UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO
FAKULTETA ZA MATEMATIKO IN FIZIKO

Peter Nose

UČINKOVITA ARITMETIKA NA ELIPTIČNIH KRIVULJAH NAD PRAŠTEVILSKIMI OBSEGI

Diplomska naloga
na univerzitetnem študiju

Mentor: prof. dr. Aleksandar Jurišić

Ljubljana, 2008

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

Zahvala

Zahvaljujem se vsem, ki so mi kakorkoli pomagali pri tem diplomskem delu. Posebna zahvala gre mojemu mentorju prof. dr. Aleksandru Jurišiću. Iskrena hvala tudi dragima mami in očetu za vso podporo in finančno pomoč pri študiju.

Kazalo

Seznam uporabljenih kratic in simbolov

$\mathbb{F}_p$	končni obseg s p elementi
$\mathbb{Z}_p$	končni obseg, katerega elementi so cela števila po modulu p, kjer je p praštevilo
W	dolžina besede/registra
$\left(\frac{a}{p}\right)$	Jacobijev simbol
$\mathcal{O}$	točka v neskončnosti

Povzetek

V tem delu bomo predstavili osnove, potrebne za razumevanje osnovnih računskih operacij v praštevilskih obsegih. Le ti so posebno zanimivi za kriptosisteme implementirane v programski opremi. Sem sodijo tudi tisti kriptosistemi, ki temeljijo na težavnosti diskretnega logaritma na eliptični krivulji. Osrednji del te naloge je študija učinkovite implementacije aritmetike na eliptičnih krivuljah nad praštevilskimi obsegi. Tu imamo v mislih predvsem računanje večkratnika točke, katerega je moč izračunati s pomočjo seštevanja in podvajanja točk. Predstavili bomo celoten postopek implementacije, posamezne izboljšave algoritmov in praktične rezultate testiranj.

Ključne besede:

praštevilski obseg, eliptične krivulje, afine koordinate, projektivne koordinate, množenje s skalarjem, NAF predstavitev.

Abstract

In this thesis we give a brief introduction to arithmetics of prime fields. These are very attractive for software-based implementations of cryptosystems such as those based on the difficulty of the discrete logarithm on elliptic curves. The main topic of this thesis are efficient implementations of arithmetics on elliptic curves over prime fields. We focus on efficient scalar multiplication of points on elliptic curves using point addition and point doubling. We also discus design, implementation, optimization and performance analysis of used algorithms.

Keywords:

prime field, elliptic curves, affine coordinates, projective coordinates, scalar multiplication, NAF representation.

Poglavje 1

Uvod

V sodobnem svetu smo vse bolj obdani z različnimi informacijami. Nagli informacijski in tehnološki razvoj nam omogoča pridobivanje in shranjevanje velike zaloge podatkov. Večina teh podatkov je nepomembnih in nas njihova varnost ne zanima, vendar obstajajo tudi podatki, ki morajo za vsako ceno ostati tajni. Nekoč smo take podatke skrbno varovali v zavarovanih sefih oziroma trezorjih, do katerih nepooblaščene osebe niso imele dostopa. V dobi računalništva se tega pristopa ne poslužujemo, saj se je dobro razvila kriptografija javnih ključev. Slednja je svoj pravi preporod doživela z razvojem interneta.

Kriptografija nam omogoča varen prenos podatkov preko nekega nezaščitenega kanala. Primer takega kanala je internet, kjer na vsakem koraku obstaja možnost, da nam nekdo prisluškuje. Seveda pa varni prenos podatkov ni vse kar nam kriptografija omogoča. Z njo lahko preverjamo tudi celovitost podatkov, overjamo pošiljatelja , preprečujemo tajenje in nadzorujemo dostop.

Kriptosistem z eliptičnimi krivuljami je eden najpomembnejših delov kriptografije z javnimi ključi. Izbrano stopnjo varnosti nam v primerjavi s kriptosistemom RSA nudi v veliko krajšem času in z manjšimi certifikati. Zato ni čudno, da prevzema pobudo med kriptosistemi z javnimi ključi.

V tem diplomskem delu bomo preverili osnovne operacije na eliptičnih krivuljah, ki so ključnega pomena za hitrost kriptosistema. Osredotočili se bomo na eliptične krivulje definirane nad praštevilskimi obsegi. V ta namen bomo v 2. poglavju vpeljali končne obsege, v 3. preučili osnovne aritmetične operacije v praštevilskem obsegu in v 4. se spoznali z osnovnimi definicijami eliptične krivulje. Sledil bo opis seštevanja in odštevanja točk, s katerima bomo računali večkratnik točke. Implementirali ga bomo na več različnih načinov ter na kratko analizirali in komentirali hitrosti posameznih operacij.

Poglavje 2

Osnove

V tem poglavju bom predstavili osnove, potrebne za razumevanje teorije eliptičnih krivulj. Definirali bomo enostavnejše algebraične strukture, preko katerih bomo prišli do definicije praštevilskih obsegov. Ti bodo posebno pomembni za analizo učinkovitosti kriptosistema. Predstavili bomo tudi modularno aritmetiko, katero bomo v 3. poglavju uporabili za računanje v praštevilskem obsegu. Nad slednjim bomo v 4. poglavju definirali tudi eliptične krivulje.

2.1 Modularna aritmetika

Naravno število b je *delitelj* naravnega števila a, če obstaja tako naravno število k, da velja $a = k \cdot b$. Če naravni števili a in b, $a > b$, nista v relaciji deljivosti, se deljenje števila a s številom b ne izide. V takem primeru pri deljenju dobimo nek ostanek, ki je naravno število manjše od delitelja. Deljenje poljubnih dveh naravnih števil a in b, $a \geq b$, lahko zapišemo kot $a = k \cdot b + r$. Število a imenujemo *deljenec*, b *delitelj*, k *kvocient* in r *ostanek*.

Primer 2.1. Število 2 je delitelj števila 10, saj se da slednjega zapisati kot $10 = 2 \cdot 5$. Število 3 ni v relaciji deljivosti s številom 10, saj ne obstaja nobeno naravno število k, da bi veljalo $10 = 3 \cdot k$. Vseeno lahko število 10 delimo s 3 in dobimo ostanek 1.

Modularna aritmetika temelji na nekem vnaprej določenem številu, ki ga bomo poimenovali *modul*. Modul je fiksno celo število, navadno večje od 1. Glavna operacija modularne aritmetike je *modularna redukcija* po modulu m. Če želimo število a reducirati po modulu m (na njemu opraviti modularno redukcijo), potem to število delimo z m in vrnemo ostanek r. Ker je število r

ostanek pri deljenju s številom m, velja $0 \leq r < m$. Pišemo

$$r = a \mod m.$$

Primer 2.2.

$$\begin{aligned} 8 &= 8 \mod 10 \\ 18 &= 8 \mod 10 \\ -2 &= 8 \mod 10 \\ 10 &= 0 \mod 10 \end{aligned}$$

Pravimo da sta števili a in b *kongruentni* po modulu m, če dajeta enak ostanek pri deljenju z m. Pišemo

$$a \equiv b \pmod m.$$

Operacije *seštevanje, odštevanje, množenje* in *potenciranje* lahko prevedemo v modularno aritmetiko tako, da končni rezultat operacije reduciramo po modulu.

Primer 2.3.

$$\begin{aligned} 5 + 6 = 11 &\equiv 1 \pmod{10} \\ 5 - 6 = -1 &\equiv 9 \pmod{10} \\ 5 \cdot 6 = 30 &\equiv 0 \pmod{10} \\ 5^6 = 15625 &\equiv 5 \pmod{10} \end{aligned}$$

Operacijo *deljenje* v modularni aritmetiki obravnavamo kot množenje z *inverzom*. Inverz elementa b je tak element b^{-1}, da velja $b \cdot b^{-1} \equiv 1 \pmod m$. Torej $a/b = a \cdot b^{-1} \pmod m$. Naj omenimo še, da nima vsako število nujno inverza, zato deljenje z nekaterimi števili ni mogoče.

Primer 2.4. Iz $3 \cdot 7 = 21 \equiv 1 \pmod{10}$ sledi, da je 7 inverzni element števila 3. Od tod sledi

$$5/3 = 5 \cdot 3^{-1} = 5 \cdot 7 = 35 \equiv 5 \pmod{10}.$$

Število 2 po modulu 10 nima inveznega elementa, zato deljenje ni mogoče.

Izrek 2.1. *Števili a in b sta kongruentni po modulu m natanko tedaj, ko je njuna razlika deljiva z m.*

Dokaz. ($\Rightarrow$) Če sta števili a in b kongruentni po modulu m, potem obe števili vrneta isti ostanek r pri deljenju z m. Torej $a = k_a m + r$ in $b = k_b m + r$. Od tod sledi

$$a - b = k_a m + r - k_b m - r = (k_a - k_b)m \Rightarrow m \mid a - b,$$

$$b - a = k_b m + r - k_a m - r = (k_b - k_a)m \Rightarrow m \mid b - a.$$

($\Leftarrow$) Naj bo $a = k_a m + r_a$ in $b = k_b m + r_b$, kjer sta r_a in r_b ostanka števil a in b pri deljenju z m. Potem velja

$$m \mid a-b = k_a m+r_a-k_b m-r_b = (k_a-k_b)m+(r_a-r_b) \Rightarrow m \mid r_a-r_b \Rightarrow r_a = r_b,$$

$$m \mid b-a = k_b m+r_b-k_a m-r_a = (k_b-k_a)m+(r_b-r_a) \Rightarrow m \mid r_b-r_a \Rightarrow r_b = r_a.$$

□

Izrek 2.2. *Naj bodo $a, u, v, p \in \mathbb{N}$, $p \nmid a$ in naj bo $au \equiv av \pmod{p}$. Potem*

$$u \equiv v \pmod{p}.$$

Dokaz. Iz enačbe $au \equiv av \pmod{p}$ po izreku 2.1 sledi, da $p \mid au-av = a(u-v)$. Ker p ne deli števila a mora deliti $u - v$. Zopet uporabimo izrek 2.1 in dobimo $u \equiv v \pmod{p}$. □

Izrek 2.3. (Fermat) *Naj bo p praštevilo. Potem za vsak $a \in \mathbb{N}$, $p \nmid a$ velja*

$$a^{p-1} \equiv 1 \pmod{p}. \tag{2.1}$$

Dokaz. Naj bo $a \in \mathbb{N}$ in $a \geq p$. Potem ga lahko zapišemo kot $a = \hat{a} + kp$, kjer je $1 \leq \hat{a} < p$. Od tod sledi

$$a^{p-1} = (\hat{a} + kp)^{p-1} = \hat{a}^{p-1} + p(\ldots) \equiv \hat{a}^{p-1} \pmod{p}.$$

Tako lahko predpostavimo, da je število a manjše od p. Naj bosta k in ℓ celi števili in naj velja $1 \leq k < \ell < p$. Potem

$$ka \not\equiv \ell a \pmod{p}. \tag{2.2}$$

Zgornjo enačbo bomo dokazali s protislovjem. Naj bo $ka \equiv \ell a \pmod{p}$. Potem lahko po izreku 2.2 obe strani enačbe krajšamo z a in dobimo $k \equiv \ell \pmod{p}$. Ker sta števil k in ℓ različni in manjši od p pridemo do protislovja.

Števila ka za $k \in \{1, 2, \ldots, (p-1)\}$ reduciramo po modulu p. Ker je p praštevilo, $a, k < p$, nobeno število ka ni deljivo s p. Ostanek pri deljenju s

p je tako element množice $\{1, 2, \ldots, p-1\}$. Iz enačbe 2.2 sledi, da dajo vsa števila ka različni ostanek pri deljenju s p. Torej modularna redukcija števil $a, 2a, \ldots, (p-1)a$ ravno pokrije vse elemente iz $\{1, \ldots, p-1\}$. Od tod sledi

$$a \cdot 2a \cdot 3a \cdot \ldots \cdot (p-1)a \equiv 1 \cdot 2 \cdot \ldots \cdot p-1 \pmod{p}.$$

Če levo stran enačbe preuredimo, dobimo

$$a^{p-1} \cdot 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1) \equiv 1 \cdot 2 \cdot \ldots \cdot p-1 \pmod{p}.$$

Ker število $1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1)$ ni deljivo s p, lahko po izreku 2.2 obe strani enačbe delimo in dobimo

$$a^{p-1} \equiv 1 \pmod{p}.$$

□

2.2 Grupa

Binarna operacija $\circ$ definirana na množici M je preslikava, ki vsakemu paru $(x, y) \in M \times M$ priredi natanko en element $x \circ y \in M$. Binarna operacija je *komutativna*, če velja $x \circ y = y \circ x$ za vsak $x, y \in M$. Če je nad množico M definirana vsaj ena operacija, pravimo da ima množica M *algebraično strukturo*.

Definicija 2.1. Naj bo M neprazna množica in $\circ$ binarna operacija definirana na množici M. Potem je algebraična struktura $(M, \circ)$ *grupa*, če velja:

- $\circ$ je *notranja* operacija, torej za vse elemente $x, y \in M$ velja $(x \circ y) \in M$,
- $\circ$ je *asociativna* operacija, torej za vse elemente $x, y, z \in M$ velja

$$(x \circ y) \circ z = x \circ (y \circ z),$$

- za operacijo $\circ$ obstaja *enota* e, to je tak element, da velja $x \circ e = e \circ x = x$ za vsak element $x \in M$,
- za vsak element $x \in M$ obstaja *inverz* y, to je tak element, da velja $x \circ y = y \circ x = e$.

Grupo $(M, \circ)$ bomo v nadaljevanju označevali z G.

Primer 2.5. Naj bo $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$ in $+_6$ operacija seštevanja po modulu 6. Potem je $(\mathbb{Z}_6, +_6)$ grupa. Preverimo še potrebne pogoje:

- $+_6$ je notranja operacija, saj seštevanje po modulu 6 vedno vrne rezultat iz $\mathbb{Z}_6$,
- $+_6$ je asociativna operacija, saj je seštevanje po modulu 6 asociativno,
- 0 je enota za operacijo $+_6$,
- vsak element $x \in \mathbb{Z}_6$ ima inverz $y = 6 - x \mod 6$.

Izrek 2.4. *Vsaka grupa G ima natanko eno enoto in vsak element grupe G ima natanko en inverz.*

Dokaz. Iz definicije grupe sledi, da ima vsaka grupa vsaj eno enoto. Denimo da obstaja grupa z dvema različnima enotama. Označili ju bomo z e_1 in e_2. Potem po definiciji enote velja $e_1 \circ e_2 = e_1 = e_2$, kar pa je v protislovju s predpostavko, da sta enoti e_1 in e_2 različni.

Po definiciji ima vsak element vsaj en inverz. Denimo, da ima element $x \in G$ dva različna inverza. Označili ju bomo z i_1 in i_2. Potem velja

$$x \circ i_1 = i_1 \circ x = e \quad \text{in} \quad x \circ i_2 = i_2 \circ x = e,$$

in od tod še

$$i_1 = i_1 \circ e = i_1 \circ (x \circ i_2) = (i_1 \circ x) \circ i_2 = e \circ i_2 = i_2,$$

kar pa je v protislovju s predpostavko, da sta inverza i_1 in i_2 različna. □

Pravimo da je grupa G *komutativna* oziroma *Abelova*, če je binarna operacija $\circ$ komutativna. Število elementov grupe imenujemo tudi *red grupe*. Grupa G je *končna*, če je njen red končen.

Definicija 2.2. $H \subseteq G$ je *podgrupa* grupe G, če:

- za vse elemente $x, y \in H$ velja $(x \circ y) \in H$,
- $e \in H$,
- za vsak element $x \in H$ velja $x^{-1} \in H$.

Primer 2.6. Naj bo $x \in G$ in G grupa z operacijo množenje. Potem je $\langle x \rangle = \{x^n \mid n \in \mathbb{Z}\}$ podgrupa grupe G. Prav tako je $\langle x \rangle = \{nx \mid n \in \mathbb{Z}\}$ podgrupa grupe G, če je ta definirana z operacijo seštevanja.

Definicija 2.3. Grupa G je *ciklična*, če obstaja tak element $x \in G$, da je $\langle x \rangle = G$. Takemu elementu x pravimo *generator* grupe G.

Primer 2.7. Grupa $(\mathbb{Z}_6, +_6)$ je ciklična grupa z generatorjema 1 in 5. Preverimo, če je 5 res generator grupe $(\mathbb{Z}_6, +_6)$.

$$\begin{aligned}
&1 \cdot 5 \equiv 5 \pmod 6,\\
&2 \cdot 5 = 5 + 5 = 10 \equiv 4 \pmod 6,\\
&3 \cdot 5 = 5 + 5 + 5 = 15 \equiv 3 \pmod 6,\\
&4 \cdot 5 = 5 + 5 + 5 + 5 = 20 \equiv 2 \pmod 6,\\
&5 \cdot 5 = 5 + 5 + 5 + 5 + 5 = 25 \equiv 1 \pmod 6,\\
&6 \cdot 5 = 5 + 5 + 5 + 5 + 5 + 5 = 30 \equiv 0 \pmod 6,\\
&7 \cdot 5 = 5 + 5 + 5 + 5 + 5 + 5 + 5 = 35 \equiv 5 \pmod 6.
\end{aligned}$$

Ker je $\langle x \rangle = \{5, 4, 3, 2, 1, 0\} = G$, je 5 generator grupe G.

Naj bo G grupa in $x \in G$. Pravimo da ima element x *končni red*, če je množica $\langle x \rangle$ končna. Red elementa x je moč množice $\langle x \rangle$, torej najmanjše naravno število n, za katerega velja (a) $x^n = e$, če je G definirana z operacijo množenja oziroma (b) $nx = e$, če je G definirana z operacijo seštevanja. Če množica $\langle x \rangle$ ni končna, pravimo da je element x *reda nič*.

2.3 Kolobar

Definicija 2.4. Algebraična struktura $(M, +, \circ)$ je *kolobar*, če velja:

- $(M, +)$ je Abelova grupa,
- binarna operacija $\circ$ je asociativna z enoto 1, ki se razlikuje od enote 0 za operacijo $+$,
- za poljubne elemente $x, y, z \in M$ velja *distributivnost*, torej

$$\begin{aligned}
a \circ (b + c) &= a \circ b + a \circ c,\\
(a + b) \circ c &= a \circ c + b \circ c.
\end{aligned}$$

Kolobar $(M, +, \circ)$ bomo v nadaljevanju označevali s K, operaciji $+$ in $\circ$ pa bomo poimenovali seštevanje in množenje. Kolobar K je *komutativen*, če je množenje komutativno.

Primer 2.8. Če $\mathbb{Z}$ množica celih števil, $+$ operacija seštevanja in $\cdot$ operacija množenja, potem je $(\mathbb{Z}, +, \cdot)$ komutativen kolobar.

Naj bo K kolobar. Pravimo, da je element $x \in K$ *obrnljiv*, če obstaja element $y \in K$, da velja $x \circ y = y \circ x = 1$. Takemu elementu pravimo *inverz* in ga označimo z oznako x^{-1}. Množico vseh obrnljivih elementov iz kolobarja K označimo z K^*.

Če v kolobarju K za neničelna elementa $x, y \in K$ velja $x \circ y = 0$, potem je x *levi delitelj niča* in y *desni delitelj niča*. V komutativnem kolobarju so zaradi komutativnosti operacije $\circ$ levi in desni delitelji niča enaki in jim pravimo *delitelji niča*.

