Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Softwarového Inženýrství

Bakalářská práce

Knihovna pro násobení polynomů

Miloslav Brožek

Vedoucí práce: Ing. Ivan Šimeček Ph.D.

11. května 2015

Poděkování

Zde bych rád poděkoval panu Ivanu Šimečkovi za ochotu, konzultace a cenné rady, které mi poskytl.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 11. května 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Brožek, Miloslav. *Knihovna pro násobení polynomů.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato práce se zabývá návrhem knihovny pro násobení polynomů. Cílem této práce je vzájemné porovnání vybraných algoritmů mezi sebou a zároveň porovnání s násobením řídkých polynomů. Oborem porovnání je zejména rychlost algoritmů, a případná diskuse o příčinách této rychlosti. Všechny algoritmy použité pro tuto práci jsou podrobně analyzovány a jejich výhody či nevýhody jsou rozebrány. Součástí této práce je též implementace navržené knihovny. Při implementaci knihovny je kladen důraz na snížení režijních nákladů či jiné časové úspory, které jsou diskutovány. Práce si dále klade za cíl nalézt mez, do které se vyplatí použít struktury pro řídké polynomy (z hlediska rychlosti), a to jednak teoreticky, tak prakticky. Na závěr je implementován edukativní GUI program, sloužící ke zjišťování vybraných statistik z běhu algoritmů.

Klíčová slova Násobení polynomů, Karatsubův algoritmus, FFT, Rychlá Fourierova Transformace, Diskrétní Fourierova Transformace, Edukativní program, Knihovna, Optimalizace algoritmů.

Abstract

The purpose of this work is the design of a library for multiplication of polynomials. The aim of this work is the mutual comparison of chosen algorithms among themselves and also comparision with the multiplication of sparse polynomials. The scope of the comparison is especially speed of algorithms, and potential discussion on the causes of speed. All the algorithms used for this work are analyzed and their advantages and disadvantages are discussed. Part of this work is also implementation of designed library. During implementation, there is an emphasis on reducing overhead costs, or other time-savings that are being discussed. The work also aims to find the limit to which it is advantageous to use the structure for sparse polynomials (for acceleration), both theoretically and practically. At the end educational GUI program is implemented which serves to detect selected statistics from algorithms runs.

Keywords Polynomial multiplication, Karatsuba algorithm, FFT, Fast Fourier Transform, Discrete Fourier Transform, Educational program, Library, Algorithms optimalization.

Obsah

Seznam obrázků

Úvod

Cílem této práce je návrh knihovny pro rychlé násobení polynomů. V práci je vybráno několik algoritmů. U nich jsou popsány výhody a nevýhody a princip jejich funkčnosti.
Je zhotoven návrh celé knihovny (a to včetně modelů). Ta je dále implementována v jazyce C++ a za pomoci efektivní implementace algoritmů je přizpůsobena cílovému systému. Zhotovená knihovna je posléze otestována několika známými polynomy.
Dalším bodem je pak zhotovení GUI programu. Ten umí po zadání dvou polynomů oba polynomy vynásobit a vypsat jejich výsledek. Kromě této funkce dále umí zjistit různé statistiky o všech v práci vybraných algoritmech. Mezi statistiky patří například například počty násobení, sčítání, zanoření rekurze a další. Pro zhotovení tohoto programu byl vybrán programovací jazyk Java.

Kapitola 1

Návrh a analýza

V této kapitole se se seznámíme se základními pojmy. Dále je zde analýza aktuálních knihoven a návrh knihovny mé.

1.1 Obecné pojmy

V této sekci bylo čerpáno zejména z: [1], [5] a [8]

1.1.1 Polynom

Polynom je komplexní funkce komplexní proměnné, tedy p: $\mathbf{C} \to \mathbf{C}$, která má pro všechna $x \in \mathbf{C}$ funkční hodnotu $p(x)$ danou vzorcem

$$p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n,$$

kde $a_0, ...a_n$ jsou nějaká reálná nebo komplexní čísla, která nazýváme *koeficienty* polynomu. Jinými slovy: je-li funkce p polynomem, existuje přirozené číslo n a konstanty $a_0, ...a_n$ tak, že pro funkční hodnoty $p(x)$ platí uvedený vzorec pro všechna $x \in \mathbf{C}$.

1.1.2 Stupeň polynomu

Nechť polynom p má koeficienty $a_0, ...a_n$. Stupeň polynomu je největší index k takový, že $a_k \neq 0$. Pokud jsou všechny koeficienty nulové (nulový polynom), prohlásíme, že stupeň je roven -1.

1.1.3 Součin polynomů

Součin polynomů p a q je funkce u daná předpisem $u(x) = p(x)q(x)$ pro všechna $x \in \mathbf{C}$. Součin polynomů p a q značíme pq.

1.1.4 Grupa

Množinu G, na které je definovaná operace $\circ : G \times G \to G$ nazýváme grupou, pokud pro tuto operaci platí:

(1) $\forall x, y, z \in G : (x \circ y) \circ z = x \circ (y \circ z)$ (asociativní zákon).

(2) $\exists e \in G$, pro které platí $\forall x \in G : e \circ x = x \circ e = x$ (existence neutrálního/jednotkového prvku).

(3) $\forall x \in G : \exists y \in G : x \circ y = y \circ x = e$ (existence opačného/inverzního prvku y pro každý prvek x).

Pokud navíc platí

(4) $\forall x, y \in G : x \circ y = y \circ x$ (komutativní zákon).

pak grupu G nazýváme *komutativní grupou*. Z historických důvodů a z úcty k norskému matematikovi, který zpracoval teorii grup a bohužel zemřel mlád na zákeřnou nemoc ve věku 26 let, se komutativní grupa nazývá též *Abelova grupa*.

1.1.5 Těleso

Těleso je množina T se dvěma operacemi obvykle označovanými $+$ *tak, že* $T \times T \to T$ a $\cdot$ *tak, že* $T \times T \to T$, které mají následující vlastnosti:

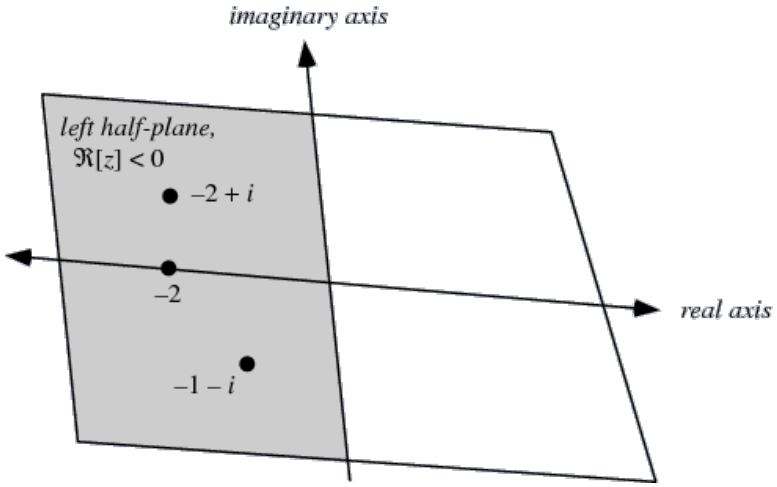
(1) T s operací „+“ je komutativní grupa. Neutrální prvek této grupy je označen symbolem 0.

(2) $T \backslash \{0\}$ s operací „$\cdot$“ je komutativní grupa. Jednotkový prvek této grupy se značí symbolem 1.

(3) Operace „+“ a „$\cdot$“ splňují distributivní zákon: $a \cdot (b + c) = a \cdot b + a \cdot c$.

1.1.6 Levá polorovina

Část komplexní roviny $z = x + iy$ s reálnou částí $\mathbf{R}[z] < 0$.



Obrázek 1.1: Grafická ukázka levé poloroviny (šedá barva). Převzato z [5].

1.1.7 Řídký čtverec polynomu

Řídký čtverec polynomu je taková mocnina polynomu $[P(x)]^2$, která má méně členů, než-li originální polynom $P(x)$.
Pro upřesnění: člen se počítá pouze za předpokladu, že má nenulový koeficient, tedy i přes to, že výsledný polynom bude mít vyšší stupeň polynomu, může mít méně členů.

1.2 Výběr několika známých polynomů

Pro otestování naivního algoritmu bylo vybráno několik známých polynomů:

1.2.1 Stabilní polynom

Přeloženo z [6].
Reálný polynom je stabilní pokud všechny jeho kořeny leží v levé polorovině.
Pro upřesnění zde uvádím několik příkladů stabilních polynomů:
Stabilní polynom *prvního řádu*: $x + a$
Stabilní polynom *druhého řádu*: $x^2 + ax + b$
Stabilní polynom *třetího řádu*: $x^3 + 2ax^2 + (a^2 + b)x + (ab - c)$

1.2.2 Nulový polynom

Polynom ve tvaru $p(x) = 0$, tedy polynom, který má všechny koeficienty nulové (jehož stupeň je tedy nulový). Pokud je nějaký jiný polynom vynásoben polynomem nulovým, výsledkem je opět polynom nulový.

1.2.3 Rényiho polynom

Přeloženo z [7].
Polynom ve tvaru $P_{28}(x) = (4x^4 + 4x^3 - 2x^2 + 2x + 1) * (-84x^{24} + 28x^{20} - 10x^{16} + 4x^{12} - 2x^8 + 2x^4 + 1)$, který má 29 výstupů a jehož čtverec jich má 28, což z něj dělá řídký čtverec polynomu.

1.2.4 Choudryho polynom

Přeloženo z [8]. Polynom ve tvaru $P_{17}(x) = (x^2 + 2x - 2) * (x^{15} + 4x^{12} - 8x^9 + 32x^6 - 160x^3 + 896)$, který má 18 výstupů a jehož čtverec jich má 17, což z něj dělá řídký čtverec polynomu.

1.2.5 Fermatovy polynomy

Přeloženo z[9] Fermatovy polynomy splňují rovnost $\mathbf{F}_n(1) = \mathbf{F}_n$ (nicméně vycházejí z Lucasova polynomu), kde $\mathbf{F}_n(x)$ jsou Fermatovy polynomy a $\mathbf{F}_n$ jsou pak Fermatova čísla $(2^n + 1)$. Příklady několika prvních Fermatových polynomů:

- $\mathbf{f}_1(x) = 3x$
- $\mathbf{f}_2(x) = 9x^2 - 4$
- $\mathbf{f}_3(x) = 27x^3 - 18x$
- $\mathbf{f}_4(x) = 81x^4 - 72x^2 + 8$
- $\mathbf{f}_5(x) = 243x^5 - 270x^3 + 60x$

Pro kontrolu, prvních 5 Fermatových čísel je: 3,5,9,17,33.

1.2.6 Fermatovy-Lucasovy polynomy

Přeloženo z [10].
Fermatovy-Lucasovy polynomy splňují rovnost $\mathbf{f}_n(1) = \mathbf{f}_n$ (nicméně vycházejí z Lucasova polynomu), kde $\mathbf{f}_n(x)$ jsou Fermatovy-Lucasovy polynomy a $\mathbf{f}_n$ jsou pak Fermatova-Lucasova čísla $(2^n - 1)$. Příklady několika prvních Fermatových-Lucasovích polynomů: 1,3,7,15,31.

- $\mathbf{F}_1(x) = 1$
- $\mathbf{F}_2(x) = 3x$
- $\mathbf{F}_3(x) = 9x^2 - 2$
- $\mathbf{F}_4(x) = 27x^3 - 12x$
- $\mathbf{F}_5(x) = 81x^4 - 54x^2 + 4$

Pro kontrolu, prvních 5 Fermatových-Lucasových čísel je: .

1.2.7 Fibonacciho polynomy

Přeloženo z [11].
Stejně jako dvě předchozí skupiny polynomů vychází i tato skupina z Lucasova polynomu. Obecná formule pro zapsání n-tého Fibonacciho polynomu vypadá následovně: $\mathbf{F}_n(x) = \frac{(x+\sqrt{x^2+4})^n - (x-\sqrt{x^2+4})^n}{2^n\sqrt{x^2+4}}$. Polynomy dále splňují rovnost $\mathbf{F}_n(1) = \mathbf{F}_n$, kde $\mathbf{F}_n(x)$ je n-tý Fibonacciho polynom a $\mathbf{F}_n$ je n-té Fibonacciho číslo. Dále tyto polynomy také splňují rekurentní vztah $\mathbf{F}_{n+1}(x) = x\mathbf{F}_n(x) + \mathbf{F}_{n-1}(x)$. Příklady prvních pěti Fibonacciho polynomů:

- $\mathbf{F}_1(x) = 1$
- $\mathbf{F}_2(x) = x$
- $\mathbf{F}_3(x) = x^2 + 1$
- $\mathbf{F}_4(x) = x^3 + 2x$
- $\mathbf{F}_5(x) = x^4 + 3x^2 + 1$

Pro kontrolu ještě přikládám prvních 5 Fibonacciho čísel: 1,1,2,3,5

1.2.8 Jacobsthalovy polynomy

Přeloženo z [12].
Tato skupina polynomů taktéž vychází z Lucasova polynomu. Polynomy zde splňují rovnost $\mathbf{J}_n(1) = \mathbf{J}_n$, přičemž $\mathbf{J}_n(x)$ je n-tý Jacobsthalův polynom a $\mathbf{J}_n$ je n-té Jacobsthalovo číslo. Zde je prvních pět Jacobsthalovo polynomů:

- $\mathbf{J}_1(x) = 1$
- $\mathbf{J}_2(x) = 1$
- $\mathbf{J}_3(x) = 2x + 1$
- $\mathbf{J}_4(x) = 4x + 1$
- $\mathbf{J}_5(x) = 4x^2 + 6x + 1$

Pro ověření ještě prvních 5 Jacobsthalovo čísel (číslováno od 1): 1,1,3,5,11

1.2.9 Pellovy polynomy

Přeloženo z [13].
Tyto polynomy, stejně jako předchozí polynomy, vycházejí z Lucasova polynomu. Platí pro ně následující rekurentní vztah: $\mathbf{P}_{n+2}(x) = 2x\mathbf{P}_{n+1}(x) + \mathbf{P}_n(x)$, kde prvními dvěma členy jsou $\mathbf{P}_0(x) = 0$ a $\mathbf{P}_1(x) = 1$. Prvních 5 Pellovo polynomů pak vypadá následovně:

- $\mathbf{P}_1(x) = 1$
- $\mathbf{P}_2(x) = 2x$
- $\mathbf{P}_3(x) = 4x^2 + 1$
- $\mathbf{P}_4(x) = 8x^3 + 4x$
- $\mathbf{P}_5(x) = 16x^4 + 12x^2 + 1$

Dále také platí rovnost $\mathbf{P}_n(x) = \mathbf{F}_n(2x)$, kde $\mathbf{P}_n(x)$ je n-tý Pellův polynom a $\mathbf{F}_n(x)$ je n-tý Fibonacciho polynom.

1.2.10 Věžové polynomy

Přeloženo z [14].

Věžový polynom je polynom ve tvaru $\mathbf{R}_{m,n}(x) = \sum_{k=0}^{min(m,n)} r_k x^k$, kde k je počet věží, které na sebe navzájem nevidí na šachovnici o velikosti $m * n$. Zde jsou příklady prvních 4 čtvercových (tedy $m = n$) věžových polynomů.

- $\mathbf{R}_1(x) = x + 1$
- $\mathbf{R}_2(x) = 2x^2 + 4x + 1$
- $\mathbf{R}_3(x) = 6x^3 + 18x^2 + 9x + 1$
- $\mathbf{R}_4(x) = 24x^4 + 96x^3 + 72x^2 + 1$

1.3 Specifikace a technologie

1.3.1 Specifikace knihovny

Jedinou funkcí, kterou tato knihovna musí podporovat je vynásobení dvou polynomů. Oba polynomy musí být diskrétní (tedy na celých číslech) stejně jako polynom výsledný (výhodou diskrétních hodnot je zejména lepší reálná přesnost a jednoduší testovatelnost, než-li u čísel reálných). Z toho plyne, že je nutné zajistit větší přesnost výsledků a to i pro metody, které pracují s čísly reálnými. Dále je snaha o co nejlepší výkonnost na daném systému, pro což byl také po diskusi s vedoucím práce vybrán programovací jazyk C++.

