Binarni polinomi

Šiljeg, Jure

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Jure Šiljeg

BINARNI POLINOMI

Diplomski rad

Voditelj rada:
doc. dr. sc. Goranka Nogo

Zagreb, 2017.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Posvećujem ovaj rad svim ljudima koji su pronašli životnu sreću, pa onda sreću i u ovoj struci. I svom bratu.

Sadržaj

Uvod

U radu se promatraju algoritmi nad binarnim polinomima. Korisno je imati pouzdane i brze algoritme za klasične operacije iz razloga što su polinomi ponajprije osnovni matematički alat. Podklasu binarnih polinoma pronalazimo u primjenama kao što su standard AES u kriptografiji ili u teoriji kodiranja (ciklički kôdovi npr.).
Rad je podijeljen u 8 poglavlja. U prvom poglavlju dajemo kratak uvod u binarne polinome i instuitivno dolazimo do njegove definicije. Upravo definicija će motivirati operacije nad takvim polinomima i u nekoliko narednih poglavlja se bavimo tim operacijama.
U drugom poglavlju se osvrćemo na zbrajanje binarnih polinoma. Dajemo ideju i analiziramo složenost.
Treće poglavlje započinjemo naivnim algoritmom za množenje polinoma. Potom dajemo pseudokôd i analiziramo složenost. Slijede profinjeniji algoritmi od kojih ćemo predstaviti Karatsubin algoritam za množenje polinoma i hibridni Karatsubin algoritam. Karatsubin algoritam će prikazati množenje polinoma pomoću samo tri množenja i rekurzivnom metodom podijeli pa vladaj postići bolju složenost od naivnog algoritma. Hibridni pristup koristi podrezivanje na nekom nivou zbog čega ne izvršavamo rekurziju do najnižeg nivoa, već koristimo prednosti naivnog množenja koje je brže za polinome manjeg stupnja.
Četvrto poglavlje obrađuje potenciranje polinoma. Tu ćemo se susresti s tri metode: Potenciranje s desna na lijevo, K-arno potenciranje s lijeva na desno i Klizeći prozori.
U petom poglavlju vidjet ćemo kako se dijele polinomi s ostatkom.
Šesto poglavlje donosi algoritam za faktorizaciju binarnih polinoma. Pojasnit ćemo što je to Q matrica i kako nam pomaže pri traženju faktorizacije. Najprije ćemo se posvetiti faktorizaciji kvadratno slobodnih polinoma i potom naći način za ekstrahiranje tog dijela iz općenitog polinoma i izračunati kratnosti čime će se naš proces traženja faktorizacije i završiti.
U sedmom poglavlju obavit ćemo testiranja i usporedbu algoritama unutar pojedine operacije i uvjeriti se da rade u skladu s teoretskim predviđanjima.
Dat ćemo i kratak pregled primjene u osmom poglavlju.

Poglavlje 1

Uvod u binarne polinome

U matematici se nerijetko zahtijeva modeliranje nekog događaja i uspostavljanja matematičkog odnosa među varijablama. Često matematičare (i druge koji matematiku primjenjuju svakodnevno) zanima što se događa s jednom varijablom ako se promijeni druga i kako uspostaviti preciznu relaciju koja uopće opisuje nekakvu pojavu. Možemo se, tako, pitati kakva će biti cijena pojedine dionice za dva dana na Wall Streetu ili kako uspješno dizajnirati tobogan smrti. Zanima li nas može li zrakoplov visine $2m$ i raspona krila $11m$ ući u hangar čiji oblik nalikuje "naopakom slovu U" kojem je maksimalna visina $20m$ i maksimalna širina $15m$, također nam može pomoći matematika. U konkretnim slučajevima - polinomi.

1.1 Definicija binarnog polinoma

Prisjetimo se za početak definicija i rezultata koji će nam poslužiti kao motivacija za definiranje binarnih polinoma.

Definicija 1.1.1. *Polinom (jedne varijable) je funkcija realnog ili kompleksnog argumenta x koju možemo zapisati u obliku $P(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$, gdje je $n \in \mathbb{N}_0$, $a_n \neq 0$ ako je $n \geq 1$. Brojeve $a_0, a_1, a_2, ..., a_n$ zovemo koeficijenti polinoma P, broj a_0 slobodni član, a koeficijent a_n vodeći član. Broj n zovemo stupanj polinoma ($deg(P) = n$).*

Definicija 1.1.2. *Neprazan skup $R = (R, +, \cdot)$ zovemo **prsten** ukoliko je za operacije zbrajanja $+ : R \times R \to R$ i množenja $\cdot : R \times R \to R$ ispunjeno sljedeće:*

(R1) $(R, +)$ *je komutativna grupa, s neutralom* $0 = 0_R$

(R2) $(R, \cdot)$ *je polugrupa* [1]

[1] Polugrupa je struktura za koju vrijedi zatvorenost i asocijativnost.

(R3) Vrijedi i lijeva i desna distributivnost "množenja prema zbrajanju", tj.

$$(\forall a, b, c \in R): \qquad a \cdot (b + c) = (a \cdot b) + (a \cdot c), \qquad (a + b) \cdot c = (a \cdot c) + (b \cdot c).$$

Definicija 1.1.3. *Neprazan skup $\mathbb{F} = (\mathbb{F}, +, \cdot)$ zovemo **polje** ukoliko je za operacije zbrajanja $+ : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ i množenja $\cdot : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ ispunjeno sljedeće:*

(F1) Polje $\mathbb{F}$ je zatvoreno na zbrajanje i množenje:

$(\forall a, b \in \mathbb{F})$*:* $\qquad a + b \in \mathbb{F}, a \cdot b \in \mathbb{F}$

(F2) Vrijedi asocijativnost zbrajanja i množenja:

$(\forall a, b, c \in \mathbb{F})$*:* $\qquad (a+b)+c = a+(b+c), \quad (a \cdot b) \cdot c = a \cdot (b \cdot c)$

(F3) Vrijedi komutativnost zbrajanja i množenja:

$(\forall a, b \in \mathbb{F})$*:* $\qquad a + b = b + a, a \cdot b = b \cdot a$

(F4) Vrijedi distributivnost "množenja prema zbrajanju":

$(\forall a, b, c \in \mathbb{F})$*:* $\qquad a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

(F5) Postojanje neutralnog elementa za zbrajanje:

$(\exists 0 \in \mathbb{F})(\forall a \in \mathbb{F})$*:* $\qquad a + 0 = 0 + a = a$

(F6) Postojanje neutralnog elementa za množenje:

$(\exists 1 \in \mathbb{F})(\forall a \in \mathbb{F})$*:* $\qquad a \cdot 1 = 1 \cdot a = a$

(F7) Postojanje inverza za zbrajanje:

$(\forall a \in \mathbb{F})(\exists(-a) \in \mathbb{F})$*:* $\qquad a + (-a) = -a + a = 0$

(F8) Postojanje inverza za množenje:

$(\forall a \in \mathbb{F})(a \neq 0)(\exists a^{-1} \in \mathbb{F})$*:* $\qquad a \cdot a^{-1} = a^{-1} \cdot a = 1.$

Ukratko, kažemo da je algebarska struktura $(A, +, \cdot)$ polje ukoliko vrijedi:

(F1') $(A, +, \cdot)$ *je prsten*

(F2') $(A\backslash\{0\}, \cdot)$ *je komutativna grupa.*

Zanimat će nas restrikcija na konačna polja[2] i njih ćemo označavati s $GF(q)$, gdje q predstavlja broj elemenata tog polja. Promotrimo sada skup ostataka pri dijeljenju s n u sljedećoj oznaci:

$$\mathbb{Z}_n := \mathbb{Z}/n\mathbb{Z} = \{[0], [1], ..., [n-1]\},\ gdje\ je [k] = \{x \in \mathbb{Z} : x \equiv k (mod\ n)\}.$$

Nad takvim skupom ćemo definirati binarne operacije zbrajanja i množenja na sljedeći način:

(1) Zbrajanje: Za $[a], [b] \in \mathbb{Z}_n$ definiramo operaciju $+_n : \mathbb{Z}_n \times \mathbb{Z}_n \to \mathbb{Z}_n$ na sljedeći način:

$$[a] +_n [b] := [a + b]. \tag{1.1}$$

(2) Množenje: Za $[a], [b] \in \mathbb{Z}_n$ definiramo operaciju $\cdot_n : \mathbb{Z}_n \times \mathbb{Z}_n \to \mathbb{Z}_n$ na sljedeći način:

$$[a] \cdot_n [b] := [a \cdot b]. \tag{1.2}$$

Nužno bi bilo provjeriti jesu li operacije (1) i (2) dobro definirane. Provjerit ćemo za zbrajanje, a množenje se napravi analogno.

$$\overbrace{[a] = [c] \wedge [b] = [d]}^{(\triangle)} \overset{?}{\Rightarrow} [a + b] = [c + d]$$
$$[a + b] \overset{1.1}{=} [a] +_n [b] \overset{(\triangle)}{=} [c] +_n [d] \overset{1.1}{=} [c + d].$$

Propozicija 1.1.4. $\mathbb{Z}_n = (\{[0], [1], ..., [n-1]\}, +_n, \cdot_n)$ *je prsten.*

Dokaz. Tvrdnja se sastoji od toga da pokažemo da vrijede svojstva (R1), (R2) i (R3).

(R1): $(\mathbb{Z}_n, +_n)$ je Abelova grupa.

- *Zatvorenost:* Neka su $[a], [b] \in \mathbb{Z}_n$. Zbog definicije (1), možemo primijetiti da vrijedi $[a] +_n [b] \in \mathbb{Z}_n$.
- *Asocijativnost:* Neka su $[a], [b], [c] \in \mathbb{Z}_n$. Vrijedi sljedeće:

$$\begin{aligned} ([a] +_n [b]) +_n [c] &\overset{1.1}{=} [a + b] +_n [c] \\ &\overset{1.1}{=} [a + b + c] \\ &\overset{1.1}{=} [a] +_n [b + c] \\ &\overset{1.1}{=} [a] +_n ([b] +_n [c]). \end{aligned}$$

[2] Još ih zovemo Galoiseva polja u čast francuskom matematičaru Évariste Galoisu (1811.-1832.).

- *Neutralni element*: Za zbrajanje je prirodno pretpostaviti da će [0] biti neutralni element. Za proizvoljni $[n] \in \mathbb{Z}_n$ uvrštavanjem lako dobijemo:

$$[n] +_n [0] \overset{1.1}{=} [n + 0] = [0] = [0 + n] \overset{1.1}{=} [0] +_n [n].$$

- *Inverzni element*: Neka je $[m] \in \mathbb{Z}_n$. Definiramo da je inverz za $[m]$ upravo $[n - m]$. Provjerom lako dobijemo da vrijedi:

$$[m] +_n [n - m] \overset{1.1}{=} [m + n - m] = [0] = [n - m + m] \overset{1.1}{=} [n - m] +_n [m].$$

- *Komutativnost*: Za proizvoljne $[a], [b] \in \mathbb{Z}_n$ je očito da tvrdnja vrijedi budući da vrijedi komutativnost zbrajanja nad prirodnim brojevima.

