Univerzita Karlova v Praze, Filozofická fakulta
Katedra logiky

Valentina Jelínková

Schönhageho-Strassenův algoritmus a jeho matematické pozadí

Schönhage-Strassen algorithm and the mathematics behind it

Bakalářská práce

Vedoucí práce:
Doc. RNDr. Vítězslav Švejdar, CSc.

2022

Prohlašuji, že jsem bakalářskou práci vypracovala samostatně a že všechny použité prameny a literatura byly řádně citovány. Práce nebyla předložena jako splnění studijní povinnosti v rámci jiného studia, nebo předložena k obhajobě v rámci jiného vysokoškolského studia, či k získání jiného nebo stejného titulu.

V Praze 30. července 2022

Valentina Jelínková

Abstrakt

Práce se zabývá Schönhageho-Strassenovým algoritmem, pro násobení velkých čísel se složitostí $O(n \log n \log\log n)$. Obsahuje nezbytné teoretické základy pro popis a pochopení algoritmu a jeho složitosti. Významný prostor je věnován Diskrétní Fourierově transformaci v komplexní a modulární aritmetice, se dvěmi různými interpretacemi algoritmu FFT.

Abstract

This thesis deals with the Schönhage-Strassen algorithm for multiplying large integers with complexity $O(n \log n \log\log n)$. It contains the necessary theoretical foundations for describing and understanding the algorithm and its complexity. A significant attention is devoted to the Discrete Fourier Transform in complex and modular arithmetic, with two different interpretations of the FFT algorithm.

Obsah

1 Úvod

Mnoho historiků považuje za významný milník naší doby rok 1448, kdy Johannes Guttenberg vynalezl knihtisk. Je ovšem zajímavé si uvědomit, že Guttenberg by číslo 1448 napsal jako MCDXLVIII. Jak bychom sčítali např. MCDXLVII + DCCCXII, nebo dokonce násobili? Poziční decimální systém byl vyvinut v Indii, okolo roku 600 n.l. Tento systém umožňuje kompaktní zápis i relativně velkých čísel a aritmetiku proveditelnou v elementárních krocích. Na rozšíření tohoto systému do Evropy měl zřejmě největší vliv perský matematik z 9. století Al Khwazirmi (česksy též Al-Chórezmí, či latinsky Al-Gorizmí) a jeho učebnice aritmetiky. Al Khwazirmi v ní uvádí nejen základní metody pro sčítání, násobení a dělení čísel, ale také pro počítání odmocnin, či cifer v čísle π. Latinský přepis jeho jména dal vzniknout slovu Algorimus.[6]

Algoritmus pro tzv. "školní", či "dlouhé"násobení, který počítá součin n-ciferných čísel v čase $O(n^2)$, byl pravděpodobně lidstvu známý mnohem dříve[1]. Po mnoho staletí tak byl tento algoritmus nejlepší a nejrychlejší metodou pro násobení větších čísel. Kolem roku 1960 mu tuto pozici přebral Karatsubův algoritmus se složitostí $O(n^{1.59})$[13]. Od tohoto momentu došlo k několika spíše drobným vylepšením násobení a jeho složitosti, od matematiků jako jsou A. L. Toom, A. Schönhage a D. E. Knuth (viz. např. [11]). Významný vliv na násobící algoritmy přinesl Cooley-Tookeyho algoritmus pro rychlou Fourierovu transformaci (FFT)[3], publikovaný roku 1965. Schöenhage se Strassenem vyvynuli dva algoritmy uplatňující FFT a publikovali je r. 1971[15].

První algoritmus využívá číselné aproximace v komplexní aritmetice, a redukuje násobení délky n na násobení délky $O(\log n)$. Jedná se o přímočaré využití FFT, respektive Věty o Konvoluci, na komplexních kořenech 1. Tento algoritmus při rekurzivní aplikaci redukuje velikost násobených čísel z n do $O(\log n)$ pro každou úroveň rekurze, a počet těchto úrovní je $\log^* n + O(1)$. Použití tohoto algoritmu rekurzivně vede k odhadu složitosti

$$M(n) = O(nM(n')) + O(n \log n), \qquad n' = O(\log n),$$

jehož řešení je

$$M(n) = O(K^{\log^* n}) n \log n \log \log n \ldots \log^{O((\log^* n)-1)} n)$$

pro nějakou konstantu $K > 0$.[10]

[1] Např. viz. Brahmagupta; Bhāskara II (1817). Algebra, with Arithmetic and Mensuration, Ze Sanskrtu přeložil Henry Thomas Colebrooke. John Murray. p. 319.

Druhý z publikovaných algoritmů je slavný Schönhage-Strassenův algoritmus, se složitostí $O(n \log n \log \log n)$, kterým se bude zabývat tato práce. Tento algoritmus si se svou asymptotickou složitostí udržel vedoucí postavení po dlouhých 36 let, kdy jej roku 2007 překonal Martin Fürer [7].

Tento algoritmus provádí výpočty konvoluce v modulární aritmetice, což se z hlediska složitosti může zdát kontraintuitivní. Musíme totiž po každé operaci ještě převést výsledek na zbytkovou třídu. Pro redukci počtu násobení v konvoluci se použije negacyklická konvoluce. Ta je počítána dvakrát, pro dvě různá nesoudělná moduli, přičemž pro výpočet v menším okruhu je použit Karatsubův algoritmus. Tyto výsledky jsou zkombinovány pomocí Čínské zbytkové věty.[1]

2 Asymptotická složitost

Asymptotická analýza funkcí
Slovo *asymptotický* znamená limitně se přibližovat k určité hodnotě či křivce. Asymptotická analýza se zabývá rychlostí růtsu funkcí, přehlíží jejich chování na malých vstupních hodnotách a zaměřuje se na jejich limitní chování, které nastává když se vstupní hodnoty blíží nekonečnu. Poskytuje nám jednoduchou charakteristiku pro efektivitu algoritmu a umožňuje porovnávat různé algoritmy vůči sobě.

Jednoduchým příkladem asymptotická analýzy je funkce $f(n) = n^2 + 3n$, kde se term $3n$ stává nepodstatný v poměru k n^2 pro velmi velké n. Řekneme tedy, že funkce $f(n)$ je asymptoticky ekvivalentní s n^2, respektive že její asymptotická složitost se rovná n^2, když $n \longrightarrow \infty$, což symbolicky zapíšeme jako $f(n) \sim n^2$.

K asymptotické časové složitosti bývá také referováno, jako k časové složitosti, či časové náročnosti (*time complexity*, *time cost*). Používá se k zapsání nejrychlejšího běhu algoritmu (*best case scenario* - nejlepší případ), a nejpomalejšího (*worst case scenario* - nejhorší případ).

V současné době jsou používány tři hlavní asymptotické notace:

Název	píšeme	symbolizují
Big-O notace	$f(n) = O(g(n))$	asymptotická horní závora
Omega notace	$f(n) = \Omega(g(n))$	asymptotická dolní závora
Theta notace	$f(n) = \Theta(g(n))$	uzavření shora i sdola

Pro analýzu, použitou v této práci, nám postačí seznámení se s Big-O notací. O ostatních notacích se lze více dozvědět např. v [12], nebo v [4].

2.0.1 Big O notace

Definice. Big-O notace Nechť g je funkce z N do R (nebo z R do R) Definujme:

$$O(g(n)) = \{f(n) : \quad \text{Existují pozitivní konstanty } c, n_0 \text{ takové, že } (|f(n)| \leq c|g(n)|) \quad \forall n \geq n_0\}$$

Prodrobněji se o tomto tématu lze dočíst např. v [12], či v [16], odkud bylo také čerpáno pro tuto kapitolu.

2.1 Složitost rekurzivních funkcí

Algoritmy typu rozděl a panuj se většinou řídí podle jednotného vzorce: rozdělí problém velikosti n, na a podproblémů, velikosti n/b, a následně kombinují výsledky v $O(n^d)$ čase, pro nějaké pozitivní konstanty a, b a d. Složitost takovýchto algoritmů tedy můžeme zapsat do rovnice:

$$T(n) = aT(n/b) + O(n^d).$$

Následující věta zjednodušuje vyhodnocení složitosti takovýchto algoritmů. Říká, že u této rovnice mohou natat pouze tři možnosti, které závisí na vztazích mezi uvedenými konstantami.

Věta 1 (Master theorem)**.** *Když:* $T(n) = aT(n/b) + O(n^d)$ *pro nějaké* $a > 0$, $b > 1$ *a* $d \geq 0$
potom:

$$T(n) = \begin{cases} O(n^d) & \text{když } d > \log_b a \\ O(n^d \log n) & \text{když } d = \log_b a \\ O(n^{\log_b a}) & \text{když } d < \log_b a \end{cases}$$

Důkaz. Pro zjednodušení předpokládejme, že n je mocninou b, což nebude mít významný vliv na výslednou složitost[2]. Všimněmě si, že velikost subproblémů klesá s faktorem b. Takovýto algoritmus tedy tvoří strom o výšce

[2]Neboť n je nanejvýš o násobek b vzdálený od nějaké mocniny b

$\log_b n$, jehož větvící faktor je a. V tomto stromě je k-tá vrstva tvořena a^k podproblémy velikosti n/b^k. Celková práce provedená na k-té vrstvě je

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k.$$

Jak se k zvětšuje od 0 (kořene) k $\log_b n$ (listům), tato čísla tvoří geometrickou řadu s poměrem a/b^d. Hledání souču takovéto řady v Big-O notaci je snadné, a dělí se na tři případy.

1. Poměr je menší než 1.
Řada je klesající a její suma v Big-O notaci je dána jejím prvním termem, $O(n^d)$.

2. Poměr je roven 1.
V tomto případě jsou všechny $O(\log n)$ termy této řady rovny $O(n^d)$.

3. Poměr je větší než 1.
Řada je rostoucí a její suma je dána jejím posledním termem, $O(n^{\log_b a})$:

$$n^d\left(\frac{a}{b^d}\right)^{\log_b n} = n^d\left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}.$$

□

Tento důkaz najdeme v [6].

3 Karatsubův algoritmus

Ruský matematik Andrej Kolmogorov, vznesl kolem roku 1956 doměnku, že tradiční algoritmus pro násobení je optimální, a že nelze násobit přirozená čísla s lepší složitostí než je $O(n^2)$. Nejspíše z toho důvodu, že kdyby lepší algoritmus existoval, už by jej do té doby někdo vymyslel. Tento odhad obhajoval také na podzimním semináři r.1960 na Moskevské univerzitě, kde problém zaujal jednoho z jeho studentů Anatoliho Karatsubu. Tehdy 23 letý Karatsuba, přišel, dle svých slov do týdne, na algoritmus, jehož složitost je $O(n^{log_2 3})$. Kolmogorova tento algoritmus nadchnul. Přednášel Karatsubův

výsledek na různých konferencích[3] a v r.1962 jej publikoval v krátkém článku, spolu s výsledkem Yuriho Ofmana[4]. Ačkoliv byl Karatsuba uveden jako spoluautor, dle svých slov se o existenci článku dozvěděl až při obdržení jeho dotisku.[13]

Karatsubův algoritmus byl ve své době překvapivým objevem, jehož publikace rozpoutala intenzivní zájem o hledání efektivnějších násobících metod. Přesto ale bývá jeho autorství v některých učebních textech opomíjeno (př. [1],[6]), a algoritmus je uváděn např. pouze jako "Rozděl a panuj násobící algoritmus". Pravděpodobně i z toho důvodu, že použitá metoda byla známá již Carlu Fridrichu Gaussovi(1777-1855) v kontextu násobení komplexních čísel.

