

Univerzita Karlova v Praze  
Matematicko-fyzikální fakulta

## BAKALÁŘSKÁ PRÁCE



Jan Vyrubalík

### Výpočetní složitost základních algoritmů počítačové algebry

Katedra algebry

Vedoucí bakalářské práce: RNDr. David Stanovský, Ph.D.  
Studijní program: Matematické metody informační bezpečnosti

2009

Na tomto místě bych rád poděkoval RNDr. Davidu Stanovskému, Ph.D za vstřícnost při úpravě tématu práce a pomoc při psaní této práce.

Prohlašuji, že jsem svou bakalářskou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne

Jan Vyrubalík

# Obsah

<b>1</b>	<b>Úvod</b>	<b>5</b>
<b>2</b>	<b>Stručný úvod do NTL</b>	<b>6</b>
2.1	Vstup, výstup a měření času . . . . .	8
2.2	Přehled jednotlivých tříd . . . . .	9
2.2.1	ZZ . . . . .	9
2.2.2	ZZ_p . . . . .	11
2.2.3	ZZX . . . . .	14
2.2.4	ZZ_pX . . . . .	16
<b>3</b>	<b>Největší společný dělitel</b>	<b>19</b>
3.1	Největší společný dělitel v oboru celých čísel . . . . .	19
3.2	Největší společný dělitel polynomů nad konečným tělesem . . . . .	22
3.3	Největší společný dělitel polynomů nad $\mathbb{Z}$ . . . . .	25
<b>4</b>	<b>Faktorizace polynomů</b>	<b>27</b>
4.1	Nad celými čísly . . . . .	27
4.1.1	Zassenhausova metoda . . . . .	28
4.1.2	Metoda van Hoeij . . . . .	28
4.2	Nad konečným tělesem . . . . .	31
4.2.1	Cantor-Zassenhaus . . . . .	31
4.2.2	Berlekamp . . . . .	33
	<b>Literatura</b>	<b>36</b>

Název práce: Výpočetní složitost základních algoritmů počítačové algebry

Autor: Jan Vyrubalík

Katedra: Katedra algebry MFF UK

Vedoucí bakalářské práce: RNDr. David Stanovský, Ph.D.

e-mail vedoucího: david.stanovsky@mff.cuni.cz

Abstrakt: Předložená práce představuje rychlý úvod do používání knihovny NTL. Obsah je zvolen s ohledem na přednášku Počítačová algebra I, tedy práce s celými čísly, konečnými tělesy a jejich polynomiálními rozšířeními. Druhou částí práce je pak přehled algoritmů implementovaných v této knihovně pro výpočet největšího společného dělitele a faktorizaci polynomů doplněný o grafy časové náročnosti těchto implementací pro náhodné vstupy. Algoritmy jsou uváděny bez důkazu s odkazem na doplňující literaturu. Průvodce po NTL je koncipován pro každého začínajícího uživatele s alespoň základní znalostí programování, pro pochopení popsaných algoritmů však doporučuji absolvovat přednášku Počítačová algebra I.

Klíčová slova: knihovna NTL, největší společný dělitel polynomů, faktorizace polynomů

Title: Computational complexity of basic computer algebra algorithms

Author: Jan Vyrubalík

Department: The Department of Algebra, MFF UK

Supervisor: RNDr. David Stanovský, Ph.D.

Supervisor's e-mail address: david.stanovsky@mff.cuni.cz

Abstract: The following work presents fast introduction in usage of the NTL library. The area covers work with integers, finite fields and polynomials over these. Second part of the work is the overview of algorithms implemented in NTL for greatest common divisor and factorization computations including graphs showing the running time of these implementations on random inputs. Algorithms are introduced without proofs with reference to literature. The NTL guidebook is designed for any beginners with basic programming knowledge.

Keywords: NTL library, greatest common divisor of polynomials, polynomial factorization

# Kapitola 1

## Úvod

Práce je koncipována jako doplňkový materiál k přednášce Počítačová algebra I. Součástí předmětu je i implementace algoritmů v jazycích C/C++ a použití knihovny NTL Victora Shoupa představuje značné ulehčení vzhledem k množství implementovaných tříd a funkcí pro výpočty z celého rozsahu přednášky. Přestože ke knihovně je dostupná i dokumentace je pro začínající uživatele možná až příliš obsáhlá a technická. A přesto postrádá popis některých užitečných funkcí, které si pak uživatel zbytečně píše sám, nebo je třeba je hledat v rozsáhlém zdrojovém kódu. Druhá kapitola práce je tedy zamýšlena jako průvodce po NTL nejen pro studenty Počítačové algebry, ale i pro každého, kdo chce v co nejkratším čase používat základní funkce například pro polynomiální aritmetiku nad celými čísly. Přestože snaha byla, aby práci mohl použít každý se znalostmi z povinných přednášek Programování I a II, a pochopení skrz příklady není nemožné, alespoň základní znalost syntaxe jazyků C/C++ je určitě výhodou.

Třetí a čtvrtá kapitola pak obsahují stručný popis algoritmů implementovaných v NTL k výpočtu největšího společného dělitele a faktorizaci polynomů. Představují tak rozšiřující materiál právě pro studenty Počítačové algebry. Popisy jsou doplněny grafy výpočetního času jednotlivých implementací. Pro tyto výpočty byl použit počítač s procesorem Intel (2,4 GHz) s 32-bitovou verzí operačního systému Windows.

# Kapitola 2

## Stručný úvod do NTL

NTL je C++ knihovna poskytující datové struktury a algoritmy pro práci s libovolně velkými čísly, vektory, maticemi a polynomy nad celými čísly a konečnými tělesy. V této práci jsou použity a popisovány převážně části věnující se polynomiální aritmetice, konkrétně algoritmy na výpočet NSD a faktorizace nad konečnými tělesy a celými čísly. V práci je popsána knihovna ve verzi 5.5. Knihovnu je možno stáhnout na stránkách V.Shoupa <http://www.shoup.net/ntl/>, kde je také odkaz na návod (v angličtině) ke kompilaci. Pro uživatele Unixových strojů je to otázkou 2 příkazů, uživatelům Windows je doporučeno ručně upravit konfigurační soubor `config.h` podle pokynů, následný návod popisuje kompilační proceduru krok za krokem. Přestože pro potřeby práce byla používána samotná knihovna NTL, doporučuji použití s knihovnou GMP (<http://gmplib.org/>). Tato knihovna obsahuje implementace nejrychlejších algoritmů pro aritmetiku celých, reálných a racionálních čísel. Tyto funkce používá NTL interně, tudíž volání samotných NTL funkcí se nijak nezmění.

Od této kapitoly budeme **single-precision** čísla chápat jako taková, jejichž binární reprezentace se vejde do slova, které je použitý procesor schopen najednou zpracovat. Pro dnešní procesory jsou to typicky 32 nebo 64-bitová čísla. **Multiple-precision** jsou pak taková čísla, pro jejichž binární reprezentaci je třeba více slov.

Představíme si následující třídy:

**ZZ** : neomezeně velká celá čísla

**ZZ\_p** : celá čísla modulo  $p$

**ZZX**, **ZZ\_pX** : polynomy nad příslušným okruhem

Před použitím jakékoliv třídy je třeba připojit odpovídající hlavičkový soubor:

`#include <NTL/???.h>`. Všechny číselné a polynomiální typy mají implementovány:

- základní aritmetické operace:  
+ - unární(-) += -= ++ -- \* \*= / /=

- operace %, %= kromě třídy ZZ\_p
- srovnávací operátory ==, !=, třída ZZ navíc disponuje operátory <, <=, >=, >
- operace >> >>= <<< << tj. operace posunu bitové reprezentace čísla doleva, doprava (násobení, dělení dvěma), v případě polynomů pak násobení, dělení mocninou proměnné.

NTL neobsahuje automatickou konverzi, tudíž je nutné funkci poskytnout právě ten typ, který očekává. Nicméně je naimplementováno množství konverzních funkcí typu `to_požadovanýtyp(proměnná)`, které lze použít, s několika málo výjimkami, pro všechny adekvátní kombinace vstupního a výstupního typu. Obecně pro většinu funkcí v NTL existují 2 verze: funkcionální a procedurální. Kromě základního případu, kdy se funkce liší pouze syntaxí, existují 3 další:

základní případ	ZZ x, a, b;	
	x = GCD(a, b);	– funkcionální tvar
	GCD(x, a, b);	– procedurální tvar
operátor hraje roli funkcionální verze	ZZ x, a, b;	
	x = a + b;	– funkcionální tvar
	add(x, a, b);	– procedurální tvar
pokud je název nejednoznačný, přidá se výstupní typ	ZZ_p x;	
	x = random_ZZ_p();	– funkcionální tvar
	random(x);	– procedurální tvar
konverzní funkce	ZZ x;	
	double a;	
	x = to_ZZ(a);	– funkcionální tvar
	conv(x, a);	– procedurální tvar

Nicméně ve výsledku je volána vždy stejná podprocedura. Použití procedurální verze je obvykle rychlejší, protože se vyvaruje vytváření dočasných objektů k ukládání výsledku. Nicméně takovéto urychlení není nijak výrazné, tudíž je obvyklé použití funkcionálního tvaru, který udržuje výsledný kód přehlednější a snáze pochopitelný. V následujícím přehledu se, kde to bude možné, omezím pouze na funkcionální tvary, v případě, že existuje pouze procedurální tvar ponechám u funkce prázdný výstupní typ `void`. Dále u funkcionálního tvaru vynechávám výstupní typ, který bude vždy zřejmý ze zaměření kapitoly, v opačném případě bude také uveden.

```
GCD(ZZ a, ZZ b)           funkcionální tvar
void GCD(ZZ x, ZZ a, ZZ b) procedurální tvar
```

## 2.1 Vstup, výstup a měření času

Standartní vstup a výstup je proveden pomocí operátorů `<<` a `>>`. Tyto operátory posílají(získávají) informace na(z) z předem definovaného proudu. Standartní (tedy výstup na obrazovku a vstup z klávesnice) jsou pojmenovány `cout` pro výpis a `cin` pro čtení jak ukazuje následující příklad, který faktorizuje ručně zadaný polynom  $f$  a vypíše dobu trvání faktorizace na obrazovku i do souboru `vystup.txt`:

```
#include<NTL/ZZX.h>
#include<NTL/ZZXFactoring.h>
#include<NTL/pair_ZZX_long.h>
#include<NTL/fileio.h>
#include<NTL/tools.h>

NTL_CLIENT

int main()
{
    ZZ a;
    ZZ c;
    double t;
    ostream out;
    vec_pair_ZZX_long factors;

    OpenWrite(out, "vystup.txt");

    cout << "zadejte polynom: ";
    cin >> a;

    t=GetTime();
    factor(c, factors, a);
    t=GetTime() -t;

    cout << " faktorizace polynomu " << a << " trvala " << t << "sekund";
    out << "stupen: " << deg(a) << ", cas: " << t << "\n";
    return 0;
}
```

Je vidět vytvoření proudu `out`, ke kterému je následně funkcí `OpenWrite(...)` vytvořen soubor `vystup.txt` pro zápis. Alternativou pro čtení ze souboru by pak bylo volání `ifstream in; OpenRead(in, "vstup.txt");`. Tyto funkce jsou přístupné po připojení hlavičkového souboru `fileio.h`. Soubor `tools.h` definuje kromě funkce `double GetTime()`, která vrací čas od spuštění programu, také například funkci `PrintTime(ostream& out, double t)` jež tento čas vypíše v tradičním tvaru do proudu `out`. Jsou definovány i funkce `long SkipWhiteSpace(istream& s)` a `long IsWhiteSpace(long a)` užitečné právě při čtení ze souboru.