Karakteristika kolobarja K, označimo jo z $\texttt{char}(K)$, je najmanjše naravno število n, za katerega velja $n \circ 1 = \underbrace{1 + 1 + \ldots + 1}_{n \text{ krat}} = 0$, kjer je 1 enota za množenje in 0 enota za seštevanje.

2.4 Končni obseg

Definicija 2.5. Komutativnemu kolobarju z enoto 1, kjer je vsak neničeln element obrnljiv, pravimo *obseg*.

Primer 2.9. Naj bo $\mathbb{Q}$ množica racionalnih števil, $+$ operacija seštevanja in $\cdot$ operacija množenja. Potem je kolobar $(\mathbb{Q}, +, \cdot)$ obseg.

Izrek 2.5. *V obsegu ni deliteljev niča.*

Dokaz. Denimo, da v obsegu O obstaja delitelj niča x. Ker je x delitelj niča obstaja $y \in O$, za katerega velja $x \circ y = 0$. Ker je O obseg obstaja $y^{-1} \in O$, za katerega velja $y \circ y^{-1} = 1$. Potem velja

$$x = x \circ 1 = x \circ (y \circ y^{-1}) = (x \circ y) \circ y^{-1} = 0 \circ y^{-1} = 0,$$

kar pa je v protislovju s predpostavko, da je x delitelj niča. □

Izrek 2.6. *Karakteristika obsega je lahko* 0 *ali p, kjer je p praštevilo.*

Dokaz. Denimo, da karakteristika obsega O ni praštevilo in ni 0. Potem je $\text{char}(O) = n = rs$ in velja $1 < r, s < n$. Ker je n najmanjše število, za katerega velja $n \circ 1 = \underbrace{1 + 1 + \ldots + 1}_{n \text{ krat}} = 0$, velja:

$$r \circ 1 = \underbrace{1 + 1 + \ldots + 1}_{r \text{ krat}} \neq 0,$$

$$s \circ 1 = \underbrace{1 + 1 + \ldots + 1}_{s \text{ krat}} \neq 0,$$

$$0 = n \circ 1 = (r \circ s) \circ 1 = (r \circ 1) \circ (s \circ 1) \Rightarrow r \circ 1 = 0 \text{ ali } s \circ 1 = 0.$$

Prispeli smo do protislovja, zato mora biti število n praštevilo. □

Obseg je *končen*, če vsebuje končno mnogo elementov. Končni obseg sestavlja množica elementov F ter dve binarni operaciji seštevanje in množenje, ki sta definirani na njej. S pomočjo teh dveh operacij lahko definiramo še dve novi operaciji, ki sta pravzaprav njuni inverzni operaciji.

- *Binarna operacija odštevanja*
  Dva elementa $x, y \in F$ odštejemo tako, da elementu x prištejemo *negativni element* od y. Negativni element $-y \in F$ je enoličen element, za katerega velja $y + (-y) = 0$. Operacijo odštevanja zapišemo kot $x - y = x + (-y)$.

- *Binarna operacija deljenja*
  Dva elementa $x, y \in F$, kjer $y \neq 0$, delimo tako, da element x množimo z inverznim elementom od y. Inverzni element $y^{-1} \in F$ je enoličen element, za katerega velja $y \circ y^{-1} = 1$. Operacijo deljenja zapišemo kot $x/y = x \circ y^{-1}$.

Primer 2.10. Naj bo $\mathbb{Z}_{13} = \{0, 1, 2, \ldots, 11, 12\}$, $+_{13}$ operacija seštevanja in $\cdot_{13}$ operacija množenja po modulu 13. Potem je $(\mathbb{Z}_{13}, +_{13}, \cdot_{13})$ končni obseg, saj velja:

- $(\mathbb{Z}_{13}, +_{13})$ je Abelova grupa,
- $\cdot_{13}$ je asociativna operacija z enoto 1, ki je različna od enote 0 za operacijo $+_{13}$,
- za operaciji $+_{13}$ in $\cdot_{13}$ velja zakon distributivnosti,
- $\cdot_{13}$ je komutativna operacija,
- vsak neničeln element je obrnljiv,

$$\begin{array}{llll} 1^{-1} = 1 & 4^{-1} = 10 & 7^{-1} = 2 & 10^{-1} = 4 \\ 2^{-1} = 7 & 5^{-1} = 8 & 8^{-1} = 5 & 11^{-1} = 6 \\ 3^{-1} = 9 & 6^{-1} = 11 & 9^{-1} = 3 & 12^{-1} = 12 \end{array}$$

$$1 \cdot 1 \equiv 1 \pmod{13} \qquad 4 \cdot 10 = 40 \equiv 1 \pmod{13}$$
$$2 \cdot 7 = 14 \equiv 1 \pmod{13} \qquad 6 \cdot 11 = 66 \equiv 1 \pmod{13}$$
$$3 \cdot 9 = 27 \equiv 1 \pmod{13} \qquad 12 \cdot 12 = 144 \equiv 1 \pmod{13}$$
$$5 \cdot 8 = 40 \equiv 1 \pmod{13}$$

- $\mathbb{Z}_{13}$ vsebuje končno mnogo elementov.

Element 2 je generator $\mathbb{Z}_{13}^*$. Potence elementa 2 so:

$$2^1 \equiv 2 \pmod{13} \qquad 2^7 = 128 \equiv 11 \pmod{13}$$
$$2^2 \equiv 4 \pmod{13} \qquad 2^8 = 256 \equiv 9 \pmod{13}$$
$$2^3 \equiv 8 \pmod{13} \qquad 2^9 = 512 \equiv 5 \pmod{13}$$
$$2^4 = 16 \equiv 3 \pmod{13} \qquad 2^{10} = 1024 \equiv 10 \pmod{13}$$
$$2^5 = 32 \equiv 6 \pmod{13} \qquad 2^{11} = 2048 \equiv 7 \pmod{13}$$
$$2^6 = 64 \equiv 12 \pmod{13} \qquad 2^{12} = 4096 \equiv 1 \pmod{13}$$

Naj bo p praštevilo, $\mathbb{Z}_p = \{0, 1, 2, \ldots, p\}$, $+$ operacija seštevanja in $\cdot$ operacija množenja po modulu p. Potem je $(\mathbb{Z}_p, +, \cdot)$ praštevilski končni obseg. Označili ga bomo kar z $\mathbb{Z}_p$.

Končni obsegi slovijo po svoji lepi algebraični strukturi in so zato zelo priljubljeni. Matematiki jih preučujejo že vrsto let in okrog njih se je razvila zelo obsežna teorija. Pomembni so na številih področjih matematike, predvsem v linearni in abstraktni algebri, v teoriji števil in algebraični geometriji, ter tudi v računalništvu, statistiki, informatiki in elektrotehniki.

Končni obsegi se nahajajo bližje nas, kot si mi lahko predstavljamo. V vsakodnevnem življenju jih lahko najdemo v CD/DVD predvajalnikih, v telefonih in modemih, kjer odkrivajo in popravljajo napake, ki se pojavijo med prenosom podatkov. S pomočjo končnih obsegov so v prakso prišle tudi eliptične krivulje, katere bomo podrobneje spoznali v nadaljevanju.

2.5 Legendrov in Jacobijev simbol

Definicija 2.6. Število a je *kvadratni ostanek* po modulu p, če obstaja tako število r, da velja

$$a \equiv r^2 \pmod{p},$$

sicer število a *ni kvadratni ostanek* po modulu p.

Če je število a kvadratni ostanek po modulu p, potem njegovemu korenu ustrezata dve števili r in $-r$. Le v primeru, ko je $a \equiv 0 \pmod{p}$ dobimo eno rešitev $r = 0$. Od tod lahko zaključimo, da je kvadratnih ostankov v $\mathbb{F}_p^*$ natanko $(p-1)/2$.

Primer 2.11. Če si izberemo $p = 7$, potem so kvadratni ostanki $\{1, 2, 4\} \in \mathbb{Z}_7^*$ z enoto 0.

r	0	1	2	3	4	5	6
r^2	0	1	4	9	16	25	36
$r^2 \pmod 7$	0	1	4	2	2	4	1

Legendrov simbol $\left(\frac{a}{p}\right)$ je število rešitev enačbe $a \equiv r^2 \pmod{p}$ minus 1, to je

$$\left(\frac{a}{p}\right) = \begin{cases} -1, & a \text{ ni kvadratni ostanek po modulu } p, \\ 0, & a \equiv 0 \pmod{p}, \\ 1, & a \text{ je kvadratni ostanek po modulu } p. \end{cases}$$

Posplošitev Legendrovega simbola je *Jacobijev simbol*. Ta je za sestavljeno število $p = p_1^{v_1} \cdot p_2^{v_2} \cdot \ldots \cdot p_k^{v_k}$ enak

$$\left(\frac{a}{p}\right) = \left(\frac{a}{p_1}\right)^{v_1} \left(\frac{a}{p_2}\right)^{v_2} \ldots \left(\frac{a}{p_k}\right)^{v_k}$$

Izrek 2.7. *Naj bosta p, q lihi praštevili. Potem za Jacobijeve simbole velja:*

(i)
$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}$$

(ii)
$$\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$$

(iii)
$$\left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}$$

(iv)
$$\left(\frac{a}{p}\right) = \left(\frac{a \pm bp}{p}\right)$$

(v)

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$$

(vi)

$$\left(\frac{a}{pq}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right)$$

(vii)

$$\left(\frac{q}{p}\right)\left(\frac{p}{q}\right) = (-1)^{(p-1)(q-1)/4}$$

Dokaz.

(i) Če $p \mid a$, potem je $a^{(p-1)/2}$ večkratnik števila p oziroma $\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \equiv 0 \pmod{p}$. Denimo da $p \nmid a$. Potem velja

$$a^{(p-1)/2} \equiv -1 \pmod{p} \quad \text{ali} \quad a^{(p-1)/2} \equiv 1 \pmod{p}.$$

Naj bo a kvadratni ostanek po modulu p. Potem obstaja tak b, da velja $b^2 \equiv a \pmod{p}$. Po izreku (2.1) lahko obe strani potenciramo na $(p-1)/2$ in dobimo

$$(b^2)^{(p-1)/2} \equiv b^{p-1} \equiv 1 \equiv a^{(p-1)/2} \pmod{p}.$$

Dokažimo še, da če je $a^{(p-1)/2} \equiv 1 \pmod{p}$, potem je a nujno kvadratni ostanek. Naj bo g generator grupe $\mathbb{Z}_p$. Potem obstaja tak $i \in \mathbb{N}$, da je $g^i \equiv a \pmod{p}$. Če obe strani potenciramo na $(p-1)/2$ dobimo

$$g^{i(p-1)/2} \equiv a^{(p-1)/2} \equiv 1 \pmod{p}.$$

Po (2.1) mora veljati da $p-1 \mid i(p-1)/2$ oziroma i je sod. Tako je $a \equiv g^i \equiv g^{2i'} \equiv g^{i'^2} \pmod{p}$ kvadratni ostanek.

(ii) Sledi neposredno iz *(i)*.

(iii) Dokaz se nahaja v [5, 187].

(iv) $\left(\frac{a \pm bp}{p}\right) \equiv (a \pm bp)^{(p-1)/2} \equiv a^{(p-1)/2} + p(\ldots) \equiv \left(\frac{a}{p}\right) \pmod{p}$.

(v)

$$\left(\frac{ab}{p}\right) \equiv (ab)^{\frac{(p-1)}{2}} \equiv a^{\frac{(p-1)}{2}} b^{\frac{(p-1)}{2}} \equiv \left(\frac{a}{p}\right)\left(\frac{b}{p}\right) \ (mod\ p)$$

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right) + kp$$

Ker je k celo število, p liho praštevilo in vrednosti Jacobijevih simbolov elementi množice $\{-1, 0, 1\}$, mora veljati, da je $k \equiv 0$ oziroma

$$\left(\frac{ab}{p}\right) \equiv \left(\frac{a}{p}\right)\left(\frac{b}{p}\right).$$

(vi) Dokaz se nahaja v [5, 183].

□

Poglavje 3

Aritmetika v praštevilskem obsegu $\mathbb{Z}_p$

V tem poglavju si bomo ogledali osnovne algoritme za računanje v praštevilskem obsegu $\mathbb{Z}_p$. Te algoritme bomo nato v poglavju (4) uporabili za seštevanje, odštevanje in točk na eliptični krivulji, za računanje večkratnikov točke ter za njihovo kompresijo in dekompresijo. Zaradi zahteve po čim bolj varnem kriptosistemu, moramo naše algoritme prilagoditi na računanje z velikimi števili. Varnost kriptosistema je namreč odvisna od računske zahtevnosti matematičnega problema, katerega zahtevnost raste z velikostjo števil. Po eni strani tako z večanjem števil pridemo do večje varnosti kriptosistema, po drugi strani pa se zaradi večjih števil povečuje časovna in prostorska zahtevnost algoritmov. V nadaljevanju bodo vsi predstavljeni algoritmi prilagojeni programski implementaciji.

3.1 Predstavitev števil

Naj bo $b \geq 2$ celo število, ki ga bomo poimenovali *osnova* oziroma *baza*. Potem se da vsako število $a \geq 0$ zapisati kot

$$a = a_{n-1}b^{n-1} + \ldots + a_1 b + a_0,$$

kjer je $0 \leq a_i < b \; \forall i$. Tak zapis imenujemo *predstavitev števila a z bazo b* in ga bomo označili z $(a_{n-1}, \ldots, a_1, a_0)_b$. Števila a_i predstavljajo števke števila a, kjer je a_{n-1} najbolj pomembna (angl. *most significant bit*) in a_0 najbolj nepomembna števka (angl. *least significant bit*). Velikost števila a bomo označili z $|a|_b$ in je enaka največjemu številu i, za katerega velja $a_i > 0$. Osnovne značilnosti take predstavitve so:

- predstavitev števila 0 z bazo b je vedno $(0)_b$,
- če predstavitev števila a z bazo b razširimo z leve z dodatnimi ničlami (angl. *padding*), potem ne spremenimo števila

$$(a_{n-1}, \ldots, a_1, a_0)_b = (0, \ldots, 0, a_{n-1}, \ldots, a_1, a_0)_b,$$

- v računalništvu se v večini primerov za bazo izbere potenco števila 2, tako da števila a_i zavzamejo samo 2 vrednosti (število a_i imenujemo tudi *bit*).

Današnji računalniki ne znajo direktno upravljati samo z enim samim bitom. Najmanjšo količino bitov, katero lahko računalnik naslavlja v pomnilniku je *bajt* in to je v večini primerov osem bitov. Na srečo pa lahko procesor s pomočjo registrov upravlja z večimi bajti hkrati. Velikost registra imenujemo *beseda* in je ena izmed pomembnejših podatkov procesorja. Današnji računalniki uporabljajo besede dolžine 64 oziroma 32 bitov, medtem ko manjše naprave, kot so pametne kartice, operirajo z 8 oziroma 16 bitnimi besedami. V naših algoritmih bomo predpostavili, da imamo W-bitno arhitekturo, kjer je W večkratnik števila 8. Z eno besedo bomo lahko tako predstavili 2^W različnih števil. Bite W-bitne besede bomo označevali s števili od 0 do $W-1$, kjer je 0 najbolj desni in $W-1$ najbolj levi bit besede.

Naj bo n oziroma

$$|a|_{2^W} = \lceil \log_{2^W} p \rceil = \left\lceil \frac{\log_2 p}{W} \right\rceil$$

dolžina zapisa števila p z bazo 2^W. Torej ga lahko predstavimo z vsaj n besedami. Ker so elementi obsega $\mathbb{Z}_p$ števila manjša od p, nam bo zadoščalo, če bomo vsako število $a = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0) \in \mathbb{Z}_p$ predstavili s tabelo n-tih besed.

a_{n-1}	a_{n-2}	a_{n-3}	$\ldots$	a_2	a_1	a_0

Tabela 3.1: Predstavitev števila $a = a_{n-1}2^{(n-1)W} + \ldots + a_2 2^{2W} + a_1 2^W + a_0$

Primer 3.1. Vzemimo število $a = 1128103691808033$ in ga zapišimo z bazo 2.

$$a = (\underbrace{1000000001000000001}_{262657}, \underbrace{00011011110100011110110100100001}_{466742561})_2$$

Predstavitev števil z bazo 2, bi bila na računalniku z 32-bitno arhitekturo nesmiselna, saj lahko procesor operira z 32 biti hkrati. Števila bi zato raje predstavili z bazo 2^{32}, torej bi število a zapisali kot $a = (262657, 466742561)_{2^{32}}$.

3.2 Notacija

Za lažje razumevanje kode bomo uvedli naslednjo notacijo. Za $0 \leq x < 2^{W+1}$ bo prirejanje $(\varepsilon, y) \leftarrow x$ pomenilo

$$y = x \mod 2^W$$

$$\varepsilon = \begin{cases} 1, & y \geq 2^W \\ 0, & \text{sicer} \end{cases}$$

Naj bo $0 \leq x, y < 2^W$ in $\varepsilon' \in \{0, 1\}$. Potem lahko

- $w = x + y + \varepsilon'$ zapišemo kot $w = \varepsilon 2^W + z$. Število ε lahko zasede samo dve vrednosti $\{0, 1\}$ in mu pravimo *prenosni* bit (angl. *carry bit*).
- $w = x - y - \varepsilon'$ zapišemo kot $w = -\varepsilon 2^W + z$. Število ε lahko zasede samo dve vrednosti $\{0, 1\}$ in mu pravimo *sposojeni* bit (angl. *borrow bit*).

Kadar bomo želeli predstaviti število a z bazo 2^W, bomo v zapisu bazo spustili. Tako bo zapis $a = (a_{n-1}, \ldots, a_0)$ enakovreden zapisu $a = (a_{n-1}, \ldots, a_0)_{2^W}$.

3.3 Seštevanje in odštevanje

Seštevanje in odštevanje sta najbolj osnovni operaciji, zato je pomembno, da je njuna implementacija hitra. Seštevanje in odštevanje bomo implementirali na isti način, kot ju izvajamo v vsakdanjem življenju. Vsako število zapišemo v izbrani bazi (npr. z bazo 10) in začnemo seštevati števke obeh števil od desne proti levi. Če vsota dveh števk presega bazo, potem tej vsoti odštejemo bazo, vsoti naslednjih dveh števk pa prištejemo 1. Tako nadaljujemo, dokler ne pridemo do konca predstavitve obeh števil. Algoritma za seštevanje in odštevanje za bazo uporabljala število 2^W.

Oba algoritma (1) in (2) nad vsakim parom besed (a_i, b_j) opravita eno osnovno operacijo. Ker so števila dolga n besed, je zahtevnost obeh algoritmov enaka $O(n)$.

Kadar imamo opravka z fiksno dolžino števil, se je potrebno obeh zank v algoritmu (1) in (2) znebiti. To lahko preprosto storimo tako, da vsebino zanke

Algoritem 1 Seštevanje

VHOD: $a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$
IZHOD: $c = a + b \pmod{2^{nW}}$, $c = (c_{n-1}, \ldots, c_0)$, prenosni bit ε
1: $(\varepsilon, c_0) \leftarrow a_0 + b_0$
2: **for** $i = 1$ to $n - 1$ **do**
3: $\quad (\varepsilon, c_i) \leftarrow a_i + b_i + \varepsilon$
4: **end for**
5: **return** (c, ε)

Algoritem 2 Odštevanje

VHOD: $a = (a_{n-1}, \ldots, a_0), b = (b_{n-1}, \ldots, b_0)$
IZHOD: $c = a - b \pmod{2^{nW}}$, $c = (c_{n-1}, \ldots, c_0)$, izposojni bit ε
1: $(\varepsilon, c_0) \leftarrow a_0 - b_0$
2: **for** $i = 1$ to $n - 1$ **do**
3: $\quad (\varepsilon, c_i) \leftarrow a_i - b_i - \varepsilon$
4: **end for**
5: **return** (c, ε)

prekopiramo za vsak i posebej. Če je velikost zank v času prevajanja znana, potem lahko odpravljanje zank opravi tudi prevajalnik (npr: *gcc* prevajalnik zagnan s parametrom *-funroll-loops*).

