Algoritmi za brzo množenje i dijeljenje velikih prirodnih brojeva

Briški, Bernard

Master's thesis / Diplomski rad

2017

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:034741

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**



Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb







SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Bernard Briški

ALGORITMI ZA BRZO MNOŽENJE I DIJELJENJE VELIKIH PRIRODNIH BROJEVA

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Saša Singer

Zagreb, rujan, 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. _____

Sadržaj

Sadržaj **iii**

Uvod **1**

1 Pozicijski zapis prirodnih brojeva **3**

2 Klasično množenje **5**
2.1 Klasični algoritam množenja 5
2.2 Aritmetička složenost klasičnog algoritma 15

3 "Podijeli pa vladaj" pristup množenju brojeva **17**
3.1 Karatsubin algoritam 17
3.2 Generalizirani Karatsubin algoritam 20
3.3 Usporedba Karatsubinog algoritma i klasičnog množenja 24

4 Fourierove transformacije u množenju brojeva **27**
4.1 Diskretna Fourierova transformacija i inverzna transformacija 27
4.2 Schönhage–Strassenov algoritam 31

5 Dijeljenje velikih brojeva **35**
5.1 Traženje aproksimacije recipročne vrijednosti 35
5.2 Složenost algoritma NREC 40

Bibliografija **43**

Uvod

Brza realizacija osnovnih aritmetičkih operacija za velike prirodne brojeve (neprikazive u računalu) ima velike primjene u računarskoj algebri i kriptografiji. Veliki brojevi se obično prikazuju nizom znamenki u nekoj izabranoj bazi, tako da se osnovne operacije na znamenkama mogu egzaktno i brzo izvesti u aritmetici računala. Algoritmi za zbrajanje i oduzimanje velikih brojeva su jednostavni i oponašaju algoritme za "ručno" računanje. Njihova složenost je linearna u duljini brojeva i optimalna. Za razliku od toga, slični algoritmi za množenje i cjelobrojno dijeljenje s ostatkom imaju kvadratnu složenost i nisu optimalni, jer postoje i mnogo brži algoritmi (poput Karatsubinog algoritma za množenje).

U radu najprije detaljno opisujemo klasični algoritam za množenje i njegovu složenost. Potom se detaljno opisuju i analiziraju algoritmi za brzo množenje velikih brojeva – strategijom "podijeli-pa-vladaj" na 2 i više dijelova, uz korištenje polinomne interpolacije i evaluacije. Također, opisan je i analiziran danas najbrži poznati Strassen–Schönhageov algoritam, koji koristi brzu Fourierovu transformaciju za množenje polinoma. Na kraju, opisan je i opći algoritam za brzo dijeljenje, preko brzog množenja, baziran na Newtonovoj metodi s progresivnim povećanjem točnosti.

Poglavlje 1

Pozicijski zapis prirodnih brojeva

U pozicijskom zapisu broja, broj opisujemo nizom njegovih znamenki u odabranoj bazi b. Dekadski zapis brojeva, koji koristimo u svakodnevnici, je upravo takav način prikaza brojeva, u bazi $b = 10$. Sljedeći jednostavan teorem nam daje osnovu za takav prikaz. Dokaz možete naći u [4].

Teorem 1.0.1. *Neka je $b \in \mathbb{N}$, $b > 2$ bilo koji prirodni broj. Za svaki prirodni broj $u \in \mathbb{N}$, postoje broj $n \in \mathbb{N}_0 = \mathbb{N} \cup \{0\}$, i brojevi $u_0, ..., u_n$ takvi da je*

$$u = \sum_{i=0}^{n} u_i b^i. \tag{1.0.1}$$

Brojevi $u_0, \ldots, u_n$ mogu biti i kompleksni. Uz dodatno ograničenje

$$u_i \in \mathbb{N}_0, \quad i = 0, \ldots, n-1,$$

*i takozvanu **normalizaciju***

$$0 \leq u_i < b, \quad i = 0, \ldots, n-1, \qquad 0 < u_n < b, \tag{1.0.2}$$

prikaz (1.0.1) *je jedinstven.*

Definicija 1.0.2. *Broj b zovemo **baza**, a n je **najviša potencija baze** ili **stupanj broja** u **u bazi** b, u oznaci*

$$\deg_b(u) = n.$$

*Brojevi $u_0, \ldots, u_n$ su **znamenke broja** u **u bazi** b, a znamenku u_n zovemo **vodeća** ili **najznačajnija** znamenka broja u.*

Ako su znamenke normalizirane, tj. vrijedi (1.0.2)*, onda relaciju* (1.0.1) *zovemo **pozicijski zapis broja** u **u bazi** b, u oznaci*

$$u = (u_n \ldots u_0)_b, \tag{1.0.3}$$

ili

$$u = u_n \dots u_0, \tag{1.0.4}$$

ako je iz konteksta jasno u kojoj bazi b se vrši prikaz.

Primijetite da je zapis (1.0.4) uobičajeni zapis koji svakodnevno koristimo za prikaz broja s bazom $b = 10$.

Definicija 1.0.3. ***Duljina broja*** *u* ***u bazi*** *b, u oznaci* $\ell_b(u)$*, je broj znamenki u pozicijskom zapisu broja u u bazi b*

$$\ell_b(u) = \deg_b(u) + 1 = \lfloor \log_b u + 1 \rfloor. \tag{1.0.5}$$

Reprezentaciju brojeva $u \in \mathbb{N}_0$, nizom znamenki u nekoj bazi b, lako je programski realizirati, uz uvjet da su sve znamenke u_i uvijek prikazive u računalu. Od osnovnih tipova podataka prikazivih u računalu, najzgodnije je znamenke prikazati **cijelim brojevima**, jer to omogućava da cjelobrojnu aritmetiku računala iskoristimo za izvođenje aritmetičkih operacija na znamenkama.

Za aritmetičke algoritme, duljina ulaznih brojeva je mjera veličine zadaće. Zbog toga se složenost aritmetičkih algoritama izražava kao funkcija duljine ulaznih brojeva. Iz (1.0.5) slijedi da duljina broja ovisi o bazi, pa prema tome i složenost ovisi o bazi. Da bi se analizirala ovisnost, potrebno je pronaći vezu između duljina brojeva u različitim bazama. To ističemo sljedećim teoremom, za kojeg dokaz možete naći u [4].

Teorem 1.0.4. *Duljine prirodnih brojeva u bilo koje dvije baze su kodominantne. Štoviše, ako su* $b_1, b_2 \geq 2$*, bilo koje dvije baze, onda je*

$$\ell_{b_2}(u) \sim \log_{b_2} b_1 \cdot \ell_{b_1}(u), \tag{1.0.6}$$

tj. duljine brojeva su asimptotski proporcionalne za velike brojeve.

Važna posljedica ovog teorema je da **složenost aritmetičkih algoritama ne ovisi bitno o bazi pozicijskog prikaza**.

Poglavlje 2

Klasično množenje

2.1 Klasični algoritam množenja

U ovom potpoglavlju se bavimo analizom klasičnog algoritma množenja. Prvo ćemo analizirati algoritam 2.1.1 množenja broja u pozicijskom zapisu (u odabranoj bazi) jednom znamenkom. Ovaj algoritam je sam za sebe koristan, a poslužit će i za razvoj općeg algoritma za množenje prirodnih brojeva.

Propozicija 2.1.1. *Ako je $u \in \mathbb{N}_0$ broj dan pozicijskim zapisom u bazi b,*

$$u = (u_n \dots u_0)_b$$

i ako je $v_0 \in \mathbb{N}_0$ i $0 \leq v_0 \leq b$, onda algoritam NMULD *daje niz znamenki pozicijskog zapisa broja $u \cdot v_0$ u bazi b,*

$$u \cdot v_0 = w = (w_k \dots w_0)_b,$$

gdje je $k = \deg_b(w) \in \{n, n+1\}$ za $v_0 > 0$ i $k = -1$, za $v_0 = 0$.

Dokaz. Pretpostavimo da je $v_0 > 0$. Ako je $v_0 = 0$, algoritam će postaviti vrijednost $\deg(w) = -1$, što odgovara $w = 0$.

Rad algoritma opisan je rekurzivnim relacijama za svaki $i = 0, \dots, n$:

$$\begin{aligned} carry_{-1} &= 0, \\ w_i &= (u_i \cdot v_0 + carry_{i-1}) \bmod b, \\ carry_i &= \lfloor (u_i \cdot v_0 + carry_{i-1})/b \rfloor. \end{aligned} \tag{2.1.1}$$

Zbog

$$carry_i \cdot b + w_i = u_i \cdot v_0 + carry_{i-1}, \quad i = 0, \dots, n, \tag{2.1.2}$$

Algoritam 2.1.1: NMULD – množenje prirodnog broja znamenkom

Podaci: niz znamenki $(u_n, \ldots, u_0)$ broja u i znamenka v_0 u odabranoj bazi b.
Rezultat: niz znamenki $(w_k, \ldots, w_0)$ broja $w = u \cdot v_0$ u bazi b.

```
1  početak
2      ako v_0 > 0 onda
3          carry ← 0;
4          za i ← 0 do deg(u) čini
5              temp ← u_i · v_0 + carry;
6              w_i ← temp mod b;
7              carry ← temp div b;
8          kraj
9          ako carry > 0 onda
10             deg(w) ← deg(u) + 1;
11             w_deg(w) ← carry;
12         inače
13             deg(w) ← deg(u);
14         kraj
15     inače
16         deg(w) ← −1;
17     kraj
18 kraj
```

indukcijom dobivamo:

$$carry_i \cdot b^{i+1} + \sum_{j=0}^{i} w_j b^j = v_0 \cdot \sum_{j=0}^{i} u_j b^j, \tag{2.1.3}$$

za $i = 0, \ldots, n$. To znači da algoritam u svakom koraku nalazi korektan produkt donjeg dijela broja u i znamenke v_0. To možemo pisati u obliku

$$carry_n \cdot b^{n+1} + w \bmod b^{n+1} = v_0(u \bmod b^{i+1}),$$

za $i = 0, \ldots, n$. Za $i = n$ izlazi

$$carry_n \cdot b^{n+1} + w \bmod b^{n+1} = v_0 \cdot u. \tag{2.1.4}$$

Tvrdimo da za svaki $i = 0, \ldots, n$ vrijedi

$$0 \le u_i \cdot v_0 + carry_{i-1} \le b^2 - b - 1, \qquad 0 \le carry_i \le b - 2. \tag{2.1.5}$$

Tvrdnju (2.1.5) dokazujemo indukcijom, koristeći $0 \leq u_i \leq b-1$ i $0 < v_0 \leq b-1$. Tada, zbog $carry_{-1} = 0$, imamo

$$0 \leq u_i \cdot v_0 \leq (b-1)^2 = b^2 - 2b + 1,$$

pa za $i = 1$ vrijedi prva relacija iz (2.1.5).

Iz (2.1.1) dobivamo

$$0 \leq carry_0 \leq \left\lfloor \frac{(b-1)^2}{b} \right\rfloor = \left\lfloor b - 2 + \frac{1}{b} \right\rfloor.$$

Zbog $b \in \mathbb{N}, b \geq 2$, vrijedi

$$\left\lfloor b - 2 + \frac{1}{b} \right\rfloor = b - 2,$$

čime je dokazano (2.1.5) za $i = 0$.

Iz pretpostavke $0 \leq carry_i \leq b-2$, dobivamo

$$0 \leq u_i \cdot v_0 + carry_{i-1} \leq (b-1)^2 + b - 2 = b^2 - b - 1,$$

a (2.1.1) daje

$$0 \leq carry_i \leq \left\lfloor b - 1 - \frac{1}{b} \right\rfloor = b - 2,$$

što dokazuje (2.1.5).

Ako je $carry_n = 0$, algoritam postavlja $\deg(w) = \deg(u)$, pa (2.1.4) daje

$$w = v_0 \cdot u.$$

Zbog $v_0 > 0$ i $u_n > 0$, iz (2.1.2) proizlazi

$$w_n = u_n \cdot v_0 + carry_{n-1} \geq u_n > 0,$$

pa je $w = (w_n \dots w_0)_b$ normalizirani prikaz broja $w = v_0 \cdot u$.

