UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika - uporabna smer (UNI)

Srečko Maksimovič

Učinkovita aritmetika v praštevilskih obsegih in implementacija eliptičnih krivulj

Diplomsko delo

Ljubljana, 2003

Zahvaljujem se vsem, ki so mi kakorkoli pomagali pri tem diplomskem delu. Posebna zahvala gre mojemu mentorju prof. dr. Aleksandru Jurišiću. Vsekakor ne smem pozabiti družine, Tine in prijateljev, ki so me ves čas študija vzpodbujali in mi pomagali.

Kazalo

PROGRAM DIPLOMSKEGA DELA

Učinkovita aritmetika v praštevilskih obsegih in implementacija eliptičnih krivulj

Delo naj predstavi matematične osnove, potrebne za razumevanje osnovnih računskih operacij končnih obsegov praštevilske karakteristike (praštevilski obsegi), ki so posebno zanimivi za kriptografijo. Glavna cilja sta predstavitev in primerjava učinkovitosti seštevanja, množenja, kvadriranja in deljenja v praštevilskih obsegih za implementiranje seštevanja na eliptični krivulji. Konkretno uporaba Mersenovih praštevil za pospešitev redukcije ter Lehmerjev algoritem za izboljšavo razširjenega Evklidovega algoritma, s katerim izračunamo multiplikativni inverz.

Mentor: A. Jurišić

Literatura:

M. Brown and D. Hankerson, J. Lopez and 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.
(Glej `http://cacr.math.uwaterloo.ca/~ajmeneze/publications/ecc-prime.ps`)

J. Sorenson, An Analysis of Lehmer's Euclidean GCD Algorithm. In *Proc. AMC ISSAC'95 Symp.*, pages 254–258, 1995.

D. E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, Vol. 2, Addison-Wesley, Reading, Ass., 2nd. edition, 1981.

Douglas R. Stinson, *Cryptography – Theory and Practice*, CRC Press, 1995.

A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press (Series on Discrete Mathematics and its Applications), 4th ed, 1999.

POVZETEK

V tem delu bomo predstavili praštevilske obsege in eliptične krivulje nad temi obsegi. Navedli bomo algoritme za osnovne aritmetične operacije v praštevilskih obsegih. Predstavili bomo tudi pseudo Mersennova praštevila in navedli algoritme za hitro modularno redukcijo, ki jo ta praštevila omogočajo. Glavni del te naloge je študij treh algoritmov (Evklidov, Binarni in Lehmerjev) za izračun inverza v praštevilskih obsegih. Predstavili bomo tudi rezultate empiričnih testiranj.

Math. Subj. Class(2000): ???

Ključne besede: Praštevilski obsegi, Mersennova praštevila, hitra redukcija, Evklidov algoritem, binarni algoritem, Lehmerjev algoritem

Key words: Prime fields, Mersenne primes, fast reduction, Euclid's algorithm, binary algorithm, Lehmer's algorithm

Uvod

V današnjem svetu, kjer je varovanje različnih informacij vedno bolj pomembno, je uporaba kriptografskih sredstev že nujnost. Programi in pripomočki za šifriranje niso več dostopni samo vladam oziroma podjetjem. Vsi si lahko uredijo vse potrebno za varno komunikacijo. Zaradi vse večje uporabe programov za šifriranje je bilo veliko storjeno, da bi le ta potekala čim hitreje, tako na hardware-skem področju, kot na software-skem. In s tem področjem se bo ukvarjala tudi ta diplomska naloga.

Osredotočili se bomo na učinkovito implementacijo aritmetike v praštevilskih obsegih, ki je temelj mnogih kriptosistemov (RSA, ECC). Čeprav bo veliko algoritmov in ugotovitev veljalo za vse kriptosisteme, ki temeljijo na praštevilskih obsegih, bo poudarek na učinkoviti implementaciji kriptosistemov z eliptičnimi krivuljami. Še posebej s tistimi eliptičnimi krivuljami priporočenimi s strani NIST (National Institute of Standards and Technology). Te krivulje oz. praštevilski obsegi nad katerimi so definirane, so izbrane zaradi posebne oblike praštevil (Mersennova praštevila), ki omogočajo hitro deljenje.

Časovno učinkovita implementacija je zelo pomembna v časovno kritičnih aplikacijah (npr. komunikacija preko interneta, ...), zato se že zelo majhne pospešitve računsko zahtevnih operacij kot so množenje, modularna redukcija in inverz še kako poznajo. Primerjali bomo nekaj algoritmov za množenje in si podrobneje ogledali algoritme za izračun inverza v praštevilskih obsegih. Za modularno redukcijo pa bomo le omenili nekaj algoritmov in se bomo bolj posvetili algoritmom za hitro redukcijo, ki jo omogočajo obsegi nad Mersennovimi oz. njim podobnimi praštevili.

Diploma je razdeljena na naslednji način. V prvem poglavju bomo definirali praštevilske obsege, ter navedli definicije in izreke, ki so potrebni za razumevanje algoritmov. Za primerjavo algoritmov, bo potrebno definirati časovno in prostorsko zahtevnost. V drugem poglavju bomo govorili o aritmetiki z velikimi števili in o aritmetiki v praštevilskih obsegih. Navedli bom algoritme za seštevanje, odštevanje, množenje in modularno redukcijo. Tretje poglavje bo predstavilo osrednji del te naloge in sicer, računanje inverza v praštevilskih obsegih ter povezavo med inverzom in največjim skupnim deliteljem. Navedli bomo tri algoritme (Evklidov, Lehmerjev in Binarni), njihove osnovne ter razširjene različice in še dodatne izboljšave, ki kar občutno prispevajo k sami izboljšavi hitrosti algoritmov. Na koncu poglavja bomo primerjali vse navedene algoritme. V četrtem poglavju bom predstavil program, ki sem ga naredil za diplomsko nalogo. Omenili bomo, težave na

katere naletimo pri implementaciji določenih algoritmov. Predstavili bomo tudi rezultate testiranja učinkovitosti algoritmov.

Poglavje 1

Osnove

V tem poglavju bomo postavili temelje in definirali vse potrebno za boljše razumevanje nadaljnjih poglavij. Predstavili bomo enostavnejše algebraične strukture in preko njih prišli do definicije praštevilskih obsegov. Le-ti so temeljna struktura za algoritme, ki jih bomo predstavili in primerjali v tej diplomi.

Praštevilske obsege najdemo v številnih računalniških sistemih in kriptosistemih. V nadaljevanju poglavja bomo predstavili računalniško aritmetiko in definirali pojem algoritma ter navedli nekaj načinov za preverjanje njihove časovne in prostorske učinkovitosti. Nato bomo govorili o odvisnosti algoritmov in njihove učinkovitosti od 'okolja', v katerem delujejo (ali so to 32 ali 64 bitni procesorji, je en sam ali več procesorjev itn.).

Cilj diplome je predstavitev in primerjava osnovnih računskih operacij v praštevilskih obsegih, ki so potrebne za učinkovito implementacijo eliptičnih krivulj. Tako bomo na koncu poglavja definirali eliptične krivulje in navedli nekaj njihovih lastnosti.

1.1 Praštevilski obseg

Preden začnemo govoriti o praštevilskih obsegih, bomo definirali pojem algebraične strukture in binarne operacije. Več o tej temi si lahko pogledate v knjigi I. Vidava Algebra [16].

Definicija 1.1. Binarna operacija *na množici M je takšno pravilo, ki vsakemu paru (a, b), $a, b \in M$, priredi natanko en element $a \circ b \in M$. Takemu elementu pravimo* **kompozitum** *a in b.*

Binarne operacije delimo na notranje in zunanje.

Definicija 1.2. *Preslikava kartezičnega produkta $M \times M$ v množico M določa neko binarno računsko operacijo v množici M. Vsaka taka operacija se imenuje* **notranja operacija** *množice M.*

Definicija 1.3. *Binarni operaciji, ki vsakemu paru* $(\alpha, a) \in R \times M$*, priredi natanko en element* $\alpha a \in M$*, pravimo* **zunanja operacija**.

Sedaj, ko smo definirali binarno operacijo, tako notranjo kot zunanjo, lahko definiramo tudi pojem algebraične strukture.

Definicija 1.4. *Če je v množici definirana vsaj ena notranja ali zunanja operacija, pravimo, da ima množica* **algebraično strukturo**. *Množica* M *z dano notranjo binarno operacijo* $(M, \circ)$ *se imenuje* **grupoid**.

Grupoid je **asociativen**, če velja

$$(a \circ b) \circ c = a \circ (b \circ c)$$

za vse elemente $a, b, c \in M$ in **komutativen**, če pri poljubnih $a, b \in M$ velja

$$a \circ b = b \circ a.$$

V splošnem grupoidi niso niti asociativni niti komutativni.
Element e v grupoidu A, za katerega velja

$$a \circ e = e \circ a = a$$

za vsak $a \in M$, imenujemo **nevtralni element** ali **enota** grupoida A. Če tak element obstaja, je en sam. V algebri imajo posebno vlogo grupoidi, v katerih velja asociativnost in zato zaslužijo posebno ime.

Definicija 1.5. *Asociativni grupoid se imenuje* **polgrupa**.

Naj ima polgrupa enoto e in naj bo $a \in G$. Če v G obstaja tak element a', da velja

$$a \circ a' = e,$$

potem imenujemo a' **desni inverzni element** elementa a. Analogno vsak element a'', ki ustreza enačbi

$$a'' \circ a = e,$$

imenujemo **levi inverzni element**.

V polgrupi se lahko zgodi, da nek element nima ne levega ne desnega inverznega elementa, ima samo leve ali samo desne, ali pa ima oba inverzna elementa. Za zadnjo možnost velja naslednji izrek.

Izrek 1.6. *Če ima v polgrupi* G *z enoto* e *kak element* a *desni inverzni element* a' *in levi inverzni element* a''*, sta inverzna elementa enaka:* $a' = a''$.

Dokaz. Izrek hitro dokažemo z uporabo asociativnostnega zakona. Poglejmo si produkt $a'' \circ a \circ a'$ in ga izračunajmo na dva načina. Velja

$$a'' \circ a \circ a' = (a'' \circ a) \circ a' = e \circ a' = a',$$

$$a'' \circ a \circ a' = a'' \circ (a \circ a') = a'' \circ e = a''.$$

Ker v polgrupi velja asociativnost in ker je e enota, smo pokazali, da če obstajata levi in desni inverzni element nekega elementa, potem sta enaka. ■

Če ima nek element a levi in desni inverzni element, bomo rekli, da je obrnljiv. Skupni levi in desni inverzni element bomo označili z a^{-1}. Potem velja

$$a \circ a^{-1} = a^{-1} \circ a = e.$$

Iz te enačbe sledi, da ima tudi a^{-1} inverzni element (in sicer a), pri čemer pišemo $(a^{-1})^{-1} = a$.

Grupa je taka polgrupa z enoto, v kateri je vsak element obrnljiv. Komutativnost ne velja v vsaki grupi. Grupe s to lastnostjo **Abelove grupe**.

V komutativni grupi ponavadi uporabimo aditivni zapis. To pomeni, da $a \circ b$ ne pišemo kot produkt ab ampak kot vsoto $a + b$. Enoto zapišemo kot 0 in inverzni element elementa a z $-a$.

Grupa ima lahko končno ali neskončno število elementov. V prvem primeru jo imenujemo **končna grupa** in število njenih elementov **moč grupe**. **Red elementa** a definiramo kot najmanjše celo število, za katerega velja $a^n = e$.

Grupa je **ciklična**, če lahko vse njene elemente zapišemo kot potenco enega izmed njenih elementov. Tak element imenujemo **generator** grupe. Vsaka ciklična grupa je komutativna.

Izrek 1.7. *Grupe s praštevilsko močjo so ciklične.*

Dokaz. Naj bo moč grupe G praštevilo p. Vzemimo $a \in G$, $a \neq e$. Red tega elementa je delitelj moči grupe G, ki je v našem primeru p. Toda praštevilo ima le delitelje 1 in p. Red elementa ni enak 1, ker je a različen od enote. Zato je red enak p. Potence $a^0 = e, a^1, \ldots, a^{p-1}$ so med seboj različne. Ker jih je p, so med njimi vsi elementi grupe G. Sledi, da je grupa G ciklična in s tem tudi komutativna. ■

Množice imajo lahko več binarnih operacij. Med bolj pomembnimi algebraičnimi strukturami z dvema binarnima operacijama spadajo **kolobarji**.

Definicija 1.8. *Kolobar je množica K z dvema binarnima operacijama. Prvo operacijo imenujemo* **seštevanje***, kompozitum dveh elementov $a, b \in K$ pa* **vsoto** *$a + b$. Drugo binarno operacijo imenujemo* **množenje***, kompozitumu rečemo tudi* **produkt** *in ga zaznamujemo z ab. Pri tem morajo biti izpolnjeni naslednji pogoji:*

(i) Za seštevanje je K komutativna grupa.

(ii) Za množenje je K polgrupa.

(iii) Obe operaciji vežeta distributivnostna zakona:

$$(a+b)\cdot c = ac + bc,$$

$$c\cdot(a+b) = ca + cb.$$

Enoto za seštevanje označimo z 0. Za množenje pa ni nujno, da obstaja enota. Če pa obstaja, jo imenujemo **identiteta** in jo označimo z 1. Če identiteta obstaja je ena sama. V kolobarju z identiteto lahko govorimo o inverznih elementih za množenje, kjer velja enako kakor pri grupah, da če obstajata levi in desni inverz, sta enaka in ju označimo z a^{-1}. Element 1 je sam sebi inverz, medtem ko 0 nima inverza. Elementom, ki imajo inverz za množenje, pravimo, da so obrnljivi.
Tako smo prišli do naslednje algebraične strukture, ki se imenuje **obseg**.

Definicija 1.9. *Obseg je tak kolobar z enotama za seštevanje (0) in množenje (1), v kateremu je vsak od nič različen element obrnljiv.*

Najmanjši obseg je sestavljen le iz obeh enot ($\mathcal{O} = \{0, 1\}$). Pri obravnavanju obsegov je pomemben podatek tudi njihova **karakteristika**. Definirana ja na naslednji način:

Definicija 1.10. *Če za obseg $\mathcal{O}$ obstaja tako pozitivno število n, da za vsak element $a \in \mathcal{O}$ velja $n \cdot a = 0$, potem jr najmanjši tak n karakteristika obsega $\mathcal{O}$. Pišemo char $\mathcal{O} = n$. Če tak n ne obstaja, potem je char $\mathcal{O} = 0$.*

Od nič različna karakteristika obsega je vedno praštevilo, glej Vidav [16, str. 107].
Poglejmo si sedaj **kongruenčno relacijo**. To je relacija, ki je zelo pomembna za kriptografijo, saj je v tesni povezavi s praštevilskimi obsegi.

Definicija 1.11. *Naj bosta a in b pozitivni celi števili. Pravimo, da je a* **kongruentno** *b po modulu n, če n deli razliko $a - b$ in pišemo $a \equiv b \pmod{n}$. Številu n pravimo* **modul** *kongruence.*

Primer 1.12. $31 \equiv 4 \pmod{9}$*, ker* 9 *deli razliko* $31 - 4 = 27$.

Omenimo nekaj lastnosti, ki veljajo za kongruenčno relacijo.

Trditev 1.13. *Za vse $a, a_1, b, b_1, c \in \mathbb{Z}$ velja:*

(i) $a \equiv b \pmod{n}$ natanko takrat, ko imata a in b enak ostanek pri deljenju z n.

(ii) (refleksivnost) $a \equiv a \pmod{n}$.

(iii) (simetrija) Če je $a \equiv b \pmod{n}$, *potem je* $b \equiv a \pmod{n}$.

(iv) (tranzitivnost) Če je $a \equiv b \pmod{n}$ *in* $b \equiv c \pmod{n}$, *potem je* $a \equiv c \pmod{n}$.

(v) Če je $a \equiv b \pmod{n}$ *in* $a_1 \equiv b_1 \pmod{n}$, *potem velja*

$$a + a_1 \equiv b + b_1 \pmod{n},$$

$$a \cdot a_1 \equiv b \cdot b_1 \pmod{n}.$$

Lastnosti (ii) do (iv) nam povedo, da je $\equiv$ ekvivalenčna relacija v $\mathbb{Z}$. V ekvivalenčnem razredu števila a so vsa števila, ki so kongruentna številu a po modulu n. Očitno je, da imamo pri fiksnem številu n, n ekvivalenčnih razredov, ki jih lahko predstavimo s števili med 0 in $n-1$. To so ravno vsi možni ostanki pri deljenju z n.

Definicija 1.14. *Množica ekvivalenčnih razredov za relacijo 'biti kongruenten po modulu n' je ravno množica ostankov pri deljenju z n, tj.* $\{0, 1, \ldots, n-1\}$. *To množico označimo z* $\mathbb{Z}_n$. *Operacije seštevanje, odštevanje in množenje v obsegu* $\mathbb{Z}_n$ *gledamo kot operacije po modulu n.*

Primer 1.15. *V obsegu* $(\mathbb{Z}_{19}, +_{19}, *_{19})$ *je vsota števil* 12 *in* 10 *enaka* 3, *ker je* $12 + 10 \equiv 3 \pmod{19}$. *Produkt števil* 12 *in* 10 *je enak* 6, *ker je* $12 \cdot 10 \equiv 6 \pmod{19}$.

Definicija 1.16. *Naj bo* $a \in \mathbb{Z}_n$. *Inverzni element števila a za množenje po modulu n, je število* $x \in \mathbb{Z}_n$, *za katerega velja* $a * x \equiv 1 \pmod{n}$. *Če tak x obstaja, je enoličen in pravimo, da je a obrnljiv in označimo* $x = a^{-1}$. *Deljenje v obsegu je definirano kot množenje z inverznim elementom.*

Izrek 1.17. *Če je p praštevilo, je* $(\mathbb{Z}_p, +_p, *_p)$ *obseg.*

Dokaz. Pokazali bomo, da je $\mathbb{Z}_p$ kolobar za modularno seštevanje in modularno množenje, ter da ima vsak od nič različen element iz $\mathbb{Z}_p$ inverz za množenje. S tem bomo pokazali, da je $\mathbb{Z}_p$ obseg.

Hitro pokažemo, da je $\mathbb{Z}_p$ grupa za modularno seštevanje. Zaprtost sledi iz definicije modularnega seštevanja. Enota za seštevanje je 0. Inverz za seštevanje elementa $a \in \mathbb{Z}_p$ je $p - a$, saj velja $a + (p - a) = p \equiv 0 \pmod{p}$. Asociativnost in komutativnost modularnega seštevanja sledita iz navadnega seštevanja. Modularno množenje je asociativno, distributivnostni zakoni pa tudi očitno veljajo.

Po Fermatovem izreku (glej [15, str. 122]) za poljuben element $a \in \mathbb{Z}_p$ velja $a^p \equiv a \pmod{p}$. Iz tega sledi

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

Tako smo pokazali, da ima poljuben od nič različen element iz obsega $\mathbb{Z}_p$ inverzen element. S tem smo tudi pokazali, da je $(\mathbb{Z}_p, +_p, \cdot_p)$ obseg. ■

Naslednji izrek nam pove, kako so obsegi s praštevilsko močjo med seboj povezani. Navedli ga bomo brez dokaza.

Izrek 1.18. *Vsak obseg praštevilske moči p je izomorfen $\mathbb{Z}_p$.*

1.2 Algoritmi

Namen te diplome je učinkovita implementacija aritmetike v praštevilskih obsegih. Da pa lahko govorimo o učinkovitosti, moramo najprej definirati, kaj algoritmi sploh so in kako se ocenjuje njihova zahtevnost.

Definicija 1.19. Algoritem *je končno zaporedje ukazov, ki se pri vseh možnih naborih vhodnih podatkov konča. Algoritem reši problem in vrne nek rezultat.*

Ko govorimo o zahtevnosti algoritmov, imamo največkrat v mislih časovno in prostorsko zahtevnost. Časovna zahtevnost pomeni, koliko časa algoritem potrebuje, da se izvede. Na to lahko gledamo tudi kot, koliko osnovnih računskih operacij algoritem potrebuje za izvedbo, saj lahko pri znani hitrosti računalnika (št. oper./sec) izračunamo potrebni čas (št. oper./hitrost proc.). Ker so lahko algoritmi različno hitri pri različnih podatkih, uvedemo še dva pojma: **pričakovani delovni čas** algoritma, ki je povprečje delovnih časov algoritma pri vseh podatkih, in **najslabši delovni čas**, ki nam pove, koliko časa je največ potrebno, da se izvede algoritem. Dostikrat je zelo težko oceniti tako pričakovani kot najslabši delovni čas oz. število osnovnih operacij potrebnih za izvedbo algoritma. Zato vpeljemo nov pojem asimptotskega delovnega časa, ki nam pove obnašanje algoritma, ko se poveča velikost vhodnih podatkov.

Vpeljimo simbola O in o.

Definicija 1.20. *Naj bosta f in g funkciji. Pišemo $f(n) = O(g(n))$, če obstajata pozitivna konstanta c in pozitivno število n_0, da velja $0 \leq f(n) \leq cg(n)$ za vse $n \geq n_0$.*

Definicija 1.21. *Naj bosta f in g funkciji. Pišemo $f(n) = o(g(n))$, če za vsako pozitivno konstanto $c > 0$ obstaja konstanta $n_0 > 0$, da velja $0 \leq f(n) < cg(n)$ za vse $n \geq n_0$ oz. povedano drugače $\lim_{n\to\infty} f(n)/g(n) = 0$.*

Simbol O torej pomeni, da $f(n)$ za dovolj velike n ne raste hitreje kot $g(n)$ do multiplikativne konstante natančno. Medtem, ko nam simbol o pove, da je $g(n)$ zgornja meja za $f(n)$.

Algoritmi so vezani na računalniško okolje, v katerem se izvajajo. Namreč algoritmi niso nujno enako učinkoviti, če delujejo na 16-bitnih procesorjih ali na 64-bitnih. Veliko je odvisno tudi od arhitekture procesorjev in njihovega nabora ukazov, saj bo npr. množenje

hitreje delovalo na procesorjih z že vgrajenim ukazom za množenje dveh 32-bitnih števil kot na tistih, ki takega ukaza ne podpirajo. Prav tako se lahko pojavijo razlike, če jih izvajamo na osebnih računalnikih, superračunalnikih ali na pametnih karticah. Problem lahko predstavlja tudi količina spomina, ki ga je v osebnih računalnikih dovolj, na pametnih karticah pa ne nujno. Zato bo včasih potrebno za pametne kartice izbrati take algoritme, ki ne bodo najhitrejši, bodo pa najbolj varčni s prostorom. Število procesorjev je zelo vpliven faktor na prenosljivost algoritmov med različnimi okolji. Nekateri algoritmi se dajo bolje paralelizirati kot drugi in s tem bolje izkoristijo dodatne procesorje. V tej nalogi bomo implementirali algoritme za Pentium II 300 Mhz procesor z 32-bitno arhitekturo.