(R2): $(\mathbb{Z}_n, \cdot_n)$ je polugrupa.

- *Zatvorenost*: Neka su $[a], [b] \in \mathbb{Z}_n$. Zbog definicije (2), možemo primijetiti da vrijedi $[a] \cdot_n [b] \in \mathbb{Z}_n$.
- *Asocijativnost*: Tvrdnja se dokazuje analogno asocijativnosti zbrajanja iz dijela (R1).

(R3):
- *Distributivnost "množenja prema zbrajanju"*: Pokazat ćemo samo lijevu distributivnost budući da se drugi smjer dobije analogno. Neka su $[a], [b], [c] \in \mathbb{Z}_n$. Tada vrijedi:

$$\begin{aligned}
[a] \cdot_n ([b] +_n [c]) &\overset{1.1}{=} [a] \cdot_n [b + c] \\
&\overset{1.2}{=} [a \cdot (b + c)] \\
&= [a \cdot b + a \cdot c] \\
&\overset{1.1}{=} [a \cdot b] +_n [a \cdot c] \\
&\overset{1.2}{=} [a] \cdot_n [b] +_n [a] \cdot_n [c].
\end{aligned}$$

□

Nama će od posebnog interesa biti prsten $\mathbb{Z}_2$ koji igra važnu ulogu u definiciji binarnog polinoma. $\mathbb{Z}_2$ je, štoviše, polje. Tvrdnja jednostavno slijedi iz (F1') i (F2'). Budući da (F1') slijedi iz 1.1.4, ostaje samo dokazati još (F2'). No, to je trivijalno jer se radi o skupu koji ima samo jedan element.

Uzmemo li u obzir prethodno definirano i dokazano, konačno smo spremni dati i definiciju binarnog polinoma.

Definicija 1.1.5. *Neka je $\mathbb{Z}_n$ polje ostataka modulo n. Polinom s koeficijentima u polju $GF(2) = \mathbb{Z}_2 = \{0, 1\}$ se zove* ***binarni polinom****.*

Ukoliko pažljivo promotrimo definiciju, postaje jasna intuicija po kojoj smo birali da su koeficijenti polinoma zapravo elementi polja. Sada nad takvim elementima smijemo raditi operacije koje su dozvoljene i u samom polju što nas, na koncu, motivira na zbrajanje, množenje, dijeljenje, potenciranje (binarnih) polinoma.

Konačna polja reda 2^m se zovu *binarna polja*. Pokazat će se da su ona interesantna u primjenama jer se efikasno mogu implementirati na hardveru koji koristi binarni sustav znamenki za rad.

Elementi polja $GF(2^m)$ su, ustvari, polinomi oblika $a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1x + a_0$, $a_i \in \{0, 1\}$ za $i = 0, 1, ..., m - 1$. Svaki polinom ćemo prikazivati m-bitnim nizom znakova (string ili lista) tako da svaki bit na k-toj poziciji u stringu odgovara koeficijentu a_k polinoma za $k \in \{0, 1, ..., m - 1\}$. Pritom, jasno, string numeriramo da najmanje značajan bit (najdesniji u zapisu) odgovara slobodnom koeficijentu. Uzmimo za primjer $GF(2^3)$ koji sadrži osam elemenata: $\{0, 1, x, x + 1, x^2, x^2 + 1, x^2 + x, x^2 + x + 1\}$ gdje je $x + 1$, zapravo, $0x^2 + 1x + 1$. Možemo ga reprezentirati kao bitovni string 011. Slično, $x^2 + x = 1x^2 + 1x + 0$, pa ga reprezentiramo s 110.

U nastavku rada bavit ćemo se vremenskom složenosti pojedinih algoritama nad binarnim polinomima. Uvedimo, stoga, oznaku koju ćemo koristiti pri analizi takvog tipa složenosti.

Definicija 1.1.6. *Neka su $f, g : \mathbb{N} \to \mathbb{R}^+$ dvije funkcije. Kažemo da je funkcija g* ***asimptotska gornja međa*** *za funkciju f ako postoji realan broj $c > 0$ i $n_0 \in \mathbb{N}$ tako da za svaki $n \geq n_0$ vrijedi $f(n) \leq cg(n)$. Oznaka: $f(n) = O(g(n))$.*

U nastavku opisujemo algoritme nad binarnim polinomima: Zbrajanje, množenje, potenciranje, dijeljenje i faktorizaciju. Svi su idejno potekli iz [1].

Poglavlje 2

Zbrajanje binarnih polinoma

2.1 Ideja

U modulo 2 aritmetici vrijedi $1 +_2 1 \equiv 0(mod\ 2)$, $1 +_2 0 \equiv 1(mod\ 2)$ i $0 +_2 0 \equiv 0(mod\ 2)$. Također možemo primijetiti da je $-1 \equiv 1(mod\ 2)$, pa će rezultati zbrajanja i oduzimanja u aritmetici brojeva modulo 2 biti uvijek jednaki. Pogledajmo primjere i uvjerimo se:

(1) $(x^2 + x + 1) + (x + 1) = x^2 + 2x + 2$. No, budući da smo u aritmetici modulo 2 gdje znamo da vrijedi da je $2 \equiv 0(mod\ 2)$, konačni rezultat je x^2.

(2) $(x^2 + x + 1) - (x + 1) = x^2$.

Ideja koja se poklapa s modularnom aritmetikom u ovom slučaju je jednostavna i odnosi se na bitovni operator *XOR*. *XOR* predstavlja binarnu operaciju koja se definira na način da je istina ukoliko je točno jedan operand istinit; često se označava znakom $\oplus$ ili $\veebar$. Jednostavno je za provjeriti upravo na nekom od prethodnih primjera (npr. (1)). Bitovni zapis prvog sumanda je 111, a drugog 011 što, na koncu, daje 100 primjenom operatora *XOR* (naš polinom x^2).

2.2 Algoritam i složenost

Sam algoritam za zbrajanje je dosta jednostavan i temelji se na primjeni *XOR* operatora na dva pripadna operanda, pa nema smisla davati pseudokôd. Vremenska složenost, iako ne bi trebala ovisiti o načinu reprezentacije ulaznih podataka, na neki način i "ovisi". Zanimljivost leži u situaciji da ako reprezentiramo polinom kao "broj" s ograničenim prostorom koji je dovoljno malen da ga CPU obradi unutar jednog procesorskog takta, nemamo razloga smatrati bitovne operacije složenijim od $O(1)$ budući da notacija O predstavlja asimptotsku karakterizaciju vremena koje algoritam provede u računanju kada ulazni podaci postaju sve

veći i veći. Stoga ne smatramo značajnim ulaz koji je konstantne veličine. Registar će u, npr., 32-bitnoj arhitekturi moći primiti 2^{32} različitih ulaza. Dakle, 2^{32} različitih ulaza koji se pohrane u jedan registar što je konačno i bez mogućnosti da "naraste", pa onda i $O(1)$. No, što s polinomom milijarditog stupnja u nekoj vrlo općenitoj situaciji? Takav podatak se ne može jednoznačno pohraniti unutar jednog "broja" koji dozvoljava samo npr. 32 bita u nekoj računalnoj arhitekturi. Stoga smo prisiljeni pamtiti koeficijente unutar nekog polja ili znakovnog niza koji onda ovisi o duljini (ista je skalabilna) i gdje se za svaki koeficijent utroši više prostora za reprezentaciju. U takvoj situaciji bismo trebali smisliti način kako ostvariti XOR operaciju nad znakovima i tada bi se jedna usporedba izvrtila u jednom CPU taktu. Vrijedi, dakle, da složenost ovisi o duljini polinoma, pa bi korektnije bilo razmatrati ovakav općenitiji pristup koji onda zahtijeva jedan CPU takt za jednu usporedbu i stoga imamo složenost $O(n)$.

Poglavlje 3

Množenje binarnih polinoma

3.1 Osnovni algoritam

Najprirodniji algoritam za množenje polinoma općenito bi bio da "pomnožimo sve što se da pomnožiti". Neka su

$$A(x) = \sum_{i=0}^{n} a_i x^i \tag{3.1}$$

$$B(x) = \sum_{i=0}^{m} b_i x^i. \tag{3.2}$$

Tada definiramo

$$C(x) := \sum_{k=0}^{n+m} c_i x^i, \text{ gdje je } c_k := \sum_{i=0}^{k} a_i b_{k-i}, \;\; k = 0, 1, ..., m+n. \tag{3.3}$$

Uzmemo li u obzir restrikciju na binarne polinome i činjenicu da ih je jednostavno reprezentirati u bilo kojem programskom jeziku, lagano možemo pretočiti stvarne operacije koje se događaju "na papiru" u one koje CPU računa samostalno. Pomnožimo li tako neki polinom s x, potencija uz svaki postojeći član će se povećati za 1. Ekvivalent tome je bitovni posmak ulijevo za 1.

$$\begin{array}{lcll} (1x^4 + 0x^3 + 1x^2 + 0x^1 + 1x^0) & * & x & = (1x^5 + 0x^4 + 1x^3 + 0x^2 + 1x^1 + 0x^0) \\ 10101 & \ll & 1 & = 101010. \end{array}$$

Ukoliko želimo pomnožiti dva polinoma $A(x)$ i $B(x)$, to napravimo tako da množimo $A(x)$ sa svakim članom od $B(x)$ posebno (svugdje radimo bitovni posmak) i pritom zbrajamo u svakom koraku nastale sumande s već postojećima (operacija *XOR*). Zbrajanje se vrši u polju $\mathbb{Z}_2$.

Pseudokôd

Algoritam 1 Naivno množenje polinoma

```
1: function SIMPLE MULTIPLICATION(A , B)
2:     t ← 0
3:     while B do
4:         if B&1 then
5:             t ← t ^ A
6:         B ← B ≫ 1
7:         A ← A ≪ 1
8:     return t
```

Pogledajmo primjer:

$$\overbrace{(x^4+x^2+1)}^{A(x)} * \overbrace{(x^3+x^2)}^{B(x)} = \underbrace{(x^4+x^2+1)*x^3}_{10101 \ll 3} \overbrace{+}^{\text{XOR}} \underbrace{(x^4+x^2+1)*x^2}_{10101 \ll 2}$$
$$= \quad 10101000 \quad XOR \quad 01010100$$
$$= \quad 11111100$$
$$= \quad x^7+x^6+x^5+x^4+x^3+x^2.$$

Jedini važni trenuci algoritma su oni gdje se u polinomu $B(x)$ javljaju koeficijenti 1 uz potencije (dakle, uz x^2 i x^3). Ti sumandi će "preživjeti" i u tim trenucima se radi operacija *XOR*.

