České vysoké učení technické v Praze
Fakulta elektrotechnická

Bakalářská práce

Faktorizace velkých čísel pomocí knihovny GMP

Peter Kováč

Vedoucí práce: Ing. Ivan Šimeček

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

srpen 2007

Poděkování

Především bych rád poděkoval rodině za podporu, bez které by pro mě psaní této práce bylo mnohem obtížnější. Dále samozřejmě vedoucímu Ing. Ivanu Šimečkovi za připomínky a návrhy, které mi pomohly zvýšit úroveň textu po stránce odborné. V neposlední řadě děkuji přátelům, zejména Ondřeji Sýkorovi, Michalu Tuláčkovi, Michalu Augustýnovi a Martinu Mrázovi, jejichž připomínky a návrhy také přispěly ke zdárnému dokončení této práce.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu. Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 20.8.2007 .....

Obsah

Seznam tabulek

Kapitola 1

Úvod

1.1 Obecný úvod

V současné době je bezpečnost jedním z nejdiskutovanějších témat, hlavně co se týče ochrany citlivých údajů, hesel a podobně. Každý je dnes chtě nechtě nucen si pamatovat mnoho různých přístupových kódů, ať už jde o PIN kódy do mobilního telefonu nebo ke kreditní kartě, případně o heslo do emailové schránky a tak dále. Díky heslům a šifrám je zaručeno, že se k důležitým informacím (popřípadě jiným hodnotným věcem) dostane pouze osoba která na to má právo, tedy ve většině případů jejich majitel. Šifry používané v dnešní době jsou v drtivé většině případů založeny na faktu, že by jejich prolomení trvalo neúnosně dlouhou dobu, pokud by to vůbec bylo současnými prostředky proveditelné. Je nutné si uvědomit, že většina informací chráněných šiframi po určité době ztrácí svoji hodnotu. Přirovnáme-li to například k sázení na sportovní utkání, pak je naprosto zbytečné mít zašifrovanou informaci o výsledku večerního zápasu, jestliže nejsme schopni tuto informaci dešifrovat dříve než druhý den ráno. Tou dobou je již tato informace relativně bezcenná, protože o výsledku utkání se může každý dočíst v novinách. Paradoxně tedy není otázkou, zda jde šifru prolomit, ale jak dlouho by to trvalo. V praxi se dnes používají asymetrické šifry, tedy šifry s jedním veřejným a jedním tajným klíčem. Klíče se navzájem samozřejmě liší, mají ovšem mezi sebou určitou matematickou vazbu vyplývající ze způsobu jejich vytvoření. Veřejný klíč se používá k šifrování dat, tajným klíčem je samozřejmě možné opět data rozšifrovat. Síla šifry spočívá v tom, že je velice obtížné, ne-li nemožné, z veřejného klíče zjistit klíč tajný.

Právě rozklad celých čísel na součin prvočísel představuje úlohu, jejíž dokončení je (při dostatečně velkých číslech) velice obtížný problém. Proto je základem několika šifrovacích metod, například RSA. Je možné dokázat, že

každé kladné celé číslo má jednoznačný (a jediný) rozklad na součin prvočísel. Prvočísla tedy mohou být chápána jako základní stavební bloky všech ostatních čísel. Co však dělá z faktorizace tak obtížný problém? Na základní škole nás učili, že každé sudé číslo je dělitelné dvěma nebo že každé číslo končící nulou a pětkou je dělitelné pěti a tak dále. Naučili nás tedy jednoduchý způsob jak ověřit dělitelnost malými prvočísly, díky čemuž jsme schopni snadno a rychle i o velkých číslech rozhodnout, zda jsou dělitelná některým z těchto malých prvočísel. Pokud však je ve všech případech odpověď záporná, jsme nuceni prověřit i prvočísla větší. Je dost dobře možné, že ani za použití silné výpočetní techniky nebudeme stále nacházet řešení. Představíme-li si, že je naším úkolem rozložit na prvočísla opravdu velké číslo, je nad slunce jasné že najít tento rozklad je velmi náročné. Samozřejmě že uvedený postup při hledání faktorizace čísla není jediný možný. V tomto textu se podíváme na různé možnosti jak tento problém řešit, představíme jejich výhody a nevýhody, ověříme použitelnost v praxi a nakonec se pokusíme navrhnout vylepšení postupu a implementace.

1.2 Popis problému

Než se ponoříme hlouběji do problematiky rozkladu čísla na součin prvočísel, zavedeme si několik pojmů abychom se tak vyhnuli případným nejasnostem z důvodů nevysvětlené terminologie. S některými běžnějšími pojmy sme se setkali již v úvodních odstavcích, pro pořádek je definujeme i zde:

Definice 1.2.1 (Dělitel) *Řekneme, že přirozené číslo* a *dělí přirozené číslo* b *(značíme $a \mid b$) pokud existuje přirozené číslo* n *takové, že $b = n \cdot a$.*

V takovém případě je tedy číslo a dělitelem čísla b. Z hlediska svých dělitelů mají čísla různé vlastnosti, což využijeme v následujících definicích.

Definice 1.2.2 (Prvočíslo) *Prvočíslem p rozumíme přirozené číslo větší než 1 takové, které má pouze dva různé přirozené dělitele a to 1 a p.*

Jak uvidíme dále, vynechání čísla *1* z výše uvedené definice je pro nás velmi důležité. Následující definice s tou první úzce souvisí.

Definice 1.2.3 (Složené číslo) *Složeným číslem c rozumíme přirozené číslo takové, které má alespoň jednoho přirozené dělitele různého od 1 a c.*

Nyní zavedeme velice důležitý pojem pro další pokračování v práci. Jak již bylo řečeno, naším cílem je faktorizovat přirozené číslo, tedy rozložit jej na součin prvočísel. Definujme tedy tento pojem přesněji.

Definice 1.2.4 (Prvočíselný rozklad) *Prvočíselným rozkladem přirozeného čísla* x *rozumíme rovnost*

$$x = p_1^{n_1} \cdot p_2^{n_2} \ldots, p_r^{n_r}$$

kde $r \geq 1$ *je přirozené číslo,* $p_1 < p_2 < \ldots < p_n$ *jsou navzájem různá prvočísla a* $n_1, n_2 \ldots n_r$ *jsou kladná přirozená čísla.*

Prvočísla patřící do prvočíselného rozkladu čísla x nazýváme také *faktory* čísla x. Zbývá nám ještě vyslovit důležitou větu o prvočíselném rozkladu a můžeme se směle pustit do popisu námi řešeného problému – faktorizace celých čísel.

Věta 1.2.1 (Základní věta elementární teorie čísel) *Pro každé přirozené číslo* $x \geq 2$ *existuje jednoznačný prvočíselný rozklad.*

Důkaz této věty je možné najít například zde [1]. Z této věty také plyne fakt, že prvočísla jsou základními stavebními kameny přirozených čísel. Nyní se můžeme, vybaveni základní terminologií, pustit do hlubšího prozkoumání celé problematiky.

Jak už jsme nastínili v úvodu, rozložit velké číslo na součin prvočísel může představovat velký problém. Jak tedy budeme postupovat, jestliže bude naším úkolem faktorizovat velké číslo N? Logicky prvním krokem je zkusit dělitelnost některým z malých prvočísel, například dvojkou, pětkou, sedmičkou. Takto můžeme velice snadno odhalit malé faktory. Šance, že náhodně vygenerované číslo N bude nějaký malý faktor obsahovat je relativně vysoká. Víme totiž, že každé druhé číslo je dělitelné dvěma, každé třetí třemi a tak dále. Stížíme si tedy úlohu a budeme předpokládat, že naše číslo N nemá žádné malé prvočíselné faktory. Protože je také relativně výpočetně levné otestovat naše číslo N na prvočíselnost (a případně ji dokázat některým z testů prvočíselnosti), budeme předpokládat, že číslo N je složené. Nyní začíná být jasné, proč je metoda postupného dělení čísla N stále většími a většími prvočísly nevhodná, pokud je číslo N dostatečně velké. Důvodem je enormní množství čísel, která bychom museli „vyzkoušet“ než bychom nalezli dělitele N v případě, že N má dělitele řádově velikosti $\sqrt{N}$. Přesto má tato metoda, zvaná „Postupné dělení“ (anglicky *Trial Division*), své opodstatnění a v určitých situací je jí z výhodou využíváno, jak ostatně uvidíme později.

Naštěstí tento postup není zdaleka jediný, který nám může pomoci při hledání faktorů N. Abychom však mohli rozebrat další postupy, musíme definovat jeden klíčový pojem:

Definice 1.2.5 (Kongruence) *Říkáme, že čísla A, B jsou kongruentní modulo N, jestliže se zbytky po dělení těchto čísel číslem N rovnají.*

Nyní využijeme následující rovnost:

$$a^2 = b^2 \quad vZ_N \tag{1.1}$$

kde N je číslo které chceme faktorizovat a a, b jsou navzájem různá přirozená čísla. Pak platí rovnost

$$a^2 - b^2 = (a-b)(a+b) = 0 \quad vZ_N \tag{1.2}$$

Spočtení největšího společného dělitele $(a-b)$ nebo $(a+b)$ s N nám dává 50 % šanci na nalezení netriviálního faktoru N. (v ostatních případech je výsledkem výpočtu číslo 1 nebo N, viz [1]). Faktorizační algoritmus, který je založena čistě na tomto principu se nazývá *Fermatova faktorizační metoda*. Nevýhodou této metody je fakt, že nalézt čísla vyhovující výše uvedené rovnici je poměrně obtížné a to jednoduše proto, že takových čísel není mnoho. V krajních případech je tato metoda dokonce zdlouhavější než metoda *Postupného dělení* uvedená v předchozím odstavci. Nezbývá nám tedy než se pokusit najít způsob, jak efektivně najít čísla a, b. Na následujícím příkladu ukážeme, jak bychom hledali vhodná čísla při použití Fermatovy metody. Zkusíme najít faktorizaci čísla *8051*. Zběžný test malými prvočísly $\{2,3,5,7\}$ nám nedává žádně řešení. Spočítáme tedy $\sqrt{8051}$, desetinnou část čísla zanedbáme. Máme tedy:

$$\sqrt{8051} \approx 89 \tag{1.3}$$

vyzkoušíme nejbližší vyšší číslo zda nevyhoví naší rovnici

$$90^2 - 8051 = 49 = 7^2 \tag{1.4}$$

tedy

$$90^2 \equiv 7^2 \pmod{8051} \tag{1.5}$$

což vyhovuje naší rovnici a máme tedy $a = 90, b = 7$. Spočtením největšího společného dělitele $(a-b)$ a *8051* nám dává *83* a $(a+b)$ a *8051* dává *97*. Obě čísla jsou faktory *8051*. V tomto případě jsme „měli štěstí“ – nalezli jsme rychle čísla a i b a následně i faktory *8051*. V drtivé většině případů však budeme nuceni prověřit obrovské množství čísel než najdeme kongruenci čtverců (tedy $a^2 \equiv b^2 \pmod{N}$). Existuje nějaký způsob jak toto hledání urychlit? Odpověď zní ano, taková možnost tu skutečně je. Při hledání kongruence čtverců totiž prověříme mnoho čísel, která rovnici vyhovovat nebudou – proč bychom tedy nezkusili z těchto nevhodných čísel vytvořit číslo vhodné? Skutečně je to možné, jak ostatně ukazuje následující příklad. V něm zkusíme faktorizovat číslo $N = 1649$:

$$41^2 - N = 32 \tag{1.6}$$

$$42^2 - N = 115 \tag{1.7}$$

$$43^2 - N = 200 \tag{1.8}$$

Ani třetí číslo není čtvercem, přesto však zde uvedené rovnice mohou vést k řešení. Všimněme si, že 1.6 a 1.8 dávají po vynásobení čtverec, tedy $32 \cdot 200 = 6400 = 80^2$ Můžeme tedy napsat následující rovnici, z níž získáme hledaná čísla a, b:

$$(41 \cdot 43)^2 \equiv 80^2 \pmod{N} \tag{1.9}$$

Tedy $a \equiv 114 \equiv 41 \cdot 43 \pmod{N}$ a $b = 80$. Pokud se ale pokusíme výše uvedený postup popsat algoritmicky, nutně se musíme pozastavit v části, kde jsme vybírali čísla tak, abychom získali čtverec. Tento výběr musíme provádět na základě nějakého klíče, protože se vzrůstajícím počtem „nevhodných" čísel také přibývá počet kombinací, které mohou, ale také nemusí poskytovat hledané řešení. Prověřovat je všechny by bylo výpočetně extrémně náročné a také bychom řešení nemuseli vůbec najít. Mějme tedy sekvenci čísel [1 ... X] ze které chceme vybrat nějakou podsekvenci, jejíž produktem je čtverec. Jak zaručíme, že taková podsekvence bude skutečně existovat a zároveň minimalizujeme potřebnou velikost hlavní sekvence, ve které budeme hledat? K nalezení odpovědi nám pomůže následující definice:

Definice 1.2.6 (B-hladké číslo) *Číslo nazýváme B-hladké, jestliže všechny jeho prvočíselné faktory jsou* $\leq B$.

Ačkoliv se na první pohled může zdát, že nám B-hladká čísla moc v našem problému nepomohou, opak je pravdou. Nejdříve se podívejme, proč bychom do naší sekvence neměli zahrnovat čísla která nejsou B-hladká. Důvodem je malá šance že se toto číslo bude vyskytovat v naší podsekvenci tvořící čtverec. Pokud číslo c není B-hladké, má nějaký prvočíselný faktor p (alespoň jeden), kde p je větší než B. Kdybychom chtěli použít c v naší subsekvenci, pak v ní nutně musí být ještě další číslo obsahující p, označme ho *c'*. Pokud by p bylo velké prvočíslo, násobků p bude málo a budou od sebe (v sekvenci čísel) hodně vzdáleny, takže nalezení čísla *c'* by bylo obtížné nebo přinejmenším zbytečně zdlouhavé. Řekněme tedy, že do naší sekvence budeme zahrnovat pouze B-hladká čísla a prozatím odsuňme otázku volby hodnoty B stranou. Zbývá nám vyřešit poslední problém – jaké množství B-hladkých čísel budeme potřebovat, abychom zaručili, že nám sekvence těchto čísel poskytne subsekvenci jejíž produktem bude čtverec? Odpověď nalezneme v této větě:

Věta 1.2.2 *Jestliže jsou* $m_1, m_2 \ldots m_k$ *kladná B-hladká celá čísla a jestliže* $k > \pi(B)$ *(kde* $\pi(B)$ *představuje počet prvočísel v intervalu* $1 \ldots B$*)pak produktem nějaké neprázdné podsekvence* (m_i) *je čtverec.*

Celý důkaz věty spolu s dalšími informacemi je možné nalézt zde [2]. Nyní je vidět, že pro zaručení existence alespoň jedné subsekvence potřebujeme nejméně $B + 1$ B-hladkých čísel. Předchozí věta nám ale také napovídá, jak ony subsekvence hledat. Využijeme faktu, že všechny faktory čísel z naší sekvence jsou $\leq B$. Díky tomu jsme schopni jednoduše najít jejich rozklad na prvočísla a z jejich exponentů vytvořit exponenční vektor. Například pro 7-hladké číslo *12 250* bude exponenční vektor vypadat následovně:

$$(1 \quad 0 \quad 3 \quad 2)$$

Protože $12250 = 2^1 \cdot 3^0 \cdot 5^3 \cdot 7^2$. Nyní tedy hledáme exponenční vektor tvořený takovou lineární kombinací našich vektorů, aby všechny jeho prvky byly sudé (je triviální ukázat že číslo, jehož prvočíselný rozklad obsahuje pouze sudé mocniny prvočísel je čtverec – operace odmocnění se z hlediska exponenů převede na dělení dvěma, tudíž aby číslo bylo čtverec, musí mít všechny tyto exponenty dělitelné dvěma beze zbytku). Protože nás zajímá pouze to, zda je exponent sudý nebo lichý, využijeme počítání (mod 2). Hledáme tedy lineární kombinaci exponenčních vektorů (mod 2) která dává nulový vektor. Jedná se v podstatě o soustavu lineárních rovnic která je ke všemu homogenní, protože hledané řešení je nulový vektor. Tento problém vyřešíme pomocí lineární algebry: Z *B + 1* exponenčních vektorů vytvoříme matici, kde námi utvořené exponenční vektory budou sloupce této matice. Eliminujeme ji například Gaussovou eliminační metodou díky čemuž budeme schopni nalézt alespoň jedno řešení. Samozřejmě, že jedno konkrétní řešení budeme schopni nalézt vždy - tím řešením je prázdné řešení, přesněji součet prázdné množiny exponenčních vektorů. Výsledkem tohoto součtu je sice nulový vektor, ale pro naše potřeby je toto řešení nepoužitelné. Výše popsaný postup se nazývá *Dixonova faktorozační metoda*. Zde jsme ji popsali pouze v hrubých obrysech, detailněji se jí budeme věnovat v následujících kapitolách, kde probereme i implementační detaily.