Ako je $carry_n > 0$, algoritam postavlja $\deg(w) = \deg(u) + 1$ i $w_{n+1} = carry_n$, pa iz (2.1.4) slijedi da je

$$w = (w_{n+1} \dots w_0)_b$$

normalizirani prikaz broja $w = v_0 \cdot u$. □

Ako s *maxint* označimo najveći prikaziv broj u računalu, u nekom izabranom tipu cijelih brojeva kojeg podržava arhitektura računala, tada zahtjev prikazivosti broja u znači

$$u_i \leq maxint, \quad i = 0, \dots, n.$$

Iz (1.0.2) izlazi prvi uvjet na dopuštenu veličinu baze

$$b \leq maxint + 1.$$

Prva relacija u (2.1.5) daje zahtjev

$$b^2 - b - 1 \leq maxint, \tag{2.1.6}$$

potreban da algoritam radi korektno i u aritmetici računala, a ne samo u teoriji.

Razmotrimo složenost ovog algoritma. Prostorna složenost je

$$\mathrm{Compl}_s(\mathrm{NMULD}) = 2\ell(u) + c,$$

s $c \leq 6$, što se može i smanjiti, ako u algoritmu broj w spremamo na mjesto broja u, a ne posebno.

Operacije div i mod brojimo svaku posebno, budući da se u višim programskim jezicima te operacije vrše odvojeno i podjednako traju, a arhitektura računala odmah daje kvocijent i ostatak. Najbolja, najgora i prosječna aritmetička složenost ovog algoritma su jednake, budući da algoritam uvijek obavlja isti broj operacija. Dakle, aritmetička složenost algoritma NMULD je

$$\mathrm{Compl}_A(\mathrm{NMULD}) = 4\ell(u) = \begin{cases} \ell(u) & \text{zbrajanja,} \\ \ell(u) & \text{množenja,} \\ 2\ell(u) & \text{dijeljenja.} \end{cases}$$

Vremenska složenost, također, linearno ovisi o duljini broja u, pa vrijedi

$$\mathrm{Compl}_T(\mathrm{NMULD}) \sim \ell(u) \sim \ell(w).$$

Sljedeći specijalan slučaj množenja je množenje potencijom baze, opisano u algoritmu 2.1.2.

Tvrdimo da je algoritam korektan, jer se množenje svodi na pomak znamenki. Njegova prostorna složenost je

$$\mathrm{Compl}_S(\mathrm{NMULB}) = 2\ell(u) + p + c,$$

gdje je $c \leq 4$. Faktor 2 se može eliminirati u slučaju da broj w spremamo na mjesto broja u. Aritmetička složenost je konstantna zbog zbrajanja stupnjeva

$$\mathrm{Compl}_A(\mathrm{NMULB}) = 1.$$

Vremenska složenost ovisna je o izboru strukture podataka za prikazivanje brojeva u računalu. Ako bi broj bio prikazan vezanom listom znamenki, izbjegli bismo pomicanje (prva petlja u algoritmu). U najgorem slučaju, vremenska složenost je

$$\mathrm{Compl}_T(\mathrm{NMULB}) \sim \ell(w) = \ell(u) + p.$$

Algoritam 2.1.2: NMULB – množenje potencijom baze

Podaci: niz znamenki $(u_n, \ldots, u_0)$ broja u i potencija $p \in \mathbb{N}_0$ u odabranoj bazi b.
Rezultat: niz znamenki $(w_k, \ldots, w_0)$ broja $w = u \cdot b^p$ u bazi b.

```
1  početak
2      ako deg(u) ≥ 0 onda
3          deg(w) ← deg(u) + p;
4          za i ← deg(u) do 0 čini
5              w_{i+p} ← u_i;
6          kraj
7          za i ← 0 do p − 1 čini
8              w_i ← 0;
9          kraj
10     inače
11         deg(w) ← −1;
12     kraj
13 kraj
```

Algoritam NMULB se, inače, rijetko koristi kao zaseban algoritam. Najčešće je taj algoritam dio nekog drugog algoritma.

Algoritmi NMULD i NMULB, zajedno sa zbrajanjem, omogućavaju direktnu realizaciju klasičnog algoritma za množenje prirodnih brojeva (algoritam 2.1.3). Naime,

$$w = u \cdot v = \sum_{i=0}^{n} u_i b^i \cdot \sum_{j=0}^{m} v_j b^j$$

možemo zapisati kao

$$w = \sum_{j=0}^{m} (u \cdot v_j) \cdot b^j \tag{2.1.7}$$

i primijeniti algoritam sličan Hornerovoj shemi, što odgovara "ručnom" množenju.

Ovdje se svaka operacija obavlja na nizovima znamenki odgovarajućih brojeva, stoga je potrebno svaku operaciju zamijeniti pozivom odgovarajućeg algoritma. Umnožak $w \cdot b$ treba zamijeniti pozivom algoritma NMULB$(w, 1, t_1)$, pri čemu t_1 označava izlaz algoritma NMULB. Umnožak $u \cdot v_j$ treba zamijeniti pozivom NMULD(u, v_j, t_2), pri čemu je t_2 rezultat izvršenja algoritma NMULD. Na kraju, treba pozvati još algoritam operacije zbrajanja (algoritam NADD, može se naći u [4]) na t_1 i t_2.

Algoritam NMUL0 ima svoje mane. Pomoćni brojevi t_1 i t_2 su dodatan trošak memorije. Također, ne koristimo neka svojstva koja bi omogućila smanjenje broja aritmetičkih

Algoritam 2.1.3: NMUL0 – množenje prirodnih brojeva

Podaci: nizovi znamenki $(u_n, \ldots, u_0)$, $(v_m, \ldots, v_0)$ brojeva u i v u odabranoj bazi b.
Rezultat: niz znamenki $(w_k, \ldots, w_0)$ broja $w = u \cdot v$ u bazi b, pri čemu je $k = \deg_b(w)$.

```
1 početak
2 |  w ← 0;
3 |  za i ← deg(v) do 0 čini
4 |  |  w ← w · b + u · v_j;
5 |  kraj
6 kraj
```

operacija. Tako je moguće, u pozivu algoritma za zbrajanje, iskoristiti činjenicu da je najniža znamenka broja t_1 jednaka nuli.

Uzimajući u obzir mane algoritma NMUL0, izlažemo efikasniji algoritam NMUL (algoritam 2.1.4).

Propozicija 2.1.2. *Ako su brojevi $u, v \in \mathbb{N}_0$ dani pozicijskim zapisima u bazi b,*

$$u = (u_n \ldots u_0)_b, \quad v = (v_m \ldots v_0)_b,$$

onda algoritam NMUL *daje niz znamenki pozicijskog zapisa broja $u \cdot v$,*

$$u \cdot v = w = (w_k \ldots w_0)_b,$$

pri čemu je

$$k = \deg_b(w) \in \{n + m, n + m + 1\}.$$

Dokaz. Ako je $u = 0$ ili $v = 0$, algoritam postavlja $\deg(w) = -1$, odnosno $w = 0$, pa zaključujemo da u tim slučajevima algoritam radi korektno. Pretpostavimo da su $u, v > 0$. Označimo s $w_{(-1)}$ polazno stanje broja (niza) w u algoritmu, a s $w_{(j)}$, $j = 0, \ldots, m$, stanje broja w nakon izvršenja vanjske petlje algoritma s brojačem j.

Znamenke broja $w_{(j)}$ označavamo s

$$w_{(j)} = (w_{n+j+1,j}, \ldots, w_{0,j}),$$

odnosno

$$w_{(j)} = \sum_{i=0}^{n+j+1} w_{i,j} b^i, \quad j = -1, 0, \ldots, m, \tag{2.1.8}$$

Algoritam 2.1.4: NMUL – množenje prirodnih brojeva

Podaci: nizovi znamenki $(u_n, \ldots, u_0)$, $(v_m, \ldots, v_0)$ brojeva u i v u odabranoj bazi b
Rezultat: niz znamenki $(w_k, \ldots, w_0)$ broja $w = u \cdot v$ u bazi b, pri čemu $k = \deg_b(w)$

```
1  početak
2      ako deg(u) ≥ 0  &  deg(w) ≥ 0 onda
3          deg(w) ← deg(u) + deg(v);
4          za i ← 0 do deg(u) čini
5              w_i ← 0;
6          kraj
7          za j ← 0 do deg(v) čini
8              carry ← 0;
9              za i ← 0 do deg(u) čini
10                 temp ← w_{i+j} + u_i · v_j + carry;
11                 w_{i+j} ← temp mod b;
12                 carry ← temp div b;
13             kraj
14             w_{deg(u)+j+1} ← carry;
15         kraj
16         ako carry > 0 onda
17             deg(w) ← deg(u) + deg(v) + 1;
18         inače
19             deg(w) ← deg(u) + deg(v);
20         kraj
21     inače
22         deg(w) ← −1;
23     kraj
24 kraj
```

pri čemu ovaj zapis ne mora biti normaliziran. Algoritam na početku postavlja

$$w_{i,-1} = 0, \quad i = 0, \ldots, n, \tag{2.1.9}$$

odnosno,

$$w_{(-1)} = 0.$$

Označimo vrijednosti varijable *carry* tijekom prolaza kroz vanjsku petlju s indeksom j, na sljedeći način: početna vrijednost je $carry_{j-1,j} = 0$, a vrijednost nakon prolaza kroz unutarnju petlju s indeksom i je $carry_{i+j,j}$, za svaki $i = 0, \ldots, n$. Za $j = 0$, rad algoritma

možemo opisati rekurzivnim relacijama za $i = 0, \ldots, n$

$$\begin{aligned} carry_{-1,0} &= 0 \\ w_{i,0} &= (u_i \cdot v_0 + carry_{i-1,0}) \bmod b \\ carry_{i,0} &= \lfloor (u_i \cdot v_0 + carry_{i-1,0})/b \rfloor \\ w_{n+1,0} &= carry_{n,0}, \end{aligned} \tag{2.1.10}$$

s tim da smo iskoristili relaciju (2.1.9). Usporedbom s (2.1.1), prema propoziciji 2.1.1, slijedi

$$w_{(0)} = u \cdot v_0. \tag{2.1.11}$$

Za $j \geq 1$, rad algoritma je opisan rekurzivnim relacijama

$$w_{i,j} = w_{i,j-1}, \quad i = 0, \ldots, j-1, \tag{2.1.12}$$

jer algoritam ne mijenja ta mjesta, te

$$\begin{aligned} carry_{j-1,j} &= 0 \\ w_{i+j,j} &= (w_{i+j,j-1} + u_i \cdot v_j + carry_{i+j-1,j}) \bmod b \\ carry_{i+j,j} &= \lfloor (w_{i+j,j-1} + u_i \cdot v_j + carry_{i+j-1,j})/b \rfloor, \quad \text{za } i = 0, \ldots, n, \end{aligned} \tag{2.1.13}$$

i prvi put se postavlja znamenka

$$w_{n+j+1,j} = carry_{n+j,j}. \tag{2.1.14}$$

Iz (2.1.13), za $i = 0, \ldots, n$, vrijedi

$$carry_{i+j,j} \cdot b + w_{i+j,j} = w_{i+j,j-1} + u_i \cdot v_j + carry_{i+j-1,i}. \tag{2.1.15}$$

Odavde indukcijom slijedi

$$carry_{i+j,j} \cdot b^{i+1} + \sum_{l=0}^{i} w_{l+j,j} b^l = \sum_{l=0}^{i} w_{l+j,j-1} b^l + v_j \cdot \sum_{l=0}^{i} u_l b^l, \quad \text{za } i = 0, \ldots, n.$$

Množenjem ove relacije s b^j dobivamo

$$carry_{i+j,j} \cdot b^{i+j+1} + \sum_{l=j}^{i+j} w_{l,j} b^l = \sum_{l=j}^{i+j} w_{l,j-1} b^l + (v_j \cdot \sum_{l=0}^{i} u_l b^l) b^j, \quad \text{za } i = 0, \ldots, n. \tag{2.1.16}$$

Iz relacije (2.1.12) slijedi

$$\sum_{l=0}^{j-1} w_{l,j} b^l = \sum_{l=0}^{j-1} w_{l,j-1} b^l,$$

odnosno, $w_{(j)} \bmod b^j = w_{(j-1)} \bmod b^j$. Zbrajanjem ove relacije i relacije (2.1.16) izlazi

$$carry_{i+j,j} \cdot b^{i+j+1} + \sum_{l=0}^{i+j} w_{l,j} b^l = \sum_{l=0}^{i+j} w_{l,j-1} b^l + (v_j \cdot \sum_{l=0}^{i} u_l b^l) b^j, \qquad \text{za } i = 0, \ldots, n.$$