1.3 Eliptične krivulje

Eliptične krivulje v zadnjih desetletjih doživljajo pravi preporod, kar se tiče njihovega preučevanje in praktične uporabe. Pred zadnjim obdobjem jih je preučevala predvsem teoretična matematika. V zadnjem obdobju pa se uveljavljajo v teoriji števil in kriptografiji npr. za preverjanje praštevilskosti, za faktorizacijo velikih števil in v kriptografiji z javnimi ključi.
Eliptične krivulje lahko v splošnem definiramo nad poljubnim končnim obsegom. Izbor le tega je odvisen od okolja uporabe eliptičnih krivulj in naših potreb (hitrejša aritmetika, manjša poraba prostora, ...). Namen te naloge je preverjanje učinkovitosti implementacije nad praštevilskim obsegom $\mathbb{Z}_p$, tako da bomo implementirali eliptične krivulje nad praštevilskimi obsegi.

Definicija 1.22. *Eliptična krivulja E nad $\mathbb{Z}_p$ je definirana z enačbo*

$$y^2 = x^3 + ax + b,$$

kjer sta $a, b \in \mathbb{Z}_p$ in velja $4a^3 + 27b^2 \neq 0 \pmod p$, skupaj z točko $\mathcal{O}$, točko neskončno. Množica $E(\mathbb{Z}_p)$ je sestavljena iz vseh točk (x, y), $x, y \in \mathbb{Z}_p$, ki zadoščajo zgornji enačbi, skupaj z točko $\mathcal{O}$.

$$E(\mathbb{Z}_p) = \{(x, y) |\ x, y \in \mathbb{Z}_p;\ y^2 = x^3 + ax + b, 4a^3 + 27b^2 \neq 0 \pmod p\} \cup \mathcal{O}.$$

Primer 1.23. *Poglejmo si sedaj primer eliptične krivulje za $p = 13$, $y^2 = x^3 + 3x + 1$. Velja $4 \cdot 3^3 + 27 \cdot 1^2 = 108 + 27 \equiv 5 \pmod{13}$. Eliptična krivulja vsebuje naslednje točke:*

$$(0, 1), (0, 12), (4, 5), (4, 8), (6, 1), (6, 12), (7, 1), (7, 12)$$

$$(8, 2), (8, 11), (9, 9), (10, 2), (10, 11), (11, 0), (12, 6), (12, 7).$$

Iz eliptične krivulje E lahko naredimo algebraično strukturo s primerno definicijo binarne operacije nad njenimi točkami. Operacija, ki jo iz zgodovinskih razlogov pišemo aditivno, je definirana na naslednji način (vse aritmetične operacije so mišljene po modulu p). Naj bosta $P = (x_1, y_1)$ in $Q = (x_2, y_2)$ točki na E. Če je $x_1 = x_2$ in $y_1 = y_2$, potem velja $P + Q = \mathcal{O}$, sicer pa velja $P + Q = (x_3, y_3)$, kjer je

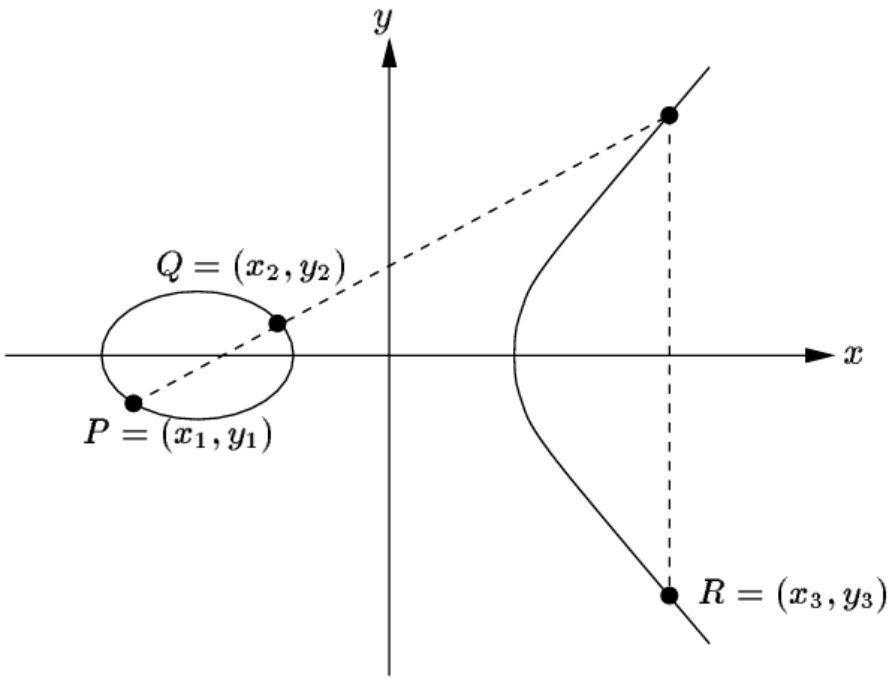
$$x_3 = \lambda^2 - x_1 - y_1,$$

$$y_3 = \lambda(x_1 - x_3) - y_1$$

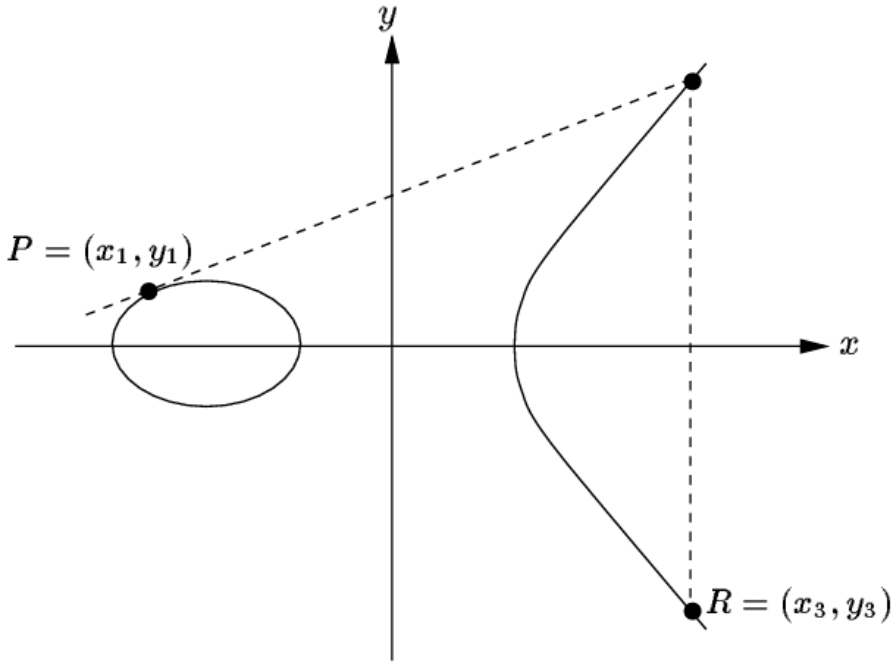
in

$$\lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & , \text{če } P \neq Q \\ \dfrac{3x_1^2 + a}{2y_1} & , \text{če } P = Q \end{cases}.$$

Na koncu definiramo še $P + \mathcal{O} = \mathcal{O} + P = P$. S tako definicijo binarne operacije seštevanja enostavno pokažemo, da je E Abelova grupa z enoto $\mathcal{O}$. Inverzni element elementa $P = (x, y)$ označimo z $-P = -(x, y)$ in velja $-P = (x, -y)$ za vsak par $(x, y) \in E(\mathbb{Z}_p)$.
Seštevanje $P + Q$ in podvojevanje $2 \cdot P$ lahko predstavimo tudi grafično (sliki 1.1 in 1.2).



Slika 1.1: Grafičen prikaz seštevanja dveh različnih točk na el. kriv. : $R = P + Q$.



Slika 1.2: Grafičen prikaz podvojevanja točke na el. kriv. : $R = 2 \cdot P$.

Primer 1.24. *Vzemimo eliptično krivuljo iz primera* 1.23 *in seštejmo točki* $P = (4, 8)$ *in* $Q = (12, 7)$. *Ker je* $P \neq Q$, *velja* $\lambda = \frac{7-8}{12-4} = \frac{1}{-8} = \frac{1}{5} = 8$. *V zadnjem koraku smo upoštevali, da je* $5^{-1} \equiv 8 \pmod{13}$. *Po krajšem izračunu dobimo* $x_3 = 0$ *in* $y_3 = 12$. *Opazimo, da je točka* $R = (0, 12)$ *res na naši eliptični krivulji.*
Pri izračunu $R = 2 \cdot P$ *dobimo najprej* $\lambda = \frac{3 \cdot 4^2 + 3}{2 \cdot 8} = \frac{51}{16} = \frac{12}{3} = 4$ *in nato še* $x_3 = 4$ *in* $y_3 = 5$. *Dobimo torej* $R = (0, 12)$.

Poglavje 2

Aritmetika v praštevilskih obsegih

Kot smo omenili že v uvodu, so praštevilski obsegi oz. modularna aritmetika sestavni del računalniških algeber, različnih kriptosistemov in veliko matematičnih aplikacij. Učinkovita implementacija operacij, kot sta npr. množenje in inverz, ki se nahajajo v temeljih veliko različnih sistemov, je zelo pomembna. Ravno zaradi razširjenosti in s tem tudi pomembnosti aritmetike v praštevilskih obsegih, je to področje dobro preučeno. Obstaja veliko algoritmov in nekaj najhitrejših izmed njih bomo tudi predstavili.

Ker je učinkovitost določene implementacije odvisna od okolja v katerem bo uporabljena, bomo v nadaljevanju predpostavili, da delujemo na Pentium II 300 MHz Intelovem procesorju. Testni računalnik ima zadostno količino spomina in prostora, tako da bosta hitrost in arhitektura procesorja edina oz. najbolj omejujoča dejavnika.

Operacije, kot so seštevanje, odštevanje in množenje z majhnimi števili tj. števili manjšimi od računalniške besede (v našem primeru 2^{32} (32-bitni procesor)), so del procesorjevih ukazov in so zelo hitre. Vendar se v praksi uporabljajo tudi večja števila oz. zelo velika števila.

V nadaljevanju poglavja bomo govorili o predstavitvi velikih števil in bomo navedli nekaj algoritmov za računanje z njimi. Predstavili bomo po en algoritem za seštevanje, odštevanje in deljenje. Mmalo bolj se bomo posvetili množenju in predstavili tri algoritme za množenje velikih števil. Ker je kvadriranje le poseben primer množenja, bomo v ta namen priredili le dva algoritma za množenje. Nato bomo navedli algoritme za modularno redukcijo. Poglavje bomo zaključili z modularno aritmetiko. Navedli bomo, kako preko algoritmov za delo s naravnimi števili dobimo algoritme za modularno aritmetiko oz. za računanje v praštevilskih obsegih.

2.1 Predstavitev velikih števil

V računalniku z 32-bitno arhitekturo so cela števila shranjena v registrih, ki imajo 32 bitov in lahko predstavijo nepredznačena števila med 0 in $2^{32} - 1$ ter predznačena števila

med -2^{31} in $2^{31}-1$. Biti besede B so oštevilčeni od 0 do 31, kjer ima skrajni desni bit indeks 0,

$$B = (b_{31}b_{30}\ldots b_2b_1b_0)_2, \text{ kjer so } b_i \in \{0,1\}.$$

Število B lahko zapišemo tudi drugače $B = \sum_{i=0}^{31} b_i 2^i$. Velika števila, tj. števila večja od računalniške besede, bomo predstavili s tabelo besed B_i,

$$\texttt{BigNum} = (B_nB_{n-1}\ldots B_2B_1B_0)_{2^{32}} \text{ oz. } \texttt{BigNum} = \sum_{i=0}^{n} B_i 2^{32i}.$$

Če je $n = 0$, potem imamo opravka z majhnimi oz. enostavnimi števili, če je $n \geq 1$ pa z velikimi števili dolžine $n+1$. Mi bomo besede predstavili z nepredznačenimi števili.

Za obseg $\mathbb{Z}_p$ oz. za praštevilo p definiramo dolžino binarne reprezentacije z $m := \lceil \log_2 p \rceil$ in število besed potrebnih za predstavitev števila p s $t := \lceil m/32 \rceil$. Za praštevila, p_{192}, p_{224}, p_{256}, p_{384}, p_{521}, priporočena s strani NIST, tako dobimo, da elemente pripadajočih obsegov predstavimo z velikimi števili dolžine $t = 6, 7, 8, 12, 17$.

2.2 Seštevanje in odštevanje

Obe operaciji sta osnovni operaciji. Izvajali ju bomo na številih z enako dolžino. Če bosta števili imeli različno dolžino, bomo manjše število z leve dopolnili z ničlami. Algoritmi, katere bomo navedli, so primerni za pozitivna cela (naravna) števila. Majhni popravki so potrebni, da algoritme posplošimo tudi na negativna števila.

Algoritem 1. Seštevanje

PODATKI : $a, b \in \mathbb{N}$, $a := (a_n, \ldots, a_0)$, $b := (b_n, \ldots, b_0)$.
REZULTAT: $c := a + b$, $c := (c_{n+1}c_n \ldots c_1c_0)$.

1. $c_0 :=$ **Add**(a_0, b_0).
2. **for** i **from** 1 **to** n **do**
   2.1. $c_i :=$ **Add_with_carry**(a_i, b_i).
3. $c_{n+1} :=$ **carry**.
4. **return** $((c_{n+1}c_n \ldots c_1c_0))$.

Seštevanje izvajamo s seštevanjem pripadajočih besed in prištevanjem prenosnega bita (ang. 'carry') k besedam z višjim redom. Na PII procesorjih je ukaz **Add_with_carry** že del procesorjevega nabora ukazov, tako da je to zelo hitra operacija.

Poglejmo si na primeru, kako seštejemo dve veliki števili, ki ju bomo predstavili z 32-bitnimi besedami. Zaradi lažjega zapisa bomo besede predstavili v šestnajstiški (hexa) reprezentaciji namesto v desetiški oz. binarni.

Primer 2.1. Seštejmo števili
$a = 582795165784562093593023490382050562333331354512285977605 6$ in
$b = 542474013766516343537289355474039436276698442823556138515 2$.

```
      edaea785 f14ae42d 4cb37b01 bbe8055b 57b101bf c290f838
+     dd3cee00 59ec7d18 3719ba87 21d5dbc0 d5e73af8 4cf83cc0
-----------------------------------------------------------
  1 caeb9586 4b376145 83cd3588 ddbde11c 2d983cb8 0f8934f8
```

Vsota $a + b$ je enaka 4032115201804575005573413490801112605663291168872983909 04.

Odštevanje je implementirano na podoben način kot seštevanje, le da ne prenašamo prenosni bit ampak si ga sposojamo (ang. 'borrow'). Algoritem je zelo hiter, še hitrejši je, če sta operaciji **Sub** in **Sub_with_borrow** del procesorjevega nabora ukazov, kar velja v primeru PII.

V algoritmu imamo pri vhodnih podatkih pogoj $a \geq b$. Ta pogoj lahko izpustimo za ceno malo zahtevnejšega algoritma. Vendar bomo v nadaljevanju pri implementaciji modularnega odštevanja ta pogoj potrebovali, tako da ga bomo kar obdržali.

Algoritem 2. Odštevanje

PODATKI : $a, b \in \mathbb{N}$, $a := (a_n \dots a_0)$, $b := (b_n \dots b_0)$.
REZULTAT: $c := a - b$, $c = (c_n c_n \dots c_1 c_0)$.

1. $c_0 := \textbf{Sub}(a_0, b_0)$.
2. **for** i **from** 1 **to** n **do**
   2.1. $c_i := \textbf{Sub\_with\_borrow}(a_i, b_i)$.
3. **return** $((c_{n+1} c_n \dots c_1 c_0))$.

Poglejmo si na primeru kako deluje odštevanje dveh velikih števil.

Primer 2.2. Odštejmo števili
$a = 582795165784562093593023490382050562333331354512285977605 6$ in
$b = 542474013766516343537289355474039436276698442823556138515 2$.

```
      edaea785 f14ae42d 4cb37b01 bbe8055b 57b101bf c290f838
-     dd3cee00 59ec7d18 3719ba87 21d5dbc0 d5e73af8 4cf83cc0
-----------------------------------------------------------
      1071b985 975e6715 1599c07a 9a12299a 81c9c6c7 7598bb78
```

Razlika $a - b$ je 1125269179551078437130312845856089998610029797335842116120 8.

Seštevanje in odštevanje z majhnimi števili sta operaciji, ki sta vsebovani v naboru današnjih procesorjev. Kot posledica tega sta zelo hitri operaciji in se izvajata v konstantnem času ($O(1)$). Implementirana algoritma izvedeta n osnovnih operacij z majhnimi števili, zato imata seštevanje in odštevanje števil z n besedami časovno zahtevnost $O(n)$.

2.3 Množenje

Množenje dveh majhnih števil je vključeno v PII kakor tudi v vse novejše procesorje in se posledično izvaja v konstantnem času ($O(1)$). Pri velikih številih, za razliko od seštevanja in odštevanja, pa je lahko časovna zahtevnost (npr. pri klasičnem množenju števil z n in m besedami) kvadratna ($O(nm)$). Zaradi višje zahtevnosti je večja potreba po učinkoviti implementaciji.

Ogledali si bomo dve implementaciji klasičnega algoritma, ter implementacijo Karatsubine metode za množenje. Obstajajo sicer algoritmi, ki so asimptotično hitrejši npr. Hitra Fourierjeva transformacija, vendar ti algoritmi niso hitrejša za števila, ki se danes uporabljajo v kriptografiji. Do izraza pridejo šele pri zelo velikih številih.

Klasično množenje nam predstavlja množenje, kjer množimo vse cifre oz. besede med seboj. Od implementacije pa bo odvisno, v kakšnem zaporedju se bodo množenja izvajala ter kako bomo hranili trenutno vsoto.

Prva implementacija bo običajno množenje, ki se ga učimo v osnovni šoli. V algoritmu nam (uv) pomeni 64-bitno združitev dveh 32-bitnih besed u in v.

Algoritem 3. Množenje 1

PODATKI : $a, b \in \mathbb{N}$, $a := (a_n \dots a_0)$, $b := (b_t \dots b_0)$.
REZULTAT: $c := a \cdot b$, $c := (c_{n+t+1} c_{n+t} \dots c_1 c_0)$.

1. **for** i **from** 0 **to** $n + t + 1$ **do** $c_i := 0$.
2. **for** i **from** 0 **to** t **do**
   - 2.1. $r := 0$.
   - 2.2. **for** j **from** 0 **to** n **do**
     - $(uv) := c_{i+j} + a_j \cdot b_i + r$.
     - $c_{i+j} := v$, $r := u$.
   - 2.3. $c_{i+n+1} := u$.
3. **return** $((c_{n+t+1} \dots c_1 c_0))$.

Na začetku algoritma postavimo vsoto c na 0. V koraku 2.2 zmnožimo j-to in i-to besedo števil a in b ter ju prištejemo pripadajoči besedi trenutne vsote. Prištejemo pa še r tj. število, ki se prenese iz prejšnjega množenja. Dobljeno vsoto razdelimo na dva dela. Spodnjih 32-bitov shranimo v trenutno vsoto. Zgornjih 32-bitov pa shranimo v r in

jih prenesemo v naslednje množenje. To naredimo za vseh $n \cdot t$ parov besed. Tako smo zmnožili vse možne pare in jih sešteli. Rezultat algoritma je število c, ki je produkt števil a in b.

Oglejmo si na primeru kako deluje Algoritem 3.

Primer 2.3. Zmnožimo števili $a = 5732$ in $b = 916$. Števili sta v desetiški reprezentaciji. Besede v našem primeru so kar cifre. S tem bo algoritem bolj nazorno prikazan. V Tabeli 2.1 so prikazani koraki Algoritma 3 za množenje števil a in b. Na koncu dobimo produkt $c = a \cdot b = 5250512$.

i	j	r	$c_{i+j} + a_j \cdot b_i + r$	u	v	c_6	c_5	c_4	c_3	c_2	c_1	c_0
0	0	0	$0 + 12 + 0$	1	2	0	0	0	0	0	0	2
	1	1	$0 + 18 + 1$	1	9	0	0	0	0	0	9	2
	2	1	$0 + 42 + 1$	4	3	0	0	0	0	3	9	2
	3	4	$0 + 30 + 4$	3	4	0	0	3	4	3	9	2
1	0	0	$9 + 2 + 0$	1	1	0	0	3	4	3	1	2
	1	1	$3 + 3 + 1$	0	7	0	0	3	4	7	1	2
	2	0	$4 + 7 + 0$	1	1	0	0	3	1	7	1	2
	3	1	$3 + 5 + 1$	0	9	0	0	9	1	7	1	2
2	0	0	$7 + 18 + 0$	2	5	0	0	9	1	5	1	2
	1	2	$1 + 27 + 2$	3	0	0	0	9	0	5	1	2
	2	3	$9 + 63 + 3$	7	5	0	0	5	0	5	1	2
	3	7	$0 + 45 + 7$	5	2	5	2	5	0	5	1	2

Tabela 2.1: Množenje števil $a = 5732$ in $b = 916$ z Algoritmom 3.

Računsko zahteven del algoritma je izračun $(c_{i+j} + a_j \cdot b_i + r)$. Pokazati moramo, da to vsoto lahko vedno predstavimo z dvema besedama. Vsako izmed števil je veliko največ $2^{32} - 1$. Skupno je torej vsota velika največ $(2^{32} - 1) + (2^{32} - 1)^2 + (2^{32} - 1) = 2^{64} - 1$ kar pomeni, da lahko vsoto shranimo v dve besedi.

Naslednja implementacija klasičnega množenja se od prejšnje razlikuje v zaporedju izvajanja množenj besed. Prejšnja implementacija je določeno besedo števila a pomnožila z vsemi besedami števila b itn. dokler nismo množili vseh besed števila a. Naslednja implementacija pa po vrsti računa besede produkta. Začnemo z skrajno desno.