Prva bitna napomena je da nas ništa ne ograničava prilikom odluke kako reprezentirati koeficijente binarnog polinoma, no najlakše je razumjeti algoritam ukoliko zamislimo da su a i b brojevi koji sadrže nule i jedinice. Ovakvim prikazom postižemo kompromis utrošenog vremena i zauzetog prostora. Dobra strana bi bila da ako se argumenti funkcije čuvaju u tipu podatka koji odgovara broju, nad njima je lagano vršiti tražene operacije budući da bilo koji programski jezik ima prirodno definirane operacije logičkih operatora *AND* i *XOR* te bitovnog posmaka. Mana tog pristupa je ograničenost prikazivih podataka jer su tipovi podataka koji odgovaraju brojevima u nekom programskom jeziku strogo definiranih veličina. Naravno, ukoliko želimo, ulazne argumente smijemo prilagoditi arhitekturi i jeziku na način koji smatramo najprikladnijim (pohrana koeficijenata u polje, znakovni niz i sl.). Taj način nam nužno ne mora garantirati da postoje gotove i implementirane operacije *AND* i *XOR*, pa je dobro tu činjenicu imati na umu prilikom odabira.

Također, bitno je voditi računa o tipu podatka u koji spremamo polinomnu reprezentaciju jer može doći do preljeva. Tipična situacija bi bila kada imamo $deg(a) + deg(b) \geq$

`MAX_BITS_PER_TYPE` gdje varijabla `MAX_BITS_PER_TYPE` označava broj bitova koji služi za prikaz određenog tipa podatka. Svaka arhitektura koju podržava Linux definira varijablu `BITS_PER_LONG` u `<asm/types.h>` na veličinu `long` tipa u programskom jeziku C (na arm arhitekturi bi to bilo 32 bita npr.).

Složenost

Analizirajmo složenost. Petlja iz 3. linije ide po cijelom drugom faktoru koji je duljine $n + 1$. Ukoliko je uvjet iz 4. linije istinit, imamo operaciju zbrajanja dvaju operanada duljine $n + 1$. Bitovni posmaci su interesantni ali ih obzirom na već razmatranu situaciju kod zbrajanja, možemo smatrati "bilo kako" budući da uzimaju najviše $O(n)$ operacija što neće, vidjet ćemo u narednim rečenicama, narušiti složenost. Kada uzmemo sve u obzir, prevagu mogu donijeti samo linije 3 i 5. Obje zahtijevaju $n+1$ operacija u najgorem slučaju, pa se algoritam "značajno" vrti samo $(n + 1) \cdot (n + 1)$ puta što daje složenost $O(n^2)$.

3.2 Karatsubin algoritam

Zanima nas postoji li brži algoritam za množenje binarnih polinoma. Strategija podijeli pa vladaj[1] jedna je od najprimjenjivijih strategija za oblikovanje algoritama. Identičnu ideju koristi Karatsubin[2] algoritam kojeg ćemo sada predstaviti. Ideja je se zasniva na tome da se polazni problem podijeli u nekoliko manjih, ali istovrsnih problema. Tada se rješenje polaznog problema relativno lagano sklopi od manjih podproblema. Algoritam radi na rekurzivnom principu, a da bi primjena ovakvog pristupa uopće i imala smisla, potrebno je imati brzo i efikasno rješenje za dovoljno male podprobleme. Za polazni polinom $A(x)$ (parnog) stupnja N definiramo da je $A(x) = A_0(x) + A_1(x) \cdot x^{\frac{N}{2}}$. Vrijedi:

$$A_0(x) = a_0 + a_1 x + ... + a_{\frac{N}{2}-1} x^{\frac{N}{2}-1},$$
$$A_1(x) = a_{\frac{N}{2}} + a_{\frac{N}{2}+1} x + ... + a_N x^{\frac{N}{2}}.$$

Analogno definiramo $B_0(x)$ i $B_1(x)$ tako da vrijedi $B(x) = B_0(x) + B_1(x) \cdot x^{\frac{N}{2}}$. Sada imamo:

$$A \cdot B = A_0 B_0 + A_0 B_1 x^{\frac{N}{2}} + A_1 B_0 x^{\frac{N}{2}} + A_1 B_1 x^N.$$

Ovakvim postupkom smo sveli množenje dvaju polinoma stupnja N na 4 množenja (i 4 zbrajanja) stupnja $\frac{N}{2}$. To bi bila osnovna ideja algoritma podijeli pa vladaj koji bi u

[1] U literaturi je poznatiji pod engleskim terminom divide and conquer (D&C) ili kao divide et impera (lat.). Riječ je o znamenitoj latinskoj izreci koja se pripisuje Juliju Cezaru. Metodom Podijeli pa vladaj rimski konzul i general Aulo Gabinije je uspio pacifizirati Judeju podijelivši Židove na pet frakcija.

[2] Anatoly Alexeevitch Karatsuba (1937.– 2008.) bio je ruski matematičar za kojeg se smatra kako je prvi primijenio pristup podijeli pa vladaj na problem množenja.

konačnici opet dao $O(N^2)$ složenost. No, jednostavnim opažanjem produkt možemo još i bolje zapisati na sljedeći način:

$$A \cdot B = \underbrace{A_1 \cdot B_1}_{D_1}(x^{\frac{N}{2}} + x^N) + \underbrace{(A_1 - A_0) \cdot (B_0 - B_1)}_{D_{0,1}} x^{\frac{N}{2}} + \underbrace{A_0 \cdot B_0}_{D_0}(1 + x^{\frac{N}{2}}). \tag{3.4}$$

U ovakvom zapisu, 3 množenja su samo skupa i to ona koja su označena znakom $\cdot$, dok su množenja potencijom od x, u osnovi, samo bit posmaci koje smo i prije rekli ignorirati. Ovakva shema predstavlja algoritam za Karatsubino množenje polinoma (KA).

Pseudokôd

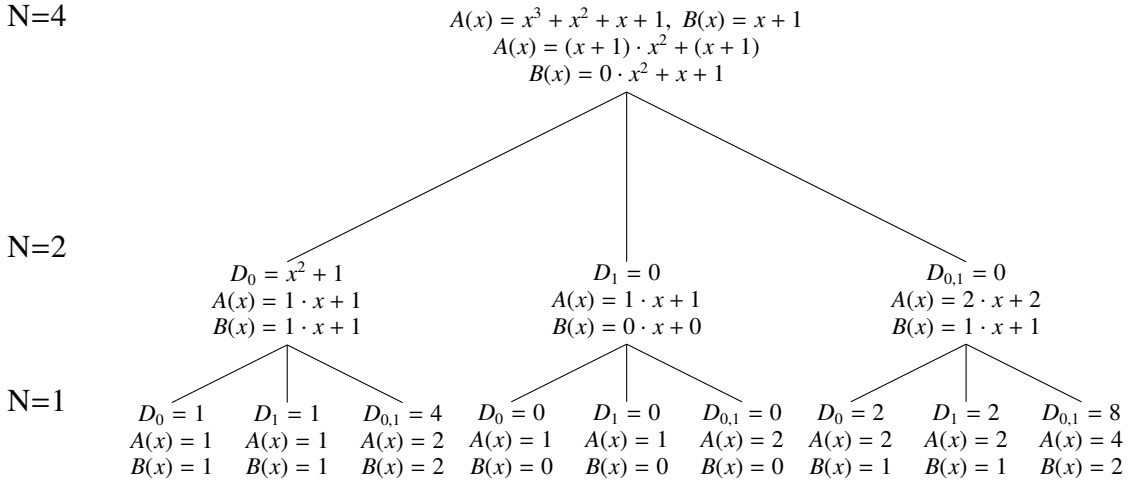
Prvotna ideja je analizirati i predstaviti rekurzivni KA za množenje polinoma stupnja $2^i - 1$ za neki $i \in \mathbb{N}$. To je poželjan zahtjev radi rekurzivnih i balansiranih poziva. Postoji i generalizirani pristup za polinome općeg stupnja, no budući da isti prelazi okvire rada, zainteresirani čitatelj za detalje može pogledati referencu [5]. Sada ćemo s N označiti broj koeficijenata. Također ćemo radi intuicije i korektnosti u općenitijem slučaju "primjenjivati" binarno oduzimanje koje je identično binarnom zbrajanju. Predloženi algoritam za polinome stupnja $2^i - 1$ (također iz [5]) radio bi na sljedećem principu:

Algoritam 2 Karatsubino množenje polinoma s 2^i koeficijenata

```
1:  function KA(A , B)
2:      N ← max(deg(A) + deg(B)) + 1
3:      if N == 1 then
4:          return A · B
5:      A ← A_0 + A_1 · x^{N/2}
6:      B ← B_0 + B_1 · x^{N/2}
7:      D_0 ← KA(A_0, B_0)
8:      D_1 ← KA(A_1, B_1)
9:      D_{0,1} ← KA(A_0 +_2 A_1, B_0 +_2 B_1)
10:     return D_1 x^N +_2 (D_{0,1} −_2 D_0 −_2 D_1) x^{N/2} +_2 D_0
```

Primjer

Za primjer izvrjednjavanja algoritma i izvršavanja rekurzije, razmotrit ćemo polinom $A(x) = x^3 + x^2 + x + 1$ i $B(x) = x + 1$. Ukoliko ručno pomnožimo ove polinome, jednostavno se dobije rezultat (u $\mathbb{Z}_2$, jasno) $C(x) = A(x) \cdot B(x) = x^4 + 1$.



Objasnit ćemo detaljnije kako smo generirali lijevo podstablo sa slike i proširiti priču analogno na ostala podstabla. Ulazni polinomi funkcije su $A(x)$ i $B(x)$. U 2. liniji se generira broj N koji ima vrijednost 4, a dobije se kao funkcija maksimuma između dva postojeća stupnja polinoma uvećana za jedan i odgovara broju koeficijenata u polinomu s većim stupnjem. U 5. i 6. liniji se generiraju gornja i donja polovica polinoma u skladu s prethodno danim formulama, pa dobijemo $A(x) = (x+1)\cdot x^2 + (x+1)$ i $B(x) = 0\cdot x^2 + x + 1$. Ti zapisi odgovaraju upravo onima s razine koju smo označili s $N = 4$. U naredne 3 linije će se događati generiranje elemenata D_0, D_1 i $D_{0,1}$ na što se odnosi nivo na dubini 1-djeca korijenskog čvora. Najprije se u liniji 7 događa rekurzivni poziv funkcije s argumentima $(x+1,\ x+1)$ i na dubini 1. Vrijednost broja N se mijenja i postaje 2. Potom se opet događa rastav oba polinoma na gornju i donju polovicu ($A(x) = 1\cdot x + 1$ i $B(x) = 1\cdot x + 1$). Nakon toga smo u liniji 7 i događa se rekurzivni poziv funkcije s parametrima $(1, 1)$. Konačno, budući da smo poslali polinome stupnja 0, ušli smo na dubinu 2 i broj N postaje jednak jedan. Umnožak se vraća natrag i to se sprema u varijablu D_0. Algoritam je sada ponovno na razini označenoj s $N = 2$, prvi put ulazi u liniju 8. Tu se sada izvrti cijeli postupak i spremi se rezultat u varijablu D_1. Potom se izračuna konačno i $D_{0,1}$ u liniji 9. Sada smo u situaciji gdje dosežemo 10. liniju na razini $N = 2$ i računamo rezultat po napisanoj formuli. Taj se rezultat spremi u varijablu $D_0 = x^2 + 1$ na dubini jedan i prelazi se na srednje dijete korijenskog čvora. Nakon što izračunamo sve vrijednosti $(D_0, D_1, D_{0,1})$ na razini $N = 2$, rezultat se spaja u konačni po formuli iz 10. linije što daje konačan rezultat.