Odavde, za $i = n$, korištenjem relacije (2.1.8) dobivamo

$$carry_{n+j,j} \cdot b^{n+j+1} + w_{(j)} \bmod b^{n+j} = w_{(j-1)} + (v_j \cdot u) b^j.$$

Iz (2.1.14) i (2.1.8) izlazi

$$w_{(j)} = w_{(j-1)} + (v_j \cdot u) b^j, \quad j = 1, \ldots, m. \tag{2.1.17}$$

Ako za bazu indukcije uzmemo relaciju (2.1.11), a za korak indukcije uzmemo relaciju (2.1.17), tada dobivamo

$$w_{(m)} = \sum_{j=0}^{m} (v_j \cdot u) b^j,$$

ili

$$w_{(m)} = u \cdot v.$$

Očito, iz (2.1.11) i (2.1.13), za sve znamenke $w_{i,j}$ vrijedi

$$0 \leq w_{i,j} < b, \tag{2.1.18}$$

za $j = 0, \ldots, m$ i za $i = 0, \ldots, n + j + 1$. Na samom kraju, algoritam provjerava zadnju vrijednost varijable *carry*, odnosno, vrijednost

$$w_{n+m+1,m} = carry_{n+m,m}.$$

Ako je ta vrijednost pozitivna, algoritam postavlja

$$\deg(w) = n + m + 1$$

i daje

$$w = w_{(m)},$$

što je korektni normalizirani rezultat. Ako je $w_{n+m+1,m} = 0$, algoritam daje

$$\deg(w) = n + m,$$

tj. postavlja

$$w = w_{(m)} \bmod b^{n+m} = w_{(m)}.$$

Potrebno je još dokazati da je tada

$$w_{\deg(w)} = w_{n+m,m} > 0.$$

Koristeći relacije (2.1.5) za bazu indukcije ($j = 0$) i normaliziranost brojeva u, v, indukcijom prvo po i, zatim po j, izlazi u (2.1.13)

$$0 \leq w_{i+j,j} + u_i \cdot v_j + carry_{i+j-1,j} \leq b^2 - 1, \quad 0 \leq carry_{i+j,j} \leq b - 1, \tag{2.1.19}$$

za sve $j = 0, \ldots, m$ i $i = 0, \ldots, n$. Za $i = n$, $j = m$, ako je $carry_{n+m,m} = 0$, onda iz (2.1.15) slijedi

$$w_{n+m,m} = w_{n+m,m-1} + u_n \cdot v_m + carry_{n+m-1,m}.$$

Iz relacija (2.1.18) i (2.1.19), koristeći $u_n > 0$, $v_m > 0$ (zbog pretpostavke da su $u, v > 0$), dobivamo

$$w_{n+m,m} \geq u_n \cdot v_m > 0,$$

čime je tvrdnja u potpunosti dokazana. □

Sada ćemo dati najjači zahtjev na veličinu baze. Taj zahtjev će biti i konačan, budući da ćemo kasnije pokazati da je i dijeljenje korektno u aritmetici računala.

Propozicija 2.1.3. *Algoritam* NMUL *radi korektno u aritmetici računala ako i samo ako baza b zadovoljava uvjet*

$$b^2 - 1 \leq maxint. \tag{2.1.20}$$

Dokaz. Relacija (2.1.19) pokazuje da je $w_{i+j} + u_i \cdot v_j + carry$ najveća vrijednost koju algoritam računa. Algoritam radi korektno u aritmetici računala ako i samo ako je ta vrijednost uvijek egzaktno prikaziva, odnosno, broj $b^2 - 1$ mora biti egzaktno prikaziv. □

Za prostornu složenost je očito

$$\mathrm{Compl}_S(\mathrm{NMUL}) = 2(\ell(u) + \ell(v)) + c,$$

s $c \leq 7$, ako računamo i mogućnost da je $\ell(w) = \ell(u) + \ell(v)$, te prostor za duljine brojeva i pomoćne varijable. Aritmetička složenost algoritma je

$$\mathrm{Compl}_A(\mathrm{NMUL}) = 5\ell(u) \cdot \ell(v) + 2 = \begin{cases} 2\ell(u) \cdot \ell(v) + 2 & \text{zbrajanja,} \\ \ell(u) \cdot \ell(v) & \text{množenja,} \\ 2\ell(u) \cdot \ell(v) & \text{dijeljenja.} \end{cases} \tag{2.1.21}$$

Vremenska složenost algoritma je

$$\mathrm{Compl}_T(\mathrm{NMUL}) \sim \ell(u) \cdot \ell(v). \tag{2.1.22}$$

U aritmetičkoj složenosti možemo uštedjeti $\ell(u)$ zbrajanja ako, umjesto početne inicijalizacije na nulu, stavimo

$$(w_{n+1} \ldots w_0) = u \cdot v_0,$$

koristeći algoritam NMULD.

2.2 Aritmetička složenost klasičnog algoritma

Zanima nas broj aritmetičkih operacija potrebnih za množenje 2 prirodna broja u pozicijskom zapisu u bazi b. U sljedećoj definiciji ćemo uvesti oznaku za taj broj kao funkciju duljine (broja znamenki) brojeva koje množimo.

Definicija 2.2.1. *Neka je* MUL *bilo koji opći algoritam za množenje prirodnih brojeva* $u, v \in \mathbb{N}$ *u pozicijskom zapisu u nekoj (može i fiksnoj) bazi* b. ***Aritmetičku složenost algoritma****, u ovisnosti o duljini brojeva* u *i* v*, označavamo s*

$$\mathrm{Mul}(n, m) = \mathrm{Compl}_A(MUL),$$

ako je $l(u) = n$ *i* $l(v) = m$*. Ako je* $n = m$*, onda skraćeno označavamo*

$$\mathrm{Mul}(n) = \mathrm{Mul}(n, n).$$

U gornjoj definiciji (a i općenito) smatramo da je algoritam **opći**, ako radi korektno za **sve** brojeve $u, v \in \mathbb{N}$, odnosno, za sve proizvoljne duljine brojeva. Zahtjev da je MUL opći algoritam za množenje prirodnih brojeva je bitan, jer bi u slučaju fiksne duljine brojeva, problem bio trivijalan. Naime, tada se može konstruirati "tablica množenja" za danu duljinu, pa se množenje tada svodi na jednostavno čitanje rezultata iz tablice. U definiciji smo dozvolili fiksiranje baze, jer u slučaju da konstruiramo tablicu množenja u bazi b, duljina brojeva je i dalje proizvoljna. Stoga je trajanje množenja ovisno o duljini brojeva, bar kao broj traženja u tablici. Uočimo još da je $\mathrm{Mul}(n, m)$ (odnosno, $\mathrm{Mul}(n)$) dobra mjera potrebnog vremena na sekvencijalnim arhitekturama.

Uz oznake u gornjoj definiciji, relacije (2.1.21) i (2.1.22) za algoritam NMUL, možemo zapisati u obliku

$$\mathrm{Mul}(n, m) = 5nm + 2, \quad \mathrm{Mul}(n) = 5n^2 + 2, \tag{2.2.1}$$

tj. broj operacija za množenje n-znamenkastih prirodnih brojeva je proporcionalan s kvadratom duljine brojeva.

Očita je činjenica da za bilo koji opći algoritam za množenje n-znamenkastih prirodnih brojeva vrijedi

$$\mathrm{Mul}(n) = \Omega(n), \tag{2.2.2}$$

jer ulaz sadrži $2n$, općenito, nezavisnih znamenki, na osnovu kojih treba postaviti $2n$ znamenki rezultata.

Prethodna dva rezultata pokazuju da je $\mathrm{Mul}(n)$ uvijek "između" n i n^2, barem asimptotski. Dakle, vidimo da ima prostora za bitno ubrzanje klasičnog algoritma.

Poglavlje 3

"Podijeli pa vladaj" pristup množenju brojeva

3.1 Karatsubin algoritam

U prošlom poglavlju smo zaključili da klasični algoritam ima prostora za ubrzavanje. Klasični algoritam ćemo ubrzati rekurzivnom metodom "podijeli pa vladaj", kojom ćemo zadaću veličine n svesti na neki niz istovrsnih zadaća manjih veličina. Postupak primjenjujemo rekurzivno sve dok problem ne postane lako rješiv, odnosno, dok veličina zadaće ne bude "dovoljno" mala.

Pretpostavimo da je $\ell(u) = \ell(v) = n$. Neka je n paran broj, odnosno

$$n = 2k.$$

Ove pretpostavke nam omogućuju da brojeve

$$u = \sum_{i=0}^{n} u_i b^i, \quad v = \sum_{i=0}^{n} v_i b^i, \tag{3.1.1}$$

možemo zapisati u bazi $b^k = b^{n/2}$, u obliku

$$u = U_1 b^k + U_0, \quad v = V_1 b^k + V_0, \tag{3.1.2}$$

pri čemu U_1, V_1 predstavljaju značajnije (gornje) polovice brojeva u, v, dok U_0, V_0 predstavljaju donje polovice tih brojeva. Ti brojevi su definirani s

$$U_1 = (u_{2k-1} \dots u_k)_b = \sum_{i=0}^{k-1} u_{i+k} b^i, \qquad U_0 = (u_{k-1} \dots u_0)_b = \sum_{i=0}^{k} u_i b^i,$$

$$V_1 = (v_{2k-1} \dots v_k)_b = \sum_{i=0}^{k-1} v_{i+k} b^i, \qquad V_0 = (v_{k-1} \dots v_0)_b = \sum_{i=0}^{k} v_i b^i.$$

Brojevi U_1, U_0, V_1, V_0 su, očito, znamenke brojeva u i v u bazi b^k. Ako je relacija (3.1.1) normalizirana u bazi b, tada je i relacija (3.1.2) normalizirana u bazi b^k.

Uz gornje oznake, produkt $w = u \cdot v$, promatran u bazi b^k, ima oblik

$$w = (U_1 \cdot V_1)b^{2k} + (U_0 \cdot V_1 + U_1 \cdot V_0)b^k + U_0 \cdot V_0. \tag{3.1.3}$$

Ovom relacijom problem produkta n-znameknastih brojeva u i v sveden je na problem nalaženja 4 produkta $n/2$-znamenkastih brojeva $(U_1 \cdot V_1,\ U_0 \cdot V_1,\ U_1 \cdot V_0,\ U_0 \cdot V_0)$, uz zbrajanja i množenja potencijom baze (pomake). Dodajmo još da ovaj zapis ne mora nužno biti normaliziran u bazi b^k. Navedene operacije i normalizaciju treba provesti na nizovima znamenki u bazi b.

Slijedi jedan pomoćni rezultat koji će nam trebati u daljnjim razmatranjima. Dokaz možete naći u [4].

Lema 3.1.1. *Neka je $t \in \mathbb{R}$ neka konstatna. Za bilo koji algoritam množenja prirodnih brojeva u pozicijskom zapisu, postoji konstanta $C \in \mathbb{N}$, takva da za svaki $n \in \mathbb{N}$ vrijedi*

$$\mathrm{Mul}(n + t) \leq \mathrm{Mul}(n) + Cn, \tag{3.1.4}$$

pri čemu C ovisi o t, ali ne i o n.

Koristeći prethodnu lemu, može se pokazati da relacija (3.1.3) ne poboljšava klasični algoritam, budući da je

$$\mathrm{Mul}(n) = 4\,\mathrm{Mul}(n/2) + c_1 n + c_2,$$

što daje

$$\mathrm{Mul}(n) = O(n^2).$$

Relaciju (3.1.3) možemo modificirati ako primijetimo da vrijedi sljedeće

$$U_0 \cdot V_1 + U_1 \cdot V_0 = (U_1 + U_0) \cdot (V_1 + V_0) - U_0 \cdot V_0 - U_1 \cdot V_1. \tag{3.1.5}$$

Primijetimo da je dovoljno naći tri produkta, za razliku od prijašnja 4 produkta.

Pomoću relacija (3.1.3) i (3.1.5) dobivamo algoritam 3.1.1 kojega je otkrio Anatoly Karatsuba 1960. godine i objavio ga 1962. godine.