Tretjo vrstico algoritma (1) in (2) lahko učinkovito implementiramo na sodobnih procesorjih. Ti namreč v svojem naboru ukazov vsebujejo ukaze, ki pri seštevanju/odštevanju upoštevajo prenosni/sposojeni bit. Tako ni potrebno dodatno preverjanje prenosnega/sposojenega bita.

Kadar seštevamo števila različne velikosti, potem moramo manjše število z leve dopolniti s toliko ničlami, da bosta obe števili enako dolgi. Šele nato lahko obe števili seštejemo. Podoben problem se nam pojavi, ko želimo vsa števila predstaviti s tabelo n-tih besed. V primeru, da neko število zaseda $m < n$ besed prostora, potem je potrebno zgornjih $n - m$ besed tabele nastaviti na 0. Pri nekaterih programskih jezikih (npr: *C*) moramo na ta problem posebno paziti, medtem ko pri drugih (npr: *Java*) tega problema sploh ni. Ti namreč že pri inicializaciji tabele vse vrednosti nastavijo na 0. V naših algoritmih bomo predpostavili prvi tip programskih jezikov.

Primer 3.2. V desetiškem sistemu seštejmo števili $248 = (2, 4, 8)_{10}$ in $963 = (9, 6, 3)_{10}$.

$$\begin{array}{rcccc} & & 2 & 4 & 8 \\ + & & 9 & 6 & 3 \\ & 1 & 1 & 1 & 0 \\ \hline & 1 & 2 & 1 & 1 \end{array}$$

Seštejmo še števili

$a = 22685491143908196737433856829760907946$ in

$b = 90741964528095889430108778316114550237.$

Njuni predstavitvi v bazi 2^{32} sta:

$a = (\texttt{0x11111111,0x44444444,0x77777777,0xAAAAAAAA})_{2^{32}}$ in

$b = (\texttt{0x44444444,0x77777777,0xAAAAAAAA,0xDDDDDDDD})_{2^{32}}$.

Ko imamo obe števili predstavljeni v bazi, ju lahko seštejmo.

	0x11111111	0x44444444	0x77777777	0xAAAAAAAA
+	0x44444444	0x77777777	0xAAAAAAAA	0xDDDDDDDD
	0	1	1	0
	0x55555555	0xBBBBBBBC	0x22222222	0x88888887

Bralec se lahko sam prepriča, da izračunana vsota v desetiškem sistemu ustreza številu 113427455672004086167542635145875458183.

3.4 Seštevanje in odštevanje v $\mathbb{Z}_p$

Seštevanje in odštevanje v $\mathbb{Z}_p$ poteka podobno kot navadno seštevanje. Razlika je le v tem, da vsoto/razliko števil računamo še po modulu p. Ker bomo predpostavili, da sta obe vhodni števili na intervalu $[0, p-1]$, bo za modularno redukcijo potrebno le eno seštevanje/odštevanje.

Zahtevnost obeh algoritmov je enaka linearni zahtevnosti seštevanja oziroma odštevanja, torej $O(n)$.

3.5 Množenje v $\mathbb{Z}_p$

Množenje v $\mathbb{Z}_p$ bomo realizirali tako, da bomo vhodni števili najprej zmnožili z algoritmom (5) oziroma (6), zmnožek pa nato z algoritmom (9) reducirali po modulu p.

Algoritem 3 Seštevanje v $\mathbb{Z}_p$

VHOD: $0 \leq a, b < p$, $a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $c = a + b \pmod{p}$, $c = (c_{n-1}, \ldots, c_0)$

```
1: uporabi algoritem (1) za izračun c = a + b (mod 2^{nW}) in prenosnega bita ε
2: if ε = 1 ∨ c ≥ p then
3:     uporabi algoritem (2) za izračun c = c − p (mod 2^{nW})
4: end if
5: return c
```

Algoritem 4 Odštevanje v $\mathbb{Z}_p$

VHOD: $0 \leq a, b < p$, $a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $c = a - b \pmod{p}$, $c = (c_{n-1}, \ldots, c_0)$

```
1: uporabi algoritem (2) za izračun c = a − b (mod 2^{nW}) in sposojenega bita ε
2: if ε = 1 then
3:     uporabi algoritem (1) za izračun c = c + p (mod 2^{nW})
4: end if
5: return c
```

Algoritem 5 Množenje 1

VHOD: $a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$
IZHOD: $c = a \cdot b$, $c = (c_{2n-1}, \ldots, c_0)$

```
1: for i = 0 to n − 1 do
2:     c_i ← 0
3: end for
4: for i = 0 to n − 1 do
5:     u ← 0
6:     for j = 0 to n − 1 do
7:         (uv) = c_{i+j} + a_i · b_j + u
8:         c_{i+j} = v
9:     end for
10:    c_{i+n} = u
11: end for
12: return c
```

V obeh algoritmih za množenje bomo predpostavili, da je (uv) združitev besed u in v. Združena beseda (uv) je velikosti $2W$ bitov in zato lahko sprejme

vsa cela števila do števila 2^{2W}. Če pogledamo sedmo vrstico algoritma (5), potem je vsota $c_{i+j} + a_i \cdot b_j + u$ velikosti $2(2^W - 1) + (2^W - 1)^2 = 2^{2W} - 1$, kar pomeni, da jo lahko zapišemo v (uv). To predpostavko lahko učinkovito izkoristimo na tistih W-bitnih procesorjih, ki omogočajo množenje dveh W-bitnih besed (npr: Intel Pentium, Sunov SPARC). Po končani operaciji se $2W$-bitni zmnožek nahaja v dveh različnih W-bitnih registrih, ki ustrezata besedama u in v.

Algoritem (5) je implementacija metode *zmnoži in seštej*. Ta na vsakem koraku zmnoži dve besedi $a_i \in (a_{n-1}, \ldots, a_0)$ in $b_j \in (b_{n-1}, \ldots, b_0)$ in ju prišteje k rezultatu c.

Primer 3.3. Zmnožimo števili $a = (0, 5, 2, 6)_{10}$ in $b = (9, 7, 1, 2)_{10}$ z algoritmom (5). Spodnja tabela nam prikazuje vrednosti spremenljivk po izvajanju osme vrstice algoritma.

i	j	a_ib_j	c_{i+j}	(uv)	u	v	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
0	0	12	0	12	1	2	-	-	-	-	0	0	0	2
	1	6	0	7	0	7	-	-	-	-	0	0	7	2
	2	42	0	42	4	2	-	-	-	-	0	2	7	2
	3	54	0	58	5	8	-	-	-	5	8	2	7	2
1	0	4	7	11	1	1	-	-	-	5	8	2	1	2
	1	2	2	5	0	5	-	-	-	5	8	5	1	2
	2	14	8	22	2	2	-	-	-	5	2	5	1	2
	3	18	5	25	2	5	-	-	2	5	2	5	1	2
2	0	10	5	15	1	5	-	-	2	5	2	5	1	2
	1	5	2	8	0	8	-	-	2	5	8	5	1	2
	2	35	5	40	4	0	-	-	2	0	8	5	1	2
	3	45	2	51	5	1	-	5	1	0	8	5	1	2
3	0	0	8	8	0	8	-	5	1	0	8	5	1	2
	1	0	0	0	0	0	-	5	1	0	8	5	1	2
	2	0	1	1	0	1	-	5	1	0	8	5	1	2
	3	0	5	5	0	5	0	5	1	0	8	5	1	2

Zapis produkta ab v obliki zmnoži in seštej zgleda kot

$$9712 \times 6 + 9712 \times 20 + 9712 \times 500 + 9712 \times 0 = 5108512.$$

Oba predlagana algoritma za množenje n besed dolgih števil potrebujeta $O(n^2)$ časa. Sredi petdesetih let je Kolmogorov trdil, da se te zahtevnosti ne da izboljšati. Kljub temu pa je leta 1960 njegov študent *Karatsuba* odkril algoritem s časovno zahtevnostjo $O = \left(n^{\log_2 3}\right)$. Karatsubinega množenja ne bomo implementirali.

Algoritem 6 Množenje 2

VHOD: $a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$
IZHOD: $c = a \cdot b$, $c = (c_{2n-1}, \ldots, c_0)$
1: $r_0 \leftarrow 0$, $r_1 \leftarrow 0$, $r_2 \leftarrow 0$
2: **for** $k = 0$ to $2n - 2$ **do**
3: **for** $\{(i, j) | i + j = k, 0 \leq i, j \leq n - 1\}$ **do**
4: $(uv) = a_i \cdot b_j$
5: $(\varepsilon, r_0) \leftarrow r_0 + v$
6: $(\varepsilon, r_1) \leftarrow r_1 + u + \varepsilon$
7: $r_2 \leftarrow r_2 + \varepsilon$
8: **end for**
9: $c_k \leftarrow r_0$, $r_0 \leftarrow r_1$, $r_1 \leftarrow r_2$, $r_2 \leftarrow 0$
10: **end for**
11: $c_{2n-1} \leftarrow r_0$
12: **return** c

3.6 Kvadriranje v $\mathbb{Z}_p$

Kvadriranje v $\mathbb{Z}_p$ bomo implementirali na isti način kot množenje. Vhodno število bomo najprej kvadrirali in nato nad rezultatom izvedli modularno redukcijo.

Algoritem (7) je rahla izboljšava algoritma (6). Ker kvadriranje lahko gledamo kot množenja števila samim s seboj, lahko število množenj algoritma (6) zmanjšamo približno za polovico. Časovna zahtevnost je še vedno enaka zahtevnosti množenja, torej $O(n^2)$.

števila (uv) v 6. vrstici algoritma (7) lahko učinkovito implementiramo kot dvojno seštevanje s prenosom ali s pomočjo premika s prenosom v levo za en bit.

Primer 3.4. Kvadrirajmo število $a = (7, 6, 9)_{10}$ z algoritmom (8). Spodnja tabela nam prikazuje vrednosti spremenljivk po izvajanju 7. in 13. vrstice algoritma.

i	j	$a_i a_j$	(uv)	u	v	$r1$	$r2$	c_5	c_4	c_3	c_2	c_1	c_0
0	1	54	56	5	6	6	5	0	0	0	0	6	1
0	2	63	77	7	7	6	7	0	0	0	7	6	1
0	-	-	3	0	3	7	1	0	1	3	7	6	1
1	2	42	51	5	1	4	5	0	1	1	3	6	1
1	-	-	10	1	0	4	0	1	0	1	3	6	1
2	-	-	5	0	5	4	0	5	9	1	3	6	1

Algoritem 7 Kvadriranje 1

VHOD: $a = (a_{n-1}, \ldots, a_0)$
IZHOD: $c = a^2$, $c = (c_{2n-1}, \ldots, c_0)$

1: $r_0 \leftarrow 0$, $r_1 \leftarrow 0$, $r_2 \leftarrow 0$
2: **for** $k = 0$ to $2n-2$ **do**
3: **for** $\{(i,j) | i + j = k, 0 \leq i \leq j \leq n-1\}$ **do**
4: $(uv) \leftarrow a_i \cdot a_j$
5: **if** $i < j$ **then**
6: $(\varepsilon, (uv)) \leftarrow (uv) \cdot 2$
7: $r_2 \leftarrow r_2 + \varepsilon$
8: **end if**
9: $(\varepsilon, r_0) \leftarrow r_0 + v$
10: $(\varepsilon, r_1) \leftarrow r_1 + u + \varepsilon$
11: $r_2 \leftarrow r_2 + \varepsilon$
12: **end for**
13: $c_k \leftarrow r_0$, $r_0 \leftarrow r_1$, $r_1 \leftarrow r_2$, $r_2 \leftarrow 0$
14: **end for**
15: $c_{2n-1} \leftarrow r_0$
16: **return** c

Algoritem 8 Kvadriranje 2

VHOD: $a = (a_{n-1}, \ldots, a_0)$
IZHOD: $c = a^2$, $c = (c_{2n-1}, \ldots, c_0)$

1: $c \leftarrow 0$
2: **for** $i = 0$ to $n-1$ **do**
3: $(uv) \leftarrow c_{2i} + a_i^2$
4: $c_{2i} \leftarrow v$, $r_1 \leftarrow u$, $r_2 \leftarrow 0$
5: **for** $j = i+1$ to $n-1$ **do**
6: $(uv) \leftarrow c_{i+j} + a_i a_j + r_1$, $r_1 \leftarrow u$
7: $(uv) \leftarrow v + a_i \cdot a_j + r_2$, $c_{i+j} \leftarrow v$, $r_2 \leftarrow u$
8: **end for**
9: $(uv) \leftarrow r_1 + r_2$, $r_2 \leftarrow u$
10: $(uv) \leftarrow c_{i+n} + v$, $c_{i+n} \leftarrow v$
11: **if** $i < n-1$ **then**
12: $c_{i+n+1} \leftarrow r_2 + u$
13: **end if**
14: **end for**
15: **return** c

3.7 Modularna redukcija

Včasih pridemo v situacijo, ko nas zanima samo ostanek pri deljenju z nekim številom. Seveda lahko ostanek izračunamo s pomočjo deljenja, vendar obstajajo metode, ki ga izračunajo še hitreje. Ena izmed njih je tudi *Barretova metoda*, ki jo bomo opisali v nadaljevanju.

Modularna redukcija s poljubnim modulom p je vseeno zelo zahtevna operacija. Ker je hitrost aritmetike na eliptičnih krivuljah zelo odvisna od hitrosti množenja v $\mathbb{Z}_p$, je priporočljivo uporabiti tak modul, za katerega bo redukcija hitrejša. V ta namen je NIST (`http://www.nist.gov/`) objavil seznam Mersennovih (a) praštevil, to so praštevila oblike $p = 2^k - 1$ in (b) psevdopraštevil, to so praštevila oblike $p = 2^m \pm 2^k \pm 1$, za katere obstajajo hitrejši algoritmi za modularno redukcijo. Ob tem se lahko vprašamo ali nam ta praštevila nudijo manjšo varnost kot poljubno izbrana praštevila. Dokaz za to zaenkrat še ne obstaja in zato uvrščamo ta praštevila med varne. V naši implementaciji bomo predpostavili, da je modul p poljuben in se teh praštevil/algoritmov ne bomo posluževali.

Deljenje realnih števil a in p lahko opravimo na več načinov. Eden izmed njih je tak, da najprej izračunamo inverz števila p do določene natančnosti in ga nato množimo z a. Inverzni element p^{-1} lahko izračunamo z iterativno Newtnovo metodo, z iteracijo

$$x \leftarrow x - x(px - 1),$$

kjer za x_0 vzamemo nek začetni približek.

Podobno tehniko lahko uporabimo na pozitivnih celih številih. Naj bo $p = (p_{n-1}, \ldots, p_0)$ in

$$R(p) = \left\lfloor \frac{2^{2nW}}{p} \right\rfloor$$

njegova *recipročna vrednost*. Če je $a = (a_{2n-1}, \ldots, a_0)$, potem velja

$$q = \left\lfloor \frac{a}{p} \right\rfloor = \left\lfloor \frac{\frac{a}{2^{(n-1)W}} \frac{2^{2nW}}{p}}{2^{(n+1)W}} \right\rfloor$$

in njegov približek je

$$\hat{q} = \left\lfloor \frac{\frac{a}{2^{(n-1)W}} R}{2^{(n+1)W}} \right\rfloor.$$

Za približek velja $q - 2 \leq \hat{q} \leq q$ in možno je pokazati, da je $q = \hat{q}$ v 90% vseh primerov. Za modularno redukcijo je tak približek zadovoljiv, saj je $\hat{a} \equiv a - \hat{q}p \equiv a \pmod{p}$. V primeru, da je $\hat{a} \geq p$ bomo potrebovali največ dve odštevanji, da dobimo pravi rezultat.

Algoritem 9 Barretova redukcija

VHOD: $a = (a_{2n-1}, \ldots, a_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $c = a \pmod{p}$, $c = (c_{n-1}, \ldots, c_0)$
1: $R \leftarrow \lfloor 2^{2nW}/p \rfloor$
2: $q \leftarrow \lfloor \lfloor a/2^{(n-1)W} \rfloor R/2^{(n+1)W} \rfloor$
3: $r_1 \leftarrow u \pmod{2^{(n+1)W}}$
4: $r_2 \leftarrow qp \pmod{2^{(n+1)W}}$
5: $c \leftarrow r_1 - r_2$
6: **if** $c < 0$ **then**
7: $\quad c \leftarrow c + 2^{(n+1)W}$
8: **end if**
9: **while** $c \geq p$ **do**
10: $\quad c \leftarrow c - p$
11: **end while**
12: **return** c

Barretovo modularno redukcijo v končnem obsegu lahko še izboljšamo. Ker je modul fiksen, ni potrebno vsakič na novo računati recipročne vrednosti. Tako lahko R izračunamo vnaprej in služi kot konstanten parameter za računanje po modulu p.

3.8 Deljenje

Pri deljenju števila a z b bomo izračunali kvocient $q = \lfloor a/b \rfloor$ in ostanek $r = a \pmod{b}$. Kadar je potrebno izračunati samo ostanek pri deljenju, se za to uporabljajo hitrejši algoritmi opisani v prejšnjem poglavju.

Primer 3.5. Zmnožimo števili $a = (\texttt{1,C,4,D,3})_{16}$ in $b = (\texttt{1,5,6})_{16}$ z algoritmom (10). Iz predstavitve števil a in b lahko določimo $n = 3$, $m = 2$, $b = 2^4 = 16$. Druga vrstica algoritma (10) nam začetni števili a in b množi z 8, tako da sta novi vrednosti $a = (\texttt{0,E,2,6,9,8})_{16}$, $b = (\texttt{A,B,0})_{16}$ in $d = 8$. Spodnja tabela nam prikazuje vrednosti spremenljivk v nadaljevanju algoritma.

i	2	1	0
$(16a_{i+n} + a_{i+n-1})$	`0xE`	`0x37`	`0x1F`
q (6. vrstica)	`0x1`	`0x5`	`0x3`
$256a_{n+i} + 16a_{n+i-1} + a_{n+i-2}$	`0xE2`	`0x376`	`0x1F9`
$q\,(16b_{n-1} + b_{n-2})$	`0xAB`	`0x357`	`0x201`
q (9. vrstica)	`0x1`	`0x5`	`0x2`
$(a_{n+i}, \ldots, a_i)_{16}$	`0xE26`	`0x3769`	`0x1F98`
$q(b_{n-1}, \ldots, b_0)_{16}$	`0xAB0`	`0x3570`	`0x1560`
c_i	`0x1`	`0x5`	`0x2`
$(a_{n+i}, \ldots, a_i)_{16}$	`0x376`	`0x1F9`	`0xA38`

Končni rezultat je torej `0x1C4D3 / 0x156 = 0x152`. Če želimo dobiti še ostanek, potem moramo število a na koncu deliti še z d. Ostanek pri deljenju števila a z b je torej `0xA38 / 0x8 = 0x147`.