Podívejme se ještě podrobněji na způsob, jakým hledáme B-hladká čísla. V *Dixonově faktorizační metodě* postupujeme číslo po čísle, odmocninou N počínaje. Protože ale hledáme čísla mající určitou vlastnost, mohl by existovat nějaký lepší (a především podstatně rychlejší) způsob, jak B-hladká čísla hledat. Připomeňme si algoritmus Erastothenova síta. Začneme s čísly v intervalu $[2 \ldots X]$. Jako první prvočíslo najdeme dvojku a vyškrtneme všechny její násobky ze seznamu. Dalším nevyškrtnutým číslem (po dvojce) je trojka, tedy je také prvočíslo. Vyškrtneme ze seznamu všechny její násobky a pokračujeme stále dál, dokud nedojdeme až k číslu $\sqrt{X}$. Pak všechna nevyškrtnutá čísla jsou prvočísla a algoritmus je u konce. Nás však ani tolik nezajímají nevyškrtnutá čísla, jako ta vyškrnutá. Dokonce čím vícekrát bylo nějaké číslo

zaškrtnuto, tím je pro nás zajímavější. Taková čísla jsou totiž dělitelná velkým počtem malých prvočísel, což nápadně připomíná B-hladká čísla. Uvedený princip hledání B-hladkých čísel využívá algoritmus *Kvadratické síto* (anglicky *Quadratic Sieve*). S novou metodou hledání těchto čísel ale přichází také různá úskalí, například volba velikosti prohledávaného intervalu. Dále je nutné vyřešit problém, co dělat v případě, že zvolený interval neposkytne dostatek B-hladkých čísel a stejně jako v *Dixonově metodě* budeme muset vhodně zvolit hranici B pro hladká čísla. Přes všechna tato úskalí je ale *Kvadratické síto* mocným nástrojem pro faktorizaci čísel a řešení problémů s ním spojených a možné optimalizace budou stěžejní částí tohoto textu.

1.3 Nástroje pro implementaci

Jako nejvhodnější prostředí pro implementaci faktorizačních algoritmů se jeví jazyk C++. Od programu budeme vyžadovat co největší výkonnost a při navrhování a testování optimalizací bude důležité mít plnou kontrolu nad „děním“ v programu. Dále budeme potřebovat hlídat paměťovou náročnost programu a zjišťovat její vliv na výkon, například při řešení rozsáhlých matic. Jazyk C++ nám toto vše umožní. Další možností by byla například Java nebo C#, naše aplikace ale nebude potřebovat žádné grafické rozhraní, které by se v těchto jazycích vytvářelo lépe a námi vybraný jazyk s největší pravděpodobností výkonnostně předčí tuto konkurenci. Protože však vestavěné datové typy jazyka C++ neumožňují jednoduše ukládat a manipulovat s celými velkými čísly (řádově desítky až stovky cifer), využijeme pro práci s těmito čísly knihovnu GMP šířenou pod GNU LGPL licencí (licenční podmínky zde [3]). Knihovnu je možné používat v jazyce C++ a umožňuje nejen snadnou manipulaci s velkými čísly a základní matematické operace s nimi, ale také pokročilejší operace prováděné modulo N nebo výpočet Legendre symbolu. Knihovna je také velmi výkonná (alespoň to autoři tvrdí) a má podrobnou dokumentaci, což značně usnadní její používání. Více o GMP knihovně lze nalézt na oficiálních stránkách [4].

1.4 Cíl práce

Vybrali jsme tedy vhodné nástoje pro implementaci a 4 algoritmy které slouží k faktorizaci čísel. Začali jsme tím nejjednodušším a postupně se propracovali až k složitějšímu, který je díky své výkonnosti používán i v praxi. Výběr algoritmů nebyl náhodný – *Postupné dělení* je přirozený způsob jak hledat faktory čísel a každý jej nevědomky často používá. Jeho výhodou je

jednoduchost a v optimalních případech také vysoká výkonnost, na druhé straně v nejhorším případě může být velice pomalý. Proto jsme se poohlédli po jiném principu hledání faktorů – kongruence čtverců se ukázala jako dobrý „trik“ a tak jsme do našeho výběru zahrnuli i *Fermatovu metodu*. Došli jsme však k závěru, že tento způsob lze ještě podstatně vylepšit a tak nám do výběru přibyli ještě dva další algoritmy, jmenovitě *Dixonova metoda* a *Kvadratické síto*. Cílem práce je implementovat všechny tyto algoritmy a odvodit s pomocí literatury jejich složitost. Implementace bude také sloužit jako prostředek k hlubšímu pochopení problémů s jednotlivými algoritmy spojených – ať už se jedná o volbu parametrů, způsob implementace nebo paměťovou náročnost. Na základě naměřených výsledků a experimentů s úpravami algoritmů se pokusíme formulovat konkrétní zásady při volbě parametrů a případně také navrhnout zlepšení implementace či algoritmu samotného. V následujících kapitolách postupně probereme všechny vybrané algoritmy do podrobna, upozorníme na problémová místa a zastavíme se u některých detailů implementace. Provedeme měření výkonu na testovacích datech a ze získaných výsledků a za pomoci teoretických znalostí navrhneme zlepšení či modifikace postupu nebo implementace algoritmů. Výsledky práce zúročíme v sekci o optimalizacích a v závěrečné kapitole shrneme výsledky naší práce. Začneme prvním z algoritmů: *Postupným dělením*.

Kapitola 2

Postupné dělení

Základní princip tohoto algoritmu jsme již zmínili, pojďme se ale na něj podívat trochu detailněji. Máme-li číslo N, budeme jej dělit postupně čísly od 2 až do $\sqrt{N}$. Nejprve se zamysleme, proč je hranice $\sqrt{N}$ postačující. Největší možný faktor který může N obsahovat je $N/2$. Tento faktor ale odhalíme už při prvním kroku postupného dělení – při dělení dvojkou, protože $2 \cdot (N/2) = N$. Druhý nejvyšší faktor by mohl být $N/3$, ale aplikováním předchozí úvahy opět zjistíme, že by byl odhalen daleko dříve. Takto můžeme postupovat stále dál, až se nám obě čísla „vyrovnají“, což nastane přesně ve chvíli kdy za nejvyšší možný faktor označíme $\sqrt{N}$. K jeho odhalení může dojít až při dělení číslem $\sqrt{N}$. Skutečně tedy postačí zvolit jako hranici pro dělení číslo $\sqrt{N}$.

Potřebujeme ale skutečně dělit číslo N všemi čísly $\leq \sqrt{N}$? Vzpomeneme-li si na příklad s Erastothenovým sítem z minulé kapitoly, je jasné že to není nutné. Nemusíme dělit N čtyřkou, protože kdyby N bylo dělitelné čtyřmi, bylo by dělitelné i dvěma, ale dvojka je menší než čtyřka tudíž bychom již měli faktor N ještě před dělením čtyřkou. Z toho plyne, že nám postačí dělit číslo N pouze prvočísly menšími než $\sqrt{N}$. Nyní máme téměř kompletní osnovu algoritmu, zbývá vyřešit situaci, kdy je nalezen faktor N, označme ho f_1. Bylo by chybou prostě pokračovat nejbližším dalším prvočíslem větším než f_1 a to hned ze dvou důvodů. Tím prvním je fakt, že číslo N může faktor f_1 obsahovat vícekrát. Proto musíme při nalezení faktoru opětovným dělením tímto faktorem ověřit zda se v čísle N nenachází další stejný faktor. Nalezení faktoru ale ústí v další událost, která má signifikantní vliv na provádění našeho algoritmu. Zaprvé, výsledek po dělení N nalezeným faktorem můžeme pro pokračování v práci brát jako nové N', stanovit tak nový limit $\sqrt{N'}$ a přitom pokračovat nejbližším větším prvočíslem než f_1. Důvod proč toto můžeme udělat tkví v tom, že již víme že N nemá žádné faktory menší než f_1, tedy ani N' nemůže žádné mít.

2.1 Implementace

Shrňme si vše co budeme pro funkční algoritmus potřebovat: Náš program bude určitě muset ukládat faktorizované číslo N a zaokrouhlenou hodnotu $\sqrt{N}$. Srdcem programu bude cyklus – v každém jeho průchodu budeme dělit N stále větším prvočíslem. Z toho vyplývá, že budeme muset mít k dispozici tabulku prvočísel až do velikosti $\sqrt{N}$. Generovat prvočísla ze běhu programu nepřipadá v úvahu – vzrůstal by tak potřebný čas pro každé jednotlivé dělení, protože bychom nejdříve museli „vypočítat“ dělitele. Ze stejného důvodu odpadá myšlenka testovat nejdříve každé číslo na prvočíselnost. Jako nejvhodnější řešení se jeví umístit tabulku prvočísel do externího souboru a načítat data z něj. Nakonec bylo zvoleno toto řešení, přesto i u něj narazíme na „rozpor“ mezi popisem algoritmu a jeho skutečnou implementací. Se vzrůstající hodnotou N (potažmo $\sqrt{N}$) totiž i roste velikost tabulky prvočísel kterou budeme potřebovat načíst. Je zřejmé, že ukládat celou tabulku do paměti počítače není možné - už při pěti milionech prvočísel by taková tabulka potřebovala asi 20 MB prostoru. Dnes to není mnoho, ale musíme si uvědomit, že by to byla pouze prvočísla o šesti cifrách, tedy abychom se dostali alespoň k limitu $\sqrt{N}$, nesmělo by číslo N mít více jak 12 cifer. Při vzrůstajícím počtu cifer N stoupá velikost tabulky prvočísel a už při osmnácti cifrách by nám paměť běžného počítače rozhodně nestačila. Jakmile vyčerpáme tabulku prvočísel, musíme se spokojit s dělením lichými čísly - tedy k poslednímu prvočíslu z tabulky přičteme dvojku, potom k novému číslu opět dvojku a tak dále. Algoritmus implementujeme jako třídu s patřičnou funkčností, což nám značně zpřehlední kód při pozdější implementaci dalších algoritmů a také umožní vytvořit jediný program obsahující 4 různé algoritmy místo odděleného programu pro každý z nich (v konečném důsledku můžeme ale díky třídám zrealizovat i tuto možnost). Třída bude obsahovat konstruktor a destruktor, jednu metodu pro vlastní faktorizaci a metodu pro resetování třídy – jednoduše tak umožníme použít instanci třídy znovu pro faktorizaci jiného čísla bez nutnosti použít druhou instanci stejné třídy. Dále privátní proměnné pro číslo N a jeho odmocninu a samozřejmě proměnnou pro aktuálně požívané prvočíslo a zdroj těchto prvočísel (tedy soubor). Vzhledem k tomu, že implementace tohoto základního algoritmu není nikterak složitá, dalšími detaily se v tomto textu zabývat nebudeme – v případě potřeby je v příloze možné nálézt kompletní dokumentaci stejně jako samotný kód s doplňujícími komentáři.

2.2 Vlastnosti algoritmu

Předtím než se pustíme do odvození složitosti a navrhování optimalizací, podíváme se na to, jaké vlastnosti *Postupné dělení* má. Vyzkoušejme, jak se bude algoritmus chovat při dvou vybraných zadáních. Nejdříve nechejme faktorizovat číslo $N = 12250$: Limit dělení bude roven $\sqrt{12250} \approx 110$. Faktor je nalezen hned v prvním kroku při dělení nejmenším prvočíslem – tedy dvojkou. Opakujeme dělení abychom vyloučili další výskyt dvojky a následně pokračujeme trojkou atd. Faktorizace skončí při dělení sedmičkou – nalezeny jsou všechny faktory:

$$12250 = 2^1 \cdot 3^0 \cdot 5^3 \cdot 7^2$$

Celkem bylo potřeba pouze deset operací dělení, první faktor byl nalezen hned prvním dělením. Nyní zkusme faktorizovat číslo $N = 12193$. Limit stanovíme na $\sqrt{12193} \approx 110$ a začneme opět dvojkou. Faktor však nalezneme až při dělení prvočíslem 89, výsledek po dělení je 137 a protože nový limit ($\sqrt{137} \approx 11$) je menší než aktuálně testované prvočíslo, je faktorizace u konce:

$$12193 = 89^1 \cdot 137^1$$

Celkem bylo v tomto případě potřeba 24 dělení. Ačkoliv obě čísla byla téměř stejně velká, faktorizace druhého čísla vyžadovala mnohem více kroků našeho algoritmu. Z toho můžeme usoudit, že hodně záleží na vlastnostech faktorizovaného čísla, přesněji řečeno na velikosti jeho faktorů. Pokud je číslo složeno z malých faktorů, je *Postupné dělení* velmi účinné, na druhé straně čím více se hodnota faktorů blíží $\sqrt{N}$, tím déle trvá jejich nalezení. Nejhorší případ nastane pokud je N prvočíslo – v takovém případě musíme totiž dělit až do $\sqrt{N}$ a teprve potom smíme s jistotou prohlásit, že N je prvočíslo. Prakticky tedy nezáleží na velikosti (počtu cifer) čísla N. Z tohoto důvodu se tento algoritmus dobře hodí na „odfiltrování“ malých faktorů z čísla N před započetím faktorizace některým z dalších algoritmů. Také je výhodné jej použít pro hledání B-hladkých čísel. Hodnota B nabývá malých hodnot, stačí tedy jako limit pro *Postupné dělení* zvolit číslo B a jestliže po konci algoritmu je testované číslo kompletně faktorizováno (tedy výsledek po dělení je roven jedné) našli jsme B-hladké číslo. V opačném případě číslo není B-hladké. Poslední nespornou výhodou algoritmu je jeho jednoduchost a z toho plynoucí rychlost. V podstatě se jedné pouze o neustálé opakované dělení čísla N nějakým prvočíslem dokud N není rovno jedné nebo není přesáhnut limit pro dělení. Na druhou stranou se v podstatě jedná o „útok hrubou silou“ – snažíme se najít faktor N prověřením všech možností. Než se pustíme do odvození složitosti, podíváme se ještě na praktické výsledky algoritmu při faktorizování různě velkých čísel.