Uz pretpostavku da potrebna 3 množenja obavljamo rekurzivno istim algoritmom, za aritmetičku složenost dobivamo

$$\mathrm{Mul}(n) \leq 2\,\mathrm{Mul}(n/2) + \mathrm{Mul}(n/2 + 1) + c_1 n. \tag{3.1.6}$$

U ovoj sumi, srednji član predstavlja složenost množenja $(U_1 + U_0) \cdot (V_1 + V_0)$, budući da ta dva broja mogu imati $n/2 + 1$ znamenki. Ako se inzistira na parnoj duljini brojeva, lema 3.1.1 garantira da se relacija (3.1.6) ne mijenja bitno ako bismo dodali vodeću nulu i

Algoritam 3.1.1: NMULK0 - Karatsuba

```
1 početak
2     t_1 ← (U_1 + U_0) · (V_1 + V_0);
3     t_2 ← U_0 · V_0;
4     t_3 ← U_1 · V_1;
5     w ← t_3 · b^{2k} + (t_1 − t_2 − t_3) · b^k + t_2;
6 kraj
```

zamijenili $\mathrm{Mul}(n/2+1)$ s $\mathrm{Mul}(n/2+2)$. Primjenom iste leme s $t = 1$ na relaciju (3.1.6), dobivamo složenost Karatsubinovog algoritma

$$\mathrm{Mul}(n) \leq 3\,\mathrm{Mul}(n/2) + c_0 n, \tag{3.1.7}$$

pri čemu je c_0 neka konstanta koja ne ovisi o n. Ako algoritam upotrebljavamo rekurzivno, onda relacija (3.1.7) vrijedi za svaki $n > 1$, a rekurzija završava kad dobijemo pojedinačne znamenke. Kada dođemo do pojedinačnih znamenki, tada koristimo aritmetiku računala u algoritmu NMULK0, pa imamo

$$\mathrm{Mul}(1) \leq c_0'. \tag{3.1.8}$$

Uzmimo $c = \max\{c_0, c_0'\}$. Konačno, dobivamo rekurzivne relacije

$$\mathrm{Mul}(1) \leq c, \qquad \mathrm{Mul}(n) \leq 3\,\mathrm{Mul}(n/2) + cn.$$

Ove rekurzivne relacije ćemo riješiti pomoću sljedeće leme.

Lema 3.1.2. *Neka je $T : \mathbb{N} \to \mathbb{R}$ monotono rastuća funkcija i neka su $a, d, c \in \mathbb{R}^+$ konstante takve da je $a > d$. Ako funkcija T zadovoljava rekurzivne jednadžbe*

$$T(1) \leq c, \qquad T(n) \leq a \cdot T(n/d) + cn, \quad n = d^k, \quad k > 0, \tag{3.1.9}$$

onda je

$$T(n) = \frac{c}{a-d}\left[a \cdot n^{\log_d a} - d \cdot n\right] \tag{3.1.10}$$

za $n = d^k$, $k > 0$, i postoji konstanta $C > 0$ takva da je

$$T(n) \leq c \cdot n^{\log_d a}, \quad \forall n \in \mathbb{N}. \tag{3.1.11}$$

Dokaz. Iz (3.1.9) se indukcijom dokaže da vrijedi

$$T(d^k) = \frac{c}{a-d}\left[a \cdot a^k - d \cdot d^k\right].$$

Relaciju (3.1.10) dobijemo za $n = d^k$ uz sljedeće jednakosti

$$a^k = d^{k \cdot \log_d a} = (d^k)^{\log_d a}.$$

Iz monotonosti funkcije i pretpostavke $a > d$, slijedi (3.1.11). □

Funkcija Mul zadovoljava uvjete funkcije T iz leme 3.1.2, budući da brojeve duljine manje od n možemo nadopuniti vodećim nulama do duljine n. Sada iz relacije (3.1.7) slijedi da za Karatsubin algoritam vrijedi

$$\text{Mul}(n) < C \cdot n^{\log_2 3}, \quad n \in \mathbb{N}. \tag{3.1.12}$$

Ovo predstavlja znatno ubrzanje klasičnog algoritma, budući da je $\log_2 3$ približno 1.585. Ovdje smo proveli samo analizu reda veličine funkcije Mul(n), budući da je preciznija analiza aritmetičke složenosti rekurzivnih algoritama, kao što je Karatsubin algoritam, ovisna o detaljima realizacije, posebno o realizaciji rekurzije.

3.2 Generalizirani Karatsubin algoritam

U Karatsubinom algoritmu smo brojeve u i v rastavljali na dva dijela. U ovom odjeljku ćemo brojeve u i v rastavljati na $r + 1$ dijelova, za neki fiksni $r \in \mathbb{N}$.

Radi jednostavnosti, neka je $n = (r+1)k$, za neki fiksni $r \in \mathbb{N}$. Brojeve u i v zapisujemo u bazi b^k u obliku

$$u = \sum_{i=0}^{r} U_i b^{ki}, \quad v = \sum_{i=0}^{r} V_i b^{ki}, \tag{3.2.1}$$

pri čemu su U_i, V_i k-znamenkasti brojevi u bazi b, za $i = 0, \ldots, r$.

Definiramo polinome $U(x)$ i $V(x)$ s

$$U(x) = \sum_{i=0}^{r} U_i x^i, \quad V(x) = \sum_{i=0}^{r} V_i x^i,$$

te $W(x)$ koji je produkt gornjih polinoma

$$W(x) = U(x) \cdot V(x) = \sum_{i=0}^{2r} W_i x^i.$$

Primijetimo da je $u = U(b^k)$ i $v = V(b^k)$, pa iz toga slijedi da je

$$w = u \cdot v = W(b^k). \tag{3.2.2}$$

Da bismo izračunali

$$w = \sum_{i=0}^{2r} W_i b^{ki}, \tag{3.2.3}$$

vidimo da je dovoljno naći koeficijente W_i, za svaki $i = 0, \ldots, 2r$. Uz te vrijednosti, računanje (3.2.3) se svodi na operacije zbrajanja i množenja potencijom baze. Ova faza zahtijeva broj operacija proporcionalan s $n = (r + 1)k$.

Polinom $W(x)$ je određen, bilo koeficijentima, bilo vrijednostima u $2r + 1$ točaka. Ako odaberemo točke $0, 1, \ldots, 2r$, tada je

$$W(i) = U(i) \cdot V(i), \quad i = 0, \ldots, 2r.$$

Budući da su točke ekvidistantne, koeficijente W_i možemo izračunati Hornerovom shemom i polinomnom aritmetikom, koristeći Newtonov oblik interpolacijskog polinoma

$$W(x) = \sum_{i=0}^{2r} \frac{1}{i!} \Delta^i W(0) \cdot \prod_{j=0}^{i-1} (x - j). \tag{3.2.4}$$

Iz (3.2.4) je lako izračunati onda $w = W(b^k)$.

Procijenimo broj aritmetičkih operacija $\mathrm{Mul}(n)$ u algoritmu 3.2.1, za $n = (r + 1)k$. Pretpostavimo da za sva množenja $W(i) = U(i) \cdot V(i)$ koristimo rekurzivno isti algoritam.

Najprije određujemo duljinu brojeva $U(i)$ i $V(i)$. Pošto je

$$U(i) = \sum_{j=0}^{r} U_j i^j, \quad i = 0, \ldots, 2r,$$

pri čemu su U_j k-znamenkasti brojevi, vrijedi da je $U_j < b^k$, $j = 0, \ldots, r$. Očito je onda

$$U(i) < b^k \cdot \sum_{j=0}^{r} i^j.$$

Brojevi $U(i)$ rastu po i, budući da su znamenke U_j u bazi b^k nenegativne. Zbog toga, za $i = 0, \ldots, 2r$ vrijedi

$$U(i) \leq U(2r) < b^k \cdot \frac{(2r)^{r+1} - 1}{2r - 1}.$$

Zbog pretpostavke da je r proizvoljan prirodan broj, neovisan o n, postoji konstanta $t \in \mathbb{N}$ takva da je

$$U(i) < b^{k+t}, \quad i = 0, \ldots, 2r. \tag{3.2.5}$$

Analogno, ista nejednakost vrijedi i za $V(i)$, $i = 0, \ldots, 2r$. Ako brojeve $U(i)$ i $V(i)$, $i = 0, \ldots, 2r$, nalazimo Hornerovom shemom, koristit će se operacije s najviše $(k + t)$-znamenkastim brojevima u bazi b.

Algoritam 3.2.1: NMULR0 – "podijeli pa vladaj"

početak
  $\{U(i), V(i)\}$;
  za $i \leftarrow 0$ **do** $2r$ **čini**
    $U(i) \leftarrow U_r$;
    za $j \leftarrow r-1$ **do** 0 **čini**
      $U(i) \leftarrow U(i) \cdot i + U_j$
    kraj
    $V(i) \leftarrow V_r$;
    za $j \leftarrow r-1$ **do** 0 **čini**
      $V(i) \leftarrow V(i) \cdot i + V_j$
    kraj
  kraj
  $\{W(i) = U(i) \cdot V(i)\}$;
  za $i \leftarrow 0$ **do** $2r$ **čini**
    $W(i) \leftarrow U(i) \cdot V(i)$;
  kraj
  { tablica konačnih razlika $\Delta^i W(0)$ };
  za $i \leftarrow 1$ **do** $2r$ **čini**
    za $j \leftarrow 2r-1$ **do** i **čini**
      $W(j) \leftarrow (W(j) - W(j-1))$;
    kraj
  kraj
  $\{w = W(b^k)\}$;
  za $i \leftarrow 2r-1$ **do** 0 **čini**
    $w \leftarrow (w \cdot b^k - w \cdot i)/(i+1) + W(i)$;
  kraj
kraj

Brojevi $i = 0, \ldots 2r$ su konstantne duljine ($\ll t$) i možemo pretpostaviti da su duljine 1. Zbog toga svaka operacija za nalaženje $U(i)$ i $V(i)$ zahtijeva najviše $c_0(k+t)$ aritmetičkih operacija na znamenkama u bazi b. Takvih je $2r \cdot (2r+1)$ operacija, pa slijedi

$$\mathrm{Compl}_A(U(i), V(i),\ i = 0, \ldots, 2r) \leq c_0 \cdot (k+t) \cdot 2r \cdot (2r+1). \tag{3.2.6}$$

Za računanje $2r+1$ produkata $W(i)$, potrebno je $(2r+1)\,\mathrm{Mul}(k+t)$ operacija. Iz propozicije 3.1.1 slijedi

$$\mathrm{Compl}_A(W(i),\ i = 0, \ldots, 2r) \leq (2r+1) \cdot [\mathrm{Mul}(k) + c_1 \cdot k]. \tag{3.2.7}$$

Nalaženje konačnih razlika zahtijeva $r\cdot(2r+1)$ oduzimanja. Pošto brojevi $W(i)$ mogu imati najviše $2(k+t)+1$ znamenku, slijedi da je

$$\mathrm{Compl}_A(\Delta^i W(0),\ i=0,\dots,2r) \le c_2\cdot r\cdot(2r+1)\cdot(2(k+t)+1). \tag{3.2.8}$$

Aritmetička složenost računanja broja w je

$$\mathrm{Compl}_A(w) \le c_3\cdot 2r\cdot(2(k+1)+1), \tag{3.2.9}$$

budući da se koriste samo zbrajanja, pomaci, množenje i dijeljenje znamenkama.

Zbrajanjem (3.2.6)–(3.2.9) slijedi da za neku konstantu c vrijedi

$$\mathrm{Mul}((r+1)k) \le (2r+1)\,\mathrm{Mul}(k) + c(r+1)k,$$

odnosno, vraćanjem $n=(r+1)k$,

$$\mathrm{Mul}(n) \le (2r+1)\,\mathrm{Mul}\left(\frac{n}{r+1}\right) + c\cdot n.$$

Kako bismo iskoristili lemu 3.1.2, konstantu c biramo tako da vrijedi $\mathrm{Mul}(1)\le c$, pa onda slijedi

$$\mathrm{Mul}(n) \le C\cdot n^{\log_{r+1}(2r+1)} \le C\cdot n^{1+\log_{r+1}2}, \tag{3.2.10}$$

gdje konstanta C ovisi samo o r i bazi b. Zbog činjenice da, kada r teži u beskonačnost, izraz $\log_{r+1}2$ teži u nulu, relacija (3.2.10) dokazuje sljedeći rezultat.

Teorem 3.2.1. *Neka je $\epsilon>0$ proizvoljan. Tada postoji algoritam množenja takav da je broj aritmetičkih operacija potrebnih za množenje dva n-znamenkasta broja u bazi b dan s*

$$\mathrm{Mul}(n) \le C(\epsilon,b)\cdot n^{1+\epsilon}, \quad \forall n\in\mathbb{N}, \tag{3.2.11}$$

gdje je $C(\epsilon,b)$ konstanta koja ovisi o ϵ i b, ali ne ovisi o n.