Algoritem 10 Deljenje

VHOD: $a = (a_{m+n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $b_{n-1} > 0$, $n > 1$
IZHOD: $c = a/b$, $c = (c_m, \ldots, c_0)$

1: $a_{m+n} \leftarrow 0$, $d \leftarrow 1$
2: **while** $b_{n-1} < 2^{W-1}$ **do**
3: $\quad a \leftarrow 2a$, $b \leftarrow 2b$, $d \leftarrow 2d$
4: **end while**
5: **for** $i = m$ down to 0 **do**
6: $\quad q \leftarrow min(\lfloor (a_{n+i}2^W + a_{n+i-1})/b_{n-1} \rfloor, 2^W - 1)$
7: $\quad$ **while** $q(b_{n-1}2^W + b_{n-2}) > (a_{n+i}2^{2W} + a_{n+i-1}2^W + a_{n+i-2})$ **do**
8: $\quad\quad q \leftarrow q - 1$
9: $\quad$ **end while**
10: $\quad (a_{n+i}, \ldots, a_i) \leftarrow (a_{n+i}, \ldots, a_i) - q(b_{n-1}, \ldots, b_0)$
11: $\quad$ **if** $(a_{n+i}, \ldots, a_i) < 0$ **then**
12: $\quad\quad q \leftarrow q - 1$
13: $\quad\quad (a_{n+i}, \ldots, a_i) \leftarrow (a_{n+i}, \ldots, a_i) + (0, b_{n-1}, \ldots, b_0)$
14: $\quad$ **end if**
15: $\quad c_i \leftarrow q$
16: **end for**
17: **return** c

3.9 Računanje inverza v $\mathbb{Z}_p$

Inverz števila $a \in \mathbb{Z}_p$ je tak element $a^{-1} \in \mathbb{Z}_p$, za katerega velja

$$a^{-1}a \equiv aa^{-1} \equiv 1 \pmod{p}.$$

Če računamo inverz v $\mathbb{Z}_p$, kjer je p praštevilo, potem ima po izreku (2.4) vsak neničeln element $a \in \mathbb{Z}_p^*$ natanko en inverz. Iz enačbe (2.1) lahko izpeljemo $a^{-1}a \equiv a^{p-2}a \equiv a^{p-1} \equiv 1 \pmod{p}$, oziroma

$$a^{-1} \equiv a^{p-2} \pmod{p}. \tag{3.1}$$

Izkaže se, da računanje inverza po enačbi (3.1) ni ravno hitro. V primerjavi z razširjenim Evklidovim algoritmom, ki ga bomo spoznali v nadaljevanju, v povprečju potrebuje skoraj dvakrat več računskih operacij.

Razširjeni Evklidov algoritem

Definicija 3.1. Naj bosta a in b pozitivni celi števili. Potem je *največji skupni delitelj* števil a in b, označili ga bomo z $\gcd(a, b)$, največje celo število d, katero deli obe števili.

Evklidov algoritem sodi med najstarejše znane algoritme. Pojavil se je okoli 300 let pr.n.št. v knjigi *Evklidovi elementi* (angl. *Euclid's elements*). Takrat naj bi Evklid poskušal za dve poljubni daljici najti tako daljico (mero), s katero bi lahko izmeril obe daljici, ne da bi pri temu dobil kak ostanek. Postopek, opisan v knjigi, je zgledal kot zaporedno odštevanje krajše daljice od daljše, dokler nista obe daljici postali enako dolgi. Pravega avtorja algoritma ne poznamo, saj naj bi ta po nekaterih teorijah nastal že 200 let pred Evklidom. Vseeno pa ga še zmeraj pripisujemo njemu.

Izrek 3.1. *Naj bosta a in b pozitivni celi števili. Potem za vsa cela števila c velja*

$$\gcd(a, b) = \gcd(b, a - cb). \tag{3.2}$$

Dokaz. Naj bo $d = \gcd(a, b)$ in $a = kd$ in $b = \ell d$, kjer sta števili k in ℓ tuji. Potem

$$\begin{aligned} \gcd(b, a - cb) &= \gcd(\ell d, kd - c\ell d) \\ &= \gcd(\ell d, (k - c\ell)d) \\ &= d \cdot \gcd(\ell, k - c\ell). \end{aligned}$$

Denimo, da je $\gcd(\ell, k - c\ell) \neq 1 = d'$. Potem $d' \mid \ell$ in $d' \mid k - c\ell$ oziroma $d' \mid k - cd'\ell'$. Od tod sledi da $d' \mid k$, kar pa je v protislovju, da sta števili k in ℓ tuji. Torej je $\gcd(b, a - cb) = d = \gcd(a, b)$. □

Evklidov algoritem za izračun $\gcd(a, b)$, kjer $a \geq b$, najprej število a deli z b. Tako dobimo kvocient k in ostanek r, ki zadoščata enačbi $a = kb + r$. Po (3.2) velja $\gcd(a, b) = \gcd(b, r)$. Problem računanja $\gcd(a, b)$ smo prevedli na manjši problem računanja $\gcd(b, r)$. Ta postopek nadaljujemo, dokler ni $a = 0$. Zadnji korak postopka vedno zgleda kot $\gcd(0, d)$ in d je največji skupni delitelj števil a in b.

Evklidov algoritem lahko razširimo tako, da najdemo taki števili x in y, ki zadoščata enačbi $ax + by = d$, kjer je $d = \gcd(a, b)$. Razširjen Evklidov algoritem med svojim izvajanjem ohranja invarianti

$$ax_1 + by_1 = u \quad \text{in} \quad ax_2 + by_y = v,$$

kjer $u \leq v$. Algoritem se konča, ko je $u = 0$. Spremenljivka v ustreza $\gcd(a, b)$, spremenljivki $x = x_2$ in $y = y_2$ pa ustrezata enačbi $ax + by = d$.

Algoritem 11 Razširjen Evklidov algoritem

VHOD: $a \geq b$, $a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$
IZHOD: $d = \gcd(a, b)$, $ax + by = d$, $d = (d_{n-1}, \ldots, d_0)$, $x = (x_{n-1}, \ldots, x_0)$, $y = (y_{n-1}, \ldots, y_0)$

1: $u \leftarrow a$, $v \leftarrow b$
2: $x_1 \leftarrow 1$, $y_1 \leftarrow 0$,$x_2 \leftarrow 0$, $y_2 \leftarrow 1$
3: **while** $u \neq 1$ **do**
4: $\quad k \leftarrow \lfloor v/u \rfloor$, $r \leftarrow v - ku$, $x \leftarrow x_2 - kx_1$, $y \leftarrow y_2 - ky_1$
5: $\quad v \leftarrow u$, $u \leftarrow r$, $x_2 \leftarrow x_1$, $x_1 \leftarrow x$, $y_2 \leftarrow y_1$, $y_1 \leftarrow y$
6: **end while**
7: $d \leftarrow v$, $x \leftarrow x_2$, $y \leftarrow y_2$
8: **return** (d, x, y)

Naj bo p praštevilo in $a \in [1, p - 1]$. Potem je $\gcd(a, p) = 1$ in razširjen algoritem najde števili x in y, ki sta rešitvi enačbe $ax + py = 1$. Od tod sledi $ax \equiv 1 \pmod{p}$ oziroma razširjen Evklidov algoritem je našel inverz števila a. Za iskanje inverza ne potrebujemo člena y, tako da se nam algoritem poenostavi v algoritem (12).

Slabost Evklidovega algoritma se nahaja v četrti vrstici algoritmov (11) in (12). Ker sta števili u in v velikosti n-tih besed, je deljenje počasna operacija. Deljenja se lahko znebimo s pomočjo binarnega algoritma. Ta za c v enačbi

Algoritem 12 Računanje inverza v $\mathbb{Z}_p$ s pomočjo Evklidovega algoritma

VHOD: $0 < a < p$, $a = (a_{n-1}, \ldots, a_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $c = a^{-1} \pmod{p}$, $c = (c_{n-1}, \ldots, c_0)$

```
1: u ← a, v ← p
2: x1 ← 1, x2 ← 0
3: while u ≠ 1 do
4:     k ← ⌊v/u⌋, r ← v − ku, x ← x2 − kx1
5:     v ← u, u ← r, x2 ← x1, x1 ← x
6: end while
7: c ← x1 (mod p)
8: return c
```

(3.2) namesto količnika k vzame 1. S tem se znebimo deljenja, po drugi strani pa algoritem počasneje teče. Namreč v vsaki zanki pete vrstice se število u oziroma v zmanjša za en bit, tako da se zanka izvede največ $2 \cdot 2^{nW}$-krat.

Algoritem 13 Binarni algoritem za gcd

VHOD: $a \geq b$, $a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$
IZHOD: $d = \gcd(a, b)$, $d = (d_{n-1}, \ldots, d_0)$

```
1: u ← a, v ← b, e ← 1
2: while u ≡ 0 (mod 2) ∧ v ≡ 0 (mod 2) do
3:     u ← u/2, v ← v/2, e ← 2e
4: end while
5: while u ≠ 0 do
6:     while u = 0 (mod 2) do
7:         u ← u/2
8:     end while
9:     while v = 0 (mod 2) do
10:        v ← v/2
11:    end while
12:    if u ≥ v then
13:        u ← u − v
14:    else
15:        v ← v − u
16:    end if
17: end while
18: d ← ev
19: return d
```

Algoritem 14 Računanje inverza v $\mathbb{Z}_p$ s pomočjo binarnega algoritma

VHOD: $0 < a < p$, p lih, $a = (a_{n-1}, \ldots, a_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $c = a^{-1} \pmod{p}$, $c = (c_{n-1}, \ldots, c_0)$

```
1: u ← a, v ← p
2: x1 ← 1, x2 ← 0
3: while u ≠ 1 ∧ v ≠ 1 do
4:     while u = 0 (mod 2) do
5:         u ← u/2
6:         if x1 = 0 (mod 2) then
7:             x1 ← x1/2
8:         else
9:             x1 ← (x1 + p)/2
10:        end if
11:    end while
12:    while v = 0 (mod 2) do
13:        v ← v/2
14:        if x2 = 0 (mod 2) then
15:            x2 ← x2/2
16:        else
17:            x2 ← (x2 + p)/2
18:        end if
19:    end while
20:    if u ≥ v then
21:        u ← u − v, x1 ← x1 − x2
22:    else
23:        v ← v − u, x2 ← x2 − x1
24:    end if
25: end while
26: if u = 1 then
27:    c ← x1 (mod p)
28: else
29:    c ← x2 (mod p)
30: end if
31: return c
```

S pomočjo binarnega algoritma lahko računamo tudi inverz. Še več, če v drugi vrstici algoritma (14) zamenjamo $x_1 \leftarrow 1$ z $x_1 \leftarrow b$, potem kot rezultat dobimo $b/a = ba^{-1} \pmod{p}$.

3.10 Korenjenje v $\mathbb{Z}_p$

Korenjenje števila $a \in \mathbb{Z}_p$ je definirano kot iskanje takega števila $b \in \mathbb{Z}_p$, za katerega velja

$$b^2 \equiv a \pmod{p}.$$

Ker ni nujno, da za vsako število obstaja koren, bomo pred korenjenjem izračunali Jacobijev simbol $\left(\frac{a}{p}\right)$. V primeru da je njegova vrednost 1, kvadratni koren obstaja in ga lahko izračunamo.

Jacobijev simbol lahko učinkovito izračunamo s pomočjo algoritma (15), ki je nastal po izreku (2.7). Njegova zahtevnost je enaka zahtevnosti Evklidovega algoritma. Namreč oba algoritma med izvajanjem izvajata modularno redukcijo večjega števila z manjšim.

Algoritem 15 Računanje Jacobijevega simbola

VHOD: $0 < a < p$, $a = (a_{n-1}, \ldots, a_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $c = \left(\frac{a}{p}\right)$, $c \in \{-1, 0, 1\}$

```
 1: c ← 1
 2: while p ≠ 1 do
 3:     if a = 0 then
 4:         return 0
 5:     end if
 6:     v ← 0
 7:     while a = 0 (mod 2) do
 8:         v ← v + 1
 9:         a ← a/2
10:     end while
11:     if v = 1 (mod 2) ∧ p = ±3 (mod 8) then
12:         c ← −c
13:     end if
14:     if a = 3 (mod 4) ∧ p = ±3 (mod 4) then
15:         c ← −c
16:     end if
17:     r ← a, a ← p (mod r), p ← r
18: end while
19: return c
```

Primer 3.6. Naj bo $a = 109608$ in $p = 163841$ praštevilo. Vrednosti spremenljiv algoritma (15) v sedemnajsti vrstici prikazuje spodnja tabela.

r	a	v	k
13701	13130	3	1
6565	571	1	-1
571	284	0	-1
71	3	2	1
3	2	0	-1
1	0	1	1

Potek algoritma lahko gledamo kot zaporedje enakosti

$$\begin{aligned}
\left(\frac{109608}{163841}\right) &\overset{(v)}{=} \left(\frac{8}{163841}\right)\left(\frac{13701}{163841}\right) \\
&\overset{(vii)}{=} \left(\frac{13130}{13701}\right) \overset{(v)}{=} \left(\frac{2}{13701}\right)\left(\frac{6565}{13701}\right) \\
&\underset{(vii)}{\overset{(iii)}{=}} -\left(\frac{574}{6565}\right) \\
&\overset{(vii)}{=} -\left(\frac{284}{571}\right) \overset{(v)}{=} -\left(\frac{4}{571}\right)\left(\frac{71}{571}\right) \\
&\overset{(vii)}{=} \left(\frac{3}{71}\right) \\
&\overset{(vii)}{=} -\left(\frac{2}{3}\right) \overset{(iii)}{=} 1.
\end{aligned}$$

Izrek 3.2. *Naj bo $a \in \mathbb{Z}_p$, p liho praštevilo in $p \not\equiv 1 \pmod 8$. Potem lahko kvadratni koren od a izračunamo po naslednjih formulah:*

- $b = \pm a^{(p+1)/4} \mod p$, *če* $p \equiv 3 \pmod 4$,
- $b = \pm a^{(p+3)/8} \mod p$, *če* $p \equiv 5 \pmod 8$ *in* $a^{(p-1)/4} = 1$,
- $b = \pm 2a(4a)^{(p-5)/8} \mod p$, *če* $p \equiv 5 \pmod 8$ *in* $a^{(p-1)/4} = -1$.

Dokaz. Ker je p liho praštevilo in $p \not\equiv 1 \pmod 8$, ga lahko zapišemo kot:

- $p \equiv 3 \pmod 4$ oz $p = 4k + 3$ in zanj velja $a^{(p-1)/2} = 1$

$$\begin{aligned}
b^2 &\equiv \left(\pm a^{(p+1)/4}\right)^2 = \left(\pm a^{k+1}\right)^2 = a^{2k+2} = a^{2k+1}a = a^{(p-1)/2}a \\
&= a \pmod p,
\end{aligned}$$

- $p \equiv 5 \pmod 8$ oz $p = 8k + 5$ in zanj velja $a^{(p-1)/4} = 1$

$$b^2 \equiv \left(\pm a^{(p+3)/8}\right)^2 = \left(\pm a^{k+1}\right)^2 = a^{2k+2} = a^{2k+1}a = a^{(p-1)/4}a$$
$$= a \pmod p,$$

- $p \equiv 5 \pmod 8$ oz $p = 8k + 5$ in zanj velja $a^{(p-1)/4} = -1$

$$b^2 \equiv \left(\pm 2a(4a)^{(p-5)/8}\right)^2 = \left(\pm 2a(4a)^k\right)^2 = 4a^2(4a)^{2k} = 4^{2k+1}a^{2k+2}$$
$$= 2^{4k+2}a^{2k+1}a = \left(\frac{2}{p}\right) a^{(p-1)/4}a = -a(-1)^{(p^2-1)/8} = -a(-1)^{8k^2-10k+3}$$
$$= a \pmod p.$$

□

Algoritem (16) nam izračuna kvadratni koren elementa $a \in \mathbb{Z}_p$. V prvi vrstici algoritma bo potrebno izračunati *sodi* in *lihi del* števila a. Sodi del je največje število 2^e, ki deli število a. Če a zapišemo v binarnem zapisu, potem je e pravzaprav dolžina zaporedja ničel od desne proti levi, ki sega od začetka pa do prve enice. Če procesor nima posebnega ukaza za izračun sodega dela, potem lahko izračunamo $a \wedge (\bar{a} + 1)$, da najdemo najnižji bit števila a, ki je različen od 0. Pri velikih številih sodi del najlažje izračunamo tako, da bite števila a premikamo v desno toliko časa, dokler ni prvi bit enak 1. Število teh premikov ustreza številu e, medtem ko a ustreza lihemu delu. V algoritmu bomo izračun sodega in lihega dela označili z $2^e r \leftarrow a$.

Omenimo še, da je algoritem (16) namenjen izračunu kvadratnega korena za $p \equiv 1 \pmod 8$. Vseeno pa dela pravilno tudi za vsa ostala praštevila.

Primer 3.7. Izračunajmo kvadratni koren števila $a = 109608$ po modulu $p = 163841$. Sodi del števila a je $e = 15$ in lihi del $r = 5$. Bralec lahko sam preveri, da je $2^{15} \cdot 5 = 109608$. Če si za m izberemo 6558, potem spodnja tabela prikazuje vrednosti spremenljivk v enajsti vrstici algoritma (16).

k	y	x	b	t
13	82347	68270	78996	31765
12	140942	56092	104389	82347
6	81165	57313	18205	52992
5	38297	101925	90687	81165
3	101925	39338	97748	119418
2	39338	163840	121372	101925
1	163840	1	41155	39338

Algoritem 16 Korenjenje v $\mathbb{Z}_p$

VHOD: $0 < a < p$, $a = (a_{n-1}, \ldots, a_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $b = (b_{n-1}, \ldots, b_0)$, za katerega velja $b^2 = a$

```
1: 2^e r ← p − 1
2: izberi naključni m, da velja (n/p) = −1
3: y ← m^r (mod p), b ← a^((r−1)/2)
4: x ← ab^2 (mod p), b ← ax (mod p)
5: while x ≢ 1 (mod p) do
6:     k ← 1
7:     while x^(2^k) ≢ 1 (mod p) do
8:         k ← k + 1
9:     end while
10:    t ← y^(2^(e−k−1)) (mod p), y ← t^2 (mod p), e ← k
11:    b ← tb (mod p), x ← xy (mod p)
12: end while
13: return b
```

Če bi za m izbrali 151770, potem bi tabela zgledala drugače.

k	y	x	b	t
13	117841	23823	41708	29456
12	157326	114423	10110	117841
8	24627	157703	87616	68138
7	113588	104152	97103	24627
5	38297	163840	42469	82676
1	163840	1	122686	39338

V prvem primeru smo dobili kvadratni koren $b_1 = 41155$ v drugem pa $b_2 = 122686$. Oba rezultata sta seveda pravilna, saj je

$$41155^2 \equiv 122686^2 \equiv 109608 \pmod{163841}.$$

Iz enačbe $41155 = 163841 - 122686$ lahko tudi ugotovimo, da sta si števili b_1 in b_2 nasprotni.

Poglavje 4

Eliptične krivulje

4.1 Uvod

Krivulja v ravnini je množica točk, ki zadoščajo enačbi $F(x, y) = 0$. Najbolj preproste ravninske krivulje so

- *premice*: $F(x, y) = ax + by + c$,
- *stožnice*: $F(x, y) = ax^2 + bx + cy^2 + dy + exy + f$,
- *kubične krivulje*: $F(x, y) = ax^3 + bx^2 + cx + dy^3 + ey^2 + fy + gx^2y + hxy + ixy^2 + j$.