Gauss si všiml, že pro výpočet produktu dvou Komplexních čísel:

$$(a + bi)(c + di) = ac - bd + (bc + ad)i \tag{3.1}$$

se zdá být zapotřebí čtyř násobení reálných čísel, ačkoliv by stačila pouze tři: ac, bd a $(a + b)(c + d)$. Neboť:

$$bc + ad = (a + b)(c + d) - ac - bd. \tag{3.2}$$

Stejný proces lze aplikovat i na libovolná přirozená čísla v binárním, decimálním, či jiném pozičním zápisu. Předpokládejme pro jednoduchost, že x a y jsou n-bitová binární čísla, a že n je mocninou 2. Trik spočívá v tom, že každé z čísel x a y rozdělíme na dvě poloviny, se kterými následně nakládáme jako se samostatnými n/2-bitovými čísly.

$$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R \quad \begin{array}{l}\text{(v případě decimálních čísel,}\\ \text{by místo } 2^{n/2} \text{ bylo } 10^{n/2}\text{)}\end{array}$$

$$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R$$

Součin x a y potom můžeme zapsat takto:

$$\begin{aligned} xy &= (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) \\ &= 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R, \end{aligned}$$

[3]Např. Proceedings of the International Congress of Mathematicians 1962", pp. 351–356, a také "6 Lectures delivered at the International Congress of Mathematicians in Stockholm, 1962".

[4]Multiplication of Multidigit Numbers on Automata, Doklady Akad. Nauk SSSR Vol. 145, No. 2, pp 293-294.

Kde je patrná podobnost s (3.1) a stačí za $(x_L y_R + x_R y_L)$
substituovat $((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R)$, tak jak je navrženo v (3.2).

Karatsuba na tomto základu postavil rekurzivní algoritmus:

`karatsuba(x,y)`

`input: binárně zapsaná přirozená čísla` x `a` y
`output: jejich součin`

$n =$ `max(délka` x`, délka` y`)`
`if` $n = 1$`: return` xy

$x_L, x_R =$ `levých` $[n/2]$ `, pravých` $[n/2]$ `bitů z` x
$y_L, y_R =$ `levých` $[n/2]$ `, pravých` $[n/2]$ `bitů z` y

$P_1 =$ `karatsuba(`x_L`,`y_L`)`
$P_2 =$ `karatsuba(`x_R`,`y_R`)`
$P_3 =$ `karatsuba(`$x_L + x_R$`,`$y_L + y_R$`)`
`return` $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$

Součty se provedou v lineárním čase, stejně jako násobení mocninou 2 (které je pouze posunem do leva). Výpočetně nejnáročnější složkou algoritmu jsou tedy 3 násobení: $x_L y_L$, $x_R y_R$ a $(x_L + x_R)(y_L + y_R)$, tj 2 násobení čísly majícími $n/2$ cifer a jedno násobení čísly mjícími max $n/2 + 1$ cifer, které se provedou rekurzivně. Karatsuba v [13] dokazuje složitost algoritmu substituční metodou, při které zohledňuje riziko navýšení o jeden bit u třetího součinu. Standardně se toto riziko opomíjí, neboť nemá vliv na výslednou složitost. Složitost algoritmu můžeme zapsat rekurzivní relací:

$$T(n) = 3T(n/2) + O(n).$$

Z Master Theoremu dostaneme: $a = 3$, $b = 2$ a $d = 1$, tedy $d < \log_b a$, tudíž složitost tohoto algoritmu je $O(n^{\log_2 3})$, což je přibližně $O(n^{1.59})$. V praxi není potřeba dostat se na samotné dno rekurze, neboť většina procesorů provede 16-ti či 32-bitové násobení jako jednu operaci. [6]

4 Polynomy

Polynom na proměnné x v algebraickém poli F reprezentuje funkci $A(x)$, kterou lze vyjádřit jako formální součet:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j.$$

Hodnoty $a_0, a_1, \ldots, a_{n-1}$ jsou koeficienty polynomu. Tyto koeficienty, spolu s proměnnou x jsou prvky F, které je typicky množinou komplexních čísel $\mathbb{C}$. Polynom $A(x)$ má stupeň k /je řádu k, když jeho nejvyšší nenulový koeficient je a_k. Píšeme $Deg(A) = k$. [5]

4.1 Násobení polynomů

Produkt dvou polynomů $A(x)$ a $B(x)$, kde $Deg(A) = m$ a $Deg(B) = n$, je polynom $C(x)$, pro který platí $C(x) = A(x)B(x)$ pro všechna $x \in F$. $Deg(C) = m + n$.

$$A(x) \times B(x) = \sum_{i=0}^{m} a_i x^i \cdot \sum_{i=0}^{n} b_i x^i = \sum_{i=0}^{m+n} \left(\sum_{j=0}^{i} a_j b_{i-j} \right) x^i,$$

tedy:

$$C(x) = \sum_{i=0}^{m+n} c_i x^i, \quad kde \quad c_i = \sum_{j=0}^{i} a_j b_{i-j}.$$

Čerpáno z [5].

4.2 Ohodnocení polynomu

Ohodnocení polynomu je výpočet jeho hodnoty pro nějaké konkrétní x . Řekněme že $x = v$, pro nějaké reálné číslo v. Abychom získali hodnotu $A(v)$ potřebujeme vynásobit $a_i v^i$ pro $i = 0, 1, 2, \ldots, n.$, a následně tyto produkty sečíst. Pro takový proces je zapotřebí $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ násobení a n sčítání. Počet násobení můžeme významně zredukovat tím, že budeme mocniny v počítat iterativně: $v^0 = 1, v^1 = v$, a $v^{i+1} = v^i v$. Potom bude zapotřebí pouze n násobení pro získání všech mocnin v, a n násobení pro výpočet produktu $a_i v^i$. Celkový počet operací pro výpočet $A(v)$ by tedy byl $2n$ násobení + n sčítání. Tento proces lze nicméně ještě jednoduše vylepšit, a to Hornerovou metodou.[16]

4.2.1 Hornerův algoritmus

Polynom $A(x)$ můžeme, díky distributivitě násobení, přepsat do následující podoby:

$$\begin{aligned} A(x) &= a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \\ &= (\ldots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0. \end{aligned}$$

Na této rovnici si všimněme, že hodnotu $A(v)$ můžeme získat následujícími kroky: $s_0 = a_n$, $s_1 = a_n v + a_{n-1}$, $s_2 = s_1 v + a_{n-2}$, $\ldots$, $s_i = s_{i-1} v + a_{n-1}$.

Z čehož dostaneme algoritmus:

```
Horner(A(x),n,v)
s = a_n
for i in range(1,n):
        s = s × v + a_{n-i}
return s
```

Pro ohodnocení polynomu Hornerovým algoritmem je zapotřebí pouze n násobení a n sčítání. [16]

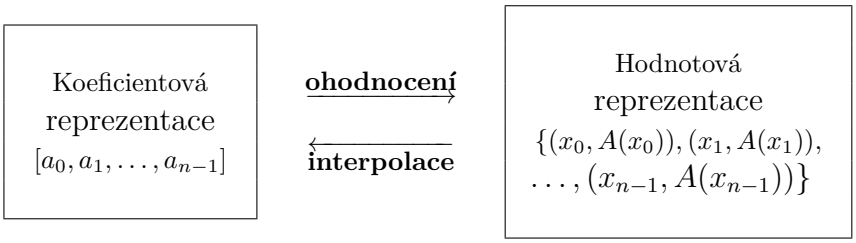
4.3 Reprezentace polynomu

Mějme polynom

$$A(x) = \sum_{j=0}^{n-1} a_j x^j.$$

Koeficientová reprezentace polynomu je vektor jeho koeficientů, tj. polynom řádu n lze reprezentovat n-prvkovým vektorem $[a_0, a_1, \ldots, a_{n-1}]$. Každý polynom má právě jednu koeficientovou reprezentaci.

Hodnotová reprezentace (neboli point-value reprezentace) takového polynomu je n-prvková množina, obsahující dvojice tvaru: $(x, A(x))$ pro n různých x. Tj.: $\{(x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_{n-1}, A(x_{n-1}))\}$.
Zvolená x_i mohou být libovolná, jedinou podmínkou je, aby byla různá. Polynom tedy může mít mnoho hodnotových reprezentací.



Obrázek 4.1: Reprezentace polynomu a převody mezi nimy.

Pro získání hodnotové reprezentace tedy potřeujeme polynom ohodnotit na n různých bodech. Určení koeficientů polynomu z jeho hodnotové reprezentace, se nazývá interpolace. Pro jasnější představu o interpolaci si můžeme rozepsat proces ohodnocení polynomu jako přenásobení čtvercové matice tvořené zvolenými x_i se sloupcovým vektorem koeficientů polynomu. Viz obr.4.2

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Obrázek 4.2: Ohodnocení polynomu na $x_0, x_1, \ldots, x_{n-1}$.

Důkaz, že z hodnotové reprezentace můžeme jednoznačně získat koeficientovou reprezentaci, vychází z pozorování, že prostřední matice v 4.2, (označme ji M), je případ **Vandermondovy matice**. U takové matice umíme určit její determinant: $det(V) = \prod_{0 \leq k < j \leq n}(x_j - x_k)$, který je nenulový právě tehdy, když jsou všechna x_i různá. Matice je tedy invertibilní, tj. dokážeme sestrojit k ní inverzní matici M^{-1}. Po přenásobení M^{-1} sloupcovým vektorem s odpovídajícími hodnotami (levý sloupcový vektor na obr.4.2), získáme koeficientovou reprezentaci polynomu. Podrobný důkaz najdeme např. v [16], nebo v [5].

5 Konvoluce

5.1 Lineární Konvoluce

Definice. Nechť $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ a $\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]$

Lineární konvoluci dvou vektorů **a** a **b** značíme $\mathbf{a} \otimes \mathbf{b}$
a jejím výsledkem je vektor $\mathbf{c} = [c_0, c_1, \ldots, c_{2n-1}]$, definovaný takto:

$$\mathbf{c} = \mathbf{a} \otimes \mathbf{b}, \qquad \mathbf{c}_i = \sum_{j=0}^{n-1} \mathbf{a}_j \mathbf{b}_{i-j} \qquad \text{pro i = 0, 1, . . . , 2n-1}$$

Přičemž $a_k = b_k = 0$, když $k < 0$ nebo $k \geq n$.

Tudíž:

$$c_0 = a_0 b_0, \quad c_1 = a_0 b_1 + a_1 b_0, \quad c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0, \quad atd.$$

Všimněme si, že $c_{2n-1} = 0$, a je zahrnut pouze pro symetrii.

Pozorování 1. *Produkt dvou polynomů $A(x)$ a $B(x)$ řádu $n-1$,*

$$A(x) = \sum_{i=0}^{n-1} a_i x^i \qquad a \qquad B(x) = \sum_{j=0}^{n-1} b_j x^j,$$

bude řádu 2n-2

$$A(x)B(x) = \sum_{i=0}^{2n-2} \left[\sum_{j=0}^{i} a_j b_{i-j} \right] x^i.$$

Zvolíme-li za vstupní vektory lineární konvoluce koeficientové reprezentace dvou polynomů, výsledný vektor bude roven koeficientové reprezentaci jejich produktu.[1]

Příklad: 1. *Nechť* $\mathbf{a} = [4, 3, 2, 1]$ *a* $\mathbf{b} = [8, 7, 6, 5]$ *jsou vstupními vektory pro lineární konvoluci. Potom* $\mathbf{c} = \mathbf{a} \otimes \mathbf{b} = [c_0, c_1, \ldots, c_7]$. *Hodnoty* $[c_0, \ldots, c_7]$ *jsou rozepsány v tabulce 1.*

$c_0 = (a_0b_0)$ *+ 0 + 0 + 0*	$(4 \cdot 8)$	*32*
$c_1 = (a_0b_1) + (a_1b_0)$*+ 0 + 0*	$(4 \cdot 7) + (3 \cdot 8)$	*52*
$c_2 = (a_0b_2) + (a_1b_1) + (a_2b_0)$*+ 0*	$(4 \cdot 6) + (3 \cdot 7) + (2 \cdot 8)$	*61*
$c_3 = (a_0b_3) + (a_1b_2) + (a_2b_1) + (a_3b_0)$	$(4 \cdot 5) + (3 \cdot 6) + (2 \cdot 7) + (1 \cdot 8)$	*60*
c_4 *= 0+* $(a_1b_3) + (a_2b_2) + (a_3b_1)$	$(3 \cdot 5) + (2 \cdot 6) + (1 \cdot 7)$	*34*
c_5 *= 0 + 0 +* $(a_2b_3) + (a_3b_2)$	$(2 \cdot 5) + (1 \cdot 6)$	*18*
c_6 *= 0 + 0 + 0 +* (a_3b_3)	$(1 \cdot 5)$	*5*
c_7 *= 0 + 0 + 0 + 0*	*0*	*0*

Tabulka 1: Příklad rozepsání indexů u lineární konvoluce

Pozorování 2. *Lineární konvoluce je totožná s klasickým násobením pod sebou, bez tzv. přenášení:*

			[1	2	3	4]
		*	[5	6	7	8]
			8	16	24	32
		7	14	21	28	
	6	12	18	24		
5	10	15	20			
5	16	34	60	61	52	32

5.2 Cyklická konvoluce

Definice. Cyklickou konvoluci dvou vektorů $\mathbf{a}$ a $\mathbf{b}$ značíme $\mathbf{c} = \mathbf{a} \oplus \mathbf{b}$, kde: $\mathbf{a} = [a_0, \ldots, a_{n-1}]$, $\mathbf{b} = [b_0, \ldots, b_{n-1}]$, $\mathbf{c} = [c_0, \ldots, c_{n-1}]$, a c_i je definované předpisem:

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} + \sum_{j=i+1}^{n-1} a_j b_{n+i-j}$$

Přičemž $a_k = b_k = 0$, když $k < 0$ nebo $k \geq n$.