1.3.2 Navrhované jazyky

C++ - Tento jazyk byl vybrán pro implementaci knihovny. Nutným předpokladem (který je splněn) je, aby byl vybraný jazyk podporován na cílovém systému. Hlavní výhodou tohoto jazyka je jeho rychlost. Další z výhod je to, že ačkoliv jazyk není čistě objektový, objekty podporuje.

Java - Jazyk Java byl vybrán pro implementaci GUI programu. Výhodou je zejména jednoduchá implementace, a to zejména díky velkému množství standardních knihoven a datových struktur (a to včetně standardních knihoven pro grafické uživatelské rozhraní).
Výhodou obou jazyků je autorova je krátkodobá zkušenost s nimi. Oba samozřejmě dobře vyhovují všem požadavkům.

1.3.3 Formáty uložení koeficientů

Je taktéž nutné zvolit vhodné formáty pro uložení struktur použitých k výpočtu. Jedná se zejména o struktury pro uložení polynomů.
Forma koeficientů - Pro uložení polynomu v této formě byla zvolena jednoduchá forma a poli s koeficienty a parametru, ve kterém je uložena velikost onoho pole. Pole pak obsahuje koeficienty vždy na indexu, který zároveň udává mocninu neznámé, která tento koeficient násobí. Jako příklad lze uvést třeba polynom '$2x^4+x^2+7$', přičemž v parametru, který hlídá velikost bude uložena hodnota 5 a pole pak bude vypadat následovně: $pol = \{2, 0, 1, 0, 7\}$. Tento formát uložení se využije zejména pro vstup a posléze i pro výpočet naivního algoritmu či Karatsubova algoritmu.
Forma bod-hodnota - Pro uložení polynomu do této formy je podobně jako u formy předchozí vhodné pole. Zde je ovšem nutné ukládat dvojici hodnot (a ne pouze hodnotu jednu). K tomu se velmi dobře hodí standardní struktura jazyka C++ - 'std::complex<double>'. Velkou výhodou této struktury je, že jsou v ní implementovány základní operace na komplexních číslech, což se hodí i do některých vybraných algoritmů. Využití je pak zejména pro algoritmus FFT. V programu je tento formát implementován jako pole struktur.
Formát uložení řídkých struktur - Pro uložení řídkých polynomů (do řídkých struktur) byla zvolena forma uložení dvojice indexu a koeficientu. Pro tento formát bylo (na rozdíl od předchozí formy) zvoleno uložení do dvou polí, přičemž jedno obsahuje indexy do pole pro formu koeficientů (tedy obsahuje mocniny u proměnné) a druhé obsahuje hodnoty koeficientů. Dále je nutné držet si jeden parametr, ve kterém je uložena velikost obou těchto polí (ty mají samozřejmě stejnou velikost). V implementaci je tento formát použit jako dvě pole. Formát je částečně podobný formátu předchozímu, nicméně je dobré oba tyto formáty rozlišit, kvůli lepšímu pochopení problematiky (včetně správného překladu některých citací).

1.3.4 Cílový systém

Jako systém, pro který je tato knihovna optimalizována byl po diskusi s vedoucím práce zvolen server STAR (*star.fit.cvut.cz*). Ten je založen na architektuře „blade“ od firmy IBM. Obsahuje celkem 14 výpočetních žiletek (uzlů), které jsou k disposici. Uzly, pro které bude optimalizováno jsou uzly 9-12. Jejich specifikace vypadá následovně:

- Dva AMD Opteron 6C Processory (Model 2435) - 2.6GHz, 6MB L3 (celkem 12 výpočetních jader).
- 26GB RAM (PC2-6400 CL6 ECC DDR2 800 VLP RDIMM)
- 146GB 10k rpm SAS HDD

1.4 Návrh funkce pro násobení polynomů

1.4.1 Výpočetní struktura funkce

Ke správným a poměrně dobrým výsledkům by samozřejmě vedlo i jednoduché zvolení algoritmu s dobrou asymptotickou složitostí. Vzhledem k tomu že mají takové algoritmy obvykle vysokou režii, nemuselo by to mít zdaleka optimální výkonnost pro určité typy polynomů (například řídké polynomy či polynomy s malým stupněm). Z toho důvodu je vhodnější rozdělit výpočet na několik stavů:

- Výpočet polynomů s malým stupněm.
- výpočet polynomů se středně velkým stupněm.
- výpočet polynomů s vysokým stupněm.
- Výpočet řídkých polynomů.

Ke každému z výše zmíněných bodů je nutné přiřadit způsob výpočtu a hranice, pro které to do výše zmíněné kategorie spadá.

1.4.2 Parametry a návratová hodnota

Funkce musí samozřejmě umět nějakým způsobem přijmout a vynásobit dva polynomy. Ty budou uloženy v uživatelem alokovaných polích, vyplněny validními hodnotami a poslány funkci jako dva z jejich parametrů. Dalšími parametry jsou délky obou polynomů. Posledním parametrem je pak volitelný signál, určující, zda-li bude využito řídkosti algoritmu. Ten může nabývat následujících tří druhů hodnot.

- Záporné číslo - Pak nebude vlastnost řídkosti polynomu vůbec využita.
- Kladné číslo - Pak bude přímo počítáno metodou vybranou pro využití řídkosti polynomu.
- Nula - Pak bude otestováno, zda-li se vyplatí metodu pro násobení řídkého polynomu využít. Nutno podotknout že samotný test má složitost $O(n)$.

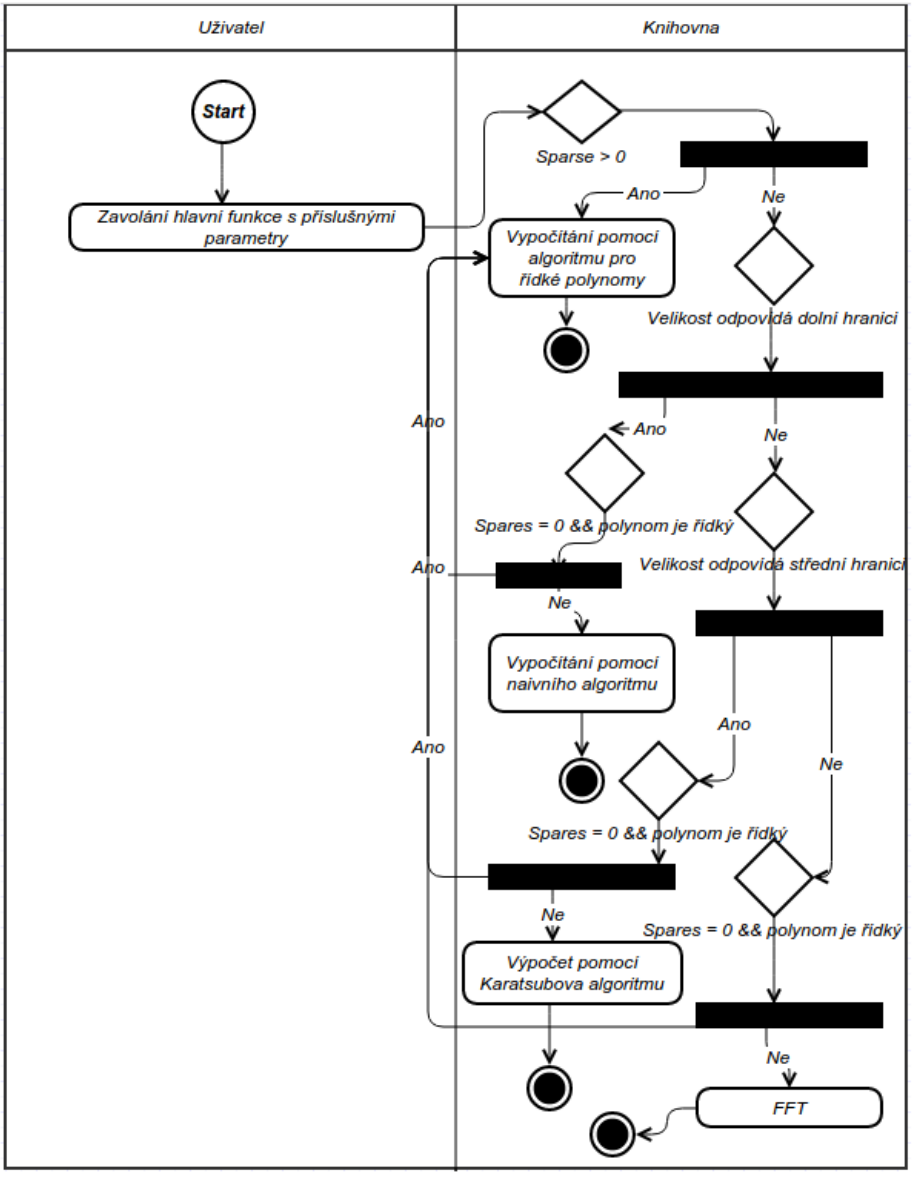
Hodnota tohoto parametru bude implicitně nastavena na nulu, pokud uživatel neřekne jinak.
Paralelní funkce dále disponuje parametrem pro určení počtu přidělených vláken.
Návratovou hodnotou funkce je pak ukazatel na pole diskrétních hodnot, které bude vyplněno výsledkem vynásobení příchozích polynomů. Pole bude alokováno přímo funkcí samotnou (pomocí operátoru *new*) na velikost danou vzorcem „$sSize + bSize - 1$“, kde *aSize* je velikostí prvního polynomu a *bSize* je velikostí polynomu druhého. Pro případ potřeby uživatele je k tomuto účelu zhotovena primitivní funkce, které umí velikost výsledného polynomu vypočítat ze zadaných parametrů (velikostí násobených polynomů).

1.4.3 Přidané funkce

Kromě funkce hlavní, na kterou je práce zaměřena, podporuje knihovna několik dalších funkcí, které může uživatel využít.
Vynásobit polynomy pomocí některého z vybraných algoritmů - Funkce, které dovolí uživateli přímo zavolat některý z vybraných algoritmů, aby jím mohl vynásobit dva polynomy.
Převody mezi řídkým formátem a formou koeficientů - Dvě funkce, díky nimž lze polynom převést z formy koeficientů do řídkého formátu a naopak.
Součet polynomů - Další z funkcí umí na vstupu přijmout dva polynomy a vrátit jejich součet. Všechny polynomy jsou ve formě koeficientů.
Odečtení polynomu - Další z funkcí umí na vstupu přijmout dva polynomy a vrátit polynom, který vznikne odečtením polynomu druhého od polynomu prvního. Všechny polynomy jsou ve formě koeficientů.
Vyhodnocení polynomu v bodě - Funkce, která umí přijmout polynom v podobě koeficientů a hodnotu proměnné a vrátit hodnotu, které vznikne po dosazení proměnné do polynomu. Například pro polynom '$6x^3 - 6x^2 + 6x$' a hodnotu proměnné x rovnu 3 by tato funkce vrátila číslo 126.
Vynásobení polynomů za využití řídkosti - Tato funkce přijme dva polynomy, první ve formě koeficientů a druhý v řídkém formátu a vynásobí je zvoleným algoritmem pro násobení řídkých polynomů.

1.5 Modely knihovny

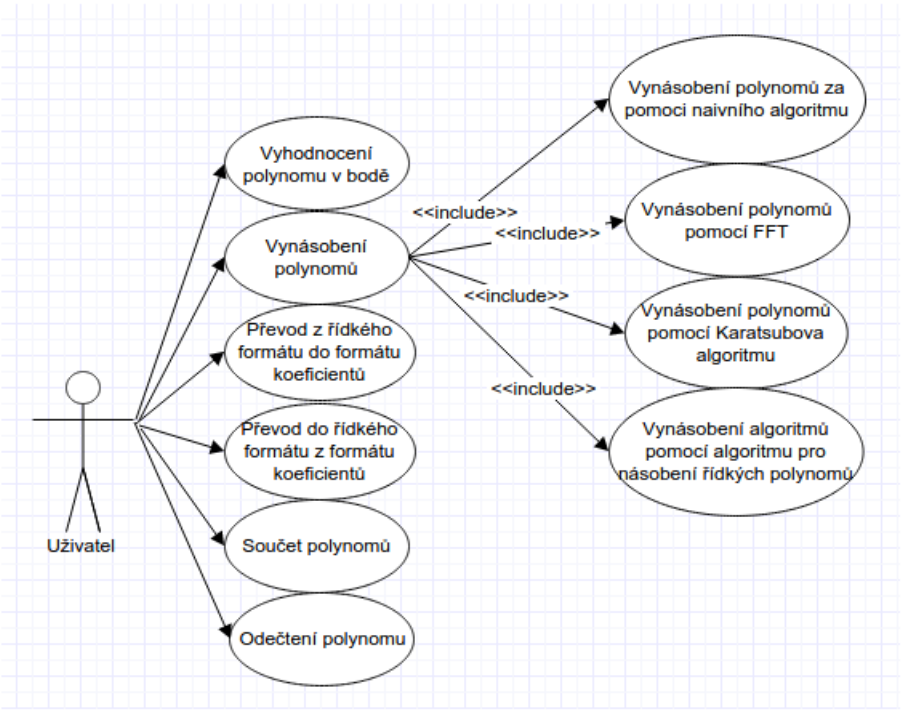
1.5.1 Diagram aktivit pro hlavní funkci



Obrázek 1.2: Business model hlavní funkce knihovny.

Jak je možné vidět z modelu na obrázku 1.2, hlavní funkce má kromě výpočtu samotného za úkol také rozhodnutí o tom, která funkce je pro vynásobení polynomů nejvhodnější. K tomu využije několik funkcí pomocných, které dokáží zjistit řídkost polynomu a s pomocí naměřených hodnot a velikosti polynomu dokáží rozhodnout, který algoritmus se vyplatí využít.

1.5.2 Diagram užití knihovny



Obrázek 1.3: Diagram užití knihovny.

Knihovna by měla (dle diagramu 1.3) kromě hlavních funkce obsahovat i několik přidaných funkcí (viz podsekce 1.4.3), které si uživatel může přímo zavolat. Jak lze z modelu vidět, sama hlavní funkce využívá funkce pro násobení polynomů (které se však nemusí nutně volat za pomoci funkce hlavní).

1.6 Rešerše knihoven

Účelem této sekce je projít a analyzovat několik knihoven, které se zaměřují na násobení polynomů. Nejdůležitější je zjistit, jaké algoritmy používají a po konzultaci s vedoucím pak vybrat jejich vhodné zástupce pro implementaci knihovny.