`GetTime()` je užitečná funkce pro měření trvání libovolného výpočtu, jistým nedostatkem však může být omezená přesnost (milisekundy).

## 2.2 Přehled jednotlivých tříd

### 2.2.1 ZZ

Tato třída je použita k reprezentaci neomezeně dlouhých celých čísel. Je implementována jako posloupnost "zzigits", kde každá zdigit je `NTL_ZZ_NBITS-1`-bitové číslo (výchozí hodnota `NTL_ZZ_NBITS=26`). Kromě základních aritmetických operací jsou přítomny i pokročilejší funkce například pro testování prvočísel.

#### Modulární aritmetika

Ačkoliv je přítomna třída `ZZ_p` konkrétně určena pro počítání s celými čísly modulo nějaké  $p$ , třída `ZZ` poskytuje přístup ke stejným funkcím, které používá `ZZ_p`. Pro vyjimečné modulární počty je tedy zbytečné vytvářet `ZZ_p` objekty, nýbrž stačí využít funkcí:

<code>AddMod(ZZ a, ZZ b, ZZ n)</code>	sčítání
<code>SubMod(ZZ a, ZZ b, ZZ n)</code>	odčítání
<code>NegateMod(ZZ a, ZZ n)</code>	negace
<code>MulMod(ZZ a, ZZ b, ZZ n)</code>	násobení
<code>PowerMod(ZZ a, ZZ e, ZZ n)</code>	mocnění $a^e$
<code>InvMod(ZZ a, ZZ n)</code>	hledání inverzu k $a$

U poslední zmíněné je třeba ohlídat, aby  $a$  bylo modulo  $n$  invertibilní, v opačném případě funkce ohlásí chybu. Alternativou je pak `long InvModStatus(ZZ x, ZZ a, ZZ n)`, která pro invertibilní  $a$  uloží  $x = a^{-1}$  a vrátí 0, v opačném případě vrátí 1 a do  $x$  uloží hodnotu `NSD(a, n)`.

#### Největší společný dělitel

`GCD(ZZ a, ZZ b)` vrátí největší společný dělitel  $a, b$ , spočtený pomocí binárního NSD algoritmu, `void XGCD(ZZ d, ZZ s, ZZ t, ZZ a, ZZ b)` pak  $d, s, t$  splňující:  $d = \text{NSD}(a, b) = as + bt$  za použití Lehmerovy metody. Oba algoritmy jsou blíže popsány v kapitole 3.1. Single-precision varianta využívající pouze základní Euklidův algoritmus je pak přístupná jako `long GCD(long a, long b)`, rozšířená verze pak jako `void XGCD(long d, long s, long t, long a, long b)`.

## Bitové operace

Pro základní posuny bitové reprezentace čísla už byly zmíněny funkce `>>`, `>>=`, `=<<`, `<<`. Pro získání informací z této reprezentace využijeme například `NumBits(ZZ a)`, která vrací počet bitů binární reprezentace čísla  $a$ , `bit(ZZ a, long k)` pro zjištění  $k$ -tého bitu čísla  $a$  nebo `weight(ZZ a)` pro Hammingovu váhu  $a$ . Ale z této reprezentace těží i funkce `NumTwos(ZZ x)`, která pokud  $x \neq 0$  vrátí maximální  $e$  takové, že  $2^e$  dělí  $x$ . Pokud se těchto dvojek chceme rovnou zbavit, použijeme `MakeOdd(ZZ x)`, která též vrací  $e$  a navíc  $x$  přepíše  $x = x/2^e$ .

## Pseudonáhodný generátor

NTL disponuje spolehlivým generátorem pseudonáhodných posloupností, který je volán pro všechny třídy a funkce generující náhodná čísla. Před jakýmkoliv pokusem o získání náhodného čísla je dobré generátor inicializovat zavoláním `SetSeed(ZZ s)`; a nastavením seedu  $s$ . Pokud není funkce zavolána před prvním použitím generátoru použije se výchozí hodnota(0). Opětovným zavoláním s hodnotou 0 se generátor vrátí do výchozího stavu. Na hodnotu seedu je několikanásobně použita hashovací funkce MD5 a následný výstup tvoří inicializační vektor pro proudovou šifru ARC4 poskytující pseudonáhodný proud bajtů. Následně můžeme volat funkce pro získání náhodného čísla podle různých kritérií: `RandomBnd(ZZ n)` vrátí náhodné číslo z rozsahu  $0 \dots n-1$ . Pro náhodné číslo z rozsahu  $0 \dots 2^l-1$  zvolíme funkci `RandomBits_ZZ(long l)` a pokud záleží na konkrétní velikosti čísla, pak funkce `RandomLen_ZZ(long l)` vrátí náhodné číslo s právě  $l$  bity, tedy z intervalu  $2^{l-1} \dots 2^l - 1$ .

```
#include<NTL/ZZ.h>
NTL_CLIENT

int main(){
    ZZ max=to_ZZ(1000);
    ZZ p;

    SetSeed(to_ZZ(123456789));
    do p=RandomBnd(max);
    while (!ProbPrime(p));

    cout << p;

    return 0;
}
```

Příklad ukazuje inicializaci generátoru a jednoduchý cyklus pro generaci náhodného prvočísla  $p < \max$ . Tímto samozřejmě generátor při každém volání programu vytvoří stejnou posloupnost bajtů a tím i stejné prvočísla. Pro opravdu náhodné výstupy

je nutné inicializovat pokaždé jiným seedem, například s využitím času. Funkce `GetTime()` z kapitoly 2.1 však k tomuto účelu není zdaleka ideální, protože vrácený čas se při každém spuštění programu změní minimálně (ale většinou ne dost, aby to funkce vůbec zachytila). Pro inicializaci časem (počtem sekund reprezentujícím čas) je možné využít například následující konstrukci (která funguje spolehlivě, pokud není program volán několikrát v průběhu jedné sekundy):

```
#include <time.h>
...
    time_t t;
    time(&t);
    long tt=(long) t;
    SetSeed(to_ZZ(tt));
...
```

### Testování prvočíselnosti a generování prvočísel

Pro hledání prvočísel je zde hlavní implementace Miller-Rabinova testu prvočíselnosti `long MillerWitness(ZZ n, ZZ w)`, jež ověří, zda  $w$  je svědkem složenosti  $n$ . Tato funkce je pak součástí testu `long ProbPrime(ZZ n, long NumTrials=10)`<sup>1</sup>, který před vlastním `NumTrials`-násobným (výchozí hodnota 10) hledáním svědka provede sérii triviálních dělení. Test je následně využit při hledání náhodného  $l$ -bitového prvočísla `RandomPrime_ZZ(long l, long NumTrials=10)` a nejmenšího prvočísla  $> m$  `NextPrime(ZZ m, long NumTrials=10)`. Další možností jak získat  $l$ -bitové prvočísla je funkce `GenPrime_ZZ(long l, long err = 80)` využívající odhady Damgard, Landrock a Pomerance k optimalizaci počtu volání Miller-Rabin testu. Pravděpodobnost, že takové prvočísla je složené je pak  $2^{-err}$ . Pro speciální případy lze využít

`GenGermainPrime_ZZ(long l, long err = 80)` generující  $l$ -bitové Sophie Germain prvočísla (tj.  $p$  takové, že  $p' = 2p + 1$  je také prvočísla) se stejnou pravděpodobností omylu.

### 2.2.2 ZZ\_p

Třída `ZZ_p` představuje nástroj pro počítání s velkými čísly modulo nějaké kladné  $p$ . Toto je nejdříve třeba nastavit zavoláním `ZZ_p::init(p)` před prvním použitím třídy. Program si dále udržuje tento aktuální modulus v paměti. Modulus může být kdykoliv změněn stejnou funkcí, přítomny jsou rovněž nástroje k uložení a následné obnově modulu, viz dále. Hodnotu aktuálního modulu vrátí funkce `ZZ_p::modulus()`. Objekty třídy jsou reprezentovány jako prvky `ZZ` z rozsahu  $0..n-1$ . Přímý přístup

<sup>1</sup>funkce je volána s výchozí hodnotou `NumTrials=10`, tudíž ji není třeba udávat (viz příklad výše), nicméně je možno ji volat i s libovolnou hodnotou: `ProbPrime(a, 50)`

(ke čtení) k této **ZZ** hodnotě poskytuje funkce `rep(ZZ_p a)`. Pro náhodné číslo stačí zavolat funkci `random_ZZ_p()`, platí stále stejná pravidla pro pseudonáhodný generátor, viz oddíl podkapitoly 2.2.1 na straně 10.

## Dělení

Protože modulus **p** obecně nemusí být prvočíslo je třeba si dávat pozor při dělení. Pokud se pokusíme dělit neinvertibilním **b** prvkem **ZZ\_p** program se ukončí a nahlásí chybu. Tomuto se dá předejít využitím následujícího užitečného nástroje NTL: definováním vlastní funkce `void CH(const ZZ_p& b)` a jejím přiřazením příkazem `ZZ_p::DivHandler = CH` jak ukazuje následující příklad. Potom při pokusu o dělení neinvertibilním **b** je namísto ukončení programu zavolána tato funkce `CH` s parametrem **b** a po jejím provedení program pokračuje dál. Hodnotu **b** touto funkcí není možné měnit, ale je například užitečné vypsát parametry při nichž k chybě došlo. To platí pro všechny verze dělení `/`, `/=`, `div(...)` stejně jako pro funkci `inv(ZZ_p a)`, která hledá inverzní hodnotu k číslu **a**. Nicméně je třeba mít se na pozoru, protože při definované `CH` program pokračuje s neočekávanou hodnotou dělení (inverzu). Při volání funkce `inv(ZZ_p a)` je v případě neinvertibilního **a** vrácena hodnota `NSD(a,ZZ_p::modulus())`, která je v tomto případě samozřejmě netriviální. V případě dělení bohužel nelze očekávat ani to **a** je vrácena náhodná hodnota `ZZ_p`.

```
#include <NTL/ZZ_p.h>
NTL_CLIENT

void chyba(const ZZ_p& b){
    cout << "chyba pri deleni cisla: " << b << "\n";
    cout << "aktualni modulus: " << ZZ_p::modulus();
}

int main(){
    ZZ p;
    cin >> p;
    ZZ_p::init(to_ZZ(p));
    ZZ_p::DivHandler = chyba;
    ZZ_p a, b;

    cin >> a;        cin >> b;
    cout << inv(a) << "\n";
    cout << a/b;
    return 0;
}
```

## Záloha a obnova modulu

Protože program může v jednu chvíli pracovat pouze s jednou hodnotou modulu, stane se, že občas dočasně potřebujeme změnit modulus a následně se k němu vrátit. K tomu využijeme třídu `ZZ_pBak`. Tato je využívána hlavně v modulárních algoritmech, kdy funkce nejdříve uloží hodnotu aktuálního modulu a po výpočtu například faktorizace jej obnoví a program pokračuje dál. Následující příklad ukazuje využití při faktorizaci celočíselných polynomů:

```
#include<NTL/ZZXFactoring.h>
...
factor(...){
...
ZZ_pBak bak;
bak.save();                               uloží aktuální modulus
spfactors = SmallPrimeFactorization(...);
if (!spfactors) {
    cout >> "Polynom je ireducibilni.";
    bak.restore(); return;                 před ukončením modulus obnoví
}
...
Multilift(...);                           liftuje modulo nejvýhodnější  $p^e$ 
bak.restore();                                $p$  už není třeba
...