Algoritem 4. Množenje 2

PODATKI : $a, b \in \mathbb{N}$, $a := (a_n \dots a_0)$, $b := (b_t \dots b_0)$.
REZULTAT: $c := a \cdot b$, $c := (c_{n+t+1} c_{n+t} \dots c_1 c_0)$.

1. $r_0 := 0$, $r_1 := 0$, $r_2 := 0$.
2. **for** k **from** 0 **to** $n + t$ **do**
   2.1. **for** $\forall\, (i, j) \in \{(i, j) |\ i + j = k,\ 0 \le i \le n,\ 0 \le j \le t\}$
   $(uv) := a_i \cdot b_j$.
   $r_0 :=$ **Add**(r_0,v), $r_1 :=$ **Add_with_carry**(r_1,u), $r_2 :=$ **Add_with_carry**$(r_2,0)$.
   2.2. $c_k := r_0$, $r_0 := r_1$, $r_2 := 0$.
3. $c_{n+t+1} := r_0$.
4. **return** $((c_{n+t+1} \dots c_1 c_0))$.

Za boljšo predstavo kako deluje Algoritem 4, si poglejmo primer množenja z njim.

Primer 2.4. Zmnožimo števili $a = 9478$ in $b = 649$. V Tabeli 2.2 imamo navedene korake algoritma. Opazimo, da je število korakov enako za oba algoritma za množenje.

k	i	j	$a_i \cdot b_j$	r_2	r_1	r_0	c_6	c_5	c_4	c_3	c_2	c_1	c_0
0	0	0	72	0	7	2							2
1	0	1	32	0	3	9							
	1	0	63	1	0	2						2	2
2	0	2	48	0	5	8							
	1	1	28	0	8	6							
	2	0	36	1	2	2					2	2	2
3	1	2	42	0	5	4							
	2	1	16	0	7	0							
	3	0	81	1	5	1				1	2	2	2
4	2	2	24	0	3	9							
	3	1	36	0	7	5			5	1	2	2	2
5	3	2	54	0	6	1		1	5	1	2	2	2
						6	6	1	5	1	2	2	2

Tabela 2.2: Množenje števil $a = 5732$ in $b = 916$ z Algoritmom 4.

Težave na katere naletimo pri implementaciji tega algoritma ležijo predvsem v vrstici 2.1, kjer moramo izračunati vse pare besed skupne dolžine k. Opazimo, da je manj dela s prenosno števko ter, da nimamo klicev trenutne vsote, kar lahko nekaj prispeva k zares učinkoviti implementaciji.

Opazimo, da imata algoritma 3 in 4 časovno zahtevnost $O(nt)$, saj v obeh algoritmih med seboj zmnožimo vseh $n \cdot t$ besed. Oba algoritma sta približno enako učinkovita. Hitrost bo odvisna od učinkovite pretvorbe v računalniški jezik. Namreč algoritem, ki ima manj numeričnih operacij in več preverjanja, inicializiranja ter potratno porabo spremenljivk, se lahko počasneje izvaja kot algoritem z več numeričimi operacijami, a manj zapleteno računalniško kodo.

Metoda Karatsube se zelo razlikuje od klasičnih množenj in temelji na naslednjem pravilu. Naj bosta $a = a_1 2^t + a_0$ in $b = b_1 2^t + b_0$ dve števili. Potem lahko množenje $a \cdot b$ izvedemo na naslednji način:

$$a \cdot b = (a_1 2^t + a_0)(b_1 2^t + b_0) = (a_1 b_1) 2^{2t} + (a_1 b_0 + a_0 b_1) 2^t + a_0 b_0.$$

Vendar pa, kot bomo pozneje pokazali, s tem ne pridobimo ničesar. Če pa vsoto $(a_1 b_0 + a_0 b_1)$ preuredimo v

$$(a_1 b_0 + a_0 b_1) = (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0,$$

imamo namesto dveh množenj in enega seštevanja eno množenje in po dve seštevanji ter odštevanji.

Poglejmo si Karatsubino metodo na primeru.

Primer 2.5. Spet bomo množili števili $a = 5732$ in $b = 916$. Števili razdelimo na zgornji in spodnji del, $a = 57 \cdot 10^2 + 32$ in $b = 9 \cdot 10^2 + 16$. Potem imamo

$$(57 \cdot 10^2 + 32) \cdot (9 \cdot 10^2 + 16) = (57 \cdot 9) 10^4 + (57 \cdot 16 + 32 \cdot 9) 10^2 + 32 \cdot 16,$$

Izračunamo naslednja množenja

$$57 \cdot 9 = 513, \quad 32 \cdot 16 = 512$$

$$(57 \cdot 16 + 32 \cdot 9) = (57 + 32)(9 + 16) - 57 \cdot 9 - 32 \cdot 16 = 89 \cdot 25 - 513 - 512 = 1200$$

in dobimo:

$$(57 \cdot 10^2 + 32) \cdot (9 \cdot 10^2 + 16) = 513 \cdot 10^4 + 1200 \cdot 10^2 + 512 = 5250512.$$

Ker je množenje dosti dražja operacija od seštevanja in odštevanja, lahko s Karatsubinim množenjem nekaj pridobimo. O tem koliko pridobimo govori naslednji izrek (glej Kozak [8, str. 167]).

Izrek 2.6. *Naj bodo a, c in r nenegativne konstante ter naj velja $c > 0$. Rešitev rekurzije*

$$T(n) = \begin{cases} T_0, & n = 1; \\ aT(\frac{n}{c}) + O(n^r), & n > 1. \end{cases}$$

je dana z

$$T(n) = \begin{cases} O(n^r), & a < c^r; \\ O(n^r \log n), & a = c^r; \\ O(n^{\log_c a}), & a > c^r. \end{cases}$$

Dokaz. Naj bo n potenca števila c, $n = c^k$. Z večkratno uporabo rekurzivne formule dobimo

$$T(n) = bn^r \sum_{i=0}^{k} q^i,$$

kjer smo z q označili $q := a/c^r$. Kadar velja $a = c^r$, je q enak 1. V tem primeru je

$$T(n) = O(bn^r k) = O(bn^r \log_c n) = O(n^r \log n).$$

Za $a < c^r$ vsota konvergira tudi, ko k raste prek vseh meja, torej je

$$T(n) \leq \frac{bn^r}{1-q},$$

in velja $T(n) = O(n^r)$. Za $q > 1$ pa nam vsota da

$$T(n) = bn^r \frac{\left(\frac{a}{c^r}\right)^{k+1} - 1}{\frac{a}{c^r} - 1} = b\frac{a^{k+1} - c^{r(k+1)}}{a - c^r} = O(a^k) = O(a^{\log_c n}) = O(n^{\log_c a}).$$

Ker je $T(n)$ nepadajoča funkcija n, trditev velja tudi za splošni n. ■

Algoritem 5. Karatsubino množenje

PODATKI : $a, b \in \mathbb{N}$, $a := (a_n \dots a_0)$, $b := (b_t \dots b_0)$, $W = 2^{32}$.
REZULTAT: $c := a \cdot b$, $c := (c_{n+t+1} c_{n+t} \dots c_1 c_0)$.

1. $r_0 := 0$, $r_1 := 0$, $r_2 := 0$.
2. **for** i **from** 0 **to** n **step** 2 **do**
   - 2.1. **for** j **from** 0 **to** t **step** 2 **do**
     - 2.1.1. $r_2 := a_{i+i} \cdot b_{j+1}$, $r_0 := a_i \cdot b_j$, $r_1 := (a_{i+i} + a_i) \cdot (b_{j+1} + b_j) - r_2 - r_0$.
     - 2.1.2. $(u_3 u_2 u_1 u_0) := r_2 \cdot W^2 + r_1 \cdot W + r_0$.
     - 2.1.3. $c_{i+j} :=$ **Add**(c_{i+j}, u_0).
     - 2.1.4. $c_{i+j+1} :=$ **Add_with_carry**(c_{i+j+1}, u_1).
     - 2.1.5. $c_{i+j+2} :=$ **Add_with_carry**(c_{i+j+2}, u_2).
     - 2.1.6. $c_{i+j+3} :=$ **Add_with_carry**(c_{i+j+3}, u_3).
     - 2.1.7. $c_{i+j+4} :=$ **Add_with_carry**$(c_{i+j+4}, 0)$.
3. **return** $((c_{n+t+1} \dots c_1 c_0))$.

V primeru Karatsubine metode je $a = 3$, $c = 2$ in $r = 1$. Če uporabimo Izrek 2.6 dobimo, da je časovna zahtevnost $O(n^{\log_2 3}) = O(n^{1.58})$. Prej smo omenili, da če množenje dveh števil preuredimo v 4 množenja pol manjših števil, ne pridobimo ničesar. To lahko pokažemo z uporabo Izreka 2.5. V tem primeru velja $a = 4$, $c = 2$ in $r = 1$ in s tem je časovna zahtevnost $O(n^{\log_2 4}) = O(n^2)$, kar je enako časovni zahtevnosti klasičnega množenja.

V Algoritmu 5 uporabimo Karatsubino idejo za množenje dveh besed števila a z dvema besedam števila b. Množenje torej izvajamo na blokih velikosti 2. Pri tem moramo opozoriti, da sta v koraku 2.1.1 besedi a_{i+1} in b_{j+1} lahko enaki nič. V koraku 2.1.2 izračunamo $(u_3u_2u_1u_0)$ tj. 128-bitni produkt dveh 64-bitnih števil. Ta produkt po besedah prištejemo pripadajočim besedam trenutne vsote.

2.4 Kvadriranje

Pri kvadriranju algoritme za množenje preuredimo tako, da upoštevajo dejstvo, da sta množenec in množitelj enaka. Navedli bomo dva algoritma, ki imata za osnovo algoritma 3 in 4. Podrobneje bomo pogledali le Algoritem 7. Opazimo, da se od Algoritma 4 razlikuje le v pogojih za korak 2.1. Tu zaradi simetrije zahtevamo, da je i manjše ali enako j. Simetrijo pri kvadriranju nadalje upoštevamo v koraku 2.1, kjer produkt (uv) množimo z 2, če sta i in j različna oz. i manjše od j. Množenje z 2 izvedemo preko premika za eno mesto v levo ($\ll 1$). Pri tem izračunu moramo paziti, saj je lahko rezultat večji od 64 bitov. Zaradi tega prenosno števko prištejemo r_2.

Algoritem 6. Kvadriranje 1

PODATKI : $a \in \mathbb{N}$, $a := (a_n \dots a_0)$.
REZULTAT: $c := a^2$, $c := (c_{2n+1}c_{2n} \dots c_1c_0)$.

1. **for** i **from** 0 **to** $2t + 1$ **do** $c_i := 0$.
2. **for** i **from** 0 **to** t **do**
   - 2.1. $(uv) := c_{2i} + a_i^2$, $c_{2i} := v$, $r = u$.
   - 2.2. **for** j **from** i **to** n **do**
     - $(uv) := c_{i+j} + 2a_j \cdot a_i + r$.
     - $c_{i+j} := v$, $r := u$.
   - 2.3. $c_{i+n+1} := u$.
3. **return** $((c_{n+t+1} \dots c_1c_0))$.

Primer 2.7. Kvadrirajmo število $a = 5732$. V Tabeli 2.3 so navedeni koraki kvadriranja. Rezultat je $a^2 = 32655624$. Opazimo, da imamo namesto $4 \times 4 = 16$ samo 10 množenj.

k	i	j	$a_i \cdot a_j$	r_2	r_1	r_0	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
0	0	0	4	0	0	4								4
1	0	1	6	0	1	2							2	4
2	0	2	14	0	2	9								
	1	1	9	0	3	8						8	2	4
3	0	3	10	0	2	3								
	1	2	21	0	6	5					5	8	2	4
4	1	3	15	0	3	6								
	2	2	49	0	8	5				5	5	8	2	4
5	2	3	35	0	7	8			8	5	5	8	2	4
6	3	3	25	0	3	2		2	8	5	5	8	2	4
						3	3	2	8	5	5	8	2	4

Tabela 2.3: Kvadrirajmo število $a = 5732$ z Algoritmom 7.

Število množenj se pri kvadriranju zaradi simetrije zmanjša za približno pol. In sicer namesto $(n+1) \cdot (n+1)$ množenj jih imamo $(n^2+n)/2$. Število množenj se ne more zmanjšati za več kot pol, saj lahko množenje dve števil izrazimo z dvemi kvadriranji preko naslednje formule:

$$x \cdot y = ((x+y)^2 - (x-y)^2)/4.$$

Vendar je izboljšava učinkovitosti za faktor 2 kar precejšnja.

Algoritem 7. Kvadriranje 2

PODATKI : $a \in \mathbb{N}$, $a := (a_n \dots a_0)$.
REZULTAT: $c := a^2$, $c := (c_{2n+1} c_{2n} \dots c_1 c_0)$.

1. $r_0 := 0$, $r_1 := 0$, $r_2 := 0$
2. **for** k **from** 0 **to** $2n$ **do**
   2.1. **for** $\forall\, (i,j) \in \{(i,\, j) |\ i + j = k,\, 0 \le i \le j < n\}$
   $(uv) := a_i \cdot a_j$.
   if $(i < j)$ **then**
   $(uv) := (uv) \ll 1$, $r_2 :=$ **Add_with_carry**$(r_2,\, 0)$.
   $r_0 :=$ **Add**(r_0, v), $r_1 :=$ **Add_with_carry** (r_1, u), $r_2 :=$ **Add_with_carry**$(r_2, 0)$.
   2.2. $c_k := r_0$, $r_0 := r_1$, $r_2 := 0$.
3. $c_{n+t+1} := r_0$.
4. **return** $((c_{n+t+1} \dots c_1 c_0))$

2.5 Deljenje

Med osnovnimi aritmetičnimi operacijami je deljenje najbolj zahtevna oz. draga operacija. V tem podpoglavju bomo opisali algoritem za deljenje večbesednih naravnih števil. Ta algoritem je del Evklidovega in Lehmerjevega algoritma, ki se uporabljata za izračun inverza v obsegu $\mathbb{Z}_p$, kar je tema naslednjega poglavja.

Algoritem 8. Deljenje

PODATKI : $a, b \in \mathbb{N}$, $a := (a_n \dots a_0)$, $b := (b_t \dots b_0)$, $n \geq t \geq 1$, $b_t \neq 0$, $W := 2^{32}$.
REZULTAT: $q := (q_{n-t}, \dots, q_0)$, $r := (r_t, \dots, r_0) : a = q \cdot y + r$, $0 \leq r < b$.

1. **for** j **from** 0 **to** $n - t$ **do** $q_j := 0$
2. **while** $(a \geq b \cdot W^{n-t})$ **do** $q_{n-t} := q_{n-t} + 1$, $a := a - b \cdot W^{n-t}$.
3. **for** i **from** n **down to** $(t + 1)$ **do**
   - 3.1. **if** $(a_i < b_t)$ **then** $q_{i-t-1} := W - 1$.
     else $q_{i-t-1} := \lfloor ((a_i \cdot W^2 + a_{i-1} \cdot W + a_{i-2})/(b_t \cdot W + b_{t-1}) \rfloor$.
   - 3.2. **while** $(q_{i-t-1}(b_t \cdot W + b_{t-1}) < a_i \cdot W^2 + a_{i-1} \cdot W + a_{i-2})$ **do**
     $q_{i-t-1} := q_{i-t-1} - 1$.
   - 3.3. $a := a - q_{i-t-1} b \cdot W^{i-t-1}$.
   - 3.4. **if** $(a < 0)$ **then** $a := a + b \cdot W^{i-t-1}$, $q_{i-t-1} := q_{i-t-1} - 1$.
4. $r := a$.
5. **return** (q, r).

Algoritem za deljenje nam vrne tako količnik kot ostanek pri deljenju. Z majhnimi spremembami nam lahko algoritem vrne le količnik oz. le ostanek.

Primer 2.8. Naj bo $a = 721948327$ in $b = 84461$, tako da sta $n = 8$ in $t = 4$. Tabela 2.4 nam prikaže korake v Algoritmu 8. Zadnja vrstica nam vrne količnik $q = 8547$ in ostanek $r = 60160$.

Pri oceni časovne zahtevnosti deljenja bomo zanemarili hitre operacije, kot so seštevanja in odštevanja ter množenja s potencami števila W, kar so ravno premiki za besedo v levo. V algoritmu $(n - t)$-krat izvedemo zanko 3. V koraku 3.1 imamo eno enostavno deljenje. Korak 3.2 prispeva dve množenji, korak 3.3 pa $(t + 1)$ množenj. Skupaj imamo torej $(t + 3)$ enostavnih množenj. Ker izvedemo $(n - t)$ korakov, je časovna zahtevnost Algoritma 8 enaka $O((n - t)t)$ oz. $O((\ln q)(\ln b))$. Nekaj dodatnih informacij o deljenju (npr. o normalizaciji) lahko dobite v diplomi R. Petka [12, str. 42].

i	q_4	q_3	q_2	q_1	q_0	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
-	0	0	0	0	0	7	2	1	9	4	8	3	2	7
8	0	9	0	0	0	7	2	1	9	4	8	3	2	7
		8	0	0	0		4	6	2	6	0	3	2	7
7		8	5	0	0			4	0	2	9	8	2	7
6		8	5	5	0			4	0	2	9	8	2	7
		8	5	4	0				6	5	1	3	8	7
5		8	5	4	8				6	5	1	3	8	7
		8	5	4	7					6	0	1	6	0

Tabela 2.4: Delimo števili $a = 721948327$ in $b = 84461$.

2.6 Modularna redukcija

Poznamo več metod za modularno redukcijo, kot so npr. klasična, Montgomeryjeva in Barretova. Nobena izmed teh metod ni najboljša v vseh primerih. Avtorji članka Comparison of three modular reduction functions [1] so ugotovili, da so algoritmi po učinkovitosti primerljivi. Klasična metoda naj bi bila najbolj primerna za enkratno redukcijo. Za modularno potenciranje pri majhnih argumentih naj bi bil najbolj primeren algoritem na osnovi Barretove metode. V splošnem pa naj bi se za najbolj učinkovito metodo izkazala Montgomeryjeva metoda.

Ker je modularna redukcija pogosta operacija v modularni aritmetiki, so pri NIST [11] za velikost praštevilskih obsegov izbrali Mersennova praštevila. Le ta, kot tudi pseudo Mersennova praštevila, omogočajo hitro modularno redukcijo. Metodo za hitro redukcijo je razvil J.A. Solinas [13]. Opomnimo naj, da v RSA kriptosistemih ne moremo izkoristiti posebne oblike Mersennovih praštevil. V kriptosistemu, ki temelji na faktorizaciji naravnega števila $n = p \cdot q$ (p in q praštevili), bi dejstvo, da sta p in/ali q Mersennovi praštevili, močno olajšalo faktorizacijo števila n.

Hitra redukcija temelji na naslednji lastnosti. Naj bo $m = 2^k - 1$ (Mersennovo število). Potem lahko $2k$-bitno število $n = 2^k T + U$ hitro reduciramo po modulu m in sicer na naslednji način:

$$m = 2^k - 1 \text{ in } 2^k = m + 1,$$

$$n = 2^k T + U = (m+1)T + U = mT + T + U \equiv T + U \pmod{m}. \tag{2.1}$$

Tako imamo namesto zahtevnega deljenja le seštevanje prvih k-bitov z zadnjimi k biti. V primeru, da je dobljeno število večje od m potem izvedemo še redukcijo po modulu m (nekaj odštevanj z modulom m). Metoda je primerna za vsa praštevila p, ki imajo posebno obliko (pseudo Mersennova praštevila)

$$p = b^k \pm c_{k-i} b^{k-1} \ldots \pm c_1 b^1 \pm c_0 b^0, \tag{2.2}$$

kjer je b velikost računalniške besede (npr. $b = 2^{32}$) in $c_i \in \{0, 1\}$. Po tej metodi naj bi pozitivno število $A < p^2$, kjer je A $2n$-besedno število, predstavili kot n-besedno število iz obsega $\mathbb{Z}_p$. Imamo torej število

$$A = \sum_{i=0}^{2n-1} A_i \cdot b^i$$

in iščemo število B

$$B = \sum_{i=0}^{n-1} B_i \cdot b^i \pmod{p},$$

za katerega bo veljalo $A \equiv B \pmod{p}$. Analogno kot pri Mersennovih številih bomo tudi tu morali potence b^i, za $i \geq n$, izraziti z linearno kombinacijo potenc b^0, b^1, ..., b^{n-1} oz. zapisano matematično

$$b^i \equiv \sum_{j=0}^{n-1} a_{ij} \cdot b^j \pmod{p}, \text{ za } i \geq n \text{ in } a_{ij} \in \mathbb{N} \cup \{0\}.$$

Tako bo za število A veljalo

$$A = \sum_{i=0}^{2n-1} A_i \cdot b^i \equiv \left(\sum_{i=0}^{n-1} A_i \cdot b^i + \sum_{i=n}^{2n-1} A_i \sum_{j=0}^{n-1} a_{ij} \cdot b^j \right) \pmod{p}. \tag{2.3}$$

Poglejmo si primer za $p = 2^{224} - 2^{96} + 1$. Vsako število A manjše od p^2 lahko zapišemo kot

$$A = \sum_{i=0}^{13} A_i b^i = \sum_{i=0}^{13} A_i 2^{32i}$$

kjer je vsak A_i 32-bitno celo število. Število A lahko zapišemo kot združitev 32-bitnih besed:

$$A = (A_{13} A_{12} \ldots A_1 A_0).$$

Oglejmo si naslednje kongruence za naš p:

$$p = b^7 - b^3 + 1 \equiv 0 \pmod{p}$$

velja

$$\begin{aligned}
b^7 &\equiv -1 + b^3 \pmod{p}, \\
b^8 &\equiv -b + b^4 \pmod{p}, \\
b^9 &\equiv -b^2 + b^5 \pmod{p}, \\
b^{10} &\equiv -b^3 + b^6 \pmod{p}, \\
b^{11} &\equiv -b^4 + b^7 \equiv -1 + b^3 - b^4 \pmod{p}, \\
b^{12} &\equiv -b^5 + b^8 \equiv -b + b^4 - b^5 \pmod{p}, \\
b^{13} &\equiv -b^6 + b^9 \equiv -b^2 + b^5 - b^6 \pmod{p}.
\end{aligned}$$

Te kongurence lahko zapišemo v matrični obliki

$$\begin{pmatrix} b^7 \\ b^8 \\ b^9 \\ b^{10} \\ b^{11} \\ b^{12} \\ b^{13} \end{pmatrix} \equiv \begin{pmatrix} -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ b \\ b^2 \\ b^3 \\ b^4 \\ b^5 \\ b^6 \end{pmatrix} \pmod{p}.$$

Potem velja

$$\sum_{i=0}^{13} A_i b^i = (A_0, A_1, \ldots, A_6) \begin{pmatrix} 1 \\ b \\ b^2 \\ b^3 \\ b^4 \\ b^5 \\ b^6 \end{pmatrix} + (A_7, A_8, \ldots, A_{13}) \begin{pmatrix} b^7 \\ b^8 \\ b^9 \\ b^{10} \\ b^{11} \\ b^{12} \\ b^{13} \end{pmatrix}$$

$$\equiv \left((A_0, A_1, \ldots, A_6) + (A_7, A_8, \ldots, A_{13}) \begin{pmatrix} -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & -1 \end{pmatrix} \right) \begin{pmatrix} 1 \\ b \\ b^2 \\ b^3 \\ b^4 \\ b^5 \\ b^6 \end{pmatrix}.$$

Iz teh enačb dobimo enačbe za B_i zapisane v matrični obliki:

$$(B_0, B_1, \ldots, B_6) = (A_0, A_1, \ldots, A_6) + (A_7, A_8, \ldots, A_{13}) \begin{pmatrix} -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & -1 \end{pmatrix}.$$

Sedaj pa še eksplicitno napišimo B_i-je.

$$\begin{aligned} B_0 &= A_0 && -A_7 - A_{11}, \\ B_1 &= A_1 && -A_8 - A_{12}, \\ B_2 &= A_2 && -A_9 - A_{13}, \\ B_3 &= A_3 + A_7 + A_{11} && -A_{10}, \\ B_4 &= A_4 + A_8 + A_{12} && -A_{11}, \\ B_5 &= A_5 + A_9 + A_{13} && -A_{12}, \\ B_6 &= A_6 + A_{10} && -A_{13}. \end{aligned}$$

Da ne bi računali števila B_i posebej in jih potem še množili z b^i, jih raje preuredimo in iz njih sestavimo večbesedna števila. Iz zapisa števil B_i je očitno kako naj taka števila sestavimo, da jih bo najmanj. Če števila sestavimo po stolpcih dobimo:

$$\begin{aligned} S_1 &= (A_6, A_5, A_4, A_3, A_2, A_1, A_0), \\ S_2 &= (A_{10}, A_9, A_8, A_7, 0, 0, 0), \\ S_3 &= (0, A_{13}, A_{12}, A_{11}, 0, 0, 0), \\ S_4 &= (A_{13}, A_{12}, A_{11}, A_{10}, A_9, A_8, A_7), \\ S_5 &= (0, 0, 0, 0, A_{13}, A_{12}, A_{11}). \end{aligned}$$

Tako za praštevilo p dobimo formulo za hitro redukcijo:

$$A \equiv B \equiv S_1 + S_2 + S_3 - S_4 - S_5 \pmod{p}.$$

Iz primera je razvidno, kako uporabimo formulo (2.3) za izračun modularne redukcije števila A po modulu p. Na isti način izpeljemo formulo za modularno redukcijo za poljuben p oblike (2.2).