Pozorování 3. *Cyklickou konvoluci (délky n) získáme z lineární konvoluce (délky $2n-1$) jednoduše tak, že k prvním n členům ($c_0, \ldots, c_{n-1}$) pod sebou přičteme zbylých $n-1$ členů ($c_n, \ldots, c_{n-2}$)*

Cyklická konvoluce výše uvedeného příkladu, tedy bude vypadat takto:

	60	*61*	*52*	*32*
+		*5*	*16*	*34*
	60	*66*	*68*	*66*

5.3 Negacyklická konvoluce

Definice. Negacyklickou konvoluci dvou vektorů $\mathbf{a}$ a $\mathbf{b}$ značíme $\mathbf{c} = \mathbf{a} \ominus \mathbf{b}$, kde: $\mathbf{a} = [a_0, \ldots, a_{n-1}]$, $\mathbf{b} = [b_0, \ldots, b_{n-1}]$, $\mathbf{c} = [c_0, \ldots, c_{n-1}]$, a

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j}$$

Přičemž $a_k = b_k = 0$, když $k < 0$ nebo $k \geq n$.

Pozorování 4. *Negacyklickou konvoluci můžeme získat z lineární konvoluce stejně jednoduše jako cyklickou, s tím rozdílem, že od prvních n členů $(c_0, \ldots, c_{n-1})$ posledních n-1 členů $(c_n, \ldots, c_{n-2})$ odečteme.*

Negacyklická konvoluce výše uvedeného příkladu, tedy bude vypadat takto:		*60*	*61*	*52*	*32*
	-		*5*	*16*	*34*
		60	*56*	*36*	*-2*

6 Fourierova transformace

Fourierova transformace je většinou definována na komplexních číslech. Pro naše účely lépe poslouží obecná definice na libovolném komutativním okruhu.

6.1 DFT

Definice. Diskrétní Fourierova Transformace
Nechť $\mathbf{R} = (R, +, \cdot, 0, 1)$, ve kterém najdeme prvek $\omega \in R$, splňující:

1. $\omega \neq 1$
2. $\omega^n = 1$
3. $\sum_{j=0}^{n-1} \omega^{jp} = 0$, pro $1 \leq p < n$

Tomuto prvku říkáme primitivní n-tý kořen jedné, a všechny prvky $\omega^0, \omega^1, \ldots, \omega^{n-1}$ jsou n-tými kořeny jedné.

Nechť $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ je n-rozměrný sloupcový vektor, jehož elementy $\in \mathbf{R}$. Předpokládejme, že n má multiplikativní inverz v $\mathbf{R}$, a že $\mathbf{R}$ má primitivní n-tý kořen jedničky. Nechť A je $n \times n$ matice, taková, že $A[i, j] = \omega^{ij}$, pro $0 \leq i, j < n$. Potom $F(\mathbf{a}) = A \cdot \mathbf{a}$, je vektor, jehož i-tý prvek, $b_i = \Sigma_{k=0}^{n-1} a_k \omega^{ik}$, pro $(0 \leq i < n)$, je Fourierovou transformací vektoru $\mathbf{a}$. Viz. obr. 6.1. Matice A je nesingulární, tj. existuje k ní inverzní matice A^{-1}, která má jednoduchou fomu, danou následujícím lemmatem. [1]

Lemma 1. *Nechť $\mathbf{R}$ je komutativní okruh, ve kterém má n multiplikativní inverz a ω je primitivní n-tý kořen 1. Nechť A je $n \times n$ matice, jejíž ij-tý element je ω^{ij} pro $0 \leq i, j < n$. Pak A^{-1} existuje a ij-tý elemet A^{-1} je $(1/n)\omega^{-ij}$.*

Důkaz:
Nechť δ^{ij} je 1 když $i = j$, a 0 když $i \neq j$. Stačí ukázat, že pokud je A^{-1}

$$F(\mathbf{a}) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

Obrázek 6.1: Diskrétní Fourierova transformace

definováno jako výše, potom $A \times A^{-1} = I_n$, tedy, že ij-tý element $A \times A^{-1}$ je

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{ik} \omega^{-kj} = \delta_i j \quad pro \quad 0 \leq i, j < n \tag{6.1}$$

Když $i = j$, tak se levá strana 6.1 zredukuje na

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^0 = 1.$$

Když $i \neq j$, nechť $q = i - j$. Potom se levá strana 6.1 zredukuje na

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{qk}, \quad -n < q < n, q \neq 0.$$

Když $q > 0$, dostaneme

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{qk} = 0,$$

jelikož ω je n-tý kořen 1 (vyplývá z 3. požadavku na ω). Když $q < 0$, potom přenásobením $\omega^{-q(n-1)}$, přeorganizováním termů v sumě, a substituováním $-q$ za q, dostaneme výraz

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{qk}, \quad 0 < q < n,$$

který má opět hodnotu 0, neboť ω je n-tý primitivní kořen 1. Čímž jsme dokázali, že platí rovnice 6.1. □

Tento důkaz najdeme v [1].

Důsledek:
Vektor $F^{-1}(\mathbf{a}) = A^{-1}$, jehož i-tý komponent je

$$\frac{1}{n}\sum_{k=0}^{n-1} a_k \omega^{-ik} \quad 0 \leq i < n,$$

je *inverzní diskrétní Fourierova transformace* vektoru $\mathbf{a}$. Inverzní transformace transformace $\mathbf{a}$ je $\mathbf{a}$. $F^{-1}(F(\mathbf{a})) = \mathbf{a}$.

Počítání Fourierovi transformace vektoru $[a_0, a_1, \ldots, a_{n-1}]$ je ekvivalentní konvertování koeficientové reprezentace polynomu $\sum_{i=0}^{n-1} a_i x^i$ do jeho hodnotové reprezentace na bodech $\omega^0, \omega^1, \ldots, \omega^{n-1}$. Stejně tak inverzní Fourierova transformace je ekvivalentní interpolaci tohoto polynomu.

Pro tuto sekci čerpáno z [1].

6.2 Věta o Konvoluci

Následující věty říkají, že pro výpočet lineární, cyklické i negacyklické konvoluce můžeme použít Fourierovu transformaci. Jelikož tato transformace slouží k převodu polynomu z jeho koeficientové do jeho hodnotové reprezentace, stačí když prvky výsledných vektorů po dvojicích přenásobíme na odpovídajících hodnotách (*componentwise/ pairwise product*), abychom dostali hodnotovou reprezentaci produktu původních polynomů. Následně použijeme inverzní transformaci, pro převod do koeficientové reprezentace, která je zároveň lineární konvolucí vstupních vektorů. Jediný problém je v tom, že produkt dvou polynomů řádu (n-1) je řádu (2n-2) a pro jeho reprezentaci je potřeba ohodnocení na (2n-1) různých bodech. Tato technikalita se řeší rozšířením vstupních vektorů o n nul.

Věta 2. *(Konvoluční theorém)*
Nechť:

$$\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}, 0, \ldots, 0]^T$$

a

$$\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}, 0, \ldots, 0]^T$$

jsou sloupcové vektory délky 2n. Nechť

$$F(\mathbf{a}) = [a'_0, a'_1, \ldots, a'_{2n-1}]^T$$

a

$$F(\mathbf{b}) = [b'_0, b'_1, \ldots, b'_{2n-1}]^T$$

jsou jejich Fourierovy transformace. Potom $\mathbf{a} \otimes \mathbf{b} = F^{-1}(F(\mathbf{a}) \times F(\mathbf{b}))$

Důkaz. Jelikož $a_i = b_i = 0$ pro $n \leq i < 2n$, všiměme si že $0 \leq l < 2n$,

$$a'_l = \sum_{j=0}^{n-1} a_j \omega^{lj} \qquad a \qquad b'_l = \sum_{k=0}^{n-1} b_k \omega^{lk}.$$

A tedy

$$a'_l b'_l = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j b_k \omega^{l(j+k)}. \tag{6.2}$$

Nechť $\mathbf{a} \otimes \mathbf{b} = [c_0, c_1, \ldots, c_{2n-1}]^T \quad a \quad F(\mathbf{a} \otimes \mathbf{b}) = [c'_0, c'_1, \ldots, c'_{2n-1}]^T$.

Jelikož $c_p = \sum_{j=0}^{2n-1} a_j b_{p-j}$ dostaneme

$$c'l = \sum_{p=0}^{2n-1} \sum_{j=0}^{2n-1} a_j b_{p-j} \omega^{lp}. \tag{6.3}$$

Záměnou sum v 6.3 a substitucí k za $p - j$ dostaneme

$$c'_l = \sum_{j=0}^{2n-1} \sum_{k=-j}^{2n-1-j} a_j b_k \omega^{l(j+k)}. \tag{6.4}$$

Jelikož $b_k = 0$ pro $k < 0$, můžeme zvýšit spodní limit vnitřní sumy na $k = 0$. Stejně tak $a_j = 0$ pro $j \geq n$, tudíž můžeme snížit horní limit vnější sumy na $n - 1$. Horní limit vnitřní sumy je minimálně n, nehledě na to jaká je hodnota j. Z toho důvodu můžeme přepsat horní limit na $n - 1$, jelikož $b_k = 0$ pro $k \geq n$. Po těchto změnách bude 6.4 identické s 6.2 a tedy $c'_l = a'_l b'_l$. Tím jsme dokázali, že $F(\mathbf{a} \otimes \mathbf{b}) = F(\mathbf{a}) \cdot F(\mathbf{b})$, z čehož plyne, že $\mathbf{a} \otimes \mathbf{b} = F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$. [1]

□

Kdybychom tento proces provedli bez rozšíření vektorů **a** a **b** o n nul, resp. ohodnotili dva polynomy řádu $n-1$ na n-tých kořenech jedné a následně provedli *pairewise* přenásobení, dostali bychom n-prvkový vektor, který by se po interpolaci rovnal cyklické konvoluci vstupních vektorů.

Věta 3. *Nechť* $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T$ *a* $\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]^T$ *jsou dva vektory délky n. Nechť* ω *je n-tý primitivní kořen jedné, a nechť* $\psi^2 = \omega$. *Předpokládejme že n má multiplikativní invers.*

1. *Cyklická konvoluce* **a** *a* **b** *je dána* $F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$.