```

Funkce `SmallPrimeFactorization()` provede faktorizaci modulo několik malých prvočísel (tudíž několikrát modulus změní), zvolí nejvýhodnější modulus  $p$  a vrátí faktorizaci modulo toto  $p$ . Pokud je množina modulárních faktorů neprázdná pokračuje spočtením meze  $e$  a provedením Henselova liftování modulo  $p^e$ . Dále již není modulu  $p$  potřeba, obnoví se původní hodnota a algoritmus pokračuje.

Mezi uložením a obnovou modulu může dojít k neomezenému počtu inicializací `ZZ_p::init(p)`. V okamžiku změny aktuálního modulu může existovat neomezený počet objektů `ZZ_p`. Pokud byl starý modulus uložen a později obnoven, mohou být tyto objekty nadále používány, jako by k žádné změně nedošlo. Pokud je však objekt vytvořen pod jedním modulem a použit pod jiným, chování programu je nepředvídatelné (operace vrací neočekávané hodnoty). Žádná explicitní kontrola se neprovádí, tudíž podle autora hrozí, že dojde k chybě, nicméně při úmyslné práci pod jiným modulem se mi nepodařilo žádnou vyvolat (kromě neúspěšného dělení, viz předchozí část o dělení). Není dobré ani automaticky předpokládat, že objekty budou spolehlivě fungovat pokud dojde ke změně (nikoliv obnově původního) modulu na stejnou hodnotu pod jakou byly vytvořeny.

Inicializace `ZZ_p::init(n)` je také poměrně náročná, hledá FFT-prvočísla  $n'_i$

tak, že jejich součin  $\prod_i n'_i < n^2$  a pro každé z nich spočte a uloží například hodnoty  $\omega_i^j$  a  $1/\omega_i^j \bmod n'_i$ , kde  $\omega_i$  je primitivní  $m_i$ -tá odmocnina z jedné pro  $m_i$  maximální tak, že  $2^{m_i} | (n_i - 1)$ . Tudíž záloha modulu realizovaná pouhým přesměrováním ukazatelů v téměř nulovém čase představuje také, hlavně pro velké hodnoty modulu, určitou časovou úsporu.

### 2.2.3 ZZX

Třída **ZZX** představuje polynomy jedné proměnné nad celými čísly **ZZ**. Takový polynom  $f$  je implementován jako vektor **ZZ** hodnot, ke kterým lze přímo přistupovat jako `f.rep[i]`, což je koeficient u  $x^i$ . Konstantní člen je tedy `f.rep[0]` a vedoucí koeficient `f.rep[stupeň f]`, pro "read-only" přístup k těmto hodnotám slouží `ConstTerm(ZZX a)` a `LeadCoeff(ZZX a)`. Přímý přístup ke koeficientům umožňuje jejich pozměnění a jinou manipulaci. U těchto operací je nicméně důležité hlídat nenulovost vedoucího koeficientu, čehož lze snadno dosáhnout zavoláním funkce `f.normalize()`, která odstraní "vedoucí nuly" a opraví informaci o délce vektoru (a tedy i stupni polynomu). Bezpečnější alternativou je čtení pomocí `GetCoeff(ZZX x, long i)` a úprava pomocí funkce `SetCoeff(ZZX x, long i, ZZ a)`, která položí koeficient u  $x^i$  roven  $a$ . Pro vstup a výstup opět fungují `cin >>` a `cout <<`, ale je nutno dodržet formát, ve kterém `[a_0 a_1 ... a_n]` reprezentuje polynom  $a_0 + a_1x + \dots + a_nx^n$ .

U této třídy není bohužel přítomna žádná funkce pro generaci náhodného polynomu. Tato se dá však snadno napsat například jako:

```
#include<NTL/ZZX.h>
...
void random_ZZX(ZZX& x, long deg, ZZ& bnd)
{
    long i;
    x.rep.SetLength(deg);
    for (i = 0; i < deg; i++)
        RandomBnd(x.rep[i], bnd);
    x.normalize();
}
```

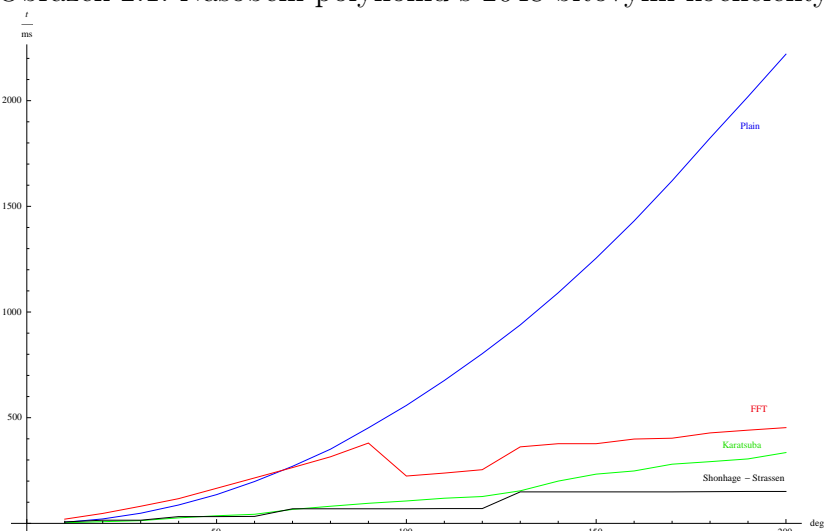
S využitím funkcí `RandomBits_ZZ()`, `RandomLen_ZZ()` lze samozřejmě snadno získat vhodné obměny.

### Násobení

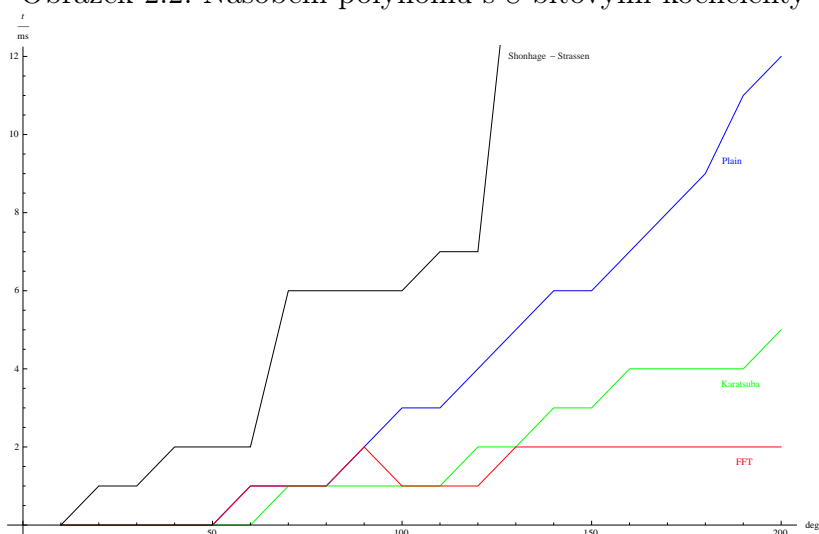
Pro násobení jsou implementovány 4 algoritmy, ze kterých se volí v závislosti na vstupních polynomech. Nicméně je možné využít i přímé volání jednotlivých algoritmů:

Klasický	PlainMul(ZZX x,ZZX a, ZXZ b)
Karatsubův algoritmus	KarMul(ZZX x,ZZX a, ZXZ b)
Schoenhage-Strassen/FFT	SSMul(ZZX x,ZZX a, ZXZ b)
CRT/FFT	HomMul(ZZX x,ZZX a, ZXZ b)

Obrázek 2.1: Násobení polynomů s 2048-bitovými koeficienty



Obrázek 2.2: Násobení polynomů s 8-bitovými koeficienty



Grafy (2.1) a (2.2) ukazují, jak Shonhage-Strassenův algoritmus je očekávaně nejrychlejším algoritmem pro polynomy vysokého stupně s velkými koeficienty. Základní "školní" násobení, použité, pokud je menší z polynomů stupně nejvýše 40

nebo má maximálně 3-bitové koeficienty, se v tomto měřítku neprosadilo avšak je zřetelně vidět rychlost Fourierovy transformace pro polynomy vysokého stupně, ale s menšími koeficienty.

## Dělení a pseudodělení

Máme-li polynomy  $a, b \in \mathbb{Z}[x]$ , pak existují polynomy  $q, r \in \mathbb{Q}[x]$  takové, že  $a = bq + r$ ,  $\deg(r) < \deg(b)$ . Nicméně tyto polynomy neleží pokaždé také v  $\mathbb{Z}[x]$ . Standartní operace dělení (`/`, `%`, `div()`, `rem()`, `DivRem()`) vrací  $q$  a/nebo  $r$  pouze pokud  $q$  a  $r$  leží v  $\mathbb{Z}[x]$ . V opačném případě ohlásí chybu a předčasně skončí. Pokud se například pokusíme monický polynom  $f$  dělit polynomem  $2x+1$ , program ohlásí chybu, protože vedoucího koeficient výsledného polynomu  $f/(2x+1)$  je roven  $\frac{1}{2}$  a tudíž neleží v  $\mathbb{Z}$ . Zda  $b|a$  v  $\mathbb{Z}[x]$  (myšleno beze zbytku) otestujeme pomocí `divide(ZZX a, ZX b)`, případně použitím funkce `divide(ZZX q, ZX a, ZX b)` v kladném případě rovnou obdržíme  $q = a/b$ . Tyto algoritmy využívají (pro rozdíl stupňů  $> 8$ ) modulární metody, provádějící dělení modulo malá prvočísla a následnou rekonstrukci pomocí Čínské věty o zbytcích. Problém dělení se dá obejít tzv. pseudodělením, pro které je přítomna funkce `PseudoDivRem(ZZX q, ZX r, ZX a, ZX b)` vracející  $q$  a  $r$  taková, že

$$\deg(r) < \deg(b) \text{ a zároveň } \text{LeadCoeff}(b)^{\deg(a)-\deg(b)+1} a = bq + r$$

Dále je třeba zmínit jednoduché, ale užitečné funkce pro výpočet obsahu polynomu `content(ZZ d, ZX f)` a jeho primitivní části `PrimitivePart(ZZX pp, ZX f)`. Pro výpočet největšího společného dělitele polynomů  $a$  a  $b$  využijete funkci `GCD(ZZX a, ZX b)` a mírně odlišnou `XGCD(ZZ r, ZX s, ZX t, ZX a, ZX b)`, která do proměnné  $r$  uloží `resultant(a, b)` a v případě, že tento je nenulový, spočítá  $s$  a  $t$  taková, že  $as + bt = r$ . K těmto výpočtům je implementován modulární algoritmus.