Prije analize složenosti, dajmo jednu bitnu napomenu. Delikatna situacija se može dogoditi ukoliko malo modificiramo zadatak. Ovako:

$$A(x) = x^7 + x + 1$$
$$B(x) = x^7 + x^2 + 1.$$

Nakon ulaza u rekurziju, već u prvoj podjeli možemo uočiti potencijalnu opasnost.

$$A_0(x) = x + 1 \qquad B_0(x) = 1 + x^2$$
$$A_1(x) = x^3 \qquad B_1(x) = x^3.$$

Problematičan je polinom $B_0(x)$. Naime, parametri algoritma zahtijevaju polinome koji imaju stupanj za jedan niži od neke potencije broja 2, a ovdje to nije slučaj. Tako bismo u nekom dijelu gdje računamo D_0 imali situaciju da je N iz 3. linije predloženog algoritma jednak 3 što nam ne odgovara. Rezultat se može podesiti tako da "nadopunimo nulama" neku reprezentaciju polinoma, ali do te mjere da polinom $B_0(x)$ "postane" 3. stupnja iako je koeficijent uz član x^3, zapravo, jednak nula. No, samo nadopunjavanje nam može predstavljati implementacijsko opterećenje ako moramo u svakom trenutku brinuti o tome. Razlika bi se vidjela na implementaciji polinoma stringom u Javi recimo gdje operacije nad stringovima i nisu toliko brze. Ide li bolje? Da, pokazat ćemo nešto bolji algoritam, ali ipak prvo dajmo analizu složenosti koju smo najavili.

Složenost

Promotrimo sada forumulu 3.4 i Algoritam 2. Obzirom na predloženi pseudokôd i jedinu skupu operaciju (množenje), s pravom smijemo napisati da rješavamo rekurziju $T(N) = 3T(\frac{N}{2}) + cN$, gdje N označava broj koeficijenata polinoma i odgovara nekoj potenciji broja 2. Neka je $c \in \mathbb{N}$. Prisjetimo se: radi se o 3 rekurzivna poziva nad polinomima veličine $\frac{N}{2}$ (linije 7, 8 i 9) dok s cN označavamo broj "fiksnih poslova" nad polinomom s N koeficijenata koji se odrade u algoritmu. Napisat ćemo samo "c" radi jednostavnosti, ali će postati jasno da je riječ o fiksnom, prirodnom broju. Objasnimo:

- Generiranje varijabli A_0, A_1, B_0 i B_1 (linije 5 i 6) zahtijeva prolazak po originalnim polinomima A i B veličine N. Od "velikih" napravimo 4 u pola manja polinoma. To se može izvesti npr. u po jednom prolasku svakog od polinoma A i B. Dakle, treba nam $2N$ operacija.

- Računanje varijable N (linija 2) može zahtijevati, u najgorem slučaju, prolazak po oba ulazna polinoma veličine N i dodavanje broja 1. Budimo široke ruke i recimo bez straha da nam treba $3N$ koraka.

- Kombiniranje rezultata (linija 10) zahtijevaju 4 zbrajanja i oduzimanja polinoma s, u najgorem slučaju, N koeficijenata. Kompletan izračun, zajedno s bitovnim posmacima, može se napraviti u $6N$ operacija.

Čitatelju može zbunjujuće djelovati linija 4 gdje kažemo da se vrati umnožak brojeva. No, ovdje takav umnožak ne predstavlja vremensko opterećenje. Razlog je jednostavan: radi se o binarnim polinomima stupnja 0. Stoga, možemo postaviti uvjete da vrati rezultat 0

ili 1 u ovisnosti o ulaznim operandima za množenje i to dobijemo u konstantnoj složenosti. Budući da takve doista ne predstavljaju problem, nećemo ih uopće ni ubrajati u izračun.

Sada ćemo dati definiciju i tehnički rezultat koji će nam koristititi prilikom dokazivanja složenosti algoritma.

Definicija 3.2.1. *Neka su $f, g : \mathbb{N} \to \mathbb{R}^+$ dvije funkcije. Kažemo da funkcija f* ***raste istom brzinom*** *kao i funkcija g ako postoje realni brojevi $c_1, c_2 > 0$ i $n_0 \in \mathbb{N}$ tako da za svaki $n \geq n_0$ vrijedi $c_1 g(n) < f(n) < c_2 g(n)$. Oznaka: $f(n) = \Theta(g(n))$.*

Teorem 3.2.2 (Master teorem)**.** *Neka je $T : \mathbb{N} \to \mathbb{R}^+$ monotono rastuća funkcija za koju vrijedi:*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$
$$T(1) = c,$$

gdje su $a,\ b,\ c \in \mathbb{N}$ te vrijedi $a \geq 1,\ b \geq 2,\ c > 0$. Ako je $f(n) \in \Theta(n^d)$ za $d \in \mathbb{N},\ d > 0$, tada vrijedi:

$$T(n) = \begin{cases} \Theta(n^d) & ako\ je\ a < b^d \\ \Theta(n^d \cdot log(n)) & ako\ je\ a = b^d \\ \Theta(n^{log_b a}) & ako\ je\ a > b^d. \end{cases}$$

Dokaz. Pogledati [4]. □

Propozicija 3.2.3. *Algoritam 2 izvrši se u složenosti $\Theta(N^{log_2 3})$.*

Dokaz. Uz prethodno opisane razloge, razmatrat ćemo rekurziju $T(N) = 3T(\frac{N}{2}) + cN$, za $N = 2^k$. Dokaz vršimo pomoću Master teorema u 2 koraka.

1. $T(N)$ je monotono rastuća:
   Dokaz vršimo indukcijom po k, za $k \in \mathbb{N}_0$.

   - Baza: Lako se vidi da za polinome stupnja 0 vrijedi T(1)=1.
   - Korak: Pretpostavimo da vrijedi tvrdnja $T(2^{k-1}) \leq T(2^k)$. Zanima nas sljedeće:

     $$T(2^k) \overset{?}{\leq} T(2^{k+1})$$

     No, ta tvrdnja lagano slijedi iz pretpostavke i činjenice da vrijedi $c \cdot 2^k \leq c \cdot 2^{k+1}$ za $c > 0$. Dakle, imamo: $3T(2^{k-1}) + c \cdot 2^k \leq 3T(2^k) + c \cdot 2^{k+1}$. Time smo proveli indukciju po k.

2. $f(N) = \Theta(N)$:
   U našem algoritmu vrijedi $f(N) = cN$. Tvrdnja lagano slijedi uz $c_1 := \frac{c}{2}$, $c_2 := 2c$ i $n_0 = 1$.

Uzevši u obzir upravo pokazano i predloženu rekurziju, sada je očito kako vrijedi $a = 3, b = 2$, $c = 1$ te $d = 1$. Napokon, treći slučaj Master teorema daje tvrdnju iz propozicije. □

Napomena 3.2.4. *Iz definicija 5.0.2 i 3.2.1, jasno se vidi kako, uz prikladnu definiciju funkcije $g(n)$ u skladu sa spomenutim definicijama, vrijedi odnos $\Theta(g(n)) \subseteq O(g(n))$. Tada ćemo, radi konzistentnosti prilikom dokazivanja tvrdnji u O notaciji, reći kako smo dokazali da se prethodni algoritam izvrši u $O(N^{log_2 3})$.*

Ostali smo dužni još poboljšanu verziju Karatsubinog algoritma gdje jednostavnom implementacijskom finesom možemo jeftino dobiti i verziju za općeniti stupanj. Promjena će se svoditi na sam početak metode. Naime, uvedemo li pravilo da polinom $A(x)$ mora biti većeg stupnja i da terminalni uvjet rekurzije postane provjera nad polinomom $B(x)$, stvar radi za polinome općenitih stupnjeva. Predložimo Karatsubin algoritam za općenite stupnjeve (ovaj algoritam će ići i u implementaciju):

Algoritam 3 Karatsubino množenje polinoma

```
1:  function KARATSUBA MULTIPLICATION(A , B)
2:      if deg(A)<deg(B) then
3:          swap(A, B)
4:      if sizeOf(B) == 0 then
5:          return 0
6:      if sizeOf(B) == 1 then
7:          return A
8:      N ← sizeOf(A)/2
9:      A_0 ← A[ :N], A_1 ← A[N: ]
10:     B_0 ← B[ :N], B_1 ← B[N: ]
11:     D_0 ← KARATSUBA MULTIPLICATION(A_0, B_0)
12:     D_1 ← KARATSUBA MULTIPLICATION(A_1, B_1)
13:     D_{0,1} ← KARATSUBA MULTIPLICATION(A_0 +_2 A_1, B_0 +_2 B_1)
14:     return D_1 x^N +_2 (D_{0,1} −_2 D_0 −_2 D_1) x^{N/2} +_2 D_0
```

Prvo što treba spomenuti da algoritam ne narušava predloženu složenost. Jedina razlika se sastoji u početnim linijama i eventualnoj zamjeni $A(x)$ i $B(x)$ u slučaju da je polinom $A(x)$ manjeg stupnja. Zamjena (ili inicijalizacija "novih polinoma") se može napraviti i u linearnoj složenosti, pa ne remeti prethodni rezultat. Intuitivno, metodom `swap(A, B)` obavljamo "zamjenu" polinoma: sadržaji polinoma $A(x)$ i $B(x)$ će biti zamijenjeni nakon

završetka metode. Od ovog trenutka nadalje ćemo koristiti metodu `sizeOf(A)` npr. koja će raditi posao ekvivalentan $deg(A) + 1$, a vraća stvarnu "duljinu" polinoma.