2.3 Výsledky měření

Pro testování algoritmu byla zvolena čísla o deseti až dvaceti cifrách rozdělená do tří skupin. Nejprve jsem nechal program faktorizovat deset „běžných čísel“, každé z nich bylo produktem různě velkých prvočísel p a q (platí že počet cifer p je vždy větší než počet cifer q a to o 1 nebo o 2). Poté jsem jako vstup programu použil deset „obtížných čísel“, každé z nich bylo opět produktem dvou prvočísel p a q, ovšem v tomto případě měly p i q stejný počet cifer nebo se počet cifer lišil o 1. Nakonec jsem nechal faktorizovat 10 prvočísel. Faktorizace probíhala na počítači s procesorem AMD Turion 64 1800 Mhz s 1 GB DDR 166 Mhz pamětí RAM. Nejdříve se podíváme na tabulku pro „běžná čísla“:

Počet cifer	Číslo N	Čas [ms]
10	3075361313	1
11	72095523673	20
12	612536607203	19
13	5015278683343	145
14	42756946481693	113
15	669311744163467	927
16	5071030764490387	896
17	70147810349529461	8340
18	533997318134394911	7793
19	6373012344022668763	178303
20	62610101737108478503	158340

Tabulka 2.1: Postupné dělení - běžná čísla

Tabulka dobře ilustruje zmíněný fakt, že rychlost Postupného dělení závisí hlavně na velikosti jeho faktorů. Nejvíce patrné to je na posledních dvou číslech. Ačkoliv je poslední číslo téměř desetkrát větší než předchozí, jeho faktorizace trvala kratší dobu. Důvodem je, že menší z faktorů má u obou čísel stejný počet cifer (konkrétně 742843631 pro dvacetimístné číslo a 915894131 pro devatenáctimístné) a hodnota faktoru pro větší z čísel je o něco menší. Proto se k němu algoritmus „propracoval“ rychleji. Následující tabulka ukazuje trvání algoritmu pro „obtížná čísla“:

Počet cifer	Číslo N	Čas [ms]
10	8969340823	20
11	72095523673	19
12	501992367287	74
13	5015278683343	246
14	56354325068989	653
15	669311744163467	952
16	4338552375397507	5284
17	70147810349529461	8492
18	496117792773977623	112595
19	6373012344022668763	207696
20	58968037447783324577	1836529

Tabulka 2.2: Postupné dělení - obtížná čísla

Zde je již vidět neustálý nárůst doby faktorizace způsobený zvětšující se velikostí faktorů čísel. Ze zřejmých důvodů nebylo možné testovat algoritmus na vyšších číslech – už pro dvacetimístné zadaní trval celý proces více než půl hodiny. Situace je ještě horší v případě prvočísel:

Počet cifer	Číslo N	Čas [ms]
10	8969340841	21
11	72095523677	66
12	501992367319	77
13	5015278683377	260
14	56354325069011	817
15	669311744163487	2830
16	4338552375397529	7112
17	70147810349529527	44697
18	496117792773977713	129618
19	6373012344022668821	534224
20	58968037447783324589	-

Tabulka 2.3: Postupné dělení - prvočísla

Pro dvacetimístné prvočíslo jsem měření neprováděl, doba faktorizace by rozhodně přesáhla 30 minut. Důvodem je samozřejmě fakt, že algoritmus zjistí, že N je prvočíslo až po dosažení limitu pro dělení.

2.4 Složitost

Přestože se jedná o jednoduchý algoritmus, musíme si při určování složitosti uvědomit, že rychlost algoritmu závisí na faktorech daného čísla a ne na jeho velikosti. Faktorizace čísla 2^{64} by trvala pár milisekund přestože se jedná o dvacetimístné číslo, na druhou stranu faktorizování prvočísla o stejném počtu cifer by mohlo trvat desítky minut. Musíme se tedy zabývat nejhorším možným případem, tedy buď má číslo N faktory velikostně blízko k $\sqrt{N}$ a nebo se dokonce jedná o prvočíslo. V takovém případě musí algoritmus prověřit všechna prvočísla menší než $\sqrt{N}$ – tj. $\pi(\sqrt{N})$ [1] prvočísel. Složitost algoritmu vyjádříme tedy takto:

$$O(\pi(\sqrt{N})) \tag{2.1}$$

Tento zápis ale není až tak výstižný, proto provedeme aproximaci hodnoty $\pi(\sqrt{N})$ a složitost zapíšeme jako:

$$O\Big(\frac{2 \cdot \sqrt{N}}{\ln N}\Big) \tag{2.2}$$

V praxi je však složitost algoritmu o něco horší. Jak jsme již uvedli v kapitole „Vlastnosti algoritmu“, není možné dělit pouze prvočísly a dříve či později se budeme muset spokojit s dělením všemi lichými čísly. V takovém případě se složitost blíží až k

$$O\Big(\frac{\sqrt{N}}{2}\Big) \tag{2.3}$$

Výsledná složitost je někde mezi 2.2 a 2.3 přičemž čím větší je číslo N, tím více se blíží k 2.3. Je tomu tak proto že s rostoucí hodnotou N je stále větší část dělení uskutečněna pomocí lichých čísel na místo prvočísel.

2.5 Optimalizace

V předchozích kapitolách jsme podrobněji probrali algoritmus *Postupného dělení*. Nyní se podívejme, zda by nebylo možné vylepšit jeho vlastnosti. V tomto případě nám jednoduchost algoritmu neposkytuje moc velké možnosti jak jej optimalizovat, přesto se však pokusíme navrhnout vylepšení. Zaměříme se na situaci kdy je algoritmus nejméně výkonný – předpokládejme tedy, že číslo N které faktorizujeme nemá žádné „malé“ faktory, ba naopak, jeho faktory se blíží hodnotě $\sqrt{N}$. Z teorie (podpořené praxí) víme, že v takovém případě má algoritmus nejhorší vlastnosti.

[1] π $(\sqrt{N})$ je počet prvočísel menších nebo rovných $\sqrt{N}$

První možností, která by nás mohla napadnout je začít dělit od $\sqrt{N}$ a postupovat směrem „dolů“, tedy k nižším číslům. Následující tabulka ukazuje jak si modifikovaný algoritmus vede v porovnání s klasickým. Jako testovací data byla využita „obtížná čísla“:

Počet cifer	Číslo N	Čas [ms]	Čas (pozměněný alg) [ms]
10	8969340823	20	1
11	72095523673	19	46
12	501992367287	74	25
13	5015278683343	246	279
14	56354325068989	653	147
15	669311744163467	952	1921
16	4338552375397507	5284	1804
17	70147810349529461	8492	20968

Tabulka 2.4: Postupné dělení - obtížná čísla

Jak je vidět, výkonnost je velice nevyrovnaná, což potvrdila i další měření za použití jiných čísel (s obdobnými vlastnostmi). Pozměněný algoritmus je rychlejší vždy, kdy platí

$$\sqrt{N} - f_1 < f_1 \tag{2.4}$$

Tedy

$$\sqrt{N} < 2 \cdot f_1 \tag{2.5}$$

Kde f_1 je menší z dvojce faktorů jimiž je N tvořeno. Bohužel modifikovaný algoritmus má podstatnou nevýhodu – při nalezení faktoru N není možné jednoduše určit, zda je výsledek po dělení N tímto faktorem prvočíslo, což původní verze umožňovala. V tomto případě jsme sice věděli, že N je tvořeno dvěma prvočísli, obecně tomu ale tak samozřejmě není. Dále je nutné podotknout, že pro velká čísla N bude i tento algoritmus potřebovat enormní množství času pro nalezení byť jediného faktoru N, přestože by tento faktor mohl být relativně blízko k $\sqrt{N}$. Závěrem tedy konstatujme, že modifikovaný algoritmus je vhodný pouze ve speciálních případech optimální skladby čísla N. Možné zlepšení funkčnosti by mohla být určitá kombinace obou přístupů, ovšem pro obrovská čísla nebo pro čísla „nevhodných“ vlastností nebude ani takto vylepšený algoritmus přínosem.

Kapitola 3

Fermatova faktorizační metoda

V předchozí kapitole jsme poznali, že *Postupné dělení* může být velmi výkonný algoritmus, ale jeho síla vždy závisí na vlastnostech faktorů daného čísla. Také se ukázalo, že pro velká čísla je doba faktorizace neúnosně dlouhá. Proto je nutné se zaměřit na jiný způsob, jak hledat faktory. Základem Fermatovy metody je reprezentace lichého čísla jako rozdílu čtverců:

$$N = a^2 - b^2 \tag{3.1}$$

$$N = (a - b)(a + b) \tag{3.2}$$

Jinými slovy, při počítání modulo N:

$$0 \equiv a^2 - b^2 \pmod{N} \tag{3.3}$$

$$b^2 \equiv a^2 \pmod{N} \tag{3.4}$$

Tento vztah nazýváme „kongruence čtverců modulo N“. Najdeme-li tedy dvě čísla a, b taková, která vyhovují rovnici 3.4, budeme je moci využít k faktorizaci čísla N pomocí vztahu 3.2. Důležit podmínkou je aby $a \neq b$, protože kdyby tomu tak nebylo, pak by se pravá strana 3.2 rovnala nule – je zřejmé že takovouto dvojci a, b bychom k faktorizaci N využít nemohli. Stejně tak pokud by se některý z faktorů $(a - b)(a + b)$ rovnal jedné, není pro nás takováto faktorizace přínosem ($N = 1 \cdot N$).

3.1 Implementace

Základní princip algoritmu známe, můžeme se tedy pustit do implementace. Stejně jako v prvním případě, i nyní se jedná o jednoduchý algoritmus (ve své základní podobě). Opět budeme potřebovat ukládat faktorizované číslo N, dále čísla A a B. V cyklu budeme hledat taková čísla A, že $A^2 - N$ je

čtverec. Pokud nebude testovaná hodnota A (respektive z ní plynoucí hodnota B) vyhovovat, přičteme k ní jedničku a zkusíme to znova. Postup budeme opakovat dokud nenajdeme vyhovující dvojci A, B. Vzhledem k tomu, že implementace *Frematovy faktorizační metody* není nikterak složitá, dalšími detaily se v tomto textu zabývat nebudeme – v případě potřeby je v příloze možné nálézt kompletní dokumentaci stejně jako samotný kód s doplňujícími komentáři.

3.2 Vlastnosti algoritmu

Pokud se zamyslíme nad tím, jak algoritmus hledá faktory N, nutně nás musí zaujmout volba hodnoty A. Algoritmus totiž kongruenci čtverců hledá spíše náhodně. Jsou zkoušeny různé hodnoty A přičemž „doufáme“, že výsledná hodnota B bude čtverec. Neustálým zvyšováním hodnoty A sice procházíme sekvenci čísel postupně, to ale nic nemění na tom, že než najdeme vyhovující hodnoty můžeme také prověřit obrovské množství hodnot nevyhovujících. Postupné procházení sekvence čísel od $\sqrt{N}$ výše by nám také mohlo napovědět něco o chování algoritmu. Řekněme, že N je liché číslo a $N = c \cdot d$, pak platí:

$$N = \left(\left(\frac{c+d}{2} \right)^2 - \left(\frac{c-d}{2} \right)^2 \right) \tag{3.5}$$

Protože je N liché tak i c a d musí být lichá čísla. Pro další úvahu budeme předpokládat, že $c > d$. Tento předpoklad nemá žádný vliv na univerzálnost úvahy, stejně dobře bychom mohli předpokládat $d > c$ a došli bychom ke stejnému výsledku. Zlomky v 3.5 jsou tedy celá čísla. Dále předpokládejme že c je nejmenší faktor N větší než $\sqrt{N}$. Porovnáním se vztahem 3.1 dostaneme:

$$a = \left(\frac{c+d}{2} \right)^2 \tag{3.6}$$

$$b = \left(\frac{c-d}{2} \right)^2 \tag{3.7}$$

Z toho plyne, že algoritmus jako první najde nejmenší faktor větší než $\sqrt{N}$ (potažmo největší faktor menší než $\sqrt{N}$), protože hodnota A se postupně stále zvětšuje, tedy musí růst i hodnota (c + d). Můžeme tedy předpokládat, že se algoritmus bude chovat nejlépe, pokud bude mít N faktory blízko $\sqrt{N}$. To nápadně připomíná modifikovaný algoritmus *Postupného dělení*, který jsme testovali v předchozí kapitole. Ukázalo se ale, že tento přístup má svá omezení a i další nevýhody. Nyní se nám ale nabízí možnost využít síly *Postupného dělení* v kombinaci s *Frematovou metodou*. Nejdříve „odfiltrujeme“

malé faktory N pomocí *Postupného dělení* a poté dokončíme faktorizaci *Fermatovou metodou*. Jediné co budeme muset udělat je zvolit vhodný limit L pro *Postupné dělení* – limit musí být dostatečně velký na to, aby odstranil co nejvíce možných malých faktorů N, ale ne zas příliš velký, protože by jinak „filtrace“ trvala moc dlouho. Než se však pustíme do stanovování limitu, musíme nejprve znát výkonnost *Fermatovy metody*. Proto se nejdříve podíváme na výsledky měření – až poté budeme mít dostatek údajů pro navrhování optimalizací a volbu parametrů.

3.3 Výsledky měření

Pro testování algoritmu byla zvolena čísla o deseti až dvaceti cifrách rozdělená do tří skupin. Nejprve jsem nechal program faktorizovat deset „běžných čísel“, každé z nich bylo produktem různě velkých prvočísel p a q (platí že počet cifer p je vždy větší než počet cifer q a to o 1 nebo o 2). Poté jsem jako vstup programu použil deset „obtížných čísel“, každé z nich bylo opět produktem dvou prvočísel p a q, ovšem v tomto případě měly p i q stejný počet cifer nebo se počet cifer lišil o 1. Nakonec jsem nechal faktorizovat 10 prvočísel. Faktorizace probíhala na počítači s procesorem AMD Turion 64 1800 Mhz s 1 GB DDR 166 Mhz pamětí RAM. Nejdříve se podíváme na tabulku pro „běžná čísla“:

Počet cifer	Číslo N	Čas [ms]
10	3075361313	188
11	72095523673	42
12	612536607203	761
13	5015278683343	690
14	42756946481693	9100
15	669311744163467	4832
16	5071030764490387	132623
17	70147810349529461	110506
18	533997318134394911	-
19	6373012344022668763	-
20	62610101737108478503	-

Tabulka 3.1: Fermatova faktorizační metoda - běžná čísla

Pro větší čísla by faktorizace trvala neúnosně dlouho a tak bylo měření přerušeno. Překvapivě, *Fermatova metoda* vykazuje horší výsledky než *Postupné dělení* pro „běžná“ čísla. Tento stav odpovídá teoretickým výsledkům odvozeným v předchozích odstavcích – tento algoritmus je nejvhodnější pro

čísla jejichž faktory jsou co nejblíže $\sqrt{N}$. Námi použitá čísla však tuto vlastnost nemají. Podívejme se, jak si algoritmus poradí s „obtížnými“ čísly:

Počet cifer	Číslo N	Čas [ms]
10	8969340823	1
11	72095523673	96
12	501992367287	1
13	5015278683343	696
14	56354325068989	44
15	669311744163467	791
16	4338552375397507	97601
17	70147810349529461	7724
18	496117792773977623	709891
19	6373012344022668763	-

Tabulka 3.2: Fermatova faktorizační metoda - obtížná čísla

Výsledky faktorizace přesně odpovídají teoretickým úvahám z předchozích oddílů. Čísla jenž mají faktory blízko $\sqrt{N}$ byla rozfaktorizována mnohem rychleji než ta, která měla faktory dále od $\sqrt{N}$. Pokud si vzpomeneme na výsledky *Postupného dělení* (tabulky 2.1 až 2.3) všimneme si, že případy kdy *Fermatova metoda* pracuje nejrychleji, jsou pro *Postupné dělení* nejhorší a naopak. Toho využijeme později v kapitole věnované optimalizacím. Předtím se ale ještě krátce pozastavme u prvočísel – jak ukáže následující tabulka, Fermatova metoda se při „faktorizaci“ prvočísel ukázala jako krajně nevhodný způsob důkazu prvočíselnosti.