## Modulární aritmetika

Pro polynomiální aritmetiku modulo  $x^n$  slouží funkce `trunc(ZZX a, long n)` pro výpočet  $a \bmod x^n$ , `MulTrunc(ZZX a, ZX b, long n)` pro součin  $a \cdot b \bmod x^n$  a `InvTrunc(ZZX a, long n)` pro hledání inverzní mocninné řady  $\hat{a}$ , takové, že  $a\hat{a} = 1 \bmod x^n$ . Pro násobení modulo obecný nenulový monický polynom  $f$  pak slouží `MulMod(ZZX a, ZX b, ZX f)`, ale pouze v případě, že vstupní polynomy jsou menšího stupně než  $f$ .

### 2.2.4 ZZ\_pX

`ZZ_pX` nám poskytuje nástroje pro počítání s polynomy nad `ZZ_p`. Podobně jako u `ZZX`, prvek `ZZ_pX` je reprezentován jako vektor hodnot `ZZ_p` představující

koeficienty polynomu. I zde jsou jednotlivé koeficienty u  $x^i$  přímo dostupné jako `f.rep[i]`, což sebou nese stejná manipulační rizika jako u **ZZX**. Dále jsou k dispozici shodně pojmenované funkce pro většinu operací jako u **ZZX**. Třída používá modulus z **ZZ\_p** tudíž, pokud jsme s ní v programu ještě nepracovali je nutné modulus inicializovat pomocí `ZZ_p::init(p)`. To opět vyžaduje nutnou opatrnost při dělení v případě neprvočíselného `p`. viz oddíl podkapitoly 2.2.2 na straně 12.

### Největší společný dělitel

Pro výpočet největšího společného dělitele je opět přítomna funkce `GCD(ZZ_pX a, ZZ_pX b)` využívající Euklidův algoritmus pro polynomy nižšího stupně a `Half-GCD` algoritmu pokud je alespoň jeden polynom stupně většího než 180, viz kapitola 3.2. `void XGCD(ZZ_pX x, ZZ_pX s, ZZ_pX t, ZZ_pX a, ZZ_pX b)` již pak využívá pouze algoritmus `Half-GCD`. Přítomny jsou však i funkce `void PlainGCD(ZZ_pX x, ZZ_pX a, ZZ_pX b)` a `void PlainXGCD(ZZ_pX x, ZZ_pX s, ZZ_pX t, ZZ_pX a, ZZ_pX b)` využívající pouze Euklidův algoritmus.

### Modulární aritmetika

Jako u **ZZX** i zde jsou přítomny stejně nazvané funkce pro výpočty modulo  $x^n$  a modulo monický ireducibilní polynom  $f$ . Avšak třída **ZZ\_pX** nabízí více možností v případě, že chceme provádět více výpočtů modulo fixní  $f$ . Potom je možné provést předvýpočet a uložit jej do speciální třídy zavoláním `ZZ_pXModulus F(f)`. Tato třída je následně využita ve funkcích optimalizovaných pro operace modulo  $f$ .

```
#include<NTL/ZZ_pX.h>
NTL_CLIENT

int main(){
    ZZ_p::init(to_ZZ(19));
    ZZ_pX f, a, b;
    long dg=10;

    cin >> f;
    ZZ_pXModulus F(f);

    do {
        random(a, dg);
        rem(b, a, F);
    } while ( deg(b) != 0 );

    cout << a;
    return 0;
}
```

V praxi tento předvýpočet znamená pro polynomy stupně větší než 21 výpočet a uložení Furierovy transformace do  $F$ . Dále tedy pro výpočty modulo  $f$  používáme tyto informace uložené v  $F$  pro:

násobení: `MulMod(ZZ_pX a, ZZ_pX b, ZZ_pXModulus F)`  
mocnění: `PowerMod(ZZ_pX a, ZZ e, ZZ_pXModulus F)`  
dělení: `DivRem(ZZ_pX q, ZZ_pX r, ZZ_pX a, ZZ_pXModulus F)`  
( $q = a/f$ ,  $r = a \bmod f$ ).  
 $x^e \bmod f$ : `PowerXMod(ZZ_pX x, ZZ e, ZZ_pXModulus F)`  
 $(x + A)^e \bmod f$ : `PowerXPlusAMod(ZZ_p A, ZZ e, ZZ_pXModulus F)`

Dokonce je zde ještě další možnost pro speciální případ, kdy provádíme mnoho výpočtů  $a \cdot b \bmod f$  s fixním  $b$  (například umocňování  $b$  modulo  $f$ ). Pak takový předvýpočet provedeme vytvořením objektu  $B$  následující třídy:

```
ZZ_pXMultiplier B(b, F);
```

a následné násobení obstará funkce

```
MulMod(ZZ_pX a, ZZ_pXMultiplier B, ZZ_pXModulus F).
```

# Kapitola 3

## Největší společný dělitel

### 3.1 Největší společný dělitel v oboru celých čísel

NTL hojně využívá níže popsané algoritmy pro výpočet NSD celých čísel ve většině zmíněných algoritmů ať už pro výpočet NSD nebo faktorizaci. Následující popis algoritmů je převzat z [3], kde lze také nalézt i jejich podrobnou analýzu.

#### Binární algoritmus

Objeven J. Steinem v roce 1961. Tento algoritmus narozdíl od Euklidova algoritmu nevyužívá operaci dělení a spoléhá se na operace odčítání a půlení (tj. dělení dvěma, nicméně využívá operace posunu binární reprezentace sudého čísla doprava). Algoritmus plyne z následujících triviálních pozorování:

- Jsou-li  $u$  a  $v$  sudé, pak  $\text{NSD}(u, v) = 2 \text{NSD}(u/2, v/2)$
- Je-li  $u$  sudé a  $v$  liché, pak  $\text{NSD}(u, v) = \text{NSD}(u/2, v)$
- $\text{NSD}(u, v) = \text{NSD}(u - v, v)$
- pokud jsou  $u, v$  liché pak  $u - v$  je sudé a  $|u - v| < \max(u, v)$

**Binární algoritmus** Pro kladná celá čísla  $u, v$  najde jejich největší společný dělitel.

1. Polož  $k := 0$  a dokud jsou  $u, v$  sudé opakuj  $k := k + 1, u := u/2, v := v/2$ .
2. (Nyní byly původní hodnoty  $u, v$  vyděleny  $2^k$  a alespoň jedna z nich je nyní lichá.)  
Pokud je  $u$  liché polož  $t := -v$  a jdi na (4) jinak polož  $t := u$ .
3. (V tuto chvíli je  $t$  sudé.) Polož  $t := t/2$ .

4. Pokud je  $t$  sudé vrať se na (3).
5. Když  $t > 0$  polož  $u := t$  jinak  $u := -t$ . (Větší z  $u, v$  je nahrazeno hodnotou  $t$ .)
6. Polož  $t := u - v$ . Jestli  $t \neq 0$  jdi zpět na (3). V opačném případě vrať hodnotu  $u2^k$ .

## Lehmerova metoda

Výrazného zlepšení rychlosti Euklidova algoritmu v případě počítání s velkými čísly může být dosaženo použitím metody D. H. Lehmera(1938). Pracující pouze s vedoucími ciframi velkých čísel je možné zvládnout většinu výpočtů s použitím single-precision aritmetiky. Spousta času se ušetří prováděním "virtuálního" výpočtu namísto opravdového počítání.

**Př:** Mějme dvě osmiciferná čísla  $u = 27182818$ ,  $v = 10000000$  a předpokládejme použití stroje pracujícího pouze se 4-cifernými slovy. Buď  $u' = 2718$ ,  $v' = 1001$ ,  $u'' = 2719$ ,  $v'' = 1000$ , pak  $u'/v'$  a  $u''/v''$  jsou odhady  $u/v$ :

$$u'/v' < u/v < u''/v'' \quad (3.1)$$

Podíl  $u/v$  udává posloupnost koeficientů  $q$  z Euklidova algoritmu. Pokud provedeme Euklidův algoritmus paralelně se single-precision hodnotami  $(u', v')$  a  $(u'', v'')$  až do bodu, kdy dostaneme odlišné koeficienty, není těžké prohlédnout, že stejnou posloupnost koeficientů bychom obdrželi při výpočtu Euklidova algoritmu s multiple-precision hodnotami  $(u, v)$ . Nyní pozorujme, jak probíhá Euklidův alg. na hodnotách  $(u', v')$  a  $(u'', v'')$ :

u'	v'	q'	u''	v''	q''
2718	1001	2	2719	1000	2
1001	716	1	1000	719	1
716	285	2	719	281	2
285	146	1	281	157	1
146	139	1	157	124	1
139	7	19	124	33	3

Prvních pět koeficientů se v obou případech shoduje, tedy musí být pravé. Ale v šestém kroku máme  $q' \neq q''$ , tedy single-precision operace skončí. Dozvěděli jsme se,

že originální výpočet s multiple-precision by probíhal následovně:

$$\begin{array}{rcc}
 & u & v & q \\
 \hline
 & u_0 & v_0 & 2 \\
 & v_0 & u_0 - 2v_0 & 1 \\
 & u_0 - 2v_0 & -u_0 + 3v_0 & 2 \\
 & -u_0 + 3v_0 & 3u_0 - 8v_0 & 1 \\
 & 3u_0 - 8v_0 & -4u_0 + 11v_0 & 1 \\
 & -4u_0 + 11v_0 & 7u_0 - 19v_0 & ?
 \end{array} \tag{3.2}$$

(Následující koeficient leží mezi 3 a 19.) Nezáleží na počtu cifer  $u, v$ , těchto prvních pět kroků Euklidova algoritmu bude vypadat stejně jako v (3.2) tak dlouho, dokud platí (3.1). Takto se můžeme vyhnout multiple-precision operacím v prvních pěti krocích a nahradit je všechny dvěma multiple-precision výpočty  $-4u_0 + 11v_0$  a  $7u_0 - 19v_0$ . V tomto případě  $u = 1268728$ ,  $v = 279726$ ; výpočet pak dále pokračuje podobným způsobem pro  $u' = 1268$ ,  $v' = 280$ ,  $u'' = 1269$ ,  $v'' = 279$ , atd. Jak ukázal příklad pět cyklů Euklidova algoritmu se podařilo zkombinovat do jediného multiple-precision výpočtu, ale s u stroje pracujícího s vícecifernými slovy by se tímto způsobem dalo ušetřit mnohem více kroků. Výsledky ukazují, že počet nahraditelných cyklů je úměrný počtu cifer používaných pro single-precision operace.

**Lehmerova metoda** (Euklidův algoritmus pro velká čísla) Nechť  $u, v$  jsou nezáporná celá čísla v multiple-precision reprezentaci. Algoritmus spočítá největší společný dělitel  $u, v$  využívajíc pomocných  $p$ -ciferných single-precision proměnných  $\hat{u}, \hat{v}, A$ ,

$B, C, D, T, q$  a multiple-precision proměnných  $t, w$ .