1.6.1 FLINT

FLINT (Fast Library for Number Theory) je knihovna v jazyce C++, která podporuje velké množství matematických funkcí, zejména z teorie čísel. Mezi nimi i funkce pro násobení polynomů. K tomu knihovna využívá následujících algoritmů:

- Naivní algoritmus - viz sekce 2.1
- Karatsubův algoritmus - viz sekce 2.2
- Kroneckerova substituce - viz podsekce 2.5.1
- Schönhageho-Strassenův algoritmus - viz podsekce 2.5.2

1.6.2 zn_poly

Jedná se o otevřenou softwarovou knihovnu, navrženou a implementovanou Davidem Harveyem z Newyorské Univerzity. Knihovna se specializuje na násobení polynomů v tělese. Zde byly využity tyto algoritmy:

- Kroneckerova substituce - viz sekce 2.5.1
- Rychlá Fourierova Transformace (dále jako FFT) - viz sekce 2.3

1.6.3 Givaro

Givaro je knihovna napsaná v jazyce C++ od francouzských autorů. Knihovna podporujíce aritmetiku a algebraické operace, jako například operace s maticemi, vektory a polynomy. Mezi operacemi pro polynomy je například faktorizace nebo násobení polynomů, pro který byl použit následující algoritmus:

- Karatsubův algoritmus - viz sekce 2.2

1.6.4 simplertimes

Tato knihovna je napsána doktorem Davidem Eisenstatem v jazyce Java. Knihovna podporuje násobení koeficientů s pouze kladnými koeficienty z oboru celých čísel do velikosti až 2^{27} a zaměřuje se zejména na rychlost výpočtu. K tomu využívá následující algoritmus:

- FFT - viz sekce 2.3

1.6.5 NumPy

NumPy je knihovna pro jazyk Python, podporující velké množství matematických funkcí, mezi kterými je i násobení polynomů. Pro násobení polynomů je využit následující algoritmus:

- FFT - viz sekce 2.3

1.6.6 NTL

NTL (Number Theory Library) je matematická knihovna podporující pokročilé matematické metody v jazyce C++. Mezi nimi je i násobení polynomů, pro které je použit tento algoritmus:

- FFT - viz sekce 2.3

1.6.7 GMP

Jedná se o knihovnu v jazyce C++, která se zaměřuje na aritmetiku s vysokou přesností. GMP podporuje přes 150 aritmetických funkcí, mezi nimiž je i násobení polynomů. To je implementováno pomocí těchto algoritmů:

- Karatsubův algoritmus - viz sekce 2.2
- Toom-Cookova metoda - viz podsekce 2.5.3

1.6.8 Závěr

Knihovnou, která nejlépe odpovídá mým představám pro návrh knihovny je část knihovny "FLINT", které se zaměřuje na polynomy. Knihovnou jsem z části čerpal inspiraci. Výhodou zde pak bude zejména možnost násobení polynomů s využitím řídkosti.

1.7 Návrh GUI programu

1.7.1 Specifikace

Cílovou platformou programu jsou zejména operační systémy na bázi Linuxu. Pro tyto účely byl vybrán jazyk Java. Program samotný musí podporovat snadné zadání dvou zdrojových polynomů a jejich následné výpočet. To vše za podpory navržené knihovny. Dále musí umět říci dobu výpočtu a pro všechny algoritmy (naivní, Karatsubův, FFT a řídké struktury) pak vyčíslit několik statistik, kterými jsou počty násobení, sčítání/odčítání, počet rekurzí, maximální zanoření, režie sčítání, alokace paměti a počet zápisů (to vše pro základní verze algoritmů, nikoliv upravené). Nakonec musí program umět zjistit čas, po který trvala doba výpočtu.

1.7.2 Návrh programu

Program by se měl skládat ze tří hlavních částí:

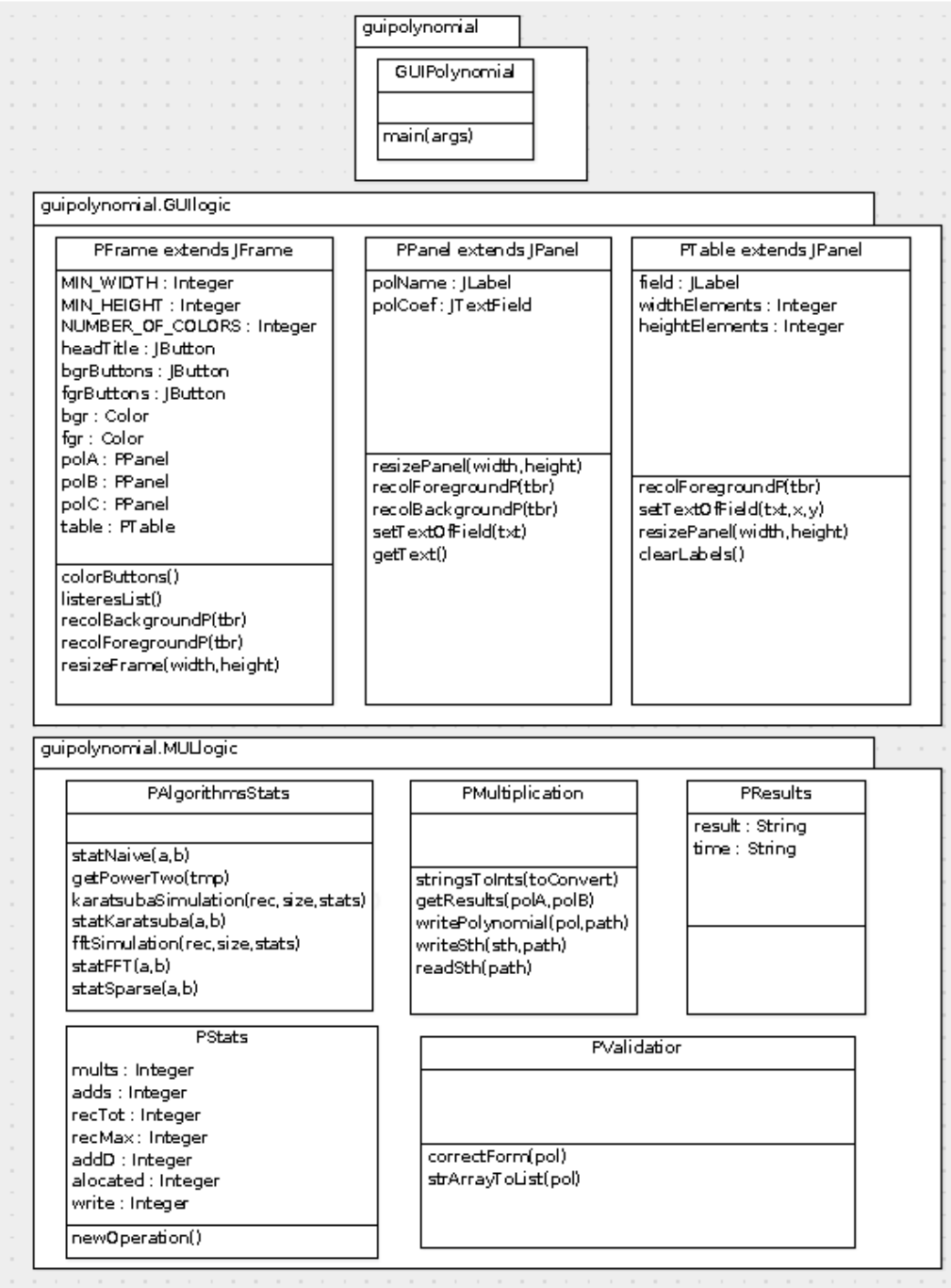
- ✪ Tlačítka, které má sloužit k uvedení programu do chodu
- ✪ Polí na vyplnění polynomů a tabulku na vyplnění statistik.
- ✪ Většinu plochy programu by měli zabírat pole na polynomy a tabulka.

Výškové rozdělení bylo tedy zvoleno následujícím způsobem:

- **Tlačítko** - 15%
- **Pole pro polynomy** - $3 \times 15\%$
- **Tabulka** - 30%

Spodních 10% bylo necháno pro volný prostor, aby nedocházelo k ořezávání textu. Jelikož není potřeba, aby bylo tlačítko roztaženo přes celou šířku programu, může se přenechat nahoře prostor pro nějaké přidané funkce (jako například obarvení pozadí či textu).
Dále je vhodným krokem udělat všechny tyto elementy dynamicky - tedy při změně velikosti programu by se měla změnit velikost všech těchto elementů.
Samozřejmě je také vhodné mít nějaký jednotný vzor pro zadávání polynomů. Pro to byly jednoduše zvolené koeficienty oddělené čárkami. Stejný vzor pak bude mít i polynom výsledný.
K propojení s knihovnou byla vybrána třída `Process`, která umí obsloužit spustitelný program. K tomu je vhodné i vytvoření jednoduchého program v jazyce C++, který obsluhuje navrženou knihovnu a zároveň komunikuje s GUI programem (díky tomu lze získat přesněji statistiky, jakými je například čas běhu výpočtu a podobně).

1.7.3 Návrh tříd programu

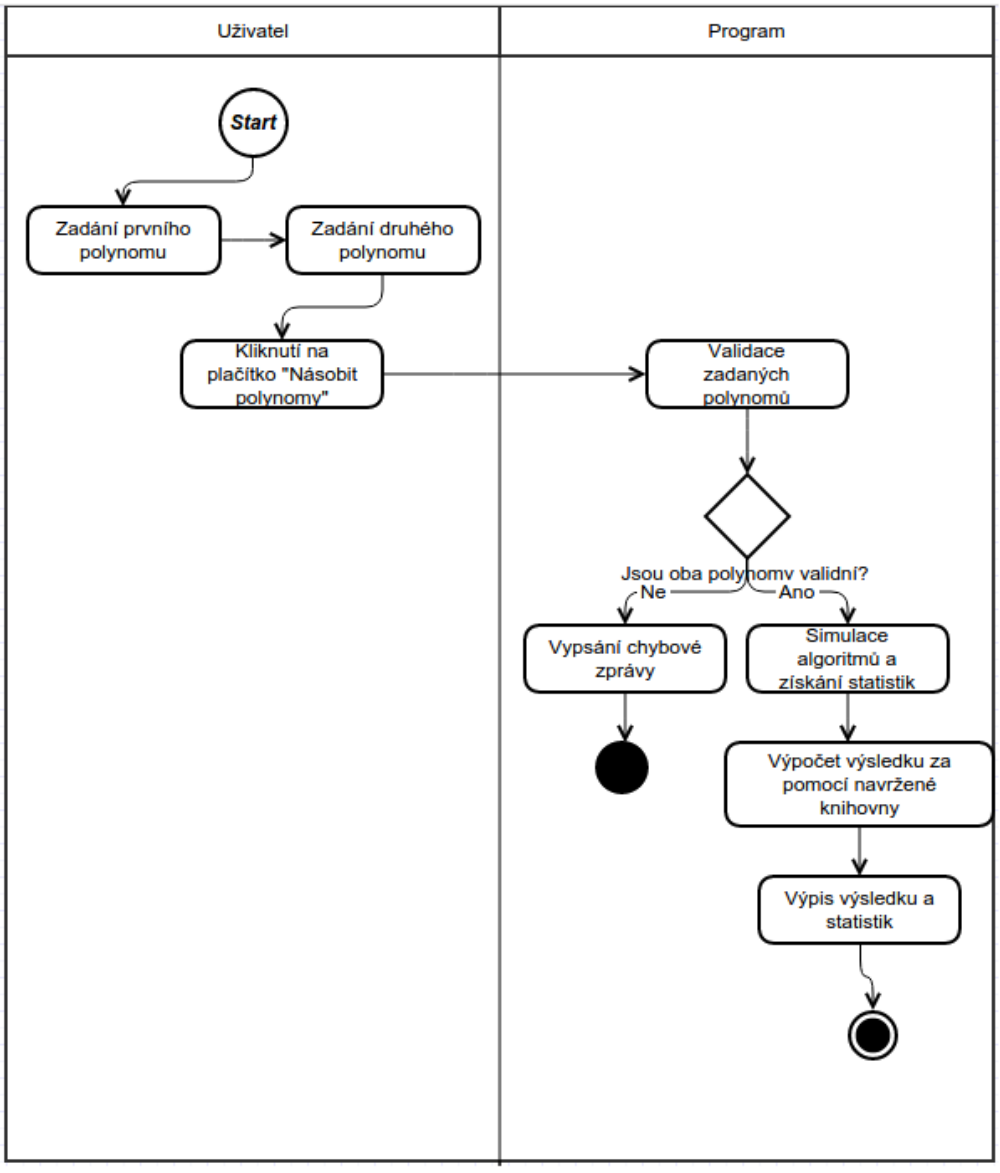


Obrázek 1.4: Návrh tříd pro GUI program.

Z návrhu 1.4 lze vidět, že kromě hlavní třídy, která se stará pouze o korektní zavolání je program rozdělen na dva balíčky. První z nich je balíček **GUIlogic**, který se stará zejména o grafiku programu. Skládá se ze tří tříd. Nejdůležitější z nich je třída `PFrame`, která zajišťuje veškerou logiku grafiky a uchovává v sobě všechny grafické komponenty. Třídy `PPanel` a `PTable` jsou pak pouze soubory komponent a vlastností. Jelikož jsou ale obě tyto třídy využity několikanásobně, umožní jejich využití přehlednost programu.
Balíček **MULlogic** se pak stará o logiku běhu programu. Prvními třídami balíčku jsou třídy `PResults` a `PStats`, které slouží jako struktury pro uchovávání statistik. Třetí třídou je pak `PValidatior`, která disponuje pouze dvěma statickými metodami. Její úkol je jednoznačný - převést vstup do vhodné formy a provést na něm kontrolu validity. Další třídou je třída `PAlgorithmsStats`, jež obsahuje několik metod, které dokáží simulovat průchod vybranými algoritmy (čímž získají požadované statistiky). Nakonec je to třída `PMultiplication`, která se stará zejména o komunikaci s navrženou knihovnou (respektive s programem, který ji obaluje).

1.7.4 Diagram aktivit programu

Na diagramu 1.5 lze vidět, že průchod programem je vcelku neměnný (s výjimkou špatného vstupu od uživatele). V návrhu se samozřejmě jedná o proces zadání a výpočtu - ať už proces skončí úspěšně či neúspěšně, nebude ukončen program samotný, nýbrž pouze proces samotného výpočtu.



Obrázek 1.5: Diagram aktivit GUI programu.

Kapitola 2

Algoritmy

Po diskusi s vedoucím práce byly (s přihlédnutím k rešerši knihoven) pro běžný výpočet zvoleny následující algoritmy:

- Naivní algoritmus
- Karatsubův algoritmus
- FFT

Ty budou v této kapitole popsány.

2.1 Naivní algoritmus

Tento algoritmus vychází z triviálního matematického popisu násobení polynomů, a to $p(x,y) = \sum_{j=0}^{n} x^j b_j * \sum_{i=0}^{m} x^i a_i = \sum_{i=0}^{m+n} (\sum_{j=0}^{i} a_j b_{i-j}) x^i$, tedy vynásobení všech prvků jednoho polynomu všemi prvky polynomu druhého. Již z tohoto popisu se dá vyčíst, že je asymptotická složitost tohoto algoritmu $O(nm)$. Pseudokód algoritmu vypadá asi následovně:

```
1: for i ← 0 to n do
2:     for j ← 0 to m do
3:         p[i + j]+ = a[i] ∗ b[j]
4:     end for
5: end for
```

Jak je z pseudokódu vidno, není zapotřebí zvlášť mnoho režie a už vůbec žádná alokace programového prostoru navíc.