Hibridni Karatsuba

Ostalo je još prostora za kratko poboljšanje. U Uvodu smo najavili i Hibridni algoritam. Što se miješa? Karatsubin algoritam i algoritam za naivno množenje polinoma. Prvo dajmo motivaciju koja ipak ne može proći bez implementacijskih motiva, pa onda i posljedica. Viši programski jezici (kao što su C, Python i Java) imaju dosta "skuplju" implementaciju rekurzije u odnosu na iteriranje. To se događa zbog zahtjeva za alokacijom novog stoga. Budući da Karatsuba koristi rekurzivne pozive koji su dosta skupi u odnosu na nerekurzivni (iterativni) pristup naivnog algoritma, vrijedilo bi pokušati kombinirati ta dva principa na način da mane jednoga zamijenimo prednostima drugoga. Tako ćemo uvesti novi filter u vidu *limita*. Limitirat ćemo da ako smo u situaciji da množimo polinome koji su "dovoljno malog stupnja" ($max\{deg(A), deg(B)\} \leq limit$), da ih pomnožimo obično i ne idemo dublje u rekurziju. Trebamo samo odlučiti koji parametar ćemo proslijediti kao *limit* i to je sve. Radi kompletnosti rješenja (i kasnije implementacije), dajmo ipak i pseudokôd koji je već i vrlo intuitivan i gotovo identičan običnom Karatsubinom algoritmu:

Algoritam 4 Hibridni Karatsubin algoritam

```
1:  function KARATSUBA HYBRID(A , B, limit)
2:      if deg(A)<deg(B) then
3:          swap(A, B)
4:      if sizeOf(B) == 0 then
5:          return 0
6:      if sizeOf(B) == 1 then
7:          return A
8:      if sizeOf(A) ≤ limit then
9:          return SIMPLE MULTIPLICATION(A, B)
10:     N ← sizeOf(A)/2
11:     A_0 ← A[ :N], A_1 ← A[N: ]
12:     B_0 ← B[ :N], B_1 ← B[N: ]
13:     D_0 ← KARATSUBA HYBRID(A_0, B_0, limit)
14:     D_1 ← KARATSUBA HYBRID(A_1, B_1, limit)
15:     D_{0,1} ← KARATSUBA HYBRID(A_0 +_2 A_1, B_0 +_2 B_1, limit)
16:     return D_1 x^N +_2 (D_{0,1} −_2 D_0 −_2 D_1) x^{N/2} +_2 D_0
```

Poglavlje 4

Potenciranje binarnih polinoma

Za efikasno potenciranje, važno je imati "dobru" metodu za množenje polinoma. Najjednostavniji pristup za računanje $p(x)^n$ bi bio da vršimo $n-1$ sukcesivnih množenja unutar grupe. No, uzmemo li u obzir samo primjenu binarnih polinoma u kriptografiji gdje red grupe tipično premašuje 2^{160} elemenata (a može ići i preko 2^{1024}), postaje jasno kako bi $n-1$ uzastopnih množenja bilo poprilično loše. Postoje dva načina za reduciranje vremena potrebnog za potenciranje. Jedan pristup bi bio da smanjimo vrijeme potrebno za množenje dvaju elemenata unutar grupe, dok bi drugi obuhvaćao smanjenje broja množenja prilikom računanja $p(x)^n$. U idealnom slučaju, algoritam bi sadržavao oba pristupa. U nastavku ćemo razmotriti algoritme nad proizvoljnim polinomima s proizvoljnim eksponentom, a bazirat će se na operacijama kvadriranja te množenja polinoma. Dodajmo da ima smisla ići u smjeru smanjenja broja operacija jer smo već analizirali vremensku složenost množenja i prikazali neka dostignuća s kojima ćemo se, u ovom radu, i zadovoljiti.
Budući da smo se množenjem bavili u prethodnom poglavlju, dajmo kratak algoritam za kvadriranje. Vrijedi činjenica da prilikom kvadriranja binarnog polinoma $p(x) = \sum_{k=0}^{d} a_k x^k$ stupnja d, dobijemo $p^2(x) = \sum_{k=0}^{d} a_k x^{2k}$. Tada je lako primijetiti da svaki član na poziciji k u zapisu polinoma pomoću niza bitova, trebamo samo pomaknuti na poziciju $2k$.

Algoritam 5 Kvadrat polinoma

```
1: function POLYNOMIAL SQUARE(A)
2:     t ← 0
3:     m ← 1
4:     while A do
5:         if A&1 then t ← t ^ m
6:         m ← m ≪ 2
7:         A ← A ≫ 1
8:     return t
```

4.1 S desna na lijevo binarno množenje

Pseudokôd

Algoritam 6 S desna na lijevo

```
1: function RIGHT-TO-LEFT BINARY EXPONENTIATION(p, n)
2:     A ← 1
3:     S ← p
4:     while (n ≠ 0) do
5:         if n & 1 then A ← MULTIPLY (A, S)
6:         n ← ⌊n/2⌋
7:         if n ≠ 0 then S ← POLYNOMIAL SQUARE(S)
8:     return A
```

Napomenimo samo da smo u liniji 5 metodu množenja dva polinoma (proizvoljnim algoritmom) nazvali `MULTIPLY`.

Primjer

Sjedeći primjer prikazuje vrijednosti od `A`, `n` i `S` tokom svake od iteracija prilikom računanja $p(x)^n$ za $n = 283$.

A	1	p	p^3	p^3	p^{11}	p^{27}	p^{27}	p^{27}	p^{27}	p^{283}
n	283	141	70	35	17	8	4	2	1	0
S	p	p^2	p^4	p^8	p^{16}	p^{32}	p^{64}	p^{128}	p^{256}	-

Složenost

Neka je $t + 1$ duljina bitovne reprezentacije broj n u bazi 2. Označimo s $wt(n)$ broj jedinica u takvom zapisu. Algoritam 6 napravi t kvadriranja i $wt(n) - 1$ množenja. Ako je n izabran slučajno iz raspona $0 \leq n < |GF(2^m)| = n$ za neki m, tada možemo računati na $\lfloor log_2 n \rfloor$ kvadriranja i $\frac{1}{2}(\lfloor log_2 n \rfloor + 1)$ množenja (u prosječnom slučaju). Najgori slučaj (svi bitovi jednaki 1) će zahtijevati jednak broj množenja i kvadriranja. Pridruživanje $1 \cdot x$ se ne računa pod množenje, kao što se niti $1 \cdot 1$ ne računa pod kvadriranje.

4.2 K-arno potenciranje s lijeva na desno

Umjesto da razbijemo eksponent na bitove u binarnoj reprezentaciji, možemo ih razbiti i na veće dijelove u bazi b koji mora biti oblika 2^k. Tada eksponent n može biti zapisan kao

$n = n_i b^i + n_{i-1} b^{i-1} + ... + n_0 b^0$. Pritom smo s n_i označili znamenke u prikazu eksponenta n u bazi b. Sada, ako računamo p^n vrijedi: $p^n = p^{n_i b^i} \cdot p^{n_{i-1} b^{i-1}} \cdot ... \cdot p^{n_0}$. Parametar k zvat ćemo širinom prozora i mora biti barem 1. Algoritam prima i traženu potenciju n, ali kao npr. polje znakova (ili string) koji odgovaraju zapisu tog broja u bazi b.

Pseudokôd

Algoritam 7 K-arno potenciranje s lijeva na desno

```
1:  function LEFT-TO-RIGHT K-ARY EXPONENTIATION(p, n=(n_t n_{t-1}... n_1, n_0)_b, k)
2:      g_0 ← 1
3:      for i=1 to 2^k do
4:          g_i ← MULTIPLY(g_{i-1}, p)
5:      A ← 1
6:      i ← t
7:      while (i ≥ 0) do
8:          for j=0 to k do
9:              A ← POLYNOMIAL SQUARE (A)
10:         A ← MULTIPLY(A, g_{n_i})
11:         i ← i − 1
12:     return A
```

Primjer

Uzmimo za primjer $b = 2$ i $n = 283$. Tada vrijedi da je $k = 1$, $n = (100011011)_2$ i $t = 8$. Predradnje i izračuni tada uključuju računanje $g_0 = 1$ i $g_1 = p$. Pokažimo na primjeru izračun $p(x)^n$.

i	n_i	A
8	1	p
7	0	p^2
6	0	p^4
5	0	p^8
4	1	p^{17}
3	1	p^{35}
2	0	p^{70}
1	1	p^{141}
0	1	p^{283}

Složenost

Neka $t+1$ označava bitovnu duljinu od n. Također, neka je $l+1$ broj k-bitnih riječi nastalih iz n. Odnosno, $l = \lceil \frac{t+1}{k} \rceil - 1 = \lfloor \frac{t}{k} \rfloor$. Broj kvadriranja u algoritmu jednak je lk.

Kratko se samo zaustavimo i dajmo pregled složenosti s ujednačenim notacijama. Uspoređujemo obzirom na notaciju O. Dakle, obzirom na najgori slučaj s pripadnim operacijama. Primijetimo: $lk = \lfloor \frac{t}{k} \rfloor \cdot k = t - t \,(mod\ k)$. Slijedi da je $t - (k-1) \leq lk \leq t$. Iz ovoga se vidi da Algoritam 7 u odnosu na Algoritam 6 može uštediti do $k-1$ kvadriranja. Optimalna vrijednost za k ovisi o broju t. Broj množenja nije diskutabilan čak ni za slučaj $k = 1$. Pogledajmo:

Algoritam	Predračun		#Kvadriranja	#Množenja
	#Kv	#Mn		
6	0	0	t	t
7	1	$2^k - 3$	$t - (k-1) \leq lk \leq t$	$l - 1$

Propozicija 4.2.1. *Za optimalnu veličinu parametra k najbolje je uzeti najmanji k za kojeg vrijedi:*

$$log(n) < \frac{k(k+1) \cdot 2^{2k}}{2^{k+1} - k - 2} + 1.$$

Dokaz. Budući da dokaz prelazi okvire rada i ovdje je samo u svrhu zanimljivosti, za više informacija čitatelju se preporučuje pogledati [3]. □

4.3 Potenciranje pomoću klizećih prozora

Ideja ovog algoritma se bazira na činjenici kako se pojedini dijelovi binarne forme eksponenta mogu češće ponavljati, te bismo, stoga, ovim algoritmom mogli reducirati prosječan broj množenja i možemo ga shvatiti kao efikasnu varijantu prethodne metode. Širinu prozora označavamo s k i mora biti barem 1. Algoritam prima i traženu potenciju n, ali kao npr. polje znakova (ili string) koji odgovaraju binarnom zapisu tog broja. Bez pseudokôda ćemo dati metodu `LONGEST BITSTRING(n)`. Ta metoda vraća najdulji podniz/podstring $n_i n_{i-1} ... n_l$ sa svojstvom $i - l + 1 \leq k \wedge n_l = 1$.

Pseudokôd

Algoritam 8 Klizeći prozor

```
1:  function SLIDING-WINDOW EXPONENTIATION(p, n=(n_t n_{t-1}... n_1, n_0)_2, k)
2:      g_1 ← p
3:      g_2 ← p^2
4:      for i=1 to 2^{k-1} do
5:          g_{2i+1} ← g_{2i-1} · g_2
6:      A ← 1
7:      i ← t
8:      while (i ≥ 0) do
9:          if n_i == 0 then
10:             A ← POLYNOMIAL SQUARE (A)
11:             i ← i − 1
12:         else
13:             j = (n_i n_{i-1}... n_l) ← LONGEST BITSTRING(n)
14:             for i=0 to sizeOf(j) do
15:                 A ← POLYNOMIAL SQUARE (A)
16:             A ← MULTIPLY(A, g_{(j)_2})
17:             i ← l − 1
18:     return A
```

Primjer

Pokažimo primjerom korak po korak uz $n = 11749 = (10110111100101)_2$ i $k = 3$. Primijetiti valja kako metoda klizećeg prozora za ovakve parametre zahtijeva 3 bitna množenja samo: za $i = 7, 4, 0$. Pogledajmo tablicu:

i	eksponent	A	prozor
13		1	101
10	101	g^5	101
7	101101	$g(^5)^8 g^5 = g^{45}$	111
4	101101111	$g(^{45})^8 g^7 = g^{367}$	-
3	1011011110	$g(^{367})^2 = g^{734}$	-
2	10110111100	$g(^{734})^2 = g^{1468}$	101
0	10110111100101	$g(^{1468})^8 g^5 = g^{11749}$	-

Napomena 4.3.1. *Ovdje ćemo znakom "-" označavati situaciju kad se ne računa prozor, već se samo obavlja kvadriranje i prelazak na sljedeći bit. To se događa u slučaju nailaska na bit* 0 *(9.linija u algoritmu 8).*

Poglavlje 5

Dijeljenje binarnih polinoma

Definicija 5.0.1. *Neka su polinomi P, Q, P_1, R s koeficijentima u GF(2).* ***Kvocijent*** *binarnih polinoma P i Q se definira na sljedeći način:*

$$\frac{P(x)}{Q(x)} := P_1(x) + \frac{R(x)}{Q(x)}.$$

Ako je $deg(P) = n > m = deg(Q)$, onda je P_1 polinom stupnja $n - m$. Za polinom R kažemo da je ostatak pri dijeljenju i to je polinom stupnja manjeg od m.