Med kubične krivulje spadajo tudi eliptične krivulje. Le te so dobile ime po elipsi, saj so v preteklosti z njimi reševali problem računanja obsega elipse[1].

Definicija 4.1. Naj bo K obseg. Weierstrassova enačba eliptične krivulje E/K je enačba oblike

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{4.1}$$

kjer so a_1, a_2, a_3, a_4, a_6 elementi obsega K.

Množica točk $E(K)$ je sestavljena iz vseh točk $(x, y) \in E(K)$, $x, y \in K$, katere zadoščajo zgornji enačbi in iz točke v neskončnosti $\mathcal{O}$. Točko $\mathcal{O}$ bomo natančneje definirali v nadaljevanju, zaenkrat bomo samo omenili, da točka $\mathcal{O}$ zadošča vsaki enačbi eliptične krivulje.

[1]Elipsa je stožnica definirana z enačbo $\frac{(x-p)^2}{a^2} + \frac{(y-q)^2}{b^2} = 1$.

Definicija 4.2. Diskriminanta Δ eliptične krivulje je definirana kot

$$\Delta = -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2 d_4 d_6,$$

kjer je

$$\begin{aligned} d_2 &= a_1^2 + 4a_2 \\ d_3 &= 2a_4 + a_1 a_3 \\ d_6 &= a_3^2 + 4a_6 \\ d_8 &= a_1^2 a_6 + 4a_2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 - a_4^2. \end{aligned}$$

Eliptična krivulja E je *nesingularna*, če je $\Delta \neq 0$.

Primer 4.1. Naj bo $p = 2003$ in $\mathbb{Z}_p$ obseg. Potem je

$$E : y^2 + 2xy + 8y = x^3 + 5x^2 + 1136x + 531 \tag{4.2}$$

eliptična krivulja, z $d_2 = 24, d_3 = 285, d_6 = 185, d_8 = 333$ in $\Delta = 1707$.

Definicija 4.3. Eliptični krivulji E_1 in E_2 nad obsegom K definirani z Weierstrassovima enačbama

$$\begin{aligned} E_1 &: y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6 \\ E_2 &: y^2 + a_1' xy + a_3' y = x^3 + a_2' x^2 + a_4' x + a_6' \end{aligned}$$

sta *izomorfni nad obsegom* K, če obstajajo $u, r, s, t \in K$, $u \neq 0$, da zamenjava spremenljivk

$$(x, y) \to (u^2 x + r, u^3 y + u^2 s x + t)$$

spremeni enačbo E_1 v E_2.

Naj bo E eliptična krivulja, definirana z enačbo (4.1), nad obsegom K, $\mathrm{char}(K) \neq 2, 3$. Potem zamenjava spremenljivk

$$(x, y) \to \left(\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1 x}{216} - \frac{a_1^3 + 4a_1 a_2 - 12a_3}{24} \right)$$

spremeni enačbo eliptične krivulje v

$$y = x^3 + ax + b. \tag{4.3}$$

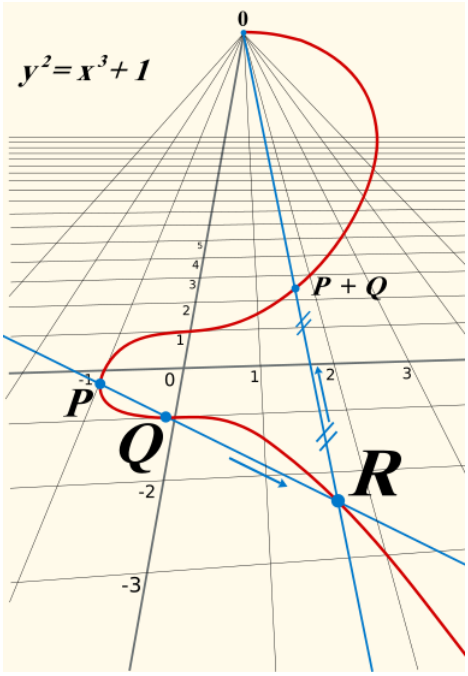
Primer 4.2. Naj bo $p = 2003$ in E eliptična krivulja, definirana z enačbo (4.2), nad obsegom $\mathbb{Z}_p$. Potem zamenjava spremenljivk $(x, y) \to (x - 2, y - x - 2)$ spremeni enačbo eliptične krivulje v

$$E : y^2 = x^3 + 1132x + 278. \tag{4.4}$$

4.2 Seštevanje na eliptični krivulji

Naj bo E eliptična krivulja nad obsegom K. Operacijo seštevanja $\oplus$ bomo definirali tako, da bo $(E(K), \oplus)$ grupa.

Če želimo sešteti različni točki $P(x_1, y_1)$ in $Q(x_2, y_2)$, potem čez ti dve točki potegnemo premico. Premica seka eliptično krivuljo E še v eni točki. Če to točko preslikamo čez x-os dobimo točko $R = P \oplus Q$. V primeru, da sta točki P in Q enaki, potem namesto sekante vzamemo tangento na eliptično krivuljo v točki P.



Slika 4.1: Seštevanje točk P in Q

Naj bosta P in Q različni točki z enako x-koordinato. Ker ima enačba $y^2 = f(x)$ največ dve rešitvi (x_1, y_1) in $(x_1, -y_1)$, teh dveh točk ne moremo sešteti. Namreč premica, ki poteka skozi točki P in Q, ne seka eliptične krivulje v nobeni drugi točki. Problem bomo rešili s pomočjo *točke v neskončnosti*, ki bo enota za operacijo $\oplus$. Označili jo bomo z $\mathcal{O}$ in si predstavljali, da leži neskončno daleč na y-osi. Tako skozi njo potekajo vse premice, ki so vzporedne y-osi in imajo obliko $x = c$, kjer je c neka konstanta.

S pomočjo enote lahko izračunamo tudi nasprotni element $-P$. Ta mora zadoščati enačbi $P \oplus (-P) = \mathcal{O}$. Iz definicije enote sledi, da ima nasprotni element enako x-koordinato in negirano y-koordinato. Nasprotni element točke P je $(x_1, -y_1)$. Ko imamo definiran nasprotni element, lahko definiramo operacijo odštevanja $\ominus$, kot prištevanje nasprotnega elementa.

Naj bo $P \neq Q$ in $x_1 \neq x_2$. Potem ima premica skozi obe točki naklon

$$\lambda = \frac{y_1 - y_2}{x_1 - x_2}.$$

Ker premica poteka skozi točko P, jo lahko zapišemo z enačbo

$$y = \lambda x + \frac{x_1 y_2 - x_2 y_1}{x_1 - x_2}.$$

Konstantni člen premice bomo označili z $\mu = (x_1 y_2 - x_2 y_1)/(x_1 - x_2)$. Presečišče premice in eliptične krivulje E, definirane z enačbo (4.1), bomo izračunali tako, da se bomo znebili spremenljivke y. Dobimo enačbo

$$(\lambda x + \mu)^2 + a_1 x(\lambda x + \mu) + a_3(\lambda x + \mu) = x^3 + a_2 x^2 + a_4 x + a_6.$$

Če vse člene zgornje enačbe spravimo na eno stran, pridemo do enačbe $r(x) = 0$, kjer je

$$r(x) = x^3 + (a_2 - \lambda^2 - a_1\lambda)x^2 + (a_4 - 2\lambda\mu - a_3\lambda - a_1\mu)x + a_6 - \mu^2 - a_3\mu.$$

Ker točki $P = (x_1, y_1)$ in $Q = (x_2, y_2)$ ustrezata presečišču premice in eliptične krivulje, sta koordinati x_1 in x_2 ničli funkcije $r(x)$ oziroma

$$r(x) = (x - x_1)(x - x_2)(x - x_3).$$

Če izenačimo koeficiente člena x^2 iz obeh enačb za $r(x)$, dobimo da x-koordinata tretjega presečišča ustreza enačbi $\lambda^2 + a_1\lambda - a_2 = x_1 + x_2 + x_3$, y-koordinata pa $y_3 = \lambda x_3 + \mu$. To točko še preslikamo čez x-os in dobimo vsoto točk $P \oplus Q = (x_3, -\lambda x_3 - \mu - a_1 x_3 - a_3)$.

V primeru, da želimo sešteti enaki točki P in Q, bomo rekli, da točko P *podvojimo*. Podvajanje točke poteka na podoben način, le da smerni koeficient premice izračunamo z odvodom. Odvajamo enačbo (4.1) in dobimo

$$2yy' + a_1 y + a_1 xy' + a_3 y' = 3x^2 + 2a_2 x + a_4.$$

Smerni koeficient tangente v točki $P = (x_1, y_1)$ je tako

$$\lambda = \frac{3x_1^2 + 2a_2 x_1 + a_4 - a_1 y_1}{2y_1 + a_1 x_1 + a_3}.$$

Primer 4.3. Naj bo $p = 2003$, E eliptična krivulja, definirana z enačbo (4.2), nad obsegom $\mathbb{Z}_p$, $P = (1118, 269)$ in $Q = (892, 529)$. Potem

$$\begin{aligned} -P &= (1118, 1493), \\ P \oplus Q &= (1681, 1706), \\ [2]P &= (1465, 677), \end{aligned}$$

kjer $[2]P$ pomeni podvajanje točke P.

Eliptične krivulje nad obsegom $\mathbb{Z}_p$

Izberimo si obseg $\mathbb{Z}_p$, kjer je p neko veliko praštevilo, različno od $2, 3$. Vsako Weierstrassovo enačbo eliptične krivulje nad tem obsegom lahko z zamenjavo spremenljivk preoblikujemo v (4.3). Za tako definirane krivulje veljajo naslednje lastnosti.

- *Enota:* $P + \mathcal{O} = \mathcal{O} + \mathcal{P} = \mathcal{P}$ za vsak $P \in E(K)$.
- *Negativni element:* Če je $P = (x, y) \in E(K)$, potem $(x, y) \oplus (x, -y) = \mathcal{O}$, oziroma $-P = (x, -y)$.
- *Seštevanje:* Naj bo $P = (x_1, y_1) \in E(K)$ in $Q = (x_2, y_2) \in E(K)$, $P \neq \pm Q$ potem $P \oplus Q = (x_3, y_3)$, kjer

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \text{ in } y_3 = \frac{y_2 - y_1}{x_2 - x_1} - (x_1 - x_3) - y_1.$$

- *Podvajanje:* Naj bo $P = (x_1, y_1)$ in $P \neq -P$. Potem $[2]P = (x_3, y_3)$, kjer

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \text{ in } y_3 = \frac{3x_1^2 + a}{2y_1} - (x_1 - x_3) - y_1.$$

4.3 Množenje s skalarjem

Eliptične krivulje tvorijo grupo in imajo zato definirano množenje točke s skalarjem. Do sedaj smo za podvajanje točke, torej za izračun vsote $P \oplus P$, uporabljali notacijo $[2]P$. To notacijo bomo razširili za poljubno število n.

Definicija 4.4. Naj bo $n \in \mathbb{N} \setminus \{0\}$ in naj bo $[n]P$ *množenje točke* P *s skalarjem* n. Potem je $[n] : E \to E$ preslikava, definirana kot

$$[n]P = \underbrace{P \oplus P \oplus \ldots \oplus P}_{n \text{ krat}}.$$

Če definiramo $[0]P = \mathcal{O}$ in $[n]P = [-n](-P)$ za $n < 0$, potem smo definicijo razširili za vse skalarje $n \in \mathrm{Z}$.

Primer 4.4. Eliptična krivulja $E : y^2 = x^3 + 4x + 20$ nad obsegom $\mathbb{Z}_{29}$ vsebuje $\#E(\mathbb{Z}_{29}) = 37$ točk. Vzemimo točko $P = (1, 5) \in E(\mathbb{Z}_{29})$ in z njo generirajmo vse elemente grupe.

$[0]P = \mathcal{O}$	$[10]P = (13, 23)$	$[20]P = (27, 27)$	$[30]P = (24, 7)$
$[1]P = (1, 5)$	$[11]P = (10, 25)$	$[21]P = (0, 7)$	$[31]P = (17, 10)$
$[2]P = (4, 19)$	$[12]P = (19, 13)$	$[22]P = (3, 28)$	$[32]P = (6, 17)$
$[3]P = (20, 3)$	$[13]P = (16, 27)$	$[23]P = (5, 7)$	$[33]P = (15, 2)$
$[4]P = (15, 27)$	$[14]P = (5, 22)$	$[24]P = (16, 2)$	$[34]P = (20, 26)$
$[5]P = (6, 12)$	$[15]P = (3, 1)$	$[25]P = (19, 16)$	$[35]P = (4, 10)$
$[6]P = (17, 19)$	$[16]P = (0, 22)$	$[26]P = (10, 4)$	$[36]P = (1, 24)$
$[7]P = (24, 22)$	$[17]P = (27, 2)$	$[27]P = (13, 6)$	$[37]P = \mathcal{O}$
$[8]P = (8, 10)$	$[18]P = (2, 23)$	$[28]P = (14, 6)$	
$[9]P = (14, 23)$	$[19]P = (2, 6)$	$[29]P = (8, 19)$	

4.4 Projektivne koordinate

Algoritmi za seštevanje in podvajanje po zgornjih enačbah za svoje delovanje potrebujejo izračunati en inverzni element, nekaj negacij in seštevanj. Če je računanje inverza v primerjavi z množenjem zelo zahtevna operacija, potem bomo videli, da je koristno točke predstaviti v *projektivnih koordinatah*.

Naj bo K obseg in c, d naravni števili. Potem lahko definiramo ekvivalenčno relacijo $\sim$ na množici $K^3 \setminus (0, 0, 0)$ kot

$$(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2),$$

če obstaja nek $\lambda \in K^*$, za katerega velja

$$X_1 = \lambda^c X_2, Y_1 = \lambda^d Y_2, Z_1 = \lambda Z_2.$$

Ekvivalenčni razred elementa $(X, Y, Z) \in K^3 \setminus (0, 0, 0)$ je

$$(X : Y : Z) = \{(\lambda^c X, \lambda^d Y, \lambda Z) : \lambda \in K^*\}.$$

Točki $(X : Y : Z)$ pravimo *projektivna točka*, (X, Y, Z) pa je njen predstavnik. Množico vseh projektivnih točk bomo označili z $\mathbb{P}(K)$. Ker velja

$$(X', Y', Z') \in (X : Y : Z) \Rightarrow (X' : Y' : Z') = (X : Y : Z),$$

je vsak element ekvivalenčnega razreda njegov predstavnik. Za $Z \neq 0$ je $(X/Z^c : Y/Z^d : 1)$ edini predstavnik projektivne točke $(X : Y : Z)$ z Z-koordinato 1. Tako vsaka projektivna točka

$$(X : Y : Z) : X, Y, Z \in K, Z \neq 0$$

pripada eni afini točki

$$(x, y) : x, y \in K,$$

kjer $(x, y) = (X/Z^c, Y/Z^d)$. Množici točk

$$\mathbb{P}(K)^0 = \{(X : Y : Z) : X, Y, Z \in K, Z = 0\}$$

bomo rekli *premica v neskončnosti*, saj njene točke ne pripadajo nobeni afini točki. Premica v neskončnosti namreč ustreza točki v neskončnosti v afinih koordinatah.

Projektivno obliko Weierstrassove enačbe eliptične krivulje E nad obsegom K dobimo tako, da zamenjamo spremenljivki x in y z X/Z^c in Y/Z^d, ter odpravimo ulomke. Če $(X, Y, Z) \in K \setminus (0, 0, 0)$ zadošča projektivni obliki enačbe, potem ji zadošča tudi vsaka $(X', Y', Z') \in (X : Y : Z)$. Torej lahko rečemo, da točka $(X : Y : Z)$ leži na eliptični krivulji E.

Primer 4.5. Naj bo $c = 1$ in $d = 1$. Potem je projektivna oblika Weierstrassove enačbe

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

nad obsegom K enaka

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3.$$

Edina točka, ki leži v neskončnosti in leži tudi na E, je točka $(0 : 1 : 0)$. Ta projektivna točka ustreza točki v neskončnosti $\mathcal{O}$.

Primer 4.6. Naj bo $p = 2003$ in E eliptična krivulja, definirana z enačbo (4.2), nad obsegom $\mathbb{Z}_p$. Potem je enačba eliptične krivulje E v projektivni obliki, kjer $c = 1$ in $d = 1$, enaka

$$E : Y^2Z + 2XYZ + 8YZ^2 = X^3 + 5X^2Z + 1136XZ^2 + 531Z^3.$$

Točka $P' = (917 : 527 : 687)$ leži na eliptični krivulji in pripada istemu ekvivalenčnemu razredu kot točka $(1118 : 269 : 1)$. Torej projektivna točka P' ustreza afini točki $P = (1118, 269)$.

Eliptične krivulje v projektivni obliki nad obsegom $\mathbb{Z}_p$

Vsako eliptično krivuljo nad obsegom $\mathbb{Z}_p$, kjer $\text{char}(\mathbb{Z}_p) \neq 2, 3$, se da zapisati z enačbo (4.3). Projektivna oblika te enačbe, kjer $c = 1$ in $d = 1$, je

$$Y^2Z = X^3 + aXZ^2 + bZ^3.$$

Tudi tu točka v neskončnosti ustreza točki $(0 : 1 : 0)$.

4.5 Algoritmi za eliptične krivulje v obsegu $\mathbb{Z}_p$

V nadaljevanju bomo predpostavili, da je eliptična krivulja E definirana z enačbo (4.3) nad obsegom $\mathbb{Z}_p$. Ker bomo za vsak algoritem preverili, koliko aritmetičnih operacij v $\mathbb{Z}_p$ potrebuje, bomo z M označili množenje, s K kvadriranje in z I računanje inverza. Ker so te operacije občutno zahtevnejše od seštevanja/odštevanja, bomo slednji dve zanemarili.

Seštevanje in odštevanje

Iz algoritma (17) je očitno, da je časovna zahtevnost seštevanja oziroma podvajanja točk v afinih koordinatah

$$I + 2M + K$$

oziroma $I + 2M + 2K$. Kadar je računanje inverza zahtevna operacija, raje računamo v projektivnih koordinatah. Seštevanje oziroma podvajanje točk v projektivnih koordinatah lahko realiziramo (če še malo izboljšamo algoritem (20)) do časovne zahtevnosti $12M + 4K$ oziroma $4M + 6K$.

Odštevanje točk, predstavljenih v afinih/projektivnih koordinatah, ima enako zahtevnost kot seštevanje. Namreč za izračun nasprotnega elementa poljubne točke na eliptični krivulji je podrebno eno samo modularno odštevanje. Ker je eno modularno odštevanje v primerjavi z množenjem/kvadriranjem poceni operacija, jo bomo zanemarili.

Predpostavili bomo, da algoritem (19) vse operacije izvaja po modulu p.