Počet cifer	Číslo N	Čas [ms]
7	2662771	433
8	19115479	2149
9	154857763	32447
10	1822413473	383309
11	13994153759	-

Tabulka 3.3: Fermatova faktorizační metoda - prvočísla

3.4 Složitost

Z teorie a praktických měření již víme mnohé o chování algoritmu, pokusme se tedy na základě těchto znalostí určit jeho složitost. Algoritmus jako

první najde faktorizaci s „nejmenšími“ hodnotami a a b. Tedy $a + b$ bude nejmenší faktor větší než $\sqrt{N}$ a $N/(a+b) = a-b$ bude největší faktor menší než $\sqrt{N}$. Mějme $N = c \cdot d$ kde c je největší faktor N menší než $\sqrt{N}$. Pak je $a = (c + d)/2$, proto je počet kroků algoritmu přibližně:

$$O(\frac{(c+d)}{2} - \sqrt{N}) \tag{3.8}$$

Uvedený vztah reprezentuje počet kroků, které musí algoritmus „absolvovat“ aby nalezl vhodné číslo a. My bychom však rádi vyjádřili složitost v závislosti na faktoru N, například na c. Toho dosáhneme několika úpravami – vyjádříme d jako N/c a upravíme zlomek:

$$O\Big(\frac{(c+d)}{2} - \sqrt{N}\Big)$$
$$O\Bigg(\frac{\Big(\frac{c^2+N}{c}\Big)}{2} - \sqrt{N}\Bigg)$$
$$O\Big(\frac{(\sqrt{N}-c)^2}{2c}\Big)$$

Zde také nalezneme vysvětlení, proč je pro prvočísla algoritmus tak nevýkonný. Pro prvočísla je totiž c = 1 a složitost je O(N) – signifikantně horší než v případě *Postupného dělení*! Na druhou stranu, algoritmus najde faktor N v několika málo krocích pokud se c blíží $\sqrt{N}$ – složitost pro takové případy se blíží O(1) jak ostatně potvrzuje následující tabulka:

Počet cifer	Číslo N	Čas [ms]
14	48783938702419	1
16	2914334121991753	3
18	547371581384847313	4
20	37697704347950405497	7

Tabulka 3.4: Fermatova faktorizační metoda - speciální čísla

Samozřejmě je výše uvedená tabulka trochu zavádějící. Jedná se o vysoce speciální případy, kdy se velikost faktoru N velmi blíží k $\sqrt{N}$ (rozdíl v řádu jednotek, maximálně desítek). Přesto je z naměřených výsledků jasně patrné, že ani *Fermatova metoda* (respektive její složitost) není závislá na velikosti čísla N jako spíše na vlastnostech jeho faktorů. Pojďme se tedy podívat jak bychom tuto vlastnost mohli využít či dokonce vylepšit.

3.5 Optimalizace

V předchozích kapitolách jsme se lehce dotkli možnosti spojení *Fermatovy metody* a *Postupného dělení*. Máme pro to zřejmý důvod, oba algoritmy se totiž vzájemně doplňují – tam kde jeden vykazuje nejhorší vlastnosti, druhý si vede mnohem lépe a naopak. Abychom tento vztah pochopili důkladněji, uvedeme jednoduchý příklad. Nechť $N = c \cdot d$ je složené číslo které chceme faktorizovat a c je prvočíselný faktor N menší než d. Čím více se bude c velikostí blížit $\sqrt{N}$, tím kratší dobu bude *Fermatova metoda* potřebovat k jeho nalezení. Na druhou stranu, *Postupné dělení* se k c dostane o to později, protože tento algoritmus nejdříve prověřuje menší čísla a postupně se propracovává k $\sqrt{N}$. Pokud ale bude c malé číslo blížící se k dvojce (nejmenší možný netriviální faktor N), bude *Fermatova metoda* potřebovat enormní množství výpočtů (blížící se k množství potřebné pro prvočísla, což je nejhorší možný případ). Naopak *Postupné dělení* najde tento faktor v prvních několika krocích, tedy velmi rychle. Zjednodušeně řečeno, oba algoritmy postupují při hledání faktorů každý z „jiné strany“. Toto vše nás vede k myšlence spojit oba algoritmy dohromady – nejdříve se pokusit najít malé faktory pomocí *Postupného dělení* omezeným nějakým limitem L a poté *Fermatovou metodou* faktorizaci dokončit. Zbývá pouze navrhnout velikost limitu L. Ideálně by měl být limit L nastaven tak, aby se k němu oba algoritmy dostaly za přibližně stejnou dobu, tedy zhruba v polovině intervalu $[2 \dots N]$. To však neplatí obecně pro všechna čísla – musíme vzít v potaz ještě několik faktů. Už z měření pro *Postupné dělení* vyplývá, že pro čísla s méně jak deseti ciframi je Postupné dělení dostatečně rychlé i v nejhorších případech (prvočísla). Proto by měl být limit L vždy alespoň pětimístné číslo. Dále je dobré vzít v úvahu jak velkou tabulku prvočísel máme k dispozici - při jejím použití má totiž *Postupné dělení* nejlepší možný výkon. Proto by měla hodnota limitu L být minimum z hodnot $(100000, \sqrt{N}/2)$. Tento způsob optimalizace přináší výhodu v tom, že eliminuje velké množství potenciálních faktorů *Postupným dělením* a poté *Fermatova metoda* pracuje lépe, protože případy pro které má nejhorší vlastnosti jsou již odfiltrovány. Nevýhodou je, že v případě, že se c blíží k hodnotě limitu, nepomůže nám tato optimalizace zrychlit hledání faktorů N a v nejhorších případech může dokonce vést i ke zpomalení celého procesu. Pokud by faktor c byl jen nepatrně větší než limit, provádění postupného dělení by bylo „ztrátou času“ a následná faktorizace *Fermatovou metodou* by trvala maximální dobu vzhledem k velikosti prohledávaného intervalu. Přitom jen několik kroků *Postupného dělení* navíc „za limit“ by vedlo k nalezení faktoru c.

Doposud jsme se zabývali „spoluprací“ *Fermatovy metody* a *Postupného dělení*. Zkusme se nyní zamyslet, zda by i samotná *Fermatova metoda* ne-

mohla být vylepšena. Při „zkoušení“ hodnot A totiž prověřujeme mnoho hodnot, které nemohou poskytnout výsledné B jako čtverec. Je to dáno tím, že s roustoucí hodnotou faktorizovaného čísla se zvětšují i rozestupy mezi čísly, které jsou perfektními čtverci. Naštěstí lze poměrně jednoduše poznat, zda nějaké číslo může být čtverec. Podívejme se na tabulku hodnot pro číslo 15483731:

A	3935	3936	3937	3938	3939	3940	3941	3942
B^2	494	8365	16238	24113	31990	39869	47750	55633
B	22.2	91.46	127.42	155.28	178.85	199.67	218.51	235.86

Tabulka 3.5: Tabulka hodnot pro číslo 15483731

Podíváme-li se na hodnoty B^2 je na první pohled vidět, že jen některé z nich mohou být čtverec. Je to dáno tím, že čtverce jsou vždy 0, 1, 4, 5, 9, 16 (mod 20) . Tyto hodnoty se navíc opakují při každém navýšení A o 10. V našem případě se opakují hodnoty 5, 9, 10, 13, 14, 18 (mod 20) – evidentně čtvercem může být pouze ona pětka. Pokud bychom ale počítali hodnotu $B^2 \pmod{20}$ pokaždé, moc bychom si tím práci neušetřili, stejně dobře bychom mohli vypočítat B a zjistit tak, zda není B^2 čtverec. B^2 je ale získáno výpočtem z A a N, takže pokud je

$$A^2 - N \equiv x \pmod{20} \tag{3.9}$$

kde X nabývá hodnot 0, 1, 4, 5, 9 nebo 16, pak můžeme vypočítat hodnotu pro A, protože N je konstanta. V našem případě nabývá X pouze hodnoty 5 a tak můžeme psát:

$$\begin{aligned} A^2 - 15483731 &\equiv 5 \pmod{20} \\ A^2 - 11 &\equiv 5 \pmod{20} \\ A^2 &\equiv 16 \pmod{20} \end{aligned}$$

Čili pro A máme řešení 4, 6, 14 a 16 (mod 20) což zjednodušíme na 4 a 6 (mod 10). Postačí nám tedy „zkoušet“ pouze takové hodnoty A, které jsou rovny 4 nebo 6 (mod 10). Nemusíme dokonce ani počítat hodnotu modulo pro všechna A – stačí nalézt první vyhovující a přičítáním desítky získáme jednoduše příslušnou další hodnotu. Pomocí této optimalizace budeme schopni zredukovat počet čísel které bude nutné prověřit, čímž dosáhneme zrychlení celého algoritmu. Přesto se však jedná o optimalizaci, která nezvyšuje výkonnost nijak dramaticky – pokud bude hledaný faktor c podstatně menší než $\sqrt{N}$, pořád bude nutné projít dlouho sekvenci čísel, ikdyž „řidší“ než v původním algoritmu.

Při návrhu lepší optimalizace využijeme situace, kdy má *Fermatova metoda* nejlepší výsledky. Mohli bychom se pokusit dosáhnout toho, aby taková situace nastala. Pokud bychom znali přibližný poměr velikostí faktorů $\frac{d}{c}$, zvolili bychom číslo $\frac{u}{v}$ blízké této hodnotě. Potom

$$N \cdot u \cdot v = (c \cdot v) \cdot (d \cdot u)$$

Faktory jsou nyní přibližně stejně velké, takže budou nalezeny velmi rychle při faktorizaci $N \cdot u \cdot v$ *Fermatovou metodou*. Pak největší společný dělitel $(N, c \cdot v) = c$ a $(N, d \cdot u) = d$, pokud c nedělí u a d nedělí v.). V praxi samozřejmě poměr faktorů c a d neznáme, proto se zde pokusíme navrhnout způsob, jak volit hodnotu $\frac{u}{v}$ při hledání „správného poměru“ abychom pokud možno zaručili, že faktor bude nalezen. Vyjdeme z předpokladu, že $N = c \cdot d$ a platí $c < \sqrt{N} < d$. Nyní budeme systematicky volit hodnotu c a vždy dopočítáme (přibližnou) hodnotu d. Z toho nám potom vyplyne potřebná hodnota $\frac{u}{v}$ – tu aproximujeme a získáme tak čísla u a v. Pak provedeme faktorizaci čísla $N \cdot u \cdot v$ s určitým limitem L, tedy jak „daleko“ od $\sqrt{N \cdot u \cdot v}$ budeme hledat. V případě nenalezení faktorů $N \cdot u \cdot v$ v daném limitu zvolíme novou hodnotu c a opakujeme celý proces. Zbývá vyřešit volbu hodnoty faktoru c. Mějme interval $[2 \ldots \sqrt{N}]$, který představuje všechny relevantní hodnoty pro c. Předpokládejme, že krajní meze jsme již prověřili dříve (dolní mez například *Postupným dělením*, horní *Fermatovou metodou* se stanoveným limitem prohledávání). Pak budeme hodnotu c volit iterativně. V Y-té iteraci zvolíme 2^{Y-1} různých hodnot

$$\frac{Z \cdot \sqrt{N}}{2^Y}$$

kde Z jsou lichá čísla z intervalu $[1 \ldots 2^Y]$. V první iteraci máme jednu hodnotu $[\frac{\sqrt{N}}{2}]$, v druhé dvě hodnoty $[\frac{\sqrt{N}}{4}, \frac{3\sqrt{N}}{4}]$, ve třetí čtyři $[\frac{\sqrt{N}}{8}, \frac{3\sqrt{N}}{8}, \frac{5\sqrt{N}}{8}, \frac{7\sqrt{N}}{8}]$ atd. Popsaný postup v podstatě prohledává daný interval stále podrobněji a podrobněji. Stanovení limitu pro samotnou faktorizaci *Fermatovou metodou* záleží na velikosti N. Její volba je klíčová pro tento algoritmus – příliš malá hodnota by znamenala velmi úzký prostor kolem zvoleného čísla a také nebezpečí vynechání určitých částí intervalu kvůli tomu, že hodnoty u a v pouze aproximujeme. Je lepší raději volit hodnotu větší, což zároveň i sníží počet potřebných iterací.

Kapitola 4

Dixonův algoritmus

V předchozích kapitolách jsme se zabývali dvěma algoritmy, které měly jeden důležitý společný rys. Jejich výkonnost vždy závisela na vlastnostech faktorů čísla N. Ukázalo se, že tato vlastnost může být někdy výhodou, ovšem častěji byla spíš hlavním limitujícím prvkem. Teď se začneme zabývat algoritmy, jejichž výkonnost záleží téměř výhradně na velikosti faktorizovaného čísla. Mohlo by se zdát, že nyní probírané postupy budou zcela odlišné, než ty v předchozích kapitolách, ale není tomu tak. Ve skutečnosti tyto algoritmy principielně vychází z *Fermatovy faktorizační metody*. I *Dixonův algoritmus* se snaží najít faktorizaci čísla N pomocí kongruence čtverců, ovšem k nalezení vhodných čísel A a B využívá jiné metody než Fermatův algoritmus. Základní princip jsme již nastínili v úvodní kapitole, proto si jej nyní ukážeme v praxi a upozorníme na některé detaily. Zkusíme faktorizovat číslo $N = 99653$, B pro hladká čísla zvolíme například 10. Máme tedy 4 prvočísla menší jak 10, tedy báze faktorů je $[2, 3, 5, 7]$. Hledání začneme číslem 316, což je první celé číslo větší než $\sqrt{N}$. Musíme najít alespoň o jedno B-hladké číslo více než máme čísel v bázi faktorů. Postupováním od čísla 316 výše najdeme těchto pět B-hladkých čísel:

$$
\begin{aligned}
547^2 \pmod N &= 250 = 2^1 \cdot 3^0 \cdot 5^3 \cdot 7^0 \\
715^2 \pmod N &= 12960 = 2^5 \cdot 3^4 \cdot 5^1 \cdot 7^0 \\
861^2 \pmod N &= 43750 = 2^1 \cdot 3^0 \cdot 5^5 \cdot 7^1 \\
893^2 \pmod N &= 225 = 2^0 \cdot 3^2 \cdot 5^2 \cdot 7^0 \\
1094^2 \pmod N &= 1000 = 2^3 \cdot 3^0 \cdot 5^3 \cdot 7^0
\end{aligned}
$$

Jejich exponenční vektory (mod 2) dáme do matice jakožto sloupce:

$$\begin{pmatrix} 11101 \\ 00000 \\ 11101 \\ 00100 \end{pmatrix}$$

Abychom nalezli lineární kombinaci sloupcových vektorů tvořící nulový vektor, potřebujeme vyřešit následující rovnici:

$$\begin{pmatrix} 11101 \\ 00000 \\ 11101 \\ 00100 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{4.1}$$

Samozřejmě nulový vektor vyhovuje naší rovnici, ale toto řešení nám nijak nepomůže, neboť $A = 0$ a $B = 0$ vždy vyhovují rovnici $A^2 \equiv B^2 \pmod{N}$. Zajímají nás pouze řešení ve kterých se vyskytuje alespoň jedna jednička, tedy alespoň jedno nalezené B-hladké číslo. V tomto příkladě jsme vlastně měli štěstí, protože se nám podařilo najít kongruenci čtverců. Nalezené B-hladké číslo $225 = 15^2$ a tedy $892^2 \equiv 15^2 \pmod{99653}$ A tedy vektor (0, 0, 0, 1, 0) je řešením rovnice 4.1. Zkusme na chvíli předstírat, že jsme na kongruenci čtverců nenarazili, tedy naše matice nemá žádný nulový sloupec. Musíme tedy nulový sloupec „vytvořit“ ze sloupců které máme. Na první pohled je vidět, že to v tomto případě nebude nikterak složité. Například součet prvního a pátého sloupce dává nulový sloupec, tedy řešení rovnice je (1, 0, 0, 0, 1). Můžeme psát:

$$(547 \cdot 1094)^2 \equiv 250 \cdot 1000 \pmod{99653} \tag{4.2}$$

$$598418^2 \equiv 500^2 \pmod{99653} \tag{4.3}$$

Z čehož nám plyne $A = 598418$ a $B = 500$. Nyní už jen stačí faktorizaci dokončit spočtením největšího společného dělitele $(A - B)$ (potažmo $A + B$) a N:

$$gcd(A - B, N) = 439$$

Druhý faktor dopočítáme jednoduchým vydělením a máme $99653 = 439 \cdot 227$.