Primijetimo još da konstanta proporcionalnosti C, kad povećavamo r, vrlo brzo raste. Iz toga izlazi da je ovaj algoritam koristan samo za ogromne duljine n. Kada je duljina n prikaziva u računalu, pokazalo se da je algoritam efikasan kada vrijedi $r\le 5$, ali i u tom slučaju n mora biti jako velik.

Klasični algoritmi množenja i Karatsubin algoritam su dovoljno dobri u praktičnoj primjeni množenja brojeva. Strategija "podijeli pa vladaj" korištenjem polinomne reprezentacije i interpolacije se u teoriji može još više poboljšati.

Ideja je uzimati veći r što je duljina n veća. Tu ideju je iznio A. L. Toom [*Complexity of a scheme made up of functional elements and realizing multiplication of integers*, Doklady Akad. Nauk SSSR, 150 (1963), 496–498], koji ju je koristio da pokaže kako se mogu konstruirati računalni strujni krugovi za množenje n-bitnih brojeva, uključujući sasvim mali broj komponenti, kako broj n raste. S. A. Cook [*On the minimum Computation time of functions*, Thesis, Harvard University, 1966, 51–77] je kasnije pokazao da se ova metoda može prilagoditi brzim računalnim programima. Detaljnije o ovom algoritmu možete pronaći u [2, str. 281–283].

3.3 Usporedba Karatsubinog algoritma i klasičnog množenja

U ovom kratkom odjeljku donosimo implementacije algoritma za klasično množenje i Karatsubinog algoritma u programskom jeziku Java.

Najprije slijedi implementacija algoritma klasičnog množenja.

```
public class ClassicMultiplication {

  public static int[] ClassicMultiply(int[] u, int[] v, int b) {

    int degU = u.length;
    int degV = v.length;
    int[] w;

    if (degU >= 0 && degV >= 0) {

      w = new int[degU + degV + 1];
      int carry;
      int degW = w.length;
      for (int i = degV - 1; i >= 0; i--) {
        carry = 0;
        for (int j = degU-1; j >= 0; j--) {
          int temp = w[i + j + 2] + v[i]*u[j] + carry;
          w[i+j+2] = temp % b;
          carry = temp / b;
        }
        w[i+1] = carry;
      }


    } else {
      w = new int[1];
      w[0] = -1;
    }

    return w;
  }
}
```

Ova implementacija je skroz identična algoritmu NMUL. Brojeve prikazujemo kao

nizove cjelobrojnih vrijednosti i onda vršimo operacije na elementima nizova. Jedina je razlika što je, u ovoj implementaciji, najznačajniji broj u pozicijskom zapisu predstavljen kao element niza s indeksom 0, sljedeći najznačajniji element je na indeksu 1, itd. Zadnji element niza je najmanje značajna znamenka broja, ali je u nizu postavljena na najvećem indeksu. Stoga, kako bi implementacija odgovarala algoritmu NMUL, onda svaku petlju provodimo tako da krenemo od zadnjeg elementa niza prema prvom elementu niza.

Slijedi implementacija Karatsubinog algoritma.

```
import java.math.BigInteger;

public class Karatsuba {

  private static BigInteger ZERO = new BigInteger("0");

  public static BigInteger karatsuba(BigInteger x, BigInteger y) {

  int N = Math.max(x.bitLength(), y.bitLength());
  if(N <= 3) {
    return x.multiply(y);
  }

  N = (N/2) + (N%2);

  BigInteger b = x.shiftRight(N);
  BigInteger a = x.subtract(b.shiftLeft(N));
  BigInteger d = y.shiftRight(N);
  BigInteger c = y.subtract(d.shiftLeft(N));

  BigInteger ac = karatsuba(a, c);
  BigInteger bd = karatsuba(b, d);
  BigInteger abcd = karatsuba(a.add(b), c.add(d));

  return
    ac.add(abcd.subtract(ac).subtract(bd).shiftLeft(N)).add(bd.shiftLeft(2*N));
  }
}
```

Očito je da je implementacija napravljena rekurzivno. Brojevi čiji produkt želimo izračunati, prikazani su klasom BigInteger. Implementacija je napravljena po uzoru na algoritam NMULK0.

Tablica 3.1 prikazuje razlike u vremenu izvođenja klasičnog množenja i Karatsubinog

algoritma za množenje, na primjeru umnoška dva 80000-bitna broja. Vidimo da je Karatsubin algoritam za takve primjere brojeva brži od klasičnog množenja, bez obzira na arhitekturu računala. Ipak, kako smo već konstatirali, za manje brojeve je razlika zanemariva.

Tablica 3.1: Usporedba klasičnog množenja i Karatsubinog algoritma – $N = 80\,000$

Performanse računala	Klasično množenje	Karatsubin algoritam
Intel Core i7-6700HQ 2.6GHz Turbo Boost up to 3.5GHz	5.103 s	3.300 s
Intel Core i3-5005U CPU @ 2.0 GHz	8.160 s	6.482 s
AMD E-350 APU 1.5GHz	21.700 s	15.260 s

Poglavlje 4

Fourierove transformacije u množenju brojeva

U ovom odjeljku opisat ćemo algoritam koji koristi Fourierove transformacije u množenju cijelih brojeva. Najprije ćemo proučiti što je to Fourierova transformacija i inverzna transformacija, a potom ćemo vidjeti kako su to Strassen i Schönhage iskoristili u svom algoritmu za množenje cijelih brojeva, koji je dosad najbrži algoritam za množenje velikih cijelih brojeva

4.1 Diskretna Fourierova transformacija i inverzna transformacija

Fourierova transformacija je obično definirana nad poljem kompleksnih brojeva. Ovdje ćemo definirati Fourierovu transformaciju nad proizvoljnim komutativnim prstenom $(R, +, \cdot)$.

Kažemo da je element $\omega \in R$ **glavni n-ti korijen iz jedinice** ako vrijedi

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

Elementi $\omega^0, \ldots, \omega^{n-1}$ zovu se n-ti korijeni iz jedinice.

Neka je $a = [a_0, \ldots, a_{n-1}]^T$ vektor stupac duljine n s elementima iz R. Pretpostavljamo da n ima multiplikativni inverz u R i da R ima glavni n-ti korijen iz jedinice ω. Neka je A $n \times n$ matrica takva da je

$$A[i, j] = \omega^{ij}, \quad 0 \leq i, j < n.$$

Vektor $F(a) = Aa$, čija je i-ta komponenta jednaka $\sum_{k=0}^{n-1} a_k \omega^{ik}$, zove se **diskretna Fourierova transformacija** od a. Matrica A je regularna, pa postoji inverz A^{-1}. Ako je ω^{ij} element matrice A u i-tom retku i j-tom stupcu, tada je na istoj poziciji u matrici A^{-1} element $(1/n)\omega^{-ij}$ (vidi lemu 7.1 u [1]). Vektor $F^{-1}(a) = A^{-1}a$, čija je i-ta komponenta (za svaki $i = 0, \ldots, n-1$)

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

zovemo **inverzna diskretna Fourierova transformacija** od a. Očito je inverz inverza od a opet a.

Postoji bliska veza između Fourierove transformacije i polinomne evaluacije i interpolacije. Neka je

$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

polinom stupnja $n-1$. Polinom se može na jedinstven način prikazati u jednom od dva oblika. Prvi oblik je kao lista koeficijenata $a_0, \ldots, a_{n-1}$, a drugi oblik je lista njegovih vrijednosti u n različitih točaka $x_0, \ldots, x_{n-1}$. Računanje Fourierove transformacije vektora $[a_0, \ldots, a_{n-1}]^T$ je ekvivalentno konvertiranju prikaza polinoma pomoću koeficijenata $a_0, \ldots, a_{n-1}$, u njegove vrijednosti u točkama $\omega_0, \ldots, \omega_{n-1}$. Obratno, inverzna Fourierova transformacija je ekvivalentna interpolaciji polinomom za dane vrijednosti u n-tim korijenima iz jedinice.

Jedna od primjena Fourierovih transformacija je u računanju konvolucijskog produkta dva vektora. Neka su $a = [a_0, \ldots, a_{n-1}]^T$ i $b = [b_0, \ldots, b_{n-1}]^T$ dva vektora stupca. Konvolucijski produkt vektora a i b je vektor $c = [c_0, \ldots, c_{2n-1}]^T$, pri čemu je $c_i = \sum_{j=0}^{n-1} a_j b_{i-j}$. Ako je $k < 0$ ili $k \geq n$, uzimamo da je $a_k = b_k = 0$. Dodatno, vrijedi $c_{2n-1} = 0$.

Motivacija za uvođenje konvolucijskog produkta je množenje polinoma. Ako imamo 2 polinoma

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \quad \text{i} \quad q(x) = \sum_{i=0}^{n-1} b_i x^i,$$

tada je umnožak ova dva polinoma jednak

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

Primijetimo da je unutarnja suma upravo konvolucijski produkt vektora koeficijenata polinoma a i b, ako zanemarimo još c_{2n-1}, koji je 0.

Ako su dva polinoma stupnja $n-1$ reprezentirana koeficijentima, onda je za računanje reprezentacije koeficijenata njihovog produkta potrebno konvoluirati 2 vektora koeficije-

nata. S druge strane, ako su polinomi reprezentirani vrijednostima u točkama n-tih korijena iz jedinice, da bi se izračunala reprezentacija vrijednosti njihovog produkta, možemo jednostavno množiti parove vrijednosti u odgovarajućim točkama. Ovo sugerira da je konvolucijski produkt vektora a i b, zapravo, inverzna transformacija od produkta po komponentama od transformacije tih vektora. Označit ćemo to ovako

$$a \star b = F^{-1}(F(a) \cdot F(b)). \tag{4.1.1}$$

Zaključak je da se konvolucijski produkt može izračunati tako da se napravi Fourierova transformacija, izračuna produkt po parovima, i na kraju se provede inverzna transformacija.

Teorem konvolucije, čiji dokaz možete naći u [1, str. 255–256], rješava problem reprezentiranja produkta 2 polinoma stupnja $n-1$, za koji trebamo $2n-2$ točaka za reprezentaciju.

Teorem 4.1.1 (Teorem konvolucije)**.** *Neka su*

$$a = [a_0, \ldots, a_{n-1}, 0, 0 \ldots, 0]^T, \quad b = [b_0, \ldots, b_{n-1}, 0, 0 \ldots, 0]^T$$

dva vektora stupca duljine $2n$*. Neka su*

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

njihove Fourierove transformacije. Tada je

$$a \star b = F^{-1}(F(a) \cdot F(b)).$$

Konvolucijski produkt dva vektora duljine n je vektor duljine $2n$. To zahtijeva punjenje vektora a i b s n nula u teoremu konvolucije. Kako bismo izbjegli punjenje, možemo koristiti "omotane" konvolucije.

Definicija 4.1.1. *Neka su vektori*

$$a = [a_0, \ldots, a_{n-1}]^T, \quad b = [b_0, \ldots, b_{n-1}]^T$$

dva vektora duljine n. ***Pozitivna omotana konvolucija*** *vektora a i b je vektor*

$$c = [c_0, \ldots, c_{n-1}]^T,$$

pri čemu je

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

Također, definiramo ***negativnu omotanu konvoluciju*** *vektora a i b kao vektor*

$$d = [d_0, \ldots, d_{n-1}],$$

pri čemu je

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

Omotane konvolucije koristit ćemo u Schönhage–Strassenovom algoritmu za množenje brojeva. Zasad primijetimo da možemo izračunati 2 polinoma stupnja $n-1$ u n-tim korijenima iz jedinice i pomnožiti dobivene parove vrijednosti u odgovarajućim korijenima. To nam daje n vrijednosti pomoću kojih možemo interpolirati jedinstveni polinom stupnja $n-1$. Vektor koeficijenata ovog jedinstvenog polinoma je, zapravo, pozitivno omotana konvolucija dva vektora koeficijenata originalna dva polinoma.