Algoritem 17 Seštevanje točk (afine koordinate)

VHOD: $P = (x_1, y_1)$, $Q = (x_2, y_2)$, $p > 3$,
$a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $R = P \oplus Q$
1: **if** $P = \mathcal{O}$ **then**
2: **return** Q
3: **else if** $Q = \mathcal{O}$ **then**
4: **return** P
5: **end if**
6: **if** $x_1 = x_2$ **then**
7: **if** $(y_1 \neq y_2) \vee (y_2 = 0)$ **then**
8: **return** $\mathcal{O}$
9: **else**
10: $\lambda \leftarrow (3x_2^2 + a) \,/\, (2y_2) \mod p$
11: **end if**
12: **else**
13: $\lambda \leftarrow (y_1 - y_2) \,/\, (x_1 - x_2) \mod p$
14: **end if**
15: $x_3 \leftarrow \lambda^2 - x_1 - x_2 \pmod p$
16: $y_3 \leftarrow (x_2 - x_3)\lambda - y_2 \pmod p$
17: **return** (x_3, y_3)

Algoritem 18 Odštevanje (afine koordinate)

VHOD: $P = (x_1, y_1)$, $Q = (x_2, y_2)$, $p > 3$,
$a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $R = P \ominus Q$
1: $y_2' \leftarrow p - y_2$
2: $R \leftarrow (x_1, y_1) \oplus (x_2, y_2')$
3: **return** R

Algoritem 19 Seštevanje točk (projektivne koordinate)

VHOD: $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$, $p > 3$,
$a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $R = P \oplus Q$

1: **if** $Z_1 = 0$ **then**
2: **return** Q
3: **else if** $Z_2 = 0$ **then**
4: **return** P
5: **end if**
6: **if** $P = Q$ **then**
7: $M \leftarrow 3X_1^2 + aZ_1^4$
8: $S \leftarrow 4X_1 + Y_1^2$
9: $T \leftarrow 8Y_1^4$
10: $X_3 \leftarrow M^2 - 2S$
11: $Y_3 \leftarrow M(S - X_2) - T$
12: $Z_3 \leftarrow 2Y1Z_1$
13: **else**
14: $U_1 \leftarrow X_1Z_2^2$
15: $S_1 \leftarrow Y_1Z_2^3$
16: $U_2 \leftarrow X_2Z_1^2$
17: $S_2 \leftarrow Y_2Z_1^3$
18: $W \leftarrow U_1 - U_2$
19: $R \leftarrow S_1 - S_2$
20: $T \leftarrow U_1 + U_2$
21: $M \leftarrow S_1 + S_2$
22: $X_3 \leftarrow R^2 - TW^2$
23: $V \leftarrow TW^2 - 2X_3$
24: $Y_3 \leftarrow (VR - MW^3)/2$
25: $Z_3 \leftarrow Z_1Z_2W$
26: **end if**
27: **return** (X_3, Y_3, Z_3)

Algoritem 20 Odštevanje (projektivne koordinate)

VHOD: $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$, $p > 3$,
$a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $R = P \ominus Q$
1: $Y_2' \leftarrow p - Y_2$
2: $R \leftarrow (X_1, Y_1, Z_1) \oplus (X_2, Y_2', Z_2)$
3: **return** R

Množenje s skalarjem

Množenje s skalarjem $[n]P$ lahko implementiramo z metodo *podvoji in prištej*. Ta se premika po bitih binarne predstavitve skalarja n od leve proti desni. Na vsakem koraku rezultat podvoji in mu v primeru, da je trenutni bit enak 1, prišteje P. Tako pristop lahko uporabimo tudi pri branju skalarja n od desne proti levi.

Algoritem 21 Množenje s skalarjem z desne proti levi

VHOD: $P = (x, y)$, $p > 3$, $k = (k_{t-1}, \ldots, k_0)$,
$a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $R = [k]P$
1: $R \leftarrow \mathcal{O}$
2: **for** $i = 0$ to $t - 1$ **do**
3: **if** $k_i = 1$ **then**
4: $R \leftarrow R \oplus P$
5: **end if**$P \leftarrow [2]P$
6: **end for**
7: **return** P

NAF predstavitev števil

Naj bo $\mathbb{Z}_p$ obseg, $\text{char}(\mathbb{Z}_p) > 3$ in $P, -P \in E(\mathbb{Z}_p)$, $x, y \in \mathbb{Z}_p$. Potem ima odštevanje točk na eliptični krivulji enako zahtevnost kot seštevanje. Če želimo to lastnost izkoristiti pri množenju točke s skalarjem, potem moramo skalar predstaviti v predznačeni obliki.

Definicija 4.5. NAF (angl. *Non-adjecent form*) predstavitev pozitivnega

Algoritem 22 Množenje s skalarjem z leve proti desni

VHOD: $P = (x, y)$, $p > 3$,
$k = (k_{t-1}, \ldots, k_0)$, $a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $R = [k]P$

```
1: R ← O
2: for i = t − 1 to 0 do
3:     R ← [2]R
4:     if k_i = 1 then
5:         R ← R ⊕ P
6:     end if
7: end for
8: return R
```

števila k, označili jo bomo z NAF(k), je izraz oblike

$$k = \sum_{i=0}^{t-1} k_i 2^i,$$

kjer je $k_i \in \{-1, 0, 1\}$, $k_{t-1} \neq 0$ in nobeni dve zaporedni števki k_i nista različni od 0. Število t imenujemo *dolžina* NAF zapisa.

Izrek 4.1. *Lastnosti* NAF *predstavitve pozitivnega števila k so:*

- *k ima enolično predstavitev v* NAF *obliki,*
- NAF(k) *ima najmanjše število neničelnih elementov od vseh zapisov s števkami $\{-1, 0, 1\}$,*
- *dolžina binarnega zapisa števila k je za največ eno krajša od dolžine* NAF *zapisa,*
- *če je dolžina* NAF *zapisa t, potem $2^t/3 < k < 2^{t+1}/3$,*
- NAF *zapis ima v povprečju približno $t/3$ neničelnih elementov.*

Dokaz. Za dokaz glej [6, 187]. □

NAF predstavitev števila lahko učinkovito izračunamo z algoritmom (23).

Algoritem 23 Izračun NAF predstavitve pozitivnega števila

VHOD: $k > 0$
IZHOD: $NAF(k) = (k_{t-1}, \ldots, k_0)$, $k_i \in \{-1, 0, 1\}$

```
1: i ← 0
2: while k ≥ 1 do
3:     if k = 1 (mod 2) then
4:         k_i ← 2 − (k (mod 4))
5:         k ← k − k_i
6:     else
7:         k_i ← 0
8:     end if
9:     k ← k/2
10:    i ← i + 1
11: end while
12: return (k_{i−1}, ..., k_0)
```

Ko imamo skalar predstavljen v NAF predstavitvi, lahko z algoritmom (24) izračunamo produkt točke s skalarjem. Naj omenimo še, da lahko algoritma (23) in (24) združimo. To je možno kljub temu, da algoritem (23) računa NAF predstavitev od desne proti levi, medtem ko algoritem (24) bere skalar od leve proti desni.

Algoritem 24 Množenje s skalarjem v NAF predstavitvi

VHOD: $P = (x, y)$, $p > 3$, NAF$(k) = (k_{t-1}, \ldots, k_0)$, $k_i \in \{-1, 0, 1\}$,
$a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $R = [k]P$

```
1: R ← O
2: for i = t − 1 to 0 do
3:     R ← [2]R
4:     if k_i = 1 then
5:         R ← R ⊕ P
6:     else if k_i = −1 then
7:         R ← R ⊖ P
8:     end if
9: end for
10: return R
```

Metode z drsečim oknom

V prejšnji metodah smo skalar, predstavljen v binarni/NAF predstavitvi, brali od leve proti desni oziroma od desne proti levi, vsako števko posebej. Tak pristop ni ravno najbolj učinkovit, saj bi lahko upoštevali več števk hkrati. V primeru, da bi upoštevali $w = 2$ števk hkrati, bi na vsakem koraku metode prišteli eno izmed $2^w = 4$ točk. Nato bi se lahko pomaknili za 2 mesti naprej. Tako bi znatno pohitrili metodo, po drugi strani pa bi potrebovali več prostora, zapomniti bi se namreč morali $2^w = 4$ točk eliptične krivulje. Izkaže se, da ne potrebujemo shraniti vseh 2^w točk, ampak je dovolj približno polovico toliko (sode produkte s skalarjem lahko dobimo iz lihih z množenjem z 2). Metode s takim pristopom imenujemo *metode drsečega okna*, saj upoštevanje w števk hkrati zgleda kot premikanje okna velikosti w po števkah skalarja.

Ker je odštevanje približno enako zahtevna operacija kot seštevanje, lahko tudi pri algoritmu (25) uporabimo tudi NAF zapis skalarja k.

Definicija 4.6. Naj bo $w \geq 2$. w-NAF predstavitev pozitivnega števila k, označili jo bomo z $\mathrm{NAF}_w(k)$, je izraz oblike

$$k = \sum_{i=0}^{t-1} k_i 2^i,$$

kjer je vsak neničelni koeficient k_i lih, $|k_i| < 2^{w-1}$, $k_{t-1} \neq 0$ in vsako zaporedje w-tih števk k_i vsebuje največ en neničeln element. Število t imenujemo *dolžina* w-NAF zapisa.

Izrek 4.2. *Lastnosti* w-NAF *predstavitve pozitivnega števila* k *so:*

- k *ima enolično predstavitev v* w-NAF *obliki,*
- $\mathrm{NAF}_2(k) = \mathrm{NAF}(k)$
- *dolžina binarnega zapisa števila* k *je za največ eno manjša od dolžine* w-NAF *zapisa,*
- NAF *zapis ima povprečno približno* $t/(w+1)$ *neničelnih elementov.*

Dokaz. Za dokaz glej [6, 187]. □

Primer 4.7. Naj bo $k = 1122334455$. Negativni koeficient $-k_i$ bomo zaradi preglednosti označili s $\bar{k_i}$, binarno predstavitev števila k pa s $(k)_2$. Izračunajmo w-NAF predstavitve števila k za $w \in \{2, 3, 4, 5, 6\}$.

Algoritem 25 Množenje s skalarjem z uporabo drsečega okna

VHOD: $P = (x, y)$, $w \geq 2$, $p > 3$, $k = (k_{t-1}, \ldots, k_0)_2$,
$a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $R = [k]P$

1: **for** $i \in \{1, 3, 5, \ldots, 2^w - 1\}$ **do**
2: $\quad P_i = [i]P$
3: **end for**
4: $R \leftarrow \mathcal{O}$
5: $i = t - 1$
6: **while** $i \geq 0$ **do**
7: $\quad$ **if** $k_i = 0$ **then**
8: $\quad\quad R \leftarrow [2]R$
9: $\quad\quad i \leftarrow i - 1$
10: $\quad$ **else**
11: $\quad\quad s \leftarrow max(i - k + 1, 0)$
12: $\quad\quad$ **while** $k_s = 0$ **do**
13: $\quad\quad\quad s \leftarrow s + 1$
14: $\quad\quad$ **end while**
15: $\quad\quad$ **for** $h = 0$ to $i - s + 1$ **do**
16: $\quad\quad\quad R \leftarrow [2]R$
17: $\quad\quad$ **end for**
18: $\quad\quad u \leftarrow (k_i, \ldots, k_s)_2$
19: $\quad\quad R \leftarrow R \oplus P_u$
20: $\quad\quad i \leftarrow s - 1$
21: $\quad$ **end if**
22: **end while**
23: **return** R

$$
\begin{array}{rl}
(k)_2 = & 1\,0\,0\,0\;\;0\;1\;0\,1\;1\;\;1\;\;0\,0\,1\,0\;1\;\;0\;\;1\,1\;1\,0\;1\,1\;0\,1\;1\,1\;1\,0\;1\,1\;1\\
NAF_2(k) = & 1\,0\,0\,0\;\;1\;\;0\;\bar{1}\,0\,0\;\bar{1}\;\;0\,1\,0\,\bar{1}\;0\;\bar{1}\;\;0\,0\,0\,\bar{1}\;0\,0\;\bar{1}\,0\,0\,0\,0\,\bar{1}\;0\,0\;\bar{1}\\
NAF_3(k) = & 1\,0\,0\,0\;\;0\;\;0\;3\,0\,0\;\bar{1}\;\;0\,0\,1\,0\,0\;3\;\;0\,0\,0\,\bar{1}\;0\,0\;\bar{1}\,0\,0\,0\,0\,\bar{1}\;0\,0\;\bar{1}\\
NAF_4(k) = & 1\,0\,0\,0\;\;0\;1\;0\,0\,0\;7\;\;0\,0\,0\,0\;5\;\;0\;\;0\,0\;7\,0\,0\,0\;7\,0\,0\,0\;\bar{1}\,0\,0\,0\;7\\
NAF_5(k) = & 1\;0\,0\,0\,0\;\bar{15}\,0\,0\,0\,0\;\bar{9}\;\;0\,0\,0\,0\,0\,11\;0\,0\,0\,0\,0\,0\;\bar{9}\,0\,0\,0\,0\,0\,0\,0\;\bar{9}\\
NAF_6(k) = & 1\,0\,0\,0\;\;0\;\;0\,0\,0\,0\,0\,23\;0\,0\,0\,0\,0\,11\;0\,0\,0\,0\,0\,0\;\bar{9}\,0\,0\,0\,0\,0\,0\,0\;\bar{9}
\end{array}
$$

Za lažje razumevanje algoritma (26) bomo uvedli notacijo k mods 2^w, ki označuje tako število u, da velja $u = k \pmod{2^w}$ in $-2^{w-1} \leq u \leq 2^{w-1}$. Naslednji algoritem učinkovito izračuna w-NAF predstavitev števila.

Algoritem 26 Izračun w-NAF predstavitve naravnega števila

VHOD: $k > 0$, $w \geq 2$
IZHOD: $\mathrm{NAF}_w(k) = (k_{t-1}, \ldots, k_0)$, $|k_i| \in \{0, 1, 3, 5, \ldots, 2^{w-1} - 1\}$

1: $i \leftarrow 0$
2: **while** $k \geq 1$ **do**
3: **if** $k = 1 \pmod 2$ **then**
4: $k_i \leftarrow k$ mods 2^w
5: $k \leftarrow k - k_i$
6: **else**
7: $k_i \leftarrow 0$
8: **end if**
9: $k \leftarrow k/2$
10: $i \leftarrow i + 1$
11: **end while**
12: **return** $(k_{i-1}, \ldots, k_0)$

Primer 4.8. Naj bo $p = 2003$, E eliptična krivulja, definirana z enačbo (4.2), nad obsegom $\mathbb{Z}_p$ in $P = (1118, 269)$. Izračunajmo $[763]P$ z algoritmom (25) in velikostjo okna $w = 3$.

$$
\begin{aligned}
P_1 &= [1]P = (1118, 269) & P_5 &= [5]P = (851, 77)\\
P_3 &= [3]P = (1081, 1674) & P_7 &= [7]P = (663, 1787)
\end{aligned}
$$

Ker je dvojiška predstavitev števila 763 enaka $(\underset{5}{\underline{101}}\ \underset{7}{\underline{111}}\ \underset{5}{\underline{101}}\ \underset{1}{\underline{1}})_2$, so vmesni rezultati algoritma naslednji.

Algoritem 27 Množenje s skalarjem v w-NAF predstavitvi

VHOD: $P = (x, y)$, $w \geq 2$, $p > 3$, $\text{NAF}_w(k) = (k_{t-1}, \ldots, k_0)$, $|k_i| \in \{0, 1, 3, 5, \ldots, 2^{w-1} - 1\}$,
$a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$, $p = (p_{n-1}, \ldots, p_0)$
IZHOD: $R = [k]P$

1: **for** $i \in 1, 3, 5, \ldots, 2^{w-1} - 1$ **do**
2: $\quad P_i = [i]P$
3: **end for**
4: $R \leftarrow \mathcal{O}$
5: **for** $i = t - 1$ to 0 **do**
6: $\quad R \leftarrow [2]R$
7: $\quad$ **if** $k_i > 0$ **then**
8: $\quad\quad R \leftarrow R \oplus P_{k_i}$
9: $\quad$ **else if** $k_i < 0$ **then**
10: $\quad\quad R \leftarrow R \ominus P_{-k_i}$
11: $\quad$ **end if**
12: **end for**
13: **return** R

$$
\begin{array}{lll}
[5]P = (851, 77) & [10]P = (4, 640) & [20]P = (836, 807) \\
[40]P = (1378, 1696) & [47]P = (1534, 747) & [94]P = (1998, 1094) \\
[188]P = (1602, 1812) & [376]P = (478, 1356) & [381]P = (1454, 981) \\
[762]P = (1970, 823) & [763]P = (1453, 1428) &
\end{array}
$$

V primeru NAF zapisa števila $763 = (\underset{3}{\underline{10\bar{1}}}00000\underset{-5}{\underline{\bar{1}0\bar{1}}})$, dobimo naslednje vmesne rezultate.

$$
\begin{array}{lll}
[3]P = (1081, 1674) & [6]P = (255, 1499) & [12]P = (459, 1270) \\
[24]P = (41, 1867) & [48]P = (1461, 904) & [96]P = (1966, 1808) \\
[192]P = (892, 529) & [348]P = (1928, 1803) & [768]P = (799, 1182) \\
[763]P = (1453, 1428) &&
\end{array}
$$

Zadnji korak $[763]P1 = [768]P1 \oplus [-5]P$ je zaradi predznačene NAF predstavitve poseben in ga zato realiziramo kot odštevanje $[768]P1 \ominus [5]P$.

4.6 Kompresija/dekompresija točke v $\mathbb{Z}_p$

Kadar imamo počasno omrežje oziroma kadar je prenos podatkov draga operacija, želimo po prenosnem kanalu prenesti čim manj bitov. Eliptične krivulje nad obsegom $\mathbb{Z}_p$ nam omogočajo, da lahko število prenosnih bitov zmanjšamo skoraj za polovico.

Eliptična krivulja E, definirana z enačbo (4.3), vsebuje največ dve točki z isto x-koordinato. Ti dve točki sta $P = (x, y)$ in $-P = (x, -y)$. V primeru da obstaja samo ena točka z določeno x-koordinato, potem mora veljat $P = -P$ oziroma $y = 0$.

Kompresija

Če želimo točko $P = (x, y)$ kompresirati, potem si zapomnimo x-koordinato in še bit $b(y) = y \mod 2$. Ker na eliptični krivulji obstajata samo dve točki z isto x-koordinato, za njiju velja, da sta koordinati y različne parnosti. Namreč, če je $y \mod p$ sodo število, potem je $-y = p - y \mod p$ liho število, saj je vsako praštevilo $p > 2$ sodo. Tako pri pošiljanju točke (x, y) pošljemo le $P_k = (x, b(y))$ in s tem zmanjšamo število prenosnih bitov skoraj za polovico. Omenimo še, da je kompresija zelo nezahtevna operacija, saj potrebuje preverjanje enega samega bita.

Dekompresija

Pri dekompresiji točke $P_k = (x, b(y))$ je potrebno veliko več dela. Prvo je potrebno izračunati vrednost $y^2 = x^3 + ax + b$, kar mora biti kvadrat v $\mathbb{Z}_p$. To lahko hitro preverimo z izračunom Jacobijevega simbola. V primeru, da je Jacobijev simbol različen od 1, je pri prenosu prišlo do napake ali pa je nekdo namerno poslal napačno kompresirano točko. Če je Jacobijev simbol enak 1, potem je bila prejeta točka pravilno kompresirana. Sledi izračun kvadratnega korena števila y^2. Ker ima kvadratni koren v $\mathbb{Z}_p$ dve rešitvi, s pomočjo bita $b(y)$ preverimo, kateri rešitvi ustreza kompresirana točka.