2.2 Karatsubův algoritmus

Jedná se o algoritmus, který využívá techniky *Rozděl a panuj*, kterou zřejmě poprvé použili v roce 1962 pánové Karatsuba a Olfman [3]. Metoda byla původně vymyšlena pouze pro násobení klasických dekadických čísel, což je ovšem jednoduše převeditelné na problematiku polynomů. Algoritmus vychází z faktu, že násobek čísel (tedy i polynomů) - $a * b$ - lze beze změny výsledku rozložit na $a_L * b_L + a_L * b_U + a_U * b_L + a_U * b_U$, kde index L značí dolní polovinu čísla (tedy dolní polovinu polynomu) a index U pak horní polovinu čísla (tedy horní polovinu polynomu). S pomocí této znalosti se pak dá vzorec upravit do podoby, která je pro počítač vhodnější:

$$\begin{gathered} ab = \\ a_L b_L + a_L b_U x^{n/2} + a_U b_L x^{n/2} + a_U b_U x^n = \\ a_L b_L + (a_L b_U + a_U b_L) x^{n/2} + a_U b_U x^n = \\ a_L b_L + (a_L b_U + a_U b_L + a_L b_L - a_L b_L + a_U b_U - a_U b_U) x^{n/2} + a_U b_U x^n = \\ a_L b_L + (a_L (b_L + b_U) + a_U (b_L + b_U) - a_L b_L - a_U b_U) x^{n/2} + a_U b_U x^n = \\ a_L b_L + ((a_L + a_U)(b_L + b_U) - a_L b_L - a_U b_U) x^{n/2} + a_U b_U x^n \end{gathered}$$

Tato forma má tu výhodu, že dva její členy ($a_L b_L$ a $a_U b_U$) se vyskytují ve vzorečku dvakrát, tudíž si je můžeme zapamatovat a počítat tak pouze tři členy místo původních pěti. Ve výsledku tak bude výpočet provádět o něco méně operací násobení polynomů, za což se zaplatí příbytkem operací sčítání polynomů. Operace sčítání polynomů je ovšem několikanásobně rychlejší, nežli operace násobení polynomů, a tak se nám tato 'výměna' vyplatí (viz [2]). Algoritmus pracuje dobře pouze pro polynomy se stupněm 2^k (kde k je celé číslo), proto je před vstupem do polynomu nutná kontrola a případná úprava. Pokud bychom chtěli tuto techniku zapsat jako pseudokód, vypadalo by to zhruba následovně:

```
1:  if aSize = 1 then
2:      res[0] = polA[0] * polB[0]
3:      return
4:  end if
5:  a_L ← lower(polA)
6:  a_U ← upper(polA)
7:  b_L ← lower(polB)
8:  b_U ← upper(polB)
9:  r_L ← recKaratsuba(a_L, b_L)
10: r_U ← recKaratsuba(a_U, b_U)
11: r_LU ← recKaratsuba((a_L + a_U), (b_L + b_U))
12: res ← r_L + (r_LU − r_L − r_U)x^{n/2} + r_U x^n
```

Asymptotickou složitost pak lze určit z rekurentního vzorce, který je možné vypozorovat z pseudokódu. Řádky $5 - 8$ mají složitost n stejně, jako řádek 12.

Dále je zde režijní konstanta (například řádek 1). Nakonec to nejdůležitější - zanoření, které má poloviční složitost na řádcích $9 - 11$. Z toho nám vychází rekurzivní vzorec: $T(n) = 3t(n/2) + Kn + C$. Asymptotická složitost pro tento vzorec je $O(n^{log_2 3})$, tedy $O(n^{1,58496})$.
Zde je ještě tabulka porovnání sčítání a násobení skalárů, mezi klasickým naivním algoritmem (NA) a Karatsubovým algoritmem (KA):

Porovnání sčítání a násobení skalárů NA a KA				
Stupeň polynomů	NA sčítání	KA sčítání	NA násobení	KA násobení
1	1	0	1	1
2	4	7	4	3
4	16	35	16	9
8	64	133	64	27
16	256	455	256	81
32	1 024	1 477	1 024	243
64	4 096	4 655	4 096	729
128	16 384	14 413	16 384	2 187
256	65 536	44 135	65 536	6 561
512	26 2144	134 197	262 144	19 683
1 024	1 048 576	406 175	1 048 576	59 049
2 048	4 194 304	1 225 693	4 194 304	177 147
4 096	16 777 216	3 691 415	16 777 216	531 441
8 192	67 108 864	11 102 917	67 108 864	1 594 323
16 384	268 435 456	33 366 095	268 435 456	4 782 969
32 768	1 073 741 824	100 212 973	1 073 741 824	14 348 907

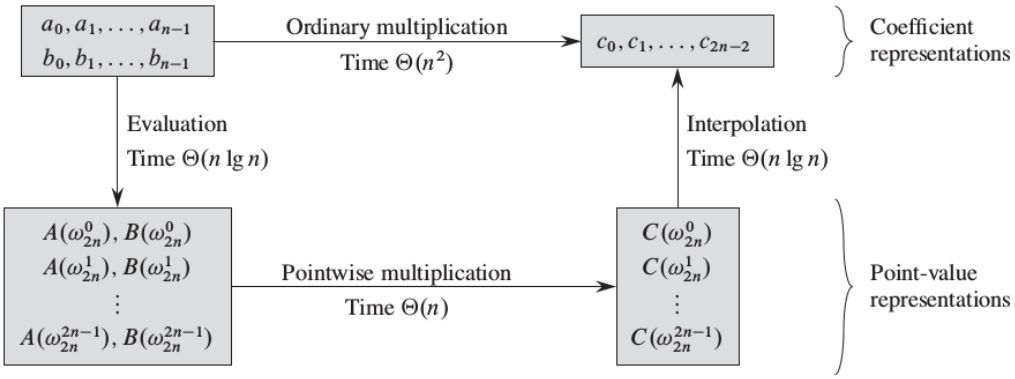
Tyto hodnoty byly získány pomocí funkcí, které simulovaly běh algoritmů pro požadované hodnoty.

V tabulce lze vidět celkové počty operacích sčítání a násobení skalárů naivního algoritmu oproti algoritmu Karatsubovu. Ačkoli je z počátku počet sčítání Karatsubova algoritmu vyšší, dostane se v celku rychle počet sčítání algoritmu naivního. V násobení je Karatsubův algoritmus lepší téměř ihned.

2.3 Rychlá Fourierova Transformace

2.3.1 Princip fungování

Přeloženo z [2].
Přímá metoda násobení dvou polynomů má asymptotickou složitost $O(n^2)$, pokud jsou polynomy ve formě koeficientů, ale pouze složitost $O(n)$, pokud jsou ve formě bod-hodnota. My ovšem dokážeme polynomy ve formě koeficientů vynásobit s asymptotickou složitostí $O(n\ log(n))$ tím, že je převedeme mezi těmito dvěma reprezentacemi.



Obrázek 2.1: Postup násobení pomocí FFT. [2]

Přeloženo z [4].
K transformaci mezi formou koeficientů a formou bod-hodnota se použije Diskrétní Fourierova Transformace (dále jako DFT) a inverzní DFT pro převod nazpátek. Jedná se o techniku rozděl a panuj, založenou na vlastnostech komplexního kořene jedničky ($\sqrt{-1}$). Na rozdíl od formy koeficientů ($A(x) = \sum_{j=0}^{n} a_j x^j$), se členy formy bod-hodnot skládají z dvojic $\{(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)\}$.
Metoda FFT původně vznikla pro zpracování signálů. Zde slouží k převodu z časové reprezentace na reprezentaci frekvence. Pro násobení polynomů je pak postup následující (viz. obrázek výše):

1. Převedení obou polynomů (ve formě koeficientů) na velikosti 2^k, kde k je celé číslo a zároveň je 2^k vyšší či stejné, než-li stupeň výsledného polynomu.

2. Vyhodnocení (*evaluation*) obou polynomů pomocí DFT (tedy převedení z formy koeficientů do formy bod-hodnota).

3. Vynásobení obou polynomů (ve formě bod-hodnota).

4. Interpolace (*interpolation*) výsledného polynomu pomocí inverzní DFT (tedy převedení z formy bod-hodnota do formy koeficientů).

Přeloženo z [4].
Komplexní kořen jedničky - Komplexní n-tý kořen jedničky je komplexní číslo ω takové, že $\omega^n = 1$. Je přesně n komplexních n-tých kořenů jedničky

$$e^{2\pi ik/n} \text{ pro k=0,1,...,n,}$$

kde $e^{ui} = cos(u) + i\ sin(u)$ (přičemž u reprezentuje úhel v radiánech). V

tomto případě tedy: $e^{2\pi ik/n} = cos(\frac{2\pi k}{n}) + i\ sin(\frac{2\pi k}{n})$ a dále je jasné, že $(e^{2\pi ik/n})^n = cos(2\pi k) + i\ sin(2\pi k) = 1$.
Přeloženo z [4].
Hlavní n-tý kořen jedničky - Hlavním n-tým kořenem jedničky nazýváme hodnotu $\omega_n = e^{\frac{2\pi i}{n}}$. Všechny ostatní komplexní n-té kořeny jedničky jsou mocninou této hodnoty. Těchto n komplexních n-tých kořenů jedničky $(\omega_n^0, \omega_n^1, ..., \omega_n^{n-1})$ formují grupu pod operací násobení, se stejnou strukturou jako $(\mathbb{Z}_n, +)$ modulo n. Díky této znalosti se nám odkrývá několik důležitých vztahů:

- $\omega_n^n = \omega_n^0 = 1$
- $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k)\ modulo\ n}$
- $\omega_n^{-1} = \omega_n^{n-1}$

Přeloženo z [4].
Lemma o zrušení - Toto lemma nám říká, že $\omega_{dn}^{dk} = \omega_n^k$. Pro důkaz nemusíme jít daleko:

$$\omega_{dn}^{dk} = (e^{2\pi i/dn})^{dk} = (e^{2\pi i/n})^k = \omega_n^k.$$

Pro každé celé číslo $n > 0$ platí, že $\omega_n^{n/2} = \omega_2 = -1$.
Přeloženo z [4].
Lemma o půlení - Pokud $n > 0$ je sudé, pak čtverec n komplexních n-tých kořenů jedničky jsou $n/2$ komplexních $n/2$-tých kořenů jedničky. K důkazu lze využít lemma o zrušení:

$$(\omega_n^k)^2 = \omega_{n/2}^k \text{ (pro všechna kladná } k).$$

Pokud tedy umocníme veškeré komplexní n-té kořeny jedničky, pak každý $n/2$-tý kořen dostaneme právě dvakrát:

$$(\omega_n^{(k+n)/2})^2 = \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = (\omega_n^k)^2,$$

tedy ω_n^k a $\omega_n^{(k+n)/2}$ mají stejný čtverec.
Přeloženo z [4].
Lemma o součtu - Pro každé celé číslo $n \geq 1$ a celé nenulové číslo k, které není dělitelné číslem n platí: $\sum_{j=0}^{n-1}(\omega_n^k)^j = 0$. K důkazu použijeme výpočet geometrické řady $(\sum_{j=0}^{n-1} x^j = \frac{x^n - 1}{x^n})$:

$$\sum_{j=0}^{n-1}(\omega_n^k)^j = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} = \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} = \frac{1 - 1}{\omega_n^k - 1} = 0$$

Přeloženo z [4].
Diskrétní Fourierova Transformace - DFT je vektor y_k, který definujeme pro $k = 0, 1, ..., n-1$, jako: $DFT_n(a) = y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$ (kde a je vektor koeficientů - $a = (a_0, a_1, ..., a_{n-1})$).

2.3.2 Algoritmus

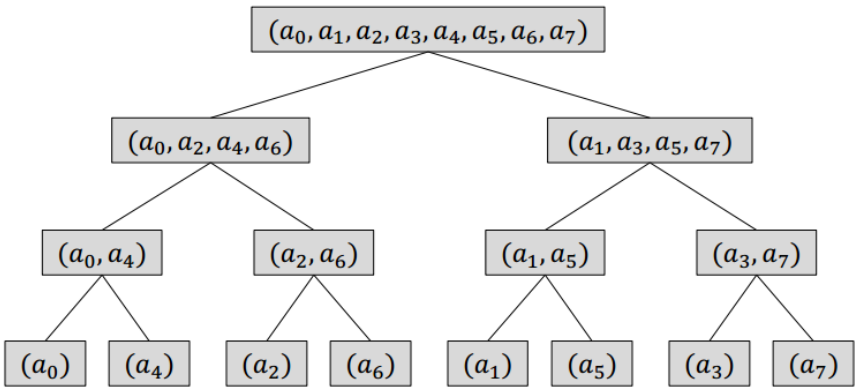
Samotný algoritmus FFT se skládá ze dvou částí - funkce hlavní, která se stará o celý průběh algoritmu (dle bodů zmíněných nahoře) a rekurzivní funkce, která hledá DFT. Pseudokód pro hlavní funkci by vypadal zhruba následovně:

```
1:  degree ← getBetterPowerTwo((getMax(aSize, bSize) * 2) − 1)
2:  polA.resize(degree)
3:  polB.resize(degree)
4:  complex ω_n(cos((2π)/degree), sin((2π)/degree))
5:  complex resA = DFT(polA, degree, ω_n)
6:  complex resB = DFT(polB, degree, ω_n)
7:  for i ← 0 to degree do
8:      resFFT[i] = resA[i] * resB[i]
9:  end for
10: complex ω_n^-1 = pow(ω_n, −1)
11: resFFT = DFT(resFFT, degree, ω_n^-1)
12: for i ← 0 to degree do
13:     resFFT[i].real()/ = degree
14: end for
15: for i ← 0 to resSize do
16:     res[i] = resFFT[i].real()
17: end for
```

Na řádcích $1-3$ se vstupní polynomy upraví na stejnou velikost (která musí být tvaru 2^k, kde k je celé číslo). Na řádku 4 je pak ω_n, která se využije v rekurzivním algoritmu DFT ($\omega_n = e^{2\pi i/n} = cos(\frac{2\pi}{n}) + icos(\frac{2\pi}{n})$, viz. podsekce 2.3.1). Řádky $5-6$ pak zajišťují převedení z formy koeficientů do formy bod-hodnota. Další tři řádky zajišťují násobení, ve formě bod-hodnota (lze vidět, že je asymptotická složitost vynásobení ve formě bod-hodnota $O(n)$). Dále pak je nutné interpolovat výsledný polynom zpět do formy koeficientů. To lze udělat tak, že na výsledný polynom pustíme stejný algoritmus, který byl použit na převod do formy bod-hodnota (řádek 11) s rozdílem, že je místo původního ω_n použita inverzní verze této hodnoty (řádek 10). Na závěr jsou pak hodnoty vyděleny velikostí n (řádek 13). Na řádku 16 musí být pro naše účely provedena nějaká aproximace, protože celý výpočet je proveden v reálných číslech, kde může vzniknou malá chyba, která by při jednoduché konverzi mohla vytvořit chybu s rozdílem až 1. I přes tuto aproximaci zde však existuje hrozba, že by výsledek mohl být chybný - to by ovšem musela vzniknout chyba zaokrouhlení při násobení reálných čísel větší než-li jedna polovina.
Druhou důležitou funkcí je rekurzivní funkce DFT, která musí umět převést polynom mezi oběma jeho reprezentacemi. Zapsáno pseudokódem vypadá tato funkce zhruba následovně:

```
1:  if polSize = 1 then
2:      return polA;
3:  end if
4:  degree/ = 2
5:  polEven ← (polA_0, polA_2, ..., polA_{n-2})
6:  polOdd ← (polA_1, polA_3, ..., polA_{n-1})
7:  ω_{n/2} ← ω_n * ω_n
8:  y_e ← DFT(polEven, degree, ω_{n/2})
9:  y_o ← DFT(polOdd, degree, ω_{n/2})
10: ω ← 1
11: for i ← 0 to degree do
12:     res[i] ← y_e[i] + y_o[i] * ω
13:     res[k + n/2] ← y_e − y_o * ω
14:     ω ← ωω_n
15: end for
16: return res
```

Řádek 8 bychom si mohli vyjádřit jako $y_e = polA_e(\omega_{n/2}^k) = polA_e(\omega_n^{2k})$, podobně, jako řádek 9 - $y_o = polA_o(\omega_{n/2}^k) = polA_o(\omega_n^{2k})$. Dále bylo zkrácena iterace (na řádcích $11-15$) na polovinu, díky využití faktu, že $-\omega_n^k = \omega_n^{k+n/2}$. Na řádku 14 se pak iterativně počítá ω_n^k.