Teorem 4.1.2. *Neka su vektori*

$$a = [a_0, \ldots, a_{n-1}]^T, \quad b = [b_0, \ldots, b_{n-1}]^T$$

dva vektora duljine n. Neka je ω glavni n-ti korijen iz jedinice i neka je $\psi^2 = \omega$. Pretpostavimo da n ima multiplikativni inverz. Vrijede sljedeće dvije tvrdnje:

1. *Pozitivno omotana konvolucija vektora a i b je dana s $F^{-1}(F(a) \cdot F(b))$*
2. *Neka je $d = [d_0, \ldots, d_{n-1}]^T$ negativno omotana konvolucija vektora a i b. Neka je*

$$\hat{a} = [a_0, \psi a_1, \ldots, \psi^{n-1} a_{n-1}]^T,$$
$$\hat{b} = [b_0, \psi b_1, \ldots, \psi^{n-1} b_{n-1}]^T \ i$$
$$\hat{d} = [d_0, \psi d_1, \ldots, \psi^{n-1} d_{n-1}]^T.$$

Tada vrijedi

$$\hat{d} = F^{-1}(F(\hat{a}) \cdot F(\hat{b})).$$

Dokaz. Dokaz ovog teorema je analogan dokazu teorema 4.1.1, uz primjenu $\psi^n = 1$. □

Jasno je da se Fourierova transformacija i inverzna transformacija vektora a u prostoru R^n može izračunati u vremenu $O_A(n^2)$, ako pretpostavimo da svaka aritmetička operacija na proizvoljnim elementima iz prstena R zahtijeva jedan korak. Međutim, kada je n potencija od 2, postoji algoritam koji u vremenu $O_A(n \log n)$ može izračunati Fourierovu transformaciju i inverznu transformaciju, i taj algoritam se smatra optimalnim. U [1, str. 257–265] možete naći psudokod takozvanog FFT (Fast Fourier Transformation) algoritma, kao i detalje i primjere oko samog algoritma.

4.2 Schönhage–Strassenov algoritam

Razmatranja iz prošlog odjeljka primijenit ćemo na množenje dva n-bitna cijela broja u i v. Radi jednostavnosti, pretpostavimo da je n potencija broja 2 (ako nije, dodamo nule na vodeća mjesta dok ne postane potencija od 2). Nadalje, izračunat ćemo produkt dva n-bitna cijela broja modulo $(2^n + 1)$. Da bismo dobili egzaktan produkt dva n-bitna cijela broja, moramo uvesti vodeće nule i pomnožiti $2n$-bitne cijele brojeve modulo $(2^{2n} + 1)$, povećavajući na taj način vrijeme izračunavanja za konstantan faktor.

Neka su u i v binarni cijeli brojevi između 0 i 2^n koje množimo modulo $(2^n + 1)$. Primijetite da broj 2^n zahtijeva $n + 1$ bitova za binarnu reprezentaciju. Ako je jedan od brojeva u i v jednak 2^n, tada ćemo ga reprezentirati nekim specijalnim simbolom (recimo, simbolom -1) i množenje će biti jednostavno riješeno kao jedan specijalan slučaj. Za ilustraciju, ako je $v = 2^n$, tada je $uv \bmod (2^n + 1)$ jednak $(2^n + 1 - u) \bmod (2^n + 1)$.

Pretpostavimo da je $n = 2^k$ i neka je $b = 2^{k/2}$, ako je k paran, inače je $b = 2^{(k-1)/2}$. Neka je $l = n/b$. Primijetite da je $l \geq b$ i da b dijeli l. Prvi korak je podijeliti u i v na b l-bitnih blokova. Imamo

$$u = u_{b-1}2^{(b-1)l} + \ldots + u_1 \cdot 2 + u_0, \qquad v = v_{b-1}2^{(b-1)l} + \ldots + v_1 \cdot 2 + v_0.$$

Produkt uv dan je s

$$uv = y_{2b-2}2^{(2b-2)l} + \ldots + y_1 \cdot 2 + y_0, \tag{4.2.1}$$

pri čemu je

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

Za $j < 0$ i $j > b - 1$ uzimamo $u_j = v_j = 0$. Izraz y_{2b-1} je 0 i uključen je samo radi simetrije.

Produkt uv možemo izračunati koristeći teorem konvolucije. Množenje Fourierovih transformacija zahtijeva $2b$ množenja. Koristeći omotanu konvoluciju, možemo smanjiti broj množenja na b. Ovo je razlog zašto računamo produkt uv modulo $2^n + 1$. Pošto je $bl = n$, imamo $2^{bl} \equiv -1 \bmod (2^n + 1)$. Prema [1, str. 266], pokazuje se da vrijedi

$$uv \equiv (w_{b-1}2^{(b-1)l} + \ldots w_1 \cdot 2 + w_0) \bmod (2^n + 1),$$

gdje je $w_i = y_i - y_{b+i}$, za $0 \leq i < b$.

Pošto je produkt 2 l-bitna broja nužno manji od 2^{2l}, i y_i i y_{b+i} su zbrojevi od $i + 1$ i $b-(i+1)$ takvih produkata, a vrijednosti w_i moraju biti između $(b-1-i)2^{2l}$ i $(i+1)2^{2l}$. Slijedi da postoji najviše $b \cdot 2^{2l}$ vrijednosti koje w_i može poprimiti. Kada bismo izračunali w_i-ove modulo $b \cdot 2^{2l}$, mogli bismo izračunati produkt uv modulo (2^n+1) u dodatnih $O(b \log(b \cdot 2^{2l}))$ koraka, dodavajući b vrijednostima w_i, zajedno s pomacima.

Za izračunavanje w_i modulo $(b \cdot 2^{2l})$, potrebno je dva puta računati w_i: jednom modulo b (označavamo w_i'), jednom modulo $(2^{2l} + 1)$ (označavamo w_i''). Budući da je b potencija

broja 2 i da je $2^{2l} + 1$ neparan broj, b i $2^{2l} + 1$ su relativno prosti. Slijedi da w_i možemo izračunati kao

$$w_i = (2^{2l} + 1)((w'_i - w''_i) \bmod b) + w''_i, \tag{4.2.2}$$

i w_i je između $(b - 1 - i)2^{2l}$ i $(i + 1)2^{2l}$. Složenost ovakvog izračunavanja w_i je $O(l + \log b)$, za svaki $i = 0, \ldots, b - 1$, što sveukupno iznosi $O(bl + b \log b)$, odnosno, $O(n)$.

Brojevi w_i se izračunavaju modulo b, uzimajući $u'_i = u_i \bmod b$ i $v'_i = v_i \bmod b$, i formirajući brojeve $\hat{u}$ i $\hat{v}$ na sljedeći način

$$\begin{aligned} \hat{u} &= u'_{b-1}00\ldots0u'_{b-2}00\ldots0u'_{b-3}00\ldots0u'_0, \\ \hat{v} &= v'_{b-1}00\ldots0v'_{b-2}00\ldots0v'_{b-3}00\ldots0v'_0. \end{aligned} \tag{4.2.3}$$

Ukupno je $2 \log b$ nula u svakom bloku nula. Složenost izračunavanja produkta $\hat{u}\hat{v}$ koristeći metodu "podijeli pa vladaj" je $O((3b \log b)^{1.6})$, dakle, manje od $O(n)$ koraka. Vidimo da je

$$\hat{u}\hat{v} = \sum_{i=0}^{2b-1} y'_i \cdot 2^{(3 \log b)i},$$

pri čemu je $y'_i = \sum_{j=0}^{2b-1} u'_j v'_j$. Budući da je $y'_i < 2^{3 \log b}$, slijedi da do y'_i-ova možemo lako doći iz produkta $\hat{u}\hat{v}$. Vrijednosti w_i modulo b možemo izračunati računanjem $(y'_i - y'_{b+i})$ modulo b.

Kada moramo izračunati svaki w_i modulo b, onda računamo svaku vrijednost w_i modulo $(2^{2l} + 1)$ pomoću omotane konvolucije. To uključuje primjenu Fourierove transformacije, množenje odgovarajućih parova i inverznu transformaciju. Neka je $\omega = 2^{4l/b}$ i neka je $m = 2^{2l} + 1$. Vrijedi da ω i b imaju multiplikativni inverz modulo m i ω je glavni b-ti korijen iz jedinice. Slijedi da je negativna omotana konvolucija od $[u_0, \psi u_1, \ldots, \psi^{b-1}u_{b-1}]$ i $[v_0, \psi v_1, \ldots, \psi^{b-1}v_{b-1}]$, pri čemu je $\psi = 2^{2l/b}$(ψ je $2b$-ti korijen iz jedinice), jednaka

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

gdje je

$$y_i = \sum_{j=0}^{b-1} u_j v_{i-j}, \quad i = 0, \ldots, 2b - 1.$$

Vrijednosti w_i se mogu dobiti odgovarajućim pomacima.

Opišimo sada korake Schönhage–Strassenovog algoritma. Ulazni podaci su n-bitni brojevi u i v, pri čemu je $n = 2^k$. Izlaz je $(n + 1)$-bitni produkt brojeva u i v modulo $2^n + 1$.

Ako je n malen, tada je produkt bolje izračunati pomoću nekih od prije spomenutih algoritama, budući da za male n, Schönhage–Strassenov algoritam ne daje bitno poboljšanje.

Ako je n velik, za parni k postavimo $b = 2^{k/2}$, a inače postavimo $b = 2^{(k-1)/2}$. Neka je $l = n/b$. Izrazimo brojeve u obliku

$$u = \sum_{i=0}^{b-1} u_i 2^{li}, \quad v = \sum_{i=0}^{b-1} v_i 2^{li},$$

pri čemu su u_i i v_i, cijeli brojevi između 0 i $2^l - 1$, za $i = 0, \ldots, b-1$ (odnosno, u_i-ovi su l-bitni blokovi od u, a v_i-ovi su l-bitni blokovi od v).

1. Uz $\psi = 2^{2l/b}$, koristeći da je ψ^2 glavni b-ti korijen iz jedinice, izračunati Fourierove transformacije modulo $2^{2l} + 1$ vektora

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

2. Izračunati produkt po parovima od Fourierovih transformacija (izračunatih u koraku 1), modulo $2^{2l} + 1$, koristeći rekurzivno ovaj isti algoritam.
3. Izračunati inverznu Fourierovu transformaciju modulo $2^{2l} + 1$ vektora produkata po parovima iz koraka 2. Rezultat računanja je

$$[w_0, \psi w_1, \ldots, \psi^{b-1} w_{b-1}]^T \bmod (2^{2l} + 1),$$

   gdje je w_i i-ta komponenta negativne omotane konvolucije vektora $[u_0, \ldots, u_{b-1}]^T$ i $[v_0, \ldots, v_{b-1}]^T$. Izračunati $w_i'' = w_i \bmod (2^{2l} + 1)$, množeći $\psi^i w_i$ s $\psi^{-i} \bmod (2^{2l} + 1)$.
4. Izračunati $w_i' = w_i \bmod b$ na sljedeći način:
   a) Neka je $u_i' = u_i \bmod b$ i neka je $v_i' = v_i \bmod b$, za $0 \le i < b$.
   b) Konstruirati brojeve $\hat{u}$ i $\hat{v}$ kao u (4.2.3).
   c) Izračunati produkt $\hat{u}\hat{v}$ pomoću Karatsubinog algoritma.
   d) Produkt $\hat{u}\hat{v}$ je $\sum_{i=0}^{2b-1} y_i' 2^{(3 \log b)i}$, pri čemu je $y_i' = \sum_{j=0}^{2b-1} u_j' v_{i-j}'$. Sada izračunamo $w_i = (y_i' - y_{b+1}') \bmod b$, za $0 \le i < b$.
5. Izračunati egzaktno w_i-ove, koristeći

$$w_i = (2^{2l} + 1)((w_i' - w_i'') \bmod b) + w_i''$$

   i vrijedi $(b - 1 - i)2^{2l} < w_i < (i + 1)2^{2l}$.
6. Izračunati

$$\sum_{j=0}^{b-1} w_i 2^{li} \bmod (2^n + 1).$$

   Ovo je i konačni rezultat.

Ostaje još vidjeti aritmetičku složenost ovog algoritma. Iz [1, str. 268, korolar teorema 7.6], slijedi da je složenost prva tri koraka algoritma jednaka $O(bl \log b + b\,\mathrm{Mul}(2l))$, pri čemu je $\mathrm{Mul}(m)$ vrijeme potrebno za množenje 2 m-bitna cijela broja rekurzivnom primjenom algoritma. U koraku 4, gradimo $\hat{u}$ i $\hat{v}$ duljine $3b \log b$ i množimo ih u $O((3b \log b)^{1.59}$ koraka. Za dovoljno velike b, to je manje od b^2, pa se složenost koraka 4 može zanemariti, zbog složenosti prva 3 koraka. Koraci 5 i 6 su složenosti $O(n)$, pa i njih možemo zanemariti.