1. Je-li  $v$  dostatečně malé k reprezentaci jako single-precision spočítej  $\text{NSD}(u, v)$  pomocí Euklidova algoritmu a skonči. V opačném případě, nechť  $\hat{u}$  je prvních  $p$  cifer čísla  $u$  a  $\hat{v}$  odpovídající počet cifer čísla  $v$ . Polož  $A := 1$ ,  $B := 0$ ,  $C := 0$ ,  $D := 1$ . Tyto proměnné reprezentují koeficienty z Euklidova algoritmu, kde

$$u = Au_0 + Bv_0, \quad v = Cu_0 + Dv_0 \tag{3.3}$$

Dále také platí:

$$u' = \hat{u} + B, \quad v' = \hat{v} + D, \quad u'' = \hat{u} + A, \quad v'' = \hat{v} + C \tag{3.4}$$

vzhledem k notaci z předchozího příkladu.

2. Polož  $q := \lfloor (\hat{u} + A) / (\hat{v} + C) \rfloor$ . Pokud  $q \neq \lfloor (\hat{u} + B) / (\hat{v} + D) \rfloor$  jdi na (4) (Tento krok testuje jestli  $q' \neq q''$  podle předchozího příkladu. Ve speciálním případě

může dojít k přetečení single-precision, což se ale stane pouze když  $\hat{u} = b^p - 1$  a  $A = 1$  nebo  $\hat{v} = b^p - 1$  a  $D = 1$ . Podmínky:

$$\begin{aligned} 0 \leq \hat{u} + A \leq b^p, & \quad 0 \leq \hat{v} + C < b^p, \\ 0 \leq \hat{u} + B < b^p, & \quad 0 \leq \hat{v} + D \leq b^p \end{aligned} \quad (3.5)$$

platí díky (3.4). Je možné, že  $\hat{v} + C = 0$  nebo  $\hat{v} + D = 0$ , ale ne obojí najednou; potom pokus dělení nulou je bráno jako "jdi na (4)".

3. Polož  $T := A - qC$ ,  $A := C$ ,  $C := T$ ,  $T := -qD$ ,  $B := D$ ,  $D := T$ ,  $T := \hat{u} - q\hat{v}$ ,  $\hat{u} := \hat{v}$ ,  $\hat{v} := T$  a vrať se na (2). (Tyto single-precision operace jsou ekvivalentem multiple-precision operací z (3.2)).
4. Pokud  $b = 0$  polož  $t := u \bmod v$ ,  $u := v$ ,  $v := t$  s použitím multiple-precision dělení. (Toto nastává pouze pokud single-precision operace nemůže simulovat žádnou z multiple-precision operací. To znamená, že Euklidův algoritmus vyžaduje velký koeficient  $q$ , což se stává velmi vyjímečně.) Jinak polož  $t := Au$ ,  $t := t + Bv$ ,  $w := Cu$ ,  $w := w + Dv$ ,  $u := t$ ,  $v := w$  (využívající jednoduchých multiple-precision operací.) Vrať se na (1).

Hodnoty  $A$ ,  $B$ ,  $C$ ,  $D$  v každém případě zůstávají single-precision jak ukazuje (3.5).

## 3.2 Největší společný dělitel polynomů nad konečným tělesem

Označme  $a_1$  a  $a_2$  dva nenulové polynomy a předpokládejme, že  $\deg(a_1) > \deg(a_2)$ . Euklidův algoritmus pak počítá PRS (posloupnost polynomiálních zbytků):

$$\begin{aligned} a_1 &= q_1 a_2 + a_3 \\ a_2 &= q_2 a_3 + a_4 \\ &\vdots \\ a_{t-1} &= q_{t-1} a_t \end{aligned}$$

s  $a_{t+1} = 0$  a  $a_t = \text{NSD}(a_1, a_2)$ . Jak už bylo zmíněno dříve NTL využívá tento algoritmus k výpočtu největšího společného dělitele nad **ZZ\_p**. Ale v případě, že stupeň jednoho ze vstupních polynomů překročí hranici `NTL_ZZ_pX_GCD_CROSSOVER` (výchozí hodnota 180), je nejprve použit "Half-GCD" algoritmus pro snížení stupně polynomů vstupujících do Euklidova algoritmu. "Half-GCD" algoritmus využívá následujícího faktu:

Pokud definujeme matici  $\mathbb{M}_i$  následujícím způsobem:

$$\mathbb{M}_i := \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix},$$

potom

$$\begin{pmatrix} a_k \\ a_{k+1} \end{pmatrix} = \mathbb{M}_{k-1} \mathbb{M}_{k-2} \dots \mathbb{M}_1 \begin{pmatrix} a_1 \\ a_2 \end{pmatrix},$$

kde  $k \leq t$ . "Half-gcd" algoritmus pak počítá součin  $\mathbb{M}_{k-1} \mathbb{M}_{k-2} \dots \mathbb{M}_1$  takový, že  $a_k$  a  $a_{k+1}$  jsou členy zbytkové posloupnosti  $a_1, a_2, \dots$ , a platí  $\deg(a_k) \leq \deg(a_1) - d$ . Máme-li ještě číslo  $n$  a polynom  $f = \sum_{i=0}^d a_i x^i$  stupně  $d > n$ , pak  $f|n$  představuje polynom stupně  $n$  tvaru:  $\sum_{i=0}^n a_{d-n+i} x^i$ . NTL využívá následující implementaci:

**HalfGCD**( $\mathbb{M}, u, v, d = \lceil \deg(u)/4 \rceil$ )

1. pokud  $v == 0$  nebo  $\deg(v) \leq \deg(u) - d$  vrať  $\mathbb{M} := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2.  $n = \deg(u) - 2d + 2$
3.  $u_1 = u|n, v_1 = v|n$
4. pokud  $d < \text{NTL\_ZZ\_pX\_HalfGCD\_CROSSOVER}$  (výchozí hodnota 25) proved' **IterHalfGCD**( $\mathbb{M}, u_1, v_1, d$ ) a skonči
5.  $d_1 = \lceil d/2 \rceil$
6. **HalfGCD**( $\mathbb{M}_1, u_1, v_1, d_1$ )
7.  $(u_1, v_1)^T = \mathbb{M} * (u_1, v_1)^T$
8.  $d_2 = \deg(v_1) - \deg(u) + n + d$
9. pokud  $\deg(v_1) == 0$  nebo  $d_2 \leq 0$  polož  $\mathbb{M} = \mathbb{M}_1$  a skonči
10. spočti  $u_1 = u_1 \bmod v_1$  a  $q = u_1/v_1$  a zaměň hodnoty  $u_1$  a  $v_1$
11.  $\mathbb{M}_1 = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} * \mathbb{M}_1$
12. **HalfGCD**( $\mathbb{M}_2, u_1, v_1, d_2$ )
13.  $\mathbb{M} = \mathbb{M}_2 * \mathbb{M}_1$

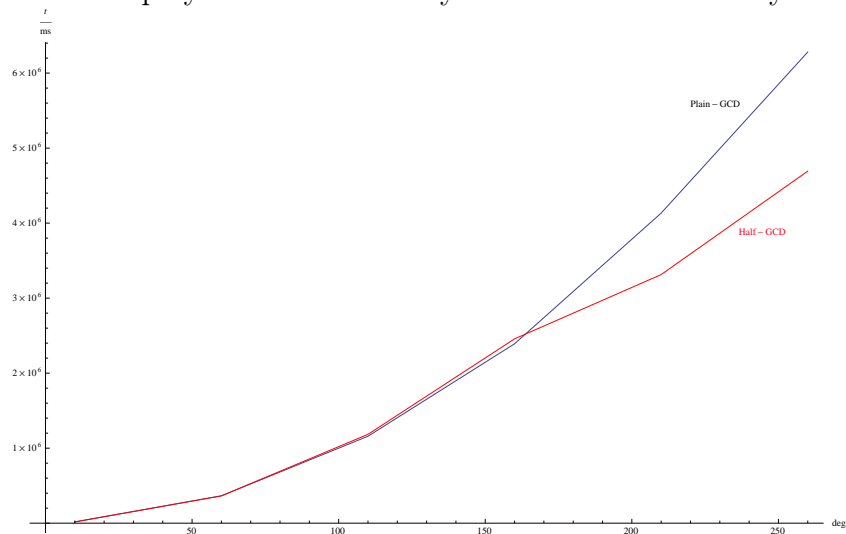
Funkce **IterHalfGCD** představuje iterovanou zjednodušenou verzi algoritmu. Budeme-li se držet notace z čtvrtého kroku u volání **IterHalfGCD** pak tato funkce vypadá následovně:

1. polož  $g = \deg(u_1) - d$  a  $\mathbb{M} := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. dokud  $\deg(v_1) > g$  počítej
  - $u_1 = u_1 \bmod v_1, q = u_1/v_1$  a zaměň hodnoty  $u_1$  a  $v_1$
  - $\mathbb{M}_1 = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} * \mathbb{M}_1$

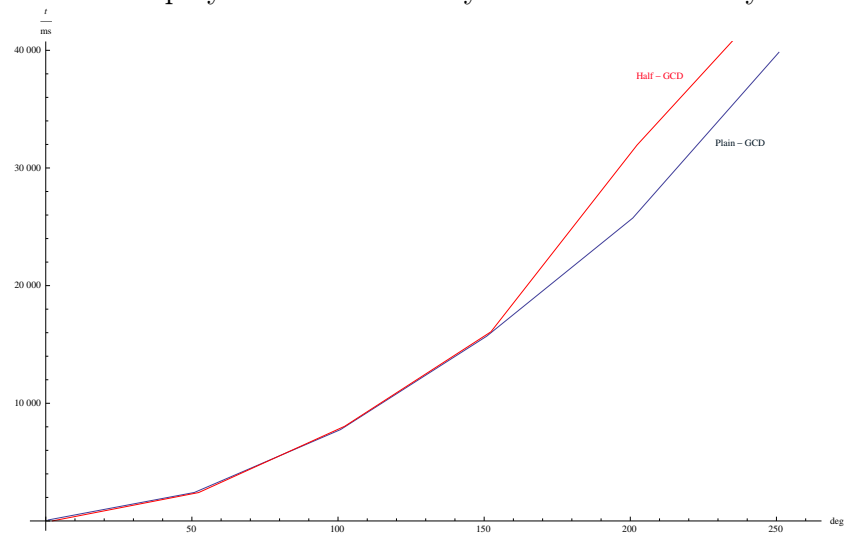
Pro popis podobné implementace a důkaz správnosti algoritmu doporučuji [2]. Hlavní odlišnosti spočívají v mezích  $d, d_1, d_2, n$  což je pravděpodobně výsledkem optimalizace. Pro  $n = \deg(a_1)$  má algoritmus složitost  $O(n \log^2(n))$ , jak je ukázáno například v [8]

Na grafu (3.1) je jasně patrné, kdy NTL začíná používat half-GCD metodu pro snížení stupně polynomů vstupujících do Euklidova algoritmu a jak výraznou časovou úsporou představuje. Pokud však počítáme s menšími koeficienty, je použití half-GCD spíše na obtíž a výpočet naopak brzdí jak ukazuje graf (3.2).