Budući da je binarna aritmetika generalno jednostavnija od aritmetike prirodnih, realnih ili kompleksnih brojeva, za očekivati je da možemo pojednostaviti dijeljenje binarnih polinoma. U nastavku slijedi algoritam koji će davati kvocijent i ostatak pri dijeljenju, a bazirat će se na jednostavnim operacijama binarnog zbrajanja uz dodatak da trebamo metodu koja pronalazi indeks prvog bita koji je jednak 1 u binarnom zapisu broja. To je, ustvari, metoda `DEG(P)`.

Napomena 5.0.2. *U predstojećem pseudokôdu ćemo vratiti kvocijent `q`, dok će ostatak ostat pohranjen u varijabli `a`.*

Pseudokôd

Algoritam 9 Kvocijent polinoma

```
1:  function QUOTIENT(a, b)
2:      db ← DEG(b)
3:      q ← 0
4:      da ← 0
5:      while db ≤ da = DEG(a) do
6:          if 0 == a then
7:              break
8:          a ← a ^ (b ≪ (da − db))
9:          q ← q ^ (1 ≪ (da − db))
10:     return q
```

Primjer

Pokažimo na primjeru algoritam za dijeljenje polinoma. Zanima nas količnik i ostatak pri dijeljenju polinoma $a(x) = x^9 + x^8 + x^6 + x^4 + x^2$, te $b(x) = x^3 + x + 1$. Računajući db u prvom koraku, vidimo da je $db = 3$. Pogledajmo tablicu koja daje za svaki korak izračun stupnja polinoma iz varijable a, kvocijenta $q(x)$ i ostatka koji ostane pohranjen u $a(x)$:

da	$a(x)$	$q(x)$
9	110010100	1000000
8	11110100	1100000
7	1000100	1110000
6	11100	1111000
4	1010	1111010
3	1	1111011

Složenost

Neka najdulji polinom ima bitovnu reprezentaciju duljine N. Metoda `DEG(P)` može zahtijevati u najgorem slučaju složenost $O(N)$. Bit posmaci i operacija *XOR* se također, u najopćenitijem slučaju, izvršavaju u složenosti $O(N)$. Opterećenje stvara petlja iz linije 5 koja se u najgorem slučaju može izvršiti u $O(N)$ što onda daje ukupnu složenost od $O(N^2)$

Poglavlje 6

Faktorizacija binarnih polinoma

"Success is going from failure to failure without losing enthusiasm."

— Winston Churchill

U ovom poglavlju ćemo detaljno predstaviti faktorizaciju binarnih polinoma. U prvom dijelu ćemo opisati faktorizaciju koji ne sadrže kvadratni faktor, u drugom dijelu ćemo dati algoritam za detekciju i uklanjanje kvadratnih faktora i na koncu ćemo dati univerzalni algoritam za faktorizaciju binarnih polinoma. Budući da je algoritama i međukoraka dosta, neke algoritme ćemo samo spomenuti i dati ih bez gotove implementacije. No, najprije se podsjetimo osnovnih definicija.

Definicija 6.0.1. *Za polinom $f(x) \in F[x]$ ćemo reći da je **ireducibilan** ako ne postoje $g(x)$, $h(x) \in F[x]$ takvi da vrijedi $f(x) = g(x)h(x)$.*

Sada, faktorizirati polinom znači napisati ga u obliku umnoška ireducibilnih faktora.

6.1 Faktorizacija kvadratno slobodnih polinoma

Za faktorizaciju ovakvih polinoma, koristit ćemo algoritam Berlekampove[1] Q-matrice opisan u [2]. No, najprije definirajmo što su to kvadratno slobodni polinomi:

[1] Elwyn Ralph Berlekamp (1940.-) američki je matematičar i professor emeritus na Berkeleyju. Berlekamp je poznat po teoriji kodiranja i kombinatornoj teoriji igara.

Definicija 6.1.1. *Za polinom $f(x) \in F[x]$ ćemo reći da je **kvadratno slobodan** ako ne postoji polinom $g(x) \in F[x]$ stupnja barem 1 tako da $g^2 \mid f$.*

Napomena 6.1.2. *Reć ćemo da polinom c ima **kvadratni faktor** ukoliko nije kvadratno slobodan.*

Sam algoritam se sastoji od dva glavna koraka: računanje jezgre matrice te faze profinjavanja koja pronalazi različite ireducibilne faktore. Neka je c binarni polinom stupnja d. Q-matrica je matrica dimenzija $d \times d$ čiji se n-ti stupac može izračunati kao $x^{2n} \ (mod\ c)$ gdje n ide od 0. Algoritam će koristiti jezgru matrice $Q - I$. Pokažimo najprije kako se računa matrica Q.

Algoritam 10 Računanje matrice Q

```
1: function Q-MATRIX(c)
2:     q ← 0,  q_1 ← 1,  Q[0] ← q
3:     x_2 ← x
4:     x_2 ← BITPOL MOD (MULTIPLY (x_2, x_2), c)
5:     for k = 1 to deg(c) do
6:         q ← BITPOL MOD (MULTIPLY (q_1, x_2), c)
7:         Q[k] ← q
8:         k ← k + 1
9:     return TRANSPOSE(Q)
```

Metoda `BITPOL MOD(A, C)` zadužena je za računanje izraza $A \ (mod\ C)$ dok metoda `TRANSPOSE(Q)` vraća transponiranu, kvadratnu matricu dimenzija $d \times d$ koja na kraju predstavlja i našu matricu Q. Koristimo još i metodu `DEG(c)` koja vraća polinom stupnja `c`. Standarno, metoda `GCD(a, b)` Euklidovim algoritmom pronalazi polinom koji je najveći zajednički djelitelj polinoma iz argumenata i u opisu rada, također, dolazi bez pseudokôda.

Prvi korak završavamo računanjem jezgre matrice $Q - I$. Rang jezgre označava broj različitih ireducibilnih faktora ukoliko je polinom c kvadratno slobodan. Jezgru matrice Q računamo metodom Gaussovih eliminacija. U radu nećemo opisivati sam algoritam i bit će naznačen metodom `NULSPACE(Q)` u nastavku.
Da bismo pronašli ireducibilne faktore od c, jezgra se mora dodatno "obraditi". Krenimo s opisom algoritma. Potrebno je razbiti početni polinom c na faktore s manjim stupnjem. To se napravi računanjem `GCD(c, b)` tih polinoma. Polinom `b` se dobije kao razlika elementa jezgrine baze i svih elemenata iz $GF(2)$ (što su samo 0 i 1). Isti proces se, potom, primjenjuje na dobivene faktore i tako sve dok broj faktora ne bude jednak broju elemenata jezgre. Napomenimo da će u jezgri matrice $Q - I$ uvijek biti vektor $(1, 0, ..., 0)$, pa njega nećemo ni uzimati u obzir. Pogledajmo:

Algoritam 11 Profinjenje faktora

```
1: function FACTORS REFINING(c, kernel)
2:     factors ← c
3:     r ← 1
4:     while sizeOf(factors) < sizeOf(kernel) do        // za sve elemente t iz jezgre
5:         for each u ∈ factors do
6:             for each s ∈ GF(2) do
7:                 g ← GCD(kernel[r] −_2 s, u)
8:                 if g ≠ 1 ∧ g ≠ u then
9:                     w ← u/g
10:                    factors ← factors − {u} ∪ {g, w}
11:                if sizeOf(factors) == sizeOf(kernel) then
12:                    return factors
13:        r ← r + 1
14:    return factors
```

Ujedinimo sve što smo napravili do sada kroz sljedeći algoritam:

Algoritam 12 Berlekampov algoritam

```
1: function BERLEKAMP'S ALGORITHM(c)
2:     Q ← Q-MATRIX(c)
3:     kernel ← NULLSPACE(Q-I)
4:     return FACTORS REFINING(c, kernel)
```

Pokažimo na primjeru rad algoritma. Neka je naš polinom $c(x) = x^7 + x^3 + x + 1$. Najprije trebamo formirati matrice Q i $Q - I$:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}}_{\text{Q}}, \qquad \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}}_{\text{Q-I}}.$$

Baza za jezgru se računa tako da transponiramo matricu $Q - I$ npr., dodamo joj s desne strane jediničnu matricu i vršimo poništavanje sve dok na lijevoj strani ”nemamo što za

poništiti", a vektori baze se nalaze desno od nul-redaka. Na taj način dobijemo vektore $kernel^{[0]} = (1,0,0,0,0,0,0)$, $kernel^{[1]} = (0,1,0,0,1,0,0)$ i $kernel^{[2]} = (0,1,0,1,0,1,1)$. Navedeni vektori, u smislu polinomne reprezentacije, odgovaraju polinomima $kernel^{[0]}$(x) = 1, $kernel^{[1]}$(x) = $x + x^4$ i $kernel^{[2]}$(x) = $x + x^3 + x^5 + x^6$. Slijedeći algoritam za profinjavanje faktora i počevši od $kernel^{[1]}$, lako se dobije tražena faktorizacija: $(1 + x)(1 + x + x^2)(1 + x + x^4)$.

6.2 Ekstrahiranje kvadratno slobodnih članova polinoma

Za testiranje slučaja gdje se pitamo ima li polinom c kvadratni faktor, trebamo izračunati $g = gcd(c, c')$ gdje je c' derivacija polinoma c. Ako je $g \neq 1$, tada c ima kvadratni faktor g. Neka je npr. $c = a \cdot b^2$, tada vrijedi $c' = a' \cdot b^2 + 2a \cdot b \cdot b' = a'b^2$. Tada je jasno da $gcd(c, c') = b^2$

Opisani algoritam nije teško pretočiti u pseudokôd:

Algoritam 13 Testiranje kvadratičnosti

```
1: function TEST SQUARE FREE(c)
2:     d ← BITPOL DERIVE(c)
3:     if d == 0 then
4:         return (1 == c ? 0 : c)
5:     g ← GCD(c, d)
6:     return (1 == g ? 0 : g)
```

`BITPOL DERIVE(c)` će u nastavku označavati metodu za deriviranje polinoma. Smatrat ćemo da je ista trivijalna za napraviti, pa je dajemo bez pseudokôda. Zanimljivu situaciju imamo ako je derivacija polinoma jednaka nula. Onda je početni polinom ili savršen kvadrat ili konstantan. Konstantan polinom je očit, a savršen kvadrat se lako vidi iz činjenice da ako imamo samo polinom s parnim potencijama da isti "umire" deriviranjem radi aritmetike[2] u $GF(2)$. Napravimo sada algoritam koji provjerava je li ulazni polinom čisti kvadrat različit od 1:

[2]Parne potencije u kvadriranom polinomu se lako dobiju idejom korištenom u Algoritmu 5 budući da "samo" radimo bitovni posmak u lijevo za "duplo".