2. *Nechť* $\mathbf{d} = [d_0, d_1, \ldots, d_{n-1}]^T$ *je negacyklická konvoluce* $\mathbf{a}$ *a* $\mathbf{b}$. *Nechť* $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$ *a* $\hat{\mathbf{d}}$ *jsou* $[a_0, \psi a_1, \ldots, \psi^{n-1} a_{n-1}]^T$, $[b_0, \psi b_1, \ldots, \psi^{n-1} b_{n-1}]T$ *a* $[d_0, \psi d_1, \ldots, \psi^{n-1} d_{n-1}]T$. *Potom* $\hat{\mathbf{d}} = F^{-1}(F(\hat{\mathbf{a}}) \cdot F(\hat{\mathbf{b}}))$

Důkaz je analogický jako u věty 2 s pozorováním, že $\psi^n = -1$. [1]

6.3 FFT

Fast Fourier Transform - Rychlá Fourierova transformace, je algoritmus, který počítá Diskrétní Fourierovu transformaci v čase $O(n \cdot log(n))$. Tento algoritmus publikovali John Cooley s Johnem Tukym r. 1965, převážně z obavy, aby si tuto metodu někdo nepatentoval. Tukey, který na metodu pravděpodobně přišel, se do publikování příliš nehrnul. Byl totiž přesvědčen, že jde o jednoduché pozorování, které je pravděpodobně již známé. To bylo v té době celkem běžné, neboť algoritmy byly považovány za druhořadé matematické objekty, nepříliš hodné vážné pozornosti. Nicméně Tukeyho předpoklad se ukázal být pravdivým. Jak se později zjistilo, britští inženýři používali FFT pro ruční výpočty již koncem 30tých let. A dokonce Fridrichu Gausovi byla tato metoda známá již kolem r. 1800. [6]

6.3.1 FFT - Rekurzivní metoda pro ohodnocení polynomu

Připomeňme si, že pro získání hodnotové reprezentace polynomu řádu $n-1$, jej potřebujeme ohodnotit na n libovolných, avšak různých bodech. Jaké body zvolit, abychom si práci co nejvíce usnadnili a urychlili? Všimněme si, že pokud zvolíme pozitivní a negativní dvojice

$$x_0, x_1, \ldots, x_{n/2-1}, -x_0, -x_1, \ldots, -x_{n/2-1},$$

zkrátíme tím počet mocnění i násobení na polovinu. Sudé mocniny x jsou totiž stejné jako sudé mocniny $-x$, a liché mocniny $-x$ se od mocnin x liší pouze záporným znaménkem. To bude platit i po přenásobení mocniny koeficientem. Polynom, ohodnocený na x_0, se bude od téhož polynomu ohodnoceného na $-x_0$, lišit pouze znaménkem u lichých termů (první term je 0-tý tedy sudý).

Rozdělme si polynom na sudé a liché termy na příkladu:

$$\begin{aligned} 3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 &= (3 + 6x^2 + x^4) + (4x + 2x^3 + 10x^5) \\ &= (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4) \end{aligned}$$

Po vytknutí x před závorku s lichými termy, jsou nyní v závorkách vepsány

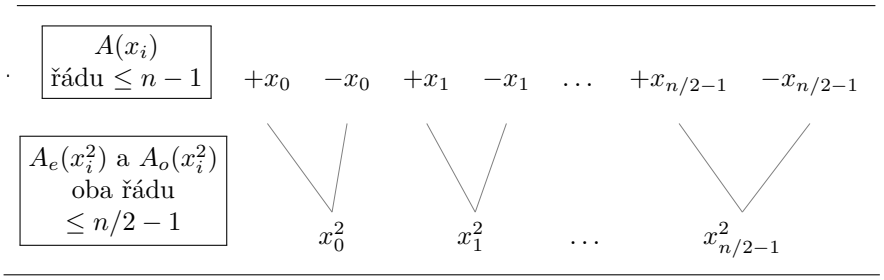
2 polynomy, obsahující pouze sudé mocniny x. Můžeme napsat:

$$A(x) = A_e(x^2) + xA_o(x^2)$$

Kde $A_e(\cdot)$ a $A_o(\cdot)$ jsou řádu $\leq n/2 - 1$ (pro jednoduchost předpokládáme že n je sudé. Ohodnocení $A(x)$ na $\pm x_i$ můžeme zapsat následovně:

$$\begin{aligned} A(x_i) &= A_e(x_i^2) + A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - A_o(x_i^2). \end{aligned}$$

Problém ohodnocení polynomu $A(x)$ na n pozitivních a negativních bodech $\pm x_0, \ldots, \pm x_{n/2-1}$, tak zredukujeme na dva podproblémy, tedy ohodnocení polynomů $A_e(y)$ a $A_o(y)$ řádu $n/2 - 1$, na pouze $n/2$ bodech: $x_0^2, \ldots, x_{n/2-1}^2$. Viz. obr.: 6.2



Obrázek 6.2: Ohodnocení A_e a A_o na $x_0^2, \ldots, x_{n/2-1}^2$ obrázek z [6]

Kdybychom tuto metodu aplikovali rekurzivně, dostali bychom algoritmus typu "rozděl a panuj", jehož časová náročnost by byla vyjádřitelná relací:

$$T(n) = 2T(n/2) + O(n),$$

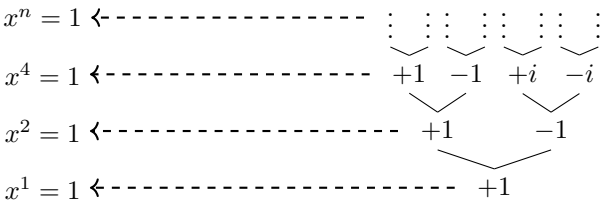
což je přesně $O(n \log_2 n)$, jak plyne z *Master theoremu*.

Nicméně použití této metody rekurzivně nebude tak jednoduché, neboť trik s ohodnocením na pozitivních a negativních párech může být použit pouze na vrchní hladině rekurze. Potřebovali bychom, aby $x_0^2, \ldots, x_{n/2-1}^2$ byly také pozitivně a negativně spárované, tedy potřebujeme druhé mocniny, které jsou záporné. To můžeme vyřešit pomocí Komplexních čísel, (nebo jak později ukážeme, též prováděním výpočtu na nějakém konečném tělese).

Tento algoritmus můžeme zakreslit binárním stromem, v jehož kořenu je polynom řádu $n - 1$, jenž chceme ohodnotit na n bodech (předpokládejme,

že n je mocninou 2). Každý potomek v tomto stromě je polynom mající o polovinu méně termů než jeho rodič. Je-li rodič řádu d, potomek je řádu $(d-1)/2$, avšak pouze za předpokladu, že jej budeme ohodnocovat na druhých mocninách $x_0, \ldots, x_d$, jimiž chceme ohodnotit jeho rodiče. Požadujeme, aby $x_0, \ldots, x_d$ tvořili pozitivní a negativní páry na všech úrovních rekurze, takže se při každém vnoření, resp. rozvětvení stromu, (kdy se x_i mocní na druhou) počet x_i snižuje na polovinu (viz. obr. 6.2), stejně jako počet termů v polynomu.

Na dno rekurze se tedy dostaneme po $\log_2(n)$ vnořeních, kdy se strom rozvětví na jednoprvkové listy. V těchto listech jsou polynomy řádu 0, repspektive konstanty. U polynomů řádu 0 nedává smysl bavit se o jejich ohodnocení, avšak z popisu tohoto algoritmu bychom tyto listy měli ohodnotit na jednom bodě, a to na: $x_i^{2^{\log_2 n}} = x_i^n$. Z tohoto popisu vyplývá 1 jako vhodný kandidát pro použití na dně naší rekurze, přičemž n-té kořeny jedné musejí být použity na jejím povrchu.



Obrázek 6.3

Na obrázku 6.3 vidíme, že n-té kořeny 1 tvoří pozitivní a negativní páry. Máme přesně n n-tých kořenů 1, tedy komplexních čísel splňujících $z^n = 1$, a jejich mocněním dostaneme $n/2$ kořenů 1, splňujících $z^{n/2} = 1$. Názorněji viz. tabulka 2. S použitím takovýchto čísel, a za předpokladu že n je mocninou 2^5 , bude tato *rozděl a panuj* procedura fungovat. Výsledný algoritmus se nazývá Rychlá Fourierova transformace (FFT viz. obr. 6.4), neboť stejně jako DFT ohodnocuje polynom na $\omega^0, \ldots, \omega^{n-1}$ pro $\omega =$ primitivní kořen 1.

Pro výpočet inverzní FFT (FFT^{-1}) stačí když v algoritmu z obr. 6.4 změníme ω za ω^{-1} a prvky výsledného vektoru vydělíme n, tedy délkou vstupního vektoru (plyne z lemmatu 1). [6] [1]

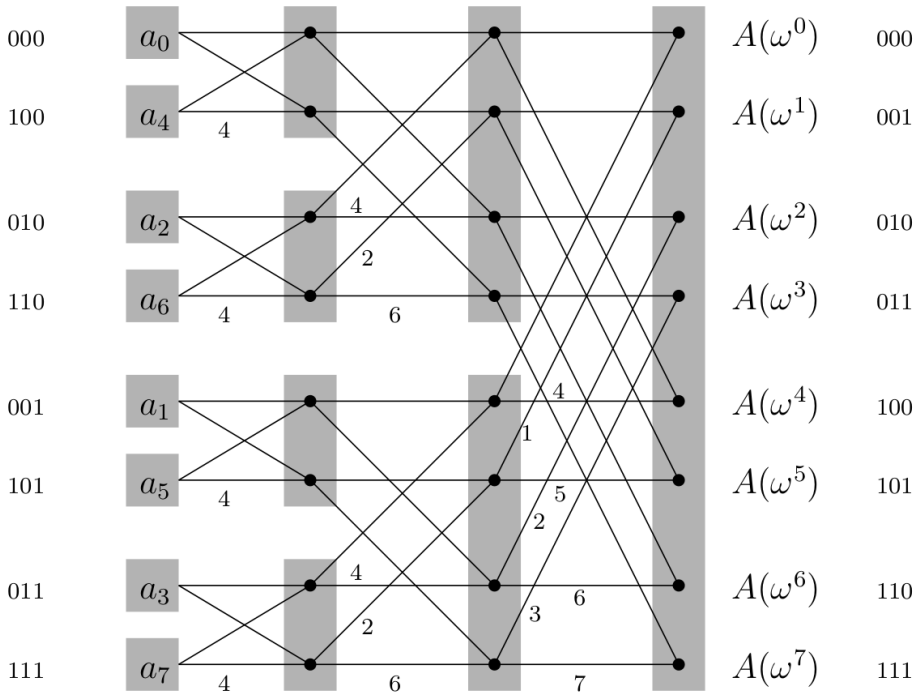
[5]koeficientovou reprezentaci jejíž délka není mocninou 2, můžeme rozšířit o nulové koeficienty pro vyšší mocniny, tak, aby byl tento požadavek splněn. V současné době jsou známy obecnější varianty algoritmu, se kterými tento požadavek již není třeba (např. [9]).