Obrázek 3.1: NSD polynomů nad konečným tělesem s 2048-bitovými koeficienty



Obrázek 3.2: NSD polynomů nad konečným tělesem s 8-bitovými koeficienty



### 3.3 Největší společný dělitel polynomů nad $\mathbb{Z}$

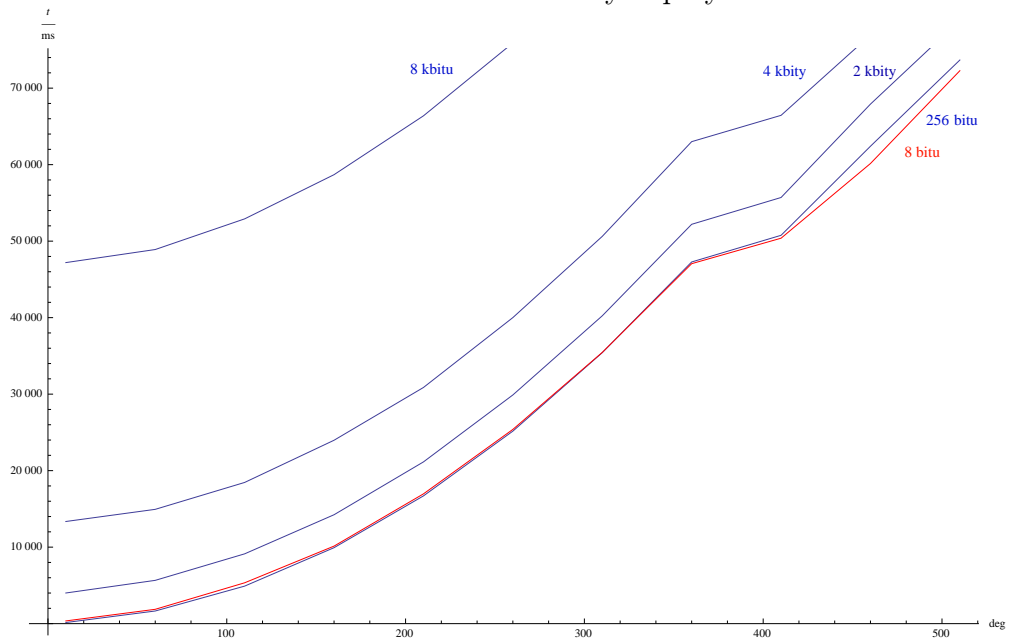
Pro výpočet NSD polynomů nad celými čísly má NTL implementován modulární algoritmus ve funkci `GCD(ZZX f, ZX g)`. Téměř shodný popis následujícího algoritmu je uveden v [1] jako Algoritmus 22. Jediným rozdílem je na první pohled (ne)omezení cyklu Landau-Mignottovou mezí. Uvnitř cyklu jsou proměnné konvertovány do třídy `ZZ_pX`, která je optimalizována pro výpočty se single-precision modulem, což je zde splněno vzhledem k postupu od nejmenšího prvočísla. NTL implementace bere jako vstup libovolné polynomy `ZZ_pX`, tudíž jsou zpočátku spočteny primitivní části a nakonec je nazpět přinásoben největší společný dělitel obsahů. Notace byla zvolena dle [1] pro snazší porovnání.

**GCD**( $f, g$ )

1. Pokud  $f$  (resp.  $g$ ) == 0 vrať  $g$  (resp.  $f$ )
2.  $c = \text{NSD}(\text{cont}(f), \text{cont}(g))$
3.  $f_1 = \text{PrimitivePart}(f)$ ,  $f_2 = \text{PrimitivePart}(g)$
4.  $d = \text{NSD}(\text{LeadCoeff}(f_1), \text{LeadCoeff}(f_2))$
5. Pro  $p$  nejmenší následující prvočíslu:
  - pokud  $p | \text{LeadCoeff}(f_1)$  nebo  $p | \text{LeadCoeff}(f_2)$  přeskoč kolo cyklu
  - $h = d \cdot \text{NSD}(f_1, f_2)$
  - pokud  $\deg(h) == 0$  vrať 1
  - pokud probíhá první cyklus nebo pokud  $\deg(h) < \deg H$  polož  $P = p$  a  $H = h$
  - pro  $\deg(h) > \deg H$  přeskoč kolo cyklu
  - pro  $\deg(h) == \deg H$ 
    - použij čínskou větu o zbytcích pro výpočet  $H'$  takového, že  $H' \equiv H \pmod{P}$  a  $H' \equiv h \pmod{p}$
    - polož  $P = P * p$
    - pokud  $H == H'$  polož  $H = \text{PrimitivePart}(H')$  a v případě, že  $H | f_1$  a zároveň  $H | f_2$  ukonči cyklus
6. vrať  $c * H$

Složitost algoritmu je v nejhorším případě  $O(n^4 \log^2(n))$  pro  $n = \max\{\deg(f), \deg(g)\}$ , ale v průměrném případě je možno počítat se složitostí  $O(n^3 \log^2(n))$ , viz. [1]. Efektivitu algoritmu bohužel není s čím porovnávat, následující graf tak alespoň ukazuje přibližnou dobu výpočtu v závislosti na stupni vstupních polynomů a bitové délce jejich koeficientů.

Obrázek 3.3: NSD celočíselných polynomů



Na grafu je pro všechny bitové délky zřetelný skok u stupně 400. Tento je způsoben přechodem na half-GCD algoritmus namísto Euklidova. Protože se zde počítá modulo malá prvočísla, NTL využívá třídu `zz_pX`, optimalizovanou pro počty modulo single-precision hodnotu, která má nastavenou pro tento přechod jinou hranici než třída `ZZ_pX`.

# Kapitola 4

## Faktorizace polynomů

### 4.1 Nad celými čísly

Funkce `void factor(ZZ& c, vec_pair_ZZX_long& factors, const ZZ& f)` používá stejný algoritmus pro faktorizaci polynomu  $f \in \mathbb{Z}[x]$  jako většina jiných implementací, tedy:

1. Nejdříve je provedena bezčtvercová faktorizace v  $\mathbb{Z}[x]$ .
2. Následně vypočtena faktorizace modulo nějaké malé prvočíslo  $p$
3. Provedeno liftování
4. Kombinací modulárních faktorů získána faktorizace  $f$

Ve druhém kroku je nejdříve provedena faktorizace modulo několik malých prvočísel a následně je zvolen nejlepší kandidát jako  $p$  (tak, že  $f \bmod p$  má nejméně faktorů). Liftování je prováděno pomocí kvadratického Henselova liftovacího algoritmu. Pro kombinační fázi jsou pak implementovány 2 metody:

- Metoda van Hoeij
- Zassenhausovu metoda

Algoritmy pro řešení prvních tří fází jsou již obecně známé, soustředně se tedy na poslední, kombinační fázi. Máme tedy množinu faktorů  $f \bmod p^k$ :

$$\{f_1, f_2, \dots, f_n\} \tag{4.1}$$

a následující kombinatorický problém: Jak najít  $S$  podmnožiny (4.1), pro které součin prvků  $S$  dělí  $f$ ?

### 4.1.1 Zassenhausova metoda

Zassenhausova metoda je více či méně hrubý průchod všemi možnými kombinacemi modulárních faktorů  $f$ . NTL obsahuje implementaci algoritmu popsanou v [7].

Podmnožiny jsou procházeny podle vzrůstající kardinality, přičemž na každou podmnožinu modulárních faktorů je aplikována sada rychlých testů na vyřazení ne-správných kombinací tak, aby se minimalizoval počet potenciálně správných kombinací vstupujících do finálního, časově náročného zkusmého dělení. Příkladem takového testu budiž  $d - 1$  test:

**d-1 test** Máme-li polynom  $a$  stupně  $d$ , potom *předposledním koeficientem polynomu  $a$*  rozumíme koeficient členu  $x^{d-1}$  tohoto polynomu. Test jednoduše ověřuje, zda předposlední koeficient součinu modulárních faktorů leží v mezích spočtených pro skutečný faktor  $f$  (pro většinu případů je tento úzký interval snadno k nalezení, viz [7]). Hlavní výhodou testu je malá náročnost, vzhledem k tomu, že používá pouze sčítání: předposlední koeficient součinu monickým polynomů je součtem jejich odpovídajících předposlední koeficientů. Úspěch testu se dále odvíjí od dvou klíčových skutečností: velikost vymezujícího intervalu pro předposlední koeficient skutečných faktorů  $f$  je typicky malá vzhledem k modulu  $p^e$ , naproti tomu se však předposlední koeficienty špatných kombinací chovají jako uniformně distribuovaná náhodná proměnná modulo  $p^e$ . S těmito znalostmi můžeme odstranit všechny kombinace, jejichž předposlední koeficient leží mimo spočtené hranice s předpokladem, že test velmi pravděpodobně vyřadí většinu špatných kombinací za malou cenu.

### 4.1.2 Metoda van Hoeij

První polynomiální algoritmus (od Lenstra, Lenstra, Lóvasz) se kombinačnímu problému vyhýbá následujícím způsobem: Namísto zkoumání všech podmnožin vezme pouze 1 prvek množiny (4.1), který použije ke konstrukci mříže vstupující do LLL algoritmu, která (pokud  $f$  není ireducibilní) obsahuje vektor  $U_g$  jehož prvky jsou koeficienty polynomu  $g$  takového, že  $\text{NSD}(g, f)$  je netriviální dělitel  $f$ . Algoritmus LLL nalezne tento vektor v polynomiálním čase vzhledem ke stupni  $f$  a velikosti jeho koeficientů. Což je také největší slabinou algoritmu, protože tyto hodnoty jsou mnohem větší než  $n$  a tudíž algoritmus je v reálném použití mnohem pomalejší než Zassenhausova metoda.

Buď  $v = (v_1 \dots v_n) \in \{0, 1\}^n$  a  $g_v = \prod f_i^{v_i}$ . A označme  $B = \{v \mid g_v \text{ je monický ireducibilní faktor } f\}$ .

