schemer*. 4. vyd. Cambridge, Massachusetts: MIT Press, 1996. ISBN 0-262-56099-2.

[Friedman96b] FRIEDMAN, D. P., FELLEISEN, M *The Seasoned Schemer.* 4. vyd. Cambridge, Massachusetts: MIT Press, 1996. ISBN 0-262-56100-X.

[R5RS] KELSEY, R, CLINGER, W, REEDS, J. (eds.), Revised5 Report on the Algorithmic Language Scheme, *Higher-Order and Symbolic Computation*, 1998, roč. 11, č. 1.

[Skoupil97] SKOUPIL, D., KOPKA, M. *Průvodce jazykem Scheme*. 1. vyd. Olomouc: Vydavatelství UP, 1997. ISBN 80-7067-693-0.

[Skoupil04A] SKOUPIL, D. *Programy a projekty v jazyku Scheme I.*

[Skoupil04C] SKOUPIL, D. *Programy a projekty v jazyku Scheme III*.

[Steele76a] STEELE, G. L., SUSSMAN, G. J. *LAMBDA: The Ultimate Imperative.*MIT AI Lab Memo AIM-353, 1976. Dostupné z FTP: ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-353.pdf

[Steele76b] STEELE, G. L. LAMBDA: *The Ultimate Declarative*. MIT AI Lab. AI Lab Memo AIM-379, 1976. Dostupné z FTP: ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-379.pdf

[Steele78] Steele, G. L. *Revised Report on Scheme –Dialect of Lisp.* MIT AI Lab. AI Lab Memo AIM-452, 1978. Dostupné z FTP: ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-452.pdf

[Steele93] STEELE, G. L., GABRIEL, R. *The Evolution of Lisp*. SIGPLAN Notices 1993, roč 28, č 3, s. 231-270. ISSN 0362-1340. Dostupné z FTP: ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/Evolution-of-Lisp.ps

5 Seznam obrázků

6 Rejstřík