Obrázek 2.2: Průchod DFT pro velikost polynomu 2^3. [4]

Z obrázku 2.2 si lze povšimnout, že maximální zanoření je logaritmické a součet operací v celé úrovni větví je lineární. Asymptotická složitost je tedy $O(n\ log(n))$.
I tento algoritmus má samozřejmě svoje nevýhody. První z nich je, že i přes

nejpříznivější počet důležitých operací je režie ze zatím uvedených algoritmů nejvyšší. Další nevýhodou (pro účely navrhované knihovny) je, že algoritmus neumí pracovat pouze s diskrétními hodnotami (celými čísly). Musí bohužel pracovat s komplexními reálnými čísly, kterému operace (jako například '*','+','/' a další) trvají několikanásobně déle, než-li stejné operace na číslech diskrétních. Další nevýhodou práce s reálnými čísly může být chybný výsledek z důvodu špatného zaokrouhlování (a případnému roztažení této chyby).

2.4 Násobení řídkých polynomů

2.4.1 Struktura a algoritmus

Na začátku je nutné vybrat vhodnou strukturu pro uložení řídkého polynomu. Pro tento účel nám postačí jednoduchá struktura s dvojicemi prvek a index. Tato struktura jde například realizovat pomocí dvou polí, přičemž prvky jednoho pole jsou nenulové hodnoty původního polynomu a prvky pole druhého pak indexy do původního pole polynomů. Pseudokód pro vyplnění takové struktury vypadá zhruba následovně:

```
1: for i ← 0 to polSize do
2:     if pol[i] ≠ 0 then
3:         sparseIndex[j] ← i
4:         sparseValue[j] ← pol[i]
5:         ++j
6:     end if
7: end for
```

Použitý algoritmus pak je v podstatě stejný, jako algoritmus naivní s tím rozdílem, že se nebude násobit nulovými koeficienty. Do řídké struktury pak stačí převést pouze jeden polynom, a to ten, kterým se bude iterovat ve vnitřním cyklu (pro druhý polynom by se pak převedení nevyplatilo, protože jím bude iterováno pouze jednou). Celý algoritmus má asi takovouto podobu:

```
1: for i ← 0 to n do
2:     if polA[i] = 0 then
3:         continue
4:     end if
5:     for j ← 0 to m do
6:         res[i + sparseIndex[j]] += polA[i] * sparseValue[j]
7:     end for
8: end for
```

2.5 Ostatní algoritmy

2.5.1 Kroneckerova substituce

Přeloženo z [15].
Tento algoritmus umožňuje násobení polynomů v grupě $Z[x]$. Pokud jsou koeficienty polynomů známé, algoritmus je převede na celé číslo, které pak vynásobí.
Asymptotická složitost tohoto algoritmu je pak $O(k + nb)$, kde k je složitost vynásobení $k - bitového$ diskrétního čísla a nb je pak čas zabalení a rozbalení (b je roven zhruba dvojnásobků počtu bitů u koeficientů).
Zde je příklad toho, k čemu algoritmus slouží a jak funguje:

$$f = 41x^3 + 49x^2 + 38x + 29$$
$$g = 19x^3 + 23x^2 + 46x + 21$$
$$f(10^4) = 41004900380029$$
$$g(10^4) = 19002300460021$$
$$h(10^4) = f(10^4)g(10^4) = 779187437354540344421320609.$$
$$h = 779x^6 + 1874x^5 + 3735x^4 + 4540x^3 + 3444x^2 + 2132x + 609$$

2.5.2 Schönhageho-Strassenův algoritmus

Přeloženo z [16].
Tato metoda umožňuje efektivně násobit velká celá čísla. Asymptotická složitost této metody je $O(nlog(n)log(log(n)))$. Algoritmus využívá FFT v kruzích. Metoda se s využitím Kroneckerovi substituce dá využít i pro násobení celočíselných polynomů.

2.5.3 Toom-Cookova metoda (Toom-3)

Přeloženo z [17].
Tento algoritmus patří, stejně jako Karatsubův algoritmus, mezi algoritmy typu "Rozděl a Panuj"(*Divide to Conquer*). Algoritmus rozdělí čísla (nebo mnohočlen) na 3 části (proto *Toom-3*), čímž dokáže redukovat 9 násobení na pouhých 5 násobení (algoritmus pracuje rekurzivně, takže dělí čísla/polynomy v každém patře). Asymptotická složitost tohoto algoritmu je $O(n^{log_3 5})$ ($log_3 5 = 1.46497...$).
Existují i další druhy tohoto algoritmu (jako například Toom-1, Toom-1.5, Toom-2.5 nebo Toom-2, které fungují na stejném principu, jako algoritmus Karatsubův).

Kapitola 3

Optimalizace algoritmů

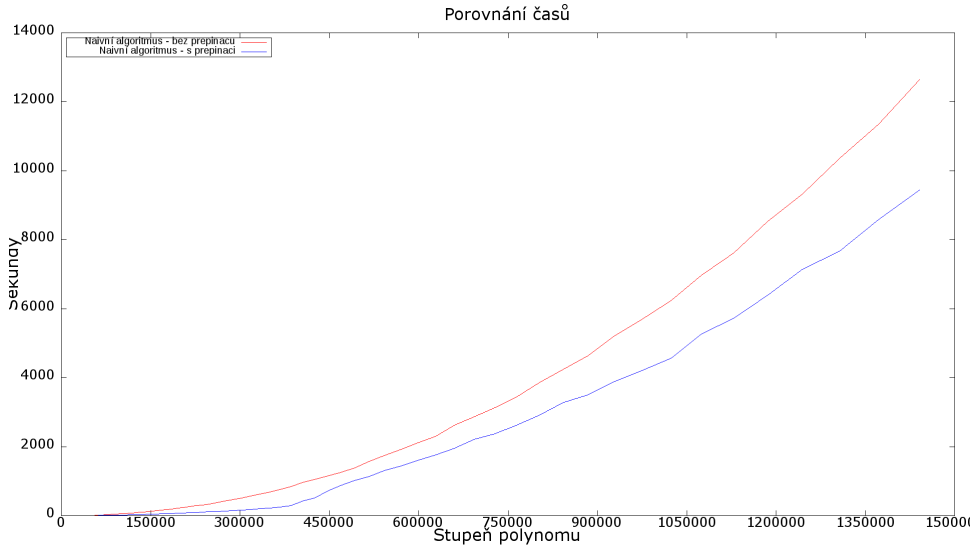
Každý z uvedených algoritmů je samozřejmě nutné zoptimalizovat a najít hranice, do kterých se vyplatí využít dané algoritmy.

3.1 Volby kompilátoru

Některé přepínače mohou nejen usnadnit práci programátora, ale dokonce i uzpůsobit program na příslušný systém. Jelikož je zde systém vybrán (server *Star*), mohou nám tyto přepínače značně pomoci s optimalizací. Po diskusi s vedoucím práce byly vybrány následující přepínače:

- `Ofast` - Tento přepínač aktivuje přepínače `O3` a `ffast-math`.
  - `O3` - Tento přepínač aktivuje několik desítek jiných optimalizačních přepínačů.
  - `ffast-math` - Tento přepínač zrychlí několik základních matematických operací (které se týkají zejména operací s nulou a nekonečnem).
- `ftree-vectorize` - Tento přepínač aktivuje přepínače `ftree-loop-vectorize` a `ftree-slp-vectorize`. Oba tyto přepínače slouží zejména k vektorizaci kódu.
- `msse4a` - Tento přepínač vybere instrukce pro námi vybraný systém.
- `mfpmath=sse` - Uzpůsobí aritmetiku řádové čárky pro námi vybraný systém.
- `ftree-vectorizer-verbose=5` - Tento přepínač je spíše informativní. Říká, které cykly byly vektorizovány, jak byly vektorizovány a pokud ne, tak proč nebyly vektorizovány.

Měření běhu bez přepínačů (3.1) bylo v průměru o $38,6\%$ pomalejší, než-li běh s přepínači.



Obrázek 3.1: Porovnání běhu naivního algoritmu bez přepínačů (*červená*) a s přepínači (*modrá*). Souřadnice x (0-1500000) značí stupeň polynomu a souřadnice y (0-14000) pak celkový čas v sekundách.

3.2 Optimalizace naivního algoritmu

3.2.1 Optimalizace algoritmu redukcí čtení a zápisů

Beze změny algoritmu bohužel již nijak nezlepšíme počet operací, jako je sčítání a násobení. Můžeme však snížit režii, která tento výpočet umožňuje. V první optimalizaci je účelem vyhnout se zbytečným zápisům (*write*) do výsledného pole. Toho může být docíleno tak, že místo původního iterování vnějším cyklem přes první polynom a vnitřním cyklem přes polynom druhý, budeme nyní iterovat vnějším cyklem přes polynom výsledný a vnitřním cyklem po obou polynomech zbývajících. Zapsáno pseudokódem bude tato optimalizace vypadat zhruba následovně:

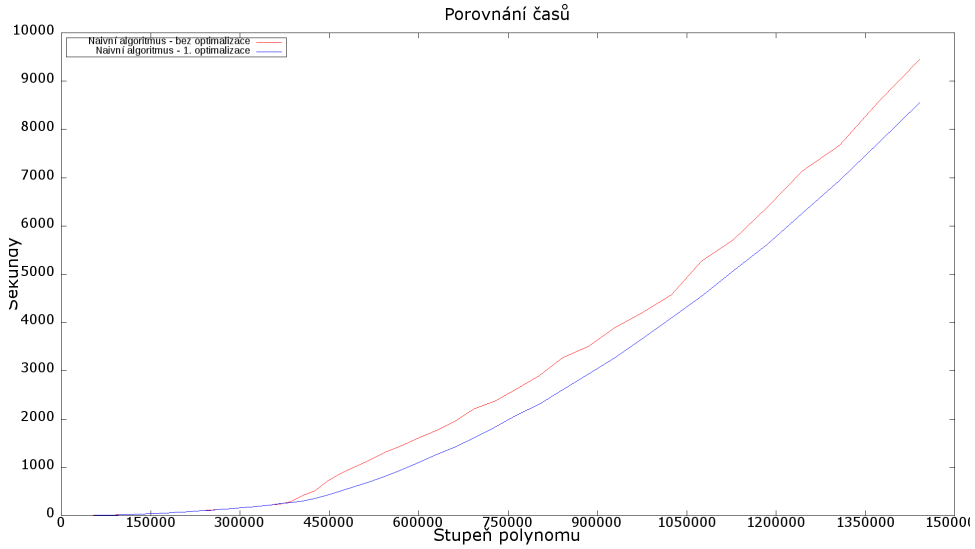
```
1: if aSize < bSize then
2:     switch first to second
3: end if
4: for i ← 0 to bSize do
5:     tmpAdd ← 0
6:     b ← i
7:     for a ← 0 to i do
8:         tmpAdd+ = polA[a] ∗ polB[b]
9:         − − s
10:    end for
11:    res[i] = tmpAdd
12: end for
13: for i ← bSize, j ← 1 to resSize do
14:    tmpAdd ← 0
15:    b ← bSize
16:    tmpSize ← getMin(i + 1, aSize)
17:    for a ← j to tmpSize do
18:        − − s
19:        tmpAdd+ = polA[a] ∗ polB[b]
20:    end for
21:    res[i] = tmpAdd
22: end for
```

Kromě již zmíněných řízení cyklů zde lze vidět i proměnou *tmpAdd* (která bude s největší pravděpodobností uložena v registrech), do které se ukládají výsledky, místo toho, aby se ukládaly do přímo do pole s výsledky. Tam se uloží až tehdy, když jsou výsledky spočteny. Tímto způsobem bude redukováno zhruba $n^2 - n$ zápisů do pole, ale naopak přibude n^2 zápisů do proměnné. Vzhledem k tomu, že potřebujeme k výpočtu znát i průběžný výsledek, bude to i obdobné s načítáním (zde to bude n^2 místo původních $n^2 - n$). I přesto, že v koncovém měřítku operací přibylo, tak se tato úprava vyplatí. Operace pro načítání a zápis do pole je totiž o dost pomalejší, než zápis či načtení z registru.

Měření optimalizace (3.2) proběhlo o 18% rychleji, než-li měření základní implementace.

3.2.2 Optimalizace algoritmu pomocí rozbalení cyklu

Převzato z [18]
Další možností, jak můžeme tento algoritmus optimalizovat, je rozbalení cyklu (*loop unroll*). To znamená, že se vybere nějaký modul (pro naše účely se hodí například číslo 8, protože chceme optimalizovat zejména pro polynomy s menším stupněm). Poté zarovnáme vnitřní cyklus (vnitřní cyklus se vybírá, protože je v něm nejvíce operací) tak, aby iteroval do čísla dělitelného vybraným modulem (zbytek je dopočten v následovaném malém cyklu). Metoda pak spočívá



Obrázek 3.2: Porovnání základní implementace naivního algoritmu (*červená*) a první optimalizace (*modrá*). Souřadnice x (0-1500000) značí stupeň polynomu a souřadnice y (0-12000) pak celkový čas v sekundách.

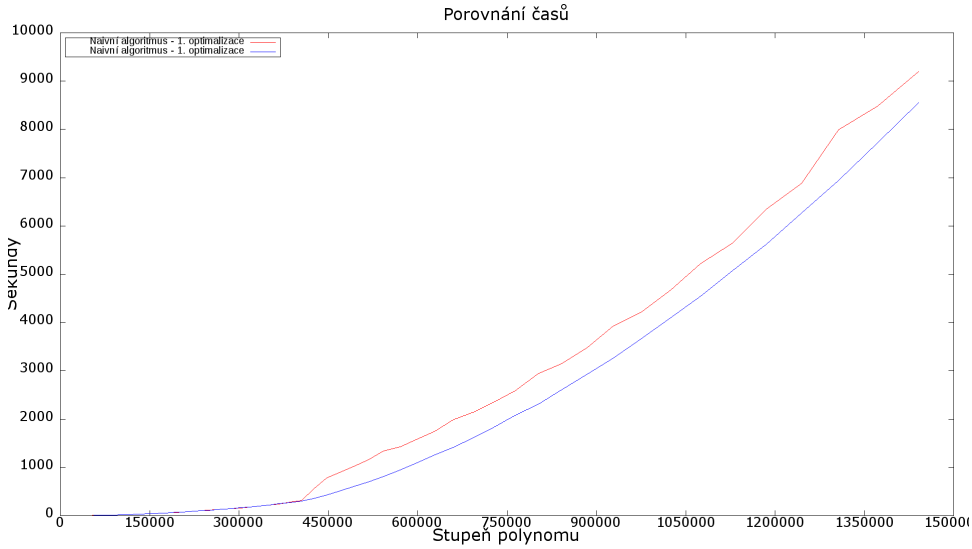
v tom, že se operace vykonávané ve vnitřním cyklu rozkopírují (tolikrát, kolik je modul) a iterovat se bude po vybraném modulu (místo po 1). To má za následek, že místo každých 8 kontrol podmínky cyklu (zda-li je i menší, než-li m) se nyní provede kontrola jedna.

Příklad transformace *loop unroll*:

```
1: for i ← 0 to n do
2:     doSomething(i++)
3: end for
4: Transformace na:
5: for i ← 0 to n do
6:     doSomething(i++)
7:     doSomething(i++)
8:     doSomething(i++)
9:     doSomething(i++)
10: end for
```

3.3

Bohužel druhá optimalizace nedosáhla zdaleka takových výsledků, jako optimalizace první. Celkově je o $14,5\%$ pomalejší.



Obrázek 3.3: Porovnání druhé optimalizace naivního algoritmu (*červená*) a první optimalizace (*modrá*). Souřadnice x (0-1500000) značí stupeň polynomu a souřadnice y (0-12000) pak celkový čas v sekundách.