Podívejme se blíže na některé části předvedeného postupu. První co by nás mohlo zaujmout je volba limitu B pro B-hladká čísla. Logicky čím velkorysejší tento limit bude, tím větší šance že nalezneme B-hladká čísla. Proč jsme tedy v tomto případě nezvolili limit například $B = 100$? Při tomto limitu bychom určitě nalezli mnohem rychleji B-hladká čísla. Problém ale tkví

v tom, že nyní nám jich nebude stačit 5. Prvočísel menších než 100 je 25, takže budeme potřebovat nejméně 26 B-hladkých čísel. Sice zvýšíme šanci na nalezení takových čísel, zároveň ale zvýšíme i jejich potřebný počet. Další věc je, že čím více B-hladkých čísel budeme mít, tím větší matice budeme nuceni řešit. Ve výše uvedeném příkladě bylo hledání řešení jednoduché, protože matice byla malá. V reálných příkladech při faktorizaci obrovských čísel jsou rozměry matice v řádech stovek, tisíců nebo i více. Je zřejmé, že volba limitu B bude jedním z klíčových momentů pro faktorizaci *Dixonovým algoritmem*. Další zajímavou pasáží je hledání lineární kombinace exponenčních vektorů v matici z rovnice 4.1. V našem příkladě bylo řešení tohoto problému triviální a to jsme ještě záměrně „přehlédli“ nalezenou kongruenci čtverců. Při faktorizaci větších čísel je ale šance na nalezení kongurence čtverců malá (až na speciální případy) – to ostatně víme již z analýzy *Fermatovy metody*. Nemůžeme se tedy spoléhat na štěstí a nezbývá nám nic jiného než matici vyřešit. Víme, že netrivální (nenulové) řešení musí existovat – to jsme zaručili tím, že matice má vždy alespoň o jeden sloupec více než řádků. Využijeme tedy *Gaussovy eliminační metody* k převedení matice na *horní trojúhelníkovou matici* a pomocí zpětného dosazení najdeme všechna řešení matice (protože veškeré výpočty provádíme modulo 2, má matice konečný počet řešení). Poté prověřováním těchto řešení hledáme netriviální faktor N. Je nutné si uvědomit, že ne každé řešení nám může poskytnout netriviální faktor. Vrátíme-li se k příkladu v předchozím odstavci, tak výpočet největšího společného dělitele $A - B$ a N (kde hodnoty A a B jsou získány z 4.3) nevede k žádnému netriviálnímu faktoru N. Známe již podrobně celý proces faktorizace *Dixonovou metodou*, pojďme se tedy podívat na způsob implementace tohoto algoritmu.

4.1 Implementace

Realizace *Dixonova algoritmu* už není tak jednoduchá jak u předchozích algoritmů, proto se na některé části programu podíváme podrobněji. Nejdříve musíme vytvořit bázi faktorů, tedy načíst všechna prvočísla menší než zvolený limit B. Protože s těmito čísly budeme pracovat velice často, vytvoříme pro ně pole hodnot typu `integer`, které nazveme `factorBase`. Než tak však učiníme, musíme přesně vědět kolik prvočísel budeme muset uložit – nejdříve tedy spočítáme kolik prvočísel je menších než B a až poté je uložíme do alokované `factorBase`. Jakmile máme připravenou bázi faktorů, můžeme se pustit do hledání B-hladkých čísel. Ty ale budeme potřebovat ukládat a to nejen jejich hodnotu, ale také exponenční vektor a i číslo, ze kterého toto B-hladké číslo vzešlo. Tuto trojci hodnot budeme v dalším textu pro jednoduchost nazývat *relací*. Pro tyto účely vytvoříme strukturu obsahující tyto tři hodnoty.

Pole těchto struktur budeme využívat pro ukládání $B + 1$ relací a následné tvorby matice exponenčních vektorů. Později jej využijeme při dopočítávání hledaných čísel A a B. Protože prvočísla v bázi faktorů budou spíše malá (a to i při velkých hodnotách N), postačí nám vestavěný typ jazyka C++ – `integer`. Pro exponenční vektor se nám výborně hodí typ `boolean` (zajímá nás pouze, zda je exponent sudý či lichý). Poslední co musí struktura obsahovat je číslo, ze kterého nám B-hladké číslo vzniklo. Jeho velikost už je nad možnosti vestavěných typů jazyka C++ (hledání začínáme u $\sqrt{N}$)a tak využijeme typ `mpz_t` knihovny GMP. Po nalezení dostatečného počtu hladkých čísel přejdeme k další fázi algoritmu - hledání vhodné kombinace těchto čísel tak, aby vytvořili kongruenci čtverců.

Matici budeme řešit *Gaussovou eliminací* – budeme potřebovat nějakým způsobem zajistit, aby v případě většího počtu řešení bylo možné prověřit všechny. Proto po *Gaussově eliminaci* identifikujeme „volné“ proměnné a vytvoříme pro ně zvláštní vektor `fVarVect` typu `boolean`. Tento vektor bude mít vždy alespoň jednu položku. Budeme na něj nahlížet jako na binární číslo – postupným „přičítáním jedničky“ tak vyzkoušíme všechny možné kombinace pro volné proměnné. Následně tento vektor dosadíme do konečného vektoru řešení a dopočítáme zbylé proměnné.

4.2 Vlastnosti algoritmu

Jak již bylo řečeno, algoritmus je svým chováním podstatně odlišný od předchozích dvou. Prakticky nezáleží na vlastnostech faktorů čísla N – důležitá je hlavně velikost tohoto čísla. Je tomu tak proto, že hledáme pouze B-hladká čísla, ne kongruenci čtverců nebo dokonce přímo faktory. Šance na nalezení B-hladkého čísla je podstatně vyšší, než při hledání čtverců/faktorů, zvlášť při velkém limitu B. Na druhou stranu B-hladkých čísel potřebujeme více než jedno, přesněji $B + 1$. Musíme tedy vyvážit dvě protichůdné síly – na jedné straně nám velký limit B zvyšuje pravděpodobnost nalezení vhodného čísla. Na druhé straně, čím větší limit máme, o to více čísel potřebujeme najít a také o to více dělení musíme při testování provést. Stejně tak velký počet relací zvětšuje matici s kterou budeme muset pracovat a prodlužuje tak dobu jejího řešení. Z toho je patrné, že volba limitu B bude jedním z klíčových momentů při používání tohoto algoritmu v praxi. S rostoucí hodnotou N také roste rozptyl těchto čísel, tím pádem i množství hodnot které budeme testovat „zbytečně“. Proto budeme muset založit volbu B hlavně na velikosti faktorizovaného čísla.

Čistě teoreticky bude algoritmu trvat stejnou dobu nalezení jakéhokoliv faktoru, ať už se bude jednat o malé prvočíslo nebo o faktor řádově $\sqrt{N}$. Plyne

to už z principu algoritmu – hledání B-hladkých čísel totiž s faktory N nemá moc společného, nelze říct, že N s malými faktory bude mít podstatně větší koncentraci B-hladkých čísel než přibližně stejně velké číslo tvořené dvěma velkými prvočíselnými faktory. V podstatě koncentrace „dobrých“ čísel bude velmi podobná. Tento fakt ověříme v následující kapitole, kde také položíme základ pro odvození volby vhodné hodnoty limitu B.

4.3 Výsledky měření

Oproti předchozím dvěma algoritmům bylo testování trochu jiné. Tentokrát pro testy nebyla použita prvočísla, protože *Dixonův algoritmus* není pro tyto případy dobře uzpůsoben – v praxi se běžně nejdříve číslo testuje na prvočíselnost některým z pravděpodobnostních algoritmů, které jsou k tomuto uzpůsobeny. K samotné faktorizaci se přejde jedině pokud je výsledkem tohoto testu, že N je složené číslo. Z teoretického rozboru vlastností algoritmu víme, že volba limitu B je klíčovou pro jeho výkon. Nejdříve tedy provedeme několik testů na jejichž základě navrhneme vhodnou metodu volby parametru B, správnost metody prověříme při dalších testech a rozebereme v kapitole o Složitosti a Optimalizaci.

Faktorizace probíhala na počítači s procesorem AMD Turion 64 1800 Mhz s 1 GB DDR 166 Mhz pamětí RAM. Pro testování vhodné volby limitu B jsem využil číslo o patnácti cifrách (složené ze dvou přibližně stejně velkých prvočísel) tak, aby faktorizace netrvala příliš dlouho, ale přitom dost dlouho na to aby rozdíly v časech byly jasně patrné. Nechal jsem číslo faktorizovat s různými hodnotami limitu B. Dá se očekávát, že výkonnost bude prudce klesat když budeme volit příliš nízké hodnoty, stejně tak bude klesat (ovšem pozvolněji) při volbě příliš vysokých hodnot. Odhadem jsem na základě zkušeností zvolil „vhodnou“ hodnotu a nechal jsem faktorizovat číslo N postupně hodnotami menšími a většími než mnou navržená optimální hodnota. Pro tento případ jsem jako optimální hodnotu zvolil rovný jeden tisíc. To představuje 168 prvočísel v bázi faktorů. Jak je vidět z následující tabulky, nebyla to zdaleka nejlepší volba.

Číslo N	limit B	Báze faktorů	Čas [ms]
669311744163467	700	125	150862
669311744163467	800	139	154885
669311744163467	900	154	104802
669311744163467	1000	168	129400
669311744163467	1100	184	107170
669311744163467	1200	196	97563
669311744163467	1300	211	83904
669311744163467	1400	222	76754
669311744163467	1500	239	111007
669311744163467	1600	251	124601
669311744163467	1700	266	124601
669311744163467	1800	278	136801
669311744163467	1900	290	124723
669311744163467	2000	303	129457

Tabulka 4.1: Dixonův algoritmus - volba limitu B

Jako optimální hodnota limitu B se jeví číslo v rozmezí 1200 až 1400, tedy asi 200 až 220 prvočísel v bázi faktorů. Všimněme si také, že výkonnost při menší bázi faktorů byla o poznání horší než pokud byla báze faktorů větší. Z toho můžeme usuzovat, že volba větší báze faktorů je lepší variantou, než zvolit příliš malou. Patrně je tomu tak proto, že řešení matice trvá podstatně méně času než hledání B-hladkých čísel. Podívejme se nyní na hodnotu limitu B doporučovanou v [2]:

$$e^{1/2\sqrt{\ln N \cdot \ln \ln N}} \tag{4.4}$$

Hodnota tohoto výrazu pro číslo N z předchozí tabulky je zhruba 242, tedy o něco více, než jsme určili my měřením (200-220). Nyní jsme provedli měření na dalších číslech přičemž jsme limit B vypočítali dle vztahu 4.4. Vždy jsme pak limit podstatně zvýšili (zdvojnásobili, případ 1) a snížili (poloviční báze faktorů, případ 2) a porovnali výsledky:

Číslo N	Báze faktorů	Čas [ms]	Změna BF 1	Změna BF 2
3075361313	60	561	1002	412
72095523673	89	963	1572	565
612536607203	114	3775	5877	4417
5015278683343	143	6405	8648	7600
42756946481693	181	19139	42411	43671
669311744163467	242	97387	127412	168396

Tabulka 4.2: Dixonův algoritmus - porovnání

Jak je vidět, velikost báze faktorů zvolená na základě vztahu 4.4 vede k dobrým výsledkům algoritmu. V kapitole o složitosti si uvedený vztah ještě připomeneme, protože je pro odvození složitosti nezbytností. Následně se pak v závěrečné kapitole o optimalizaci zamyslíme nad platností tohoto vztahu pro různě velká čísla – jak uvidíme, může i na první pohled dobrý vzorec vykazovat v praxi horší výsledky.

4.4 Složitost

Při určování složitosti *Dixonova algoritmu* si musíme uvědomit, že jeho složitost úzce souvisí s pravděpodobností na nalezení B-hladkého čísla, tedy i s volbou hodnoty B (potažmo velikosti báze faktorů). Pokud zvolíme optimálně číslo B a ideálně vyvážíme dvě protichůdné síly, tedy pravděpodobnost nalezení B-hladkého čísla a počet takových čísel který musíme najít, dosáhneme také ideální složitosti *Dixonova algoritmu*. Protože se nám vztah 4.4 osvědčil v praxi, použijeme jej jako základ při odvození složitosti. Podívejme se nejdříve na jednotlivé kroky algoritmu: máme číslo k a platí $1 \leq k < N$ a musíme vypočítat největšího společného dělitele k a N na což spotřebujeme $O(\ln^3 N)$ kroků. Následně musíme spočítat $Q(k) = k^2 \pmod{\mathrm{N}}$ což vyžaduje $O(\ln^2 N)$ operací. Nyní je nutné zjistit, zda je Q(k) B-hladké číslo dělením pvočísly v bázi faktorů. Pokud toto provedeme chytře, vystačíme si s $O(F \cdot \ln^3 N)$ operacemi, kde F je počet čísel v bázi faktorů. Nyní přichází na řadu ona pravděpodobnost, že číslo k bude B-hladké a také fakt, že těchto čísel potřebujeme $B + 1$. Uvedeme výsledný vztah s odkazem na [5] pro detailní popis jak se k tomuto výsledku dostaneme:

$$O(F^2 \cdot \ln \ln N) + O\Big(F^2 \cdot \ln^3 N \cdot \Big(\frac{\psi(N,y)}{N}\Big)^{-1}\Big) \tag{4.5}$$

Kde výraz $\left(\frac{\psi(N,y)}{N}\right)$ představuje pravděpodobnost, že číslo z intervalu $[1 \ldots N]$ má všechny prvočíselné faktory menší než y. Máme tedy výraz pro složitost hledání B-hladkých čísel. Nyní se podívejme na další část algoritmu, konkrétně na řešení matice. Gaussova elmininace nás bude stát $O(F^3)$ kroků. Tímto získáme řešení matice a budeme počítat hodnotu lineárních kombinací B-hladkých čísel. Jeden takový výpočet vyžaduje $O(F^2 \cdot \ln \ln N)$ kroků. K tomu ještě musíme přičíst dělení exponenčních vektorů dvěma, tedy $O(F^2 \cdot \ln N)$ kroků. Celkově nám vychází složitost této části algoritmu na:

$$O(F^3) + O(F^2 \cdot \ln \ln N) + O(F^2 \cdot \ln N) \tag{4.6}$$

Nyní budeme počítat obě části relace, tedy čísla A a B. Pro první případ máme pouze násobení čísel modulo N, což je možné provést za $O(F \cdot \ln^2 N)$

kroků. Druhý případ je obtížnější, protože pracujeme s více čísly – přesněji s exponenčním vektorem. Zde je zapotřebí $O(F \cdot \ln F \cdot \ln^2 N) + O(F \cdot \ln^2 N \cdot \ln \ln N)$ kroků. Nakonec pouze spočítáme největšího společného dělitele $A + B$: $O(\log^2 N)$ kroků. Celkem tedy máme pro poslední část algoritmu složitost:

$$O(F \cdot \ln^2 N) + O(F \cdot \ln F \cdot \ln^2 N) + O(F \cdot \ln^2 N \cdot \ln \ln N) + O(\ln^2 N) \quad (4.7)$$

Což zjednodušíme na

$$O(F \cdot \ln F \cdot \ln^2 N) + O(F \cdot \ln^2 N \cdot \ln \ln N) \quad (4.8)$$

Sečtením 4.5, 4.6 a 4.8 a zvolením vhodného limitu B (z něj plyne hodnota F) dle 4.4 dostaneme výraz pro optimální složitost *Dixonova algoritmu*:

$$e^{2 \cdot \sqrt{\ln N \cdot \ln \ln N}} \quad (4.9)$$

Pro podrobnější odvození odkážeme čtenáře na [5].