Pošto je $n = bl$ i b je najviše $\sqrt{n}$, dobivamo nejednakost

$$\mathrm{Mul}(n) \leq cn \log n + b\,\mathrm{Mul}(2l), \tag{4.2.4}$$

za neku konstantu c i dovoljno veliki n. Neka je $\mathrm{Mul}'(n) = \mathrm{Mul}(n)/n$. Tada prethodna relacija postaje

$$\mathrm{Mul}'(n) \leq c \log n + 2\,\mathrm{Mul}'(2l). \tag{4.2.5}$$

Pošto je $l \leq 2\sqrt{n}$, vrijedi

$$\mathrm{Mul}'(n) \leq c \log n + 2\,\mathrm{Mul}'(4\sqrt{n}). \tag{4.2.6}$$

Zamjenom $\mathrm{Mul}'(4\sqrt{n})$ s $c' \log(4\sqrt{n}) \log\log 4\sqrt{n}$ dobivamo

$$\mathrm{Mul}'(n) \leq c \log n + 4c' \log\left(2 + \frac{1}{2}\log n\right) + c' \log n \log\left(2 + \frac{1}{2}\log n\right).$$

Za velike n, također, vrijedi $2 + \frac{1}{2}\log n \leq \frac{2}{3}\log n$. Iz toga slijedi

$$\mathrm{Mul}'(n) \leq c' \log n + 4c' \log\frac{2}{3} + 4c' \log\log n + c' \log\frac{2}{3} \log n + c' \log n \log\log n. \tag{4.2.7}$$

Iz posljednje relacije vidimo da su prva četiri izraza u sumi dominirana zadnjim izrazom, koji je negativan. Zaključujemo da vrijedi $\mathrm{Mul}'(n) \leq c' \log n \log\log n$, odnosno, $\mathrm{Mul}(n) \leq c' n \log n \log\log n$. Dakle, aritmetička složenost Schönhage–Strassenovog algoritma je $O(n \log n \log\log n)$.

Poglavlje 5

Dijeljenje velikih brojeva

5.1 Traženje aproksimacije recipročne vrijednosti

Dijeljenje brojeva u i v, zapravo, predstavlja množenje broja u recipročnom vrijednošću broja v. Stoga je razumno pretpostavljati da se dijeljenje prirodnih brojeva može obaviti podjednako brzo kao i množenje prirodnih brojeva. Problem nastaje što recipročna vrijednost nekog broja više nije cijeli broj i može imati beskonačni prikaz u bazi b. Cilj je pronaći neku dovoljno točnu aproksimaciju za $1/v$.

Pošto nas zanima cjelobrojni kvocijent

$$q = \left\lfloor \frac{u}{v} \right\rfloor,$$

aproksimacija za $1/v$ mora biti toliko točna da nam omogući nalaženje broja q ili njegove dobre procjene $\hat{q}$.

Neka je $\ell(u) = n$ i $\ell(v) = m$. Kada bismo našli aproksimaciju w za $1/v$ s točnošću ϵ

$$1/v = w + \epsilon,$$

gdje je

$$|\epsilon| \leq b^{-n},$$

onda je

$$\left| \frac{u}{v} - u \cdot w \right| = u \cdot |\epsilon| \leq 1. \tag{5.1.1}$$

Uzmemo li

$$\hat{q} = \lfloor u \cdot w \rfloor,$$

onda je

$$|q - \hat{q}| \leq 1. \tag{5.1.2}$$

Za nalaženje w ćemo koristiti Newtonovu metodu za rješavanje jednadžbe

$$\frac{1}{x} - v = 0.$$

Pretpostavimo da možemo naći dovoljno dobru startnu aproksimaciju $w_{(0)}$ za $1/v$. Svaku sljedeću aproksimaciju dobivamo koristeći rekurzivnu relaciju

$$w_{(k+1)} = w_{(k)} \cdot (2 - v \cdot w_{(k)}), \quad k \geq 0. \tag{5.1.3}$$

Označimo s $e_{(k)}$ pogrešku k-te aproksimacije

$$e_{(k)} = \frac{1}{v} - w_{(k)}, \quad k \geq 0.$$

Iz (5.1.3) slijedi

$$e_{(k+1)} = v \cdot e_{(k)}^2, \quad k \geq 0. \tag{5.1.4}$$

Ako osiguramo da vrijedi $|e_{(k)}| \leq b^{-(m+d)}$, onda iz (5.1.4), zbog $v < b^m$, slijedi

$$|e_{(k+1)}| \leq b^{-(m+2d)}. \tag{5.1.5}$$

Zbog $b^{m-1} \leq v < b^m$, za $1/v$ vrijedi

$$b^{-m} < \frac{1}{v} \leq b^{-(m-1)}, \tag{5.1.6}$$

pa slijedi da svaki sljedeći korak udvostručuje broj točnih znamenki, pod pretpostavkom da se u (5.1.3) operacije izvode egzaktno.

Za praktičnu realizaciju moramo brojeve $w_{(k)}$ nekako prikazati nekim konačnim nizom znamenki u bazi b. Također, bilo bi dobro da se čitav proces realizira korištenjem aritmetike prirodnih brojeva.

Zbog relacije (5.1.6), broj $1/v$ ima sljedeći prikaz u bazi b

$$\frac{1}{v} = b^{-(n-1)} \cdot \sum_{i=0}^{\infty} v_i' b^{-i},$$

gdje je

$$0 \leq v_i' < b, \quad i \in \mathbb{N}_0,$$

i vrijedi

$$b^{-m} < \frac{1}{v} < b^{-(m-1)} \iff v_0' = 0 \text{ i } v_1' > 0$$
$$\frac{1}{v} = b^{-(m-1)} \iff v_0' = 1 \text{ i } v_i' = 0, \quad \forall i \in \mathbb{N}.$$

Pretpostavimo da imamo zadanu preciznost p i da želimo izračunati aproksimaciju w' za $1/w$, takvu da vrijedi

$$b^{m-1+p} \cdot w' = \left\lfloor \frac{b^{m-1+p}}{v} \right\rfloor = \sum_{i=0}^{p} v'_i b^{p-i} = \sum_{i=0}^{p} w_i b^i, \tag{5.1.7}$$

pri čemu je $w_i = v'_{p-i}$. Uočimo da je $\ell(w') = p$, osim u slučaju $v = b^{m-1}$, kada je duljina od w' jednaka $p - 1$.

Iz ovoga vidimo da u relaciji (5.1.3) lako možemo prijeći na cijele brojeve. Ako tu relaciju pomnožimo s b^{m-1+p} i definiramo

$$w'_{(k)} = b^{m-1+p} \cdot w_{(k)}, \quad k \geq 0,$$

dobivamo rekurzivnu relaciju

$$w'_{(k+1)} = 2w'_{(k)} - \frac{v}{b^{m-1+p}} \cdot (w'_{(k)})^2, \quad k \geq 0, \tag{5.1.8}$$

u kojoj se svaka operacija može obaviti koristeći cjelobrojnu aritmetiku.

Ipak, (5.1.8) se ne isplati cijelo vrijeme provoditi u zadanoj točnosti, jer u ranoj fazi algoritma sve znamenke broja $w'_{(k)}$ sudjeluju u operacijama, iako ih većina nije točna.

Još jedna mogućnost je da p povećavamo tokom iteriranja, dupliranjem p u svakoj sljedećoj iteraciji. U tom slučaju, treba i od broja v koristiti samo dio vodećih znamenki u (5.1.8), pazeći da se ne gubi na točnosti.

Radi jednostavnosti, pretpostavimo $b = 2$ i da je broj m potencija broja 2, tj.

$$m = 2^l,$$

i da je tražena točnost $p = m$. Ova ograničenja se mogu izbjeći, što ćemo opisati kasnije. Osnovna ideja algoritma 5.1.1 (NREC) je da u k-tom koraku koristimo točnost $p_k = 2^k$ i da koristimo samo p_k vodećih znamenki broja v.

Teorem 5.1.1. *Algoritam* NREC *nalazi broj $w \in \mathbb{N}$ takav da vrijedi*

$$w = \lfloor 2^{2m-1}/v \rfloor, \tag{5.1.9}$$

odnosno, vrijedi

$$v \cdot w = 2^{2m-1} - r, \quad 0 \leq r < v. \tag{5.1.10}$$

Dokaz. Želimo dokazati da algoritam nalazi niz brojeva

$$w_{(k)} = \left\lfloor \frac{2^{2p-1}}{\lfloor v/2^{m-p} \rfloor} \right\rfloor, \quad k = 0, \ldots, l, \tag{5.1.11}$$

Algoritam 5.1.1: NREC – recipročni prirodni broj

Podaci: Niz znamenki $(v_{m-1}, \ldots, v_0)$ prirodnog broja v u bazi 2, uz pretpostavku da za $m = \ell(v)$ vrijedi $m = 2^l$, za neki $l \in \mathbb{N}_0$

Rezultat: Niz znamenki $(w_{r-1}, \ldots, w_0)$ broja $w = \lfloor 2^{2m-1}/v \rfloor$, gdje je $r = \ell(w) \in \{m, m+1\}$

početak
  $m \leftarrow \deg(v) + 1$;
  $(w_1, w_0) \leftarrow (1, 0)$;
  $p \leftarrow 1$;
  ako $m > 1$ **onda**
    ponavljaj
      $p \leftarrow 2 \cdot p$;
      $(s_{2p-1}, \ldots, s_0) \leftarrow (w_{p/2}, \ldots, w_0) \cdot 2^{3p/2} - (w_{p/2}, \ldots, w_0)^2 \cdot (v_{m-1}, \ldots, v_{m-p})$;
      $(w_p, \ldots, w_0) \leftarrow (s_{2p-1}, \ldots, s_{p-1})$;
      za $i \leftarrow 2$ **do** 0 **čini**
        ako $((w_p, \ldots, w_0) + 2^i) \cdot (v_{m-1}, \ldots, w_{m-p}) \leq 2^{2p-1}$ **onda**
          $(w_p, \ldots, w_0) \leftarrow (w_p, \ldots, w_0) + 2^i$;
        inače
        kraj
      kraj
    dok $p = m$;
  inače
  kraj
  ako $w_m = 0$ **onda**
    $\deg(w) \leftarrow \deg(v)$;
  inače
    $\deg(w) \leftarrow \deg(v) + 1$;
  kraj
kraj

pri čemu je $w = w_{(0)}$ na početku algoritma i $w = w_{(k)}$ na kraju petlje, u kojoj je $p = 2^k$, za $k = 1, \ldots, l$.

Relacija (5.1.9) izlazi iz (5.1.11), zbog $m = 2^l$, pošto algoritam vraća $w = w_{(l)}$. Da bismo dokazali (5.1.11), provest ćemo indukciju po k.

Za $k = 0$ je $p = 1$, iz čega, zbog $v_{m-1} > 0$ i $b = 2$, vrijedi

$$\lfloor v/2^{m-1} \rfloor = v_{m-1} = 1.$$

Pošto, na početku, algoritam postavlja w na $(0,1) = 2$, očito vrijedi (5.1.11).

Za korak indukcije, pretpostavimo prvo da (5.1.11) vrijedi za svaki $k < l$ i označimo $p = 2^{k+1}$. Za jednostavniji zapis, označimo

$$v_{(k)} = \lfloor v/2^{m-p/2} \rfloor = (v_{m-1}, ..., v_{m-p/2})$$
$$v_{(k+1)} = \lfloor v/2^{m-p} \rfloor \quad = (v_{m-1}, ..., v_{m-p}).$$

Slijedi

$$v_{(k+1)} = v_{(k)} \cdot 2^{p/2} + v'_{(k)},$$

uz $v'_{(k)} = (v_{m-p/2-1}, \ldots, v_{m-p})$.