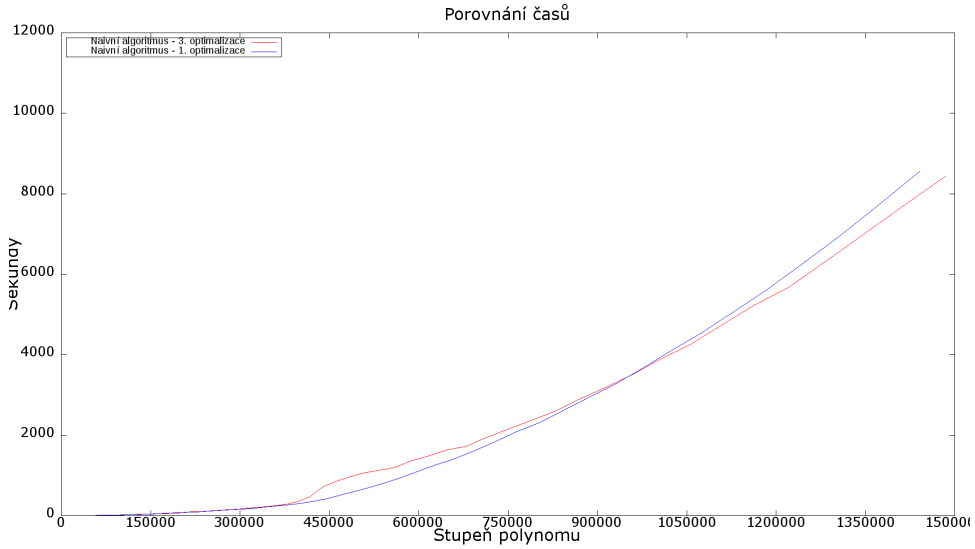
3.2.3 Optimalizace algoritmu pomocí vektorizace

Třetím a posledním návrhem na optimalizaci je vektorizace. Metoda spočívá v převedení původních proměnných či polí na vektory, a na těch provádět požadované operace. Výhodou je, že vektor má větší velikost, než-li původní čísla, a tak lže požadované operace provádět najednou. Bohužel je velikost čísel z důvodu přesnosti 64 bitů a velikost vektorů na serveru *Star* je pouze 128 bitů (takže zrychlení nebude příliš velké).
K vektorizaci lze využít standardní přepínače překladače g++ (**ftree-vectorize** a **ftree-vectorizer-verbose=5**).

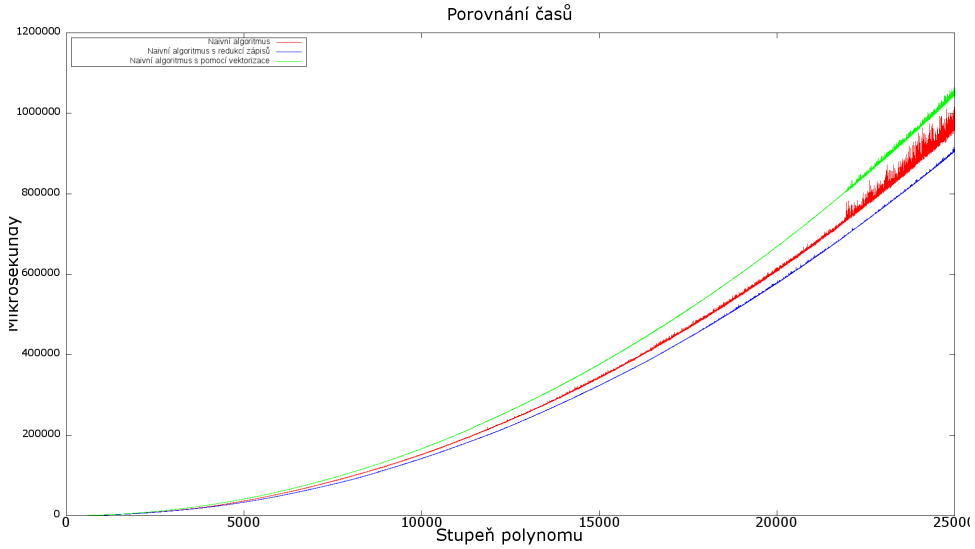
Druhá optimalizace dosáhla v celkovém měřítku (3.4) velmi podobných výsledků, jako optimalizace třetí. Rozdíl je ovšem v tom, že druhá optimalizace dosahovala lepších výsledků zejména z počátku, kdežto třetí optimalizace jich dosahovala až pro velmi velká čísla (což je bohužel pro naše účely nevhodné).

3.2.4 Závěrečné porovnání

Na závěr ještě část měření pro malá čísla, podle které bude později určena hranice. Z měření (3.5) nám opět nejlépe vyšla první optimalizace, a proto bude od teď používaná jako reprezentant naivního algoritmu (při hledání spodní hranice).



Obrázek 3.4: Porovnání třetí optimalizace naivního algoritmu (*červená*) a první optimalizace (*modrá*). Souřadnice x (0-1500000) značí stupeň polynomu a souřadnice y (0-12000) pak celkový čas v sekundách.



Obrázek 3.5: Porovnání nezoptimalizovaného naivního algoritmu (*červená*), první optimalizace (*modrá*) a třetí optimalizace (*zelená*) pro nízké stupně polynomu. Souřadnice x (0-25000) značí stupeň polynomu a souřadnice y (0-1200000) pak celkový čas v mikrosekundách.

3.3 Optimalizace Karatsubova algoritmu

Výhodou tohoto algoritmu je, že všechny optimalizace mohou být jednoduše provedeny najednou.
Pomocné pole pomocí ukazatelů - Řádky $5-8$ v pseudokódu mohou svádět k alokaci nových polí, do kterých by se nakopírovaly příslušné poloviny polynomů. To by si ovšem sebou neslo hned dvě nevýhody - samotný výpočet by mohl zabrat příliš zbytečné paměti a zbytečná lineární složitost navíc v každé rekurzi. Jazyk *C++* nám naštěstí dovoluje řešit tuto problematikou pouze za pomoci ukazatelů. Je možné si tedy v rekurzích předávat pouze ukazatel na příslušnou část polynomu, což nám zabere pouze konstantní čas v každé rekurzi.
Mezivýsledky pomocí pomocného pole - U mezivýsledků (řádky $9-10$) už to tak jednoduše bohužel nejde. Problémem je, že si výsledky potřebujeme zapamatovat a využít je dvakrát (což je podstata toho algoritmu). K tomu už pomocné pole potřeba je. Při šikovném použití nám ovšem stačí takováto pole dvě (alokují se na začátku algoritmu a v rekurzivních funkcích se již předávají pouze ukazatele na ně). Bohužel je však před každým použitím nutné je vynulovat.
Ukončení rekurze - Rekurze v pseudokódu končí až tehdy, když jsou velikosti polí s polynomy rovny 1 (viz. řádek 1). To je ovšem zbytečné, protože zejména pro malé stupně polynomů se vyplatí spíše naivní algoritmus, než-li algoritmus Karatsubův. Původní jednička tedy lze nahradit o něco vyšším číslem a nahradit závěrečné násobení algoritmem naivním. Pro tento účel bylo vybráno číslo 32.
Rozbalení cyklů - Bohužel v těle funkce stále zůstává několik cyklů. Ty můžeme (obdobně jako u algoritmu naivního) zrychlit metodou rozbalení cyklů. To nám ušetří čas, díky méně častým kontrolám podmínek cyklu. Zde to má navíc tu výhodu, pokud zvolíme vhodnou velikost rozbalení cyklu, nebudeme se muset starat o okolní režii (jakou je třeba dopočítávání zbytku po dělení, nalezení zbytku po dělení a podobně). To máme zajištěno díky kombinaci dvou věcí:

- Z definice algoritmu samotného máme zajištěno, že velikosti příchozích polí budou vždy násobky čísla 2.
- Známe číslo (ukončení rekurze), které značí zároveň maximální velikost příchozích polí.

Kombinací těchto znalostí pak můžeme zvolit vhodné číslo takové, abychom se o okolní režii nemuseli starat. Toto číslo musí splňovat podmínku, že $k = N/2^m$, kde k je číslo které hledáme, N je velikost ukončení rekurze a m je celé číslo, pro které platí, že $2^m \leq N$. Samozřejmě nejvhodnější je, aby číslo k bylo co nejpodobnější číslu N, protože poté bude zrychlení největší.

Předčasné ukončení rekurze - Nevýhodou tohoto algoritmu, oproti algoritmu naivnímu je fakt, že oba polynomy jsou upraveny do podoby 2^k (kde k je celé číslo) a to včetně polynomů o velikostech $\{2^{k-1}, 2^{k-1}+1, ..., 2^k\}$. Tímto bohužel získáváme valné množství zbytečných operací navíc. Nicméně i tento problém lze poněkud redukovat. Stačí si v rekurzi držet kromě ukazatelů a velikostí i aktuální pozici a původní stupeň polynomu. Pokud pak aktuální pozice překročí původní stupeň polynomu, můžeme rekurzi zastavit, protože víme, že zbytek polynomu je již vyplněn pouze nulami a vynásobení polynomu polynomem nulovým (tedy polynomem se stupněm -1) získáme opět polynom nulový. Ačkoliv tímto způsobem není možná redukce všech operací navíc, redukce je znatelná.
Zefektivnění dopočítání - Ačkoliv se naivní algoritmus zná jako dobrá volba pro dopočítání, je zbytečné komplikovat si dopočítávání zbytečnou režií. Vzhledem k tomu že známe konkrétní číslo, můžeme provést výpočet i bez jakékoliv režie, čímž ušetříme valné množství času. Kód bude vypadat zhruba následovně:

$$
\begin{array}{l}
resPol[0] += polA[0] * polB[0]; \\
resPol[1] += polA[1] * polB[0] + polA[0] * polB[1]; \\
... \\
resPol[61] += polA[31] * polB[30] + polA[30] * polB[31]; \\
resPol[62] += polA[31] * polB[31]
\end{array}
$$

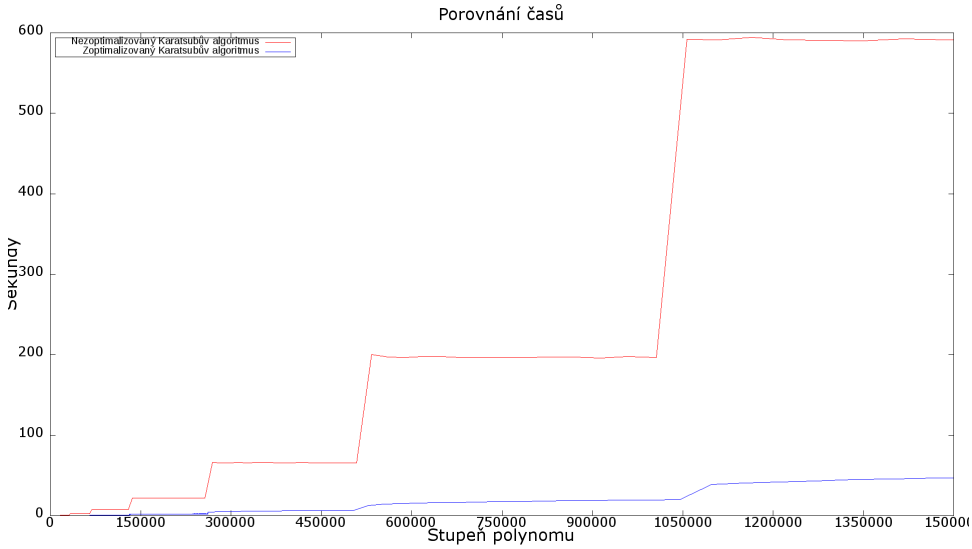
Mírnou nevýhodou této optimalizace (a to zejména pro programátora) je, že se zněkolikanásobí počet řádek kódu.

3.6

Prvním poznatkem je, že optimalizace zrychlily (dle měření) průběh algoritmu více než desetinásobně. Dále si lze povšimnout, že u původního algoritmu je vždy velký skok vždy pro hodnoty, který jsou rovny 2^k (kde k je celé číslo), kdežto u verze optimalizované je vzestup více kontinuální.

3.3.1 Optimalizace FFT

Rozdělení funkce DFT - Prvním návrhem je, rozdělit funkci na dvě. Samotné rozdělení bohužel nic nezrychlí, nicméně pomůže dalším optimalizacím. Výhodou je, že to nestojí téměř žádný výpočetní čas navíc (pouze to stojí řádky kódu navíc). Rozdělení proběhne tak, že místo původní kontroly velikosti na 1, bude nyní kontrola na nějaké vyšší číslo (pro naše účely 16) a při shodě se pak zavolá původní funkce s ukončením na velikosti 1.
Rozbalení cyklů - Tak jako u všech algoritmů tohoto typu se i zde dá optimalizovat pomocí rozbalení cyklů. Výhodou samozřejmě je, že jsou funkce rozdělené, a proto lze rozbalit cykli na určenou hodnotu bez jakýchkoliv zbytečných kontrol (dopočítávání menších čísel). Tomu navíc nahrává (stejně jako



Obrázek 3.6: Porovnání základní implementace Karatsubova algoritmu (*červená*) a jeho optimalizace (*modrá*). Souřadnice x (0-1500000) značí stupeň polynomu a souřadnice y (0-600) pak celkový čas v sekundách.

u Karatsubova algoritmu), že příchozí pole je velikosti 2^k (kde k je celé číslo). Rozbalením pak zlepšíme výpočetní čas díky menšímu počtu kontrol v cyklech.

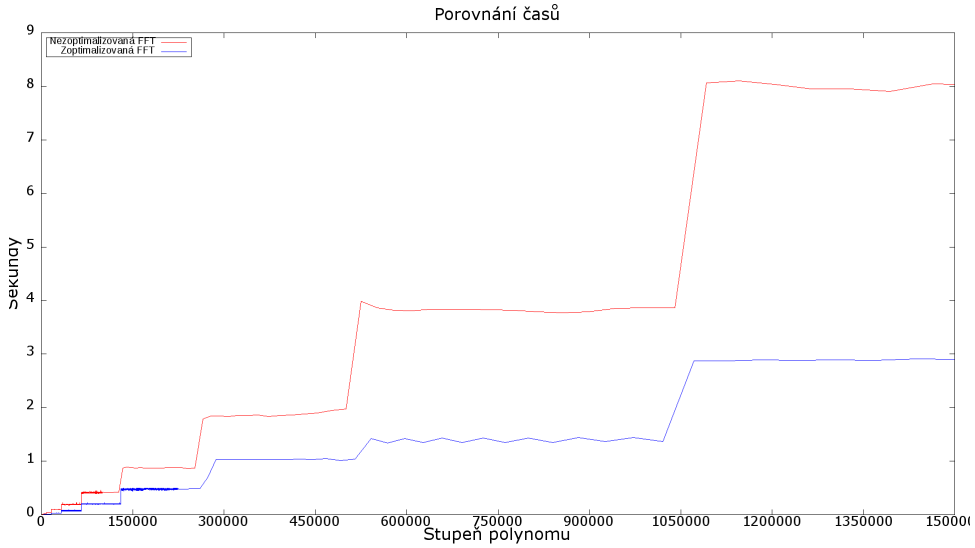
Zavedení skoků a ukazatelů - Další optimalizací je zavedení skoků a ukazatelů. To znamená, že se do funkce DFT bude místo původního pole posílat bude posílat ukazatel na začátek pole a velikost skoku platných prvků. To ušetří zbytečné překopírovávání a vytváření polí pro hodnoty liché (*odd*) a sudé (*even*). Původní lineární operace překopírování (v každé větvi) se nyní nahradí konstantními operacemi nalezení ukazatele a zvýšení skoku. Dopad na algoritmus je pak takoví, že se v cyklech bude přičítat po parametru *jump* (skok), místo původního přičítání po jedničkách.

Redukce násobení - Pokud první iteraci vystrčíme před tělo závěrečného cyklu, můžeme operaci $\omega \leftarrow \omega\omega_n$ umístit na začátek iterace (místo původního konce iterace). Tím se zbavíme jednoho násobení komplexních čísel pro každou větev stromu (v pseudokódu si lze povšimnout, že poslední operace je opravdu nevyužitá).