Algoritem 9. Hitra redukcija po modulu $p_{192} = 2^{192} - 2^{64} - 1$

PODATKI : $a := (a_5, a_4, a_3, a_2, a_1, a_0)$, a_i so 64-bitna besede, $0 \leq a < p_{192}^2$.
REZULTAT: $c := a \pmod{p_{192}}$.

1. **define** 192-bit integers: $S_1 := (a_2, a_1, a_0)$, $S_2 := (0, a_3, a_3)$,
   $S_3 := (a_4, a_4, 0)$, $S_4 := (a_5, a_5, a_5)$.
2. **return** $((S_1 + S_2 + S_3 + S_4) \pmod{p_{192}})$.

Algoritem 10. Hitra redukcija po modulu $p_{224} = 2^{224} - 2^{96} + 1$

PODATKI : $a := (a_{13}, \ldots, a_2, a_1, a_0)$, a_i so 32-bitna besede, $0 \leq a < p_{224}^2$.
REZULTAT: $c := a \pmod{p_{224}}$.

1. **define** 224-bit integers: $S_1 := (a_6, a_5, a_4, a_3, a_2, a_1, a_0)$,
   $S_2 := (a_{10}, a_9, a_8, a_7, 0, 0, 0)$, $S_3 := (0, a_{13}, a_{12}, a_{11}, 0, 0, 0)$,
   $S_4 := (a_{13}, a_{12}, a_{11}, a_{10}, a_9, a_8, a_7)$, $S_5 := (0, 0, 0, 0, a_{13}, a_{12}, a_{11})$.
2. **return** $((S_1 + S_2 + S_3 - S_4 - S_5) \pmod{p_{224}})$.

Algoritem 11. Hitra redukcija po modulu $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} + 1$

PODATKI : $a := (a_{15}, \ldots, a_2, a_1, a_0)$, a_i so 32-bitna besede, $0 \leq a < p_{256}^2$.
REZULTAT: $c := a \pmod{p_{256}}$.

1. **define** 256-bit integers: $S_1 := (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$,
   $S_2 := (a_{15}, a_{14}, a_{13}, a_{12}, a_{11}, 0, 0, 0)$, $S_3 := (0, , a_{15}, a_{14}, a_{13}, a_{12}, 0, 0, 0)$,
   $S_4 := (a_{15}, a_{14}, 0, 0, 0, a_{10}, a_9, a_8)$, $S_5 := (a_8, a_{13}, a_{15}, a_{14}, a_{13}, a_{11}, a_{10}, a_9)$,
   $S_6 := (a_{10}, a_8, 0, 0, 0, a_{13}, a_{12}, a_{11})$, $S_7 := (a_{11}, a_9, 0, 0, a_{15}, a_{14}, a_{13}, a_{12})$,
   $S_8 := (a_{12}, 0, a_{10}, a_9, a_8, a_{15}, a_{14}, a_{13})$, $S_9 := (a_{13}, 0, a_{11}, a_{10}, a_9, 0, a_{15}, a_{14})$.
2. **return** $((S_1 + 2 \cdot S_2 + 2 \cdot S_3 + S_4 + S_5 - S_6 - S_7 - S_8 - S_9) \pmod{p_{256}})$.

Algoritem 12. Hitra redukcija po modulu $p_{256} = 2^{384} - 2^{128} - 2^{96} + 2^{32} + 1$

PODATKI : $a := (a_{23}, \ldots, a_2, a_1, a_0)$, a_i so 32-bitna besede, $0 \leq a < p_{384}^2$.
REZULTAT: $c := a \pmod{p_{384}}$.

1. Define 384-bit integers: $S_1 := (a_{11}, a_{10}, a_9, a_8, a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$,
   $S_2 := (0, 0, 0, 0, 0, a_{23}, a_{22}, a_{21}, 0, 0, 0, 0)$,
   $S_3 := (a_{23}, a_{22}, a_{21}, a_{20}, a_{19}, a_{18}, a_{17}, a_{16}, a_{15}, a_{14}, a_{13}, a_{12})$,
   $S_4 := (a_{20}, a_{19}, a_{18}, a_{17}, a_{16}, a_{15}, a_{14}, a_{13}, a_{12}, a_{23}, a_{22}, a_{21})$,
   $S_5 := (a_{19}, a_{18}, a_{17}, a_{16}, a_{15}, a_{14}, a_{13}, a_{12}, a_{20}, 0, a_{23}, 0)$,
   $S_6 := (0, 0, 0, 0, a_{23}, a_{22}, a_{21}, a_{20}, 0, 0, 0, 0)$, $S_7 := (0, 0, 0, 0, 0, 0, a_{23}, a_{22}, a_{21}, 0, 0, a_{20})$
   $S_8 := (a_{22}, a_{21}, a_{20}, a_{19}, a_{18}, a_{17}, a_{16}, a_{15}, a_{14}, a_{13}, a_{12}, a_{23})$,
   $S_9 := (0, 0, 0, 0, 0, 0, 0, a_{23}, a_{22}, a_{21}, a_{20}, 0)$, $S_{10} := (0, 0, 0, 0, 0, 0, 0, a_{23}, a_{23}, 0, 0, 0)$.
2. **return** $((S_1 + 2 \cdot S_2 + S_3 + S_4 + S_5 + S_6 + S_7 - S_8 - S_9 - S_{10}) \pmod{p_{256}})$.

Algoritem 13. Hitra redukcija po modulu $p_{521} = 2^{521} - 1$

PODATKI : $a := (a_{1041}, \ldots, a_2, a_1, a_0)$, a_i so binarna števila, $0 \leq a < p_{521}^2$.
REZULTAT: $c := a \pmod{p_{192}}$.

1. **define** 521-bit integers: $S_1 := (a_{1041}, \ldots, a_{523}, a_{522})$, $S_2 := (a_{521}, \ldots, a_2, a_1, a_0)$.
2. **return** $((S_1 + S_2) \pmod{p_{521}})$.

Če pozorno pogledamo Algoritem 13 opazimo, da števili S_1 in S_2 ne sestavimo z 32 bitnimi besedami. Razlog za to se skriva v praštevilu p_{521}, ker 521 ni večkratnik števila 32. Praštevilo $p_{521} = 2^{521} - 1$ je Mersennovo in zaradi tega modularno redukcijo izvedemo po formuli (2.1) in sicer tako, da seštejemo zgornjih 521 bitov s spodnjimi.

Oglejmo si na primeru delovanje Algoritma 10.

Primer 2.9. Hočemo izračunati vrednost števila a po modulu p_{224}:

$$a = (\texttt{a95b0058}, \texttt{74c467c0}, \texttt{a521e0dd}, \texttt{9c2748c0}, \texttt{1dbfa445}, \texttt{af1d2f0b}, \texttt{0534f200},$$
$$\texttt{36b20c40}, \texttt{408a4600}, \texttt{66bd19e3}, \texttt{5ff25709}, \texttt{b48269d8}, \texttt{38c78b6d}, \texttt{10e9bb6f}).$$

Najprej moramo izračunati števila S_i:

$S_1 = ($ 36b20c40, 408a4600, 66bd19e3, 5ff25709, b48269d8, 38c78b6d, 10e9bb6f$)$,
$S_2 = ($ 9c2748c0, 1dbfa445, af1d2f0b, 0534f200, 0, 0, 0$)$,
$S_3 = ($ 0, a95b0058, 74c467c0, a521e0dd, 0, 0, 0$)$,
$S_4 = ($ a95b0058, 74c467c0, a521e0dd, 9c2748c0, 1dbfa445, af1d2f0b, 0534f200$)$,
$S_5 = ($ 0, 0, 0, 0, a95b0058, 74c467c0, a521e0dd$)$.

Nato izračunamo $S_1 + S_2 + S_3 - S_4 - S_5$ in dobimo rezultat $c \equiv a \pmod{p_{224}}$, kjer je

$$c = (\texttt{297e54a8}, \texttt{92e082dd}, \texttt{e57ccfd1}, \texttt{6e21e125}, \texttt{ed67c53a}, \texttt{14e5f4a1}, \texttt{6692e892}).$$

2.7 Modularna aritmetika

Sedaj, ko smo navedli algoritme za modularno redukcijo, lahko povemo kako potekajo algoritmi za modularno seštevanje, odštevanje in množenje.

Pri modularnem seštevanju dveh števil iz obsega $\mathbb{Z}_p$, uporabimo Algoritem 1, pri čemer moramo med koraka 3 in 4 dodati naslednjo vrstico:

$$\textbf{if } c \geq p \textbf{ then } c = c - p.$$

Namreč pri seštevanju dveh števil iz obsega $\mathbb{Z}_p$, je lahko vsota večja oz. enaka p. Da bo vsota v $\mathbb{Z}_p$, moramo v tem primeru odšteti p.

Podobne ugotovitve veljajo za modularno odštevanje. Če odštevamo dve števili a in b, iz praštevilskega obsega $\mathbb{Z}_p$, dobimo v primeru, ko je $a \geq b$, z Algoritmom 2 pravilen rezultat. Saj je razlika večja ali enaka nič in manjša od $p - 1$. V primeru, ko je b večji od a, pa moramo razliki $a - b$ prišteti še p oz. najprej odšteti $b - a$ in od števila p odšteti to razliko.

Pri modularnem množenju pa števili zmnožimo z enim izmed omenjenih algoritmov za množenje in potem izvedemo še modularno redukcijo. Dobljeni rezultat je modularni produkt danih števil in je, kot posledica modularne redukcije, v obsegu $\mathbb{Z}_p$.

2.8 Zaključek

Navedli smo algoritme za vse osnovne aritmetične operacije ter za vse operacije v praštevilskih obsegih razen inverza. Algoritmom za računanje inverza se bomo v celoti posvetili v naslednjem poglavju.

Primerjave osnovnih algoritmov bomo navedli v četrtem poglavju. Tam se bomo podrobneje posvetili implementaciji le teh. Omenili bomo tudi težave, do katerih prihaja pri učinkoviti implementaciji, ter kako smo se z njimi soočili.

Poglavje 3

Inverz v praštevilskih obsegih

Računanje inverza v praštevilskih obsegih je zelo pomembna in časovno zelo zahtevna operacija, ki se med drugim uporablja tudi pri implementaciji eliptičnih krivulj. Zaradi vsega tega smo tej temi namenili posebno poglavje.

Algoritmi za računanje inverza so v tesni povezavi z algoritmi za iskanje največjega skupnega delitelja, ki so zelo pomemben del računalniških sistemov. Poleg tega se uporabljajo tudi za krajšanje ulomkov in reševanje Diofantskih enačb.

V tem poglavju bomo predstavili tri algoritme za iskanje največjega skupnega delitelja in njihove razširitve, preko katerih izračunamo inverz danega elementa. Najprej bomo predstavili Evklidov algoritem, enega najstarejših algoritmov, ki so še v uporabi. Nato bomo pogledali, kako deluje izboljšava Evklidovega algoritma za velika števila, tj. Lehmerjev algoritem. Na koncu poglavja bomo predstavili še Binarni algoritem, ki deluje na drugačnem principu kot prva dva algoritma.

3.1 Evklidov algoritem

Algoritem, ki ga je Evklid zapisal pred približno 2300 leti, je eden izmed najstarejših algoritmov, ki so še vedno v uporabi. Razloga za to sta njegova enostavnost in učinkovitost. Algoritem za dani pozitivni celi števili a in b vrne njun največji skupni delitelj $d = D(a, b)$. Njegova razširjena verzija, pa nam poleg največjega skupnega delitelja, vrne še celi števili u in v, za kateri velja

$$u \cdot a + v \cdot b = d.$$

In prav v tej enačbi se skriva povezava med Evklidovim algoritmom oz. algoritmi za iskanje največjega skupnega delitelja in algoritmi za iskanje inverza v praštevilskih obsegih. Namreč, če razširjeni Evklidov algoritem izvedemo za števili a in p ($a \in \mathbb{Z}_p$), dobimo števili u in v, za katere velja $u \cdot a + v \cdot p = d$. Potem iz tega sledi

$$u \cdot a + v \cdot p \equiv u \cdot a \equiv d \pmod{p}.$$

Ker je p praštevilo, je $d = 1$. Torej je število $u \pmod p$ ravno inverz števila a v obsegu $\mathbb{Z}_p$, saj velja

$$u \cdot a \equiv 1 \pmod p.$$

Preden nadaljujemo z Evklidovim algoritmom, bomo navedli nekaj lastnosti in definicij, ki veljajo za največji skupni delitelj dveh števil.

Definicija 3.1. *Največji skupni delitelj dveh celih števil a in b, ki nista obe enaki nič, je največje naravno število, ki deli obe števili a in b. Označimo ga z $D(a, b)$ in definiramo še $D(0, 0) = 0$.*

Iz definicije največjega skupnega delitelja za števili $a, b \in \mathbb{Z}$ očitno sledijo naslednje lastnosti:

$$D(a, b) = D(b, a),$$

$$D(a, b) = D(-a, b),$$

$$D(a, 0) = |a|.$$

Ideja Evklidovega algoritma je razvidna iz naslednje trditve.

Trditev 3.2. *Za poljubni celi števili a in b velja*

$$D(a, b) = D(b, a - b)$$

Dokaz. Naj bo d poljuben delitelj števil a in b. Očitno sledi, da d deli razliko $a - b$. Tako smo pokazali, da $D(a, b)$ deli $D(b, a - b)$. Naj bo d' poljuben delitelj števil b in $a - b$. Potem hitro pokažemo, da d' deli a. Namreč velja $b = u \cdot d'$ in $a - b = v \cdot d'$. Iz tega sledi $a = b + v \cdot d' = u \cdot d' + v \cdot d' = (u + v) \cdot d'$. Pokazali smo, da $D(b, a - b)$ deli $D(a, b)$. S tem je trditev dokazana, saj iz $x \mid y$ in $y \mid x$ sledi $x = y$. ■

Posledica 3.3. *Za poljubni celi števili a, b in q velja*

$$D(a, b) = D(b, a - q \cdot b)$$

Iz Posledice 3.3 je razvidna ideja Evklidovega algoritma. Namreč na začetku Evklidovega algoritma delimo število a s številom b in dobimo kvocient q in ostanek r. Iz posledice sledi $D(a, b) = D(b, a - q \cdot b) = D(b, r)$. Algoritem nadaljujemo s števili b in r. Tako postopoma zmanjšujemo števili, za katere računamo največji skupni delitelj, pri čemer se ta ohranja. Na koncu algoritma bomo dobili $D(x, 0) = x$, kar pomeni, da je zadnje od nič različno število največji skupni delitelj števil a in b. Zapišimo algoritem v

bolj znani obliki. Naj bosta $a_0 = a$, $a_1 = b$ in $a \geq b$. Evklidov algoritem izvaja naslednje korake:

$$\begin{aligned}
a_0 &= q_1 \cdot a_1 + a_2, && q_1 = \lfloor \frac{a_0}{a_1} \rfloor \geq 1,\ 0 < a_2 < a_1,\\
a_1 &= q_2 \cdot a_2 + a_3, && q_2 = \lfloor \frac{a_1}{a_2} \rfloor \geq 1,\ 0 < a_3 < a_2,\\
&\vdots\\
a_{i-1} &= q_i \cdot a_i + a_{i+1}, && q_i = \lfloor \frac{a_{i-1}}{a_i} \rfloor \geq 1,\ 0 < a_{i+1} < a_i,\\
&\vdots\\
a_{n-2} &= q_{n-1} \cdot a_{n-1} + a_n, && q_{n-1} = \lfloor \frac{a_{n-2}}{a_{n-1}} \rfloor \geq 1,\ 0 < a_n < a_{n-1},\\
a_{n-1} &= q_n \cdot a_n + 0, && q_n = \lfloor \frac{a_{n-1}}{a_n} \rfloor \geq 2,\ 0 = a_{n+1} < a_n.
\end{aligned}$$

Zaporedju $\{a_i\}$ pravimo Evklidovo zaporedje oz. zaradi naslednjega zapisa

$$a_0 = a, \quad a_1 = b,$$

$$a_{i+1} = a_{i-1} - q_i \cdot a_i \ \text{ oz. } \ a_{i+1} = a_{i-1} \pmod{a_i}$$

tudi **zaporedje ostankov**. Zaporedju $\{q_i\}$ pravimo **zaporedje kvocientov**. Za zaporedje $\{a_i\}$ velja

$$a_0 > a_1 > a_2 > \ldots > a_n > a_{n+1} = 0.$$

Iz Posledice 3.3 in definicije Evklidovega zaporedja pa, kot smo že omenili, sledi

$$D(a_0, a_1) = D(a_1, a_2) = \ldots = D(a_{n-1}, a_n) = D(a_n, a_{n+1}) = D(a_n, 0) = a_n. \tag{3.1}$$

Algoritem 14. Evklidov algoritem

PODATKI : a, $b \in \mathbb{N}$, $a \geq b$.
REZULTAT: $d := D(a, b)$.

1. $a_0 := a$, $a_1 := b$.
2. **while** $a_1 > 0$ **then**
   2.1. $q := \lfloor a_0 / a_1 \rfloor$.
   2.2. $r :=$ a_0 - $q \cdot a_1$.
   2.3. $a_0 := a_1$, $a_1 = r$.
3. **return** (a_0).

Dokažimo sedaj pravilnost Evklidovega algoritma.

Trditev 3.4. *Evklidov algoritem se konča po končnem številu korakov in vrne pravilen rezultat.*

Dokaz. Iz že pokazanega sledi, da je Evklidovo zaporedje $\{a_i\}$ padajoče ter, da so členi zaporedja nenegativna cela števila. Tako očitno sledi, da se Evklidov algoritem konča po končnem številu korakov. Iz definicije Evklidovega zaporedja in iz enakosti (3.1) sledi, da je dobljeni rezultat res največji skupni delitelj števil a in b. ■

Poglejmo si primer Evklidovega algoritma za števili $a = 29179$ in $b = 2383$.

Primer 3.5.

$$\begin{aligned} 29179 &= 12 \cdot 2383 + 583 \\ 2383 &= 4 \cdot 583 + 51 \\ 583 &= 11 \cdot 51 + 22 \\ 51 &= 2 \cdot 22 + 7 \\ 22 &= 3 \cdot 7 + 1 \\ 7 &= 7 \cdot 1 \end{aligned}$$

Sedaj si še poglejmo, kako je z časovno zahtevnostjo Evklidovega algoritma. Ogledali si bomo, kdaj algoritem izvede največ korakov.

Definicija 3.6. *Zaporedje $\{F_i\}$, kjer sta začetni vrednosti zaporedja $F_0 = 1$ in $F_1 = 1$, definirano rekurzivno z*

$$F_{i+1} = F_i + F_{i-1},$$

imenujemo Fibonaccijevo zaporedje.

Trditev 3.7. *(Lamé, 1845) Naj bosta a in b naravni števili in naj Evklidov algoritem za (a, b) izvede skupaj n korakov. Potem velja*

$$a \geq F_{n+1},\ b \geq F_n,$$

kjer je F_i i-ti člen Fibonaccijevega zaporedja.