`FFT(a,`ω`)`

`input:` Koeficientová reprezentace polynomu $A(x)$ řádu $< n$,
$a = [a_0, a_1, \ldots, a_{n-1}]$, `kde` n `je mocninou 2.`
`output:` Hodnotová reprezentace $[A(\omega^0), A(\omega)^1, \ldots, A(\omega)^{n-1}]$

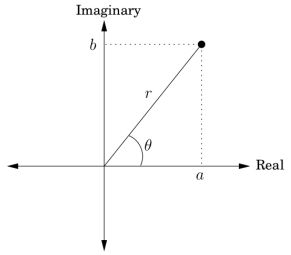
`if len(a) == 1: return a`
$(s_0, s_1, \ldots, s_{n/2-1}) =$ `FFT`$((a_0, a_2, \ldots, a_{n-1}), \omega^2)$
$(s'_0, s'_1, \ldots, s'_{n/2-1}) =$ `FFT`$((a_1, a_3, \ldots, a_{n-1}), \omega^2)$
`For j = 0 to n/2-1 :`
$r_j = s_j + \omega^j s'_j$
$r_{j+n/2} = s_j - \omega^j s'_j$
`return` $(r_0, r_1, \ldots, r_{n-1})$

Obrázek 6.4: FFT Algoritmus [6]



Obrázek 6.5: FFT Diagram [6]

Shrnutí základních faktů o komplexních číslech



Komplexní rovina

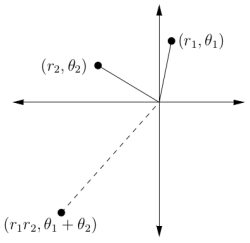
číslo $z = a + bi$ je zakresleno na pozici (a, b)

Polární koordináty:

$z = r(\cos\theta + i\sin\theta) = re^{i\theta}$ [a], značíme (r, θ)

- *délka* $r = \sqrt{a^2 + b^2}$
- *úhel* $\theta \in [0, 2\pi) : \cos\theta = a/r$, $\sin\theta = b/r$.
- θ může být zredukována modulo 2π

[a] Plyne z Eulerovy rovnice $e^{i\varphi} = \cos\varphi + i\sin\varphi$



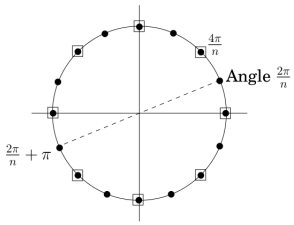
Násobení v polárních koordinátech

Délky se znásobí a úhly se sečtou:

$(r_1, \theta_1) \times (r_2, \theta_2) = (r_1r_2, \theta_1 + \theta_2)$

Pro každé $z = (r, \theta)$,

- $-z = (r, \theta + \pi)$ jelikož $-1 = (1, \pi)$.
- když z leží na jednotkové kružnici, (tj. $r = 1$), tak $z^n = (1, n\theta)$.

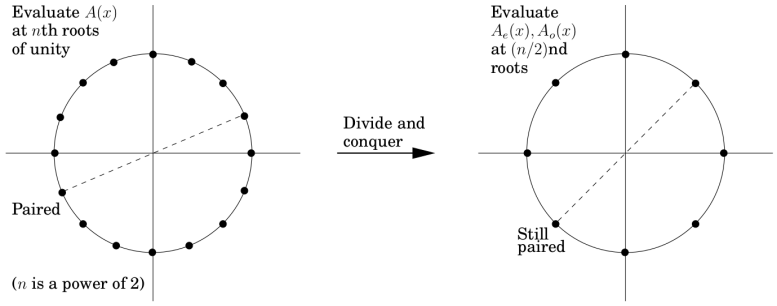


Komplexní *n*-té kořeny 1

Jak plyne z pravidla pro násobení, řešení rovnice $z^n = 1$, je $z = (1, \theta)$, kde θ je násobek $2\pi/n$. (na obr. vlevo $n = 16$)

pro sudé n:

- Tato čísla jsou pozitivně a negativně spárovaná: $-(1, 0) = (1, \theta) = (1, \theta + \pi)$
- Jejich druhé mocniny jsou $(n/2)$-té kořeny 1. Na obr. vlevo jsou zvýrazněny pomocí čtverečků.



Znázornění procesu mocnění n-tých kořenů 1.

Tabulka 2: Obrázky i text převzaty z [6]

Algoritmus FFT na vstupu $a_0, \ldots, a_7$ vykreslí diagram zobrazený na obr. 6.5. V levém sloupci tohoto diagramu vidíme, že prvky vstupní posloupnosti mají přeházené pořadí. To je dáno rekurzivním rozdělováním na sudé a liché poloviny. Čísla vedle čar značí kterou mocninou omegy je násoben prvek liché posloupnosti. Pro toto přenásobení jsou ve skutečnosti použity pouze $\omega^0, \ldots, \omega^3$, které jsou našimi pozitivními x_i. Ostatní mocniny dostaneme z algoritmu pouhou změnou znaménka, což plyne z definice ω, coby primitivního osmého kořene 1, neboť $\omega^4 = -1$, tudíž $\omega^5 = \omega^4 \times \omega^1 = -\omega^1$, $\omega^6 = -\omega^2$ a $\omega^7 = -\omega^3$.

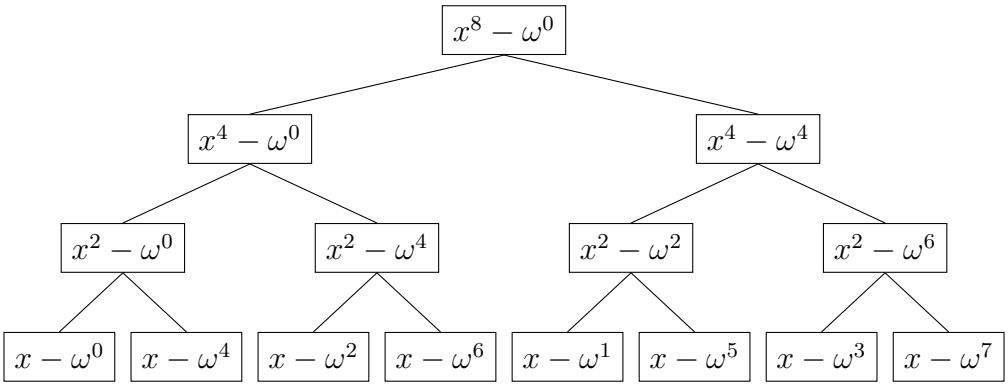
6.3.2 FFT - počítání zbytku pro $A(x)/x - \omega^i$

Trochu jinou interpretaci a podobu algoritmu FFT nabízí kniha [1]. Tato interpretace vychází z pozorování, že ohodnocení polynomu $P(x)$ na $x = a$ je ekvivalentní počítání zbytku po dělení polynomu $P(x)$ polynomem $x - a$. Neboť můžeme napsat $P(x) = (x - a)q(x) + c$, (kde c je konstanta), z čehož plyne $P(a) = c$.

Ohondocení polynomu $A(x)$ na n-tých kořenech jedné $\omega^0, \omega^1, \ldots, \omega^{n-1}$ je tedy ekvivalentní počítání zbytků po dělení $A(x)$ polynomy $(x - \omega^0), (x - \omega^1), \ldots, (x - \omega^{n-1})$. Pro urychlení takového procesu, se nejprve polynomy $x - \omega^i$ spolu po dvojicích přenásobí (tyto dvojice ještě blíže specifikujeme). Výsledných $n/2$ polynomů se taktéž po dvojicích přenásobí a tento proces se opakuje, až získáme pouze dva výsledné polynomy q_1 a q_2, z nichž každý je produktem poloviny z $x - \omega^i$. V dalším kroku dělíme $A(x)$ polynomy q_1 a q_2 a získáme zbytky r_1 a r_2, jenž jsou polynomy řádu $\leq n/2 - 1$. Nyní budeme počítat zbytek po dělení polynomů r_1 a r_2 polynomy, jenž jsou dětmi q_1, resp. q_2. Výsledky budeme tímto způsobem posouvat až k listům, kde nakonec získáme zbytky po dělení $A(x)$ polynomy $(x-\omega^0), (x-\omega^1), \ldots, (x-\omega^{n-1})$ respektive ohodnocení $A(x)$ na $\omega^0, \omega^1, \ldots, \omega^{n-1}$.

Obr. 6.6 zobrazuje výše popsaný binární strom (pro $n = 8$), ve kterém je každý vrchol produktem svých dětí. Všimněme si, že polynomy v listech jsou přeházeny (permutovány) stejným způsobem jako vstupní koeficienty u diagramu na obr. 6.5. V tomto případě je to záměrně, neboť výsledný produkt takovýchto dvojic je snadné spočítat, a zároveň všechny vzniklé polynomy jsou tvaru $x^{2t} - \omega^s$ pro nějaké s a t. To opět vyplývá z definice omegy a důsledku $\omega^4 = -1$. Všechny násobené dvojice jsou tedy ve tvaru $(a-b)(a+b)$.

Následující lemma nám poskytuje snadnou metodu pro počítání zbytku po dělení libovolného polynomu řádu $(2t-1)$ polynomem $x^t - c$, proveditelnou v $O(t)$ krocích.



Obrázek 6.6: čerpáno z [1]

Lemma 2. *Nechť*

$$P(x) = \sum_{j=0}^{2t-1} a_j x^j$$

a nechť c je konstanta. Potom zbytek po dělení $P(x)/(x^t - c)$ je

$$R(x) = \sum_{j=0}^{t-1} (a_j + c \cdot a_{j+t}) x^j.$$

Důkaz. Pozorování, že $P(x)$ můžeme vyjádřit jako

$$\left[\sum_{j=0}^{t-1} a_{j+t} \cdot x^j \right] (x^t - c) + r(x).$$

□

V knize [1] najdeme dva algoritmy pro výpočet FFT. První (viz. Obr.6.7) je spíše schématický, určený pro jasnější představu o této metodě. Druhý (viz. Obr.6.8) je zjednodušený, a pracuje pouze s koeficienty a binárním zápisem indexů a exponentů. V prvním algoritmu využijeme funkci `rev(i)`, která pro přirozené číslo i vrátí přirozené číslo j, jehož binární zápis je zrcadlově převráceným zápisem i. Tj. pro i, takové že $0 \leq i < 2^k$ jehož binární zápis je posloupnost $[d_0, d_1, \ldots, d_{k-1}]$, `rev(i)` $= j$, jehož bin. zápis bude $[d_{k-1}, d_{k-2}, \ldots, d_1, d_0]$.

Věta 4. *Algoritmus z obr. 6.7 pracuje v čase $O(n \log n)$.*

Důkaz. Řádky 6 a 7 vyžadují $O(2^m)$ kroků, pokaždé, když jsou zavolány. Pro fixní m, bude for loop na řádcích 3-7 spuštěn $n/2^{m+1}$-krát, o celkové

První schématický algoritmus FFT z knihy [1]

`input:` Vektor $a = [a_0, a_1, \ldots, a_{n-1}]$, kde $n = 2^k$ pro $k \in \mathbb{N}$

`output:` Vektor $F(a) = [b_0, b_1, \ldots, b_{n-1}]$, kde $b_i = \sum_{j=0}^{n-1} a_j \omega^{ji}$, pro $0 \leq i < n$.

1. nechť r_{0k} je $\sum_{j=0}^{n-1} a_j x^j$ — `comment:` Polynom je reprezentován svými koeficienty. Řádek 1. tedy nezahrnuje žádné výpočty. Proměnná r_{lm} reprezentuje zbytek po dělení polynomu $\sum_{j=0}^{n-1} a_j x^j$ polynomem q_{lm}.
2. `for` m `in range`$(k-1, 0)(step = -1)$`:`
3. `for` l `in range`$(0, n-1)(step = 2^{m+1})$`:`
4. nechť $r_{l,m+1}$ je $\sum j = 0^{2^{m+1}-1} a_j x^j$; `comment:` Zbytky nižších řádů budeme počítat z koeficientů $r_{l,m+1}$;
5. $s =$ `rev`$(l/2^m)$
6. $r_{lm}(x) = \sum_{j=0}^{2^m-1}(a_j + \omega^s a_{j+2^m})x^j$
7. $r_{l+2^m,m}(x) = \sum_{j=0}^{2^m-1}(a_j + \omega^{s+n/2} a_{j+2^m})x^j$
8. `for` i `in range`$(0, n-1)$`:`
9. $b_{\texttt{rev(i)}} = r_{i,0}$

Obrázek 6.7

Zjednodušený FFT algoritmus z knihy [1]

`input:` Vektor $a = [a_0, a_1, \ldots, a_{n-1}]$, kde $n = 2^k$ pro $k \in \mathbb{N}$

`output:` Vektor $F(a) = [b_0, b_1, \ldots, b_{n-1}]$, kde $b_i = \sum_{j=0}^{n-1} a_j \omega^{ji}$, pro $0 \leq i < n$.