Změna ukončení rekurze - Algoritmus lze vylepšit tím, že ukončíme rekurzi předčasně (dříve než-li s velikostí pole 1). Optimální je končit například již u velikosti pole 4. Můžeme si povšimnout, že ω_4^k je z množiny $\{1, -1, i, -i\}$ a tudíž by při násobení těmito čísly vzniklo mnoho operací navíc (není ani potřeba operaci násobení vůbec využít, stačí operace '+' a '-').

Převod obou polynomů zároveň - V hlavní funkci si lze povšimnout, že funkce DFT je volána na oba polynomy. Vzhledem k tomu, že je velikost po-

lynomů (po úpravě) stejná, bude i průběh funkce DFT stejný. Obě funkce se mohou zkopírovat, a příslušně upravit pro dva polynomy zároveň (původní funkce musejí zůstat pro zpáteční převod). Touto optimalizací ušetříme veškerou režii funkce DFT (nejen těla cyklů, ale i získávání mocnin ω_n a podobně).



Obrázek 3.7: Porovnání základní implementace FFT (*červená*) a její optimalizace (*modrá*). Souřadnice x (0-1500000) značí stupeň polynomu a souřadnice y (0-9) pak celkový čas v sekundách.

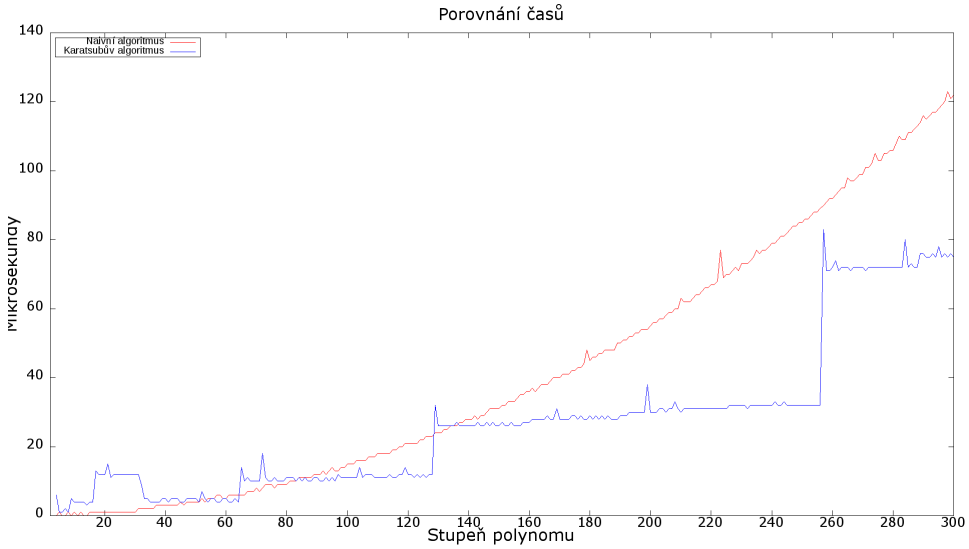
Kapitola 4

Realizace a testování

V této kapitole se pokusíme dát dohromady vše, co bylo popsáno v kapitolách předchozích. To je provedeno pomocí měření. Nakonec je výsledek podroben testům.

4.1 Hledání hranic

4.1.1 Hranice mezi algoritmem naivním a Karatsubovým



Obrázek 4.1: Porovnání naivního algoritmu (*červená*) a Karatsubova algoritmu (*modrá*). Souřadnice x (0-300) značí stupeň polynomu a souřadnice y (0-140) pak celkový čas v mikrosekundách.

Lze si povšimnout (obrázek 4.1), že Karatsubův algoritmus nemá zda-

leka tak plynulý vývoj, jako algoritmus naivní (i přes provedené optimalizace stále obsahuje velké skoky u hodnot 2^k). Tento fakt způsobil, že se oba grafy protnuly několikrát (konkrétně třikrát). Nás bude zajímat, zejména poslední protnutí.

Zde je ještě tabulka přesných hodnot měření.

Stupeň polynomů	Naivní algoritmus	Karatsubův algoritmus
130	24	26
131	24	26
132	25	26
133	25	27
134	26	26
135	26	26
136	26	26
137	27	26
138	27	26
139	28	26
140	28	27

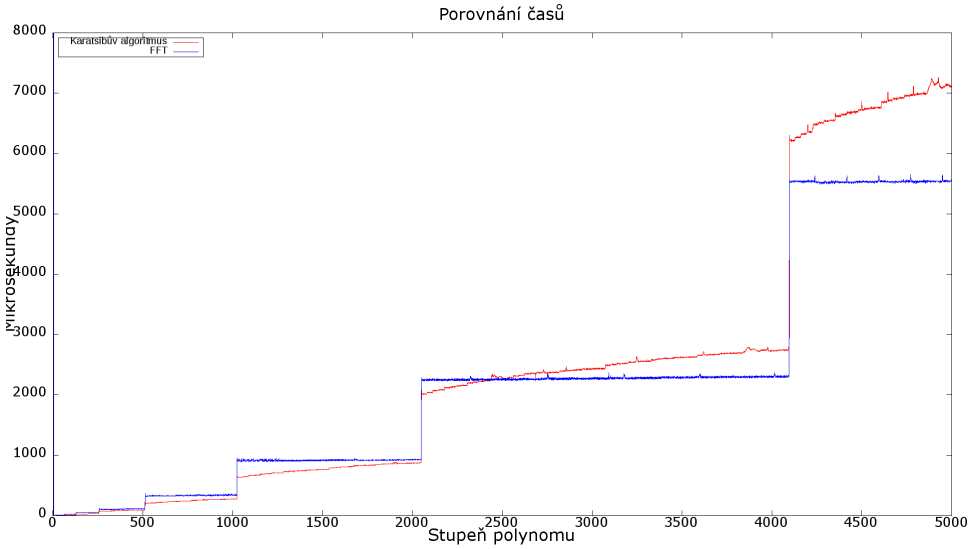
Tato tabulka nám ukazuje počet mikrosekund nutný k výpočtu polynomu určitého stupně (viz první sloupec) pro naivní a Karatsubův algoritmus. Z tabulky lze dolní hranici určit například jako 135.

4.1.2 Hranice mezi algoritmem Karatsubovým a FFT

4.2
Můžeme si povšimnout, že mají z počátku oba algoritmy podobný vývoj (nejen strukturou grafu, ale také časy). Vývoj grafu Karatsubova algoritmu je díky optimalizacím o trochu plynulejší.

Zde je ještě tabulka přesných hodnot měření.

Stupeň polynomů	Karatsubův algoritmus	Rychlá Fourierova Transformace
2425	2234	2247
2426	2238	2262
2427	2239	2244
2428	2229	2268
2429	2230	2240
2430	2234	2263
2431	2242	2242
2432	2237	2263
2433	2250	2243
2434	2260	2253
2435	2258	2240



Obrázek 4.2: Porovnání Karatsubova algoritmu (*červená*) a FFT (*modrá*). Souřadnice x (0-5000) značí stupeň polynomu a souřadnice y (0-8000) pak celkový čas v mikrosekundách.

Tato tabulka nám ukazuje počet mikrosekund nutný k výpočtu polynomu určitého stupně (viz první sloupec) pro FFT a Karatsubův algoritmus. Z tabulky lze dolní hranici určit například jako 2433 (není samozřejmě nutné měřit s přesností na jednotky - nicméně když už tyto data máme, není ani důvod je nepoužít přesně).

4.2 Násobení řídkých polynomů

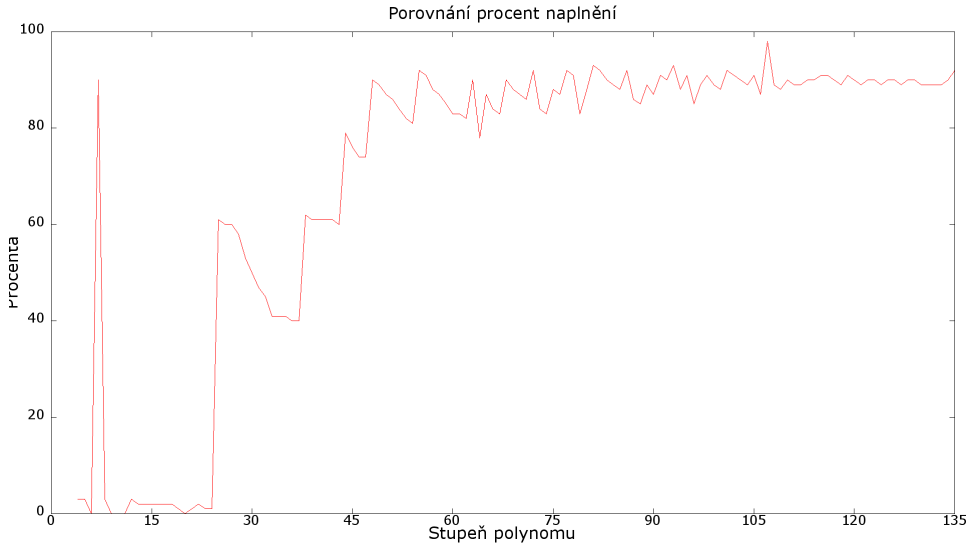
4.2.1 Měření

Měření pro zjištění hranice řídkosti probíhá celkem třikrát (pro každý z vybraných algoritmů). Probíhá tak, že se zjistí, na jakém procentu řídkosti poprvé překročí algoritmus pro počítání řídkých polynomů čas vybraného algoritmu (měření probíhá několikrát pro každý z vybraných stupňů polynomů a výsledek se zprůměruje). Hranice je pak odhadnuta dle uskutečněných měření.
Z časových důvodů není možné měřit všechny možnosti řídkosti. Oba polynomy proto budou mít řídkost stejnou. V knihovně pak bude řídkost obou polynomů zprůměrována. Následující vztah pak zajistí, že násobení řídkých polynomů nebude nikdy zavoláno, pokud nebude čas opravdu lepší:

$$\begin{aligned}(k+l)(k-l) \;???\; k^2\\ k^2 - kl + kl - l^2 \;???\; k^2\\ -l^2 \;???\; 0\\ -l^2 \;\leq\; 0\end{aligned}$$

Tento vztah nám říká, že pokud bude jeden polynom mít menší řídkost, než li průměr obou polynomů (a druhý tedy naopak vyšší řídkost), bude čas lepší, než kdyby oba polynomy byli blíže průměru (či přímo na průměru). Tento vztah lze využít i pro stupeň polynomů.

4.2.2 Hranice řídkosti u naivního algoritmu

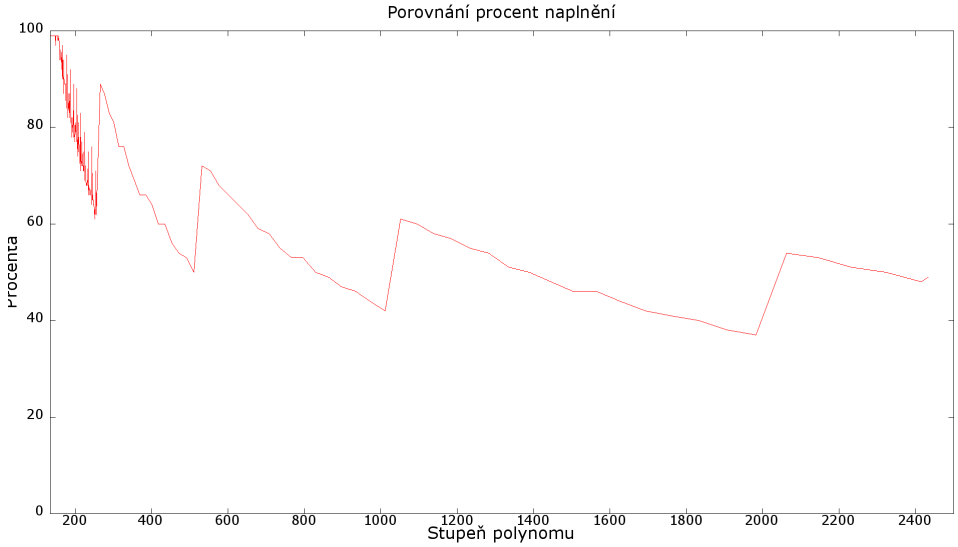


Obrázek 4.3: Měření za účelem zjištění hranice řídkosti pro naivní algoritmus. Souřadnice x (0-135) značí stupeň polynomu a souřadnice y (0-100) jsou pak procenta řídkosti.

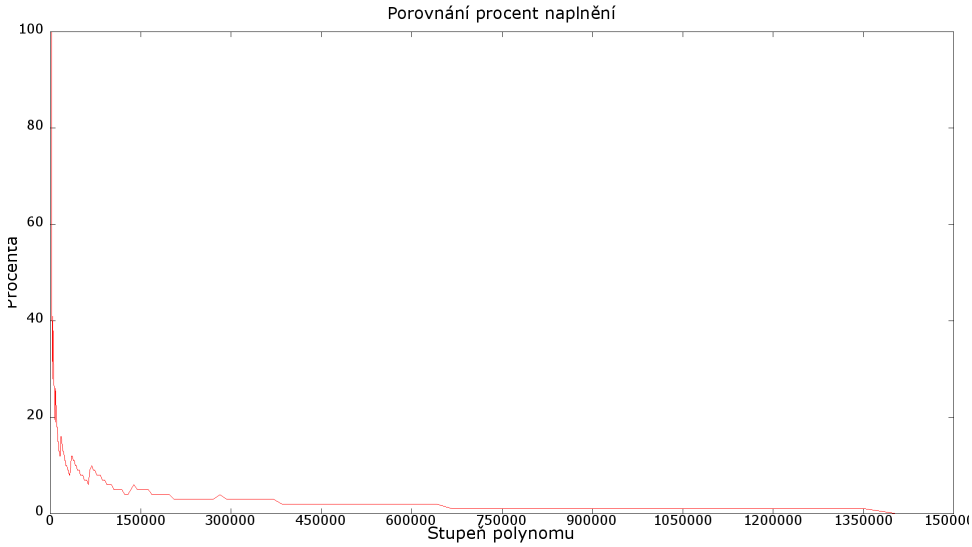
Z měření (4.3) lze odvodit, že pro polynomy se stupněm menším jak 25 nemá vůbec smysl řídkost polynomu řešit. Dále se tato hranice s narůstajícím stupněm polynomu přibližuje 90%. Bohužel probíhá měření na velice malých časech, a proto je zde valné množství chyb v měření.

4.2.3 Hranice řídkosti u Karatsubova algoritmu

Z tohoto měření (4.4) lze odvodit, že se na rozdíl od naivního algoritmu hranice řídkosti stále snižuje. To je způsobeno tím, že v předešlém měření byla složitost stejná, nicméně řídké násobení mělo vyšší konstantu. Zde je tomu naopak. Dále stojí za povšimnutí zuby, které jsou ve stejných místech, jako mocniny dvojky (což je způsobeno strukturou Karatsubova algoritmu). Hranice na měřeném intervalu kolísá zejména kolem 50% řídkosti, nicméně velikost oscilace je až 10% na obě strany.



Obrázek 4.4: Měření za účelem zjištění hranice řídkosti pro Karatsubův algoritmus. Souřadnice x (135-2433) značí stupeň polynomu a souřadnice y (0-100) jsou pak procenta řídkosti.



Obrázek 4.5: Měření za účelem zjištění hranice řídkosti pro FFT. Souřadnice x (0-1500000) značí stupeň polynomu a souřadnice y (0-100) jsou pak procenta řídkosti.