Sada, po pretpostavci indukcije, na početku koraka petlje s $p = 2^{k+1}$, vrijedi

$$w_{(k)} = \left\lfloor \frac{2^{p-1}}{v_{(k)}} \right\rfloor = (w_{p/2}, \ldots, w_0).$$

To znači da vrijedi

$$v_{(k)} \cdot w_{(k)} = 2^{p-1} - r_{(k)}, \quad 0 \leq r_{(k)} < v_{(k)}. \tag{5.1.12}$$

U algoritmu, u prvoj naredbi petlje, računa se broj

$$s = 2^{3p/2} \cdot w_{(k)} - v_{(k+1)} \cdot w_{(k)}^2. \tag{5.1.13}$$

Pošto je $v_m = 1$, vrijedi da je $v_{(k)} \geq 2^{p/2-1}$, pa je $w_{(k)} \leq 2^{p/2}$. To pokazuje da je

$$s \leq 2^{3p/2} \cdot 2^{p/2} - v_{(k+1)} \cdot w_{(k)}^2 < 2^{2p}.$$

Dakle, algoritam egzaktno računa broj $s = (s_{2p-1}, \ldots, s_0)$.

Sada promatramo produkt $s \cdot v_{(k+1)}$. Iz (5.1.13) izlazi

$$s \cdot w_{(k+1)} = s \cdot (v_{(k)} \cdot 2^{p/2} + v'_{(k)}) = v_{(k)} w_{(k)} 2^{2p} + v'_{(k)} w_{(k)} 2^{3p/2} - (v_{(k)} w_{(k)} 2^{2p} + v'_{(k)} w_{(k)})^2.$$

Primjenom relacije (5.1.12) dobivamo

$$s v_{(k+1)} = 2^{3p-2} - (2^{p/2} r_{(k)} - v'_{(k)} w_{(k)})^2.$$

Dijeljenjem s 2^{p-1}, dobivamo

$$2^{2p-1} = \frac{s v_{(k+1)}}{2^{p-1}} + t, \tag{5.1.14}$$

gdje je

$$t = 2^{-(p-1)} \cdot (2^{p/2} r_{(k)} - v'_{(k)} w_{(k)})^2.$$

Iz relacije (5.1.12) slijedi $r_{(k)} < 2^{p/2}$, jer je i $v_{(k)} < 2^{p/2}$. Također, vrijedi i $w_{(k)} \leq 2^{p/2}$ i $v'_{(k)} < 2^{p/2}$, iz čega izlazi

$$|2^{p/2} r_{(k)} - v'_{(k)} w_{(k)}| < 2^p,$$

što daje

$$0 \le t \le 2^{p+1}. \tag{5.1.15}$$

Algoritam postavlja

$$w = \lfloor s/2^{p-1} \rfloor.$$

Zbog $t \ge 0$, iz (5.1.14) slijedi

$$2^{2p-1} \ge \frac{sv_{(k+1)}}{2^{p-1}} \ge v_{(k+1)} \cdot \lfloor s/2^{p-1} \rfloor > v_{(k+1)} \cdot \left(\frac{s}{2^p - 1} - 1 \right) = 2^{2p-1} - v_{(k+1)} - t,$$

što, uz $v_{(k+1)} < 2^p$ i $t < 2^{p+1}$, daje

$$2^{2p-1} \ge v_{(k+1)} \cdot w > 2^{2p-1} - 2^{p+1} - 2^p.$$

To znači da je

$$v_{(k+1)} \cdot w = 2^{2p-1} - r', \quad 0 \le r' < 2^{p+1} + 2^p.$$

Pošto je $v_{(k+1)} < 2^{p-1}$, slijedi

$$0 \le r' < 6v_{(k+1)}.$$

Korekcijom $w_{(k+1)} = w + c$ (algoritam na kraju upravo vrši ovakvu korekciju), s $c < 6$, možemo postići

$$v_{(k+1)} \cdot w_{(k+1)} = 2^{2p-1} - r_{(k+1)}, \quad 0 \le r_{(k+1)} < v_{(k+1)},$$

čime smo dokazali relaciju (5.1.11). □

Primijetimo da je algoritam rekurzivan i da, zbog korekcija, vrijednosti $w_{(k)}$ ne izlaze direktno Newtonovom metodom.

5.2 Složenost algoritma NREC

Označimo s Rec(m) aritmetičku složenost algoritma NREC. Ona ovisi o složenosti algoritma za množenje kojeg koristimo u algoritmu.

Za nalaženje $w = w_{(l)}$, potrebno je pronaći $w_{(l-1)}$ i obaviti zadnji prolaz kroz petlju u algoritmu, s $p = m = 2^l$.

Za nalaženje broja s, potrebno je kvadrirati broj $w_{(l-1)}$ i pomnožiti s v, što uz zbrajanja i pomak, iznosi

$$\mathrm{Compl}_A(s) \le \mathrm{Mul}(m/2 + 1) + \mathrm{Mul}(m + 2) + c_1 m.$$

Faza korekcije zahtijeva najviše 3 množenja, uz nekoliko zbrajanja i uspoređivanja. Moguće je korekciju napraviti i sa samo jednim množenjem, uz dodatno zbrajanje i uspoređivanje. Nalaženje broja w iz s zahtijeva samo pomak.

Iz propozicije 3.1.1, slijedi

$$\mathrm{Rec}(m) \leq \mathrm{Rec}(m/2) + \frac{5}{2}\,\mathrm{Mul}(m) + cm.$$

Ako odaberemo konstantu c tako da je

$$\mathrm{Rec}(1) \leq c + \frac{5}{2}\,\mathrm{Mul}(1),$$

onda je

$$\mathrm{Rec}(m) \leq \sum_{k=0}^{l} \frac{5}{2}\,\mathrm{Mul}(2^k) + c2^k.$$

Pretpostavimo da $\mathrm{Mul}(n)$ raste barem linearno, odnosno, pretpostavljamo da vrijedi

$$\mathrm{Mul}\left(\frac{1}{2}n\right) \leq \frac{1}{2}\,\mathrm{Mul}(n), \quad \forall n \geq 2.$$

Tada je

$$\mathrm{Rec}(m) \leq 5\,\mathrm{Mul}(m) + cm, \quad \forall m \in \mathbb{N}.$$

Lako se vidi da postoji konstanta c' takva da je

$$\mathrm{Rec}(m) \leq c'\,\mathrm{Mul}(m), \quad \forall m \in \mathbb{N}.$$

Može se i pokazati da vrijedi

$$\mathrm{Rec}(m) = \Theta(\mathrm{Mul}(m)).$$

Istaknimo još da se algoritam NREC može generalizirati na baze $b > 2$, ali je tada korekcija znatno kompliciranija. Također, u algoritmu NREC pretpostavili smo da je duljina m potencija broja 2 i da je tražena preciznost $p = m$. U [4, str. 117, Napomena 1.6.4] možete naći kratak opis postupka kada m ne mora biti potencija broja 2.

Za cjelobrojno dijeljenje, relacija $|\epsilon| \leq b^{-n}$ pokazuje da je potrebno $1/v$ naći s preciznošću $p = n - m + 1$.

Bibliografija

[1] A. V. Aho, J. E. Hopcroft i Ullman J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, 1976.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading, 1997.

[3] Saša Singer, *Predavanja iz Oblikovanja i analize algoritama*, https://web.math.pmf.unizg.hr/~singer/oaa/scans/rek_alg.pdf.

[4] ———, *Skripta iz aritmetičkih i algebarskih algoritama*, https://web.math.pmf.unizg.hr/~singer/aa_alg/00_aa.pdf.

Sažetak

Efikasna realizacija osnovnih aritmetičkih operacija na velikim brojevima se, obično, izvodi tako da brojeve prikažemo kao niz znamenki u nekoj odabranoj bazi, tako da se osnovne operacije na znamenkama egzaktno i brzo izvode u aritmetici računala. Algoritmi za zbrajanje i oduzimanje brojeva koji oponašaju algoritme za "ručno" računanje, linearno ovise o duljini brojeva i optimalni su. S druge strane, algoritmi koji oponašaju "ručno" množenje i dijeljenje su kvadratne složenosti.

U ovom radu smo pokazali da postoje bolji algoritmi za množenje i dijeljenje velikih brojeva. Najprije smo detaljno opisali algoritam klasičnog ("ručnog") množenja, za kojeg smo dokazali da ima kvadratnu složenost. Prvo poboljšanje tog algoritma smo dobili metodom "podijeli pa vladaj" – podijelili smo brojeve na 2 i više dijelova. Uz korištenje polinomne interpolacije i evaluacije, dobili smo algoritam čija složenost ne ovisi više kvadratno o duljini brojeva. Taj algoritam se zove Karatsubin algoritam, u čast A. Karatsube, koji je prvi došao do tog algoritma. Također, napravili smo i implementaciju klasičnog i Karatsubinog algoritma u programskom jeziku Java te smo na primjeru pokazali da je Karasubin algoritam efikasniji od klasičnog algoritma.

Još bolji algoritam je Schönhage–Strassenov algoritam, koji koristi brzu Fourierovu transformaciju za množenje polinoma. Jedno poglavlje smo posvetili opisu tog algoritma te pokazali da je njegova složenost $O(n \log n \log\log n)$. To je, trenutno, najbrži poznati algoritam za množenje brojeva.

Na kraju smo opisali i opći algoritam za brzo dijeljenje, preko brzog množenja, koji je baziran na Newtonovoj metodi s progresivnim povećanjem točnosti. Dokazali smo da složenost algoritma za brzo dijeljenje linearno ovisi o izabranom algoritmu za množenje.

Summary

Efficient realization of basic arithmetic operations on big natural numbers is usually performed by representing big numbers as arrays of digits in a chosen base, so that arithmetic operations on digits are performed exactly and quickly in computer arithmetics. Addition and subtraction algorithms, which simulate ”manual” addition and subtraction, have linear complexity, depending on number length. Also, these algorithms are optimal. On the other hand, ”manual” multiplication and division algorithms have quadratic complexity, and they are not optimal.

At the beginning of this thesis, we described the classic multiplication algorithm and prooved that the complexity of this algorithm is quadratic. We got the first improvement in complexity by designing an algorithm with the ”divide and conquer” strategy. By splitting numbers in 2 or more pieces, and with usage of polynomial interpolation and evaluation, we got an algorithm whose complexity is less than quadratic. This algorithm is called Karatsuba's algorithm, in honor of A. Karatsuba, who invented this algorithm. We have also implemented the classic and Karatsuba's algorithm in Java programming language, and showed an example of improvement in speed by using Karatsuba's algorithm, instead of the classic algorithm.

An even better multiplication algorithm is the Schönhage–Strassen algorithm, which uses the Fast Fourier Transform for polynomial multiplication. We have one chapter dedicated to describing this algorithm which has the complexity $O(n \log n \log \log n)$. This is the fastest multiplication algorithm, so far.

At the end, we have described a general division algorithm, based on fast multiplication, which uses Newton's method with progressive accuracy improvement. We have proved that the complexity of this algorithm depends linearly on the complexity of the chosen multiplication algorithm.

Životopis

Rođen sam 17. srpnja 1992. godine u Bjelovaru. Osnovnu školu sam završio u Garešnici 2007. godine, odličnim uspjehom. Za vrijeme osnovnoškolskog obrazovanja, sudjelovao sam na županijskim natjecanjima iz matematike, hrvatskog jezika i povijesti. 2007. godine sam sudjelovao na državnoj smotri LiDraNo u Novigradu Istarskom. U Kutini sam 2011. završio prirodoslovno–matematičku gimnaziju, također, odličnim uspjehom. Za vrijeme srednjoškolskog obrazovanja, sudjelovao sam na županijskim natjecanjima iz matematike, povijesti i geografije. 2011. godine sam sudjelovao na državnom natjecanju iz geografije u Poreču. Iste godine sam u Zagrebu upisao preddiplomski studij Matematika–smjer inženjerski na Prirodoslovno–matematičkom fakultetu, koji sam završio u srpnju 2015. godine, postavši tako prvostupnik matematike. U rujnu iste godine upisao sam diplomski studij Računarstvo i matematika na Matematičkom odsjeku PMF-a.

U travnju 2014. godine zaposlio sam se u Iskon Internetu u odjelu Održavanja pristupne mreže. Tamo sam radio na dijagnostici i uklanjanju kvarova na uslugama korisnika. Također sam napravio neke interne web aplikacije koje su automatizirale neke procese dijagnostike. Od ožujka 2017. godine zaposlen sam u Photomathu kao softverski inženjer. Glavni zadaci su mi razvoj matematičkog solvera i algoritama koji simuliraju ljudsko izračunavanje matematičkih zadataka. Još sam honorarno radio za tvrtku Visage-Technologies kao suradnik u projektima koji se koriste za izradu aplikacija temeljenih na prepoznavanju ljudskog lica.