Dokaz. Trditev bomo dokazali s pomočjo indukcije za predpostavko $a_{n-i} \geq F_i$. Spomnimo se, da velja $q_i \geq 1$ in $q_n \geq 2$. Za $i = 0$ velja $a_n \geq 1 = F_0$. Za $i = 1$ pa velja $a_{n-1} = q_n \cdot a_n \geq 1 = F_1$. Indukcijski korak očitno velja za $i \geq 2$: $a_{n-i} = q_{n-i+1} \cdot a_{n-i+1} + a_{n-i+2} \geq F_{i-1} + F_{i-2} = F_i$. ■

Posledica 3.8. *Število korakov, ki jih Evklidov algoritem izvede pri računanju največjega skupnega delitelja naravnih števil a in b $(1 \leq a, b \leq N)$, je največ*

$$\left\lceil \frac{\ln(\sqrt{5}N)}{\ln((1+\sqrt{5})/2)} \right\rceil - 2 \ \approx\ 2.078 \ln N + 1.672.$$

Dokaz, ki je bolj tehnične narave, bomo izpustili, da se ne bi spuščali v podrobnosti Fibonaccijevih zaporedij. Najdete ga lahko npr. v [7, str. 320].

Poglejmo si še, kakšna je asimptotska časovna zahtevnost Evklidovega algoritma.

Trditev 3.9. *Časovna zahtevnost Evklidovega algoritma je* $O((\ln a)(\ln b))$.

Dokaz. To oceno dobimo iz naslednjih dejstev. En korak algoritma $a_{i-1} = q_i \cdot a_i + a_{i+1}$ ima časovno zahtevnost $O((\ln a_i)(\ln q_i))$, kar sledi iz časovne zahtevnosti deljenja (glej str. 27). Za celoten algoritem dobimo

$$\sum_{i=1}^{n} O((\ln a_i)(\ln q_i)) \leq O((\ln a_1)\sum_{i=1}^{n}(\ln q_i)) \leq O((\ln a_1)(\ln \prod_{i=1}^{n} q_i)) \leq O((\ln a_1)(\ln a_0)).$$

V izpeljavi smo pri zadnjem koraku upoštevali $a_0 \geq q_1 a_1 \geq q_1 q_2 a_2 \geq q_1 q_2 \ldots q_n = \prod_{i=1}^{n} q_i$. Ker sta v našem primeru $a = a_0$ in $b = a_1$, je časovna zahtevnost res $O((\ln a)(\ln b))$. ■

Pri izračunu časovne zahtevnosti Evklidovega algoritma upoštevamo, da se spreminja velikost števil a_i. Če to ne bi upoštevali, bi imeli $O(\ln n)$ korakov zahtevnosti $O(\ln^2 n)$, kar bi skupaj zneslo $O(\ln^3 n)$.

V [7, pog. 4.5.3] si lahko pogledamo, kako je D. E. Knuth ocenil povprečno število korakov, ki jih izvede Evklidov algoritem za števili a in b ($1 \leq a, b \leq N$), z

$$\frac{12 \ln 2}{\pi^2} \ln N + 0.14 \;\approx\; 0.843 \ln N + 0.14.$$

Razširjeni Evklidov algoritem nam pri danih a in b vrne u, v in d, za katere velja $u \cdot a + v \cdot b = d$. Pri tem moramo poleg zaporedja $\{a_i\}$ računati še zaporedji $\{u_i\}$ in $\{v_i\}$, za kateri velja

$$\begin{aligned} u_0 \cdot a + v_0 \cdot b &= a_0, \\ u_1 \cdot a + v_1 \cdot b &= a_1, \\ &\vdots \\ u_i \cdot a + v_i \cdot b &= a_i, \\ &\vdots \\ u_n \cdot a + v_n \cdot b &= a_n. \end{aligned} \tag{3.2}$$

Začetne člene zaporedij $\{u_i\}$ in $\{v_i\}$ izberemo tako, da bo enakost (3.2) veljala za $i = 0, 1$. Kot očitna izbira se nam ponujajo naslednje vrednosti: $u_0 = 1$, $u_1 = 0$, $v_0 = 0$ in $v_1 = 1$.

Že prej smo omenili, da imata zaporedji $\{u_i\}$ in $\{v_i\}$ pomembno vlogo pri računanju inverza. Zaporedji $\{u_i\}$ in $\{v_i\}$ in zaporedje $\{a_i\}$ so povezani preko naslednje zveze:

$$(a_{i+1}, u_{i+1}, v_{i+1}) = (a_{i-1}, u_{i-1}, v_{i-1}) - q_i \cdot (a_i, u_i, v_i).$$

Iz te zveze očitno sledi, da če zaporedji $\{u_i\}$ in $\{v_i\}$ računamo po tej formuli, potem se zveza (3.2) ohranja in na koncu dobimo $u_n \cdot a + v_n \cdot b = a_n = d$.

Podajmo opis Razširjenega Evklidovega algoritma.

Algoritem 15. Razširjeni Evklidov algoritem

PODATKI : $a, b \in \mathbb{N}$, $a \geq b$.
REZULTAT: (u, v, d) : $u \cdot a + v \cdot b = d = D(a, b)$.

1. $a_0 := a$, $a_1 := b$, $u_0 := 1$, $u_1 := 0$, $v_0 := 0$, $v_1 := 1$.
2. **while** $a_1 > 0$ **then**
   - 2.1. $q := \lfloor a_0/a_1 \rfloor$.
   - 2.2. $r := a_0 - q \cdot a_1, a_0 := a_1, a_1 := r$.
   - 2.3. $r := u_0 - q \cdot u_1, u_0 := u_1, u_1 := r$.
   - 2.4. $r := v_0 - q \cdot v_1, v_0 := v_1, v_1 := r$.
3. **return** $((u_0, v_0, a_0))$.

Opazimo, da se Razširjeni Evklidov algoritem od navadnega Evklidovega algoritma razlikuje le po vrsticah 2.3 in 2.4. Pravilnost algoritma smo že pokazali preko dokaza pravilnosti Evklidovega algoritma in s tem, da se z algoritmom ohranja zveza $u_i \cdot a + v_i \cdot b = a_i$.

Asimptotska časovna zahtevnost Razširjenega Evklidovega algoritma je enaka asimptotski časovni zahtevnosti nerazširjene verzije in sicer $O((\ln a)(\ln b))$. To je res, saj je število korakov enako in časovna zahtevnost koraka se ne poveča. Število operacij, ki jih izvedemo na vsakem koraku, pa se poveča za faktor 3.

Oglejmo si delovanje Razširjenega Evklidovega algoritma na primeru.

Primer 3.10. Vzemimo števili $a = 29179$ in $b = 2383$ in izvedimo Razširjeni Evklidov algoritem.

$$\begin{aligned}
1 \cdot 29179 \;+\; & 0 \cdot 2383 &&= 29179 &&= A_0 \\
0 \cdot 29179 \;+\; & 1 \cdot 2383 &&= 2383 &&= A_1 \\
1 \cdot 29179 \;-\; & 12 \cdot 2383 &&= 583 &&= A_2 \\
-4 \cdot 29179 \;+\; & 49 \cdot 2383 &&= 51 &&= A_3 \\
45 \cdot 29179 \;-\; & 551 \cdot 2383 &&= 22 &&= A_4 \\
-94 \cdot 29179 \;+\; & 1151 \cdot 2383 &&= 7 &&= A_5 \\
327 \cdot 29179 \;-\; & 4004 \cdot 2383 &&= 1 &&= A_6
\end{aligned}$$

Vidimo, da je $a_6 = 1$. To pomeni, da sta si števili 29179 in 2383 tuji. ■

Poglejmo kaj se zgodi, če zadnjo enačbo gledamo po modulu 29179.

$$327 \cdot 29179 - 4004 \cdot 2383 \equiv -4004 \cdot 2383 \pmod{29179}$$

$$(29179 - 4004) \cdot 2383 \equiv 25175 \cdot 2383 \equiv 1 \pmod{29179}$$

Torej je število 25175 inverz števila 2383 v praštevilskem obsegu $\mathbb{Z}_{29179}$. Iz primera tudi opazimo, da če algoritem uporabljamo le za izračun inverza, potem ni potrebno računati zaporedja $\{u_i\}$. Saj, če je $a = p$, in če enačbe gledamo po modulu p, prvi člen $u_i \cdot p$ odpade. Kadar računamo samo inverz, lahko korak 2.3 izpustimo. Zato se število operacij, ki jih izvedemo na vsakem koraku algoritma, poveča za faktor 2 in ne 3.

Definicija 3.11. *Zaporedje $\{x_i\}$ je alternirajoče, če velja $x_n(-1)^n \geq 0$ ali $x_n(-1)^{n+1} \geq 0$.*

Pokažimo, da sta zaporedji $\{u_i\}$ in $\{v_i\}$ alternirajoči. To bomo potrebovali kasneje, ko bomo govorili o izboljšavah Evklidovega oz. Lehmerjevega algoritma.

Trditev 3.12. *Zaporedji $\{u_i\}$ in $\{v_i\}$ sta alternirajoči.*

Dokaz. To najlažje pokažemo, če definiramo zaporedji $\{x_i\}$ in $\{y_i\}$:

$$(x_0, y_0) = (1, 0),$$

$$(x_1, y_1) = (0, 1),$$

$$(x_{i+2}, y_{i+2}) = (x_i, y_i) + q_{i+1} \cdot (x_{i+1}, y_{i+1}),$$

ki imata samo pozitivne elemente. Če primerjamo zaporedji $\{x_i\}$ in $\{y_i\}$ z $\{u_i\}$ in $\{v_i\}$, opazimo naslednje:

$$u_i = (-1)^i x_i = (-1)^i |u_i|,$$

$$v_i = (-1)^{i+1} y_i = (-1)^{i+1} |v_i|,$$

$$|u_i| = x_i, \ |v_i| = y_i.$$

Za $i = 0$, 1 enakosti veljata, po indukcijskem koraku pa dobimo

$$\begin{aligned} u_{i+2} &= u_i - q_{i+1} \cdot u_{i+1} \\ &= (-1)^i x_i - q_{i+1} \cdot (-1)^{i+1} x_{i+1} \\ &= (-1)^i (x_i + q_{i+1} \cdot x_{i+1}) \\ &= (-1)^{i+2} x_{i+2} \end{aligned}$$

oz.

$$\begin{aligned} v_{i+2} &= v_i - q_{i+1} \cdot v_{i+1} \\ &= (-1)^{i+1} y_i - q_{i+1} \cdot (-1)^{i+2} y_{i+1} \\ &= (-1)^{i+1} (y_i + q_{i+1} \cdot y_{i+1}) \\ &= (-1)^{i+3} y_{i+2}. \end{aligned}$$

■

Navedimo še nekaj lastnosti, ki veljajo za zaporedji $\{u_i\}$ in $\{v_i\}$.

Trditev 3.13. *Za zaporedji* $\{u_i\}$, $\{v_i\}$ *in* $\{q_i\}$ *iz Razširjenega Evklidovega algoritma velja:*

(i) $|u_i| \leq |u_{i+1}|$ *in* $|v_i| \leq |v_{i+1}| \qquad i \geq 0,$

(ii) $q_1 \cdot |u_i| \leq |v_i|, \qquad i \geq 1.$

Dokaz. Točka (i) očitno sledi iz enakosti

$$(|u_{i+2}|, |v_{i+2}|) = (|u_i|, |v_i|) + q_{i+1} \cdot (|u_{i+1}|, |v_{i+1}|),$$

ki je posledica Trditve 3.12. Točko (ii) dokažemo z indukcijo. Za $i = 1, 2$ velja:

$$|u_1| = 0 \leq q_1 \cdot 1 = q_1 \cdot |v_1|,$$

$$|u_2| = 1 \leq q_1 = |v_2|.$$

Če upoštevamo $q_1 \cdot |u_i| \leq |v_i|$, $q_1 \cdot |u_{i+1}| \leq |v_{i+1}|$ in izvedemo indukcijski korak, dobimo

$$|v_{i+2}| = |v_i| + q_{i+1} \cdot |v_{i+1}| \geq q_1 \cdot |u_i| + q_1 \cdot q_{i+1} \cdot |u_{i+1}| = q_1 \cdot |u_{i+2}|.$$

■

3.2 Lehmerjev algoritem

Kadar imamo opravka z velikimi števili, je operacija deljenja časovno zelo zahtevna. Zaradi tega Evklidov algoritem ni učinkovit za velika števila. D.H. Lehmer [9] je opazil, da obstaja način, pri katerem se lahko izognemo deljenju z velikimi števili. Ugotovil je namreč, da dostikrat lahko deljenje velikih števili A in B nadomestimo z deljenjem njunih vodilnih delov $a = \lfloor A/2^h \rfloor$ in $b = \lfloor B/2^h \rfloor$. Pogosto bomo lahko izračunali kvocient $Q = \lfloor A/B \rfloor$ kar iz števil a in b. Tako nam ne bo potrebno deliti veliki števili A in B, ampak bomo do kvocienta Q prišli na računsko enostavnejši način.

Lehmerjeva ideja je, da s čim manj računanja poiščemo števili q in q' za kateri bo veljalo

$$q' \leq Q \leq q.$$

V algoritmu bomo primerjali q in q' in če bosta enaki, bo pomenilo, da velja $q = Q = q'$. Lehmer je predlagal naslednje vrednosti za q in q': $q' = \lfloor a_n/(b_n + 1) \rfloor$, $q = \lfloor (a_n + 1)/b_n \rfloor$. Hitro preverimo, da res velja $q' \leq Q \leq q$. Poglejmo si Lehmerjevo idejo na primeru.

Algoritem 16. Lehmerjev algoritem

PODATKI : a, $b \in \mathbb{N}$, $a \geq b$, $W = 2^{32}$.
REZULTAT: $d := D(a, b)$.

1. $A_0 := a$, $A_1 := b$.
2. **while** $A_1 \geq W$
   - 2.1. $a_0 := \lfloor A_0/2^h \rfloor$, $a_1 := \lfloor A_1/2^h \rfloor$ za neki $h > 0$ $(0 \leq a1 < a0 < W)$.
   - 2.2. $u_0 := 1$, $u_1 := 0$, $v_0 := 0$, $v_1 := 1$.
   - 2.3. **while** $(a_1 + u_1) \neq 0$ **and** $(a_1 + v_1) \neq 0$ **do**
     - 2.3.1. $q := \lfloor (a_0 + u_0)/(a_1 + u_1) \rfloor$, $q' := \lfloor (a_0 + v_0)/(a_1 + v_1) \rfloor$.
     - 2.3.2. **if** $q \neq q'$ **goto** 2.4.
     - 2.3.3. $r := a_0 - q \cdot a_1,\ a_0 := a_1,\ a_1 :=: r$.
     - 2.3.4. $r := u_0 - q \cdot u_1,\ u_0 := u_1,\ u_1 := r$.
     - 2.3.5. $r := v_0 - q \cdot v_1,\ v_0 := v_1,\ v_1 := r$.
   - 2.4. **if** $v_0 = 0$ **then**
     - 2.4.1. $(q, R) := \mathrm{Div}(A_0, A_1)$ (kvocient in ostanek dobimo hkrati pri deljenju).
     - 2.4.2. $A_0 := A_1, A_1 := R$.

     else
     - 2.4.3. $R := u_0 \cdot A_0 + v_0 \cdot A_1$, $T := u_1 \cdot A_0 + v_1 \cdot A_1$, $A_0 := R$, $A_1 := T$.
3. **Izračunaj** A_0 z Evklidovim algoritmom za A_0 in A_1.
4. **return** (A_0).

Primer 3.14. Recimo, da hočemo deliti števili $A = 768\ 454\ 923$ in $B = 542\ 167\ 814$. Namesto, da delimo števili A in B, izračunamo $q' = \left\lfloor \frac{768}{543} \right\rfloor$ in $q = \left\lfloor \frac{769}{542} \right\rfloor$. V našem primeru velja $q = q' = 1$. Ker sta q in q' enaka pomeni, da se nenakost $q' \leq \lfloor A/B \rfloor \leq q$ spremeni v enakost $q' = \lfloor A/B \rfloor = q$, in dobimo $\lfloor A/B \rfloor = 1$. Vidimo, da smo se v našem primeru izognili deljenju dveh velikih števil, ter namesto njega izvedli dve deljenji majhnih števil.

V Lehmerjevem algoritmu, podobno kot pri Evklidovem, računamo zaporedje kvocientov $\{Q_i\}$. Razlika med obema algoritmoma bo v tem, da bomo v Lehmerjevem algoritmu poskušali izračunati Q_i brez deljenja števil A_i in A_{i+1}. Kvocient Q_i bomo poskušali izračunati s pomočjo Lehmerjeve ideje ($q' \leq Q_i \leq q$). V algoritmu bomo izvajali t.i. Lehmerjeve korake (aritmetika z majhnimi števili), dokler bo veljalo $q = q'$, saj v tem primeru velja $q = Q_i$. Če predpostavimo, da smo na i-tem koraku, ter da smo k-krat zapored izvedli Lehmerjev korak, potem smo se izognili k-tim deljenjem velikih števil in eksplicitnem računanju $k - 2$ elementov Evklidovega zaporedja $\{A_i\}$. Ko kvocienta ne bosta več enaka, bomo izračunali A_{i+k} in A_{i+k+1} in nadaljevali z algoritmom. V algoritmu

se bo lahko zgodilo, da na i-tem koraku ne bomo izvedli niti enega Lehmerjevega koraka. V tem primeru bomo morali deliti veliki števili A_i in A_{i+1}. Ko bo število A_{i+1} manjše od velikosti računalniške besede W, nadaljujemo z Evklidovim algoritmom, saj v tem primeru deljenje ni več tako draga operacija. Sedaj, ko smo algoritem podrobno razložili, ga še dokažimo.

Trditev 3.15. *Lehmerjev algoritem se konča po končnem številu korakov in vrne pravilen rezultat.*

Dokaz. V prvem koraku smo, kot pri Evklidovem algoritmu, definiriali začetne vrednosti Evklidovega zaporedja $\{A_i\}$. V algoritmu smo zaradi lažjega zapisa tekoča elementa zaporedja $\{A_i\}$ označili kar z A_0 in A_1. V dokazu se bomo tem oznakam izognili in bomo predpostavili, da sta na i-tem koraku tekoča elementa A_i in A_{i+1}.

V drugem koraku vidimo, da zanko izvajamo, dokler A_{i+1} ni manjši od velikosti besede ($W = 2^{32}$). Ko je A_{i+1} manjši od W, nadaljujemo z Evklidovim algoritmom.

V korakih 2.1 do 2.3.5 imamo t.i. Lehmerjeve korake. Opazimo, da so zelo podobni korakom Razširjenega Evklidovega algoritma. Razlika je v tem, da v Lehmerjevih korakih računamo dva kvocienta, ter da končamo z zanko, ko sta kvocienta med seboj različna. Zaradi lažjega razumevanja bomo pisali $A'_0 = A_i$ in $A'_1 = A_{i+1}$. S tem bomo poudarili, da smo na začetku izvajanja Lehmerjevih korakov. V koraku 2.1 določimo števili a_0 in a_1, ki predstavljata vodilni del števil A'_0 in A'_1. Števili a_0 in a_1 dobimo tako, da odrežemo h bitov števil A'_0 in A'_1. Pri čemer število h določimo tako, da bo število a_0 velik 32 bitov. Opravka bomo imeli z zaporedji $\{A'_k\}$, $\{a_k\}$, $\{u_k\}$, $\{v_k\}$, $\{q_k\}$ in $\{q'_k\}$. Podobno kot pri Razširjenem Evklidovem algoritmu v koraku 2.2 določimo števila začetna člena zaporedij $\{u_k\}$ in $\{v_k\}$ tako, da bo veljalo

$$u_0 \cdot a_0 + v_0 \cdot a_1 = a_0, \qquad u_0 \cdot A'_0 + v_0 \cdot A'_1 = A'_0,$$

$$u_1 \cdot a_0 + v_1 \cdot a_1 = a_1, \qquad u_1 \cdot A'_0 + v_1 \cdot A'_1 = A'_1.$$

Če je q_k enak Q_{i+k}, potem so zaporedja $\{A'_k\}$, $\{a_k\}$, $\{u_k\}$, $\{v_k\}$ in $\{q_k\}$ povezana z naslednjo rekurzivno formulo:

$$(A'_{k+2}, a_{k+2}, u_{k+2}, v_{k+2}) = (A'_k, a_k, u_k, v_k) - q_{k+1}(A'_{k+1}, a_{k+1}, u_{k+1}, v_{k+1}), \tag{3.3}$$

za $k = 0, 1, 2, \ldots$. Kot smo omenili v razlagi algoritma, zaporedja $\{A'_k\}$ ne računamo, ampak izračunamo le zadnja člena zaporedja. Za zaporedja $\{A'_k\}$, $\{u_k\}$ in $\{v_k\}$ velja

$$u_k \cdot A'_0 + v_k \cdot A'_1 = A'_k. \tag{3.4}$$

Tako na koncu Lehmerjevih korakov, v koraku 2.4.3, po formuli (3.4) iz u_k, v_k, u_{k+1} in v_{k+1} izračunamo $A_{i+k} = A'_k$ in $A_{i+k+1} = A'_{k+1}$ ter nadaljujemo z algoritmom.

Pokazati moramo še, da je v zanki 2.3 kvocient Q_{i+k} vedno med kvocientoma q_k in q'_k. Na začetku imamo $u_0 = 1$, $u_1 = 0$, $v_0 = 0$, $v_1 = 1$ in očitno velja

$$\frac{a_0 + v_0}{a_1 + v_1} < \frac{A'_0}{A'_1} < \frac{a_0 + u_0}{a_1 + u_1}. \tag{3.5}$$

Torej za cele dele velja

$$q'_1 \leq Q_{i+1} \leq q_1.$$

Oglejmo si naslednje neenakosti in dokažimo, da je po enem koraku zanke 2.3 kvocient Q_{i+k} še vedno med q_k in q'_k. Iz (3.5) sledi

$$\frac{a_0 + v_0}{a_1 + v_1} - q_1 < \frac{A'_0}{A'_1} - q_1 < \frac{a_0 + u_0}{a_1 + u_1} - q_1,$$

$$\frac{a_0 + v_0 - q_1 \cdot (a_1 + v_1)}{a_1 + v_1} < \frac{A'_0 - q_1 \cdot A'_1}{A'_1} < \frac{a_0 + u_0 - q_1 \cdot (a_1 + u_1)}{a_1 + u_1}$$

in zaradi (3.3) velja

$$\frac{a_2 + v_2}{a_1 + v_1} < \frac{A'_2}{A'_1} < \frac{a_2 + u_2}{a_1 + u_1}.$$

Če prejšnjo neenakost preuredimo, dobimo

$$\frac{a_1 + u_1}{a_2 + u_2} < \frac{A'_1}{A'_2} < \frac{a_1 + v_1}{a_2 + v_2}.$$

Če gledamo samo celi del, pa velja

$$\left\lfloor \frac{a_1 + v_1}{a_2 + v_2} \right\rfloor \geq \left\lfloor \frac{A'_1}{A'_2} \right\rfloor \geq \left\lfloor \frac{a_1 + u_1}{a_2 + u_2} \right\rfloor.$$

To je pa ravno

$$q_2 \leq Q_{i+2} \leq q'_2.$$

Pokazali smo torej, da je kvocient Q_{i+k} vedno med kvocientoma q_k in q'_k.