1. `for` i `in range`$(0, 2^k - 1)$`:`
   $R[i] = a_i$
2. `for` l `in range`$(0, k - 1)$`:`
3. `for` i `in ragnge`$(0, 2^{k-1})$`:`
4. $S[i] = R[i]$
5. `for` i `in range`$(0, 2^k - 1)$`:`
   `comment:` Nechť $[d_0, d_1, \ldots, d_{k-1}]$ je binární reprezentace i.
6. $R[[d_0, \ldots, d_{k-1}]] = S[[d_0, \ldots, d_{l-1}, 0, d_{l+1}, \ldots, d_{k-1}]]$
   $+\omega^{[d_l, d_{l-1}, \ldots, d_0, 0, \ldots, 0]} S[[d_0, \ldots, d_{l-1}, 1, d_{l+1}, \ldots, d_{k-1}]]$
7. `for` i `in range`$(0, 2^{k-1})$`:`
   $b_{[d_0, \ldots, d_{k-1}]} = R[[d_{k-1}, \ldots, d_0]]$

Obrázek 6.8

složitosti $O(n)$, nehledě na velikosti m. Vnější cyklus, začínající na řádku 2, bude vykonán $\log n$-krát. Cyklus na řádku 8 neobsahuje žádné aritmetické operace. Celková složitost tohoto algoritmu je tedy $O(n \log n)$. [1]

□

Ve větě 4 předpokládáme, že n je fixní, a tudíž můžeme mocniny ω, a hodnoty s a `rev(`l`)` spočítat předem, a použít v algoritmu jako konstanty. Kdyby bylo třeba aby n byl parametr programu, stále můžeme mocniny ω spočítat v $O(n)$ krocích. Navíc, kdyby bylo $O(\log n)$ kroků použito na výpočet s a `rev(l)` z l na řádcích 5-8, těchto kalkulací by nebylo více než $3n$. Celková časová složitost tohoto algoritmu je tedy $O(n \log n)$.

6.4 FFT v modulární aritmetice

Při aplikacích FFT pro výpočet konvoluce je často potřeba přesný výsledek. Počítáme-li FFT v okruhu komplexnícxh čísel, musíme aproximovat reálná čísla s určitou konečnou přesností, čímž se do výpočtu mohou zavést chyby. Těmto chybám se vyhneme, počítáme-li konvoluci na konečném tělese.

Definice. Těleso je algebraický systém $(\mathbf{A}, +, \cdot, 0, 1)$ pro který platí:

1. systém je okruh s multiplikativní identitou 1,
2. násobení je komutativní,
3. každý element $a \in A - \{0\}$ má multiplikativní invers a^{-1}, takový že $aa^{-1} = a^{-1}a = 1$.

Cheme-li například zkonvolvovat posloupnosti $\mathbf{a} = [a_0, a_1, a_2, 0, 0]$ a $\mathbf{b} = [b_0, b_1, b_2, 0, 0]$ můžeme použít 2 jako pátý primitivní kořen 1 a provádět výpočty modulo 31. Transformací $\mathbf{a}$ a $\mathbf{b}$, provedením componentwise produktu a následnou inverzní transformací výsledku získáme vektor $\mathbf{c} = \mathbf{a} \otimes \mathbf{b}$, ve kterém je každý element modulo 31. Při volbě dostatečně velkého modula, respektive tělesa, můžeme říci že jsme spočítali konvoluci přesně.

Není vždy zřejmé zda pro dané n dokážeme najít vhodné ω a m, takové že ω je n-tý kořen jedné v okruhu $\mathbb{Z}_m$. Zároveň nechceme, aby m bylo zbytečně velké, neboť by operace modulo m byly příliš nákladné. Pokud ovšem budou naše n a ω pozitivními mocninami 2, můžeme počítat konvoluci pomocí FFT v okruhu celých čísel modulo $(\omega^{n/2} + 1)$, což se dozvíme ve větě 5. Nejprve ale musíme uvést pár pomocných lemmat.

V lemmatech 3 a 4 předpokládáme, že $R = (S, +, \cdot, 0, 1)$ je komutativní okruh a $n = 2^k, k \geq 1$.

Lemma 3. $\forall a \in S$,

$$\sum_{i=0}^{n-1} a^i = \prod_{i=0}^{k-1} (1 + a^{2^i})$$

Důkaz. Důkaz indukcí na k, případ $k = 1$ je triviální, nyní si všimněme že

$$\sum_{i=0}^{n-1} a^i = (1 + a) \sum_{i=0}^{n/2-1} (a^2)^i. \tag{6.5}$$

Z indukční hypotézy a substitucí a^2 za a dostaneme

$$\sum_{i=0}^{n/2-1} (a^2)^i = \prod_{i=0}^{k-2} \left[1 + (a^2)^{2^i}\right] = \prod_{i=1}^{k-1} \left[1 + a^{2^i}\right]. \tag{6.6}$$

Z indukčí hypotézy vyplývá, že můžeme substituovat 6.6 za pravou stranu 6.5. □

Lemma 4. *Nechť $m = \omega^{n/2} + 1$, kde $\omega \in S$, a $\omega \neq 0$. Potom pro $1 \leq p < n$ platí $\sum_{i=0}^{n-1} \omega^{ip} \equiv 0$ modulo m.*

Důkaz. Z lemmatu 3 stačí ukázat že $1 + \omega^{2^j p} \equiv 0$ modulo m pro nějaké $j, 0 \leq j < k$. Nechť $p = 2^s p'$, kde p' je liché a $0 \leq s < k$. Zvolme j takové, že $j + s = k - 1$. Potom $1 + \omega^{2^j p} = 1 + \omega^{2^{k-1} p'} = 1 + (m-1)^{p'}$. Ale $(m-1) \equiv -1$ modulo m, a p' je liché, takže $(m-1)^{p'} \equiv -1$ modulo m. Z toho plyne $1 + \omega^{2^j p} \equiv 0$ modulo m pro $j = k - 1 - s$. □

Věta 5. *Nechť n a ω jsou pozitivní mocniny 2 a nechť $m = \omega^{n/2} + 1$. Nechť $\mathbf{R}_m$ je okruh celých čísel modulo m. Potom v $\mathbf{R}_m$ má n multiplikativní invers modulo m a ω je primitivní n-tý kořen 1.*

Důkaz. Jelikož n je mocninou 2 a m je liché, jsou n a m nesoudělná, a tudíž n má multiplikativní invers modulo m.[6] Jelikož $\omega \neq 1, \omega^n = \omega^{n/2}\omega^{n/2} \equiv (-1)(-1) = 1$ modulo $(\omega^{n/2+1})$. Z lemmatu 4 potom vyplývá že ω je primitivní n-tý kořen 1 v $\mathbf{R}_m$. □

Lemma 5. *Nechť $m = \omega^p + 1$ a nechť $a = \sum_{i=0}^{l-1} a_i \omega^{pi}$, kde $\forall i (0 \leq a_i < \omega^p)$. Potom $a \equiv \sum_{i=0}^{l-1} a_i (-1)^i$ modulo m.*

Důkaz. $\omega^p \equiv -1$ modulo m.

□

[6] viz. Bezoutovy koeficienty v kapitole o Čínské zbytkové větě.

Příklad: 2. *Nechť $n = 4$, $\omega = 2$ a $m = 2^2 + 1$. Tudíž z lemmatu 5 dostaneme $p = 2$. Uvažujme $a = 101100$. Potom $a_0 = 00, a_1 = 11, a_2 = 10$. Spočítáme $a_0 - a_1 + a_2 = -1$, přičteme m a zjistíme, že $a \equiv 4$ modulo 5. Jelikož $a = 44$, výsledek odpovídá.*

Věta 6. *Nechť ω a n jsou pozitivní mocniny 2 a nechť $m = \omega^{n/2} + 1$. Nechť $a = [a_0, a_1, \ldots, a_{n-1}]$ je vektor jehož prvky jsou přirozená čísla, taková že $0 \leq a_i < m$ pro všechna i. Potom diskrétní Fourierova transformace vektoru a a její invers mohou být spočítány modulo m v čase $O_B(n^2 \log n \log \omega)$, kde $O_B(\cdot)$ označuje složitost úměrnou počtu bitových operací.*

Důkaz. Pro transformaci použijeme FFT algoritmus. Pro inverzní transformaci substituujeme ω^{-1} za ω, a výsledné hodnoty přenásobíme n^{-1}. Celé číslo modulo m můžeme reprezentovat posloupností o $b = ((n/2) \log \omega) + 1$ bitech. Jelikož $m = 2^{b-1} + 1$, můžeme zbytkové třídy modulo m reprezentovat bitovými řetězci od $00 \ldots 0$ do $100 \ldots 0$.

FFT algoritmus bude vyžadovat sčítání čísel modulo m a násobení mocninami ω modulo m. Tyto operace se provedou $O(n \log n)$-krát. S použitím lemmatu 5, je pro sčítání modulo m potřeba pouze $O_B(b)$ kroků, kde $b = ((n/2) \log n) + 1$. Násobení ω^p, kde $0 \leq p < n$, je ekvivalentní posunu doleva o $(p \log \omega)$ míst, neboť ω je mocnina dvojky. Výsledné číslo má nanejvýš $3b - 2$ bitů, a tedy z lematu 5, posun a počítání zbytků vyžaduje $O_B(b)$ kroků. Z toho dostáváme, že Fourierova transformace (v popředném směru), má časovou náročnost $O_B(bn \log n)$ t.j. $O_B(n^2 \log n \log \omega)$.

Inverzní transformace zahrnuje násobení ω^{-p}, a n^{-1}. Jelikož $\omega^p \omega^{n-p} \equiv 1$ modulo m, dostaneme $\omega^{n-p} \equiv \omega^{-p}$ modulo m. Tudíž výsledek po násobení ω^{-p} může být dosažen násobením ω^{n-p}, což je pouhý posun o $(n - p) \log \omega$ míst, a výsledné číslo má nejvýše $3b - 2$ bitů. Zbytkové třídy znovu počítáme podle lemmatu 5 v $O_B(b)$ krocích. Pro násobení n^{-1}, kde $n = 2^k$ se provede posun doleva o $n \log \omega - k$ míst, čímž získáme číslo mající (opět) nejvýše $3b - 2$ bitů, a spočteme zbytek pomocí 5. Tudíž inverzní transformace také vyžaduje pouze $O_B(n^2 \log n \log \omega)$ kroků. [1]

□

Důsledek věty 6

Nechť $O_B(M(k))$ je počet kroků potřebných pro výpočet produktu dvou k-bitových čísel. Nechť $\mathbf{a}$ a $\mathbf{b}$ jsou vektory délky n, obsahující přirozená čísla v rozsahu 0 až ω^n, kde n a ω jsou mocniny 2. Potom můžeme počítat $\mathbf{a} \otimes \mathbf{b}$, $\mathbf{a} \oplus \mathbf{b}$, nebo $\mathbf{a} \ominus \mathbf{b}$ modulo $\omega^n + 1$ v čase

$$O_B(\text{MAX}[n^2 \log n \log \omega, nM(n \log \omega)]).$$

Z čehož první term je počet kroků potřebných pro transformaci, druhý term je cena provedení $2n$ násobků $(n \log \omega + 1)$-bitových čísel.[1]

7 Čínská zbytková věta

Tato věta nám pomáhá řešit problém převodu celého čísla z jeho modulární reprezentace, do jeho poziční (tzv. radixové) reprezentace.