Metoda Marka van Hoeij se kombinatorickému problému nevyhýbá, ale namísto k výpočtu samotného  $g$ , používá LLL algoritmus k nalezení množiny  $B$ . Vektory  $B$

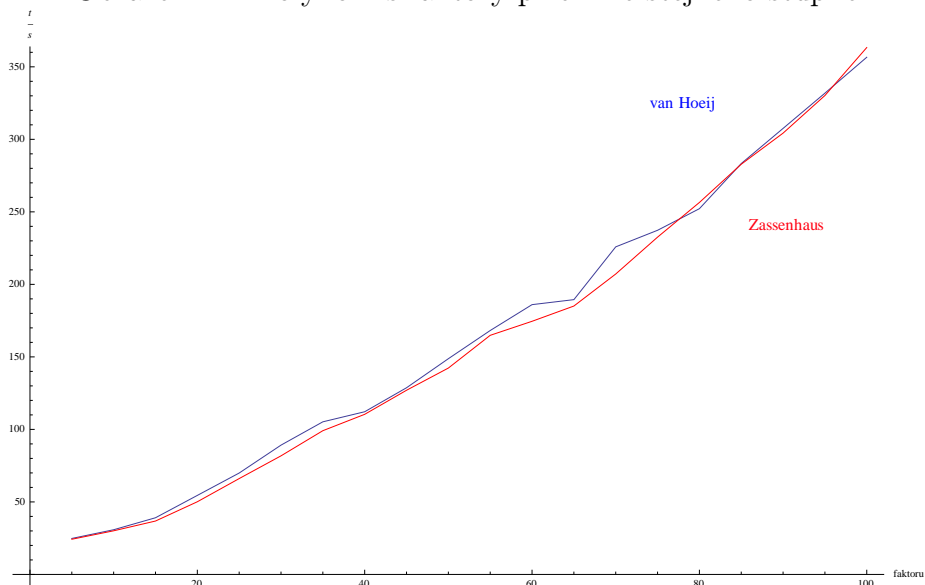
mají v porovnání s  $U_g$  mnohem menší (1-bitové) souřadnice a vzhledem k tomu, že mají pouze  $n$  souřadnic, jsou i mnohem kratší než  $U_g$ . Buď  $e_1 \dots e_n$  standartní báze  $\mathbb{Z}^n$ . Pak pro nějaké  $s$ ,  $s \ll n$ , definujme  $(n + s)$ -složkové vektory  $V_i = (e_i || T_{a,b}(f_i))$  pro  $i = 1 \dots n$  a  $E_j$  pro  $j = 1 \dots s$ , které mají nulové všechny souřadnice, kromě  $(n + j)$ -té, která je rovna  $p^{a-b}$ . Vstupem do LLL je pak báze mříže daná těmito vektory  $V_i$  a  $E_j$ . Zobrazení  $T_{a,b} : \mathbb{Z}_{p^k}[x] \rightarrow \mathbb{Z}_{p^k}^s$  je kromě jiného zvoleno tak, aby souřadnice vektoru nenesly žádnou informaci (kromě množiny B) o hledaném faktoru  $g$ . To znamená, že časová složitost LLL v tomto případě závisí pouze na  $n$  a nikoliv na  $\deg(f)$  nebo velikosti koeficientů  $f$ . Metoda poskytuje značnou volnost ve volbě  $s$ ,  $a$ ,  $b$  a jsou v tomto směru prováděny četné testy a heuristiky, viz např. [5].

Vzhledem k náročnosti i jen popisu algoritmu na teoretický podklad a značení, odkazují na [4], kde se nachází vše potřebné včetně několika optimalizačních tipů a experimentů.

## Srovnání

Rychlé testy v Zassenhausově algoritmu samozřejmě neodstraní všechny nesprávné kombinace a jejich efektivita se výrazně liší vzhledem ke vstupu, tudíž v nejhorším případě má metoda stále exponenciální složitost vzhledem k počtu modulárních faktorů  $n$ , nicméně toto rychlé testování výrazně urychluje běh faktorizace pro obecný polynom. Van Hoejiho metoda skončí v polynomiálním čase  $O(n)$ , jak ukazuje autor v [4]. Přesnější asymptotický odhad prozatím nebyl nikým zveřejněn.

Obrázek 4.1: Polynom s faktory přibližně stejného stupně



Graf (4.1) znázorňuje dobu faktorizace náhodných polynomů stupně 500. Tyto polynomy jsou sestaveny z vzrůstajícího počtu faktorů přibližně stejného stupně  $(\frac{\deg(f)}{\text{faktorů}} \pm 1)$  s 64 bitovými koeficienty. Provedeny byly i testy faktorizace polynomů  $f$  takových, že  $f$  sestává z jednoho faktoru stupně  $\deg(f)/2$  a ostatních faktorů malého přibližně stejného stupně, nicméně průběh takového grafu je téměř totožný. Časová náročnost zpracování "asymetricky složených" polynomů také není nijak výrazně odlišná od graficky znázorněného případu.

Ukazuje se tedy, že pro takové náhodné polynomy je kombinační fáze faktorizace vzhledem k předchozím časově zanedbatelná a oba algoritmy pracují přibližně stejnou dobu. V. Shoup se v poznámkách zmiňuje, že implementace Zassenhausova algoritmu rychle zpracovává vstupy až do 30-40 modulárních faktorů, nicméně graf jasně ukazuje, že se vyrovná van Hoejiho metodě i s daleko větší množinou modulárních faktorů.

Obrázek 4.2: Tabulka faktorizace polynomů  $\mathbb{Z}[x]$

	Zassenhaus	van Hoeij
$P1$	2.2	2.128
$P2$	2.839	2.931
$P3$	5.834	5.887
$P4$	> 5000	35.6
$P5$	68.234	1.325
$P6$	2.402	2.404
$P7$	> 5000	23.55
$P8$		77.519
$S6$		1.329
$S7$		7.18
$S8$		98.254
$M12_5$		279.494
$M12_6$		519.629

Tabulka (4.2) zobrazuje časy faktorizace polynomů  $P1, \dots, P8, S6, S7, S8, M12_5, M12_6$  sesbíraných Paulem Zimmermannem and Markem van Hoeij, navržených speciálně pro testování faktorizačních algoritmů. Tyto představují zajímavější situace hlavně pro kombinační fázi faktorizace, tudíž zvýrazní časové rozdíly mezi porovnávanými algoritmy. V. Shoup popisuje polynomy na stránce:

<http://www.shoup.net/ntl/doc/tour-time.html>, kde jsou také ke stažení. V porovnání s výsledky V. Shoupa jsou mnou naměřené časy výrazně pomalejší, což je zřejmě

důsledek nevyužití knihovny GMP. I tak je velice zajímavý časový rozdíl u polynomu P4 faktorizovaného Zassenhausovou metodou, algoritmus úspěšně našel jeden faktor stupně 66, ale po následné nečekaně dlouhé době procházení nesprávných kombinací jsem jej ukončil při průchodu podmnožin velikosti 18. Z naměřených výsledků vyplývá, že pro náhodný polynom Zassenhausova metoda pracuje srovnatelně s van Hoejiho, ale v případě speciálně konstruovaných polynomů už je jasně vidět převládající rychlost nového algoritmu.

## 4.2 Nad konečným tělesem

Jádrem obou níže popsaných algoritmů je faktorizace už připravených bezčtvercových polynomů. Obě implementace volají pro tento úkol stejnou funkci `void SquareFree Decomp(vec_pair_ZZ_pX_long u, ZZ_pX f)`. Vzhledem k popisu algoritmu v [1] a tudíž předpokládané znalosti bude v následujícím textu vynechán.

### 4.2.1 Cantor-Zassenhaus

Funkce `void CanZass(vec_ZZ_pX factors, ZZ_pX ff)` představuje implementaci Cantor-Zassenhausovu algoritmu. Tento sestává ze tří hlavních fází:

**Bezčtvercová faktorizace** Vstupem je polynom  $f \in \mathbb{F}_p[x]$  stupně  $n$ , výstupem pak  $f_1, \dots, f_n \in \mathbb{F}_p[x]$  takové, že:

$$f = f_1 \cdot f_2^2 \cdot \dots \cdot f_n^n$$

**"Distinct-degree" faktorizace** Vstupem je bezčtvercový polynom  $f \in \mathbb{F}_p[x]$  stupně  $n$ , výstupem  $f^{[1]}, \dots, f^{[n]} \in \mathbb{F}_p[x]$  takové, že pro  $1 \leq d \leq n$  je  $f^{[d]}$  součinem  $r = n/d$  ireducibilních faktorů  $f$  stupně  $d$ . Motivací pro tento algoritmus je fakt, že pro libovolná nezáporná  $a, b$  je polynom  $x^{p^a} - x^{p^b} \in \mathbb{F}_p[x]$  dělitelný právě takovými ireducibilními polynomy z  $\mathbb{F}_p[x]$  jejichž stupně dělí  $a - b$ . Implementace využívá "baby step/giant step" techniku, kterou představili E.kartofel a V.Shoup(1995). Mějme  $B = \lfloor n/2 \rfloor$ ,  $l = \lfloor \sqrt{B} \rfloor$  a  $m = \lceil B/l \rceil$ .

1. Pro  $0 \leq i \leq l$  spočítej  $h_i = x^{p^i} \bmod f$
2. Pro  $1 \leq j \leq m$  spočítej  $H_j = x^{lj} \bmod f$
3. Pro  $1 \leq j \leq m$  spočítej

$$I_j = \prod_{0 \leq i < l} (H_j - h_i) \bmod f$$

4. Nastav  $f^{[1]}, \dots, f^{[n]}$  rovno 1 a pokračuj:
  - $f^* \leftarrow f$ ;
  - for  $j \leftarrow 1$  to  $m$  do
    - $\{g \leftarrow \text{NSD}(f^*, I_j); f^* \leftarrow f^*/g$
    - for  $i \leftarrow l-1$  down to 0 do
      - $\{f^{[l-j-i]} \leftarrow \text{NSD}(g, H_j - h_i); g \leftarrow g/f^{[l-j-i]}\}$
  - if  $f^* \neq 1$  then  $f^{[\text{deg}(f^*)]} \leftarrow f^*$

**"Equal-degree" faktorizace** Vstupem je polynom  $f \in \mathbb{F}_p[x]$  stupně  $n$  a číslo  $d$  takové, že  $f$  je součinem monických ireducibilních polynomů, všech stupně  $d$ . Výstupem je pak množina ireducibilních faktorů  $f$ . Algoritmus řešící tento problém představuje opakované volání následujících dvou kroků, dokud nezískáme očekávaný rozklad:

1. **Výpočet stopy** Zvol náhodný polynom  $r \in \mathbb{F}_p[x]$  stupně menšího než  $n$  a vypočítej  $g = T_d(r) \bmod f$ , kde pro  $k \geq 0$  definujeme  $T_k \in \mathbb{F}_p[x]$  jako:

$$T_k = \sum_{0 \leq i < k} x^{pi}$$

2. **Extrakce faktorů** Máme-li polynomy  $a, b \in \mathbb{F}_p[x]$  se stupni  $\text{deg}(a) > \text{deg}(b)$ , potom *minimálním polynomem*  $b$  modulo  $a$  myslíme monický polynom  $b'$  minimálního stupně, který splňuje  $b'(b) \equiv 0 \pmod{a}$ . Poslední fáze algoritmu používá následující extrakční metodu využívající takový minimální polynom. Buď polynom  $f$  vstup do "equal-degree" faktorizace a  $g$  polynom spočtený v předchozím kroku.

1. Najdi  $h$  minimální polynom  $g \bmod f$
2. Zavolej s  $f, g, h$  :
  - Zvol náhodné  $b \in \mathbb{F}_p$
  - Spočti  $h_1 = \text{NSD}((x+b)^{(p-1)/2}, h)$  a  $h_2 = h/h_1$  (pro náhodné  $b$  předpokládáme, že  $h_1, h_2$  jsou dělitelé  $h$  přibližně stejného stupně)
  - Spočti  $f_1 = \text{NSD}(h_1(h), f)$  a  $f_2 = f/f_1$
  - Rekurzivně zavolej s  $f \leftarrow f_1, \quad g \leftarrow (g \bmod f_1), \quad h \leftarrow h_1$
  - Rekurzivně zavolej s  $f \leftarrow f_2, \quad g \leftarrow (g \bmod f_2), \quad h \leftarrow h_2$

Implementace v NTL využívá elegantnější způsob dělení  $h$  na dva faktory, jak ukazují kroky 5 a 6 straně 33 v popisu Berlekampova algoritmu, který tuto metodu také využívá. Pro srovnání bylo záměrně zvoleno odpovídající značení.