Poglejmo si sedaj korake v točki 2.4. Na začetku se vprašajmo, kdaj je v_0 enak nič. V koraku 2.2 definiramo $v_0 = 0$ in $v_1 = 1$. Če se zanka 2.3 vsaj enkrat v celoti izvede, potem v_0 ne more več imeti vrednost 0. V tem primeru bi prišli do protislovja s tem, da se a_0 zmanjšuje, mi pa bi imeli $a_i = a_0$ za nek $i > 0$. Torej v primeru, da je $v_0 = 0$, ne moramo simulirati deljenja velikih števil in moramo deljenje dveh velikih števil A_i in A_{i+1} tudi izvesti. Če pa v_0 ni enako nič, potem smo vsaj enkrat izvedli korake 2.3.3, 2.3.4, 2.3.5 in potem moramo na koncu Lehmerjevih korakov preračunati še nove A'_k in A'_{k+1}, do katerih pridemo preko formule (3.4). To storimo v koraku 2.4.3. Števili A'_k in A'_{k+1} sta nova A_{i+k} in A_{i+k+1}.

S tem smo prišli do konca dokaza pravilnosti algoritma. Vidimo, da se lahko dostikrat izognemo deljenju velikih števil. Ko postane A_{i+1} manjši od računalniške besede, nadaljujemo z Evklidovim algoritmom, čigar pravilnost smo pa že pokazali. ■

Algoritem 17. Razširjeni Lehmerjev algoritem

PODATKI : $a, b \in \mathbb{N}$, $a \geq b$, $W = 2^{32}$.
REZULTAT: (u, v, d) : $u \cdot a + v \cdot b = d = D(a, b)$.

1. $A_0 := a$, $A_1 := b$, $U_0 := 1$, $U_1 := 0$, $V_0 := 0$, $V_1 := 1$.
2. **while** $A_1 \geq W$
   - 2.1. $a_0 := \lfloor A_0/2^h \rfloor$, $a_1 := \lfloor A_1/2^h \rfloor$ za neki $h > 0$ $(0 \leq a1 < a0 < W)$.
   - 2.2. $u_0 := 1$, $u_1 := 0$, $v_0 := 0$, $v_1 := 1$.
   - 2.3. **while** $(a_1 + u_1) \neq 0$ **and** $(a_1 + v_1) \neq 0$ **do**
     - 2.3.1. $q := \left\lfloor \frac{a_i + u_0}{a_{i+1} + u_1} \right\rfloor$, $q' := \left\lfloor \frac{a_i + v_0}{a_{i+1} + v_1} \right\rfloor$.
     - 2.3.2. **if** $q \neq q'$ **goto** 2.4.
     - 2.3.3. $r := a_0 - q \cdot a_1$, $a_0 := a_1$, $a_1 := r$.
     - 2.3.4. $r := u_0 - q \cdot u_1$, $u_0 := u_1$, $u_1 := r$.
     - 2.3.5. $r := v_0 - q \cdot v_1$, $v_0 := v_1$, $v_1 := r$.
   - 2.4. **if** $v_0 = 0$ **then**
     - 2.4.1. $(q, R) := \mathrm{Div}(A_0, A_1)$ (kvocient in ostanek dobimo hkrati pri deljenju).
     - 2.4.2. $A_0 := A_1, A_1 := R$.
     - 2.4.3. $R := U_0 - q \cdot U_1$, $U_0 := U_1$, $U_1 := R$.
     - 2.4.4. $R := V_0 - q \cdot V_1$, $V_0 := V_1$, $V_1 := R$.

     else
     - 2.4.5. $R := u_0 \cdot A_0 + v_0 \cdot A_1$, $T := u_1 \cdot A_0 + v_1 \cdot A_1$, $A_0 := R$, $A_1 := T$.
     - 2.4.6. $R := u_0 \cdot U_0 + v_0 \cdot U_1$, $T := u_1 \cdot U_0 + v_1 \cdot A_1$, $U_0 := R$, $U_1 := T$.
     - 2.4.7. $R := u_0 \cdot V_0 + v_0 \cdot V_1$, $T := u_1 \cdot V_0 + v_1 \cdot V_1$, $V_0 := R$, $V_1 := T$.
3. **Izračunaj** (U_0, V_0, A_0) z Razširjenim Evklidovim algoritmom za A_0 in A_1.
4. **return** $((U_0, V_0, A_0))$.

V Razširjenem Lehmerjevem algoritmu moramo dodatno računati zaporedji $\{U_i\}$ in $\{V_i\}$. Opazimo, da osnovnemu Lehmerjevemu algoritmu dodamo še štiri vrstice. V teh vrsticah bomo preračunavali zaporedji $\{U_i\}$ in $\{V_i\}$ in sicer na tak način, da bomo ohranjali naslednjo enakost

$$U_i \cdot A_0 + V_i \cdot A_1 = A_i. \tag{3.6}$$

Ta enakost pa je, kot smo že pokazali, ključna za izračun inverza.
Dokažimo Razširjeni Lehmerjev algoritem.

Izrek 3.16. *Razširjeni Lehmerjev algoritem se konča po končnem številu korakov in vrne pravilni rezultat.*

Dokaz. Da izračunamo inverz, moramo računati zaporedji $\{U_i\}$ in $\{V_i\}$, za katere mora veljati

$$U_0 = 1, U_1 = 0, V_0 = 0, V_1 = 1,$$

$$U_i \cdot A_0 + V_i \cdot A_1 = A_i.$$

Če hočemo, da se ta enakost ohranja, moramo vsakič, ko izračunamo A_{i+2} iz A_i in A_{i+1}, na isti način izračunati tudi U_{i+2} in V_{i+2}. Izračun A_{i+2} se izvede v korakih 2.4.1, 2.4.2 in 2.4.5. Poglejmo si prvi primer. Na i-tem koraku imamo torej enačbi

$$U_i \cdot A_0 + V_i \cdot A_1 = A_i, \tag{3.7}$$

$$U_{i+1} \cdot A_0 + V_{i+1} \cdot A_1 = A_{i+1}. \tag{3.8}$$

Če drugo enačbo pomnožimo z Q_{i+1} in odštejemo od prve dobimo:

$$(U_i - Q_{i+1} \cdot U_{i+1}) \cdot A_0 + (V_i - Q_{i+1} \cdot V_{i+1}) \cdot A_1 = (A_i - Q_{i+1} \cdot A_{i+1})$$

oz. ravno

$$U_{i+2} \cdot A_0 + V_{i+2} \cdot A_1 = A_{i+2}.$$

V drugem primeru, tj. v koraku 2.4.5, pa moramo po nekaj izvedenih korakih 2.3 izračunati, iz formule (3.4), vrednosti A_{i+2} in A_{i+3} iz A_i in A_{i+1} ter u_j, u_{j+1}, v_j in v_{j+1}. Če pomnožimo enačbo (3.7) z u_0 in enačbo (3.8) z v_0 dobimo enačbi

$$u_0 \cdot U_i \cdot A_0 + u_0 \cdot V_i \cdot A_1 = u_0 \cdot A_i,$$

$$v_0 \cdot U_{i+1} \cdot A_0 + v_0 \cdot V_{i+1} \cdot A_1 = v_0 \cdot A_{i+1},$$

ki ju nato seštejemo v

$$(u_0 \cdot U_i + v_0 \cdot U_{i+1}) \cdot A_0 + (u_0 \cdot V_i + v_0 \cdot V_{i+1}) \cdot A_1 = u_0 \cdot A_i + v_0 \cdot A_{i+1}.$$

Zadnja enačba je ravno

$$U_{i+2} \cdot A_0 + V_{i+2} \cdot A_1 = A_{i+2}.$$

Če pa enačbi (3.7) in (3.8) pomnožimo z u_1 in v_1 in ju nato seštejemo dobimo

$$(u_1 \cdot U_i + v_1 \cdot U_{i+1}) \cdot A_0 + (u_1 \cdot V_i + v_1 \cdot V_{i+1}) \cdot A_1 = u_1 \cdot A_i + v_1 \cdot A_{i+1},$$

kar je po formuli (3.6) ravno

$$U_{i+3} \cdot A_0 + V_{i+3} \cdot A_1 = A_{i+3}.$$

S tem smo prišli do konca dokaza pravilnosti algoritma. Vidimo, da se ohranja enačba (3.6), in da se števila A_i zmanjšujejo. Ko je A_1 manjši od računalniške besede, potem nadaljujemo z Razširjenim Evklidovim algoritmom, čigar pravilnost smo že pokazali. ■

Poglejmo kakšno časovno zahtevnost imata Lehmerjev in Razširjeni Lehmerjev algoritem.

Trditev 3.17. *Časovna zahtevnost Lehmerjevega in Razširjenega Lehmerjevega algoritma je* $O(\ln^2 n)$.

Dokaz. Knuth [7, str. 305] je ugotovil, da je kvocient $Q_i = \lfloor A_{i-1}/A_i \rfloor$ manjši od 1000 v 99.856 odstotkih primerov. Tako lahko skoraj vedno izračunamo Q_i s pomočjo aritmetike majhnih števil. V primeru, da kvocient Q_i ne moremo izračunati na tak način, izvedemo Evklidov korak za števili A_{i-1} in A_i. S tem smo pokazali, da asimptotska časovna zahtevnost Lehmerjevega algoritma, ne more biti slabša od asimptotske časovne zahtevnosti Evklidovega algoritma, ki je $O(\ln^2 n)$.

Razširjeni Lehmerjev algoritem se od Lehmerjevega algoritma razlikuje po štirih dodatnih vrsticah v koraku 2.4, ki so računsko primerljive z operacijami, ki jih izvajamo znotraj tega koraka. Te vrstice sicer vplivajo na absolutno časovno zahtevnost ne pa na asimptotsko. Zahtevnost koraka 2.4 se tako poveča za faktor 3, kot posledica računanja zaporedij $\{U_i\}$ in $\{V_i\}$. ■

Poglejmo si na naslednjem primeru, kako deluje Lehmerjev algoritem. Zaradi lažjega zapisa in enostavnosti bomo vzeli, da bo velikost besede W enaka 1000 in ne 2^{32}.

Primer 3.18. Pogledali si bomo primer Lehmerjevega algoritma za $a = 768\ 454\ 923$ in $b = 542\ 167\ 814$. V Tabeli 3.1 si lahko ogledamo, kako izgledajo glavni koraki algoritma. Tabela 3.2 prikazuje korake z aritmetiko z majhnimi števili.

A_0	A_1	q	q'	referenca
768 454 923	542 167 814	1	1	Tabela 3.2 (i)
89 593 596	47 099 917	1	1	Tabela 3.2 (ii)
4 606 238	1 037 537	4	4	Tabela 3.2 (iii)
456 090	125 357	3	3	Tabela 3.2 (iv)
34 681	10 657	3	3	Tabela 3.2 (v)
10 657	2 710	3	3	Tabela 3.2 (vi)
2 527	183			Razširjeni Evklidov alg.
183	148			Razširjeni Evklidov alg.
148	35			Razširjeni Evklidov alg.
35	8			Razširjeni Evklidov alg.
8	3			Razširjeni Evklidov alg.
3	2			Razširjeni Evklidov alg.
2	1			Razširjeni Evklidov alg.
1	0			Razširjeni Evklidov alg.

Tabela 3.1: Lehmerjev algoritem (glej Primer 3.18).

Opazimo, da je v našem primeru relativno majhno število korakov, ki uporabljajo aritmetiko z majhnimi števili. To lahko pojasnimo z velikostjo besede W. Saj, če je

beseda W velika $2^{32} = 4\ 294\ 967\ 296$, kot v današnjih računalnikih, in ne 1000 kot v našem primeru, se število korakov, ki uporabljajo aritmetiko z majhnimi števili poveča.

Preden si bomo pogledali izboljšave Lehmerjevega algoritma, navedimo lastnost, ki

	a_0	a_1	u_0	v_0	u_1	v_1	q	q'
(i)	768	542	1	0	0	1	1	1
	542	226	0	1	1	-1	2	2
	226	90	1	-1	-2	3	2	2
	90	46	-2	3	5	-7	1	2
(ii)	895	470	1	0	0	1	1	1
	470	425	0	1	1	-4	1	1
	425	45	1	-1	-1	2	9	9
	45	20	-1	2	10	-19	1	47
(iii)	460	103	1	0	0	1	4	4
	103	48	0	1	1	-4	2	2
	48	7	1	-4	-2	9	9	2
(iv)	456	125	1	0	0	1	3	3
	125	81	0	1	1	-3	1	1
	81	44	1	-3	-1	4	1	1
	44	37	-1	4	2	-7	1	1
	37	7	2	-7	-3	11	9	1
(v)	346	106	1	0	0	1	3	3
	106	28	0	1	1	-3	3	4
(vi)	106	27	1	0	0	1	3	3
	27	25	0	1	1	-3	1	1
	25	2	1	-3	-1	4	26	3

Tabela 3.2: Aritmetika z majhnimi števili iz Primera 3.18.

velja za zaporedja $\{a_i\}$, $\{u_i\}$ in $\{v_i\}$, ki jih najdemo v t.i. Lehmerjevih korakih.

Trditev 3.19. *Naj bodo $\{a_i\}$, $\{u_i\}$ in $\{v_i\}$ zaporedja definirana v Lehmerjevem algoritmu in naj bo W velikost računalniške besede. Za vsak $i \leq k$, kjer je k število Lehmerjevih korakov, velja*

$$0 \leq a_i + u_i \leq W,$$

$$0 \leq a_i + v_i \leq W.$$

Dokaz. Naj bo $x_i = a_i + u_i$ in $y_i = a_i + v_i$. Ko izvedemo Lehmerjev korak velja $q_i = \lfloor a_{i-1}/a_i \rfloor = \lfloor (a_{i-1} + u_{i-1})/(a_i + u_i) \rfloor = \lfloor (a_{i-1} + v_{i-1})/(a_i + v_i) \rfloor$. Iz (3.3) sledi, da tudi za zaporedji $\{x_i\}$ in $\{y_i\}$ ter za $i \leq k$, kjer je k število Lehmerjevih korakov, velja rekurzija $(x_{i+1}, y_{i+1}) = (x_{i-1}, y_{i-1}) - q_i \cdot (x_i, y_i)$. To pa pomeni, da sta $\{x_i\}$ in $\{y_i\}$ Evklidovi zaporedji, torej sta pozitivni in padajoči. Za $i = 0, 1$ neenakosti očitno sledita iz definicij zaporedij $\{a_i\}$, $\{u_i\}$ in $\{v_i\}$. Skupaj smo torej pokazali, da velja $0 \leq x_i,\ y_i \leq W$

za $i \leq k$. ■

Lehmer je za izhod iz zanke 2.3 računal kvocienta $q = \lfloor (a_i + u_i)/(a_{i+1} + u_{i+1}) \rfloor$ in $q' = \lfloor (a_i + v_i)/(a_{i+1} + v_{i+1}) \rfloor$. Nato ju je primerjal med seboj, in če sta bila enaka nadaljeval, sicer pa šel na korak 2.4 oz. na aritmetiko z velikim števili. Collins [3] je razvil boljši pogoj

$$a_{i+1} \geq |v_{i+1}| \text{ in } a_i - a_{i+1} \geq |v_{i+1} - v_i| \text{ za vse } i \leq k,$$

kjer je k število korakov, pri katerih se zaporedji q_i in Q_i ujemata. Pogoj ima to prednost, da računamo le en kvocient in ne dva ter samo zaporedje v_i in ne u_i. Na koncu izračunamo u_k in u_{k+1} iz naslednjih formul

$$u_k \cdot a + v_k \cdot b = a_k \Rightarrow u_k = (a_k - v_k \cdot b)/a,$$

$$u_{k+1} \cdot a + v_{k+1} \cdot b = a_{k+1} \Rightarrow u_{k+1} = (a_{k+1} - v_{k+1} \cdot b)/a.$$

T. Jebelean [5] je ta pogoj še malo izboljšal.

Izrek 3.20. Naj bosta $a > b > 0$ naravni števili. Potem se k členov zaporedja kvocientov za začetne vrednosti (a, b) in $(2^h a + A', 2^h b + B')$ ujema za poljuben $h > 0$ in za poljubna $A', B' < 2^h$ natanko takrat, ko za vse $i \leq k$ velja

$$\begin{gathered} a_{i+1} + u_{i+1} \geq 0, \quad a_i + v_i \geq a_{i+1} + v_{i+1} \text{ v primeru, ko je } i \text{ sod} \\ \text{ali} \\ a_{i+1} + v_{i+1} \geq 0, \quad a_i + u_i \geq a_{i+1} + u_{i+1} \text{ v primeru, ko je } i \text{ lih.} \end{gathered}$$

Dokaz. Uporabljali bomo iste oznake kot dosedaj. Zaporedje A_i bo zaporedje ostankov za dan par števil

$$(A, B) = (2^h a + A', 2^h b + B').$$

Če preoblikujemo enakost (3.5), dobimo za lihe i

$$\frac{a_{i-1} + v_{i-1}}{a_i + v_i} < \frac{a_{i-1}}{a_i}, \frac{A_{i-1}}{A_i} < \frac{a_{i-1} + u_{i-1}}{a_i + u_i}.$$

V algoritmu izvajamo Lehmerjeve korake, če velja $a_i + v_i \neq 0$ in $a_i + u_i \neq 0$. V tem primeru ne delimo z 0. Z upoštevanjem (3.3) dobimo

$$\frac{q_i(a_i + v_i) + (a_{i+1} + v_{i+1})}{a_i + v_i} < \frac{q_i a_i + a_{i+1}}{a_i}, \frac{Q_i A_i + A_{i+1}}{A_i} < \frac{q_i(a_i + u_i) + (a_{i+1} + u_{i+1})}{a_i + u_i},$$

$$q_i + \frac{a_{i+1} + v_{i+1}}{a_i + v_i} < q_i + \frac{a_{i+1}}{a_i}, Q_i + \frac{A_{i+1}}{A_i} < q_i + \frac{a_{i+1} + u_{i+1}}{a_i + u_i}. \tag{3.9}$$

Vemo, da sta ulomka a_{i+1}/a_i in A_{i+1}/A_i na intervalu $[0, 1)$. Tako enakost $q_i = Q_i$ sledi iz naslednjih dveh neenakosti:

$$\frac{a_{i+1} + v_{i+1}}{a_i + v_i} \geq 0, \tag{3.10}$$

$$\frac{a_{i+1}+u_{i+1}}{a_i+u_i} \leq 1. \tag{3.11}$$

To je res saj, če v tem primeru gledamo le celi del v (3.9), dobimo

$$q_i = q_i = Q_i < q_i + 1.$$

V (3.10) upoštevamo, da je v_i pozitiven za liha števila i, in dobimo

$$a_{i+1} + v_{i+1} \geq 0.$$

Iz Trditve 3.19 sledi, da je $a_i + u_i > 0$, in s tem (3.11) preide v

$$a_i + u_i \geq a_{i+1} + u_{i+1}.$$

S tem smo pokazali izrek za lihe i. Če v (3.9) upoštevamo, da je $q_i = Q_i$, ter v vseh štirih členih odštejemo q_i dobimo

$$\frac{a_{i+1}+v_{i+1}}{a_i+v_i} < \frac{a_{i+1}}{a_i}, \frac{A_{i+1}}{A_i} < \frac{a_{i+1}+u_{i+1}}{a_i+u_i},$$

oz.

$$\frac{a_i+u_i}{a_{i+1}+u_{i+1}} < \frac{a_i}{a_{i+1}}, \frac{A_i}{A_{i+1}} < \frac{a_i+v_i}{a_{i+1}+v_{i+1}}.$$

Nadaljujemo z algoritmom in na podoben način za sode i dobimo

$$a_{i+1} + u_{i+1} \geq 0 \quad \text{in} \quad a_{i+1} + v_{i+1} \leq a_i + v_i.$$

■

Izrek 3.20 nam pove ne samo zadosten pogoj, da velja $q_i = Q_i$, ampak tudi potreben pogoj, da velja $q_i = Q_i$ za vse $i \leq k_{max}$, pri čemer je k_{max} število Lehmerjevih korakov. Hitro prepričamo, da je to res. Če kakšen od pogojov v Izreku 3.20 ne bi veljal, potem bi lahko hitro našli števili $q_i \neq Q_i$, za kateri bi neenakost 3.9 kljub temu veljala.

Če primerjamo Collinsov in Jebeleanov pogoj in upoštevamo Trditev 3.13, dobimo za lihe i

$$a_{i+1} \geq -v_{i+1},\ a_i - a_{i+1} \geq -(v_{i+1} - v_i) \geq u_{i+1} - u_i$$

in za sode i

$$a_{i+1} \geq v_{i+1} \geq -u_{i+1},\ a_i - a_{i+1} \geq v_{i+1} - v_i.$$

Vidimo, da so Jebeleanovi pogoji boljši, v smislu, da niso tako omejujoči oz. strogi kot Collinsovi.

Podobno kot pri Razširjenem Evklidovem algoritmu, tudi pri Razširjenem Lehmerjevem algoritmu ne računamo zaporedja $\{U_i\}$, če algoritem uporabimo samo za izračun inverza elementa b. V tem primeru Algoritem 17 izvedemo za podatke $a = p$ in b. Torej za izračun inverza v Algoritmu 17 preskočimo koraka 2.4.3 in 2.4.6.

3.3 Binarni algoritem

Evklidov algoritem in njegova izboljšava Lehmerjev algoritem temeljita na deljenju členov zaporedja $\{A_i\}$ oz. njihovih vodilnih členov. Binarni algoritem temelji na drugačnem principu. Preden ga razložimo, si najprej poglejmo nekaj lem o največjih skupnih deliteljih.