Algoritam 14 Čisti kvadrat polinoma stupnja većeg od 1

```
1: function TEST PURE SQUARE(c)
2:     if c == 1 then
3:         return 0
4:     c ← c ∧ a
5:     return 0 == c
```

Ovdje ćemo varijablom `a` smatrati niz koji se sastoji od naizmjeničnih nula i jedinica. U programskom jeziku C bismo mogli npr. proslijediti varijablu koja je tipa `ulong` i naznačiti, npr., `ulong a = 0xaaaaaaaaaaaaaaaaUL`. Ukoliko je polinom savršen kvadrat, možemo izračunati i njegov kvadratni korijen.

Algoritam 15 Korijen savršenog polinoma

```
1: function PURE SQRT(c)
2:     t ← 0, mc ← 1, mt ← 1
3:     while deg(mc) ≤ deg(c) do
4:         if mc ∧ c then
5:             t ← t | mt
6:         mc ← mc ≪ 2
7:         mt ← mt ≪ 1
8:     return t
```

Za algoritam faktorizacije za generalni slučaj polinoma, moramo ekstrahirati produkt svih različitih ireducibilnih faktora (kvadratno slobodan dio) polaznog polinoma. Dajmo sada algoritam koji vraća polinom u kojem su parni eksponenti reducirani:

Algoritam 16 Reducirani polinom

```
1: function REDUCE(c)
2:     s ← TEST SQUARE FREE(c)
3:     if 0 == s then
4:         return c        // c is square free
5:     f ← BITPOL DIV(c, s)
6:     while TEST PURE SQUARE(s, a) do
7:         s ← PURE SQRT(s)
8:     g ← GCD(s, f)
9:     s ← BITPOL DIV(s, g)
10:    f ← MULTIPLY(s, f)
11:    return f
```

Spomenimo da metoda `BITPOL DIV(s, g)` računa rezultat pri dijeljenju polinoma `s` polinomom `g`. Konačno, za ekstrahiranje kvadratno slobodnih dijelova polinoma treba pozivati metodu `REDUCE(c)` dok ne dobijemo da reducirani polinom odgovara ulazu:

Algoritam 17 SQUARE FREE PART

```
1: function SQUARE FREE(c)
2:     z ← c
3:     while z ≠ (t ← REDUCE(z)) do
4:         z ← t
5:     return z
```

Metoda za redukciju će biti pozvana najviše $log_2(n)$ puta za polinom stupnja n. Najgori slučaj je, dakako, savršena potencija $p = a^{2^k-1}$ gdje je $2^k - 1 \leq n$. Primijetiti valja kako je $2^k - 1 = 1 + 2(2^{k-1} - 1)$, pa će se redukcijom p podijeliti kao $p = a \cdot s^2 \mapsto a \cdot s$ gdje je, opet, s oblika $a^{2^{k-1}-1}$ što opet daje početnu formu.

6.3 Faktorizacija polinoma općenitog stupnja

Metoda za faktorizaciju za općeniti slučaj ekstrahira kvadratno slobodan dio f od ulaza c, koristi Berlekampov algoritam za faktoriziranje f i potom "osvježava" eksponente u skladu s polinomom $s = c/f$. Napomenimo kako se metoda za računanje jezgre poziva samo jednom, a možemo smatrati da metoda `write(f, e)` radi "lijepi" ispis faktoriziranog polinoma s pripadnim potencijama tih faktora. Evo konačno i samog algoritma:

Algoritam 18 Faktorizacija u općenitom slučaju

```
1:  function FACTORIZATION(c, f, e)
2:      if DEG(c) ≤ 1 then        // trivijalni slučajevi: 0, 1, x, x+1
3:          return c
4:
5:      cf ← SQUARE FREE(c)       // dohvati kvadratno slobodan dio
6:      fct ← BERKLEKAMP'S ALGORITHM(cf, f)       // ... i faktoriziraj ga
7:
8:      for j = 0 to fct do       // svi eksponenti su 1
9:          e[j] ← 1
10:         j ← j + 1
11:
12:     // Ovdje f[], e[] su valjane faktorizacije kvadratno slobodnog dijela od cf
13:
14:     // Osvježi eksponente s kvadratnim dijelom
15:
16:     cs ← BITPOL DIV(c, cf)
17:     for j = 0 to fct do
18:         if 1 == cs then break
19:         fj ← f[j]
20:         g ← GCD(cs, fj)
21:         while 1 ≠ g do
22:             e[j] ← e[j] + 1
23:             cs ← BITPOL DIV(cs, fj)
24:             if 1 == cs then break
25:             g ← GCD(cs, fj)
26:         j ← j + 1
27:     write(f, e)
28:     return f
```

Poglavlje 7

Rezultati testiranja i primjene

7.1 Detalji implementacije

U završnoj fazi ćemo prikazati rezultate kroz dvije grupe. Prva testna grupa će se odnositi na algoritme koje uspoređujemo unutar neke operacije i tu ćemo obraditi množenje. Druga grupa će uključivati testiranje ostalih operacija: zbrajanje, dijeljenje i faktorizaciju. Spomenute sadrže samo po jedan algoritam te ćemo ih testirati obzirom na veličinu ulaza.
Testove izvodimo na računalu s Intel® Core™2 Duo Processor T7200 procesorom i radnom memorijom od 2GB. Svi algoritmi su implementirani u programskom jeziku *Java* (verzija 1.8.0_111) izuzev algoritama za operaciju množenja koji su u *Pythonu* (verzija 3.6.0). Python je omogućio jednostavniju implementaciju algoritama nad množenjem radi lakše manipulacije s tipovima podataka: Java-dio koristi stringove ili liste za reprezentaciju polinoma i svih operacija nad njima; Python-dio koristi bit-liste (liste s elementima 0 ili 1). Polinomi implementirani stringom su takvi da se koeficijent uz najveću potenciju nalazi na nalijevijoj poziciji u stringu; bit-liste kod množenja imaju obratnu logiku (tehnički detalj). Polinom bilo kojeg stupnja se generira kao slučajan niz znakova 0 i 1 (opet, ili kroz string ili listu). Pomoćne metode koje rade logički *AND*, *OR* ili *XOR* su implementirani obrascem *Strategija*. Postoje i pomoćne metode za operacije nad polinomima koje se nalaze u razredu *UtilPolynomial* dok se pomoćne metode za operacije nad matricom nalaze u razredu *UtilMatrix*.
Algoritme smo uspoređivali po kriteriju vremena potrebnog za izvršavanje operacije (zbrajanje, množenje, dijeljenje, faktorizacija) ili po broju ključnih operacija (predizračun, kvadriranje, množenje) koje se događaju unutar nekog algoritma (potenciranje).

7.2 Zbrajanje

Uzmemo li u obzir da je zbrajanje trivijalna operacija koja se bazira na prolasku po stringovima i činjenicu da se zbroj dva polinoma sa stupnjem reda veličine 10 000 računaju za 6 ms, ovdje nemamo što posebno ni pokazati. Recimo samo da ne možemo u običnoj for-petlji ni izračunati vjerodostojno rezultate. Primjerice, polinomi stupnja 100 i stupnja 1000 se oba izračunaju za po 2 ms. Testiranje se moglo obaviti za ulaze (stringove) do 10^7 znakova zbog limita nad memorijom. Polinome generiramo na slučajan način osiguravajući vodeću jedinicu (radi očekivanog stupnja). Ipak, spomenimo simbolično u tablici neke rezultate:

Stupanj polinoma	Vrijeme izvršavanja (ms)
10^2	2
10^3	2
10^4	6
10^5	15
10^6	72
10^7	754

7.3 Dijeljenje

Testirat ćemo tako da ćemo povećavati stupanj djeljenika 10 puta počevši od stupnja 100 i zaključno sa stupnjem 1 000 000. Djeljenik se generira na slučajan način, a za djelitelja ćemo uzeti fiksni polinom stupnja 10. Zanima nas vrijeme izračuna. Pogledajmo priloženu tablicu:

Stupanj polinoma	Vrijeme izvršavanja (s)
10^2	0.006
10^3	0.038
10^4	3.125
10^5	447.974
10^6	54 393.432

7.4 Faktorizacija

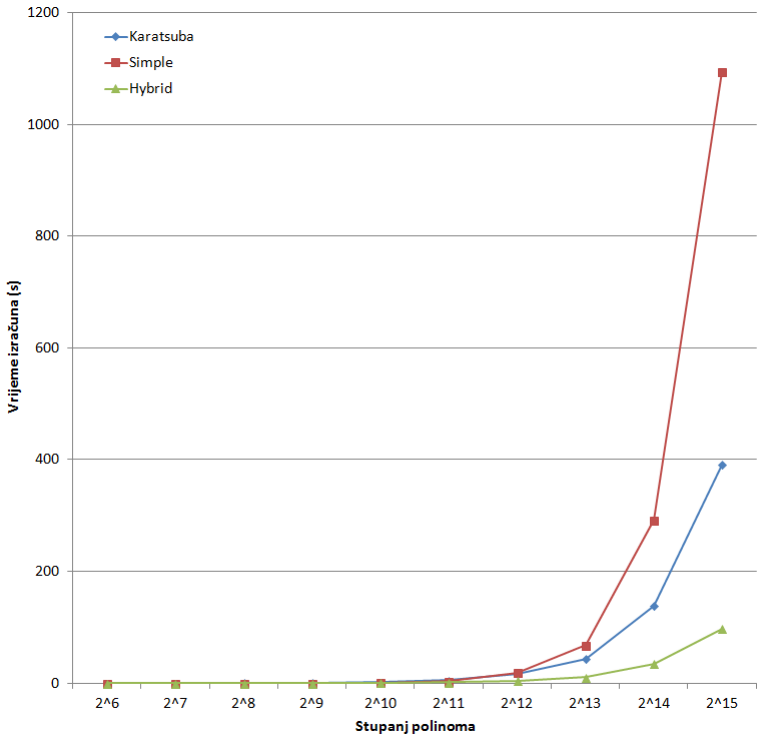
Faktorizaciju ćemo testirati tako da najprije generiramo sve polinome nekih odabranih stupnjeva, a kreću se od 5. do 20. Za svaku skupinu ćemo testirati vrijeme izvršavanja i to ponuditi kroz sljedeću tablicu:

Stupanj polinoma	Vrijeme izvršavanja (s)
5	0.02
8	0.057
10	0.211
12	0.92
15	14.249
17	101.2
20	1 094.181

7.5 Množenje

Testirat ćemo algoritme za jednostavno množenje (*Simple*), Karatsubino množenje (*Karatsuba*) i hibridno množenje (*Hybrid*). Testove vršimo nad polinomima s duljinom između 2^6 i 2^{15} bitova. Iako algoritam radi za bilo koje stupnjeve, izabrat ćemo najkompliciraniji slučaj prilikom testiranja: oba polinoma imaju maksimalne stupnjeve. Za svaki algoritam i svaku duljinu polinoma generira se slučajno 10 instanci i uzima se prosječno vrijeme koje potom prikazujemo u tablici.
Rezultati su u skladu s očekivanjem: Karatsubin algoritam nakon dovoljno velikih ulaznih polinoma prestigne obično množenje. Hibridni rezultati su još i najbolji iz razloga što se za manje stupnjeve brže množi običnim algoritmom nego Karatsubinim. Podrezivanje smo vršili za polinome stupnja manjeg od 60. Ćelije u tablici niže predstavljaju vrijeme u sekundama.