Není zaručeno, že výstupem "equal-degree" faktorizace je pokaždé úplná faktorizace  $f$ , v takovém případě je nutno jej opakovat s nově zvoleným náhodným polynomem  $g$ . Nicméně pravděpodobnost, že se tak stane je rovna  $r^2/p$ , tudíž pro dostatečně velká  $p$  nijak znepokojující.

## 4.2.2 Berlekamp

NTL ve funkci `void berlekamp(vec_pair_ZZ_pX_long factors, ZZ_pX f)` obsahuje implementaci Berlekampovy Null-Space metody popsané v [6] s extrakční metodou využívající minimální polynom. Na začátku algoritmus provede bezčtvercovou faktorizaci a pro každý bezčtvercový polynom  $f$  stupně  $\deg(f) = n$  algoritmus pokračuje následovně:

1. Spočítej  $h = x^p \bmod f$
2. Sestroj  $n \times n$  matici :

$$Q = (h^0 \bmod f | h^1 \bmod f | \dots | h^{n-1} \bmod f) - I_n ,$$

kde každý polynom představuje sloupcový vektor, jehož souřadnice jsou tvořeny odpovídajícími koeficienty a  $I_n$  je  $n \times n$  matice identity

3. Pomocí Gaussovy eliminace uprav  $G$ , zjisti  $d$  dimenzi řešení matice  $G$
4. Zvol libovolné netriviální řešení  $g$  matice  $G$
5. Spočítej  $h$  minimální polynom  $g \bmod f$  a jeho kořeny  $\{a_0, a_1, \dots, a_r\}$
6. Zavolej s hodnotami  $\text{min}=0, \text{max}=r, f, g$ :
  - pokud  $\text{min}=\text{max}$  přidej  $f$  mezi faktory, jinak pokračuj
  - z kořenů  $\{a_{\text{min}}, \dots, a_{\text{max}}\}$  spočti  $\hat{h} = \prod_{i=\text{min}}^{\text{max}} (x - a_i)$
  - $a = \hat{h}(g) \bmod f$
  - $f_1 = \text{NSD}(a, f), f_2 = f/f_1$
  - $\text{mid}=(\text{min}+\text{max})/2$
  - rekurzivně volej s hodnotami:  $\{a_{\text{min}}, \dots, a_{\text{mid}}\}, f \leftarrow f_1, g \leftarrow g \bmod f_1$
  - rekurzivně volej s hodnotami:  $\{a_{\text{mid}+1}, \dots, a_{\text{max}}\}, f \leftarrow f_2, g \leftarrow g \bmod f_2$
7. Pokud je počet faktorů  $< d$  opakuj algoritmus od 4. kroku s jiným řešením  $G$  pro každý faktor stupně  $> 1$

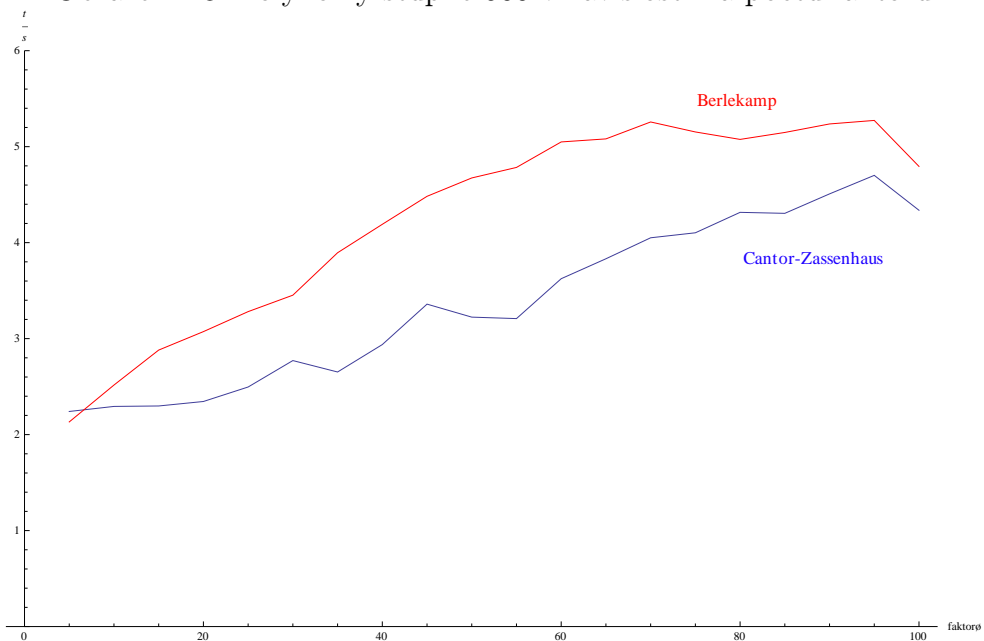
## Srovnání

Následující odhady jsou převzaty z [6]. Pro měření časové složitosti algoritmů budeme počítat aritmetické operace v  $\mathbb{Z}_p$ . Pro klíčovou operaci násobení budeme předpok-

ládat, že vynásobení dvou polynomů stupně  $d$  představuje  $O(M(d))$  operací, kde  $M(d) = d \log(d) \log(\log(d))$ . Což představuje odhad při použití rychlé Fourierovy transformace (FFT).

Berlekampův algoritmus provádí v nejhorším případě  $O(n^3 + M(n) \log(n) \log(p))$  operací. Nicméně pro malé počty faktorů  $r$  (což je typický případ pro náhodný vstup), řekněme  $r = O(n/\log(n))$  algoritmus provede pouze  $O(n^3 + M(n) \log(p))$  operací. Exponent 3 může být redukován asymptoticky rychlejšími (nicméně obecně nepraktickými) metodami na úpravu matic.

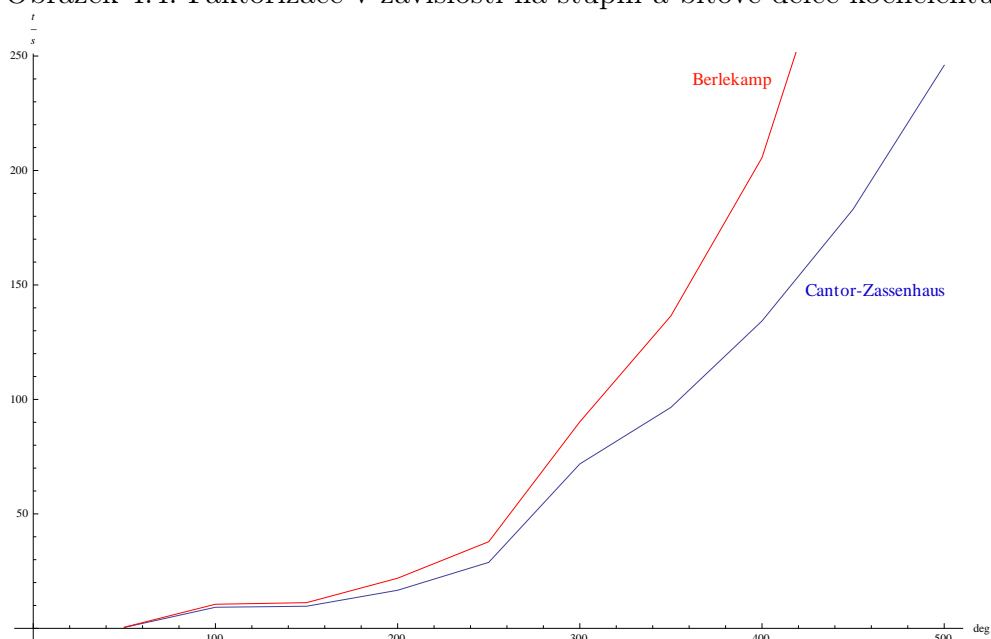
Obrázek 4.3: Polynomy stupně 500 v závislosti na počtu faktorů



Zaměříme-li se na dvě hlavní části Cantor-Zassenhausova algoritmu, potom "distinct-degree" faktorizační část využívá  $O(n^{2.5} + M(n) \log(p))$  operací, kde ovšem zahrnuté konstanty jsou poměrně malé a tudíž činí tuto metodu velmi praktickou. "Equal-degree" faktorizace používá s využitím techniky pro rychlý výpočet stopy v nejhorším případě  $O(n^2 + M(n) \log(n) \log(p))$  skalárních operací. I zde jsou zahrnuté konstanty dostatečně malé pro praktické použití. Poznamenejme ještě, že pro náhodný polynom vstupující do Zassenhausova algoritmu již do této fáze většinou postupují polynomy s jedním (v tom případě fáze vůbec neproběhne) nebo dvěma faktory, kdy je výše popsáný odhad redukován na  $(n^2 \log(n) + \log(p))$ . Ve výsledku je tedy implementace Cantor-Zassenhausova algoritmu využívá v nejhorším případě  $O(n^{2.5} + M(n) \log(n) \log(p))$  operací.

Co se reálných výsledků týče, graf (4.3) zobrazuje faktorizaci polynomů stupně 500 modulo 64 bitové prvočíslo. Zřetelně ukazuje, že ani rychle vzrůstající počet faktorů nemá výrazný vliv na dobu běhu faktorizace. Pro graf (4.4) jsou faktorizovány polynomy  $f$  rostoucího stupně a velikosti modulu  $p$  ve vztahu  $\lceil \log_2(p) \rceil = \deg(f) + 5$ . Je jasně patrné, že implementace Cantor-Zassenhausova algoritmu je výrazně rychlejší pro takové polynomy už od stupně 50 (je rychlejší i pro menší stupně v řádu desetin sekundy, což je ovšem vzhledem k měřítku grafu zanedbatelné) .

Obrázek 4.4: Faktorizace v závislosti na stupni a bitové délce koeficientů



# Literatura

- [1] RNDr. David Stanovský, Ph.D.: *Počítačová algebra 2008/09*, dostupné na <http://www.karlin.mff.cuni.cz/stanovsk/vyuka/palg.htm>.
- [2] PETER ROELSE: *Factoring High-degree Polynomials over  $F^2$  with Niederreiter's Algorithm on the IBM SP2*, Mathematics of Computation., Volume 68, Issue 226(April 1999), 869-880.
- [3] Donald Ervin Knuth: *The Art of Computer Programming, volume 2*, Addison-Wesley, Reading, Massachusetts, 1997.
- [4] Mark van Hoeij: *Factoring polynomials and the knapsack problem*, J. Number Theory, Volume 95, Issue 2(August 2002), 167-189.
- [5] Karim Belabas: *A relative van Hoeij algorithm over number fields*, J. Symbolic Comput., Volume 37, Issue 5(May 2004), 641-668.
- [6] Victor Shoup: *A New Polynomial Factorization Algorithm and its Implementation*, J. Symbolic Comput., Volume 20, Issue 4(October 1995), 363-397.
- [7] J. Abbott, V. Shoup, P. Zimmermann: *Factorization in  $\mathbb{Z}[x]$ : The Searching Phase*, Proceedings of the 2000 international symposium on Symbolic and algebraic computation (2000), 1-7.
- [8] Chee-Keng Yap: *Fundamental Problems in Algorithmic Algebra*, Oxford University Press, 1998.