Primer 4.9. Naj bo $p = 2003$ in E eliptična krivulja, definirana z enačbo (4.4), nad obsegom $\mathbb{Z}_p$. Potem lahko točko $P = (1120, 1391)$ kompresiramo v $P_k = (1120, 1)$. Če želimo točko $P_k = (1120, 1)$ dekompresirati, moramo najprej izračunati kvadratni koren od

$$1120^3 + 1132 \cdot 1120 + 278 \cdot 1120 = 1406196118 = 1986 \pmod{2003}.$$

Kvadratni koren po modulu 2003 od 1986 je ali 612 ali 1391. Ker je bil poleg koordinate x poslan tudi bit parnosti $b(y) = 1$ vemo, da je iskani kvadratni koren liho število. Poslana je bila torej točka $P = (1120, 1391)$.

Poglavje 5

Implementacija in testiranje

V tem poglavju bomo opisali okolje, v katerem smo implementirali tako aritmetiko praštevilskih obsegov, kot aritmetiko na eliptičnih krivuljah. Opisali bomo, kateri programski jezik smo si izbrali, kako smo predstavili števila, kako smo stvari implementirali in na kakšen način smo testirali pravilnost metod.

Programski jezik C

Programski jezik C je splošno-namenski, višje-nivojski, proceduralni programski jezik. Razvil ga je Dennis Ritchie v Bell-ovih laboratorijih (*AT&T Bell Labs*) med leti 1969 in 1973. Precejšni delež k nastanku sta prispevala tudi Ken Thompson in Martin Richards, kot avtorja programskih jezikov B in BCPL. Slednja dva štejeta kot njegova predhodnika in po njima naj bi dobil tudi ime C .

Nastanek C-ja je v veliki meri povezan z razvojem operacijskega sistema UNIX. Ta je bil prvotno napisan v zbirnem jeziku za DEC-ov mini-računalnik PDP-7. Ko so avtorji želeli prenesti UNIX na 16-bitni mini-računalnik PDP-11, so ugotovili, da ima programski jezik B kar nekaj pomanjkljivosti. Ena izmed njih je bila tudi nesposobnost naslavljanja bajta podatkov v pomnilniku. To je povzročilo nastanek prve verzije programskega jezika C. Osnovna verzija UNIX sistema za PDP-11 je bila vseeno napisana v zbirnem jeziku, a do leta 1973 skoraj v celoti prepisana v C. UNIX je tako postal eden izmed prvih operacijskih sistemov, ki ni bil napisan v zbirnem jeziku.

Leta 1978 sta Brian Kernighan in Dennis Ritchie objavila prvo izdajo knjige *Programski jezik C* (angl. *The C Programming Language*). Ta knjiga je dolga leta služila kot neformalna specifikacija programskega jezika C, bolje poznanega pod imenom *K&R C*. Leta 1983 je ANSI (American National Standards Institute) ustanovil skupino za standardizacijo programskega jezika C.

Standard je bil tako sprejet leta 1989 in ratificiran pod ANSI X3.159-1989. Tej verziji programskega jezika C pravimo tudi *Standard C, C89* ali *ANSI C*, opisan pa je tudi v drugi izdaji zgoraj omenjene knjige. Leta 1990 je ANSI C z rahlimi spremembami sprejel tudi ISO (International Organization for Standardization) kot standard ISO/IEC 9899:1990. Tej verziji programskega jezika C pravimo tudi *C90* in je oznaka za isti programski jezik kot C89. V poznih devetdesetih letih pride do ponovne spremembe standarda in tako z objavo standarda ISO/IEC 9899:1999 programski jezik C dobi novo verzijo *C99*. Omenjeni standard med drugimi vpelje tudi `inline` funkcije, nove podatkovne tipe kot so `long long int` in `complex`, tabele katerih dolžina je določena ob izvajanju programa in enovrstične komentarje.

Programski jezik C je bil sprva namenjen programiranju sistemov, vendar se zaradi svojih prednosti (preprostost, prenosljivost, hitrost, ...) uporablja tudi pri razvijanja uporabniških programov.

5.1 Implementacija

Vse programe opisane v 3. in 4. poglavju bomo implementirali v programskem jeziku C. Za njega smo se odločili predvsem zaradi hitrosti in preprostosti. Izbrali smo ga tudi zaradi uporabnosti/združljivosti, kar nam bo posebno prav prišlo pri testiranju, ko bomo s pomočjo ostalih knjižnic in programov testirali naš program. Za programski jezik C obstaja tudi veliko dobrih prevajalnikov. Mi bomo naš program prevedli s prevajalnikom *gcc*.

Celotni program bo namenjen delovanju na lastnem osebnem računalniku, torej na AMD Athlonu 2200MHz, s 1024 Mb spomina. Od končne verzije programa ne bomo zahtevali, da bo deloval tudi na ostalih računalnikih. To pa zato, ker bomo v 6. poglavju zaradi optimizacije uporabili specifične ukaze v zbirnem jeziku za naš procesor.

Glavni cilj implementacije je čim hitrejše izvajanje osnovnih eliptičnih operacij, kot so seštevanje, odštevanje in množenje točke s skalarjem, na 192 bitnih eliptičnih krivuljah. S pomočjo implementacije bomo tako zlahka ugotovili katere metode so bolj in katere manj primerne za uporabo. Tu imamo predvsem v mislih, ali je bolje računati s projektivnimi ali z afinimi koordinatami.

Implementacijo bomo razdelili na 2 vsebinsko različna dela.

- V prvem delu se bomo posvetili aritmetiki v praštevilskem obsegu. Tu bomo implementirali vse metode 3. poglavja, ki bodo služile kot podlaga

za aritmetiko na eliptičnih krivuljah. Njihova hitrost tako igrala ključno vlogo.

- Drugi del je namenjen celotni aritmetiki eliptičnih krivulj. V ta del tako sodijo vse metode 4. poglavja z izjemo metod (26) in (27). Slednjih dve namreč ne moremo združiti, saj metoda (26) računa w-NAF predstavitev števila od desne proti levi, medtem ko metoda (27) bere w-NAF predstavitev od leve proti desni.

5.2 Testiranje

Aritmetika v $\mathbb{Z}_p$

Ko smo celotno aritmetiko 3. poglavja sprogramirali, je potrebno preveriti še njeno pravilnost. Poslužili se bomo testiranja posameznih primerov. Izbrali si bomo poljuben vhod metode in preverili pravilnost pripadajočega izhoda. Če ta postopek opravimo na vseh možnih vhodih, potem smo pravzaprav dokazali pravilnost metode. Pri 192 bitnih številih, bi tako za seštevanje morali preveriti $2^{192} \cdot 2^{192}$ vhodnih primerov, kar je občutno preveč. Zato bomo vhodne primere raje izbirali naključno in se zadovoljili, če naša metoda na vseh vrne pravilni rezultat.

Preverjanje pravilnosti rezultata lahko realiziramo na več načinov. Lahko bi si izbrali neko poljubno, že implementirano aritmetiko velikih števil (npr: *GMP*, *BigNum Math*, *OpenSSL*, ...) in jo vzporedno poganjali na istih primerih. V primeru, da sta rezultata enaka, je naša metoda vrnila pravilni rezultat. Tako testiranje predpostavlja, da poljubno izbrana implementacija pravilno deluje.

Naše testiranje bo potekalo s pomočjo skriptnega jezika *python*. Ta zna operirati s poljubno dolgimi števili, poleg tega pa vključevanje pythonovega interpreterja v C izjemno preprosto.

Primer 5.1. Naj bo $W = 32$, $n = len = 1$ in testirajmo s pomočjo metode (5.1) seštevanje v praštevilskem obsegu. Predpostavili bomo, da se v 10. in 11. vrstici števili a in b nastavita na 0x00002008 in 0x00111111. V vrsticah 12-15 števili a in b seštejemo z metodo *add* in rezultat zapišemo kot zaporedje znakov v spremenljivko *buffer=*"0x00113119". Vrstice 23-32 so namenjene ustvarjanju ukaza, ki ga bomo kasneje podali interpreterju. V našem primeru spremenljivka *buf* ustreza nizu "*a*=`0x00002008+0x00111111`". Ta niz nato skupaj z nizom "0x00113119" podamo metodi (5.2). Slednja v interpreterju

izvede ukaza a=0x00002008+0x00111111 in a=str(hex(a)). Prvi ukaz je namenjen izračunu produkta, katerega rezultat je zapisan v pythonov objekt a. Drugi ukaz pretvori vrednost produkta a najprej v šestnajstiško obliko in nato še v niz. Končno v 21. vrstici primerjamo niza "0x00113119" in "0x113119".

Koda 5.1: Testiranje seštevanja

```
1  void test_add(int len, int numRep) {
2      WORD carry;
3      WORD a[len];
4      WORD b[len];
5      WORD c[len];
6      char buffer[len*2*sizeof(WORD)+1];
7
8
9      for (; numRep > 0; numRep--) {
10         random_num(a, len);
11         random_num(b, len);
12
13         add(a, b, c, len, &carry);
14
15         toString(a, len, buffer);
16         printf("a  =  %s\n", buffer);
17
18         toString(b, len, buffer);
19         printf("b  =  %s\n", buffer);
20
21         toString(c, len, buffer);
22         printf("c  =  %s\n", buffer);
23
24         char buf[7+4*len*sizeof(WORD)+1];
25         buf[0] = 'a';
26         buf[1] = '=';
27         buf[2] = '0';
28         buf[3] = 'x';
29         toString(a, len, buf+4);
30         buf[4+len*2*sizeof(WORD)] = '+';
31         buf[5+len*2*sizeof(WORD)] = '0';
32         buf[6+len*2*sizeof(WORD)] = 'x';
33         toString(b,len,buf+7+len*2*sizeof(WORD));
34
35         run_py(buf, buffer);
36     }
37 }
```

Koda 5.2: Koda za preverjanje rezultata

```
void run_py(char *py_command, char *method_result){
    PyObject * module, * dict, * obj;
    char * res;

    Py_Initialize();

    // run pyhon
    PyRun_SimpleString(py_command);
    // convert to hexadecimal
    PyRun_SimpleString("a = str(hex(a))");
    // get result
    module = PyImport_AddModule("__main__");
    dict = PyModule_GetDict(module);
    obj = PyMapping_GetItemString(dict, "a");
    res = PyString_AsString(obj);

    // print result
    printf("p = %s\n", res);

    // compare both results
    compare(method_result, res);

    Py_Finalize();
}
```

Naj omenimo še, da za vsako metodo ni potrebno klicati pythonovega interpreterja. Za primer vzemimo metodo *inv_ea*. Če predpostavimo, da metode za množenje in modularno redukcijo delujejo pravilno, lahko pravilnost izhoda a^{-1} preverimo z enačbo $aa^{-1} \equiv 1 \pmod{p}$.

Aritmetika na eliptičnih krivuljah

Aritmetiko na eliptičnih krivuljah bomo testirali s pomočjo knjižnice *OpenSSL* (`http://www.openssl.org/`). Testiranje metod bo potekalo samo na določenih krivuljah, ki se nahajajo v tej knjižnici. Vsaka eliptična krivulja bo podana s kratkim opisom, s koeficienti a in b, s praštevilskim modulom p, generatorsko točko s koordinatama x in y ter z njenim redom. Testirali bomo na podoben način kot pri aritmetiki v $\mathbb{Z}_p$. Torej za poljubne vhodne podatke bomo preverili, če se izhodni podatki ujemajo z izhodnimi podatki OpenSSL-a.

Vhodni podatki metod iz poglavja (4) bodo vedno vsebovali tudi neko točko eliptične krivulje. Te si ne moremo izbrati naključno, saj mora ustrezati enačbi (4.3). Izračun točke po tej enačbi je zahtevna operacija, zato bomo

namesto naključne točke raje uporabili generator. Pravilnost metod bomo preverili z zaporednim prištevanjem generatorja samemu sebi, m generatorja in z množenjem generatorja z naključno izbranim skalarjem.

Koda 5.3: Koda za inicializacijo krivulje/točk

```
1  // OPENSSL init
2  BN_CTX *ctx = NULL;
3  BIGNUM *p, *a, *b, *x, *y;
4  EC_GROUP *group;
5  EC_POINT *P,*Q;
6
7  p = BN_new(); a = BN_new(); b = BN_new();
8  x = BN_new(); y = BN_new();
9
10 if (!BN_hex2bn(&p, (*crv).p))     ABORT;
11 if (!BN_hex2bn(&a, (*crv).a))     ABORT;
12 if (!BN_hex2bn(&b, (*crv).b))     ABORT;
13
14 group = EC_GROUP_new(EC_GFp_mont_method());
15 EC_GROUP_set_curve_GFp(group, p, a, b, ctx);
16
17 P = EC_POINT_new(group);
18 Q = EC_POINT_new(group);
19 BN_hex2bn(&x, (*crv).x);
20 BN_hex2bn(&y, (*crv).y);
21 EC_POINT_set_affine_coordinates_GFp
22                    (group, P, x, y, ctx);
23 EC_POINT_set_affine_coordinates_GFp
24                    (group, Q, x, y, ctx);
25
26 // MY APP init
27 point_aff *PP, *QQ;
28 curve *E;
29
30 int len = (*crv).len / (8 * sizeof(WORD));
31 E = init_curve(crv);
32 PP = init_point_aff((*crv).x, (*crv).y, len);
33 QQ = init_point_aff((*crv).x, (*crv).y, len);
```

Koda 5.4: Koda za preverjanje seštevanja

```
void c_add_aff(int numRep,
               EC_POINT *P, EC_POINT *Q,
               EC_GROUP *group, BN_CTX *ctx,
               point_aff *PP, point_aff *QQ,
               curve *E, int len) {

   for (; numRep > 0; numRep--) {
      // OPENSSL
      EC_POINT_add(group, Q, P, Q, ctx);
      print_ssl_aff(Q, group, ctx);

      // MY APP
      add_aff(PP, QQ, QQ, E, (*E).p, len);
      print_aff(QQ, len);

      compare_points(QQ, Q, group, ctx, len);
   }
}
```

Poglavje 6

Analiza in izboljšava učinkovitosti

Vse algoritme 3. in 4. poglavja smo sprogramirali in testirali v 5 poglavju. Sedaj pride na vrsto pohitritev oziroma optimizacija. Kodo bomo optimizirali s pomočjo *analize zahtevnosti* (angl. *perforance analysis, profiling*), katera s podatki, zbranimi med izvajanjem programa, analizira obnašanje programa. Tukaj imamo v mislih predvsem prostorske in časovne zahtevnosti posameznih metod. Cilj analize je ugotoviti, katere metode oziroma kateri deli kode se izvajajo najdlje časa in kateri deli porabijo največ prostora. Ti deli kode predstavljajo ozko grlo našega programa in jih je potrebno izboljšati oziroma se jih v celoti znebiti.

Programu, ki izvaja analizo učinkovitosti, pravimo *analizator* (angl. *profiler*). Analizator med izvajanjem programa spremlja njegovo obnašanje, natančneje število in dolžino posameznih klicev funkcij. Vse te podatke si med izvajanjem zabeleži in nam jih po končanem izvajanju programa ponudi prikazane v različnih oblikah. Najbolj pogosti obliki sta:

sled (angl. *trace*) - prikaz toka zabeleženih dogodkov,

opis (angl. *profile*) - prikaz statistike opazovanih dogodkov.

Naše zanimanje bo usmerjeno predvsem v drugo obliko, katero lahko prikažemo na dva različna načina.

Kratek opis (angl. *flat profile*) je povzetek celotne informacije, zbrane med izvajanjem programa. Vsebuje število klicev, povprečni in skupni čas izvajanja posamezne funkcije, itd. S takim prikazom lahko hitro ugotovimo, katere dele programa je potrebno izboljšati.

Graf klicev (angl. *Call graph profile*) nam za vsako funkcijo pokaže, kolikokrat je bila klicana s strani drugih funkcij ali sebe. S takim prikazom lažje ugotovimo medsebojne relacije funkcij. Glede na relacije lahko sklepamo, katere funkcije bi bilo potrebno odstraniti in katere nadomestiti z novimi.

Za zbiranje podatkov se analizator poslužuje številnih tehnik, kot so vstavljanje dodatne kode na začetek in konec funkcij, uporaba posebnih registrov za merjenje zahtevnosti, uporaba prekinitev in še marsikaj.

Predenj se bomo spustili v optimizacijo, bomo najprej upoštevali idejo spodnjih dveh citatov. Izkaže se namreč, da večina napisanih programov sledi nenapisanem pravilu 80:20. Ta pravi, da se bo 20% celotne kode izvajalo 80% časa. To pravilo bomo v nadaljevanju upoštevali in tako izboljševali samo tiste dele programa, ki porabijo največ časa.

> *More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.*
> **William A. Wulf**

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*
> **Donald E. Knuth**

Primer 6.1. Denimo da naš program za svoje izvajanje porabi 100s časa, od tega $80s$ v metodi A in $20s$ v metodi B. Sedaj bi bilo nesmiselno izboljševati metodo B. V primeru, da bi čas, potreben za njeno izvajanje, zmanjšali za polovico, torej na $10s$, bi celotni program še zmeraj rabil $90s$ za svoje delovanje. Zato se raje osredotočimo na metodo A, saj z 12% pohitritvijo dosežemo enak učinek.

Analizator gprof

Za analizo učinkovitosti naše implementacije bomo uporabili program *gprof*. Njegova zgodovina sega daleč nazaj v leto 1979, ko je UNIX operacijski sistem vseboval orodje *prof*. Ta je beležil, koliko procesorskega časa je posamezna funkcija potrebovala za svoje delovanje. Leta 1982 se je prof, z dodano podporo za izpis grafa klicev, preimenoval v gprof.