4.5 Optimalizace

Samozřejmě musíme připustit, že v určitých případech může i tento algoritmus těžit ze speciálních vlastností faktorů N. Ku příkladu si představme, že má N faktor velmi blízký $\sqrt{N}$. Potom je vysoce pravděpodobné, že při hledání B-hladkých čísel dojdeme až k tomuto faktoru. Proto v algoritmu vždy počítáme největšího společného dělitele testovaného čísla a čísla N. Je to ale nutné, přesněji řečeno, je to vždy výhodné? Podat zde jasnou odpověď není tak snadné. Z praktického hlediska je však tento výpočet v drtivé většině případů (téměř vždy) nadbytečný. Proč? Čísla s takovýmito faktory totiž můžeme lehce odfiltrovat pomocí *Fermatovy metody*, která má pro tyto případy vynikající výsledky. Nadbytečnost této operace plyne i z praktických měření. U čísel řádově desetimístných se ještě vyplatí tuto operaci v algoritmu ponechat, ale jak číslo roste, stává se tento krok nadbytečným, protože i faktory relativně „blízké“ $\sqrt{N}$ jsou příliš daleko na to aby byly tímto způsobem nalezeny.

Zaměřme se ještě na řešení matice. Jak jsme poznali v příkladu z úvodu kapitoly, je možné, že v naší matici najdeme nějaké speciální případy. Tím prvním je samozřejmě nalezení kongruence čtverců, v řeči naší matice tedy nulový sloupec. Je jasné, že tento případ bude velmi ojedinělý, na druhou stranu by byla chyba ho úplně opomenout. Proto přidáme do algoritmu ještě drobnou úpravu – při tvorbě exponenčního vektoru budeme jednoduše hlídat, zda je některý exponent lichý. Pokud alespoň jeden takový najdeme, nic se

neděje. Pokud ale žádný takový nenajdeme a číslo bude B-hladké, našli jsme kongruenci čtverců. Místo toho abychom tuto relaci přidali do matice, okamžitě zkusíme zda pro nás z ní neplynou faktory N. Pokud ano, máme štěstí a algoritmus může skončit, pokud ne, pokračujeme v hledání B-hladkých čísel. Důležité je ovšem to, že nalezenou kongruenci čtverců do matice nepřidáváme. Je tomu tak proto, že by tam byla navíc. Mohla by totiž patřit do každé výsledné lineární kombinace, stejně tak jako do žádné, na výsledku by to v podstatě nic nezměnilo. Toto ale není jediný speciální případ, který můžeme v matici objevit. Dalším je nulový řádek. Každý takovýto řádek nám totiž přidává jednu volnou proměnnou, což pro nás představuje větší počet řešení a tak větší šanci na nalezení faktoru N. Pořád tu ale je ještě další zajímavý případ, který může v naší matici nastat. Tím je řádek, ve kterém je pouze jedna jednička. Ačkoliv se může na první pohled zdát, že se nejedná o nic zajímavého, opak je pravdou. Jestliže máme v řádku pouze jedinou jedničku, není možné vytvořit lineární kombinaci takovou, aby výsledkem byl nulový vektor a přitom byl sloupec na kterém tato jednička je zahrnut do této kombinace. Řečeno jinými slovy, máme prvočíslo, které je v lichém počtu přítomno pouze v jediném B-hladkém čísle, ve všech ostatních je v počtu sudém. Z toho plyne, že můžeme směle odstranit řádek s tímto prvočíslem a zároveň s ním i ono B-hladké číslo, které obsahovalo lichý výskyt vyřazeného prvočísla v exponentu (tedy příslušný sloupec). Takovýmto řádkům budeme říkat „singletony“ – řádky pouze s jednou jedničkou. Odstraňováním singletonů zmenšujeme velikost matice a tím i dobu jejího řešení, aniž bychom nějak ovlivnili výsledek. Nutno podotknout, že tento proces musí být prováděn iterativně – odstraněním sloupce a řádku jsme mohli vytvořit další speciální místa v matici. Proto předtím, než začneme matici řešit, provedeme tento „preprocessing“. Pro malé matice nemá smysl a spíš by řešení prodloužil, nehledě na fakt, že by mohl matici zničit – odstraněných řádků by mohlo být příliš mnoho (všechny). Pro velké matice je však tento postup nezbytný.

Vylepšit algoritmus však můžeme i jinými postupy, než jen úpravou jeho kroků. Výhodou hledání B-hladkých čísel je fakt, že je tento úkon možné rozdělit mezi více řešitelů. Budeme potřebovat řídícího řešitele který bude hledání B-hladkých čísel „organizovat“. Ten zvolí nějaký limit I a podle něj rozešle každému z podřízených řešitelů meze intervalu, kde má hledat vhodná čísla. Tedy první řešitel obdrží interval $[\sqrt{N} \ldots \sqrt{N} + I]$, druhý $[\sqrt{N}+I \ldots \sqrt{N}+(2 \cdot I)]$ a tak dále. Jednotliví řešitelé budou řídícímu členovi posílat nalezená vhodná čísla a ten je bude ukládat u sebe. Pokud některý řešitel dosáhne konce intervalu dříve než je nalezeno dostatek B-hladkých čísel, je mu přidělen interval nový. Jakmile je nalezeno dostatek čísel, řídící řešitel zastaví ostatní a z nasbíraných B-hladkých čísel vytvoří matici kte-

rou následně vyřeší. Jak je vidět, řešitelé mohou být třeba jen jednotlivá vlákna/procesy na jednom systému nebo samostatné počítače v nějaké síti.

Kapitola 5

Kvadratické síto

Dosud jsme se zabývali algoritmy, které se v praxi v podstatě nepoužívají, pokud chceme pomýšlet na skutečně velká čísla. Nyní se konečně zaměříme na používaný algoritmus, jenž je pro čísla řádově až 120 míst nejrychlejší. *Kvadratické síto* vychází z *Dixonova algoritmu* a dalo by se říct, že je vlastně jeho vylepšením. Nárůst výkonnosti je zde ale opravdu značný, o čemž se ostatně přesvědčíme při faktorizaci. Jak jsme již upozornili v kapitole o Dixonově algoritmu, dobu provádění faktorizace ovlivňuje nejvíce hledání B-hladkých čísel. Řešení matice trvá pouze zlomek celkového času. *Kvadratické síto* optimalizuje právě ono hledání B-hladkých čísel, proto je rozdíl mezi oběma algoritmy co se výkonnosti týče obrovský.

Podívejme se tedy podrobněji na způsob jakým *Kvadratické síto* hledá B-hladká čísla. Základem je interval čísel $[2 \dots Y]$. Nyní aplikujeme-li na tento interval algoritmus Erastothenova síta budou všechny neoznačené položky intervalu prvočísly a jak jsme již řekli v úvodu, všechny označené položky budou čísla složená. Navíc čím vícekrát „označená“ některá položka bude, tím lépe pro nás. Takové číslo má totiž mnoho různých faktorů. V případě že žádný z těchto faktorů není větší než limit B, jedná se o B-hladké číslo. Podstatné pro nás je ovšem to, že faktory jsou v intervalu rozprostřeny rovnoměrně. Konkrétně například faktor 5 obsahuje každé páté číslo v našem intervalu. Z toho plyne, že nemusíme dělit všechna čísla intervalu pětkou při hledání B-hladkých čísel, ale pouze každé páté. Při rostoucích hodnotách limitu B nám tedy sice stoupá počet prvočísel kterými budeme muset dělit, zároveň se ale také snižuje počet nutných dělení pro vyšší prvočísla.

Nás však nezajímají B-hladká čísla v intervalu $[2 \dots X]$. Potřebujeme je najít v sekvenci čísel vypočtených z polynomu $x^2 - N$ kde hodnota x začíná $\sqrt{N}$ a stále roste. To je naštěstí jen malá překážka, přesto ji musíme vyřešit. Abychom mohli efektivně hledat „správná“ čísla v naší polynomiální

sekvenci, musíme nejdříve vyřešit rovnici

$$x^2 - N \equiv 0 \pmod{p} \tag{5.1}$$

Pro $p > 2$ má rovnice buď dvě nebo žádné řešení, pokud p dělí N tak má řešení jen jedno. Samozřejmě tento speciální případ nám okamžitě dává faktor N. Pokud neexistuje žádné řešení, nebudeme prvočíslem prohledávat interval. Řekněme tedy že máme dvě řešení, označme je a_1 a a_2. Potom najdeme násobky prvočísla p v sekvenci čísel pro která platí $x^2 \equiv a_i \pmod{p}$ pro $i = 1, 2$. Výhodou je, že jakmile najdeme první číslo které vyhovuje této rovnici, tak přičítáním p k jeho hodnotě jednoduše získáme další. Vše si ukážeme na následujícím příkladu:

$$N \equiv 9 \pmod{11}$$

Pak $a_1 = 3$, $a_2 = N - a_1 = 8$. Hodnoty kde 11 dělí $x^2 - N$ jsou ty, kde x je rovno 3 nebo 8 modulo 11. Jakmile najdeme první takové číslo x, přičítáním 11 lehce najdeme další x dělitelná jedenácti. Řešení byla dvě, proto máme i dvě různé sekvence ve kterých budeme přičítat 11. Všimněme si ještě jedné věci - ačkoliv hledáme B-hladká čísla, nebereme v potaz všechny prvočíselné faktory menší nebo rovné B. Pro některá prvočísla totiž neexistuje řešení vztahu 5.1 a tak bychom jejich násobky hledali v sekvenci čísel $x^2 - N$ velmi těžko.

Popsali jsme hlavní rozdíl mezi *Dixonovou metodou* a *Kvadratickým sítem*. Se změnou algoritmu jdou ruku v ruce také rozdíly v implementaci, pojďme se tedy podívat v čem spočívají.

5.1 Implementace

Na první rozdíl oproti Dixonově algoritmu narazíme hned na začátku. Při vytváření báze faktorů budeme nuceni vybrat z prvočísel menších než zvolená hranice B taková, pro která existuje řešení vztahu 5.1. Jak ovšem taková čísla poznáme? Odpovědí je výpočet *Legendre symbolu* daného prvočísla. Tento symbol je definován takto:

Definice 5.1.1 (Legendre symbol) *Jestliže je p liché prvočíslo a d je celé číslo, pak Legendre symbol*

$$\left(\frac{d}{p}\right) = \begin{cases} 0 & \textit{jestliže } p \textit{ dělí } d \\ 1 & \textit{jestliže } d \textit{ je čtverec modulo } p \\ -1 & \textit{jestliže } d \textit{ není čtverec modulo } p \end{cases}$$

Nás zajímá právě případ kdy je *Legendre symbol* prvočísla roven 1. Potom totiž existuje celé číslo k takové, že $k^2 \equiv d \pmod{p}$. Když za d dosadíme N, dostaneme přesně vztah 5.1. Takže pro všechna prvočísla menší než B vypočítáme *Legendre symbol* vzhledem k N a do báze faktorů přídáme ta, pro která je tento symbol roven jedné. K výpočtu Legendre symbolu využijeme funkci GMP – `mpz_legendre`. Musíme ale vyřešit další problém. Sice víme, že pro prvočísla z naší báze faktorů existují řešení vztahu 5.1, neznáme ale jejich hodnotu. K určení těchto hodnot budeme potřebovat Shanks-Tonelliho algoritmus. Implementace tohoto algoritmu nebyla v době psaní této práce ještě stálou součástí knihovny GMP, ovšem na stránkách autorů bylo možné si funkci `mpz_sqrtm` dohledat a dodatečně přidat. Této možnosti jsme využili abychom nemuseli zbytečně algoritmus implementovat sami. Funkce vrací výsledek pouze jeden, přestože ve skutečnosti jsou dva, ovšem druhý dopočítáme snadno: Jestliže je d_1 výsledkem funkce `mpz_sqrtm` pak $d_2 = N - d_1$. Nyní je patrné že budeme muset bázi faktorů ukládat trochu jinak než v případě *Dixonova algoritmu*. Pro všechna prvočísla totiž budeme muset uložit hodnoty d_1 a d_2. Proto si vytvoříme strukturu `factBase` – ta bude obsahovat prvočíslo `int p` a obě řešení `int a_1, a_2`. Pro ukládání báze faktorů použijeme pole těchto struktur nazvané `factorBase`.

Nyní máme vytvořenou bázi faktorů, ovšem čeká nás další rozdíl oproti *Dixonově algoritmu*. V jeho případě jsme totiž každé číslo vypočtené z polynomu $y(x) = x^2 - N$ okamžitě testovali, zda není B-hladké. V *Kvadratickém sítu* by ale tento postup byl krajně nelogický. Nevyužili bychom totiž vůbec principu síta. Proto si nejdříve vypočítáme větší množství hodnot polynomu $x^2 - N$ – vytvoříme tak interval na který aplikujeme princip kvadratického síta. Volbu velikosti intervalu zatím odložme na později. Důležité je, že v intervalu budeme potřebovat znát nejen hodnotu $y(x) = x^2 - N$ ale také hodnotu x. Vytvoříme tedy třídu `mpzList`, tentokrát pouze se dvěma položkami – `mpz_t num` pro $x^2 - N$ hodnotu a `mpz_t x` pro x hodnotu. Pole těchto struktur nazveme `list` a bude představovat náš interval hodnot který budeme prosévat. Hodnotu `x` využijeme při hledání *y(x)* dělitelných prvočíslem p, hodnotu `num` budeme prvočísly dělit. B-hladké číslo pak v intervalu najdeme tak, že `num` bude rovno jedné. Potom z `x` vypočítáme znovu num a vytvoříme relaci. V případě, že by nám zvolený interval neposkytl dostatek relací, posuneme se v polynomilální sekvenci dále vutvořením nového intervalu od čísla, kde jsme v předchozím intervalu skončili a provedeme celý proces prosévání znovu. Tento postup opakujeme až do okamžiku kdy máme dostatek relací.

Následně vytvoříme matici z nalezených relací, kterou vyřešíme stejně jako v případě *Dixonova algoritmu*. Postup je zde již stejný a tak se detailům věnovat nebudeme. Podrobnější informace o implementaci může čtenář nalézt

v dokumentaci k programu a v komentářích v samotném kódu.