Stupanj polinoma	Simple	Karatsuba	Hybrid
2^6	0.002	0.010	0.002
2^7	0.010	0.036	0.007
2^8	0.052	0.142	0.034
2^9	0.233	0.451	0.115
2^{10}	1.133	1.650	0.401
2^{11}	4.397	5.117	1.329
2^{12}	19.048	16.268	4.091
2^{13}	68.617	43.560	10.579
2^{14}	292.851	138.232	34.246
2^{15}	1094.961	391.681	97.792



Slika 7.1: Grafički prikaz uspješnosti algoritama za množenje binarnih polinoma

Poglavlje 8

Dodatak – primjena binarnih polinoma

Recimo ponešto o primjeni binarnih polinoma u nekim granama.

8.1 Kriptografija

Krajem 60-tih i početkom 70-tih godina 20. stoljeća, razvojem financijskih transakcija, kriptografija postaje zanimljiva sve većem broju potencijalnih korisnika. Dotad je glavna primjena kriptografije bila u vojne i diplomatske svrhe, pa je bilo normalno da svaka država (ili čak svaka zainteresirana državna organizacija) koristi svoju šifru za koju je vjerovala da je najbolja. No, tada se pojavila potreba za šifrom koju će moći koristiti korisnici širom svijeta, i u koju će svi oni moći imati povjerenje. Dakle, pojavila se potreba uvođenja *standarda u kriptografiji*.

IBM-ov tim kriptografa je 1974. godine razvio algoritam je zasnovan na tzv. Feistelovoj[1] šifri, a jedna od glavnih ideja je alternirana uporaba supstitucija i transpozicija kroz više iteracija (tzv. rundi). Predloženi algoritam je nakon nekih preinaka, u kojima je sudjelovala i National Security Agency (NSA), prihvaćen kao standard 1976. godine i dobio je ime Data Encryption Standard (DES). Konačno razbijanje DES-a se dogodilo tek 1998. godine. EFF[2] je za $250 000 zaista napravila "DES Cracker", koji je razbijao poruke šifrirane DES-om za 56 sati i pojavila se potreba za boljim algoritmima. Godine 1997. National Institute of Standards and Technology (NIST) objavio je natječaj za kriptosustav koji bi trebao kao opće prihvaćeni standard zamijeniti DES,a pobjednik natječaja dobio bi ime Advanced Encryption Standard (AES). Konačno, 2000. godine za pobjednika natječaja proglašen je AES RIJNDAEL. Specifičnost navedenog algoritma leži

[1] Horst Feistel(1915. – 1990.) uveo je 1973.godine ideju koju koriste gotovo svi simetrični, blokovni algoritmi koji su danas u uporabi.

[2] Electronic Frontier Foundation (EEF) međunarodna je neprofitna organizacija posvećena obrani digitalnih prava utemeljena u Sjedinjenim Američkim Državama.

u činjenici da koristi operacije u polju $GF(2^8)$. Elementi polja su, ustvari, polinomi oblika $a_7x^7 + a_6x^6 + ... + a_1x + a_0$, $a_i \in \{0, 1\}$, a operacije su zbrajanje i množenje polinoma iz $\mathbb{Z}_2$ u varijabli x (oznaka: $\mathbb{Z}_2[x]$) modulo fiksni ireducibilni[3] polinom $g(x) = x^8 + x^4 + x^3 + x + 1$.

8.2 Teorija kodiranja

Prilikom prijenosa podataka na veće udaljenosti npr., uslijed šuma u komunikacijskom kanalu može doći do gubljenja informacija. Primimo li bit 0 umjesto bita 1, može se struktura podataka bitno narušiti. Nekad to može biti krivo presnimljena mp3 pjesma, dok drugi put mogu biti slike loše kvalitete koje stižu iz svemira. Ispravljanje nastalih pogrješaka postaje još važnije ukoliko se šalju enkriptirani podaci koji onda, ako su krivo poslani, mogu propagirati dosta štete. Teorija kodiranja je područje primijenjene matematike koja uključuje proučavanje i otkrivanje raznih kôdnih shema koje se koriste za povećanje broja grješaka koje se mogu ispraviti tijekom prijenosa podataka. Recimo da želimo poslati kôdnu riječ $c = 11111$, šum može oštetiti podatke i da u tom slučaju primimo $r = 01101$. U tom slučaju bismo rekli da je vektor pogrješke $e = 10010$ budući da vrijedi $c + e = r$. Općenito, neka je m poruka koju šaljemo: $m \rightarrow enkripcija \rightarrow c \rightarrow šum \rightarrow c + e = r \rightarrow dekripcija \rightarrow \widetilde{m}$. Naša želja je da vrijedi $m = \widetilde{m}$. Konkretna primjena se može pronaći u cikličkim kodovima koji koriste shift registre npr.

[3]Za polinom $p(x) \in K[x]$ stupnja n, gdje je K polje, ćemo reći da je ireducibilan u prstenu polinoma $K[x]$ ukoliko ne postoje polinomi $q(x), r(x) \in K[x]$ stupnja barem jedan, takvi da je $p(x) = q(x)r(x)$.

Bibliografija

[1] J. Arndt, *Matters Computational: Ideas, Algorithms, Source Code*, Springer-Verlag New York, 2010.

[2] E. R. Berlekamp, *Factoring polynomials over finite fields*, Bell System Technical Journal **46** (1967), 1853–1859, http://www.ams.org/journals/mcom/1970-24-111/S0025-5718-1970-0276200-X/.

[3] H. Cohen, *A course in Computational Algebraic Number Theory, Graduate Texts in Mathematics*, sv. 11, Springer-Verlag Berlin, 2000.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest i C.S. Stein, *Introduction to Algorithms*, 2nd edition izd., MIT Press and McGraw-Hill.

[5] A. Weimerskirch i C. Paar, *Generalizations of the Karatsuba Algorithm for Efficient Implementations*, (2006), 4–6, http://www.ei.rub.de/media/crypto/veroeffentlichungen/2010/08/08/kaweb.pdf.

[6] S. Singer, *Uvod u složenost algoritama*, https://web.math.pmf.unizg.hr/~singer/oaa/scans/pog_1.pdf.

Sažetak

U ovom radu smo definirali binarne polinome i opisali operacije nad njima. Objasnili smo ključne točke algoritama, dali primjere izvrijednjavanja i analizirali složenosti uz popratne komentare. Za množenje i potenciranje smo ponudili nekoliko verzija algoritama dok su dijeljenje i zbrajanje zbog, respektivno, okvira rada i jednostavnosti, obrađeni na prigodi suvisao način. Faktorizaciju smo, pak, detaljno razradili i u konačnici objasnili kako doći do generaliziranog pristupa u rastavu bilo kojeg polinoma.
Zaključili smo da je hibridni Karatsubin brži od Karatsubinog, a Karatsubin od naivnog množenja.
Testna vremena se slažu s teorijskim predviđanjima za algoritme množenja. Također je jasno postalo da se polinomi mogu posebno efikasno zbrajati i oduzimati zbog jednostavnosti algoritma, a dali smo rezultate i za ostale operacije. Vidljivo je da za složenije ulaze ne možemo tako brzo dobiti rezultate: dijeljenje u slučaju djelitelja sa stupnjem preko milijun; faktorizacija svih polinoma stupnja većeg od dvadeset.

Summary

In this thesis we presented binary polynomials through definition and operations on them. We explained key points of algorithms, gave examples of evaluation, pseudocodes of algorithms itself and analysed time complexity with comments in form suitable for appropriate thesis. We also offered few versions of algorithm for multiplication and exponentiation, and gave appropriate care and explanation with pseudocodes for division and addition. Factorization was described in detail: all necessary key steps for general approach in getting factors.
We concluded that hybrid Karatsuba has better time complexity than Karatsuba, and Karatsuba is, in Big-O notation, faster than simple multiplication algorithm.
Testing times are in accordance with theoretical predictions for multiplication algorithms. Also, addition and subtraction may be extremely fast because of simplicity of algorithm itself and structure of GF(2). We provided results for the rest of operations, too. It is obvious that we are not able to get results very fast when input size grows: division in case of divident's degree higher than one million and factorization of polynomials with degree higher than twenty.

Životopis

Rođen sam 18. rujna 1991. godine u Splitu. Nakon završene osnovne škole upisao sam gimnaziju Metković, prirodoslovno-matematički smjer.
Maturirao sam 2010. godine s odličnim uspjehom na Državnoj maturi uz dodatno položena 4 predmeta.
U jesen sam upisao preddiplomski sveučilišni studij Matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu. Preddiplomski studij završio sam 2013. godine kada sam upisao diplomski sveučilišni studij Računarstvo i matematika na istom fakultetu. Tijekom zadnje godine studija radio sam u zagrebačkoj kompaniji Ericsson Nikola Tesla d.d. u svojstvu student helpera. Trenutno sam zaposlen u Mrežnim tehnologijama VERSO d.o.o. na projektu Actility.

Zahvale

Najprije se zahvaljujem mentorici rada doc. dr. sc. Goranki Nogo na savjetima, brizi i pedantnosti prilikom izrade rada.
Zahvalio bih se i svim profesorima koji su mi omogućili napredak i znanje, ali me i oblikovali kao akademskog građanina i naučili strpljenju. Zahvale šaljem i doc. dr. sc. Marku Čupiću koji mi je usadio ljubav prema Javi i postao beskrajna inspiracija za ”štrikanje”. Hvala i svim kolegama; bivšim i sadašnjim djevojkama i prijateljima; svim ljudima koji su mi uljepšali ove prekrasne godine i bez kojih bi ovo bio jako siromašan period. A nije bio. Posebne zahvale šaljem svojoj obitelji koja je uvijek bila tu za mene i moj najveći oslonac u cijeloj priči. Hvala mama za svaku saslušanu jadikovku! Hvala i Marti, Anti, Davoru i Marinu - prijateljima za cijeli život. I za kraj se želim zahvaliti i mojoj Semi koja je uljepšala, a onda i olakšala sve nedaće koje su se u međuvremenu našle na putu.