4.2.4 Hranice řídkosti u FFT

Z měření (4.5) lze odvodit, že z počátku se se vyplatí použít řídký algoritmus pro polynomy s řídkostí do 10% a stupněm maximálně 70000. Dále však možná

řídkost postupně klesá, a pro polynomy se stupněm vyšším, než li 1400000 se algoritmus nevyplatí použít téměř vůbec (použitelná řídkost je menší, než-li 1%).

4.3 Testování knihovny

Pro testování knihovny jsem zvolil následující postup:

<1> Otestování naivního algoritmu několika malými známými polynomy.

<2> Porovnání správnosti výstupů naivního algoritmu oproti výstupům všech dalších algoritmů pro velké množství náhodně velkých polynomů.

4.3.1 Testování naivního algoritmu

V této sekci je popsáno několik testů, které byly provedeny. Úplné testování (včetně výpisů z logu a závěru) je umístěno v souboru příloh. Všechny testy proběhly úspěšně. a) Vynásobení prvních třech řádů stabilního polynomu polynomem nulovým. Výsledkem všech těchto operací by měla vyjít 0. Konstanty pro stabilní polynomy zvolíme následovně: a=2, b=3,c=5.
b) Další z testů pak spočívá ve vynásobení Rényiho polynomu a dále jeho umocnění. Správnost výstupu si (samozřejmě mimo spočítání některým z ověřených způsobů) můžeme ověřit na počtu členů, kterých musí vyjít nejprve 29 a posléze 28.
c) Nyní zkusíme pomocí stejného postupu, jako u polynomu předchozího otestovat i Choudryho polynom.
d) Čtvrtým testem pak bude vynásobení chromatického polynomu kubického grafu polynomem konstantním. Tuto činnost provedeme právě třikrát, a to s konstantními polynomy: $p_1(x) = 1, p_2(x) = -1ap_3(x) = 2$. Výsledkem by pak měl být původní polynom, jehož všechny koeficienty jsou vynásobeny jediným nenulovým koeficientem zvoleného konstantního polynomu.
e) V tomto testu bude algoritmus testován čtverci Jacobsthalovo polynomů. Po dosazení čísla 1 do proměnné x ve výsledku by mělo dát čtverec n-tého Jackobsthalova čísla. Výsledky pak budou jednotlivě ověřeny pomocí jazyka Wolfram Mathematica, který umí úlohy tohoto typu řešit. Pro otestování byli zvoleny 2.,3.,4. a 5. Jackobsthalův polynom.
f) Šestým testem pak je vzájemné vynásobení Fibonacciho polynomů. Vzájemně jsou v testu mezi sebou násobeny druhý a třetí Fibonacciho polynom a posléze čtvrtý a pátý Fibonacciho polynom. Po dosazení hodnoty jedna do výsledků by měl vyjít násobek 2. a 3. Fibonacciho čísla a poté 4. a 5. Fibonacciho čísla (tedy 2 a 15).
g) Dalším testem pak je vzájemné vynásobení Fermatovo a Fermatovo-Lucasovo polynomů. Vzájemně se mezi sebou vynásobí vždy Fermatův polynom a Fermatův-Lucasův polynom n-tého stupně, a to pro n od 1 do 5. Po dosazení jedničky

do výsledků by pak měl vyjít součin Fermatova a Fermatova-Lucasova n-tého čísla.
h) Osmým testem pak bude ověření pomocí rekurentního vztahu v Pellovo polynomech. Pelovi polynomy pro n od 2 do 4 budou vynásobeny polynomem '$2x$' a výsledkem by měl být $\mathbf{P}_{n+1} - \mathbf{P}_{n-1}$.
i) Další test testuje vynásobení polynomu polynomem 'x'. Tímto polynomem jsou vynásobeny první tři čtvercové věžové polynomy. Výsledek testu by měl být na výstupu podobný, jako původní polynom, nicméně posunutý o jedna doleva (tedy by měl končit nulou).
j) Testem desátým a zároveň testem posledním je ověření několika známých vztahů a to vztahů následujících:

- $(x - b)(x + b) = x^2 - b^2$
- $(x + b)^2 = x^2 + 2xb + b^2$
- $(x - b)^2 = x^2 - 2xb + b^2$
- $(x + b)(x^2 - bx + b^2) = x^3 + b^3$

Zvolme za 'b' například číslo 2.

4.3.2 Testování zbylých algoritmů

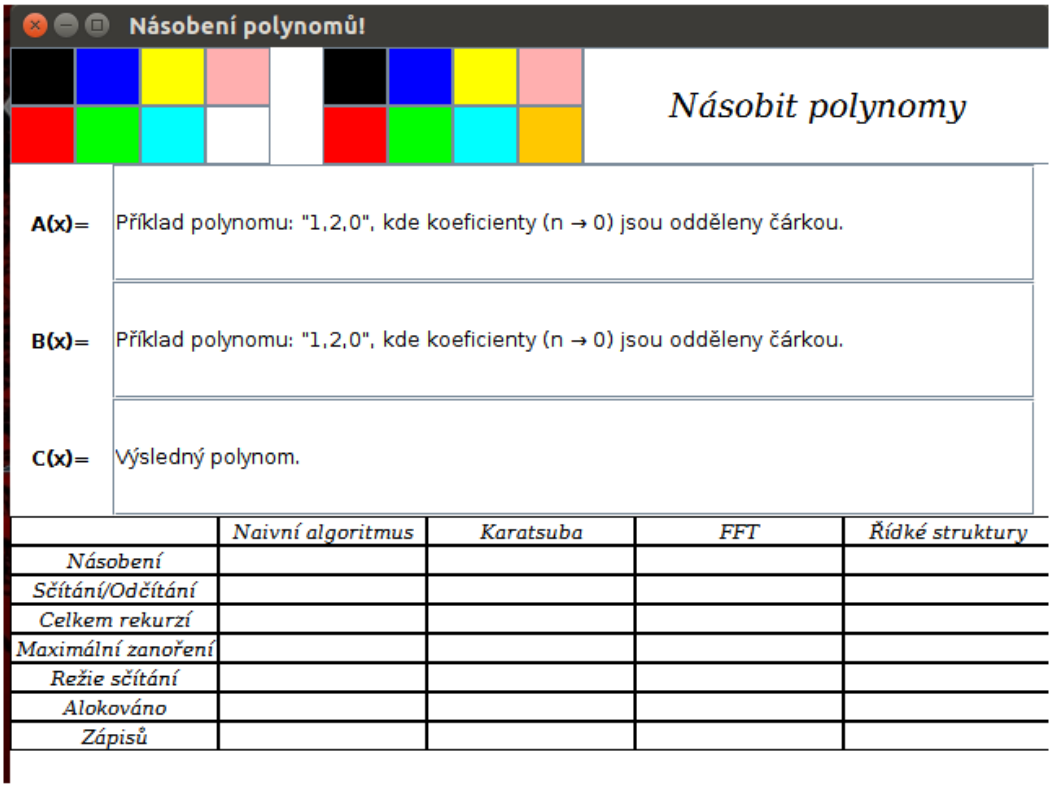
Pro otestování zbylých algoritmů byla zvolena metoda, ve které se naplnily dva náhodně velké polynomy (velikost byla přiměřená algoritmům) a jejich výstupy se navzájem otestovali proti sobě, zda-li odpovídají. Tato metoda byla několikrát zopakována (v řádu stovek až tisíců iterací). Prvně se algoritmy testovali oproti algoritmu naivnímu (který byl již ověřen v předchozím testu). Dále se však testovali i proti sobě, protože naivní algoritmus nedovoluje z časových důvodů polynomy s vysokým stupněm.

4.4 Realizace GUI programu

4.4.1 Program za běhu

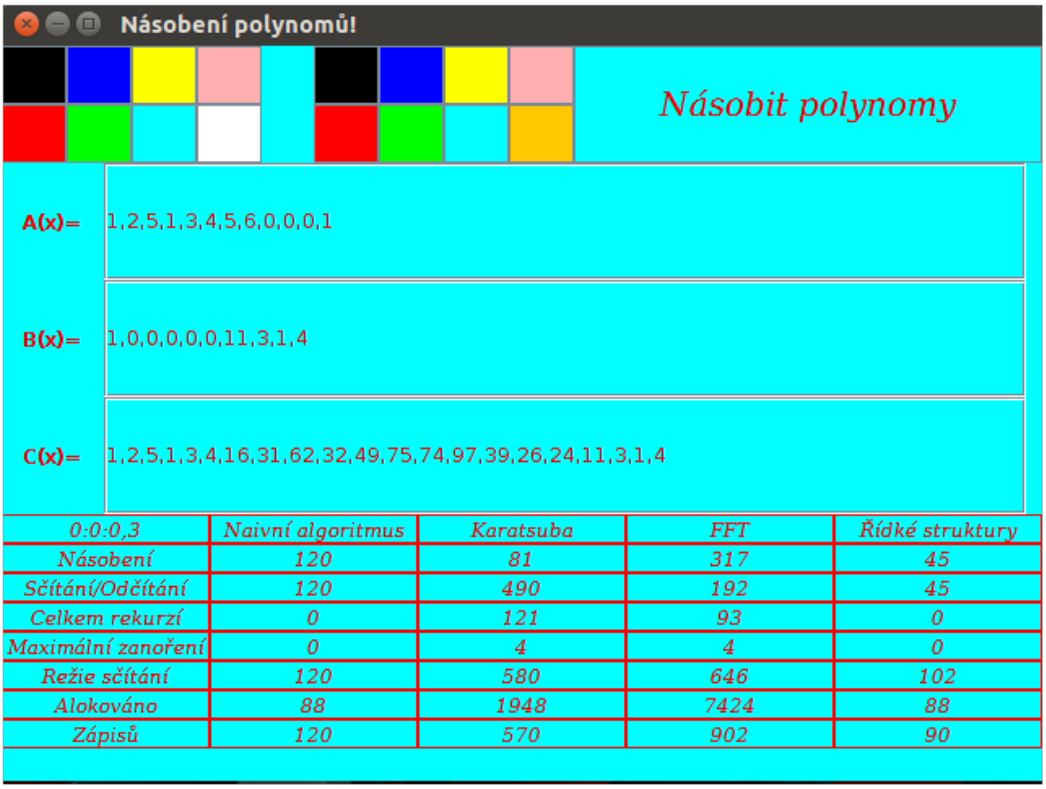
4.4.2 Funkčnost

Jedná se o jednoduchý edukativní program, sloužící zejména pro zjišťování statistik o běhu některých algoritmů pro násobení polynomů. Polynomy, které mají být násobeny se jednoduše vyplní do textových polí (vedle '$A(x) =$' a '$B(x) =$') a stiskne se tlačítko násobit polynomy (vpravo nahoře). Program poté pustí simulace algoritmů a za pomoci navržené knihovny pak spočítá výsledek, který se zobrazí v textovém poli (vedle '$C(x) =$'). Statistiky lze pak nalézt v dolní tabulce, ve které je i výsledný čas (ve formátu "Hodiny:Minuty:Sekundy,Milisekundy"). Featurou programu je pak několik tlačítek v levém horním rohu, které umožňují uživateli změnit barvu písma a



Obrázek 4.6: Ukázka GUI programu po spuštění.

pozadí. Celý program je uzpůsoben spíše pro menší polynomy (je nutné je vyplňovat ručně - není zde žádný generátor).



0:0:0,3	Naivní algoritmus	Karatsuba	FFT	Řídké struktury
Násobení	120	81	317	45
Sčítání/Odčítání	120	490	192	45
Celkem rekurzí	0	121	93	0
Maximální zanoření	0	4	4	0
Režie sčítání	120	580	646	102
Alokováno	88	1948	7424	88
Zápisů	120	570	902	90

Obrázek 4.7: Příklad GUI programu po provedení výpočtu.

Závěr

Knihovna byla úspěšně navržena. Z rešerše knihoven a diskuse s vedoucím práce byly vybrány příslušné algoritmy. Jejich výhody byly prodiskutovány. Byly vybrány algoritmy: naivní, Karatsubův a FFT. Všechny algoritmy byly diskutovány. Dále byly navrženy optimalizace implementací těchto algoritmů. Příslušně optimalizované algoritmy byly pak změřeny a byly nalezeny hranice, pro které bylo vhodné tyto algoritmy využívat. Dále byl implementován algoritmus pro násobení polynomů řídkých. Pro všechny dosud implementované algoritmy byla pak nalezena další hranice, která určuje, zda-li se nevyplatí využít spíše algoritmus pro násobení polynomů řídkých. Všechny tyto poznatky byli použity k vhodné implementaci navrhované knihovny.
Jako další byl navržen a implementován edukativní GUI program sloužící k získávání statistik z běhů algoritmů. Program byl úspěšně propojen s knihovnou, která zde slouží k rychlému získání výsledků a časů. Správná funkčnost knihovny a programu byla ověřena na známých polynomech.
Během návrhu nedošlo k žádnému zásadnímu problému, tudíž šlo při implementaci postupovat vždy podle zamýšlených návrhů. Veškeré časy byly měřeny pomocí třídy, využívající C++ knihovnu `ctime`.

Literatura

[1] Petr Olšák, Úvod do algebry, zejména lineární. Fakulta elektrotechnická ČVUT v Praze, 1 edice, 2009.

[2] Thomas H. Cormen, Charles E. Leisterson, Ribakd K, Ruvestm Ckuffird Stein, Introduction to Algorithms. The MIT Press, 3 edice, 2009.

[3] Anatolii Alexeevitch Karatsuba a Yuri Ofman. Multiplication of multidigit numbers on automata. Soviet Physics—Doklady, 7(7):595–596, 1963. Překlad článku v Doklady Akademii Nauk SSSR, 145(2), 1962.

[4] Dr. Maier, D. (18.5.2011). Polynomials and the Fast Fourier Transform (FFT). Algorithm Design and Analysis. Portlandská Státní Univerzita.

[5] Weisstein, Eric W. "Left Half-Plane."Z MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LeftHalf-Plane.html

[6] Weisstein, Eric W. "Stable Polynomial."Z MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/StablePolynomial.html

[7] Weisstein, Eric W. "Rényi's Polynomial."Z MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/RenyisPolynomial.html

[8] Weisstein, Eric W. "Sparse Polynomial Square."Z MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/SparsePolynomialSquare.html

[9] Weisstein, Eric W. "Fermat Polynomial."Z MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/FermatPolynomial.html

[10] Weisstein, Eric W. "Fermat-Lucas Polynomial."Z MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Fermat-LucasPolynomial.html

[11] Weisstein, Eric W. "Fibonacci Polynomial."Z MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/FibonacciPolynomial.html

[12] Weisstein, Eric W. "Jacobsthal Polynomial."Z MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/JacobsthalPolynomial.html

[13] Weisstein, Eric W. "Pell Polynomial."Z MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PellPolynomial.html

[14] Weisstein, Eric W. "Rook Polynomial."Z MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/RookPolynomial.html

[15] Harvey, David. (Únor 2009). Faster polynomial multiplication via multipoint Kronecker substitution. CS Theory Seminar. Newyorská Univerzita.

[16] Kruppa, A. (2.2.2009). Fast Integer Multiplication with Schonhage-Strassen's Algorithm. Tým CACAO - LORIA, Nancy.

[17] D. Knuth. The Art of Computer Programming, Volume 2. Třetí edice, Addison-Wesley, 1997.

[18] I. Šimecek, M. Šoch (2013). Kompilátorové optimalizace I: Metody transformací zdrojových kódů. Efektivní Implementace Algoritmů. České Vysoké Učení Technické - Fakulta Informačních Technologií.

Příloha **A**

Seznam použitých zkratek

GUI Graphical user interface

FFT Fast Fourier Transformation

DFT Discrete Fourier Transformation

Příloha B

Obsah přiloženého CD

```
.
+-- GUIPolynomal.jar
+-- onf
|   \-- pictures
|       +-- ico.png
|       \-- pol.png
+-- polymul.cpp
+-- polymul.h
+-- polymulrw
\-- testovani.pdf
```