Věta 7. *Nechť $p_0, \ldots, p_{k-1}$, jsou nesoudělná přirozená čísla, a $P = \prod_{i=0}^{k-1} p_i$. Nechť $q_0, \ldots, q_{k-1}$ jsou přirozená čísla pro která platí $0 \leq q_i < p_i$, což můžeme zapsat jako soustavu kongruencí:*

$$\begin{aligned} x &\equiv q_0 \text{ modulo } p_0 \\ &\vdots \\ x &\equiv q_{k-1} \text{ modulo } p_{k-1} \end{aligned}$$

Potom existuje právě jedno takové x, pro které platí

$$0 \leq x < P \quad \& \quad x \equiv q_i \text{ modulo } (p_i) \text{ pro } 0 \leq i < k.$$

Unikátnost se dokazuje jednoduše: Předpokládejme, že x a y jsou dvě různá řešení pro všechny tyto kongruence. Potom $x \equiv y$ modulo (p_i) pro $1 \leq i < k$ a tedy $x - y$ je násobek každého p_i. Jelikož jsou p_i nesoudělná $x - y$ je dělitelné samotným P a tudíž rozdíl mezi x a y je minimálně P. Tím pádem se nemohou obě vejít do intervalu od 0 do $P - 1$. [1] [14]

Existenci takového x dokážeme pomocí jeho konstrukce v lemmatu 7.1. Před samotnou konstrukcí se nám bude hodit algoritmus, pro nalezení tzv. *Bézoutových koeficientů*. To jsou taková u a v pro která platí $ua + vb = g$, kde g je největší společný dělitel (gcd) čísel a a b. Pokud jsou a a b nesoudělná, platí $ua + vb = 1$ a tím pádem je $ua \equiv 1$ modulo b a $vb \equiv 1$ modulo a. To znamená že u je multiplikativní invers k a modulo b, a v je multiplikativní invers k b modulo a. Takováto u, v můžeme spočítat například algoritmem *Extended Gcd* z knihy [2] na obr. 7.1.

Extended Gcd.

`input:` celá čísla a a b
`output:` celá čísla (g, u, v) taková že $g = gcd(a, b) = ua + vb$

$(u, v) = (1, 0)$
$(v, x) = (0, 1)$
`while` $b \neq 0$ `do`
    $(q, r) = \text{DivRem}(a, b)$ `comment:` libovolný algoritmus který provede dělení a/b se zbytkem, tj najde takové q, r že $a = qb + r$.
    $(a, b) = (b, r)$
    $(u, w) = (w, u - qw)$
    $(v, x) = (x, v - qx)$
`return` (a, u, v)

Obrázek 7.1: Extended gcd algoritmus z [2]

Lemma 6. *Nechť $p_0, \ldots, p_{k-1}$, jsou nesoudělná přirozená čísla, $q_0, \ldots, q_{k-1}$ jsou přirozená čísla, pro která platí $0 \leq q_i < p_i$. Nechť c_i je produkt všech p_j kromě p_i (t.j. $c_i = P/p_i$, kde $P = \prod_{j=0}^{k-1}$) Nechť d_i je c_i^{-1} modulo p_i (takže $d_i c_i \equiv 1$ modulo p_i, a $0 \leq d_i < p_i$). Potom*

$$x \equiv \sum_{i-0}^{k-1} c_i d_i q_i \text{ modulo } P. \tag{7.1}$$

Důkaz. Jelikož p_j jsou vzájemně nesoudělná, víme že d_i existuje, a je unikátní. Zároveň c_i je dělitelné všemi p_j pro $j \neq i$, takže $c_i d_i q_i \equiv 0$ modulo p_j pro $j \neq i$. Tudíž

$$\sum_{i=0}^{k-1} c_i d_i q_i \equiv c_j d_j q_j \text{ modulo } p_j.$$

Jelikož $c_j d_j \equiv 1$ modulo p_j, dostaneme

$$\sum_{i=0}^{k-1} c_i d_i q_i \equiv q_j \text{ modulo } p_j$$

Tento vztah platí i v případě výpočtu modulo p, neboť p_j dělí P, a tudíž platí 7.1. [1] □

8 Schönhage-Strassenův algoritmus

Nechť u a v jsou binárně zapsaná přirozená čísla mezi 0 a 2^n, která chceme násobit modulo $(2^n + 1)$. Všimněme si, že pro reprezentaci 2^n je potřeba

$n+1$ bitů. Pokud se u nebo v rovná 2^n budou reprezentováni jako speciální symbol, -1 a násobení je řešeno jako specilní případ. Když $u = 2^n$, tak uv modulo $(2^n + 1)$ získáme výpočtem $(2^n + 1 - v)$ modulo $(2^n + 1)$.

Předpokládejme že $n = 2^k$, nechť $b = 2^{k/2}$ je-li k sudé, a nechť $b = 2^{(k-1)/2}$ je-li k liché. Nechť $l = n/b$. Všimněme si, že $l \geq b$ a že b dělí l. První krok je rozdělit u a v na b bloků, každý po l bitech. Vytvoříme tak b-rozměrné vektory, jejichž prvky jsou $\leq 2^l - 1$.

Platí tedy

$$u = u_{b-1}2^{(b-1)l} + \cdots + u_1 2^l + u_0 \quad = \quad \sum_{i=0}^{b-1} u_i 2^{il},$$

a

$$v = v_{b-1}2^{(b-1)l} + \cdots + v_1 2^l + v_0 \quad = \quad \sum_{i=0}^{b-1} v_i 2^{il}.$$

Produkt u a v je

$$uv = y_{2b-2}2^{(2b-2)l} + \cdots + y_1 2^l + y_0 \quad = \quad \sum_{i=0}^{2b-1} y_i 2^{il}, \tag{8.1}$$

kde

$$y_i = \sum_{j=0}^{b-1} u_j v_{i-j}, \qquad 0 \leq i < 2b. \tag{8.2}$$

Pro ($j < 0$ nebo $j > b - 1$ vezmene $u_j = v_j = 0$. Term y_{2b-1} je 0 a je uveden pouze pro symetrii.)

Pro výpočet produktu uv můžeme uplatnit větu o konvoluci. *Componentwise* produkt Fourierových transformací bude vyžadovat $2b$ náobení, v případě lineární konvoluce, ale pouze b násobení použijeme-li cyklickou, či negacyklickou konvoluci.

Rozdělme si y_i v 8.2 na dva případy v závislosti na velikosti i:

$$y_i = \begin{cases} \sum_{i=0}^{j} u_i v_{j-i} & \text{když } j < b \\ \\ \sum_{i=j-(b-1)}^{b-1} u_i v_{j-i} & \text{když } j \geq b \end{cases} \tag{8.3}$$

Všimněme si, že $2^{bl} = 2^n$ a $2^n \equiv -1$ modulo $(2^n + 1)$. Budeme-li počítat uv modulo $(2^n + 1)$, dostaneme ve druhém případě 8.3 kde $j \geq b$ záporné y_i, neboť bude přenásovebené $(-1)2^{(j-b)l}$ (podle 8.1) a můžeme produkt uv modulo $(2^n + 1)$ přepsat na

$$uv = \sum_{i=0}^{b-1} (y_i - y_{b+i}) 2^{il} \quad \text{modulo } (2^n + 1). \tag{8.4}$$

Nechť $(y_i - y_{b+i}) = w_i$. Z pozorování 4 dostaneme, že vektor $[w_0, \ldots, w_{b-1}]$ je negacyklickou konvolucí koeficientové reprezentace u a v.

$$uv \equiv (w_{b-1}2^{(b-1)l}, + \cdots + w_1 2^l + w_0) \text{ modulo } (2^n + 1)$$

Jelikož produkt dvou l-bitových čísel musí být menší než 2^{2l} a jelikož y_i a y_{b+i} jsou součty $i+1$ a $b-(i+1)$ takových produktů, w_i musí být v rozsahu $-(b-1-i)2^{2l} < w_i < (i+1)2^{2l}$. Máme tedy nejvýše $b2^{2l}$ hodnot, které může w_i nabývat. Pro výpočet uv tedy musíme zvolit větší modulo, než je $b2^{2l}$.[1]

V tomto algoritmu se použije o trošku větší modulo a to $b(2^{2l} + 1)$. Abychom získali w_i modulo $b(2^{2l} + 1)$, spočítámeme nejprve $w'_i = w_i$ modulo b a $w"_i = w_i$ modulo $(2^{2l} + 1)$, a tyto výsledky spojíme pomocí Čínské zbytkové věty.

Z lemmatu 7.1 dostáváme:

$$p_1 = b, \qquad p_2 = (2^{2l} - 1), \qquad u_1 = w'_i, \qquad u_2 = w"_i,$$

$$c_1 = \frac{p_1 p_2}{p_1} = p_2 = (2^{2l} + 1), \quad c_2 = \frac{p_1 p_2}{p_2} = p_1 = b,$$

$$d_1 = (c_1^{-1} \bmod p_1) = ((2^{2l} + 1) \bmod b) = 1, \text{ [7]}$$

$$d_2 = (c_2^{-1} \bmod p_2) = (b^{-1} \bmod (2^{2l} + 1)) = ((2^{2l} + 1) - \frac{2^{2l}}{b})\text{[8]}.$$

Připomeňme že

$$w_i \equiv \sum_{j=1}^{2} c_j d_j u_j \text{ modulo } (P),$$

[7] Jelikož b je mocninou 2 a $b \leq (2^{2l})$, tak b dělí (2^{2l}) a tedy multiplikativní invers $(2^{2l} + 1)$ modulo b je 1.

[8] $b((2^{2l} + 1 - \frac{2^{2l}}{b})) = b(2^{2l} - 1) - 2^{2l} \equiv 1$ modulo $(2^{2l} + 1)$

tudíž

$$w_i \equiv (2^{2l}+1) \times 1 \times w'_i + b \times ((2^{2l}+1) - \frac{2^{2l}}{b}) \times w"_i \text{ modulo } b(2^{2l}+1),$$

z čehož dostaneme

$$w_i \equiv w'_i(2^{2l}+1) + w"_i b(2^{2l}+1) - w"_i(2^{2l}) \text{ modulo } b(2^{2l}+1).$$

Prostřední term $w"_i b(2^{2l}+1)$ bude $\equiv 0$ modulo $b(2^{2l}+1)$, tudíž můžeme napsat

$$w_i \equiv w'_i(2^{2l}+1) - (w"_i)2^{2l} \text{ modulo } b(2^{2l}+1),$$

z čehož dostaneme

$$w_i \equiv (2^{2l}+1)(w'_i - w"_i) + w"_i \text{ modulo } b(2^{2l}+1).$$

[17]

V knize [1] je použita trochu odlišná formule

$$w_i \equiv (2^{2l}+1)(w'_i - w"_i \text{ modulo } b) + w"_i,$$

kde w_i je mezi $-(b-1-i)2^{2l}$ a $(i+1)2^{2l}$. Práce potřebná pro scpočítání w_i z w'_i a $w"_i$ je $O(l+\log b)$ pro každé w_i, celkem tedy $O(bl+b\log b)$, tedy $O(n)$ [1].

Pro získání w'_i nejprve vezmeme $u'_i = u_i$ modulo b a $v'_i = v_i$ modulo b, čímž získáme $(\log b)$-bitová čísla, mezi která vložíme $(2\log b)$ nul, viz. obr 8.1.

$$\hat{u} = u'_{b-1}\ 00\ldots0\ u'_{b-2}\ 00\ldots0\ u'_{b-3}\ \ldots00\ldots0\ u'_0$$

$$\hat{v} = \underbrace{v'_{b-1}}_{\log b}\ \underbrace{00\ldots0}_{2\log b}\ v'_{b-2}\ 00\ldots0\ v'_{b-3}\ \ldots00\ldots0\ v'_0$$

Obrázek 8.1

Takto vytvoříme $(3b\log b)$-bitová čísla $\hat{u}$ a $\hat{v}$, která spolu znásobíme pomocí Karatsubova algoritmu. Toto násobení bude tedy vyžadovat nanejvýš $O((3b\log b)^{1.6})$, to je méně než $O(n)$ kroků. Je zřejmé, že

$$\hat{u}\hat{v} = \sum_{i=0}^{2b-1} y'_i 2^{(3\log b)i}, \text{ kde } y'_i = \sum_{j=0}^{2b-1} u'_j v'_{i-j}.$$

Jelikož $y'_i < 2^{3 \log b}$, můžeme y'_i z produktu $\hat{u}\hat{v}$ snadno extrahovat. Hodnoty w_i modulo b následně spočítáme pomocí $(y'_i - y'_{b+i})$ modulo b.