5.2 Vlastnosti algoritmu

Kvadratické síto má téměř stejné vlastnosti jako *Dixonův algoritmus*. Hlavní rozdíl zde spočívá v tom, že prohledáváme najednou velký interval čísel a nehledáme B-hladká čísla prověřováním jednoho čísla po druhém. Velikost intervalu by však mohla být velmi podstatná. Malý interval pravděpodobně nevyužije potenciálu prosévání, protože se více zdržíme při hledání první vhodné hodnoty a tak výhody plynoucí ze snadného hledání dalších vhodných hodnot (čísel dělitelných prvočíslem p) budou menší. Na druhou stranu, příliš velký interval bude zabírat více paměti a zřejmě bude vyžadovat více přesunů dat v počítači a tedy nová zdržení při výpočtech. Je důležité si uvědomit, že interval procházíme pro každé prvočíslo od začátku až do konce, tedy pro nás není dobré, pokud je interval v paměti „rozkouskován". Při měření se podíváme na to, jaký vliv má velikost intervalu na výkonnost algoritmu – v literatuře je totiž tato věc poněkud opomíjena. Pokusíme se tedy zjistit co má vliv na optimální velikost intervalu a jak tuto optimální velikost zjistit.

Stejně jako v případě *Dixonova algoritmu*, i zde bude klíčová volba hodnoty B, respektive velikosti báze faktorů. Prověříme, zda je doporučovaná hodnota pro *Dixonův algoritmus* vhodná i pro *Kvadratické síto* a případně provedeme její korekci.

5.3 Výsledky měření

Pro testování byla využita čísla v rozmezí od 20 až do 30 míst (vyšší hodnoty jen v některých případech). Všechna čísla byla vytvořena z prvočísel p a q o stejném nebo maximálně o jedna se lišícím počtu cifer. Opět pro testování nebyla využita prvočísla – důvod je stejný jako v případě Dixonova algoritmu, totiž že v praxi se nejdříve využije testu prvočíselnosti, než se přejde k samotné faktorizaci. Faktorizace probíhala na počítači s procesorem AMD Turion 64 1800 Mhz s 1 GB DDR 166 Mhz pamětí RAM. V následující tabulce jsou výsledky faktorizace pro čísla 20 až 30 míst, kde limit B byl vypočten na základě vztahu 4.4 (hodnota vhodná pro *Dixonův algoritmus*) a pro všechna čísla byl zvolen stejně velký interval - 1500 čísel:

Číslo N	Báze faktorů	limit B	Čas [ms]
58968037447783324577	63	900	23774
581587150621066686563	88	1100	502578
6432429779944953129001	111	1354	11889
76714095317141075716423	129	1672	43424
716805582589122914288827	136	2015	784156
4697267292063632778022211	173	2351	705534
74798688525150394570524403	223	2944	425827
481987791745464868524615121	243	3417	63109
770213198651657898444169115 9	284	4248	110984
64209081550358711183908252057	341	5003	196093
625756966826201801926451911009	398	5949	173822

Tabulka 5.1: Kvadratické síto - měření

Nejříve porovnejme výkon *Kvadratického síta* s výkonem *Dixonova algoritmu* z hlediska velikosti báze faktorů. Při pohledu na tabulku 4.2 vidíme, že velikost báze faktorů je daleko menší. Pro 15ti místné číslo jsme potřebovali v případě *Dixonova algoritmu* 242 prvočísel a faktorizace trvala 97 sekund. U *Kvadratického síta* stejně velká báze faktorů postačila k faktorizaci 27 místného čísla a to ještě v kratším čase (63 sekund). Nárůst výkonu je tedy jasně patrný.

5.3.1 Báze faktorů

Při prohlížení tabulky 5.1 si ale všimněme jednoho důležitého faktu. Doba faktorizace rozhodně neroste pravidelně tak jako roste velikost čísla N. Menší odchylky jsou samozřejmě pochopitelné, protože nejen počet cifer ale i samotná velikost čísla určuje pravděpodobnost na nalezení B-hladkého čísla (jiné je to pro čísla $3 \cdot 10^{25}$ než pro čísla $7 \cdot 10^{25}$ přestože mají stejný počet cifer). Ovšem odchylky jsou značné a způsobuje je něco jiného. Všimněme si jak se mění velikost limitu B a velikost báze faktorů. Například limit B se nám oproti hodnotě pro 23 místné číslo zvětšil u 24 místného o 20 procent, zatímco počet čísel v bázi faktorů vzrostl pouze o 5 procent. To nám poukazuje na další rodíl mezi *Dixonovým algoritmem* a *Kvadratickým sítem* – u prvního jmenovaného nám z limitu B plyne jednoznačně i velikost báze faktorů, ovšem u druhého tomu tak není. Navíc u *Kvadratického síta* se může velikost báze faktorů lišit i při stejném limitu B pro čísla se shodným počtem cifer – stačí, aby byla jen nepatrně různá velikostně. Za tento fakt může samozřejmě volba čísel které zařadíme do báze faktorů. Podle výsledků v předchozí tabulce to vypadá, že je báze faktorů v některých případech až příliš malá. Podívejme se co se stane, zvýšíme-li limit B o 50 procent:

Číslo N	Báze faktorů	limit B	Čas [ms]
58968037447783324577	95	1350	6559
581587150621066686563	125	1650	109342
6432429779944953129001	160	2031	10185
76714095317141075716423	192	2508	17825
716805582589122914288827	195	3022	251337
4697267292063632778022211	246	3526	293825
747986885251503945705244 03	308	4416	330368
481987791745464868524615121	353	5125	41059
770213198651657898444169 1159	409	6372	68909
642090815503587111839082 52057	476	7504	107577
625756966826201801926451911009	575	8917	106673

Tabulka 5.2: Kvadratické síto - měření 2

Výkon nyní vzrostl opravdu značně, většinou o zhruba 40 procent, v některých případech dokonce ještě mnohonásobně více. Všimněme si ale opět některých zajímavých rozdílů – pro 23 místné číslo máme 192 prvočísel v bázi faktorů, ale pro 24 místné číslo jich je pouze o 3 více! Přitom limit B vzrostl o 20 procent, kdežto velikost báze faktorů o pouhých 1,5 procenta. To vše naznačuje, že při vytváření báze faktorů pro *Kvadratické síto* nemůžeme vycházet pouze z limitu B, ale také z toho, kolik prvočísel nám pro bázi faktorů tento limit dá. Jak ukázal předchozí příklad, nepomůže nám ani navýšení limitu B, protože báze faktorů se nezvětší ve stejném (a někdy ani přibližném) poměru. Proto velikost báze faktorů určíme přímo a podle toho budeme načítat prvočísla dokud jich nebudeme mít požadovaný počet. Než však navrhneme způsob jakým volit velikost báze faktorů, musíme najít alespoň pro jedno číslo nejoptimálnější hodnotu, abychom měli z čeho vyjít. Následující tabulka by nám mohla lecos napovědět. Faktorizováno bylo jedno dvacetimístné číslo s různě velkými bázemi faktorů:

Báze faktorů	136	141	147	152	162	171	178	183	189
Čas [ms]	7507	7484	6973	5401	2315	2102	2900	4252	3831

Tabulka 5.3: Kvadratické síto - velikost báze faktorů

Nejlepšího výsledku dosáhl algoritmus při faktorizaci se 171 a 178 prvočísly v bázi faktorů. Vzpomeňme si ještě na hodnotu limitu B pro takto velké číslo – 900. Zde byl použit limit 2400-2500. Když ovšem spočítáme, kolik prvočísel je menších než 900, zjistíme, že je jich přesně 154, což je téměř shodná hodnota pro jakou měl algoritmus nejlepší výsledky. Mohli bychom tedy volbu velikosti báze faktorů udělat následujícím způsobem: spočítáme kolik prvočísel je menších než limit B vhodný pro dané číslo a načteme jich přesně tolik.

Předtím než tento postup otestujeme, zastavíme se ještě u jedné důležité věci. Při testování velikosti báze faktorů a jejího vlivu na rychlost faktorizace v příkladu výše jsme se nezastavili u hodnoty 189. Ve skutečnosti by měla být tabulka mnohem větší, testovali jsme totiž až hodnotu 331. Konkrétně pro tento případ trvala faktorizace jen něco málo přes 1100 milisekund, což je podstatně méně než kterýkoliv údaj v tabulce. Důvod proč dle této hodnoty nebudeme odvozovat optimální velikost báze faktorů je prostě ten, že by nás tato hodnota vedla k příliš velkým číslům. Pro dvacetimístné číslo nám to vadit nemusí, matice o rozměrech 331 krát 332 není žádný problém, ovšem pro podstatně větší čísla by už byl rozdíl neúnosný a námi odvozený „vzorec“ by byl zcela nepoužitelný. Následující tabulka ukazuje výsledky faktorizace při volbě velikosti báze faktorů dle postupu navrženého v tomto odstavci:

Číslo N	Báze fakt.	Max. prvočíslo	Čas [ms]
589680374477833245 77	154	2207	2543
581587150621066686563	184	2659	23695
6432429779944953129001	217	2857	6200
76714095317141075716423	260	3499	12405
716805582589122914288827	304	4903	116048
469726729206363277802221 1	349	5297	126691
747986885251503945705244 03	425	6211	142667
481987791745464868524615121	481	7237	28553
770213198651657898444169115 9	582	9419	47154
642090815503587111839082520 57	670	11243	75763
625756966826201801926451911009	781	12959	81551

Tabulka 5.4: Kvadratické síto - velikost báze faktorů

Nárůst výkonu je opět velmi patrný oproti předchozímu měření, přičemž velikost báze faktorů jsme drželi v „rozumných“ mezích. Vypadá to, že námi zvolená metoda volby velikosti báze faktorů přináší dobré výsledky. Podívejme se ještě jak se naše volba projeví při faktorizaci 37 místného čísla:

Báze faktorů	Max. prvočíslo	Čas [ms]
952	15991	1444298
1862	33647	716052

Tabulka 5.5: Kvadratické síto - ověření metody

I v tomto případě vzrostl výkon podle očekávání, můžeme být tedy s výsledkem testů spokojeni. V následující kapitole se zaměříme na volbu intervalu pro prosévání – pokusíme se zjistit jaký vliv má velikost tohoto intervalu na rychlost faktorizace.

5.3.2 Interval prosévání

Volba velikosti intervalu nebude mít s největší pravděpodobností tak velký vliv na výkon algoritmu, ovšem přesto může velmi špatná hodnota značně zpomalit výpočet. Proto bychom měli znát alespoň krajní hodnoty, kterých bychom se měli vyvarovat. Následující tabulka ukazuje dobu faktorizace 28 místného čísla v závislosti na velikosti intervalu prosévání. Báze faktorů byla ve všech případech stejná:

Interval	5000	10000	50000	100000	300000	400000	500000	700000
Čas [ms]	50087	45645	46870	53881	68451	68632	72539	86896

Tabulka 5.6: Kvadratické síto - interval prosévání

Vidíme že při zvyšování intervalu nad 100 000 dochází k pozvolnému prodlužování doby faktorizace, nejlepšího výsledku dosáhly hodnoty 10 a 50 tisíc. Velký interval zabírá extrémní množství paměti a také zde může dojít k situaci, že například již v polovině intervalu máme dostatek relací – potom „kontrolujeme“ velké množství čísel zbytečně. Ještě se zastavme u krajních hodnot velikosti intervalu:

Interval	25	50	75	100	200	3000000
Čas [ms]	59776	58793	60356	57382	56573	130983

Tabulka 5.7: Kvadratické síto - interval prosévání

Je vidět že volba příliš malého intervalu je mnohem lepší variantou, než interval příliš velký. Při našich testech jsme většinou volili interval okolo hodnoty 1500 což byla relativně dobrá volba, přestože o něco větší interval by byl optimálnější. V zásadě můžeme říct že by měl interval obsahovat alespoň tolik čísel, kolik je nejvyšší prvočíslo v bázi faktorů. Samozřejmě toto pravidlo přestává platit jakmile hodnota tohoto prvočísla roste řádově nad stovky tisíc, potom je totiž interval příliš velký a zabírá obrovské množství paměti. Přesné hodnoty se budou samozřejmě lišit od hardwarového vybavení počítače - zejména od velikosti a rychlosti pamětí RAM a cache procesoru.

5.4 Složitost

Při určování složitosti *Kvadratického síta* vyjdeme z výsledků získáných v kapitole o *Dixonově algoritmu*. Složitost *Kvadratického síta* odvodíme porovnáním rozdílu mezi oběma algoritmy který se skrývá v hledání B-hladkých čísel. Ostatní kroky algoritmu (tedy řešení matice, počítání čísel A a B z

jejích řešení a následně faktorů N) zůstaly totiž beze změny. Oproti aplikování postupného dělení (dělíme prvočísly z báze faktorů) na každé číslo v případě *Dixonova algoritmu* uplatňujeme v případě Kvadratického síta princip „prosévání“ – čas strávený na jednom čísle se dramaticky zmenší. Díky tomuto zlepšení bude potřeba jen asi $\ln \ln B$ kroků na jedno číslo. V porovnání se zhruba B kroky v případě postupného dělení je to obrovské zrychlení. S odkazem na [2] a [7] uvádíme výslednou složitost:

$$e^{\sqrt{\ln N \cdot \ln \ln N}} \tag{5.2}$$

V porovnání s *Dixonovým algoritmem* zde nemáme násobek dvojky v exponentu. Na první pohled se to nezdá jako dramatické zlepšení, ale společně s dalšími dobrými vlastnostmi algoritmu znamená tato složitost možnost faktorizovat daleko větší čísla než to bylo možné s *Dixonovým algoritmem.* Je totiž nutné si uvědomit, že jsme vylepšili nejpodstatnější část algoritmu – hledání B-hladkých čísel.

5.5 Optimalizace

Pro *Kvadratické síto* můžeme využít stejných postupů optimalizace jako pro *Dixonův algoritmus* - paralelizace na více řešitelů i preprocessing matice se zde uplatní stejně dobře, ne-li lépe. Ovšem nám se naskýtají ještě další možnosti, jak vylepšit tento algoritmus. Mějme B-hladké číslo h a číslo $c_1 = h \cdot p$ kde p je nějaké číslo větší než limit B. Pokud je p menší než B^2, pak p musí být prvočíslo (vzpomeňme na algoritmus postupného dělení). Nyní pokud najdeme další číslo stejných vlastností jako má c_1 (nazveme jej c_2), pak můžeme z těchto dvou čísel vytvořit B-hladké číslo. Na příkladu si ukážeme jak na to. Faktorizujeme číslo $N = 163$ s bází faktorů $\{2, 3, 5, 7\}$ a limitem $B = 8$, první dvě čísla mající vlastnost čísel c_1 a c_2 jsou:

$$\begin{aligned} 15^2 &\equiv 62 \equiv 2 \cdot 31 \pmod{163} \\ 16^2 &\equiv 93 \equiv 3 \cdot 31 \pmod{163} \end{aligned}$$

Číslo $p = 31$ splňuje obě podmínky: $B < p < B^2$. Nyní obě čísla vynásobíme mezi sebou:

$$(15 \cdot 16)^2 \equiv 2 \cdot 3 \cdot 31^2 \pmod{163}$$

a obě strany rovnice vynásobíme $(31^{-1})^2 \pmod{163}$. 31^{-1} (modulo 163) je 142 takže:

$$\begin{aligned} (15 \cdot 16 \cdot 142)^2 &\equiv 2 \cdot 3 \pmod{163} \\ 13^2 &\equiv 2 \cdot 3 \pmod{163} \end{aligned}$$

Což nám dává úplnou relaci. Takto vzniklým relacím říkáme „cykly“. Samozřejmě tak jak stoupá velikost N a s tím i B, zvyšuje se počet možných prvočísel, která nám mohou pomoci vytvořit cykly a tím další relace. Zobecněním uvedeného postupu dostaneme:

$$c_1 \equiv t^2 \equiv u \cdot p \pmod{N}$$
$$c_2 \equiv r^2 \equiv v \cdot p \pmod{N}$$
$$\left(\frac{t \cdot r}{p}\right)^2 \equiv u \cdot v \pmod{N} \tag{5.3}$$

Kde r, t, u, v jsou celá čísla a p je prvočíslo z intervalu $[B \dots B^2]$. Na druhou stranu je také možné, že nám cyklus poskytne relaci kterou už jsme nalezli předtím. V tomto případě by tomu tak nejspíš bylo, protože 13 je první celé číslo větší než $\sqrt{163}$ a tedy tady by začínala naše polynomiální sekvence. Ovšem na druhou stranu, jev „jednoho velkého prvočísla“ je poměrně častý a tak cykly velkou měrou přispívají k zisku dostatečného množství B-hladkých čísel. Při testech pomocí programu *msieve* (viz [8]) vyplynulo, že za pomocí cyklů je možné získát až 50% relací. Samozřejmě že tvorba relací vyžaduje prostředky navíc. Částečné relace musíme ukládat a následně kombinovat dohromady ty, které obsahují stejné „velké prvočíslo“ – ovšem obrovské procento relací které je takto mnohdy získáno jednoznačně mluví pro tuto metodu. Při rostoucích hodnotách limitu B ale není možné brát v úvahu všechna prvočísla menší než B^2, proto se často volí takzvaný limit „velkého prvočísla“. Ten je typicky roven $64 \cdot B$.