Lema 3.21. Za naravni števili a in b veljajo naslednje enakosti:

(i) $D(2 \cdot a, 2 \cdot b) = 2 \cdot D(a, b)$

(ii) $D(2a - 1, 2 \cdot b) = D(2a - 1, b)$

(iii) $D(a, b) = D(a - b, b)$.

Ta lema je osnova za binarni algoritem. Saj preko prve točke najprej dobimo največjo potenco $K = 2^k$, ki deli števili a in b. Po deljenju števil a in b s K je vsaj eno število liho. Nato ponavljamo drugo točko, dokler nista obe števili lihi. Po tem uporabimo tretjo točko, kjer odštejemo dve lihi števili, in tako dobimo spet eno sodo število. Nato spet začnemo z drugo točko in tako naprej, dokler ne postane eno izmed števil a in b enako nič.

Za lažje razumevanje Razširjenega Binarnega algoritma bomo prvo navedli Binarni algoritem, ki ga uporabljamo za iskanje največjega skupnega delitelja dveh števil, in nato še njegovo razširjeno različico.

Algoritem 18. Binarni algoritem

PODATKI : $a, b \in \mathbb{N}$, $a \geq b$.
REZULTAT: $d = \mathrm{D}(a, b)$.

1. $K := 1$.
2. **while** (a, b sodi števili) **then**
   2.1. $a := a/2$, $b := b/2$, $K := 2 \cdot K$.
3. **while** $a \neq 0$ **then**
   3.1. **while** (a sodo število) **then** $a := a/2$.
   3.2. **while** (b sodo število) **then** $b := b/2$.
   3.3. **if** $a \geq b$ **then** $a := a - b$ **else** $b := b - a$.
4. **return** $(K \cdot b)$.

Prepričajmo se v pravilnost algoritma.

Trditev 3.22. *Binarni algoritem se konča po končnem številu korakov in vrne pravilni rezultat.*

Dokaz. Pri dokazu bomo uporabili Lemo 3.21. Najprej opazimo, da v 2. koraku, delimo števili a in b z 2, dokler sta obe števili sodi. Hkrati s tem izračunamo K, največjo potenco števila 2, ki še deli obe števili a in b. Zaradi lažjega zapisa bomo v nadaljevanju dokaza pisali $a' = a/K$ in $b' = b/K$. V koraku 3.1 delimo število a' z 2 dokler ne postane liho. V koraku 3.2 postopek ponovimo za število b'. Iz točke (ii) sledi, da se s tem ne spremeni največji skupni delitelj. Pred korakom 3.3 sta obe števili lihi. V koraku 3.3 odštejemo manjše število od večjega in razliko shranimo v večje število. Iz točke (iii) sledi, da se z odštevanjem ne spremeni največji skupni delitelj števil a' in b'. Po koraku 3.3 torej postane eno izmed števil sodo in lahko nadaljujemo z zanko. Pokazali smo, da se z izvajanjem zanke 3, največji skupni delitelj števil a' in b' ne spreminja. Na koncu algoritma, po točki (i) Leme 3.14, pomnožimo največji skupni delitelj števil a' in b' z številom K in tako dobimo največji skupni delitelj števil a in b.

Poglejmo si sedaj ali in kdaj se algoritem konča. Vidimo, da se vsaj eno izmed števil a' in b' zmanjša po vsaki ponovitvi koraka 3. Algoritem se konča, ko je število a' enako nič. Do tega lahko pride le v primeru, ko sta bili v koraku 3.3 števili a' in b' enaki. Kar pomeni, da je $D(a', b') = d'$ oz. $D(a, b) = K \cdot d'$. S tem smo pokazali, da se algoritem konča in vrne pravilen rezultat. ■

Poglejmo si kako je s časovno zahtevnostjo Binarnega algoritma.

Trditev 3.23. *Časovna zahtevnost Binarnega algoritma je* $O(\ln^2 n)$.

Dokaz. Vidimo, da se v vsaki ponovitvi 3. koraka, eno izmed števil zmanjša vsaj za pol in tako imamo največ $O(\ln n)$ teh korakov. Vsak korak pa je sestavljen iz osnovnih operacij in sicer deljenja z dva (premik za 1 v desno oz. *RightShift*) ter odštevanja. Obe operaciji imata časovno zahtevnost $O(\ln n)$). Torej ima Binarni algoritem časovno zahtevnost $O(\ln^2 n)$. ■

Binarni algoritem opravi sicer več korakov kot Evklidov algoritem, ampak so ti koraki sestavljeni iz operacij, ki so zelo hitre. V naslednjem poglavju bomo videli, kako se obnese ta princip v primerjavi z Evklidovim oz. Lehmerjevim algoritmom.

Primer 3.24. Naj bosta a in b enaka kot v Primeru 3.18 ($a = 768454923$, $b = 542167814$). Poglejmo si tabelo 3.3.. Opazimo, da se izvede natanko eden izmed korakov 3.1 in 3.2, saj je le eden izmed a in b sod v koraku 3. Po koncu korakov 3.1 in 3.2 sta spet obe števili lihi in z odštevanjem dveh lihih števil dobimo spet eno sodo število.

Poglejmo si sedaj še Razširjeni Binarni algoritem. Prav tako kot pri Razširjenemu Evklidovem in Lehmerjevem algoritmu, moramo tudi tu za vsak $i \in \{0, \ldots, n\}$ ohranjati enačbo

$$U_i \cdot a + V_i \cdot b = A_i.$$

a	b	i	Po koraku?	a	b	i	Po koraku?
768 454 923	542 167 814	0	Začetek	49 410	751	12	Korak 3.3
768 454 923	271 083 907	1	Korak 3.2	24 705	751	13	Korak 3.1
497 371 016	271 083 907	1	Korak 3.3	23 954	751	13	Korak 3.3
62 171 377	271 083 907	2	Korak 3.1	11 977	751	14	Korak 3.1
62 171 377	208 912 530	2	Korak 3.3	11 226	751	14	Korak 3.3
62 171 377	104 456 265	3	Korak 3.2	5 613	751	15	Korak 3.1
62 171 377	42 284 888	3	Korak 3.3	4 862	751	15	Korak 3.3
62 171 377	5 285 611	4	Korak 3.2	2 431	751	16	Korak 3.1
56 885 766	5 285 611	4	Korak 3.3	1 680	751	16	Korak 3.3
28 442 883	5 285 611	5	Korak 3.1	105	751	17	Korak 3.1
23 157 272	5 285 611	5	Korak 3.3	105	646	17	Korak 3.3
2 894 659	5 285 611	6	Korak 3.1	105	323	18	Korak 3.2
2 894 659	2 390 952	6	Korak 3.3	105	218	18	Korak 3.3
2 894 659	298 869	7	Korak 3.2	105	109	19	Korak 3.2
2 595 790	298 869	7	Korak 3.3	105	4	19	Korak 3.3
1 297 895	298 869	8	Korak 3.1	105	1	20	Korak 3.2
999 026	298 869	8	Korak 3.3	104	1	20	Korak 3.3
499 513	298 869	9	Korak 3.1	13	1	21	Korak 3.1
200 644	298 869	9	Korak 3.3	12	1	21	Korak 3.3
501 61	298 869	10	Korak 3.1	3	1	22	Korak 3.1
501 61	248 708	10	Korak 3.3	2	1	22	Korak 3.3
501 61	62 177	11	Korak 3.2	1	1	23	Korak 3.1
501 61	12 016	11	Korak 3.3	0	1	23	Korak 3.3
501 61	751	12	Korak 3.2				

Tabela 3.3: Binarni algoritem (glej Primer 3.24).

Algoritem 19. Razširjeni Binarni algoritem

PODATKI : a, $b \in \mathbb{N}$, $a \geq b$.
REZULTAT: (u, v, d) : $u \cdot a + v \cdot b = d = \mathrm{D}(a, b)$.

1. $a' := a$, $b' := b$, $K := 1$.
2. **while** (a', b' oba sodi števili) **then**
   2.1. $a' := a'/2$, $b' := b'/2$, $K := 2 \cdot K$.
3. $A_0 := a'$, $A_1 := b'$, $U_0 := 1$, $U_1 := 0$, $V_0 := 0$, $V_1 := 1$.
4. **while** $A_0 \neq 0$ **then**
   4.1. **while** (A_0 sodo število) **then**
      4.1.1. $A_0 := A_0/2$.
      4.1.2. **if** (U_0, V_0 sodi števili) **then**
         $U_0 := U_0/2$, $V_0 := V_0/2$.
         else
         $U_0 := (U_0 + b')/2$, $V_0 := (V_0 - a')/2$.
   4.2. **while** (A_1 sodo število) **then**
      4.2.1. $A_1 := A_1/2$.
      4.2.2. **if** (U_1, V_1 sodi števili) **then**
         $U_1 := U_1/2$, $V_1 := V_1/2$.
         else
         $U_1 := (U_1 + b')/2$, $V_1 := (V_1 - a')/2$.
   4.3. **if** $A_0 \geq A_1$ **then**
      4.3.1 $A_0 := A_0 - A_1$, $U_0 := U_0 - U_1$, $V_0 := V_0 - V_1$.
      else
      4.3.2$A_1 := A_1 - A_0$, $U_1 := U_1 - U_0$, $V_1 := V_1 - V_0$.
5. **return** $((K \cdot U_1, K \cdot V_1, K \cdot A_1))$.

Prepričajmo se o pravilnosti Razširjenega Binarnega algoritma.

Trditev 3.25. *Razširjeni Binarni algoritem se konča po končnem številu korakov in vrne pravilni rezultat.*

Dokaz. Prepričajmo se, da algoritem res ohranja enačbo $U_i \cdot a' + V_i \cdot b' = A_i$. Za $i = 0, 1$ enačba očitno velja. Poglejmo si sedaj, kako koraka 4.1 in 4.2 vplivata na to enačbo. V prvem primeru (korak 4.1.2) sta U_0 in V_0 sodi števili in velja

$$\frac{U_i}{2} \cdot a' + \frac{V_i}{2} \cdot b' = \frac{A_i}{2},$$

oz.

$$U_{i+1} \cdot a' + V_{i+1} \cdot b' = A_{i+1}.$$

Če pa je vsaj en izmed U_0 in V_0 lih, imamo:

$$U_i \cdot a' + V_i \cdot b' = A_i \tag{3.12}$$

$$\begin{gathered} U_i \cdot a' + a' \cdot b' - a' \cdot b' + V_i \cdot b' = A_i \\ (U_i + b') \cdot a' + (V_i - a') \cdot b' = A_i \\ \frac{U_i + b'}{2} \cdot a' + \frac{V_i - a'}{2} \cdot b' = \frac{A_i}{2} \\ U_{i+1} \cdot a' + V_{i+1} \cdot b' = A_{i+1}. \end{gathered} \tag{3.13}$$

V enakosti (3.13) vidimo, da delimo $U_i + b'$ in $V_i - a'$ z 2. Pokazati moramo, da sta ta kvocienta celi števili. Spomnimo se, da je po koraku 2 vsaj eno izmed števil a in b liho. Iz enačbe (3.12) in dejstva, da je A_i sod, sledi, da sta člena $U_i \cdot a'$ in $V_i \cdot b'$ oba liha ali oba soda. Če sta oba člena liha, potem so števila U_i, V_i, a' in b' liha in s tem števili $U_i + b'$ in $V_i - a'$ obe deljivi z 2. Če pa sta oba člena soda, imamo dve možnosti. V prvi sta U_i in b' soda, v drugi pa V_i in a'. V obeh primerih sta števili $U_i + b'$ in $V_i - a'$ sodi. Na isti način pokažemo, da se v koraku 4.2 enačba (3.12) ohranja.

Če je A_j večji od A_{j+1}, potem v koraku 4.3.1 odštejemo enačbo (3.12) za $i = j$ od enačbe (3.12) za $i = j + 1$:

$$(U_j - U_{j+1}) \cdot a' + (V_j - V_{j+1}) \cdot b' = A_j - A_{j+1},$$

$$U_{j+2} \cdot a' + V_{j+2} \cdot b' = A_{j+2}.$$

Obratno naredimo v primeru, če je A_{j+1} večji od A_j.

Iz dokaza Binarnega algoritma sledi, da se Razširjeni Binarni algoritem konča po končnem številu korakov. Pokazali smo, da se enakost (3.12) ohranja, in s tem pokazali pravilno delovanje razširjenega algoritma. ■

Poglejmo si na nekaj korakih konkretnega primera idejo Razširjenega Binarnega algoritma.

$$\begin{aligned}
1 \cdot 29179 \;&+\; 0 \cdot 2383 \;&=\; 29179 & \\
0 \cdot 29179 \;&+\; 1 \cdot 2383 \;&=\; 2383 & \\
1 \cdot 29179 \;&-\; 1 \cdot 2383 \;&=\; 26796 & \text{ korak 4.4} \\
\frac{1 + 2383}{2} \cdot 29179 \;&+\; \frac{-1 - 29179}{2} \cdot 2383 \;&=\; \frac{26796}{2} & \text{ korak 4.1} \\
1192 \cdot 29179 \;&-\; 14590 \cdot 2383 \;&=\; 13398 & \\
\frac{1192}{2} \cdot 29179 \;&-\; \frac{14590}{2} \cdot 2383 \;&=\; \frac{13398}{2} & \text{ korak 4.1} \\
596 \cdot 29179 \;&-\; 7295 \cdot 2383 \;&=\; 6699 &
\end{aligned}$$

Pri računanju inverza v obsegu $\mathbb{Z}_p$, lahko preskočimo 2. korak Razširjenega Binarnega algoritma. Namreč število a je v tem primeru enako praštevilu p, ki pa ni deljivo z 2.

Iz istih razlogov kot pri Evklidovem in Lehmerjevem algoritmu pri računanju inverza, ne računamo zaporedja $\{U_i\}$.

Knuth [7, str. 307-313] je na približnem modelu računal povprečno število korakov pri izvajanju Binarnega algoritma. Posledica približnega modela so približne ocene, ki se ne ujemajo z njegovimi empirično pridobljenimi ocenami. Ugotovil je, da za poljubni števili med 0 in 2^N, Binarni algoritem izvede

$$0.70N + O(1) \text{ odštevanj} \quad \text{in} \quad 1.41N + O(1) \text{ premikov v desno.}$$

V naslednjem poglavju bomo preverili kako se Knuthove ocene ujemajo z rezultati naših testiranj.

Zaključek

Navedli smo tri najboljše in posledično najbolj razširjene algoritme za izračun največjega skupnega delitelja. Razširjene verzije teh algoritmov uporabljamo pri računanju inverza v $\mathbb{Z}_p$. Za majhna števila je Evklidov algoritem še vedno najenostavnejši in dovolj hiter za uporabo. Pri velikih številih sta pa boljši izbiri Lehmerjev in Binarni algoritem. Kateri izmed njiju je hitrejši, pa bomo videli v naslednjem poglavju.

Poglavje 4

Implementacija

V tem poglavju bomo opisali okolje, v katerem smo implementirali algoritme za aritmetiko v praštevilskih obsegih. Navedli bomo težave, s katerimi se soočimo pri učinkoviti implementaciji in poskušali bomo najti načine, kako jih rešimo. Na koncu poglavja bomo predstavili rezultate testiranj in na njihovi podlagi poskušali priti do določenih zaključkov, ki nam pomagajo izbrati med različnimi algoritmi.

4.1 Okolje

Kot smo že omenili, je učinkovita implementacija pogojena z okoljem uporabe. Mi smo za okolje vzeli Pentium II 300 MHz računalnik z zadostno količino spomina, tako da le ta ne vpliva na implementacijo. Program **ZpTests.exe** je bil preveden z Borlandovim C++ Builder-jem. Testiranja pa so potekala v Windows XP operacijskem sistemu.

V programu smo velika naravna števila predstavili z dvema razredoma **NumZZ** in **NumZp**. V obeh razredih velika števila predstavimo s tabelo nepredznačenih celih števil (*unsigned long int*) s to razliko, da v drugem razredu poskrbimo, da so števila elementi praštevilskega obsega $\mathbb{Z}_p$ za neko praštevilo p. Da bi dosegli čim večjo učinkovitost, smo na določenih mestih uporabili kodo napisano v zbirniku (ang. assembler). S tem smo dosegli večjo kontrolo nad samimi matematičnimi operacijami in algoritmi. Razlika med učinkovitostjo med obema verzijama je bila več kot očitna (tudi za faktor 2 in več).

4.2 Funkcije

V tem razdelku bomo najprej predstavili implementacije algoritme za aritmetiko z naravnimi števili.

Seštevanje in odštevanje

Implementaciji seštevanja in odštevanja sta enostavni. Težava je le dejstvo, da ukaza **Add_with_carry** in **Sub_with_borrow** nista podprta v programskem jeziku C++. Zato smo korak 2.1 Algoritma 1 in Algoritma 2 sprogramirali v zbirniku. Pri tem se pojavi problem s shranjevanjem 'carry' oz. 'borrow' števke. Namreč veliko ukazov uporablja in s tem spreminja del registra, kjer se hrani ta števka (CF bit oz. 'carry flag'). Tako imamo lahko v naslednjem koraku spremenjeno prenosno števko. Zato smo to števko eksplicitno shranili in glede na njo v naslednjem koraku določili CF bit. Pri seštevanju in odštevanju imamo zaradi enostavnosti obeh algoritmov možnost dodatne pohitritve z uporabo zbirnika za celoten algoritem. Vprašanje je, če bi se lahko v tem primeru izognili eksplicitnemu shranjevanju prenosne števke.

Množenje

Za množenje dveh velikih števil smo navedli 3 algoritme. Algoritem 3 je običajen algoritem za množenje dveh števil. Prednost algoritma je enostavnost in zato posledično lažji prenos kode v zbirnik, kar vpliva na hitrost algoritma. Majhna pomankljivost tega algoritma je klicanje spremenljivke c_{i+j} v koraku 2.2. Pri res učinkovitih implementacijah pridejo do izraza tudi klici spremenljivk in uporaba pomožnih spremenljivk.

Algoritem 4 ima prednost ravno na tem področju. Iz algoritma vidimo, da spremenljivke c_{i+j} uporabimo samo za shranjevanje besed produkta, in jih drugače ne uporabljamo v algoritmu. Spremenjlivke r_0, r_1 in r_2 naj bi v našem algoritmu predstavljale 3 registre. S tem se izognemo uporabi pomožnih spremenljivk. Vendar je slabost PII procesorjev majhno število registrov. Zato moramo biti zelo pozorni pri zapisu koraka 2.1 v zbirnik. Pri Algoritmu 4 predstavlja največ težav učinkovit izračun vseh možnih elementov (i, j) množice $\{(i,\, j)|\ i + j = k,\ 0 \leq i \leq n,\ 0 \leq j \leq t\}$. Mi smo to implementirali na naslednji način. Pri danem številu k izračunamo i po naslednji formuli

$$\textbf{if} \quad (k > t) \quad \textbf{then} \quad i = 0 \quad \textbf{else} \quad i = k - t - 1,$$

kjer je t število besed števila b. Število j pa določimo iz $k = i + j$. Pri tako določenih i in j izvajamo našo zanko dokler velja $i \leq k$ in $i < n$.

Med algoritmi za množenje smo imeli največ težav pri implementaciji Karatsubinega množenja oz. Algoritma 5. Namreč koraka 2.1.1 in 2.1.2, kljub navidezni enostavnosti, težko zelo učinkovito implementiramo. Pri izračunu števila r_1 lahko pride do prekoračitve obsega, saj sta lahko člena $(a_{i+i} + a_i)$ in $(b_{j+1} + b_j)$ večja od 2^{32} in posledično je lahko njun produkt večji od 2^{64}. Prav tako se pojavijo težave z izračunom $(u_3u_2u_1u_0)$, saj, kot smo že omenili, imajo PII procesorji majhno število registrov.

Kvadriranje

Za kvadriranje smo navedli Algoritma 6 in 7, ki sta imela za osnovo Algoritma 3 in 4. Pri implementaciji obeh algoritmov se soočamo z istimi težavami kot pri Algoritmih 3 in 4.

Pojavi se še težava z implementiranjem premika 64-bitnega števila (dveh 32-bitnih registrov) v levo. Vendar imajo procesorji Pentium vgrajen ukaz **SHLD** (Double Precision Shift Left), ki izračuna iskani premik.

Deljenje
Kot smo omenili v drugem poglavju, je deljenje časovno najbolj zahtevna osnovna aritmetična operacija. V samem algoritmu ni veliko možnosti za pohitritve, saj uporabljamo že hitre algoritme za osnovne operacije kot so seštevanje, odštevanje in množenje. Ostane nam le še korak 3.1, kjer izvedemo deljenje dveh števil. In ravno ta korak je računsko najzahtevnejši oz. najpočasnejši. Algoritem 8 je najpočasnejši, ko v koraku 3.1 delimo z majhnim številom. To posledično slabo vpliva na Lehmerjev algoritem, kjer se izvede največ deljenj, ko delimo z majhnim številom, saj sta takrat kvocienta q in q' največkrat različna.

Modularna redukcija
Hitra modularna redukcija je enostavna za implementiranje. Težava je v sestavljanju števil S_i, kjer imamo veliko klicev spremenljivke a. Vendar so kljub vsemu Algoritmi 9-13 hitrejši od ostalih algoritmov za modularno redukcijo. Za primerjavo sem implementiral še Barretovo redukcijo.

Do sedaj omenjene algoritme enostavno prevedemo v praštevilski obseg. Saj v bistvu le izvedemo algoritem in poskrbimo, da je rezultat v obsegu $\mathbb{Z}_p$. V primeru seštevanja in odštevanja je dovolj, da odštejemo oz. prištejemo praštevilo p, če rezultat ni v obsegu $\mathbb{Z}_p$. V primeru množenja ali kvadriranja pa izvedemo modularno redukcijo.

Algoritmi za inverz so sestavljeni iz osnovnih aritmetičnimih operacij in jih lahko dodatno pohitrimo le z izboljšavo algoritmov za osnovne aritmetičnime operacije oz. z izboljšavo samih algoritmov.

4.3 Rezultati

Testirali smo vse opisane algoritme in še enega dodatnega (Barretova redukcija). Algoritme smo velikokrat ponovili in nato izračunali povprečje vseh časov njihovega izvajanja. Najprej navedimo razultate testiranj za osnovne algoritme za aritmetiko naravnih števil (Tabela 4.1). Po stolpcih imamo rezultate glede na število besed, ki se ujemajo z dolžinami praštevil p_{192}, p_{224}, p_{256}, p_{384} in p_{521}. Tako bomo lahko kasneje rezultate primerjali z algoritmi za aritmetiko v $\mathbb{Z}_p$.