K získání w_i modulo $(2^{2l} + 1)$ použijeme negacyklickou konvoluci. Nechť $\omega = 2^{4l/b}$ a $m = 2^{2l} + 1$ z věty 5 víme, že ω a b mají multiplikativní inverzy modulo m, a že ω je primitivní b-tý kořen 1. Negacyklickou konvoluci $[u_0, \ldots, u_{b-1}] \ominus [v_0, \ldots, v_{b-1}]$ získáme pomocí $[u_0, \psi u_1, \ldots, \psi^{b-1} u_{b-1}]$ a $[v_0, \psi v_1, \ldots, \psi^{b-1}]$, kde $\psi = 2^{2l/b}$ a ψ je $2b$-tý kořen 1). Z věty 2 tak dostaneme

$$[(y_0 - y_b), \psi(y_1 - y_{b+1}), \ldots, \psi^{b-1}(y_{b-1} - y_{2b-1})] \text{ modulo } (2^{2l} + 1),$$

kde $y_i = \sum_{j=0}^{b-1} u_j v_{i-j}$ pro $0 \leq i \leq 2b - 1$. [1]

8.1 Algoritmus - přehled

`input`: Dvě n-bitová čísla u, v, kde $n = 2^k$.
`output`: $(n + 1)$-bitový produkt uv modulo $2^n + 1$.

Pokud je n malé, znásobíme u a v modulo $2^n + 1$ libovolným algoritmem. Pro velká n, nechť $b = 2^{k/2}$ je-li k sudé, jinak $b = 2^{(k-1)/2}$ a nechť $l = n/b$. Vyjádříme $u = \sum_{i=0}^{b-1} u_i 2^{li}$ a $v = \sum_{i=0}^{b-1} v_i 2^{li}$, kde u_i a v_i jsou přirozená čísla mezi 0 a $2^{2l} - 1$.

1. Spočítáme $w"_i = w_i$ mod $(2^{2l} + 1)$ následujícím způsobem:

   (a) Spočítáme Fourierovi transformace vektorů

   $$[u_0, \psi u_1, \ldots, \psi^{b-1} u_{b-1}] \quad \text{a} \quad [v_0, \psi v_1, \ldots, \psi^{b-1} v_{b-1}],$$

   kde $\psi = 2^{2l/b}$, a $\omega = \psi^2$ použijeme jako b-tý primitivní kořen 1.

   (b) Spočítáme *pairwiese* produkt těchto transformací rekurzivně, pomocí tohoto algoritmu. Situace, ve které se jedno z čísel rovná 2^{2l} je řešena jako speciální případ.

   (c) Nyní spočítáme inverzní Fourierovu transformaci získaného vektoru modulo $2^{2l} + 1$. Výsledkem této transformace bude vektor $[w_0, \psi w_1, \ldots, \psi^{b-1} w_{b-1}]$ modulo $2^{2l} + 1$, kde w_i je i-tý term negacyklické konvoluce $[u_0, u_1, \ldots, u_{b-1}]$ a $[v_0, v_1, \ldots, v_{b-1}]$.

   (d) Spočítáme $w"_i = w_i$ modulo $2^{2l}+1$ tak, že každé $\psi^i w_i$ přenásobíme ψ^{-i} modulo $2^{2l} + 1$.

2. Spočítáme $w'_i = w_i$ mod b následujícím způsobem:

(a) Nechť $u'_i = u_i$ modulo b a $v'_i = v_i$ modulo b pro $0 \leq i < b$.

(b) Zkonstruujeme čísla $\hat{u}$ a $\hat{v}$ vložením $2 \log b$ nul mezi každé dva prvky vzniklých vektorů, viz obr 8.1. Takže $\hat{u} = \sum_{i=0}^{b-1} u'_i 2^{(3 \log b)i}$ a $\hat{v} = \sum_{i=0}^{b-1} v'_i 2^{(3 \log b)i}$.

(c) Spočítáme produkt $\hat{u}\hat{v}$ pomocí karatsubova algoritmu.

(d) Produkt $\hat{u}\hat{v}$ je $\sum_{i=0}^{2b-1} y_i 2^{(3 \log b)i}$, kde $y_i = \sum_{j=0}^{2b-1} u'_j v'_{i-j}$. Z tohoto produku získáme w_i modulo b vyhodnocením $w'_i = (y'_i - y'_{b+i})$ modulo b pro $0 \leq i < b$.

3. Spočítáme hodnotoy w_i pomocí formule

$$w_i = (2^{2l} + 1)((w'_i - w"_i) \text{ modulo } b) + w"_i,$$

a w'_i je mezi $(b-1-i)2^{2l}$ a $(i+1)2^{2l}$. Jestli bude $w_i \geq (i+1)2^{2l}$ odečteme od něj $b(2^{2l} + 1)$.

4. Spočítáme $\sum_{j=0}^{b-1} w_i 2^{li}$ modulo $(2^n + 1)$ což je požadovaný výsledek.

[1]

8.2 Složitost

Věta 8. *Složitost Schönhage-Strassenova algoritmu je* $O(n \log n \log \log n)$.

Důkaz. Z důsledku k větě 6 se výpočty v prvním odstavci provedou v čase

$$O_B[bl \log b + bM(2l])],$$

kde $M(m)$ je čas potřebný pro násobení dvou m-bitových čísel rekurzivní aplikací tohoto algoritmu. Ve druhém odstavci konstruujeme čísla $\hat{u}$ a $\hat{v}$ délky $(3b \log b)$ a násobíme je v $O_B[(3b \log b)^{1.59}]$ krocích. Pro dostatečně velké b bude $(3b \log b)^{1.59} < b^2$ a tudíž může být tento čas ignorován z hlediska termu $O_B(b^2 \log b)$ v prvním odstavci. Třetí a čtvrtý odstavec jsou oba počítány v čase $O(n)$ a také mohou být ignorovány.

Jelikož $n = bl$ a b je nejvýše $\sqrt{n}$, obdržíme rekurzivní relaci

$$M(n) \leq cn \log n + bM(2l) \tag{8.5}$$

pro nějakou konstantu c a dostatečně velké n. Nechť

$$M'(n) = M(n)/n.$$

Potom se 8.5 přepíše na

$$M'(n) \leq c \log n + 2M'(2l). \tag{8.6}$$

Jelikož $l \leq 2\sqrt{n}$, dostáváme

$$M'(n) \leq c \log n + 2M'(4\sqrt{n}), \tag{8.7}$$

z čehož plyne $M'(n) \leq c' \log n \log\log n$ pro vhodné c'. Substituujme $(c' \log(4\sqrt{n}) \log\log 4\sqrt{n})$ za $M'(4\sqrt{n})$ v 8.7. Přímou manipulací dostaneme

$$M'(n) \leq c \log n + 4c' \log(2 + \frac{1}{2} \log n) + c' \log n \log(2 + \frac{1}{2} \log n).$$

Pro velké n, $2 + \frac{1}{2} \log n \leq \frac{2}{3} \log n$. Tudíž

$$M'(n) \leq c \log n + 4c' \log \frac{2}{3} + 4c' \log\log n +$$

$$c' \log \frac{2}{3} \log n + c' \log n + c' \log n \log\log n. \tag{8.8}$$

Pro velké n a dostatečně velké c' bude prvním čtyřem termům dominovat pátý, který je negativní. Tudíž $M'(n) \leq c' \log n \log\log n$. Z čehož můžeme vyvodit $M(n) \leq c' n \log n \log\log n$. [1]

□

9 Závěr

Schönhage-Strassenův algoritmus je díky jeho vysoce optimalizované implementaci, zahrnuté ve volně dostupné GNU Multiple Precision Arithmetic Library (GMP) [8], v současné době pravděpodobně nejrozšířenějším algoritmem pro násobení velkých čísel. GPM stojí za schopnostmi většiny hlavních současných výpočetních systémů, které zpracovávají velkočíselné operace [10].

Schönhage se Strassenem vedle dvou algoritmů v článku [15] zároveň uvedli domněnku, že nejnižší dosažitelná hranice pro složitost násobících algoritmů je $O(n \log n)$. Této hranice dosahli teprve roku 2021 australský matematik David Harvey s holandským matematikem Jorisem Van Der Hoevenem [10]. Tímto objevem překonali svůj dřívější rekord z roku 2016 [11], kdy spolu s Gregoriem Lecerfem přebrali prvenství Martinu Fürerovi.

Jak uvádějí v abstraktu, ve svém nejnovějším algoritmu použili "Gausovskou resamplovací techniku", díky které zredukovali problém násobení

celých čísel na kolekci multidimenzionálních Fouriérových transformací (v $\mathbb{C}$). Tyto transformace následně vyhodnocují pomocí Nussbaumerovy rychlé polynomiální transformace [10].

Není to tak dávno, co jsme se domnívali, že $O(n^2)$ je optimální stav. Od té doby nastal z historické perspektivy překotný vývoj, během kterého jsme se dostali z polynomiální do linearitmické složitostní třídy. Z uvedených algoritmů můžeme usoudit, že každé drobné vylepšení složitosti časové si vyžádalo netriviální navýšení složitosti teoretické.

Reference

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company 1974.

[2] R. Brent, P. Zimmermann. *Modern Computer Arithmetic*, Cambridge Monographs on Applied and Computational Mathematics, vol.18, Cambridge University Press, Cambridge, 2011. MR 2760886

[3] J. W. Cooley, J. W. Tukey. *An Algorithm for the Machine Calculation of Complex Fourier Series*. Mathematics of Computation. 19 (90):297301. 1965

[4] T. H. Cormen, C. Lee, E. Lin. *Introduction to algorithms, Instructor's Manual*. The MIT Press, McGraw-Hill, 2002.

[5] T. H. Cormen, C. E. Leiserson, R.L. Rivest, C. Stein. *Introduction to Algorithms*, Fourth edition. The MIT Press, 2022.

[6] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms*. McGraw-Hill Education, 2006.

[7] M. Fürer. Faster integer multiplication. In *Proceedings of the Thirty-Ninth ACM Symposium on Theory of Computing, STOC 2007*, pages 57-66, New York, NY, USA, 2007. ACM Press.

[8] P Gaudry, A Kruppa, P. Zimmermann, *A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm*, ISSAC 2007, ACM, New York 2007, pp. 167-174. MR.2396199

[9] D. Harvey. A casche-friendly truncated FFT. *Theoretical computer science*, 410(27-29):2649-2658, 2009. ISSN 0304 - 3975.

[10] D. Harvey, J. Van Der Hoeven. Integer multiplication in time $O(n \log n)$, *Annals of Mathematics* 193 (2), 563-617, 2021.

[11] D. Harvey, J. Van Der Hoeven, G. Lecerf. Even faster multiplication. *Journal of Complexity*, 36: 1-30, 2016.

[12] R. Karamagi. *Design and Analysis of Algorithms*. Lulu Press, Inc, 2020.

[13] A. A. Karatsuba *The Complexity of Computations*
Proceedings of the Steklov Institute of Mathematics. 211: 169–183.
Translation from Trudy Mat. Inst. Steklova, (1995), 211, 186–202.

[14] D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, Addison Wesley, 1998.

[15] A. Schönhage, V. Strassen. *Schnelle Multiplikation großer Zahlen.* Computing, 7:281-292, 1971.

[16] R. M. Vermani, S. Vermani. *An Elementary Approach to Design and Analysis of Algorithms.* World Scientific, 2019.

[17] Přednáška: Computer Algorithms 2. Instructor: Prof. Dr. Shashank K. Mehta, Department of Computer Science and Engineering, IIT Kanpur, Lecture 26 - Schonhage-Strassen Algorithm. https://freevideolectures.com/course/3060/computer-algorithms/26