Uporaba analizatorja gprof s prevajalnikom gcc je preprosta in je sestavljena iz treh korakov:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 27.77      1.23     1.23 27949536     0.04     0.04  sub
 18.51      2.05     0.82 20835130     0.04     0.04  add
 17.38      2.82     0.77 53523002     0.01     0.01  div_by_2
 11.96      3.35     0.53   100000     5.30    38.49  inv_bin
  7.67      3.69     0.34  1000000     0.34     0.36  mul2
  6.32      3.97     0.28 27149246     0.01     0.01  equal_one
  4.29      4.16     0.19 13999402     0.01     0.01  geq
  2.03      4.25     0.09 13849646     0.01     0.07  sub_mod
  1.81      4.33     0.08   200000     0.40     0.42  sqr2
  1.13      4.38     0.05  2300000     0.02     0.02  copy_num
  1.13      4.43     0.05   400000     0.12     0.91  mod_barret
  0.00      4.43     0.00   400000     0.00     0.07  add_mod
  0.00      4.43     0.00   200000     0.00     0.00  equal
  0.00      4.43     0.00   100000     0.00    44.30  add_aff
```

Slika 6.1: Izpis programa gprof za metodo `add_aff`

- izvorno kodo `moj_program.c` prevedemo z dodatnim parametrom `-pg`,
- program `moj_program.out` poženemo,
- z ukazom `gprof moj_program.out gmon.out -možnosti` izpišemo statistiko.

Optimizacija kode

Prevajalnik gcc

Optimizacijo kode bomo pričeli pri prevajanju. Večina sodobnih prevajalnikov, med njimi je tudi gcc, zna optimizirati kodo. Z dodatnimi parametri pri prevajanju prevajalniku sporočimo, kako želimo optimizirati kodo. Navadno kodo optimiziramo tako, da prevedeni program zasede čim manj prostora ali teče čim hitreje. Na žalost sta ta dva načina optimizacije medsebojno zelo povezana,

zato se nam lahko zgodi, da pri optimizaciji hitrosti naraste velikost kode in obratno.

Prevajalnik gcc razdeli optimizacijo v tri nivoje. Glede na posamezni nivo se lahko uporabnik odloči, katere možnosti želi uporabiti pri optimizaciji in katere ne. Navadno prevajanje ne vključuje optimizacije in jo je potrebno vključiti s parametrom `-O`, kateremu sledi nivo optimizacije.

- Prvi nivo optimizacije je namenjen optimizaciji velikosti in hitrosti kode v čim krajšem času. Taka optimizacije ne vpliva bistveno na čas prevajanja in zato tudi ni zelo učinkovita.

- Drugi nivo se poslužuje optimizacij znotraj določene arhitekture. Pri tem poskuša ohraniti razmerje med velikostjo in hitrostjo programa. Tako ne uporablja vrinjenih funkcij (angl. *inline functions*) in razvijanja zank (angl. *loop unrolling*).

- Tretji nivo z vsemi svojimi sredstvi optimizira hitrost, medtem ko velikost kode zanemari. Taka optimizacija lahko zaradi vrinjenih funkcij poveča velikost kode, kar lahko posledično vpliva tudi na hitrost izvajanja. Kmalu se lahko zgodi, da velikost prevedene kode presega velikosti ukaznega predpomnilnika (angl. *instruction cache*), kar lahko privede do večjega števila zgrešitev v predpomnilniku. Tako je včasih bolje, da program optimiziramo na nivoju 2 kot na nivoju 3.

Poleg nivoja optimizacije lahko prevajalniku podamo tudi tip arhitekture, za katero prevajamo program. Tip arhitekture gcc prevajalniku podamo s parametrom `-march=tip_arhitekture`.

Naše programe bomo prevajali s parametri `-O3 -march=athlon`, saj želimo čim hitrejšo optimizacijo kode za lastni računalnik.

Vrinjene funkcije

Programski jezik C nam omogoča uporabo *vrinjenih funkcij*. Vrinjena funkcija se v programski kodi od navadne funkcije razlikuje le v deklaraciji. Njena deklaracija uporablja rezervirano besedo *inline*, ki je pravzaprav napotek prevajalniku, da naj vsak klic vrinjene funkcije zamenja s celotnim telesom funkcije. Tako smo celotno funkcijo pravzaprav 'vrinili' v kodo, od koder je bila funkcija klicana.

Vrinjene funkcije imajo svoje prednosti in slabosti. Največja prednost je v hitrosti izvajanja programa. Ker je prevajalnik namesto klica funkcije vstavil njeno telo, se znebimo klica funkcije. Znebili pa smo se tudi časovnega zamika,

ki bi se v prvotnem primeru pojavi zaradi klica. Po drugi strani pa se je zaradi vstavljanja celotnega telesa povečala velikost kode. Če smo vrinjeno funkcijo klicali na stotih različnih mestih, potem se je telo funkcije stokrat kopiralo v kodo. Od tod lahko sklepamo, da je dobro vrivati samo manjše funkcije, ki se v kodi malokrat pojavijo, a se med tekom programa večkrat kličejo.

V naši implementaciji je veliko kandidatov za vrinjene funkcije. Vse osnovne funkcije, kot so seštevanje, odštevanje in deljenje z 2, se iz ostalih funkcij večkrat kličejo. Poleg tega pa so zelo kratke, kar pomeni, da se koda po vrivanju funkcij ne bo pretirano razširila.

Zbirni jezik

Zadnjo stopnjo optimizacije bomo opravili s pomočjo analizatorja gprof in zbirnega jezika za naš procesor. Ne pozabimo, da želimo optimizirati hitrost osnovnih operacij na eliptičnih krivuljah. V ta namen bomo vse metode eliptičnih krivulj pognali in s pomočjo analizatorja ugotovili, katere funkcije porabijo največ procesorskega časa.

Začeli smo s testiranjem metode seštevanja in podvajanja v afinih koordinatah. Slika (6.1) nam prikazuje izpis analizatorja gprof po 10.000 klicih funkcije `add_aff`. Od tod lahko takoj ugotovimo, da so največji porabniki časa metode `add, sub, div_by_2`. To seveda ni nič novega, saj so to osnovne metode, ki se na splošno velikokrat uporabljajo. Podobni izpis dobimo tudi, če poženemo metodo podvajanja. Za povečanje hitrosti seštevanja in podvajanja smo vse tri metode prepisali v zbirni jezik.

Nadaljevali smo z analizo metod seštevanja in podvajanja v projektivnih koordinatah. Podobno kot v prejšnjem primeru, tudi tu večkrat poženemo obe metodi in ju analiziramo. Dobljeni rezultati se popolnoma razlikujejo od prejšnjih. Najbolj klicana oziroma najbolj potratna funkcija je bila metoda množenja, katera je skoraj povsod porabila več kot 60% celotnega časa. Za pohitritev metod v projektivnih koordinatah smo množenje prepisali v zbirni jezik.

Ostale so nam še metode množenja s skalarjem. Tudi pri vseh teh metodah analizator vrne isti rezultat. Kadar računamo z afinimi koordinatami, so najbolj potratne funkcije seštevanja, odštevanja in deljenja z 2, medtem ko računanje s projektivnimi koordinatami v največji meri upočasnjuje metoda množenja.

Rezultati optimizacije

Hitrosti metod bomo testirali s pomočjo analizatorja gprof. Le ta nam vrne tudi čas, ki ga je metoda potrebovala za svoje delovanje. Mogoče ta čas ni najbolj natančen, vendar so rezultati dovolj dobri za osnovno primerjavo posameznih metod med seboj. Hitrosti posameznih metod bomo prikazali v grafu, kjer bomo uporabili naslednje notacije.

neoptimizirano pomeni program, dobljen s prevajanjem osnovne kode brez kakršnekoli optimizacije. Ta implementacija ni odvisna niti od dolžine števil niti od arhitekture.

gcc -O3 je prevedena osnovna koda s tretjim nivojem optimizacije prevajalnika gcc.

gcc -O3 + inline vključuje prejšnjo optimizacijo, poleg tega pa so osnovne funkcije vrinjene v kodo.

gcc -O3 + asm + inline vključuje prejšnjo optimizacijo, poleg tega pa so časovno najbolj zahtevne funkcije prepisane v zbirni jezik

mod + R pomeni modularno Barretovo redukcijo, kjer recipročno vrednost računamo sproti.

mod - R pomeni modularno Barretovo redukcijo, kjer je recipročna vrednost vnaprej izračunana.

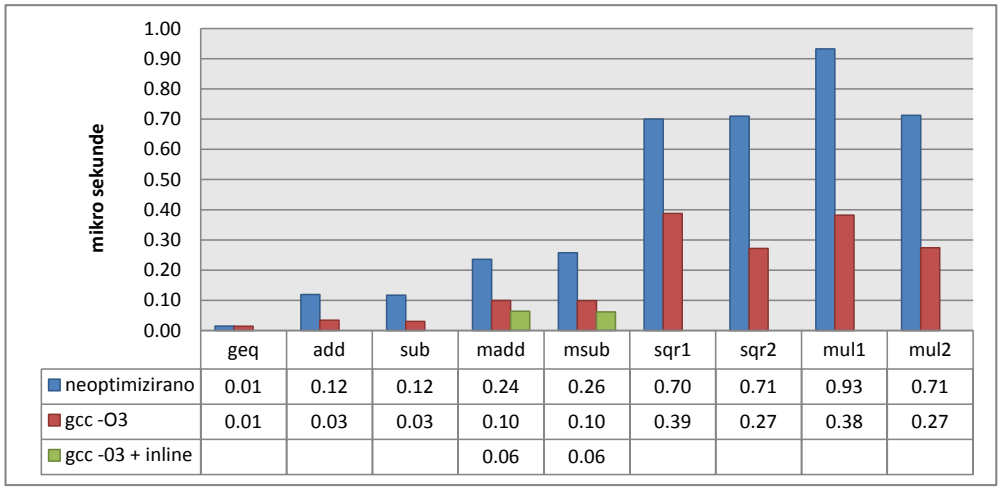
a_metoda je oznaka za metodo, ki računa z afinimi koordinatami.

p_metoda je oznaka za metodo, ki računa s projektivnimi koordinatami.

wX je oznaka za metodo drsečega okna, z velikostjo okna X bitov.

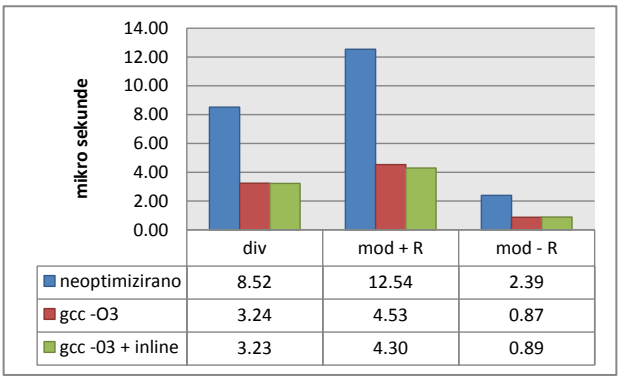
Osnovne funkcije v $\mathbb{Z}_p$ bomo testirali na 192 bitnih številih. Vsako število bomo predstavili s šestimi besedami dolžine 32 bitov. Metode za aritmetiko v $\mathbb{Z}_p$ večinoma niso odvisne od vhodnih podatkov, tako da lahko vse testiramo na istih številih.

Rezultati so podani na slikah (6.2), (6.3) in (6.4). Kot je bilo pričakovati, sta metodi seštevanja in odštevanja bistveno hitrejši od vseh ostalih metod. Od modularnega seštevanja in odštevanja se razlikujeta približno za faktor 2, kar je tudi pričakovana vrednost, glede na vsa možna vhodna števila. Kot lahko vidimo iz slike (6.3), je računanje ostanka z modularno Barretovo redukcijo veliko hitrejše, kot računanje s pomočjo deljenja. Poleg tega se iz slike (6.4) vidi,



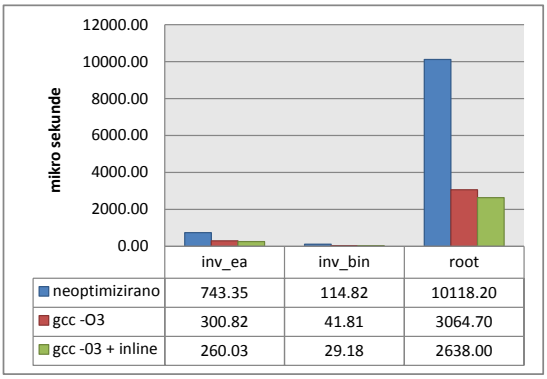
	geq	add	sub	madd	msub	sqr1	sqr2	mul1	mul2
neoptimizirano	0.01	0.12	0.12	0.24	0.26	0.70	0.71	0.93	0.71
gcc -O3	0.01	0.03	0.03	0.10	0.10	0.39	0.27	0.38	0.27
gcc -03 + inline				0.06	0.06				

Slika 6.2: Grafični prikaz hitrosti osnovnih funkcij v $\mathbb{Z}_p$ (a)



	div	mod + R	mod - R
neoptimizirano	8.52	12.54	2.39
gcc -O3	3.24	4.53	0.87
gcc -03 + inline	3.23	4.30	0.89

Slika 6.3: Grafični prikaz hitrosti osnovnih funkcij v $\mathbb{Z}_p$ (b)

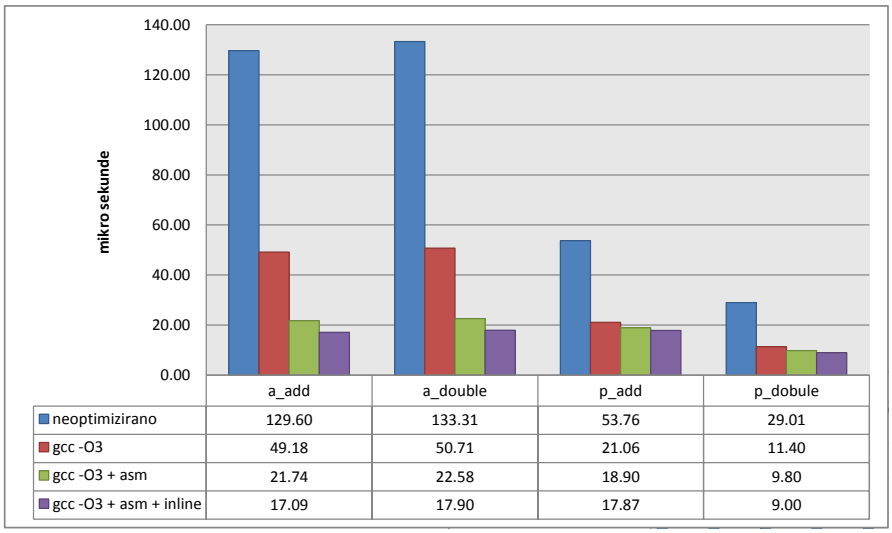


Slika 6.4: Grafični prikaz hitrosti osnovnih funkcij v $\mathbb{Z}_p$ (c)

da je deljenje prezahtevna operacija, saj binarni algoritem za izračun inverza deluje veliko hitreje od Evklidovega algoritma. Obe operaciji sta v primerjavi z osnovnimi operacijami počasni, a vseeno veliko hitrejši od korenjenja. Ta operacija je izmed vseh najpočasnejša, vendar to pravzaprav ni pomembno, saj korenjenje uporabljamo samo pri dekompresiji točke.

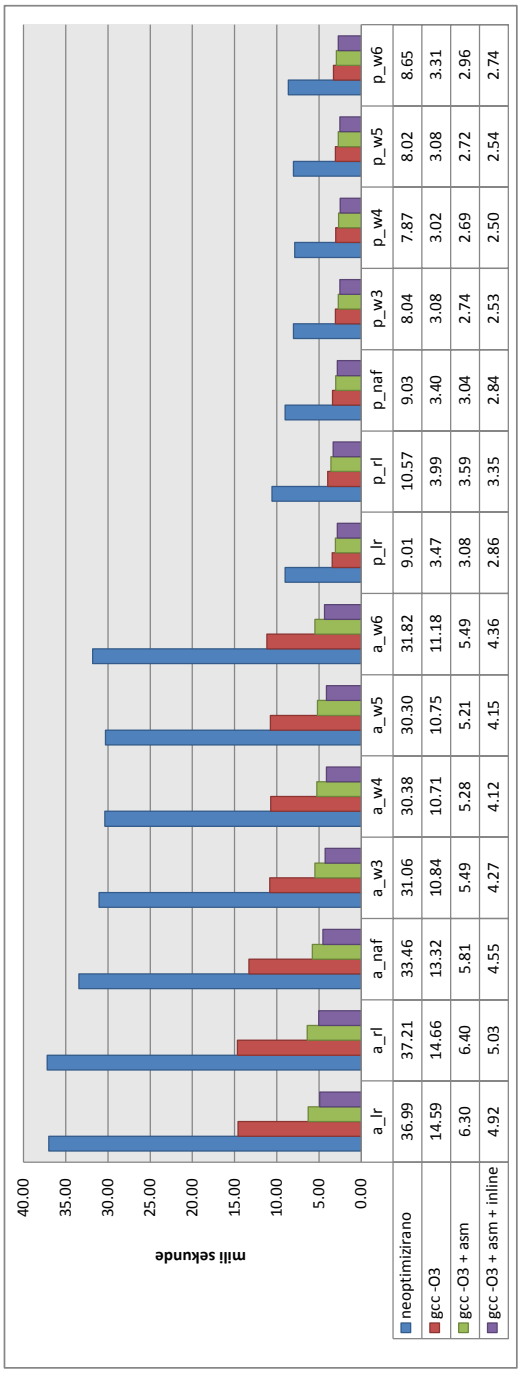
Metode eliptičnih krivulj bomo testirali na vnaprej izbrani 192 bitni eliptični krivulji. Izbrali si bomo generator te eliptične krivulje in z njim izmerili hitrost seštevanja/podvajanja in množenja s skalarjem. Pri množenju je potrebno posebno paziti, kako izbiramo skalarje. Napačna izbira lahko privede do nezaželenih rezultatov. Če si izbiramo take skalare, ki v predstavitvi z bazo 2 ne bodo imeli nobene ničle, medtem ko bodo v NAF predstavitvi skoraj vse števke enake nič, lahko testiranje privede do nepravilnih rezultatov. Mi bomo množenje s skalarjem večkrat ponovili, zato se bomo pri našem testiranju zadovoljili z naključnimi izbori skalarjev.

Rezultati metod so podani na slikah (6.5) in (6.6). Kot lahko opazimo, je seštevanje v projektivnih koordinatah občutno hitrejše od seštevanja v afinih. Razlog za to se skriva v hitrem množenju in v počasnem računanju inverza. Posledično je tudi množenje s skalarjem veliko hitrejša operacija v projektivnih koordinatah. Če primerjamo metode množenja s skalarjem med seboj, potem ugotovimo, da sta metodi drsečega okna in množenje z skalarjem v NAF predstavitvi, rahlo hitrejši od navadnega množenja s skalarjem.



	a_add	a_double	p_add	p_dobule
neoptimizirano	129.60	133.31	53.76	29.01
gcc -O3	49.18	50.71	21.06	11.40
gcc -O3 + asm	21.74	22.58	18.90	9.80
gcc -O3 + asm + inline	17.09	17.90	17.87	9.00

Slika 6.5: Grafični prikaz hitrosti osnovnih funkcij na eliptični krivulji (a)



Slika 6.6: Grafični prikaz hitrosti osnovnih funkcij na eliptični krivulji (b)

Poglavje 7

Zaključek

Po celotni implementaciji in analizi hitrosti lahko zaključimo, da je vse operacije na eliptičnih krivuljah nad $\mathbb{Z}_p$ priporočljivo opravljati v projektivnih koordinatah. Operaciji seštevanje in odštevanje sta občutno počasnejši v afinih koordinatah, kar posledično vpliva tudi na počasnost množenja s skalarjem. Slednjega je najbolje izvajati s skalarjem predstavljenim v NAF predstavitvi. Podobne hitrosti dosežemo tudi z metodo drsečega okna, tako da se lahko odločimo tudi za slednjo.

Velik del optimizacije nam opravi že sam prevajalnik. Taka optimizacija je enostavna, učinkovita in ne zahteva nič programerskega časa. Kadar nam taka optimizacija ne zadostuje, lahko najbolj zahtevne funkcije prevedemo v zbirni jezik. S tem na žalost izgubimo prenosljivost programa. V praksi naletimo tudi na primere, ko je potrebno računati samo nad praštevilskim obsegom fiksne dolžine. Takrat lahko implementacijo še pohitrimo in ob tem ohranimo prenosljivost. Kateri način optimizacije pride v poštev je seveda odvisno od posameznega problema.

Literatura

[1] H. Cohen in G. Frey, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, 2006.

[2] D. Hankerson, A. Menezes in S.A. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag, 2004

[3] M. Brown, D. Hankerson, J. Lopez in A. Menezes, *Software Implementation of the NIST Elliptic Curves Over Prime Fields*, In *Proc. CT-RSA*, Topics in Cryptology - CT-RSA 2001, LNCS 2020, pages 250–265, 2001

[4] K. Fong, D. Hankerson, J. Lopez in A. Menezes, *Field Inversion and Point Halving Revisited*, CORR 2003-18

[5] R. Lidl in H. Niederreiter, *Introduction to Finite Fields and their Applications*, Cambridge University Press, 1994

[6] G. Reitwiesner, *Binary arithmetic*, Advances in Computers, 1:231–308, 1960

Izjava

Izjavljam, da sem diplomsko nalogo izdelal samostojno pod vodstvom mentorja prof. dr. Aleksandra Jurišića. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.

Ljubljana, 11.09.2008 Peter Nose