Opazimo, da sta algoritma za seštevanje in odštevanje približno enako hitra. Množenje 1 je počasnejše od Množenja 2. Razlika je še posebej očitna pri višjih obsegih. Algoritem 5 tj. Karatsubin algoritem je primerljiv z obema ostalima algoritmima za množenje. Ven-

Št. besed	6	7	8	12	17
Seštevanje (Alg. 1)	1.61833	1.71848	1.79534	2.21117	2.84137
Odštevanje (Alg. 2)	1.30107	1.42031	1.65492	1.98478	2.53193
Množenje 1 (Alg. 3)	6.54837	8.13323	9.91821	19.1686	34.8341
Množenje 2 (Alg. 4)	5.93773	7.17219	8.75706	15.2988	26.6252
Karatsubino mn. (Alg. 5)	5.92454	8.99896	8.89707	17.9789	36.9994
Kvadriranje 1 (Alg. 6)	4.67615	5.57927	6.83426	12.3329	21.5981
Kvadriranje 2 (Alg. 7)	5.36009	6.51171	7.82608	14.0226	24.1739

Tabela 4.1: Časi izvajanj (v μs) (brez redukcije).

dar tudi on pri višjih obsegih zaostaja za Množenjem 2. Algoritem 4, tj. Množenje 2, je najhitrejši izmed izbranih algoritmov za množenje. Opazimo, da je Kvadriranje 2, ki ima za osnovo Algoritem 4, počasnejši od Kvadriranja 1, ki ima za osnovo Algoritem 3. Opazimo tudi, da sta sicer hitrejša od množenj, vendar ne za polovico. To lahko razložimo, z dodatno kompleksnostjo algoritmov v primerjavi z algoritmi za množenje.

Obseg	$\mathbb{Z}_{p_{192}}$	$\mathbb{Z}_{p_{224}}$	$\mathbb{Z}_{p_{256}}$	$\mathbb{Z}_{p_{384}}$	$\mathbb{Z}_{p_{521}}$
Mod. seštevanje	2.32147	2.53718	2.63873	3.37707	4.72891
Mod. Odštevanje	2.17022	2.23673	2.41762	3.19255	4.59042
Mod. množenje 1	11.2018	17.8161	25.5902	33.4236	46.4397
Mod. množenje 2	10.5633	16.8404	24.1446	29.5179	37.9986
Mod. karatsubino mn.	10.7181	18.5581	24.6079	32.5351	48.4878
Mod. kvadriranje 1	9.47032	15.4234	22.6323	27.1692	32.6451
Mod. kvadriranje 2	10.1308	16.3779	22.7349	27.4697	34.7445
Hitra redukcija	3.85276	8.52807	14.017	12.6929	9.90272
Barretova redukcija	33.4867	36.6032	40.1169	56.4512	79.4204
Evklidov alg.	2129.58	2664.66	3238.09	5642.49	8256.27
Binarni alg.	1146.85	1425.11	1754.63	3169.02	4577.42
Lehmerjev alg.	735.368	997.637	1210.46	2064.15	2863.32
Leh. alg. (Collins)	720.828	984.732	1197.74	2060.52	2880.95
Leh. alg. (Jebelean)	706.106	968.375	1177.57	2025.80	2806.79

Tabela 4.2: Časi izvajanj (v μs) v NIST obsegih.

V Tabeli 4.2 so rezultati testiranj algoritmov za praštevilske obsege. Modularno seštevanje in odštevanje sta dražja od navadnega seštevanja oz. odštevanja. Dražja sta ravno za dodatno odštevanje, ki ga izvedemo (v polovici primerov) zato, da je dobljeni rezultat v obsegu $\mathbb{Z}_p$. Vsa modularna množenja in kvadriranja se od osnovnih algoritmov razlikujejo le po tem, da na koncu izvedemo še (hitro) redukcijo. Iz Tabel 4.1 in 4.2 vidimo, da se rezultati osnovnih in modularnih množenj in kvadriranj, razlikujejo ravno za čas izvajanja ene hitre redukcije. Hitra redukcija je občutno hitrejša od Barretove

redukcije in res upraviči svoje ime. Opazimo, da časi izvajanj hitre redukcije ne naraščajo nujno s številom besed. Namreč hitra redukcija je odvisna od praštevila p. Tako vidimo, da imamo v primeru p_{521} oz. v Algoritmu 13 eno seštevanje dveh 17 besednih števil. V primeru p_{256} oz. v Algoritmu 11 pa imamo 4 seštevanja, 4 odštevanja in 2 premika v levo 8 besednih števil, kar je dražje od samo enega seštevanja dveh 17 besednih števil.

Inverz

V Tabeli 4.2 opazimo, da je Evklidov algoritem najpočasnejši izmed algoritmov za inverz. To je posledica počasnosti deljenja za velika števila. Hitro izračunamo, da je približno 1.8 krat počasnejši od Binarnega algoritma in približno 2.8 krat od Lehmerjevih algoritmov. Če uporabimo Knuthovo oceno za povprečno število korakov v Evklidovem algoritmu,

$$\frac{12\ln 2}{\pi^2}\ln N + 0.14 \approx \frac{12\ln 2}{\pi^2}\ln 2^n = \frac{12\ln^2 2}{\pi^2}n,$$

za naše obsege, ugotovimo, da se ocena ujema z številom korakov za vse te obsege (Tabela 4.3). Opazimo tudi, da je kvocient q_i skoraj vedno manjši od računalniške besed W, tako imamo v koraku 2.3 Razširjenega Evklidovega algoritma, skoraj vedno množenje velikega števila z majhnim številom. Seštevanja v obsegu $\mathbb{Z}_{p_{521}}$ so posledica modularnega seštevanja, ki nastopa v modularni redukciji.

Obseg	$\mathbb{Z}_{p_{192}}$	$\mathbb{Z}_{p_{224}}$	$\mathbb{Z}_{p_{256}}$	$\mathbb{Z}_{p_{384}}$	$\mathbb{Z}_{p_{521}}$
Št. seštevanj	0	0	0	0	89.378
Št. odštevanj	169.412	234.572	268.563	403.179	457.969
Št. množenj	0	0.001	0	0	0
Št. mn. z majh. št.	112.776	130.803	149.601	224.802	305.156
Št. deljenj	93.586	111.654	130.258	205.645	285.982
Št. korakov	112.686	130.789	149.547	224.897	305.147

Tabela 4.3: Število večbesednih operacij pri Evklidovem alg.

Binarni algoritem je učinkovit za velika števila, ker ne vsebuje deljenja, ampak le seštevanja, odštevanja in premike v desno. Število korakov Binarnega algoritma se ujema s Knuthovo oceno

$$št.korakov = 0.70N + O(1),$$

kjer za N velja $a, b \leq 2^N$.

Lehmerjev algoritem je v vseh svojih različicah najhitrejši algoritem za izračun inverza. V povprečju je za tretjino hitrejši od Binarnega algoritma. V Tabelah 4.5-7 nam število Evklidovih korakov, pomeni število korakov, ki jih izvedemo, ko je v Lehmerjevem algoritmu $A_1 < W$. Število Lehmerjevih korakov nam pove, v koliko korakih se pri izvajanju aritmetike z majhnimi števili q_i ujema z Q_{i+k}. Hiter izračun nam pove, da imamo v povprečju 7.5 'uspešnih' Lehmerjevih korakov. Ker je Lehmerjev algoritem v osnovi enak

Obseg	$\mathbb{Z}_{p_{192}}$	$\mathbb{Z}_{p_{224}}$	$\mathbb{Z}_{p_{256}}$	$\mathbb{Z}_{p_{384}}$	$\mathbb{Z}_{p_{521}}$
Št. seštevanj	136.565	157.250	181.835	272.363	369.037
Št. odštevanj	339.509	396.549	452.694	678.106	919.834
Št. premikov v desno	542.946	633.246	723.640	1085.466	1471.996
Št. korakov	135.916	158.284	180.910	271.230	367.411

Tabela 4.4: Število večbesednih operacij pri Binarnem alg.

Evklidovem algoritmu, se mora število deljenj oz. korakov v obeh algoritmih ujemati. Če v Lehmerjevem algoritmu ali v njegovih inačicah, seštejemo število deljenj, Lehmerjevih in Evklidovih korakov, dobimo število, ki se dokaj natančno ujema s številom deljenj v Evklidovem algoritmu. Iz Tabel 4.5-7 vidimo, da ni nekih posebnih razlik med vsemi različicami Lehmerjevega algoritma. Tako, da je glede na Tabelo 4.2, Lehmerjev algoritem z Jebeleanovim pogojem najboljša izbira za računanje inverza.

Obseg	$\mathbb{Z}_{p_{192}}$	$\mathbb{Z}_{p_{224}}$	$\mathbb{Z}_{p_{256}}$	$\mathbb{Z}_{p_{384}}$	$\mathbb{Z}_{p_{521}}$
Št. seštevanj	0	0	0	0	82.349
Št. odštevanj	86.500	136.604	153.016	220.711	210.235
Št. množenj	0.002	0	0	0	0.001
Št. mn. z majh. št.	114.653	133.876	152.986	230.288	312.464
Št. deljenj	1.006	1.013	1.009	1.016	1.022
Št. Lehm. korakov	95.594	114.192	133.555	207.818	288.487
Št. Evk. korakov	15.740	16.155	16.136	16.039	16.238
Št. korakov	13.215	15.609	18.023	27.670	37.937

Tabela 4.5: Število večbesednih operacij pri Lehmerjevem alg.

Obseg	$\mathbb{Z}_{p_{192}}$	$\mathbb{Z}_{p_{224}}$	$\mathbb{Z}_{p_{256}}$	$\mathbb{Z}_{p_{384}}$	$\mathbb{Z}_{p_{521}}$
Št. seštevanj	0	0	0	0	82.487
Št. odštevanj	89.866	138.955	156.274	225.385	216.855
Št. množenj	0	0	0.003	0.002	0
Št. mn. z majh. št.	116.993	115.138	156.531	235.670	320.239
Št. deljenj	0.024	0.011	0.014	0.027	0.023
Št. Lehm. korakov	96.358	115.138	133.970	208.510	288.258
Št. Evk. korakov	16.339	16.219	15.914	15.808	16.112
Št. korakov	12.609	15.106	17.562	27.446	38.017

Tabela 4.6: Število večbesednih operacij pri Lehmerjevem alg. (Collins).

Obseg	$\mathbb{Z}_{p_{192}}$	$\mathbb{Z}_{p_{224}}$	$\mathbb{Z}_{p_{256}}$	$\mathbb{Z}_{p_{384}}$	$\mathbb{Z}_{p_{521}}$
Št. seštevanj	0	0	0	0	80.606
Št. odštevanj	88.337	135.754	156.274	220.192	211.609
Št. množenj	0	0.001	0.003	0	0
Št. mn. z majh. št.	114.623	133.398	156.531	229.931	312.264
Št. deljenj	0.012	0.010	0.014	0.014	0.019
Št. Lehm. korakov	96.428	115.235	133.970	208.214	288.721
Št. Evk. korakov	16.076	16.023	15.996	16.099	16.194
Št. korakov	12.308	14.726	17.562	26.768	37.052

Tabela 4.7: Število večbesednih operacij pri Binarnem alg. (Jebelean).

Druga okolja

Ena izmed glavnih motivacij za to diplomo je članek Software Implementation of the NIST Elliptic Curves Over Prime Fields [2], v katerem so preučevali učinkovito implementacijo NIST eliptičnih krivulj nad praštevilskimi obsegi. Algoritmi, katere sem navedel, se ujemajo z algoritmi, ki so bili izbrani v [2]. V tem članku so algoritme implementirali na 400 MHz Pentiumu. Tudi oni so prišli do ugotovitve, da se z uporabo zbirnika hitrost algoritmov poveča za faktor 2 in več. Njihovi časi so približno 10-krat hitrejši od mojih. Vzroki za to razliko so hitrejši računalnik, večja uporaba zbirnika ter bolje optimizirana koda v zbirniku, ki do potankosti izkoristi zmožnosti procesorja. Opazimo, da se razmerja med seštevanjem in množenjem, množenjem in inverzom ter med hitro in Barretovo redukcijo kar ujemajo z našimi rezultati. Zaradi velike razlike med množenjem in inverzom, so avtorji članka [2] predlagali drugačne (projektivne) koordinate za predstavitev točk na eliptičnih krivuljah. Operacije na eliptičnih krivuljah v teh koordinatah ne vsebujejo več deljenja, ampak le ta nadomestimo z nekaj dodatnimi množenji.

Zaradi primerjave sem testiranja izvedel še na treh računalnikih (667 MHz Pentiumu 3, 900 MHz Athlon Thunderbirdu in 1700 MHz Athlon XP 2100+). Rezultati teh testiranj se nahajajo v Tabelah 4.8-10. Opazimo, da so časi približno tolikokrat hitrejši, kolikokrat so dani procesorji hitrejši od 300 MHz Pentiuma. Razmerja med operacijami se ohranjajo. Izjema je le Algoritem 7 oz. Kvadriranje 2, ki je na obeh Athlon procesorjih hitrejši od Kvadriranja 1. To je posledica arhitekture Athlon procesorjev, ki se razlikuje od arhitekture Pentium procesorjev.

4.4 Zaključek

Po vseh testiranjih lahko končno izberemo najboljše kandidate za posamezne operacije. Za seštevanje in odštevanje smo navedli samo en algoritem tako, da tu nimamo kaj izbrati. Pri množenju se je najbolje iskazal Algoritem 4 oz. Množenje 2. Pri kvadriranju pa Algo-

Obseg	$\mathbb{Z}_{p_{192}}$	$\mathbb{Z}_{p_{224}}$	$\mathbb{Z}_{p_{256}}$	$\mathbb{Z}_{p_{384}}$	$\mathbb{Z}_{p_{521}}$
Seštevanje	0.72500	0.75000	0.83500	1.04333	1.29000
Odštevanje	0.65000	0.70500	0.79666	0.95333	1.22666
Množenje 1	3.02500	3.90000	4.70000	9.14166	16.9333
Množenje 2	2.74166	3.49166	4.12500	7.35833	12.9083
Karatsubino mn.	2.73333	4.25000	4.34166	8.55000	17.8833
Kvadriranje 1	2.15833	2.74166	3.28333	5.62500	10.1500
Kvadriranje 2	2.63333	3.02500	3.65000	6.67500	11.5500
Mod. seštevanje	1.10500	1.21500	1.30000	1.52666	2.29000
Mod. odštevanje	1.02000	1.06500	1.14333	1.47500	1.75333
Mod. množenje 1	5.40000	8.38333	11.7083	15.7083	22.2500
Mod. množenje 2	5.00833	7.97500	10.9833	13.8500	18.1416
Mod. karatsubino mn.	5.05833	8.89166	11.2583	15.2000	23.2666
Mod. kvadriranje 1	4.42500	7.19166	10.6583	12.3583	15.4416
Mod. kvadriranje 2	4.81666	7.55000	10.8416	13.4000	16.9916
Hitra redukcija	1.69000	3.95666	6.37666	5.84333	4.50333
Barretova redukcija	15.6333	17.2766	18.8966	22.5953	37.4433
Evklidov alg.	978.333	1265.00	1536.66	2663.33	3983.33
Binarni alg.	541.666	658.333	808.333	1491.66	2165.00
Lehmerjev alg.	356.666	468.333	595.000	973.333	1403.33
Leh. alg. (Collins)	348.333	468.333	570.000	968.333	1390.00
Leh. alg. (Jebelean)	328.333	456.666	576.666	935.000	1370.00

Tabela 4.8: Časi izvajanj (v μs) na Pentiumu 3 (667 MHz) .

ritem 6 oz. Kvadriranje 1. Hitra modularna redukcija je kar občutno hitrejša od Barretove redukcije. Za izračun inverza v praštevilskih obsegih pa izberemo Razširjeni Lehmerjev algoritem z Jebeleanovim pogojem.

Dodatne izboljšave oz. boljši časi bi bili doseženi, če bi vsak obseg posebej sprogramirali. Vendar bi razmerja med algoritmi ostala okvirno enaka. Prav tako bi lahko z več časa preizkusili druga okolja in druge prevajalnike ter preverili, kako se naša implementacija obnese v teh okoljih.

Obseg	$\mathbb{Z}_{p_{192}}$	$\mathbb{Z}_{p_{224}}$	$\mathbb{Z}_{p_{256}}$	$\mathbb{Z}_{p_{384}}$	$\mathbb{Z}_{p_{521}}$
Seštevanje	0.43400	0.47400	0.50566	0.62250	0.79933
Odštevanje	0.44083	0.47066	0.50750	0.62250	0.78100
Množenje 1	2.00333	2.52666	3.25500	6.44083	12.1675
Množenje 2	1.67750	2.09500	2.46916	4.61750	8.06000
Karatsubino mn.	1.95416	3.03750	3.10500	6.26583	13.1200
Kvadriranje 1	1.58666	1.95333	2.37166	4.59083	8.47250
Kvadriranje 2	1.38583	1.72166	2.00333	3.60666	6.10166
Mod. seštevanje	0.68783	0.74450	0.79283	0.97633	1.48700
Mod. odštevanje	0.62983	0.67416	0.71966	0.88450	1.18150
Mod. množenje 1	3.44666	5.35000	7.82083	10.5566	15.3308
Mod. množenje 2	3.05333	4.82416	6.97750	8.59666	11.3333
Mod. karatsubino mn.	3.41250	5.87500	7.65250	10.3233	16.4575
Mod. kvadriranje 1	3.01250	4.72250	6.99416	8.68000	11.6583
Mod. kvadriranje 2	2.80416	4.51666	6.63416	7.67916	9.49750
Hitra redukcija	1.14866	2.39666	4.13566	3.59900	2.76733
Barretova redukcija	9.11666	10.2116	11.3130	16.5140	23.8573
Evklidov alg.	562.500	732.666	877.833	1535.50	2316.83
Binarni alg.	342.166	430.666	529.166	971.500	1349.83
Lehmerjev alg.	206.833	282.166	328.833	560.833	804.666
Leh. alg. (Collins)	197.000	280.333	323.833	559.166	811.166
Leh. alg. (Jebelean)	190.333	273.833	317.166	545.666	787.833

Tabela 4.9: Časi izvajanj (v μs) na Athlon Thunderbirdu (900 MHz).

Obseg	$\mathbb{Z}_{p_{192}}$	$\mathbb{Z}_{p_{224}}$	$\mathbb{Z}_{p_{256}}$	$\mathbb{Z}_{p_{384}}$	$\mathbb{Z}_{p_{521}}$
Seštevanje	0.21400	0.22433	0.25600	0.33333	0.46366
Odštevanje	0.20433	0.21866	0.26000	0.30166	0.39600
Množenje 1	1.01333	1.35500	1.66666	3.33166	6.32833
Množenje 2	0.88166	1.12000	1.30500	2.42666	4.16666
Karatsubino mn.	0.96333	1.58500	1.58833	3.20500	6.92833
Kvadriranje 1	0.80833	1.01500	1.19500	2.44500	4.40500
Kvadriranje 2	0.70500	0.88333	1.04500	1.90333	3.20333
Mod. seštevanje	0.35933	0.39600	0.41200	0.48933	0.75433
Mod. odštevanje	0.29166	0.32166	0.38033	0.44866	0.54133
Mod. množenje 1	1.80000	2.78666	4.03833	5.41666	8.07333
Mod. množenje 2	1.59000	2.50000	3.67500	4.37500	5.96166
Mod. karatsubino mn.	1.74500	3.15000	4.06000	5.41833	8.56833
Mod. kvadriranje 1	1.48333	2.50000	3.72500	4.68500	6.30166
Mod. kvadriranje 2	1.50833	2.37166	3.49166	4.13833	5.05333
Hitra redukcija	0.57333	1.19800	2.16600	1.91600	1.47800
Barretova redukcija	4.80200	5.38600	5.94933	8.61333	12.3640
Evklidov alg.	297.000	390.666	460.833	810.000	1216.16
Binarni alg.	190.000	224.000	273.666	508.166	710.500
Lehmerjev alg.	107.000	145.833	166.500	297.000	427.000
Leh. alg. (Collins)	101.500	151.166	166.666	289.000	427.333
Leh. alg. (Jebelean)	98.8333	153.500	164.500	283.666	414.500

Tabela 4.10: Časi izvajanj (v μs) na Athlon XP 2100+ (1700 MHz).

Literatura

[1] A. Bosselaers, R. Govaerts, J. Vandewalle, *Comparison of three modular reduction functions*, Advances in Cryptology-Crypto 93, **LNCS 773** (1994), 175–186.

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

[3] G.E. Collins, *The computing time of the Euclidean algorithm*, SIAM Journal on Computing, **3** (1974), 1–10.

[4] IEEE, *P1363 Standard Specifications for Public-Key Cryptography*, http://grouper.ieee.org/groups/1363/index.html.

[5] T. Jebelean, *Improving the Multiprecision Euclidean Algorithm*, RISC-Linz Report 92-69.

[6] T. Jebelean, *A generalization of the Binary GCD algorithm*, ISSAC'93, Kiew, 1993.

[7] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms, Volume 2*, Addison-Wesley, Reading, Mass., 2nd edition, 1981.

[8] J. Kozak, *Podatkovne strukture in algoritmi*, Društvo matematikov, fizikov in astronomov SR Slovenije, Ljubljana, 1986.

[9] D.H. Lehmer, *Euclid's algorithm for large numbers*, American Mathematics Monthly, **45** (1938), 227–233.

[10] A. J. Menezes, P. C. Van Oorschot, S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press LLC, 1997.

[11] National Institute of Standards and Technology, *Digitial Signature Standard*, FIPS Publications 186-2, Februrary 2000.

[12] R. Petek, *RSA kriptosistem*, Diplomsko delo, 2000.

[13] J.A. Solinas, *Generalized Mersenne numbers*, Technical Report CORR 99-39, Dept. of C&O, University of Waterloo, 1999.

[14] J. Sorenson, *An Analysis of Lehmer's Euclidean GCD Algorithm*, In *Proc. AMC ISSAC'95 Symp.*, 1995, pages 254–258.

[15] D. R. Stinson, *Cryptography: Theory and Practice*, CRC Press, 1995.

[16] I. Vidav, *Algebra*, Društvo matematikov, fizikov in astronomov SR Slovenije, Ljubljana, 1989.