Při testech jsme si všimli, že přírůstek relací za jednotku času, respektive za určitý počet prohledaných intervalů, nebyl pořád stejný. Naopak, jak prosévání pokračovalo a interval se posouval dále od $\sqrt{N}$, počet nalezených relaxí klesal. Tento jev není pouze náhodný, je tomu tak vždy. Důvod je ten, že jak rostou čísla která kontrolujeme zda nejsou B-hladká, klesá šance že tuto vlastnost budou mít. Všimněme si ale jedné věci. Mějme číslo $N = 34642897$. Spočítáme první dva členy naší polynomiální sekvence dle $(\sqrt{N} + x)^2 - N$:

$$x_0 = (5886 + 0)^2 - 34643897 = 2099$$
$$x_1 = (5886 + 1)^2 - 34643897 = 13872$$

Z těchto konkrétních hodnot vidíme hned dvě věci. Ta první, o které jsme se zminili již dříve, je fakt, že čísla v polynomiální sekvenci rostou velmi rychle. Ale je tu jedna mnohem důležitější věc, která s tou první úzce souvisí. Pokud se podíváme na interval mezi oběma čísly, je jasné že mnoho z nich bude B-hladkých (ku příkladu 4096 je 2^{12}). Existuje nějaký způsob, jak bychom se mohli k těmto „vynechaným“ číslům dostat? Samozřejmě - pomocí více

polynomů. Využijeme polynomů ve tvaru $g_{a,b}(x) = (a \cdot x + b)^2 - N$ kde $b^2 - N = a \cdot c$, tedy $b^2 - N$ je násobkem a. Potom můžeme psát $g_{a,b}(x) = a \cdot (a \cdot x^2 + 2 \cdot b \cdot x + c)$ a pokud je a čtverec, stačí nám brát v úvahu $(a \cdot x^2 + 2 \cdot b \cdot x + c)$. Využitím více polynomů se můžeme vyvarovat postupného klesání nalezených relací tím, že jakmile jedna polynomiální sekvence přestane poskytovat dostatečné množství relací, přepneme na jiný polynom. Navíc je tento způsob ideální pro paralelní zpracování - každý řešitel dostane vlastní polynom (nebo i více) a nezávisle na ostatních hledá B-hladká čísla.

Při rostoucích hodnotách faktorizovaných čísel nám také roste matice kterou budeme muset na konci řešit. Vzniká tak nejen problém jak takovouto obrovskou matici ukládat, ale také jak ji řešit konvenčními algoritmy. Řešením je využít řídkosti matice – převážná většina položek je totiž nulových a tak je vhodné využít formátu jenž ukládá matici na základě jejich jedničkových položek – v podstatě se jedná o určitou formu komprese. Nevýhodou je, že tento „komprimovaný“ formát je nevhodný pro většinu algoritmů které s maticí pracují (v našem případě *Gaussova eliminace*). Ovšem pro preprocessing matice (odstraňování singletonů a podobně) je tento formát v pořádku. Proto můžeme matici uložit komprimovaně, provést preprocessing a následně ji vyřešit ve standartním formátu, případně upravit algoritmus tak aby mohl pracovat i s komprimovaným formátem. Pro opravdu velké matice se ale *Gaussova eliminace* nehodí a tak se využívá jiných algoritmů, například iterativního *Lanczosova algoritmu*.

Kapitola 6

Závěr

Dokončili jsme rozbor i implementaci všech čtyř zvolených algoritmů a tak nyní nastal ten správný čas pro zhodnocení výsledků naší práce. Už v průběhu prací na programu se ukázalo, že byl poněkud podceněn rozsah problematiky. Tato domněnka pak byla potvrzena při psaní tohoto dokumentu. Zejména podobnost *Dixonova algoritmu* a *Kvadratického síta* znamenala čas navíc při analýze a návrhu optimalizací, protože oba algoritmy pracují na téměř stejném principu. Mnohem lepší možností by bylo využít *Fermatovy metody* a z ní vycházejícího *Dixonova algoritmu* pouze jako prostředek sloužící k lepšímu pochopení funkčnosti *Kvadratického síta* a implementačně řešit pouze tento algoritmus. Důvodem je i fakt, že *Kvadratické síto* je v praxi používáno, narozdíl od předchozích dvou zmiňovaných. Výběr algoritmů ovšem nebyl náhodný a sledoval praktický cíl: Jako první jsme zvolili *Postupné dělení*, protože je to algoritmus který přirozeně používáme i v běžném životě aniž si to uvědomujeme (samozřejmě pro triviální úlohy – malá čísla). Je také nedílnou součástí pokročilejších algoritmů, kde dílčí výpočty provádí právě *Postupné dělení*. *Fermatovu metodu* jsme vybrali proto, že představuje úplně jiný přístup k faktorizaci a je základní myšlenkou mnoha dalších algoritmů. Analýza principu toho algoritmu nám pomohla důkladněji pochopit složitější algoritmy vycházející s *Fermatovy metody*. *Dixonova metoda* pro nás představovala posun principu představeného ve *Fermatově algoritmu* opět o krok dál a přinesla i problémy, se kterými jsme se v předchozích dvou případech nesetkali – volba parametrů zde hraje klíčovou roli pro výkonnost algoritmu. Nakonec jsme se zabývali *Kvadratickým sítem*, algoritmem který je dodnes nejrychlejší pro čísla o dvaceti až stodvaceti cifrách.

U všech algoritmů jsme nejdříve uvedli princip na kterém pracují a následně se zabývali některými implementačními detaily. Poté jsme algoritmus analyzovali a odvodili jeho vlastnosti pro různé vstupní hodnoty a parametry. Následně jsme na četných měřeních teoretické výsledky ověřili a získané

znalosti využili k odvození složitosti algoritmu (také za pomoci literatury). Veškerá práce nakonec vyústila v návrh různých vylepšení metody či implementace.

Za stěžejní část práce považujeme právě kapitolu věnovanou *Kvadratickému sítu*. I pro ostatní algoritmy se nám podařilo navrhnout vylepšení, která mohou pomoci zlepšit výkonnost (zejména systematické prohledávání intervalu ve *Fermatově metodě*), nejvíc jsme se ale zaměřili na *Kvadratické síto*. Věnovali jsme se hlavně tématům, kterým není v literatuře věnováno tolik prostoru, či jsou dokonce opomíjeny celkově. Tím je hlavně volba velikosti báze faktorů a intervalu prosévání. Zjistili jsme, že volit bázi faktorů jen na základě limitu B pro hladká čísla není ideální způsob a tak jsme metodu poněkud upravili a její funkčnost ověřili měřeními. Výsledky byly uspokojivé a pro námi testované hodnoty vykazoval algoritmus skutečně lepších výsledků než pro základní nastavení odvozené z *Dixonova algoritmu*. Bohužel rozsah práce a časová náročnost měření nám nedovolila ověřit metodu i pro vyšší hodnoty N – doporučujeme tedy tuto metodu podrobit dalšímu zkoumání, všeobecně je totiž známo, že metody volby báze faktorů platné pro čísla nižších řádů vykazují horší výsledky pro čísla řádů vyšších. Nutnost volit bázi faktorů ideálně je velmi podstatná pro *Kvadratické síto* – pro nižší čísla se dá kvalita metody ověřit měřením a v současné době se doporučované velikosti bází faktorů opírají spíše o praktická měření než o teoretické výsledky. Ovšem pro opravdu velká N je shromažďování výsledků četných měření příliš časově i výpočetně náročné a tak je nutné mít k dispozici metodu, která co nejpřesněji určí vhodnou velikost báze faktorů.

Volba velikosti intervalu prosévání nemá dle měření na výkonnost až tak signifikantní vliv, ovšem přesto je možné dobrou volbou zlepšit chování algoritmu. Podařilo se nám stanovit meze pro velikost intervalu a doporučit nejlepší hodnotu pro námi testovaný rozsah čísel. Jsme si ale vědomi, že na správnou volbu intervalu má velký vliv také použitý hardware – bylo by vhodné dále prozkoumat vliv rychlosti a velikosti interních pamětí počítače (hlavně cache procesoru) na tento parametr algoritmu, stejně jako prověřit chování pro větší čísla N.

S ohledem na rozsah práce si myslím, že můžeme být s výsledkem spokojeni. Podařilo se nám zaměřit se na oblasti problému, které nejsou prozkoumány dostatečně a navrhli jsme určitá zlepšení volby jednotlivých parametrů. Některé z navržených metod vyžadují hlubší prozkoumání a větší množství testů a tak jsme je doporučili pro další výzkum. Doufám, že tento text poslouží jako dobrý podklad pro pokračování práce a ulehčí práci kolegům, kteří se tímto tématem budou v budoucnu zabývat.

Literatura

[1] Velebil, J.: *Diskrétní matematika a logika*, ČVUT FEL, katedra matematiky, Praha, 2006

[2] Pomerance, C.: *Smooth numbers and the quadratic sieve*

[3] GNU LGPL licence: *http://www.gnu.org/copyleft/lesser.html*

[4] GNU Multiprecision Library: *http://gmplib.org/*

[5] Kiming, I.: *The ψ-function and the complexity of Dixon's factoring algorithm*

[6] Olšák, P.: *Lineární algebra*, ČVUT FEL, katedra matematiky, Praha, 2006

[7] Pomerance, C.: *A tale of two sieves*

[8] Papadopoulos J.: *Msieve*, http://www.boo.net/ jasonp/qs.html

[9] Landquist E.: *The quadratic sieve factoring algorithm*, Math 488: Cryptographic algorithms, 2001

Příloha A

Uživatelská příručka

Program je určený pro operační systém Windows XP (32 bit), měl by ale pracovat i na starších verzích (2000/ME). Vyžaduje alespoň 10 MB prostoru na disku a 128 MB RAM. Dále ke své funkci potřebuje knihovnu GMP (viz literatura a obsah přiloženého CD). Mělo by být možné program přeložit a používat i na Unixových OS, ovšem toto nebylo testováno. Není nutné program nijak instalovat, spustitelný soubor je plně funkční. Program nedisponuje žádným uživatelským rozhraním (GUI) a je ovládán čistě přes vstupní dávkový soubor. Výstup programu je prováděn na standardní výstup, ale doporučuje se přesměrovávat jej do souboru (viz dále). Ke své práci program potřebuje tabulku prvočísel dostatečné velikosti uloženou v souboru. Čísla musí být seřazena podle velikosti (vzestupně) a musí být oddělena mezerou nebo znakem nového řádku. Příklad vhodného souboru s tabulkou prvočísel je možné nalézt na přiloženém CD (viz následující příloha B, soubor "pt.dat").

Program vyžaduje pro korektní funkci vždy dva parametry: první je cesta k dávkovému souboru s příkazy, druhý je cesta k souboru s tabulkou prvočísel. Spuštění programu tedy může vypadat například takto: `"faktorGMP qs30.in pt.dat"`, kde `qs30.in` je soubor s příkazy pro faktorizaci a `pt.dat` je soubor s tabulkou prvočísel. Uvedené pořadí souborů je závazné, nelze tedy uvést nejdříve soubor s prvočísly a poté až dávkový soubor. Jména souborů smí být libovolná.

A.1 Struktura příkazového souboru

Soubor by měl obsahovat alespoň jeden příkaz. Každý z příkazů v souboru musí začínat číslem které specifikuje algoritmus a z toho plynoucí počet parametrů které bude nutné načíst. Po načtení všech parametrů je provedena faktorizace a až po jejím skončení se začne provádět další příkaz. Pokud po

faktorizaci čísla není načten správný argument, je program ukončen. Přípustné argumenty (načtené hodnoty) jsou jen dvě: Konec souboru nebo číslo z množiny {1, 3, 4, 5}. Zde je detailní popis povolených příkazů:

- **Postupné dělení** má číslo `1`. Algoritmus má dva parametry, první je číslo N které chceme faktorizovat a druhý je počet prvočísel které chceme načíst. Například pro faktorizaci čísla `125` s `20` prvočísly pomocí postupného dělení bychom zadali: `1 125 20`
- **Fermatova metoda** má číslo `3`. Algoritmus má pouze jeden parametr – číslo N které chceme faktorizovat. Například pro faktorizaci čísla `125` pomocí Fermatovy metody bychom zadali: `2 125`
- **Dixonova metoda** má číslo `4`. Algoritmus má dva parametry, první je číslo N které chceme faktorizovat a druhý je limit B pro hladká čísla. Například pro faktorizaci čísla `125` s použitím všech prvočísel menších než `20` pomocí Dixonovy metody bychom zadali: `4 125 20`
- **Kvadratické síto** má číslo `5`. Algoritmus má tři parametry – první je číslo N které chceme faktorizovat, druhý je počet prvočísel pro bázi faktorů a třetí velikost intervalu prosévání. Například pro faktorizaci čísla `125` s použitím `4` prvočísel a s intervalem o `100` číslech pomocí Kvadratického síta bychom zadali: `5 125 4 100`

Doporučuje se umisťovat jednotlivé příkazy na řádky pro větší přehlednost souboru. Ten může vypadat například takto:

```
1  58968037447783324577  1000
3  6373012344022668763
4  5071030764490387      1979
5  58968037447783324577  900   1500
```

Tabulka A.1: Příklad dávkového souboru

Protože je výstup programu prováděn vždy na standartní výstup, doporučujeme jej přesměrovat do souboru. Toho dosáhneme spuštěním programu s příkazem pro přesměrování a cestou k souboru, například: `"faktorGMP qs30.in pt.dat > qs30.out"`. Soubor bude pravidelně aktualizován – vždy po dokončení faktorizace jednoho čísla (tedy po dokončení příkazu) nebo po naplnění bufferu programu.

Příloha B

Obsah přiloženého CD

Složka `data` obsahuje ukázková měření, tabulky prvočísel a zdrojový kód tohoto textu ve formátu TeX. Ve složce `exe` je umístěn přeložený program spolu s připravenou tabulkou prvočísel. Dokumentace je uložena ve složce `html` – generováno programem *Doxygen*. Veškerá literatura použitá při vytváření práce byla umístěna do složky `literatura` (ve formátu PDF). Zdrojový kód programu se nachází ve složce `src`. Text této práce v PDF je ve složce `text`. V souboru readme.txt umístěném v kořenovém adresáři je možné nalézt další informace o obsahu jednotlivých složek.